Synthesis: An Efficient Implementation of Fundamental Operating System Services

Henry Massalin

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences.

Columbia University
1992

Synthesis: An Efficient Implementation of Fundamental Operating System Services

Henry Massalin

Abstract

This dissertation shows that operating systems can provide fundamental services an order of magnitude more efficiently than traditional implementations. It describes the implementation of a new operating system kernel, Synthesis, that achieves this level of performance.

The Synthesis kernel combines several new techniques to provide high performance without sacrificing the expressive power or security of the system. The new ideas include:

- *Run-time code synthesis* — a systematic way of creating executable machine code at runtime to optimize frequently-used kernel routines — queues, buffers, context switchers, interrupt handlers, and system call dispatchers — for specific situations, greatly reducing their execution time.

- *Fine-grain scheduling* — a new process-scheduling technique based on the idea of feedback that performs frequent scheduling actions and policy adjustments (at sub-millisecond intervals) resulting in an adaptive, self-tuning system that can support real-time data streams.

- *Lock-free optimistic synchronization* is shown to be a practical, efficient alternative to lock-based synchronization methods for the implementation of multiprocessor operating system kernels.

- An extensible kernel design that provides for simple expansion to support new kernel services and hardware devices while allowing a tight coupling between the kernel and the applications, blurring the distinction between user and kernel services.

The result is a significant performance improvement over traditional operating system implementations in addition to providing new services.

Contents

List of Figures

List of Tables

Acknowledgements

Many people contributed to making this research effort a success.

First and foremost, I want to thank my advisor, Calton Pu. He was instrumental in bringing this thesis to fruition. He helped clarify the ideas buried in my "collection of fast assembly-language routines," and his dedication through difficult times encouraged me to keep pushing forward. Without him, this dissertation would not exist.

I am also greatly indebted to the other members of my committee: Dan Duchamp, Bob Sproull, Sal Stolfo, and John Zahorjan. Their valuable insight and timely suggestions helped speed this dissertation to completion.

My sincerest appreciation and deepest 'Qua!'s go to Renate Valencia. Her unselfish love and affection and incredible amount of emotional support helped me through some of my darkest hours here at Columbia and gave me the courage to continue on. Thanks also to Matthew, her son, for letting me borrow Goofymeyer, his stuffed dog.

Many other friends in many places have helped in many ways; I am grateful to Emilie Dao for her generous help and support trying to help me understand myself and for the fun times we had together; to John Underkoffler and Clea Waite for their ear in times of personal uncertainty; to Mike Hawley and Olin Shivers, for their interesting conversation, rich ideas, and untiring willingness to "look at a few more sentences"; to Ken Phillips, for the thoughts we shared over countless cups of coffee; to Mort Meyerson, whose generosity in those final days helped to dissipate some of the pressure; to Brewster Kahle, who always has a ready ear and a warm hug to offer; to Domenic Frontiere and family, who are some of the most hospitable people I know; and to all my friends at Cooper Union, who made my undergrad and teaching years there so enjoyable.

I also wish to thank Ming-Chao Chiang, Tom Matthews, and Tim Jones, the project students who worked so hard on parts of the Synthesis system. Thanks also go to all the people in the administrative offices, particularly Germaine, who made sure all the paperwork flowed smoothly between the various offices and who helped schedule my thesis defense on record short notice. I particularly want to thank my friends here at Columbia — Cliff Beshers, Shu-Wie Chen, Ashutosh Dutta, Edward Hee, John Ioannidis, Paul Kanevsky, Fred Korz, David Kurlander, Jong Lim, James Tanis, and George Wolberg, to name just a few. The countless dinners, good times, and piggy-back rides we shared helped make my stay here that much more enjoyable.

I also wish to extend special thanks to the people at the University of Washington, especially Ed Lazowska, Hank Levy, Ed Felten, David Keppel (*a.k.a.* Pardo), Dylan McNamee, and Raj Vaswani, whose boundless energy and happiness always gave me something to look forward to when visiting Seattle or traveling to conferences and workshops. Special thanks to Raj, Dylan, Ed Felten and Jan and Denny Prichard, for creating that 'carry' tee shirt and making me feel special; and to Lauren Bricker, Denise Draper, and John Zahorjan for piggy-back rides of unparalleled quality and length.

Thanks goes to Sony corporation for the use of their machine; to Motorola for supplying most of the parts used to build my computer, the Quamachine; and to Burr Brown for their generous donation of digital audio chips.

And finally, I want to thank my family, whose patience endured solidly to the end. Thanks to my mother and father, who always welcomed me home even when I was too busy to talk to them. Thanks, too, to my sister Lucy, sometimes the only person with whom I could share my feelings, and to my brother, Peter, who is always challenging me to a bicycle ride.

In appreciation, I offer to all a warm, heartfelt

— Qua! —

1 Introduction

I must Create a System, or be enslav'd by another Man's;
I will not Reason and Compare: my business is to Create.
— William Blake *Jerusalem*

1.1 Purpose

This dissertation shows that operating systems can provide fundamental services an order of magnitude more efficiently than traditional implementations. It describes the implementation of a new operating system kernel, Synthesis, that achieves this level of performance.

The Synthesis kernel combines several new techniques to provide high performance without sacrificing the expressive power or security of the system. The new ideas include:

- *Run-time code synthesis* — a systematic way of creating executable machine code at runtime to optimize frequently-used kernel routines — queues, buffers, context switchers, interrupt handlers, and system call dispatchers — for specific situations, greatly reducing their execution time.

- *Fine-grain scheduling* — a new process-scheduling technique based on the idea of feedback that performs frequent scheduling actions and policy adjustments (at sub-millisecond intervals) resulting in an adaptive, self-tuning system that can support real-time data streams.
- *Lock-free optimistic synchronization* is shown to be a practical, efficient alternative to lock-based synchronization methods for the implementation of multiprocessor operating system kernels.
- An extensible kernel design that provides for simple expansion to support new kernel services and hardware devices while allowing a tight coupling between the kernel and the applications, blurring the distinction between user and kernel services.

The result is a significant performance improvement over traditional operating system implementations in addition to providing new services.

The text is structured as follows: The remainder of this chapter summarizes the project. It begins with a brief history, showing how my dissatisfaction with the performance of computer software led me to do this research. It ends with an overview of the Synthesis kernel and the hardware it runs on. The intent is to establish context for the remaining chapters and reduce the need for forward references in the text.

Chapter 2 examines the design decisions and tradeoffs in existing operating systems. It puts forth arguments telling why I believe some of these decisions and tradeoffs should be reconsidered, and points out how Synthesis addresses the issues.

The next four chapters present the new implementation techniques. Chapter 3 explains run-time kernel code synthesis. Chapter 4 describes the structure of the Synthesis kernel. Chapter 5 explains the lock-free data structures and algorithms used in Synthesis. Chapter 6 talks about fine-grain scheduling. Each chapter includes measurements that prove the effectiveness of each idea.

Application-level measurements of the system as a whole and comparisons with other systems are found in chapter 7. The dissertation closes with chapter 8, which contains conclusions and directions for further work.

1.2 History and Motivation

This section gives a brief history of the Synthesis project. By giving the reader a glimpse of what was going through my mind while doing this research, I establish context and make the new ideas easier to grasp by showing the motivation behind them.

. . .

In 1983, the first UNIX-based workstations were being introduced. I was unhappy with the performance of computers of that day, particularly that of workstations relative to what DOS-based PCs could deliver. Among other things, I found it hard to believe that the workstations could not drive even one serial line at a full 19,200 baud — approximately 2000 characters per second.[1] I remember asking myself and others: "There is a full half-millisecond time between characters. What could the operating system possibly be doing for that long?" No one had a clear answer. Even at the relatively slow machine speed of that day — approximately one million machine instructions per second — the processor could execute 500 machine instructions in the time a character was transmitted. I could not understand why 500 instructions were not sufficient to read a character from a queue and have it available to write to the device's control register by the time the previous one had been transmitted.

That summer, I decided to try building a small computer system and writing some operating systems software. I thought it would be fun, and I wanted to see how far I could get. I teamed up with a fellow student, James Arleth, and together we built the precursor of what was later to become an experimental machine known as the *Quamachine*. It was a two-processor machine based on the 68000 CPU [4], but designed in such a way that it could be split into two independently-operating halves, so we each would have a computer to take with us after we graduated. Jim did most of the hardware design while I concentrated on software.

The first version of the software [19] consisted of drivers for the machine's serial ports and 8-bit analog I/O ports, a simple multi-tasker, and an unusual debug monitor that included a rudimentary C-language compiler/interpreter as its front end. It was quite small

[1] This is still true today despite an order-of-magnitude speed increase in the processor hardware, and attests to a comparable increase in operating system overhead. Specifically, the Sony NEWS 1860 workstation, running release 4.0 of Sony's version of UNIX, places a software limit of 9600 baud on the machine's serial lines. If I force the line to go faster through the use of kernel hackery, the operating system loses data each time a burst longer than about 200 characters arrives at high speed.

— everything fit into the machine's 16 kilobyte ROM, and ran comfortably in its 16 kilobyte RAM. And it did drive the serial ports at 19,200 baud. Not just one, but all four of them, concurrently. Even though it lacked many fundamental services, such as a filesystem, and could not be considered a "real" operating system in the sense of UNIX, it was the precursor of the Synthesis kernel, though I did not know it at the time.

After entering the PhD program at Columbia in the fall of 1984, I continued to develop the system in my spare time, improving both the hardware and the software, and also experimenting with other interests — electronic music and signal processing. During this time, the CPU was upgraded several times as Motorola released new processors in the 68000 family. Currently, the Quamachine uses a 68030 processor rated for 33 MHz, but running at 50MHz, thanks to a homebrew clock circuit, special memory decoding tricks, a higher-than-spec operating voltage, and an ice-cube to cool the processor.

But as the software was fleshed out with more features and new services, it became slower. Each new service required new code and data structures that often interacted with other, unrelated, services, slowing them down. I saw my system slowly acquiring the ills of UNIX, going down the same road to inefficiency. This gave me insight into the inefficiency of UNIX. I noticed that, often, the mere presence of a feature or capability incurs some cost, even when not being used. For example, as the number of services and options multiply, extra code is required to select from among them, and to check for possible interference between them. This code does no useful work processing the application's data, yet it adds overhead to each and every call.

Suddenly, I had a glimmer of an idea of how to prevent this inefficiency from creeping into my system: runtime code generation! All along I had been using a monitor program with a C-language front end as my "shell." I could install and remove services as needed, so that no service would impose its overhead until it was used. I thought that perhaps there might be a way to automate the process, so that the correct code would be created and installed each time a service was used, and automatically removed when it was no longer needed. This is how the concept of creating code at runtime came to be. I hoped that this could provide relief from the inefficiencies that plague other full-featured operating systems.

I was dabbling with these ideas, still in my spare time, when Calton Pu joined the faculty at Columbia as I was entering my third year. I went to speak with him since I was still unsure of my research plans and looking for a new advisor. Calton brought with him some interesting research problems, among them the efficient implementation of object-based systems. He had labored through his dissertation and knew where the problems were. Looking at my system, he thought that my ideas might solve that problem one day, and encouraged me to forge ahead.

The project took shape toward the end of that semester. Calton had gone home for Christmas, and came back with the name Synthesis, chosen for the main idea: run-time kernel code synthesis. He helped package the ideas into a coherent set of concepts, and we wrote our first paper in February of 1987.

I knew then what the topic of my dissertation would be. I started mapping out the structure of the basic services and slowly restructured the kernel to use code synthesis throughout. Every operation was subject to intense scrutiny. I recall the joy felt the day I discovered how to perform a "putchar" (place character into buffer) operation in four machine instructions rather than the five I had been using (or eight, using the common C-language macro). After all, "putchar" is a common operation, and I found it both satisfying and amusing that eliminating one machine instruction resulted in a 4% overall gain in performance for some of my music applications. I continued experimenting with electronic music, which by then had become more than a hobby, and, as shown in section 6.3, offered a convincing demonstration that Synthesis did deliver the kind of performance claimed.

Over time, this type of semi-playful, semi-serious work toward a fully functional kernel inspired the other features in Synthesis — fine-grained scheduling, lock-free synchronization, and the kernel structure.

Fine-grained scheduling was inspired by work in music and signal-processing. The early kernel's scheduler often needed tweaking in order to get a new music synthesis program to run in real-time. While early Synthesis was fast enough to make real-time signal processing possible by handling interrupts and context switches efficiently, it lacked a guarantee that real-time tasks got sufficient CPU as the machine load increased. I had considered the use of task priorities in scheduling, but decided against them, partly because of the programming effort involved, but mostly because I had observed other systems that used priorities, and they did not seem to fully solve the problem. Instead, I got the idea that the scheduler could deduce how much CPU time to give each stage of processing by measuring the data accumulation at each stage. That is how fine-grained scheduling was born. It seemed easy enough to do, and a few days later I had it working.

The overall structure of the kernel was another idea developed over time. Initially, the kernel was an ad-hoc mass of procedures, some of which created code, some of which didn't. Runtime code generation was not well understood, and I did not know the best way to structure such a system. For each place in the kernel where code-synthesis would be beneficial, I wrote special code to do the job. But even though the kernel was lacking in overall structure, I did not see that as negative. This was a period where freedom to experiment led to valuable insights, and, as I found myself repeating certain things, an overall structure gradually became clear.

Optimistic synchronization was a result of these experiments. I had started writing the kernel using disabled interrupts to implement critical sections, as is usually done in other single-processor operating systems. But the limitations of this method were soon brought out in my real-time signal processing work, which depends on the timely servicing of frequent interrupts, and therefore cannot run in a system that disables interrupts for too long. I therefore looked for alternatives to inter-process synchronization. I observed that in many cases, such as in a single-producer/single-consumer queue, the producer and consumer interact only when the queue is full or empty. During other times, they each work on different parts of the queue, and can do so independently, without synchronization. My interest in this area was further piqued when I read about the "Compare-and-Swap" instructions on the 68030 processor, which allows concurrent data structures to be implemented without using locks.

1.3 Synthesis Overview

1.3.1 Kernel Structure

The Synthesis kernel is designed to support a real, full-featured operating system with functionality on the level of UNIX [28] and Mach [1]. It is built out of many small, independent modules called *quajects*. A quaject is an abstract data type — a collection of code and data with a well-defined interface that performs a specific function. The interface encompasses not just the quaject's entry points, but also all its external invocations, making it possible to dynamically link quajects, thereby building up kernel services. Some examples of quajects include various kinds of queues and buffers, threads, TTY input and output editors, terminal emulators, and text and graphics windows.

All higher-level kernel services are created by instantiating and linking two or more quajects through their interfaces. For example, a UNIX-like TTY device is built using the following quajects: a raw serial device driver, two queues, an input editor, an output format converter, and a system-call dispatcher. The wide choice of quajects and linkages allows Synthesis to support a wide range of different system interfaces at the user level. For example, Synthesis includes a (partial) UNIX emulator that runs some SUN-3 binaries. At the same time, a different application might use a different interface, for example, one that supports asynchronous I/O.

1.3.2 Implementation Ideas

One of the ways Synthesis achieves order-of-magnitude gains in efficiency is through the technique of kernel code synthesis. Kernel code synthesis creates, on-the-fly, specialized (thus short and fast) kernel routines for specific situations, reducing the execution path for frequently used kernel calls. For example, queue quajects have their buffer and pointer addresses hard-coded using self-relative addressing; thread quajects have their system-call dispatch and context-switch code specially crafted to speed these operations. Section 3.3 illustrates the specific code created for these and other examples. This hard-coding eliminates indirection and reduces parameter passing, improving execution speed. Extensive use of the processor's self-relative addressing capability retains the benefits of relocatability and easy sharing. Shared libraries of non-specialized code handle less-frequently occurring cases and keep the memory requirements low. Chapter 3 explains this idea in detail and also introduces the idea of executable data structures, which are highly efficient "self-traversing" structures.

Synthesis handles real-time data streams with fine-grain scheduling. Fine-grain scheduling measures process progress and performs frequent scheduling actions and policy adjustments at sub-millisecond intervals resulting in an adaptive, self-tuning system usable in a real-time environment. This idea is explained in chapter 6, and is illustrated with various music-synthesizer and signal-processing applications, all of which run in real time under Synthesis.

Finally, lock-free optimistic synchronization increases concurrency within the multi-threaded synthesis kernel and enhances Synthesis support for multiprocessors. Synthesis also includes a reentrant, optimistically-synchronized C-language runtime library suitable for use in multi-threaded and multi-processor applications written in C.

1.3.3 Implementation Language

Synthesis is written in 68030 macro assembly language. Despite its obvious flaws — the lack of portability and the difficulty of writing complex programs — I chose assembler because no higher-level language provides both efficient execution and support for runtime code-generation. I also felt that it would be an interesting experiment to write a medium-size system in assembler, which allows unrestricted access to the machine's architecture, and perhaps discover new coding idioms that have not yet been captured in a higher-level language. Section 7.4.1 reports on the experience.

A powerful macro facility helped minimize the difficulty of writing complex programs. It also let me postpone making some difficult system-wide design decisions, and let me easily change them after they were made. For example, quaject definition is a declarative macro in the language. The structure of this macro and the code it produced changed several times during the course of system development. Even the object-file ".o" format is defined entirely by source-code macros, not by the assembler itself, and allows for easy expansion to accommodate new ideas.

1.3.4 Target Hardware

At the time of this writing, Synthesis runs on two machines: the Quamachine and the Sony NEWS 1860 workstation.

The Quamachine is a home-brew, experimental 68030-based computer system designed to aid systems research and measurement. Measurement facilities include an instruction counter, a memory reference counter, hardware program tracing, and an interval timer with 20-nanosecond resolution. As their names imply, the instruction counter keeps a count of machine instructions executed by the processor, and the memory reference counter keeps a count of memory references issued by the processor. The processor can operate at any clock speed from 1 MHz up to 50 MHz. Normally it runs at 50 MHz. But by setting the processor speed to 16 MHz and introducing 1 wait-state into the memory access, the Quamachine closely matches the performance characteristics of a the SUN-3/160, allowing direct measurements and comparisons with that machine and its operating system.

Other features of the Quamachine include 256 kilobytes of no-wait-state ROM that holds the entire Synthesis kernel, monitor, and runtime libraries; $2\frac{1}{2}$ megabytes of no-wait-state main memory; a 2Kx2Kx8-bit framebuffer with graphics co-processor; and audio I/O devices: stereo 16-bit analog output, stereo 16-bit analog input, and a compact disc (CD) player digital interface.

The Sony NEWS 1860 is a workstation with two 68030 processors. It is a commercially available machine, making Synthesis potentially accessible to other interested researchers. It has two processors, which, while not a large number, nevertheless demonstrates Synthesis multiprocessor support. While its architecture is not symmetric — one processor is the main processor and the other is the I/O processor — Synthesis treats it as if it were a symmetric multiprocessor, scheduling tasks on either processor without preference, except those that require something that is accessible from one processor and not the other.

1.3.5 UNIX Emulator

A partial UNIX emulator runs on top of the Synthesis kernel and emulates certain SUNOS kernel calls [31]. Although the emulator supports only a subset of the UNIX system calls — time constraints have forced an "implement-as-the-need-arises" strategy — the set supported is sufficiently rich to provide many benefits. It helps with the problem of acquiring application software for a new operating system by allowing the use of SUN-3 binaries. It further demonstrates the generality of Synthesis by setting the lower bound — emulating a widely used system. And, most important from the research point of view, it allows a direct comparison between Synthesis and UNIX. Section 7.2.1 presents measurements showing that the Synthesis emulation of UNIX is several times more efficient than native UNIX running the same set of programs on comparable hardware.

2

Previous Work

If I have seen farther than others, it is because
I was standing on the shoulders of giants.

— Isaac Newton

If I have not seen as far as others, it is because
giants were standing on my shoulders.

— Hal Abelson

In computer science, we stand on each other's feet.

— Brian K. Reid

2.1 Overview

This chapter sketches an overview of some of the classical goals of operating system design and tells how existing designs have addressed them. This provides a background against which the new techniques in Synthesis can be contrasted. I argue that some of the classical goals need to be reconsidered in light of new requirements and point out the new goals that have steered the design of Synthesis.

There are four areas in which Synthesis makes strong departures from classical designs: overall kernel structure, the pervasive use of run-time code generation, the management of concurrency and synchronization, and novel use of feedback mechanisms in scheduling. The rest of this chapter discusses each of these four topics in turn, but first, it is useful to consider some broad design issues.

2.2 The Tradeoff Between Throughput and Latency

The oldest goal in building operating systems has been to achieve high performance. There are two common measures of performance: throughput and latency. Throughput is a measure of how much useful work is done per unit time. Latency is a measure of how long it takes to finish an individual piece of work. Traditionally, high performance meant increasing the throughput — performing the most work in the minimum time. But traditional ways of increasing throughput also tend to increase latency.

The classic way of increasing throughput is by batching data into large chunks which are then processed together. This way, the high overhead of initiating the processing is amortized over a large quantity of data. But batching increases latency because data that could otherwise be output instead sits in a buffer, waiting while it fills, causing delays. This happens at all levels. The mainframe batch systems of the 1960's made efficient use of machines, achieving high throughput but at the expense of intolerable latency for users and grossly inefficient use of people's time. In the 1970's, the shift toward timesharing operating systems made for a slightly less efficient use of the machine, but personal productivity was enormously improved. However, calls to the operating system were expensive, which meant that data had to be passed in big, buffered chunks in order to amortize the overhead.

This is still true today. For example, the Sony NEWS workstation, running Sony's version of UNIX release 4.0C (a derivative of Berkeley UNIX), takes 260 microseconds to write a single character to an I/O pipe connecting to another program. But writing 1024 characters takes 450 microseconds — a little more than twice the cost of writing a single character. Looking at it another way, over 900 characters can be written in the time taken by the invocation overhead. The reasons for using buffering are obvious. In fact, Sony's program libraries use larger, 8192-character buffers to further amortize the overhead and increase throughput. Such large-scale buffering greatly increases latency and indeed the general trend has been to parcel systems into big pieces that communicate with high overhead, compounding the delays.

In light of these large overheads, it is interesting to examine the history of operating system performance, paying particular attention to the important, low-level operations that

System Function	Time for 1 char (μs)	Time for 1024 chars (μs)
Write to a pipe	260	450
Write to a file	340	420
Read from a pipe	190	610
Read from a file	380	460

System Function	Time (μs)
Dispatch system call (`getpid`)	40
Context Switch	170

Sony NEWS 1860 workstation, 68030 processor, 25MHz, 1 w/s, UNIX Release 4.0C.

Table 2.1: Overhead of Various System Calls, UNIX Release 4.0C

System Function	Time for 1 char (μs)	Time for 1024 chars (μs)
Write to a pipe	470	740
Write to a file	370	600
Read from a pipe	550	760
Read from a file	350	580

System Function	Time (μs)
Dispatch system call (`getpid`)	88
Context Switch	510

NeXT workstation, 68030 processor, 25MHz, 1 w/s, Mach Release 2.1.

Table 2.2: Overhead of Various System Calls, Mach

are exercised often, such as context switch and system call dispatch. We find that operating systems have historically exhibited large invocation overheads. Due to its popularity and wide availability, UNIX is one of the more-studied systems, and I use it here as a baseline for performance comparisons.

In one study, Feder compares the evolution of UNIX system performance over time and over different machines [13]. He studies the AT&T releases of UNIX— System 3 and System 5 — spanning a time period from the mid-70's to late 1982 and shows that UNIX performance had improved roughly 25% during this time. Among the measurements shown is the time taken to execute the `getpid` (get process id) system call. This system call fetches a tiny amount of information (one integer) from the kernel, and its speed is a good indicator of the cost of the system call mechanism. For the VAX-11/750 minicomputer, Feder reports a time of 420 microseconds for `getpid` and 1000 microseconds for context switch.

I have duplicated some of these experiments on the Sony NEWS workstation, a machine of roughly 10 times the performance of the VAX-11/750. Table 2.1 summarizes the results.[1] On this machine, `getpid` takes 40 microseconds, and a context switch takes 170 microseconds. These numbers suggest that, since 1982, the performance of UNIX has remained relatively constant compared to the speed of the hardware.

A study done by Ousterhout [22] shows that operating system speed has not kept pace with hardware speed. The reasons he finds are that memory bandwidth and disk speed have not kept up with the dramatic increases in processor speed. Since operating systems tend to make heavier use of these resources than the typical application, this has a negative effect on operating system performance relative to how the processor's speed is measured.

But I believe there are further reasons for the large overhead in existing systems. As new applications demand more functionality, the tendency has been simply to layer on more functions. This can slow down the whole system because often the mere existence of a feature forces extra processing steps, regardless of whether that feature is being used or not. New features often require extra code or more levels of indirection to select from among them. Kernels become larger and more complicated, leading designers to restructure their operating systems to manage the complexity and improve understandability and maintainability. This restructuring, if not carefully done, can reduce performance by introducing

[1] Even though the Sony machine has two processors, one of them is dedicated exclusively to handling device I/O and does not run any UNIX code. This second processor does not affect the outcome of the tests, which are designed to measure UNIX system overhead, not device I/O capacity. The file read and write benchmarks were to an in-core file system. There was no disk activity.

extra layers and overhead where there was none before.

For example, the Mach operating system offers a wide range of new features, such as threads and flexible virtual memory management, all packaged in a small, modular, easy-to-port kernel [1]. But it does not perform very well compared to Sony's UNIX. Table 2.2 shows the results of the previous experiment, repeated under Mach on the NeXT machine. Both the NeXT machine and the Sony workstation use the Motorola 68030 processor, and both run at 25MHz. All but one of the measurements show reduced performance compared to Sony's UNIX. Crucial low-level functions, such as context switch and system call dispatch, are two to three times slower in this version of Mach.

Another reason for the large overheads might be that invocation overhead *per se* has not been subject to intense scrutiny. Designers tend to optimize the most frequently occurring special cases in the services offered, while the cases most frequently used tend to be those that were historically fast, since those are the ones people would have tended to use more. This self-reinforcing loop has the effect of encouraging optimizations that maintain the status quo with regard to relative performance, while eschewing optimizations that may have less immediate payoff but hold the promise of greater eventual return. Since large invocation overheads can usually be hidden with buffering, there has not been a large impetus to optimize in this direction.

Instead of attacking the problem of high kernel overhead directly, performance problems are being solved with more buffering, applied in ever more ingenious ways to a wider array of services. Look, for example, at recent advances in thread management. A number of researchers begin with the premise that kernel thread operations are necessarily expensive, and go on to describe the implementation of a user-level threads package [17] [5] [33] [3]. Since much of the work is now done at the user-level by subscheduling one or more kernel-supplied threads, they can avoid many kernel invocations and their associated overhead.

But there is a tradeoff: increased performance for operations at the user level come with increased overhead and latency when communicating with the kernel. One reason is that kernel calls no longer happen directly, but first go through the user-level code. Another reason could be that optimizing kernel invocations are no longer deemed to be as important, since they occur less often. For example, while Anderson reports order-of-magnitude performance improvement for user-level thread operations on the DEC CVAX multiprocessor compared to the native Topaz kernel threads implementation, the cost of invoking the kernel thread operations had been increased by a factor of 5 over Topaz threads [3].

The factor of 5 is significant because, ultimately, programs interact with the outside world through kernel invocations. Increasing the overhead limits the rate at which a program can invoke the kernel and therefore, interact with the outside world.

Taken to the limit, the things that remain fast are those local to an application, those that can be done at user-level without invoking the kernel often. But in a world of increasing interactions and communications between machines — all of which require kernel intervention — I do not think this is a wise optimization strategy. Distributed computing stresses the importance of low latency, both because throughput can actually suffer if machines spend time waiting for each others' responses rather than doing work, and because there are so many interactions with other machines that even a small delay in each is magnified, leading to uneven response time to the user.

Improvement is clearly required to ensure consistent performance and controlled latencies, particularly when processing richer media like interactive sound and video. For example, in an application involving 8-bit audio sampled at 8KHz, using a 4096-byte buffer leads to a 1/2-second delay per stage of processing. This is unacceptable for real-time, interactive audio work. The basic system overhead must be reduced so that time-sensitive applications can use smaller buffers, reducing latency while maintaining throughput. But there is little room for revolutionary increases in performance when the fundamental operating system mechanisms, such as system call dispatch and context switch, are slow, and furthermore, show no trend in becoming faster. In general, existing designs have not focused on lower-level, low-overhead mechanisms, preferring instead to solve performance problems with more buffering.

This dissertation shows that the unusual goal of providing high throughput with low latency can be achieved. There are many factors in the design of Synthesis that accomplish this result, which will be discussed at length in subsequent chapters. But let us now consider four important aspects of the Synthesis design that depart from common precedents and trends.

2.3 Kernel Structure

2.3.1 The Trend from Monolithic to Diffuse

Early kernels tended to be large, isolated, monolithic structures that were hard to maintain. IBM's MVS is a classic example [11]. Unix initially embodied the "small is beautiful" ideal [28]. It captured some of the most elegant ideas of its day in a kernel design that, while still monolithic, was small, easy to understand and maintain, and provided a synergistic, productive, and highly portable set of system tools. However, its subsequent evolution and gradual accumulation of new services resulted in operating systems like System V and Berkeley's BSD 4.3, whose large, sprawling kernels hearken back to MVS.

These problems became apparent to several research teams, and a number of new system projects intended to address the problem were begun. For example, recognizing the need for clean, elegant services, the Mach group at CMU started with the BSD kernel and factored services into user-level tasks, leaving behind a very small kernel of common, central services [1]. Taking a different approach, the Plan 9 group at AT&T Bell Laboratories chose to carve the monolithic kernel into three sub-kernels, one for managing files, one for computation, and one for user interfaces [24]. Their idea is to more accurately and flexibly fit the networks of heterogeneous machines that are common in large organizations today.

There are difficulties with all these approaches. In the case of Mach, the goal of *kernelizing* the system by placing different services into separate user-level tasks forces additional parameter passing and context switches, adding overhead to every kernel invocation. Communication between the pieces relies heavily on message passing and remote procedure call. This adds considerable overhead despite the research that has gone into making them fast [12]. While Mach has addressed the issues of monolithic design and maintainability, it exacerbates the overhead and latency of system services. Plan 9 has chosen to focus on a particular cut of the system: large networks of machines. While it addresses the chosen problem well and extends the productive virtues of Unix, its arrangement may not be as suitable for other machine topologies or features, for example, the isolated workstation in a private residence, or those with richer forms of input and output, such as sound and video, which I believe will be common in the near future.

In a sense, kernelized systems can hide ugliness by partitioning it away. The kernel alone is not useful without a great deal of affiliated user-level service. Many papers publish numbers touting small kernel sizes but these hide the large amount of code that has been moved to user-level services. Some people argue that the size of user-level services does not count as much, because they are pageable and are not constrained to occupy real memory. But I argue: is it really a good idea to page out operating system services? This can only result in increased latency and unpredictable response time.

In general, I agree that the diffusion of the kernel structure is a good idea but find it unfortunate that current-generation kernelized systems tend to be slow, even in spite of ongoing efforts to make them faster. Perhaps people commonly accept that some loss of performance is the inevitable result of partitioning, and are willing to suffer that loss in return for greatly increased maintainability and extensibility.

My dissertation shows that this need not be the case: Synthesis addresses the issues of structuring and performance. Its quaject-based kernel structure keeps the modularity, protection, and extensibility demanded of modern-day operating systems. At the same time Synthesis delivers performance an order of magnitude better than existing systems, as evidenced by the experiments in Chapter 7. Its kernel services are subdivided into even finer chunks than kernelized systems like Mach. Any service can be composed of pieces that run at either user- or kernel-level: the distinction is blurred.

Synthesis breaks the batch-mode thinking that has led to systems that wait for all the data to arrive before any subsequent processing is allowed to take place, when in fact subsequent processing could proceed in parallel with the continuing arrival of data. Witness a typical system's handling of network packets: the whole packet is received, buffered, and checksummed before being handed over for further processing, when instead the address fields could be examined and lookups performed in parallel with the reception of the rest of the packet, reducing packet handling latency. Some network gateways do this type of cut-through routing for packet forwarding. But in a general-purpose operating system, the high overhead of system calls and context switches in existing systems discourage this type of thinking in preference to batching. By reconsidering the design, Synthesis compounds the savings. Low-overhead system calls and context switches encourage frequent use to better streamline processing and take advantage of the inherent parallelism achieved by a pipeline, reducing overhead and latency even further.

2.3.2 Services and Interfaces

A good operating system provides numerous useful services to make applications easy to write and easy to interconnect. To this end, it establishes conventions for packaging applications so that formats and interfaces are reasonably well standardized. The conventions encompass two forms: the *model*, which refers to the set of abstractions that guide the overall thinking and design; and the *interface*, which refers to the set of operations supported and how they are invoked. Ideally, we want a simple model, a powerful interface, and high performance. But these three are often at odds.

Witness the MVS I/O system, which has a complex model but offers a powerful interface and high performance. Its numerous options offer the benefit of detailed, precise control over each device, but with the drawback that even simple I/O requires complex programming.

UNIX is at the other end of the scale. UNIX promoted the idea of encapsulating I/O in terms of a single, simple abstraction. All common I/O is accomplished by reading or writing a stream of bytes to a file-like object, regardless of whether the I/O is meant to be viewed on the the user's terminal, stored as a file on disk, or used as input to another program. Treating I/O in a common manner offers great convenience and utility. It becomes trivial to write and test a new program, viewing its output on the screen. Once the program is working, the output can be sent to the intended file on disk without changing a line of code or recompiling.

But an oversimplified model of I/O brings with it a loss of precise control. This loss is not important for the great many UNIX tools — it is more than compensated by the synergies of a diverse set of connectable programs. But other, more complex applications such as a database management system (DBMS) require more detailed control over I/O [30]. Minimally, for a DBMS to provide reasonable crash recovery, it must know when a `write` operation has successfully finished placing the data on disk; in UNIX, a `write` only copies the data to a kernel buffer, movement of data from there to disk occurs later, asynchronously, so in the event of an untimely crash, data waiting in the buffers will be lost. Furthermore, a well-written DBMS has a good idea as to which areas of a file are likely to be needed in the future and its performance improves if this knowledge can be communicated to the operating system; by contrast, UNIX hides the details of kernel buffering, impeding such optimizations in exchange for a simpler interface.

Later versions of UNIX extended the model, making up some of the loss, but these extensions were not “clean” in the sense of the original UNIX design. They were added piecemeal as the need arose. For example, `ioctl` (for I/O controls) and the `select` system call help support out-of-band stream controls and non-blocking (polled) I/O, but these solutions are neither general nor uniform. Furthermore, the granularity with which UNIX considers an operation “‘non-blocking” is measured in tens of milliseconds. While this was acceptable for the person-typing-on-a-terminal mode of user interaction of the early 1980’s, it is clearly inappropriate for handling higher rate interactive data, such as sound and video.

Interactive games and real-time processing are two examples of areas where the classic models are insufficient. UNIX and its variants have no asynchronous `read`, for example, that would allow a program to monitor the keyboard while also updating the player’s screen. A conceptually simple application to record a user’s typing along with its timing and later play it back with the correct timing takes several pages of code to accomplish under UNIX, and then it cannot be done well enough if, say, instead of a keyboard we have a musical instrument.

The newer systems, such as Mach, provide extensions and new capabilities but within the framework of the same basic model, hence the problems persist. The result is that the finer aspects of stream control, of real-time processing, or of the handling of time-sensitive data in general have not been satisfactorily addressed in existing systems.

2.3.3 Managing Diverse Types of I/O

The multiplexing of I/O and handling of the machine’s I/O devices is one of the three most important functions of an operating system. (Managing the processor and memory are the other two.) It is perhaps the most difficult function to perform well, because there can be many different types of I/O devices, each with its own special features and requirements.

Existing systems handle diverse types of I/O devices by defining a few common internal formats for I/O and mapping each device to the closest one. General-purpose routines in the kernel then operate on each format. UNIX, for example, has two major internal formats, which they call “I/O models”: the block model for disk-like devices and the character model for terminal-like devices [26].

But common formats force compromise. There is a performance penalty paid when mismatches between the native device format and the internal format make translations necessary. These translations can be expensive if the "distance" between the internal format and a particular device is large. In addition, some functionality might be lost, because common formats, however general, cannot capture everything. There could be some features in a device that do not map well into the chosen format and those features become difficult if not impossible to access. Since operating systems tend to be structured around older formats, chosen at time when the prevalent I/O devices were terminals and disks, it is not surprising that they have difficulty handling the new rich media devices, such as music and video.

Synthesis breaks this tradeoff. The quaject structuring of the kernel allows new I/O formats to be created to suit the performance characteristics of unusual devices. Indeed, it is not inconceivable that every device has its own format, specially tailored to precisely fit its characteristics. Differences between a device format and what the application expects are spanned using translation, as in existing systems. But unlike existing systems, where translation is used to map into a common format, Synthesis maps directly from the device format to the needs of the application, eliminating the intermediate, internal format and its associated buffering and translation costs. This lets knowledgable applications use the highly efficient device-level interfaces when very high performance and detailed control are of utmost importance, but also preserves the ability of any application to work with any device, as in the Unix common-I/O approach. Since the code is runtime-generated for each specific translation, performance is good. The efficient emulation of Unix under Synthesis bears witness to this.

2.3.4 Managing Processes

Managing the machine's processors is the second important function of an operating system. It involves two parts: multiplexing the processors among the tasks, and controlling task execution and querying its state. But in contrast to the many control and query functions offered for I/O, existing operating systems provide only limited control over task execution. For example, the Unix system call for this purpose, `ptrace`, works only between a parent task and its children. It is archaic and terribly inefficient, meant solely for use by debuggers and apparently implemented as an afterthought. Mach threads, while supporting some rudimentary calls, sometimes lacks desirable generality: a Mach thread cannot suspend itself, for example, and the thread controls do not work between threads in different tasks.

In this sense, Mach threads only add parallelism to an existing abstraction — the UNIX process — Mach does not develop the thread idea to its fullest potential. Both these systems lack general functions to start, stop, query, and modify an arbitrary task's execution without arrangements having been made beforehand, for example, by starting the task from within a debugger.

In contrast, Synthesis provides detailed thread control, comparable to the level of control found for other operating system services, such as I/O. Section 4.3.2 lists the operations supported, which work between any pair of threads, even between unrelated threads in different address spaces and even on the kernel's threads, if there is sufficient privilege. Because of their exceptionally low overhead — only ten to twenty times the cost of a null procedure call — they provide unprecedented data collection and measurement abilities and unparalleled support for debuggers.

3

Kernel Code Generator

For, behold, I create new heavens and a new earth.
— The Bible, Isaiah

3.1 Fundamentals

Kernel code synthesis is the name given to the idea of creating executable machine code at runtime as a means of improving operating system performance. This idea distinguishes Synthesis from all other operating systems research efforts, and is what helps make Synthesis efficient.

Runtime code generation is the process of creating executable machine code during program execution for use later during the same execution [16]. This is in contrast to the usual way, where all the code that a program runs has been created at compile time, before program execution starts. In the case of an operating system kernel like Synthesis, the "program" is the operating system kernel, and the term "program execution" refers to the kernel's execution, which lasts from the time the system is started to the time it is shut down.

There are performance benefits in doing runtime code generation because there is more information available at runtime. Special code can be created based on the particular data to be processed, rather than relying on general-purpose code that is slower. Runtime code generation can extend the benefits of detailed compile-time analysis by allowing certain data-dependent optimizations to be postponed to runtime, where they can be done more effectively because there is more information about the data. We want to make the best possible use of the information available at compile-time, and use runtime code generation to optimize data-dependent execution.

The goal of runtime code generation can be stated simply:

Never evaluate something more than once.

For example, suppose that the expression "$A * A + A * B + B * B$" is to be evaluated for many different A while holding $B = 1$. It is more efficient to evaluate the reduced expression obtained by replacing B with 1: "$A * A + A + 1$". Finding opportunities for such optimizations and performing them is the focus of this chapter.

The problem is one of knowing how soon we can know what value a variable has, and how that information can be used to improve the program's code. In the previous example, if it can be deduced at compile time that $B = 1$, then a good compiler can perform precisely the reduction shown. But usually we can not know ahead of time what value a variable will have. B might be the result of a long calculation whose value is hard if not impossible to predict until the program is actually run. But when it is run, and we know B, runtime code generation allows us to use the newly-acquired information to reduce the expression.

Specifically, we create specialized code once the value of B becomes known, using an idea called *partial evaluation* [15]. Partial evaluation is the building of simpler, easier-to-evaluate expressions from complex ones by substituting variables that have a known, constant value with that constant. When two or more of these constants are combined in an arithmetic or logical operation, or when one of the constants is an identity for the operation, the operation can be eliminated. In the previous example, we no longer have to compute $B * B$, since we know it is 1, and we do not need to compute $A * B$, since we know it is A.

There are strong parallels between runtime code generation and compiler code generation, and many of the ideas and terminology carry over from one to the other. Indeed, anything that a compiler does to create executable code can also be performed at runtime. But because compilation is an off-line process, there is usually less concern about the cost of code generation and therefore one has a wider palette of techniques to choose from. A compiler can afford to use powerful, time-consuming analysis methods and perform sophisticated optimizations — a luxury not always available at runtime.

Three optimizations are of special interest to us, not only because they are easy to do, but because they are also effective in improving code quality. They are: *constant folding*, *constant propagation*, and *procedure inlining*. Constant folding replaces constant expressions like $5 * 4$ with the equivalent value, 20. Constant propagation replaces variables that have known, constant value with that constant. For example, the fragment $x = 5; y = 4 * x$ becomes $x = 5; y = 4 * 5$ through constant propagation; $4 * 5$ then becomes 20 through constant folding. Procedure inlining substitutes the body of a procedure, with its local variables appropriately renamed to avoid conflicts, in place of its call.

There are three costs associated with runtime code generation: creation cost, paid each time a piece of code is created; execution cost, paid each time the code is used; and management costs, to keep track of where the code is and how it is being used. The hope is to use the information available at runtime to create better code than would otherwise be possible. In order to win, the savings of using the runtime-created code must exceed the cost of creating and managing that code. This means that for many applications, a fast code generator that creates good code will be superior to a slow code generator that creates excellent code. (The management problem is analogous to keeping track of ordinary, heap-allocated data structures, and the costs are similar, so they will not be considered further.)

Synthesis focuses on techniques for implementing very fast runtime code generation. The goal is to broaden its applicability and extend its benefits, making it cheap enough so that even expressions and procedures that are not re-used often still benefit from having their code custom-created at runtime. To this end, the places where runtime code generation is used are limited to those where it is clear at compile time what the possible reductions will be. The following paragraphs describe the idea, while the next section describes the specific techniques.

A fast runtime code generator can be built by making full use of the information available at compile time. In our example, we know at compile time that B will be held constant, but we do not know what the constant will be. But we can predict at compile-time what form the reduced expression will have: $A * A + C_1 * A + C_2$. Using this knowledge, we can build a simple code generator for the expression that copies a code template representing $A * A + C_1 * A + C_2$ into newly allocated memory and computes and fills the constants: $C_1 = B$ and $C_2 = B * B$. A code template is a fragment of code which has been compiled but contains "holes" for key values.

Optimizations to the runtime-created code can also be pre-computed. In this example, interesting optimizations occur when B is 0, 1, or a power of two. Separate templates for each of these cases allow the most efficient code possible to be generated. The point is that there is plenty of information available at compile time to allow not just simple substitution of variables by constants, but also interesting and useful optimizations to happen at runtime with minimal analysis.

The general idea is: treat runtime code generation as if it were just another "function" to be optimized, and apply the idea of partial evaluation recursively. That is, just as in the previous example we partially-evaluate the expression $A * A + A * B + B * B$ with respect to the variable held constant, we can partially-evaluate the optimizations with respect to the parameters that the functions will be specialized under, with the result being specialized code-generator functions.

Looking at a more complex example, suppose that the compiler knows, either through static control-flow analysis, or simply by the programmer telling it through some directives, that the function $f(p1, ...) = 4 * p1 + ...$ will be specialized at runtime for constant $p1$. The compiler can deduce that the expression $4*p1$ will reduce to a constant, but it does not know what particular value that constant will have. It can capture this knowledge in a custom code generator for f that computes the value $4 * p1$ when $p1$ becomes known and stores it in the correct spot in the machine code of the specialized function f, bypassing the need for analysis at runtime. In another example, consider the function g, `g(p1, ...) = if(p1 < 10) S1; else S2;`, also to be specialized for constant parameter $p1$. Since parameter $p1$ will be constant, we know at compile time that the `if` statement will be either always true, or always false. We just don't know which. But again, we can create a specialized generator for g, one that evaluates the conditional when it becomes known and emits either $S1$ or $S2$ depending on the result.

The idea applies recursively. For example, once we have a code generator for a particular kind of expression or statement, that same generator can be used each time that kind of expression occurs, even if it is in a different part of the program. Doing this limits the proliferation of code generators and keeps the program size small. The resulting runtime code generator has a hierarchical structure, with generators for the large functions calling sub-generators to create the individual statements, which in turn call yet lower-level generators, and so on, until at the bottom we have very simple generators that, for example, move a constant into a machine register in the most efficient way possible.

3.2 Methods of Runtime Code Generation

The three methods Synthesis uses to create machine code are: *factoring invariants*, *collapsing layers*, and *executable data structures*.

3.2.1 Factoring Invariants

The factoring invariants method is equivalent to partial evaluation where it is known at compile time the variables over which a function will be partially evaluated. It is based on the observation that a functional restriction is usually easier to calculate than the original function. Consider a general function:

$$F_{big}(p_1, p_2, \ldots, p_n).$$

If we know that parameter p_1 will be held constant over a set of invocations, we can factor it out to obtain an equivalent composite function:

$$[F^{create}(p_1)](p_2, \ldots, p_n) \equiv F_{big}(p_1, p_2, \ldots, p_n).$$

F^{create} is a second-order function. Given the parameter p_1, F^{create} returns another function, F_{small}, which is the restriction of F_{big} that has absorbed the constant argument p_1:

$$F_{small}(p_2, \ldots, p_n) \subset F_{big}(p_1, p_2, \ldots, p_n).$$

If F^{create} is independent of global data, then for a given p_1, F^{create} will always compute the same F_{small} regardless of global state. This allows $F^{create}(p_1)$ to be evaluated once and the resulting F_{small} used thereafter. If F_{small} is executed m times, generating and using it pays off when

$$Cost(F^{create}) + m * Cost(F_{small}) < m * Cost(F_{big}).$$

As the "factoring invariants" name suggests, this method resembles the constant propagation and constant folding optimizations done by compilers. The analogy is strong, but the difference is also significant. Constant folding eliminates static code and calculations. In addition, Factoring Invariants can also simplify dynamic data structure traversals that depend on the constant parameter $p1$.

For example, we can apply this idea to improve the performance of the `read` system function. When `read`ing a particular file, constant parameters include the device that the file resides on, the address of the kernel buffers, and the process performing the `read`. We can use file `open` as F^{create}; the F_{small} it generates becomes our `read` function. F^{create} consists of many small procedure templates, each of which knows how to generate code for a basic operation such as "read disk block", "process TTY input", or "enqueue data." The parameters passed to F^{create} determine which of these code-generating procedures are called and in what order. The final F_{small} is created by filling these templates with addresses of the process table, device registers, and the like.

3.2.2 Collapsing Layers

The collapsing layers method is equivalent to procedure inlining where it is known at compile time which procedures might be inlined. It is based on the observation that in a layered design, separation between layers is a part of specification, not implementation. In other words, procedure calls and context switches between functional layers can be bypassed at execution time. Let us consider an example from the layered OSI model:

$$F_{big}(p_1, p_2, \ldots, p_n) = F_{applica}(p_1, F_{present}(p_2, F_{session}(\ldots F_{datalnk}(p_n)\ldots))).$$

$F_{applica}$ is a function at the Application layer that calls successive lower layers to send a message. Through in-line code substitution of $F_{present}$ in $F_{applica}$, we can obtain an equivalent flat function by eliminating the procedure call from the Application to the Presentation layer:

$$F^{flat}_{applica}(p_1, p_2, F_{session}(\ldots)) \equiv F_{applica}(p_1, F_{present}(p_2, F_{session}(\ldots))).$$

The process to eliminate the procedure call can be embedded into two second-order functions. $F^{create}_{present}$ returns code equivalent to $F_{present}$ and suitable for in-line insertion. $F^{create}_{applica}$ incorporates that code to generate $F^{flat}_{applica}$.

$$F^{create}_{applica}(p_1, F^{create}_{present}(p_2, \ldots), F^{flat}_{applica}(p_1, p_2, \ldots)).$$

This technique is analogous to in-line code substitution for procedure calls in compiler code generation. In addition to the elimination of procedure calls, the resulting code typically exhibit opportunities for further optimization, such as Factoring Invariants and elimination of data copying.

By induction, $F^{create}_{present}$ can eliminate the procedure call to the Session layer, and down through all layers. When we execute $F^{create}_{applica}$ to establish a virtual circuit, the $F^{flat}_{applica}$ code used thereafter to send and receive messages may consist of only sequential code. The performance gain analysis is similar to the one for factoring invariants.

3.2.3 Executable Data Structures

The executable data structures method reduces the traversal time of data structures that are frequently traversed in a preferred way. It works by storing node-specific traversal code along with the data in each node, making the data structure *self-traversing*.

Consider an active job queue managed by a simple round-robin scheduler. Each element in the queue contains two short sequences of code: `stopjob` and `startjob`. The `stopjob` saves the registers and branches into the next job's `startjob` routine (in the next element in queue). The `startjob` restores the new job's registers, installs the address of its own `stopjob` in the timer interrupt vector table, and resumes processing.

An interrupt causing a context switch will execute the current program's `stopjob`, which saves the current state and branches directly into the next job's `startjob`. Note that the scheduler has been taken out of the loop. It is the queue itself that does the context switch, with a critical path on the order of ten machine instructions. The scheduler intervenes only to insert and delete elements from the queue.

3.2.4 Performance Gains

Runtime code generation and partial evaluation can be thought of as a way of caching frequently visited states. It is interesting to contrast this type of caching with the caching that existing systems do using ordinary data structures. Generally, systems use data structures to capture state and remember expensive-to-compute values. For example, when a file is opened, a data structure is built to describe the file, including its location on disk and a pointer to the procedure to be used to read it. The `read` procedure interprets state stored in the data structure to determine what work is to be done and how to do it.

In contrast, code synthesis encodes state directly into generated procedures. The resulting performance gains extend beyond just saving the cost of interpreting a data structure. To see this, let us examine the performance gains obtained from hard-wiring a constant directly into the code compared to fetching it from a data structure. Hardwiring embeds

```
char buf[100], *bufp = &buf[0], *endp = &buf[100];

Put(c)
{
    *bufp++ = c;
    if(bufp == endp)
        flush();
}

Put:   // (character is passed register d0)
    move.l  (bufp),a0   // (1) Load buffer pointer into register a0
    move.b  d0,(a0)+    // (2) Store the character and increment the a0 register
    move.l  a0,(bufp)   // (3) Update the buffer pointer
    cmp.l   (endp),a0   // (4) Test for end-of-buffer
    beq     flush       // ... if end, jump to flush routine
    rts                 // ... otherwise return
```

Figure 3.1: Hand-crafted assembler implementation of a buffer

the constant in the instruction stream, so there is an immediate savings that comes from eliminating one or two levels of indirection and obviating the need to pass the structure pointer. These can be attributed to "saving the cost of interpretation." But hardwiring also opens up the possibility of further optimizations, such as constant folding, while fetching from a data structure admits no such optimizations. Constant folding becomes possible because once it is known that a parameter will be, say, 2, all pure functions of that parameter will likewise be constant and can be evaluated once and the constant result used thereafter. A similar flavor of optimization arises with `IF`-statements. In the code fragment "`if(C) S1; else S2;`", where the conditional, `C`, depends only on constant parameters, the generated code will contain either `S1` or `S2`, never both, and no test. It is with this cascade of optimization possibilities that code synthesis obtains its most significant performance gains. The following section illustrates some of the places in the kernel where runtime code generation is used to advantage.

3.3 Uses of Code Synthesis in the Kernel

3.3.1 Buffers and Queues

Buffers and queues can be implemented more efficiently with runtime code generation than without.

```
Put:   // (character is passed register d0)
    move.l  (P),a0       // Load buffer pointer into register a0
    move.b  d0,(a0,D)    // Store the character
    addq.w  #1,(P+2)     // Update the buffer pointer and test if reached end
    beq     flush        // ... if end, jump to flush routine
    rts                  // ... otherwise return
```

Figure 3.2: Better buffer implementation using code synthesis

	Cold cache	Warm cache
Code-synthesis (CPU cycles)	29	20
Hand-crafted assembly (CPU cycles)	37	28
Speedup	1.4	1.4

68030 CPU, 25MHz, 1-wait-state main memory

Table 3.1: CPU Cycles for Buffer-Put

Figure 3.1 shows a good, hand-written 68030 assembler implementation of a buffer. The C language code illustrates the intended function, while the 68030 assembler code shows the work involved. The work consists of: (1) loading the buffer pointer into a machine register; (2) storing the character in memory while incrementing the pointer register; (3) updating the buffer pointer in memory; and (4) testing for the end-of-buffer condition. This fragment executes in 28 machine cycles not counting the procedure call overhead.

Figure 3.2 shows the code-synthesis implementation of a buffer, which is 40% faster. Table 3.1 gives the actual measurements. The improvement comes from the elimination of the `cmp` instruction, for a savings of 8 cycles. The code relies on the implicit test for zero that occurs at the end of every arithmetic operation. Specifically, we arrange that the lower 16 bits of the pointer variable be zero when the end of buffer is reached, so that incrementing the pointer also implicitly tests for end-of-buffer.

This is done for a general pointer as follows. The original `bufp` pointer is represented as the sum of two quantities: a pointer-like variable, P, and a constant displacement, D. Their sum, $P + D$, gives the current position in the buffer, and takes the place of the

10^7 Executions of:	Execution time, seconds	Size: Bytes/Invocation
UNIX "putchar" macro	21.4 user; 0.1 system	132
Synthesis "putchar" macro	13.0 user; 0.1 system	30
Synthesis "putchar" function	19.0 user; 0.1 system	8

Table 3.2: Comparison of C-Language "stdio" Libraries

original `bufp` pointer. The character is stored in the buffer using the "`move.b d0,(a0,D)`" instruction which is just as fast as a simple register-indirect store. The displacement, D, is chosen so that when $P + D$ points to the end of the buffer, P is 0 modulo 2^{16}, that is, the least significant 16 bits of P are zero. The "`addq.w #1,(P+2)`" instruction then increments the lower 16 bits of the buffer pointer and also implicitly tests for end-of-buffer, which is indicated by a 0 result. For buffer sizes greater than 2^{16} bytes, the `flush` routine can propagate the carry-out to the upper bits, flushing the buffer when the true end is reached.

This performance gain can only be had using runtime code generation, because D must be a constant, embedded in the buffer's machine code, to take advantage of the fast memory-reference instruction. Were D a variable, the loss of fetching its value and indexing would offset the gain from eliminating the compare instruction. The 40% savings is significant because buffers and queues are used often. Another advantage is improved locality of reference: code synthesis puts both code and data in the same page of memory, increasing the likelihood of cache hits in the memory management unit's address translation cache.

Outside the kernel, the Synthesis implementation of the C-language I/O library, "stdio," uses code-synthesized buffers at the user level. In a simple experiment, I replaced the UNIX stdio library with the Synthesis version. I compiled and ran a simple test program that invokes the `putchar` macro ten million times, using first the native UNIX stdio library supplied with the Sony NEWS workstation, and then the Synthesis version. Table 3.2 shows the Synthesis macro version is 1.6 times faster, and over 4 times smaller, than the UNIX version.

The drastic reduction in code size comes about because code synthesis can take advantage of the extra knowledge available at runtime to eliminate execution paths that cannot be taken. The `putchar` operation, as defined in the C library, actually supports three kinds of buffering: block-buffered, line-buffered and unbuffered. Even though only one of these can be in effect at any one time, the C `putchar` macro must include code to handle all of them, since it cannot know ahead of time which one will be used. In contrast, code synthesis creates only the code handling the kind of buffering actually desired for the particular file being written to. Since `putchar`, being a macro, is expanded in-line every time it appears in the source code, the savings accumulate rapidly.

Table 3.2 also shows that the Synthesis "putchar" *function* is slightly faster than the UNIX macro — a dramatic result, that even incurring a procedure call overhead, code synthesis still shows a speed advantage over conventional code in-lined with a macro.

3.3.2 Context Switches

One reason that context switches are expensive in traditional systems like UNIX is that they always save and restore the entire CPU context, even though that may not be necessary. For example, a process that did not use floating point since it was switched in does not need to have its floating-point registers saved when it is switched out. Another reason is that saving context is often implemented as a two-step procedure: the CPU registers are first placed in a holding area, freeing them so they can be used to perform calculations and traverse data structures to find out where the context was to have been put, and finally copying it there from the holding area.

A Synthesis context switch takes less time because only the part of the context being used is preserved, not all of it, and because the critical path traversing the ready queue is minimized with an executable data structure.

The first step is to know how much context to preserve. Context switches can happen synchronously or asynchronously with thread execution. Asynchronous context switches are the result of external events forcing preemption of the processor, for example, at the end of a CPU quantum. Since they can happen at any time, it is hard to know in advance how much context is being used, so we preserve all of it. Synchronous context switches, on the other hand, happen as a result of the thread requesting them, for example, when relinquishing the CPU to wait for an I/O operation to finish. Since they occur in specific, well-defined points in the thread's execution, we can know exactly how much context will be needed and therefore can arrange to preserve only that much. For example, suppose a

```
proc:
        :
        :
    {Save necessary context}
    bsr     swtch
res:
    {Restore necessary context}
        :
        :

swtch:
    move.l  (Current),a0        // (1) Get address of current thread's TTE
    move.l  sp,(a0)             // (2) Save its stack pointer
    bsr     find_next_thread    // (3) Find another thread to run
    move.l  a0,(Current)        // (4) Make that one current
    move.l  (a0),sp             // (5) Load its stack pointer
    rts                         // (6) Go run it!
```

Figure 3.3: Context Switch

`read` procedure needs to block and wait for I/O to finish. Since it has already saved some registers on the stack as part of the normal procedure-call mechanism, there is no need to preserve them again as they will only be overwritten upon return.

Figure 3.3 illustrates the general idea. When a kernel thread executes code that decides that it should block, it saves whatever context it wishes to preserve on the active stack. It then calls the scheduler, `swtch`; doing so places the thread's program counter on the stack. At this point, the top of stack contains the address where the thread is to resume execution when it unblocks, with the machine registers and the rest of the context below that. In other words, the thread's context has been reduced to a single register: its stack pointer. The scheduler stores the stack pointer into the thread's control block, known as the *thread table entry* (TTE), which holds the thread state when it is not executing. It then selects another thread to run, shown as a call to the `find_next_thread` procedure in the figure, but actually implemented as an executable data structure as discussed later. The variable `Current` is updated to reflect the new thread and its stack pointer is loaded into the CPU. A return-from-subroutine (`rts`) instruction starts the thread running. It continues where it had left off (at label `res`), where it pops the previously-saved state off the stack and proceeds with its work.

Figure 3.4 shows two TTEs. Each TTE contains code fragments that help with context switching: `sw_in` and `sw_in_mmu`, which loads the processor state from the TTE; and `sw_out`, which stores processor state back into the TTE. These code fragments are created specially for each thread. To switch in a thread for execution, the processor executes the

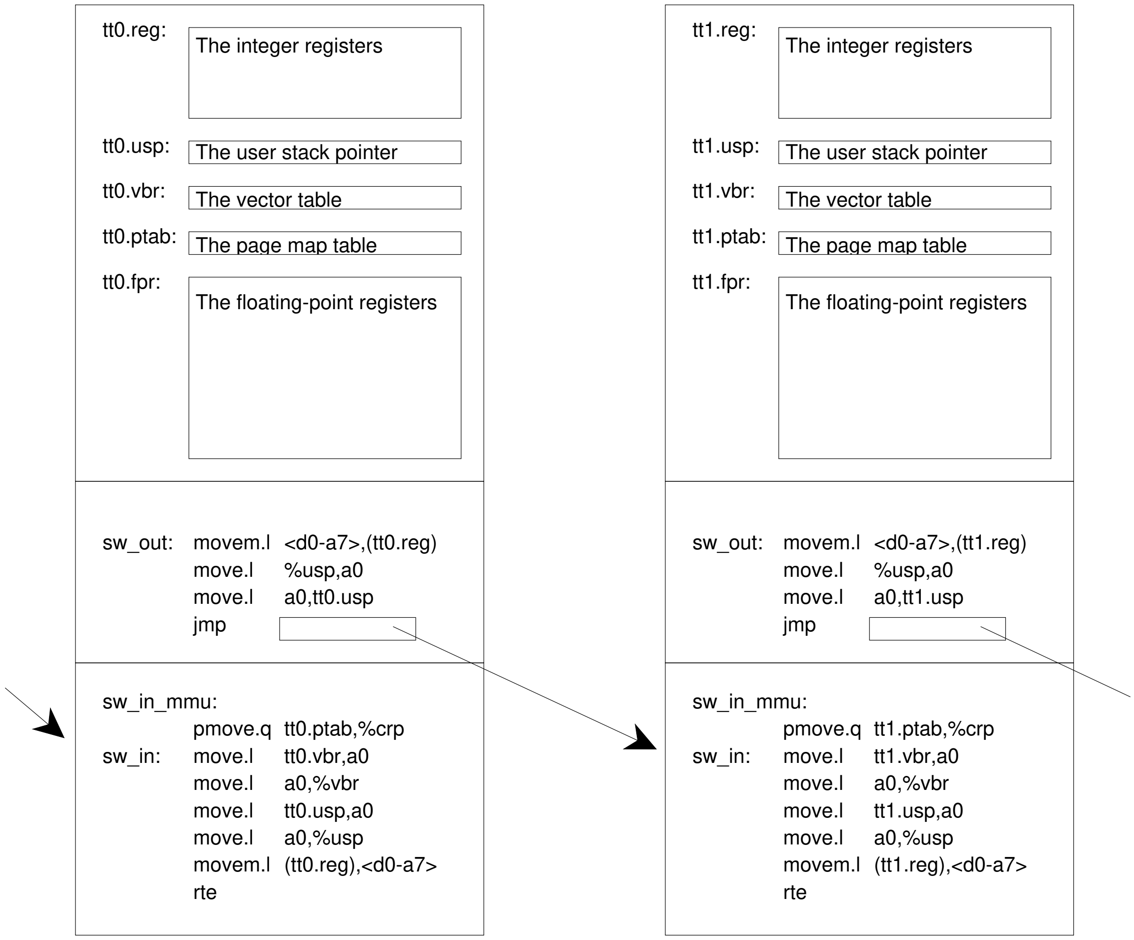


Figure 3.4: Thread Context

thread's `sw_in` or `sw_in_mmu` procedure. To switch out a thread, the processor executes the thread's `sw_out` procedure.

Notice how the ready-to-run threads (waiting for CPU) are chained in an executable circular queue. A `jmp` instruction at the end of the `sw_out` procedure of the preceding thread points to the `sw_in` procedure of the following thread. Assume thread-0 is currently running. When its time quantum expires, the timer interrupt is vectored to thread-0's `sw_out`. This procedure saves the CPU registers into thread-0's register save area (`TT0.reg`). The `jmp` instruction then directs control flow to one of two entry points of the next thread's (thread-1) context-switch-in procedure, `sw_in` or `sw_in_mmu`. Control flows to `sw_in_mmu` when a change of address space is required; otherwise control flows to `sw_in`. The switch-in procedure then

Type of context switch	Time (μs)
Integer registers only	10.5
Floating-point	52
Integer, change address space	16
Floating-point, change address space	56
Null procedure call (C language)	1.4
Sony NEWS, UNIX	170
NeXT Machine, Mach	510

68030 CPU, 25MHz, 1-wait-state main memory, cold cache

Table 3.3: Cost of Thread Scheduling and Context Switch

loads the CPU's vector base register with the address of thread-1's vector table, restores the processor's general registers, and resumes execution of thread-1. The entire switch takes 10.5 microseconds to switch integer-only contexts between threads in the same address space, or 56 microseconds including the floating point context and a change in address space.[1]

Table 3.3 summarizes the time taken by the various types of context switches in Synthesis, saving and restoring all the integer registers. These times include the hardware interrupt service overhead — they show the elapsed time from the execution of the last instruction in the suspended thread to the first instruction in the next thread. Previously published papers report somewhat lower figures [25] [18]. This is because they did not include the interrupt-service overhead, and because of some extra overhead incurred in handling the 68882 floating point unit on the Sony NEWS workstation that does not occur on the Quamachine, as discussed later. For comparison, a call to a null procedure in the C language takes 1.4 microseconds, and the Sony UNIX context switch takes 170 microseconds.

[1]Previous papers incorrectly cite a floating-point context switch time of 15 μs [25] [18]. This error is believed to have been caused by a bug in the Synthesis assembler, which incorrectly filled the operand field of the floating-point move-multiple-registers instruction causing it to preserve just one register, instead of all eight. Since very few Synthesis applications use floating point, this bug remained undetected for a long time.

In addition to reducing ready-queue traversal time, specialized context-switch code enables further optimizations, to move only needed data. The previous paragraph already touched on one of the optimizations: bypassing the MMU address space switch when it is not needed. The other optimizations occur in the handling of floating point registers, described now, and in the handling of interrupts, described in the next section.

Switching the floating point context is expensive because of the large amount of state that must be saved. The registers are 96 bits wide; moving all eight registers requires 24 transfers of 32 bits each. The 68882 coprocessor compounds this cost, because each word transferred requires two bus cycles: one to fetch it from the coprocessor, and one to write it to memory. The result is that it takes about 50 microseconds just to save and restore the hundred-plus bytes of information comprising the floating point coprocessor state. This is more than five times the cost of doing an entire context switch without the floating point.

Since preserving floating point context is so expensive, we use runtime tests to see if floating point had been used to avoid saving state that is not needed. Threads start out assuming floating point will not be used, and their context-switch code is created without it. When context-switching out, the context-save code checks whether the floating point unit had been used. It does this using the `fsave` instruction of the Motorola 68882 floating point coprocessor, which saves only the internal microcode state of the floating point processor [20]. The saved state can be tested to see if it is not null. If so, the user-visible floating-point state is saved, and the context-switch code re-created to include the floating-point context in subsequent context switches. Since the majority of threads in Synthesis do not use floating point, the savings are significant.

Unfortunately, after a thread executes its first floating point instruction, floating point context will have to be preserved from that point on, even if no further floating-point instructions are issued. The context must be restored upon switch-in because a floating point instruction might be executed. The context must be saved upon switch-out even if no floating point instructions had been executed since switch-in because the 68882 cannot detect a lack of instruction execution. It can only tell us if its state is completely null. This is bad because sometimes a thread may use floating-point at first, for example, to initialize a table, and then not again. But with the 68882, we can only optimize the case when floating point is never used.

The Quamachine has hardware to alleviate the problem. Its floating-point unit — also a 68882 — can be enabled and disabled by software command, allowing a lazyevaluation of floating-point context switches. Switching in a thread for execution loads its integer state and disables the floating-point unit. When a thread executes its first floating point instruction since the switch, it takes an illegal instruction trap. The kernel then loads the necessary state, first saving any prior state that may have been left there, re-enables the floating-point unit, and the thread resumes with the interrupted instruction. The trap is taken only on the first floating-point instruction following a switch, and adds only 3 μs to the overhead of restoring the state. This is more than compensated for by the other savings: integer context-switch becomes 1.5 μs faster because there is no need for an `fsave` instruction to test for possible floating-point use; and even floating-point threads benefit when they block without a floating point instruction being issued since they were switched in, saving the cost of restoring and then saving that context. Indeed, if only a single thread is using floating point, the floating point context is never switched, remaining in the coprocessor.

3.3.3 Interrupt Handling

A special case of context switching occurs in interrupt handling. Many systems, such as UNIX, perform a full context switch on each interrupt. For example, an examination of the running Sony UNIX kernel reveals that not only are all integer registers saved on each interrupt, but the active portion of the floating-point context as well. This is one of the reasons that interrupt handling is expensive on a traditional system, and the reason why the designers of those systems try hard to avoid frequent interrupts. As shown earlier, preserving the floating-point state can be very expensive. Doing so is superfluous unless the interrupt handler uses floating point; most do not.

Synthesis interrupt handling is faster because it saves and restores only the part of the context that will be used by the service routine, not all of it. Code synthesis allows partial context to be saved efficiently. Since different interrupt procedures use different amounts of context, we can not, in general, know how much context to preserve until the interrupt is linked to its service procedure. Furthermore, it may be desirable to change service procedures, for example, when changing or installing new I/O drivers in the running kernel. Without code synthesis, we would have to save the union of all contexts used by all procedures that could be called from the interrupt, slowing down all because of the needs of a few.

Examples taken from the Synthesis Sound-IO device driver illustrate the ideas and provide performance numbers. The Sound-IO device is a general-purpose, high-quality audio input and output device with stereo, 16-bit analog-to-digital and digital-to-analog converters, and a direct-digital input channel from a CD player. This device interrupts the processor once for every sound sample — 44100 times per second — a very high number by conventional measures. It is normally inconceivable to attach such high-rate interrupt sources to the main processor. Sony UNIX, for example, can service a maximum of 20,000 interrupts per second, and such a device could not be handled at all.[2] Efficient interrupt handing is mandatory, and the rest of this section shows how Synthesis can service high interrupt rates efficiently.

Several benefits of runtime code generation combine to improve the efficiency of interrupt handing in Synthesis: the use of the high-speed buffering code described in Section 3.3.1, the ability to create interrupt routines that save and restore only the part of the context being used, and the use of layer-collapsing to merge separate functions together.

Figure 3.5 shows the actual Synthesis code created to handle the Sound-IO interrupts when only the CD-player is active. It begins by saving a single register, `a0`, since that is the only one used. This is followed by the code for the specific sound I/O channels, in this case, the CD-player. The code is similar to the fast buffer described in 3.3.1, synthesized to move data from the CD port directly into the user's buffer. If the other input sources (such as the A-to-D input) also become active, the interrupt routine is re-written, placing their buffer code immediately following the CD-player's. The code ends by restoring the `a0` register and returning from interrupt.

Figure 3.6 shows the best I have been able to achieve using hand-written assembly language, without the use of code synthesis. Like the Synthesis version, this uses only a single register, so we save and restore only that one.[3] But without code synthesis, we must include code for all the Sound-IO sources — CD, AD, and DA — testing and branching around the parts for the currently inactive channels. In addition, we can no longer use the

[2] The Sony workstation has two processors, one of which is dedicated to I/O, including servicing I/O interrupts using a somewhat lighter-weight mechanism. They solve the overhead problem with specialized processors — a trend that appears to be increasingly common. But this solution compounds latency, and does not negate my point, which is that existing operating systems have high overhead that discourage frequent invocations.

[3] Most existing systems neglect even this simple optimization. They save and restore all the registers, all the time.

```
intr:
    move.l  a0,-(sp)        // Save register a0
    move.l  (P),a0          // Get buffer pointer into reg. a0
    move.l  (cd_port),(a0,D)// Store CD data into address P+D
    addq.w  #4,(P+2)        // Increment low 16 bits of P.
    beq     cd_done         // ... flush buffer if full
    move.l  (sp)+,a0        // Restore register a0
    rte                     // Return from interrupt
```

Figure 3.5: Synthesized Code for Sound Interrupt Processing – CD Active

```
s_intr:
    move.l  a0,-(sp)        // Save register a0
    tst.b   (cd_active)     // Is the CD device active?
    beq     cd_no           // ... no, jump
    move.l  (cd_buf),a0     // Get CD buffer pointer into reg. a0
    move.l  (cd_port),(a0)+ // Store CD data; increment pointer
    move.l  a0,(cd_buf)     // Update CD buffer pointer
    subq.l  #1,(cd_cnt)     // Decrement buffer count
    beq     cd_flush        // ... jump if buffer full
cd_no:
    tst.b   (ad_active)     // Is the AD device active?
    beq     ad_no           // ... no, jump
       :
       :  [handle AD device, similar to CD code]
       :
ad_no:
    tst.b   (da_active)     // Is the DA device active?
    beq     da_no           // ... no, jump
       :
       :  [handle DA device, similar to CD code]
       :
da_no:
    move.l  (sp)+,a0        // Restore register a0
    rte                     // Return from interrupt
```

Figure 3.6: Sound Interrupt Processing, Hand-Assembler

fast buffer implementation of section 3.3.1 because that requires code synthesis.

Figure 3.7 shows another version, this one written in C, and invoked by a short assembly-language dispatch routine. It preserves only those registers clobbered by C procedure calls, and is representative of a carefully-written interrupt routine in C.

The performance differences are summarized in Table 3.4. Measurements are divided into three groups. The first group, consisting of just a single row, shows the time taken by the hardware to process an interrupt and immediately return from it, without doing anything else. The second group shows the time taken by the various implementations of the interrupt handler when just the CD-player input channel is active. The third group is like the second, but with two active sources: the CD-player and AD channels.

```
s_intr:
    movem.l <d0-d2,a0-a2>,-(sp)
    bsr     _sound_intr
    movem.l (sp)+,<d0-d2,a0-a2>
    rte

sound_intr()
{
    if(cd_active) {
        *cd_buf++ = *cd_port;
        if(--cd_cnt < 0)
            cd_flush();
    }
    if(ad_active) {
        ...
    }
    if(da_active) {
        ...
    }
}
```

Figure 3.7: Sound Interrupt Processing, C Code

	Time in μS	Speedup
Null Interrupt	2.0	—
CD-in, code-synth	3.7	—
CD-in, assembler	6.0	2.4
CD-in, C	9.7	4.5
CD-in, C & UNIX	17.1	8.9
CD+DA, code-synth	5.1	—
CD+DA, assembler	7.7	1.8
CD+DA, C	11.3	3.0
CD+DA, C & UNIX	18.9	5.5

68030 CPU, 25MHz, 1-wait-state main memory, cold cache

Table 3.4: Processing Time for Sound-IO Interrupts

Within each group of measurements, there are four rows. The first three rows show the time taken by the code synthesis, hand-assembler, and C implementations of the interrupt handler, in that order. The code fragments measured are those of figures 3.5, 3.6, and 3.7; the C code was compiled on the Sony NEWS workstation using "`cc -O`". The last row shows the time taken by the C version of the handler, but dispatched the way that Sony UNIX does, preserving all the machines registers prior to the call. The left column tells the elapsed execution time, in microseconds. The right column gives the ratio of times between the code synthesis implementation and the others. The null-interrupt time is subtracted before computing the ratio to give a better picture of what the implementation-specific performance increases are.

As can be seen from the table, the performance gains of using code synthesis are impressive. With only one channel active, we get more than twice the performance of hand-written assembly language, almost five times more efficient than well-written C, and very nearly an order of magnitude better than traditional UNIX interrupt service. Furthermore, the non-code-synthesis versions of the driver supports only the two-channel, 16-bit linear-encoding sound format. Extending support, as Synthesis does, to other sound formats, such as μ-Law, either requires more tests in the sound interrupt handler or an extra level of format conversion code between the application and the sound driver. Either option adds overhead that is not present in the code synthesis version, and would increase the time shown.

With two channels active, the gain is still significant though somewhat less than that for one channel. The reason is that the overhead-reducing optimizations of code synthesis — collapsing layers and preserving only context that is used — become less important as the amount of work increases. But other optimizations of code synthesis, such as the fast buffer, continue to be effective and scale with the work load. In the limit, as the number of active channels becomes large, the C and assembly versions perform equally well, and the code synthesis version is about 40% faster.

3.3.4 System Calls

Another use of code synthesis is to minimize the overhead of invoking system calls. In Synthesis the term "system call" is somewhat of a misnomer because the Synthesis system interface is based on procedure calls. A Synthesis system call is really a procedure call

```
// --- User-level stub procedure ---
proc:
    moveq   #N,d2               // Load procedure index
    trap    #15                 // Trap to kernel
    rts                         // Return

// --- Dispatch to kernel procedure ---
trap15:
    cmp.w   #MAX,d2             // Check that procedure index is in range
    bhs     bad_call            // ... jump if not
    move.l  (tab$,pc,d2*4),a2   // Get the procedure address
    jsr     (a2)                // Call it
    rte                         // Return to user-level

    .align 4                    // Table of kernel procedure addresses...
tab$:
    dc.l    fn0, fn1, fn2, fn3, ..., fnN
```

Figure 3.8: User-to-Kernel Procedure Call

that happens to cross the protection boundary between user and kernel. This is important because, as we will see in Chapter 4, each Synthesis service has a set of procedures associated with it that delivers that service. Since the set of services provided is extensible, we need a more general way of invoking them. Combining procedure calls with runtime code generation lets us do this efficiently.

Figure 3.8 shows how. The generated code consists of two parts: a user part, shown at the top of the figure, and a kernel part, shown at the bottom. The user part loads the procedure index number into the `d2` register and executes the `trap` instruction, switching the processor into kernel mode where it executes the kernel part of the code, beginning at label `trap15`. The kernel part begins with a limit check on the procedure index number, ensuring that the index is inside the table area and preventing cheating by buggy or malicious user code that may pass a bogus number. It then indexes the table and calls the kernel procedure. The kernel procedure typically performs its own checks, such as verifying that all pointers are valid, before proceeding with the work. It returns with the `rte` instruction, which takes the thread back into user mode, where it returns control to the caller. Since the user program can only specify an index into the procedure table, and not the procedure address itself, only the allowed procedures may be called, and only at the appropriate entry points. Even if the user part of the generated code is overwritten either accidentally or maliciously, it can never cause the kernel to do something that could not have been done through some other, untampered, sequence of calls.

Runtime code generation gives the following advantages: each thread has its own table of vectors for exceptions and interrupts, including `trap 15`. This means that each thread's kernel calls vector directly to the correct dispatch procedure, saving a level of indirection that would otherwise have been required. This dispatch procedure, since it is thread-specific, can hard-wire certain constants, such as `MAX` and the base address of the kernel procedure table, saving the time of fetching them from a data structure.

Furthermore, by thinking of kernel invocation not as a system call — which conjures up thoughts of heavyweight processing and large overheads — but as a procedure call, many other optimizations become easier to see. For example, ordinary procedures preserve only those registers which they use; kernel procedures can do likewise. Procedure calling conventions also do not require that all the registers be preserved across a call. Often, a number of registers are allowed to be "trashed" by the call, so that simple procedures can execute without preserving anything at all. Kernel procedures can follow this same convention. The fact that kernel procedures are called from user level does not make them special; one merely has to properly address the issues of protection, which is discussed further in Section 3.4.2.

Besides dispatch, we also need to address the problem of how to move data between user space and kernel as efficiently as possible. There are two kinds of moves required: passing procedure arguments and return values, and passing large buffers of data. For passing arguments, the user-level stub procedures are generated to pass as many arguments as possible in the CPU registers, bypassing the expense of accessing the user stack from kernel mode. Return values are likewise passed in registers, and moved elsewhere as needed by the user-level stub procedure. This is similar in idea to using CPU registers for passing short messages in the V system [9].

Passing large data buffers is made efficient using virtual memory tricks. The main idea is: when the kernel is invoked, it has the user address space mapped in. Synthesis reserves part of each address space for the kernel. This part is normally inaccessible from user programs. But when the processor executes the `trap` and switches into kernel mode, the kernel part of the address space becomes accessible in addition to the user part, and the kernel procedure can easily move data back and forth using the ordinary machine instructions. Prior to beginning such a move, the kernel procedure checks that no pointer refers to locations outside the user's address space — an easy check due to the way the virtual addresses are chosen: a single limit-comparison (two instructions) suffices.

Further optimizations are also possible. Since the user-level stub is a real procedure, it can be in-line substituted into its caller. This can be done lazily — the stub is written so that each time a call happens, it fetches the return address from the stack and modifies that point in the caller. Since the stubs are small, space expansion is minimal. Besides being effective, this mechanism requires minimal support from the language system to identify potential in-lineable procedure calls.

Another optimization bypasses the kernel procedure dispatcher. There are 16 possible `trap`s on the 68030. Three of these are already used, leaving 13 free for other purposes, such as to directly call heavily-used kernel procedures. If a particular kernel procedure is expected to be used often, an application can invoke the `cache_procedure` call, and Synthesis will allocate an unused `trap`, set it to call the kernel procedure directly, and re-create the user-level stub to issue this trap. Since this trap directly calls the kernel procedure, there is no longer any need for a limit check or a dispatch table. Pre-assigned traps can also be used to import execution environments. Indeed, the Synthesis equivalent of the UNIX concept of "stdin" and "stdout" is implemented with cached kernel procedure calls. Specifically, `trap 1` writes to `stdout`, and `trap 2` reads from `stdin`.

Combining both optimizations results in a kernel procedure call that costs just a little more than a `trap` instruction. The various costs are summarized in Table 3.5. The middle block of measurements show the cost of various Synthesis null kernel procedure calls: the general-dispatched, non-inlined case; the general-dispatched, with the user-level stub inlined into the application's code; cached-kernel-trap, non-inlined; and cached-kernel-trap, inlined. For comparison, the cost of a null `trap` and a null procedure call in the C language is shown on the top two lines, and the cost of the trivial `getpid` system call in UNIX and Mach is shown on the bottom two lines.

3.4 Other Issues

3.4.1 Kernel Size

Kernel size inflation is an important concern in Synthesis due to the potential redundancy in the many F_{small} and F^{flat} programs generated by the same F^{create}. This could be particularly bad if layer collapsing were used too enthusiastically. To limit memory use, F^{create} can generate either in-line code or subroutine calls to shared code. The decision of

	μS, cold cache	μS, warm cache
C procedure call	1.2	1.0
Null trap	1.9	1.6
Kernel call, general dispatch	4.2	3.5
Kernel call, general, in-lined	3.5	2.9
Kernel call, cached-trap	3.5	2.7
Kernel call, cached and in-lined	2.7	2.1
UNIX, `getpid`	40.	—
Mach, `getpid`	88.	—

68030 CPU, 25MHz, 1-wait-state main memory

Table 3.5: Cost of Null System Call

when to expand in-line is made by the programmer writing F^{create}. Full, memory-hungry in-line expansion is usually reserved for specific uses where its benefits are greatest: the performance-critical, frequently-executed paths of a function, where the performance gains justify increased memory use. Less frequently executed parts of a function are stored in a common area, shared by all instances through subroutine calls.

In-line expansion does not always cost memory. If a function is small enough, expanding it in-line can take the same or less space than calling it. Examples of functions that are small enough include character-string comparisons and buffer-copy. For functions with many runtime-invariant parameters, the size expansion of inlining is offset by a size decrease that comes from not having to pass as many parameters.

In practice, the actual memory needs are modest. Table 3.6 shows the total memory used by a full kernel — including I/O buffers, virtual memory, network support, and a window system with two memory-resident fonts.

System Activity	Memory Use, as code + data (Kbytes)
Boot image for full kernel	140
One thread running	Boot + 0.4 + 8
File system and disk buffers	Boot + 6 + 400
100 threads, 300 open files	Boot + 80 + 1400

Table 3.6: Kernel Memory Requirements

3.4.2 Protecting Synthesized Code

The classic solutions used by other systems to protect their kernels from unauthorized tampering by user-level applications also work in the presence of synthesized code. Like many other systems, Synthesis needs at least two hardware-supported protection domains: a privileged mode that allows access to all the machine's resources, and a restricted mode that lets ordinary calculations happen but restricts access to resources. The privileged mode is called *supervisor mode*, and the restricted mode, *user mode*.

Kernel data and code — both synthesized and not — are protected using memory management to make the kernel part of each address space inaccessible to user-level programs. Synthesized routines run in supervisor mode, so they can perform privileged operations such as accessing protected buffer pages.

User-level programs enter supervisor mode using the `trap` instruction. This instruction provides a controlled — and the only — way for user-level programs to enter supervisor mode. The synthesized routine implementing the desired system service is accessed through a jump table in the protected area of the address space. The user program specifies an index into this table, ensuring the synthesized routines are always entered at the proper entry points. This protection mechanism is similar to Hydra's use of C-lists to prevent the forgery of capabilities [34].

Once in kernel mode, the synthesized code handling the requested service can begin to do its job. Further protection is unnecessary because, by design, the kernel code generator only creates code that touches data the application is allowed to touch. For example, were a file inaccessible, its `read` procedure would never have been generated. Just before returning control to the caller, the synthesized code reverts to the previous (user-level) mode.

There is still the question of invalidating the code when the operation it performs is no longer valid — for example, invalidating the `read` procedure after a files has been closed. Currently, this is done by setting the corresponding function pointers in the KPT to an invalid address, preventing further calls to the function. The function's reference counter is then decremented, and its memory freed when the count reaches zero.

3.4.3 Non-coherent Instruction Cache

A common assumption in the design of processors is that a program's instructions will not change as the program runs. For that reason, most processor's instruction caches are not coherent — writes to memory are not reflected in the cache. Runtime code generation violates this assumption, requiring that the instruction cache be flushed whenever code changes happen. Too much cache flushing reduces performance, both because programs execute slower when the needed instructions are not in cache and because flushing itself may be an expensive operation.

The performance of self-modifying code, like that found in executable data structures, suffers the most from an incoherent instruction cache. This is because the ratio of code modification to use tends to be high. Ideally, we would like to flush with cache-line granularity to avoid losing good entries. Some caches provide only an all-or-nothing flush. But even line-at-a-time granularity has its disadvantages: it needs machine registers to hold the parameters, registers that may not be available during interrupt service without incurring the cost of saving and restoring them. Unfortunately for Synthesis, most cases of self-modifying code actually occur inside interrupt service routines where small amounts of data (e.g., one character for a TTY line) must be processed with minimal overhead. Fortunately, in all important cases the cost has been reduced to zero through careful layout of the code in memory using knowledge of the 68030 cache architecture to cause the subsequent instruction fetch to replace the cache line that needs flushing. But that trick is neither general nor portable.

For the vast majority of code synthesis applications, an incoherent cache is not a big problem. The cost of flushing even a large cache contributes relatively little compared to the cost of allocating memory and creating the code. If code generation happens infrequently relative to the code's use, as is usually the case, the performance hit is small.

Besides the performance hit from a cold cache, cache flushing itself may be slow. On the 68030 processor, for example, the instruction to flush the cache is privileged. Although this causes no special problems for the Synthesis kernel, it does force user-level programs that modify code to make a system call to flush the cache. I do not see any protection-related reason why that instruction must be privileged; perhaps making it so simplified processor design.

3.5 Summary

This chapter showed: (1) that code synthesis allows important operating system functions such as buffering, context switching, interrupt handing, and system call dispatch to be implemented 1.4 to 2.4 times more efficiently than is possible using the best assembly-language implementation without code synthesis and 1.5 to 5 times better than well-written C code; (2) that code synthesis is also effective at the user-level, achieving an 80% improvement for basic operations such as `putchar`; and (3) that the anticipated size penalty does not, in fact, happen.

Before leaving this section, I want to call a moment's more attention to the interrupt handlers of Section 3.3.3. At first glance — and even on the second and third — the C-language code it looks to be as minimal as it can get. There does not seem to be any fat to cut. Table 3.4 has shown otherwise. The point is that sometimes, sources of overhead are hidden, not so easy to spot and optimize. They are a result of assumptions made and the programming language used, whether it be in the form of a common calling convention for procedures, or in conventions followed to simplify linking routines to interrupts. This section has shown that code synthesis is an important technique that enables general procedure interfacing while preserving — and often bettering — the efficiency of custom-crafted code.

The next chapter now shows how Synthesis is structured and how synergy between kernel code synthesis and good software engineering leads to a system that is general and easily expandable, but at the same time efficient.

4

Kernel Structure

All things should be made as simple as possible, but no simpler.
— Albert Einstein

4.1 Quajects

Quajects are the building blocks out of which all Synthesis kernel services are composed. The name is derived from the term "object" of Object-Oriented (O-O) systems, which they strongly resemble [32]. The similarity is strong, but the difference is significant. Like objects, quajects encapsulate data and provide a well-defined interface to access it. Unlike objects, quajects use a code-synthesis implementation to achieve high performance, but lack high-level language support and inheritance.

Kernel quajects can be broadly classified into four kinds: thread, memory, I/O, and device. Thread quajects encapsulate the unit of execution, memory quajects the unit of data storage, I/O quajects the unit of data movement, and device quajects the machine's interfaces to the outside world. Each kind of quaject is defined and implemented independently.

Basic quajects implement fundamental services that cannot be had through any combination of other quajects. Threads and queues are two examples of basic quajects;

Name	Purpose
Thread	Implements threads
Queue	Implements FIFO queues
Buffer	Data buffering
Dcache	Data caching (e.g., for disks)
FSmap	File to flat storage mapping
Clock	The system clock
CookTTYin	Keyboard input editor
CookTTYout	Output editor and format conversion
VT-100	Emulates DEC's VT100 terminal
Twindow	Text display window
Gwindow	Graphics (bit-mapped) display window
Probe	Measurements and statistics gathering
Sytab	Symbol table (associative mapping)

Table 4.1: List of Basic Quajects

Table 4.1 contains a list of the basic quajects in Synthesis. More complex kernel services are built out of the basic quajects by composition. For example, the Synthesis kernel has no pre-defined notion of a "process." But a UNIX-like process can be created by instantiating a thread quaject, a memory quaject, some I/O quajects, and interconnecting them in a particular way.

4.1.1 Quaject Interfaces

The interface to a quaject consists of *callentries*, *callbacks*, and *callouts*. A client uses the services of a quaject by calling a callentry. Normally a callentry invocation simply returns. Exceptional situations return along callbacks. Callouts are places in the quaject

```
+----------+-------------------+----------+
|   Qput   |                   |   Qget   |
+----------+       ---+--+--+  +----------+
|   Qfull  |   o  o  |  |  |   |  Qempty  |
+----------+       ---+--+--+  +----------+
| Qfull-1  |                   | Qempty+1 |
+----------+-------------------+----------+
```

Figure 4.1: Queue Quaject

where external calls to other quaject's callentries happen. Tables 4.2, 4.3, and 4.4 list the interfaces to the Synthesis basic kernel quajects.

Callentries are analogous to methods in O-O systems. The other two, callbacks and callouts, have no direct analogue in O-O systems. Conceptually, a callout is a function pointer that has been initialized to point to another quaject's callentry; callbacks point back to the invoker. Callouts are an important part of the interface because they specify what type of external call is needed, making it possible to dynamically link one of several different quaject's callentries to a particular callout, so long as the type matches. For example, the Synthesis `buffer` quaject has a `flush` callout which is invoked when the buffer is full. This enables the same buffer implementation to be used throughout the kernel simply be instantiating a `buffer` quaject and linking its `flush` callout to whatever downstream processing is appropriate for the instance.

The quaject interface is better illustrated using a simple quaject as an example — the FIFO queue, shown in Figure 4.1. The Synthesis kernel supports four different types of queues, to optimize for the varying synchronization needs of different combinations of single or multiple producers and consumers (synchronization is discussed in Chapter 5). All four types support the same abstract type [6], defined by two callentry references, `Q_put` and `Q_get`, which put and get elements of the queue. Both these callentry references return synchronously under the normal condition (successful insertion or deletion). Under other conditions, the queue returns through the callbacks.

The queue has four callbacks which are used to return queue-full and queue-empty conditions back to caller. `Q_empty` is invoked when a `Q_get` fails because the queue is empty. `Q_full` is invoked when a `Q_put` fails because the queue is full. `Q_empty+1` is called after a previous `Q_get` had failed and then an element was inserted. And `Q_full-1` is called after a previous `Q_put` had failed and then an element was deleted. The idea is: instead of returning a condition code for interpretation by the invoker, the queue quaject directly calls

Quaject	Interface	Name	Purpose
Queue	Callentry	Q_put	Insert element into queue
		Q_get	Remove element from queue
	Callback	Q_full	Notify that the queue is full
		Q_full-1	Notify that the queue is no longer full
		Q_empty	Notify that the queue is empty
		Q_empty-1	Notify that the queue is no longer empty
BufferOut	Callentry	put	Insert an element into the buffer
		write	Insert a string of elements into the buffer
		flush	Force buffer contents to output
	Callout	flush	Dump out the full buffer
BufferIn	Callentry	get	Get a single element from the buffer
		read	Get a string of elements from the buffer
	Callout	fill	Replenish the empty buffer
CookTTYin	Callentry	getchar	Read a processed character from the edit buffer
		read	Read a string of characters from the edit buffer
	Callout	raw_get	Get new characters from user's keyboard
		echo	Echo user's typed characters
CookTTYout	Callentry	putchar	Send a character out for processing
		write	Send a string of characters out for processing
	Callout	raw_write	Write out processed characters to display
VT100	Callentry	putchar	Write a character to the virtual VT-100 screen
		write	Write a string of characters
		update	Propagate changes to the virtual screen image
		refresh	Propagate the entier virtual screen image
FSmap	Callentry	aread	Asynchronous read from file
		awrite	Asynchronous write to file
	Callout	ca_read	Read from disk cache
		ca_write	Write to disk cache
Dcache	Callentry	read	Read from data cache
		write	Write to data cache
	Callout	bk_read	Read from backing store
		bk_write	Write to backing store
T_window	Callentry	write	Write a string of (character,attribute) pairs
G_window	Callentry	blit	Copy a rectangular array of pixels to window

Table 4.2: Interface to I/O Quajects

Quaject	Interface	Name	Purpose
Thread	Callentry	suspend	Suspends thread execution
		resume	Resumes thread execution
		stop	Prevents execution
		step	Executes one instruction then stops
		interrupt	Send a software interrupt
		signal	Send a software signal
		wait	Wait for an event
		notify	Notify that event has happened
	Callout	read[i]	Read from quaject i
		write[i]	Write to quaject i
		call[i][e]	Call callentry e in quaject i
Clock	Callentry	gettime	Get the time of day, in "ticks"
		getunits	Learn how many "ticks" there are in a second
		alarm	Set an alarm: call given procedure at given time
		cancel	Cancel an alarm
	Callout	call[i]	Call procedure i upon alarm expiration
Probe	Callentry	probe	Tell which procedure to measure
		show	Display statistics
Symtab	Callentry	lookup	Lookup a string; return its associated value
		add	Add entry to symbol table

Table 4.3: Interface to other Kernel Quajects

the appropriate handling routines supplied by the invoker, speeding execution by eliminating the interpretation of return status codes.

4.1.2 Creating and Destroying Quajects

Each class of quaject has `create` and `destroy` callentries that instantiate and destroy members of that class, including creating all their runtime-generated code. Creating a quaject involves allocating a single block of memory for its data and code, then initializing portions of that memory. With few exceptions, all of a quaject's runtime-generated code is created during this initialization. This generally involves copying the appropriate code template, determined by the type of quaject being created and the situation in which it is to be used, and then filling in the address fields in the instructions that reference quaject-specific data items. There are two exceptions to the rule. One is when the quaject implementation uses self-modifying code. The other occurs during the handling of callouts

Quaject	Interface	Name	Purpose
Serial_in	Callentry	enable	Enable input
		disable	Disable input
	Callout	putchar	Write received characher
Serial_out	Callentry	enable	Enable output
		disable	Disable output
	Callout	getchar	Obtain characher to send
Sound_CD	Callentry	enable	Enable input
		disable	Disable input
	Callout	put_sample	Store sound sample received from CD player
Sound_DA	Callentry	enable	Enable output
		disable	Disable output
	Callout	get_sample	Get new sound sample to send to A/D device
Framebuffer	Callentry	blit	Copy memory bitmap to framebuffer
		intr_ctl	Enable or disable interrupts
	Callout	Vsync	Vertical sync interrupt
		Hsync	Horizontal sync interrupt
Disk	Callentry	aread	Asyncronous read
		awrite	Asynchronous write
		format	Format the disk
		blk_size	Learn the disk's block size
	Callout	new_disk	(Floppy) disk has been changed

Table 4.4: Interface to Device Quajects

when linking one quaject to another. This is covered in the next section.

Kernel quajects are created whenever they are needed to build higher-level services. For example, opening an I/O pipe creates a queue; opening a text window creates three quajects: a window, a VT-100 terminal emulator, and a TTY-cooker. Which quajects get created and how they are interconnected is determined by the implementation of each service.

Quajects may also be created at the user level, simply by calling the class's `create` callentry from a user-level thread. The effect is identical to creating kernel quajects, except that user memory is allocated and filled, and the resulting quajects execute in user-mode, not kernel. The kernel does not concern itself with what happens to such user-level quajects. It merely offers creation and linkage services to applications that want to use them.

Quajects are destroyed when they are no longer needed. Invoking the `destroy` callentry signals that a particular thread no longer needs a quaject. The quaject itself is not actually destroyed until all references to it are severed. Reference counts are used. There is the possibility that circular references prevent destruction of otherwise useless quajects but this has not been a problem because quajects tend to be connected in cycle-free graphs. Destroying quajects does not immediately deallocate their memory. They are instead placed in the inactive list for their class. This speeds subsequent creation because much of the code-generation and initialization work had been already done.[1] As heap memory runs out, memory belonging to quajects on the inactive list is recycled.

4.1.3 Resolving References

The kernel resolves quaject callentry and callbacks references when linking quajects to build services. Conceptually, callouts and callback are function pointers that are initialized to point to other quaject's callentries when quajects are linked. For example, when attaching a queue to a source of data, the kernel fills the callouts of the data source with the addresses of the corresponding callentries in the queue and initializes the queue's callbacks with the addresses of the corresponding exception handlers in the data source. If the source of data is a thread, the address of the queue's `Q_put` callentry is stored in the thread's `write` callout, the queue's `Q_full` callback is linked to the thread's `suspend` callentry, and the queue's `Q_full-1` callback is linked to the thread's `resume` callentry. See Figure 4.2.

In the actual implementation, a callout is a "hole" in the quaject's memory where linkage-specific runtime generated code is placed. Generally, this code consists of zero or more instructions that save any machine registers used by both caller and callee quajects, followed by a `jsr` instruction to invoke the target callentry, followed by zero or more instructions to restore the previously saved registers. The callout's code might also perform a context switch if the called quaject is in a different address space. Or, in the case when the code comprising the called quaject's callentry is in the same address space and is smaller than the space set aside for the callout, the callentry is copied in its entirety into the callout. This is how the layer-collapsing, in-line expansion optimization of Section 3.2.2 works. A flag bit in each callentry tells if it uses self-modifying code, in which case, the copy does not happen.

Most linkage is done without referencing any symbol tables, but using information

[1] Performance measurements in this dissertation were carried out without using the inactive list, but creating fresh quajects as needed.

that is known at system generation time. Basically, the linking consists of blindly storing addresses in various places, being assured that they will always "land" in the correct place in the generated code. Similarly, no runtime type checking is required, as all such information has been resolved at system generation time.

Not all references must be specified or filled. Each quaject provides default values for its callout and callbacks that define what happens when a particular callout or callback is needed but not connected. The action can be as simple as printing an error message and aborting the operation or as complicated as dynamically creating the missing quaject, linking the reference, and continuing.

In addition, the kernel can also resolve references in response to execution traps that invoke the dynamic linker. Such references are represented by ASCII names. The name `Q_get`, for example, refers to the queue's callentry. A symbol-table quaject maps the string names into the actual addresses and displacements. For example, the `Q_get` callentry is represented in the symbol table as a displacement from the start of the queue quaject. Which quaject is being referenced is usually clear from context. For example, callentries are usually invoked using a register-plus-offset addressing mode; the register contains the address of the quaject in question. When not, an additional parameter disambiguates the reference.

4.1.4 Building Services

Higher-level kernel services are built by composing several basic quajects. I now show, by means of an example, how a data channel is put together. The example illustrates the usage of queues and reference resolution. It also shows how a data channel can support two kinds of interfaces, blocking and non-blocking, using the same quaject building block. The queue quaject used is of type `ByteQueue`.[2]

Figure 4.2 shows a producer thread using the `Q_put` callentry to store bytes in the queue. The ByteQueue's `Q_full` callback is linked to the thread's `suspend` callentry; the ByteQueue's `Q_full-1` callback is linked to the thread's `resume` callentry. As long as the queue is not full, calls to `Q_put` enqueue the data and return normally. When the queue becomes full, the queue invokes the `Q_full` callback, suspending the producer thread. When

[2]The actual implementation of Synthesis V.1 uses an optimized version of ByteQueue that has a string-oriented interface to reduce looping, but the semantics is the same.

Kind of Reference	User Thread		ByteQueue	ByteQueue	Device Driver	Hardware
callentry	**write**	$\Longrightarrow$	`Q_put`	`Q_get`	$\Longleftarrow$	*send-complete* interrupt
callback	**suspend**	$\Longleftarrow$	`Q_full`	`Q_empty`	$\Longrightarrow$	turn off *send-complete*
callback	**resume**	$\Longleftarrow$	`Q_full-1`	`Q_empty+1`	$\Longrightarrow$	turn on *send-complete*

Figure 4.2: Blocking `write`

Reference	Thread		ByteQueue
callentry	`write`	$\Longrightarrow$	`Q_put`
callback	`return to caller`	$\Longleftarrow$	`Q_full`
callback	if (more work) goto `Q_put`	$\Longleftarrow$	`Q_full-1`

Figure 4.3: Non-blocking write

the ByteQueue's reader removes a byte, the `Q_full-1` callback is invoked, awakening the producer thread. This implements the familiar synchronous interface to an I/O stream.

Contrast this with Figure 4.3, which shows a non-blocking interface to the same data channel implemented using the same queue quaject. Only the connections between ByteQueue and the thread change. The thread's `write` callout still connects to the queue's `Q_put` callentry. But the queue's callbacks no longer invoke procedures that suspend or resume the producer thread. Instead, they return control back to the producer thread, functioning, in effect, like interrupts that signal events — in this example, the filling and emptying of the queue. When the queue fills, the `Q_full` callback returns control back to the producer thread, freeing it to do other things without waiting for output to drain and without having written the bytes that did not fit. The thread knows the write is incomplete because control flow returns through the callback, not through `Q_put`. After output drains, `Q_full-1` is called, invoking an exception handler in the producer thread which checks whether there are remaining bytes to write, and if so, it goes back to `Q_put` to finish the job.

Ritchie's Stream I/O system has a similar flavor: it too provides a framework for attaching stages of processing to an I/O stream [27]. But stream-I/O's queueing structure is fixed, the implementation is based on messages, and the I/O is synchronous. Unlike Stream-I/O, quajects offer a finer level of control and expanded possibilities for connection. The previous example illustrates this by showing how the same queue quaject can be connected in different ways to provide either synchronous or asynchronous I/O. Furthermore, quajects extend the idea to include non-I/O services as well, such as threads.

4.1.5 Summary

In the implementation of Synthesis kernel, quajects provide encapsulation and make all inter-module dependencies explicit. Although quajects differ from objects in traditional O-O systems because of a procedural interface and run-time code generation implementation, the benefits of encapsulation and abstraction are preserved in a highly efficient implementation.

I have shown, using the data channel as an example, how quajects are composed to provide important services in the Synthesis kernel. That example also illustrates the main points of a quaject interface:

- Callentry references implement O-O–like methods and bypass interpretation in the invoked quaject.
- Callback references implement return codes and bypass interpretation in the invoker.
- The operation semantics are determined dynamically by the quaject interconnections, independent of the quaject's implementation.

This last point is fundamental in allowing a true orthogonal quaject implementation, for example, enabling a queue to be implemented without needing any knowledge of how threads work — not even how to suspend and resume them.

The next section shows how the quaject ideas fit together to provide user-level services.

4.2 Procedure-Based Kernel

Two fundamental ideas underlie how Synthesis is structured and how the services are invoked:

- *Every callentry is a real, distinct procedure.*
- *Services are invoked by calling these procedures.*

Quaject callentries are small procedures stored at known, fixed offsets from the base of the block of memory that holds the quaject's state. For simple callentries, the entire procedure is stored in the allocated space of the structure. Quajects such as buffers and queues have their callentries expanded in this manner, using all the runtime code-generation ideas discussed in Chapter 3. For more complex callentries, the procedures usually consist of some instance-specific code to handle the common execution paths, followed by code that loads the base pointer of the quaject's structure into a machine register and jumps to shared code implementing the rest of the callentry.

This representation differs from that of methods in object-oriented languages such as C++. In these languages, the object's structure contain pointers to generic methods for that class of object, not the methods themselves. The language system passes a pointer to the object's structure as an extra parameter to the procedure implementing each method. This makes it hard to use an object's method as a real function, one whose address can be passed to other functions without also passing and dealing with the extra parameter.

It is this difference that forms the basis of Synthesis quaject composition and extensible kernel service. Every callentry is a real procedure, each with a unique address and expecting no "extraneous" parameters. Each queue's `Q_put`, for example, takes exactly one parameter: the data to be enqueued. This property is fundamental for easy quaject composition: each quaject in a chain simply calls the next, without passing an arbitrarily long array of structure pointers downstream, one for each quaject.

4.2.1 Calling Kernel Procedures

The discussion until now assumes that the callentries reside in the same address space and execute at the same privilege level as their caller, so that direct procedure call is possible. But when user-level programs invoke kernel quajects, e.g., to read a file, the invocation crosses a protection boundary. A direct procedure call would not work because the kernel routine needs to run in supervisor mode.

In a conventional operating system, such as UNIX, application programs invoke the kernel by making *system calls*. But while system calls provide a controlled, protected way for a user-level program to invoke procedures in the kernel, they are limited in that they allow access to only a fixed set of procedures in the kernel. For Synthesis to be extensible, it needs an extensible kernel call mechanism; a mechanism that supports a protected, user-level interface to arbitrary kernel quajects.

The user-level interface is supplied with *stub* quajects. Stub quajects reside in the user address space and have the same callentries, with the same offsets, as the kernel quaject which they represent. Invoking a stub's callentry from user-level results in the corresponding kernel quaject's callentry being invoked and the results returned back.

This is implemented in the following way. The stub's callentries consist of tiny procedures that load a number into a machine register and then executes a `trap` instruction. The number identifies the desired kernel procedure. The `trap` switches the processor into kernel mode, where it executes the kernel-procedure dispatcher. The dispatcher uses the procedure number parameter to index a thread-specific table of kernel procedure addresses. Simple limit checks ensure the index is in range and that only the allowed procedures are called. If the checks pass, the dispatcher invokes the kernel procedure on the behalf of the user-level application.

There are many benefits to this design. One is that it extends the kernel quaject interface transparently to user-level, allowing kernel quajects to be composed with user-level quajects. Its callentries are real procedures: their addresses can be passed to other functions or stored in tables; they can be in-line substituted into other procedures and optimized using the code-synthesis techniques of Section 3.2 applied at the user level. Another advantage, which has already been discussed in Section 3.3.4, is that a very efficient implementation exists. The result is that the protection boundary becomes fluid; what is placed in the kernel and what is done at user-level can be chosen at will, not dictated by the design of the system. In short, all the advantages of kernel quajects have been extended out to user level.

4.2.2 Protection

Kernel procedure calls are protected because the user program can only specify indices into the kernel procedure table (KPT), so the kernel quajects are guaranteed to execute only from legitimate entry points, and because the index is checked before being used, only valid entries in the table can be accessed.

4.2.3 Dynamic Linking

Synthesis supports two flavors of dynamic linking: *load-link*, which resolves external references at program load time, before execution begins; and *run-link*, which resolves references at runtime as they are needed. Run-link has the advantage of allowing execution of programs with undefined references as long as the execution path does not cross them, simplifying debugging and testing of unfinished programs.

Dynamic linking does not prevent sharing or paging of executable code. It is possible to share dynamically-linked code because the runtime libraries always map to the same address in all address spaces. It is possible to page run-linked code and throw away infrequently used pages instead of writing them to backing store because the dynamic linker will re-link the references should the old page be needed again.

4.3 Threads of Execution

Synthesis threads are light-weight processes, implemented by the thread quaject. Each Synthesis thread (called simply "thread" from now on) executes in a context, defined by the thread table entry (TTE), which is the data part of the thread quaject holding the thread state and which contains:

- The register save area to hold the thread's machine registers when the thread is not executing.
- The kernel procedure table (KPT) — that table of callouts described in 4.2.1.
- The signal table, used to dispatch software signals.
- The address mapping tables for virtual memory.

- The vector table — the hardware-defined array of starting addresses of exception handlers. The hardware consults this table to dispatch the hardware-detected exceptions: hardware interrupts, error traps (like division by zero), memory faults, and software-traps (system calls).
- The context-switch-in and context-switch-out procedures comprising the executable data structure of the ready queue.

Of these, the last two are unusual. The context-switch-in and -out procedures were already discussed in Section 3.3.2, which explains how executable data structures are used to implement fast context switching. Giving each thread its own vector table also differs from usual practice, which makes the vector table a global structure, shared by all threads or processes. By having a separate vector table per thread, Synthesis saves the dispatching cost of thread-specific exceptions. Since most of the exceptions are thread specific, the savings is significant. Examples include all the error traps, such as division by zero, and the VM-related traps, such as translation fault.

4.3.1 Execution Modes

Threads can execute in one of two modes: supervisor mode and user mode. When a thread calls the kernel by issuing the `trap` instruction, it changes modes from user to supervisor. This view of things is in contrast to having a kernel server process run the kernel call on the behalf of the client thread. Each thread's memory mapping tables are set so that as the thread switches to supervisor mode, the kernel memory space becomes accessible in addition to the user space, in effect, "unioning" the kernel memory space with the user memory space. (This implies the set of addresses used must be disjoint.) Consequently, the kernel call may move data between the user memory and the kernel memory easily, without using special machine instructions, such as "`moves`" (move from/to alternate address space), that take longer to execute. Other memory spaces are outside the kernel space, inaccessible even from supervisor mode except through special instructions. Since no quaject's code contains those special instructions, Synthesis can easily enforce memory access restrictions for its kernel calls by using the normal user-level memory-access checks provided by the memory management unit. It first checks that no pointer is in the kernel portion of the address space (an easy check), and then proceeds to move the data. If an illegal access happens, or if a non-resident page is referenced, the thread will take a translation-fault exception, even from supervisor mode; the fault handler then reads in the referenced page from backing store if it was missing or prints the diagnostic message if the access is disallowed. (All this works because all quajects are reentrant, and since system calls are built out of quajects, all system calls are reentrant.)

Synthesis threads also provide a mechanism where routines executing in supervisor mode can make protected calls to user-mode procedures. It is mostly used to allow user-mode handling of exceptions that arise during supervisor execution, for example, someone typing "Control-C" while the thread is in the middle of a kernel call. It is also expected to find use in a future implementation of remote procedure call. The hard part in allowing user-level procedure calls is not in making the call, but arranging for a protected return from user-mode back to supervisor. This is done by pushing a special, exception-causing return address on the user stack. When the user procedure finishes and returns, the exception is raised, putting the thread back into supervisor mode.

4.3.2 Thread Operations

As a quaject, the thread supports several operations, defined by its callentries. They are: `suspend`, `resume`, `stop`, `step`, `interrupt`, `signal`, `setsignal`, `wait`, and `notify`. The last four overlap functionality with the first five, but are included for programmer convenience.[3]

`Suspend` and `resume` control thread execution, disabling or re-enabling it. They are often the targets of I/O quajects' callbacks, implementing blocking I/O. `Stop` and `step` support debuggers: `stop` prevents thread execution; `step` causes a stopped thread to execute a single machine instruction and then re-enter the stopped state. The difference between `stop` and `suspend` is that a suspended thread still executes in response to interrupts and signals while a stopped one does not. `Resume` continues thread execution from either the stopped or suspended state.

`Interrupt` causes a thread to call a specified procedure, as if a hardware interrupt had happened. It takes two parameters, an address and a mode, and it causes the thread to call the procedure at the specified address in either user or supervisor mode according to

[3] In the current implementation, the thread quaject is really a composition of two lower-level quajects, neither of them externally visible: a basic_thread quaject which supports the five fundamental operations listed; and a hi_thread quaject, which adds the higher-level operations. I'm debating whether I want to make the basic_thread quaject visible.

the mode parameter. Suspended threads can be interrupted: they will execute the interrupt procedure and then re-enter the suspended state.

`Signal` is like `interrupt`, but with a level of indirection for protection and isolation. It takes an integer parameter, the *signal number*, and indexes the thread's signal-table with it, obtaining the address and mode parameters that are then passed to `interrupt`. `Setsignal` associates signal numbers with addresses of interrupt procedures and execution modes. It takes three parameters: the signal number, an address, and a mode; and it fills the table slot corresponding to the signal number with the address and mode.

`Wait` waits for events to happen. It takes one parameter, an integer representing an event, and it suspends the thread until that event occurs. `Notify` informs the thread of the occurrence of events. It too takes one parameter, an integer representing an event, and it resumes the thread if it had been waiting for this event. The thread system does not concern itself with what is an event nor how the assignment of events to integers is made.

4.3.3 Scheduling

The Synthesis scheduling policy is round-robin with an adaptively adjusted CPU quantum per thread. Instead of priorities, Synthesis uses *fine-grain scheduling*, which assigns larger or smaller quanta to threads based on a "need to execute" criterion. A detailed explanation on fine-grain scheduling is postponed to Chapter 6. Here, I give only a brief informal summary.

A thread's "need to execute" is determined by the rate at which I/O data flows through its I/O channels compared to the rate at which which the running thread produces or consumes this I/O. Since CPU time consumed by the thread is an increasing function of the data flow, the faster the I/O rate the faster a thread needs to run. Therefore, the scheduling algorithm assigns a larger CPU quantum to the thread. This kind of scheduling must have a fine granularity since the CPU requirements for a given I/O rate and the I/O rate itself may change quickly, requiring the scheduling policy to adapt to the changes.

Effective CPU time received by a thread is determined by the quantum assigned to that thread divided by the sum of quanta assigned to all threads. Priorities can be simulated and preferential treatment can be given to certain threads in two ways: raise a thread's CPU quantum and reorder the ready queue as threads block and unblock. As an event unblocks a thread, its TTE is placed at the front of the ready queue, giving it immediate access to the CPU. This minimizes response time to events. Synthesis' low-overhead context switch allows quanta to be considerably shorter than that of other operating systems without incurring excessive overhead. Nevertheless, to minimize time spent context switching, CPU quanta are adjusted to be as large as possible while maintaining the fine granularity. A typical quantum is on the order of a few hundred microseconds.

4.4 Input and Output

In Synthesis, I/O includes all data flow among hardware devices and address spaces. Data move along logical channels called *data channels*, which connect sources of data with the destinations.

4.4.1 Producer/Consumer

The Synthesis implementation of the channel model I/O follows the well-known producer/consumer paradigm. Each data channel has a control flow that directs its data flow. Depending on the origin and scheduling of the control flow, a producer or consumer can be either *active* or *passive*. An active producer (or consumer) runs on a thread and calls functions submitting (or requesting) its output (or input). A thread performing `write`s is active. A passive producer (or consumer) does not run of its own; it sits passively, waiting for one of its I/O functions to be called, then using the thread that called the function to initiate the I/O. A TTY window is passive; characters appear on the window only in response to other thread's I/O. There are three cases of producer/consumer relationships, which we shall consider in turn.

The simplest is an active producer and a passive consumer, or vice-versa. This case, called active-passive, has a simple implementation. When there is only one producer and one consumer, a procedure call does the job. If there are multiple producers, we serialize their access. If there are multiple consumers, each consumer is called in turn.

The most common producer/consumer relationship has both an active producer and an active consumer. This case, called active-active, requires a queue to mediate the two. For a single producer and a single consumer, an ordinary queue suffices. For cases with multiple participants on either the producer or consumer side, we use one of the optimistically-synchronized concurrent-access queues described in section 5.2.2. Each queue may be synchronous (blocking) or asynchronous (using signals) depending on the situation.

The last case is a passive producer and a passive consumer. Here, we use a *pump* quaject that reads data from the producer and writes it to the consumer. This works for multiple passive producers and consumers as well.

4.4.2 Hardware Devices

Physical I/O devices are encapsulated in quajects called device servers. The device server interface generally mirrors the basic, "raw" interface of the physical device. Its I/O operations typically include asynchronous `read` and `write` of fixed-length data records and device-specific query and control functions. Each device server may have its own thread(s) or not. A polling I/O server runs continuously on its own thread. An interrupt-driven server blocks after initialization. The server without threads runs when its physical device generates an interrupt, invoking one of its callentries. Device servers are created at boot time, one server for each device, and persist until the system is shut down. Device servers can also be added as the system runs, but this must be done from a kernel thread — currently there is no protected, user-level way to do this.

Higher-level I/O streams are created by composing a device server with one or more *filter* quajects. There are three important functions that a filter quaject can perform: mapping one style of interface to another (e.g., asynchronous to synchronous), mapping one data format to another (e.g., EBCDIC to ASCII, byte-reversal), and editing data (e.g., backspacing). For example, the Synthesis equivalent of UNIX cooked `tty` interface is a filter that processes the output from the raw `tty` device server, buffers it, and performs editing as called for by the erase and kill control characters.

4.5 Virtual Memory

A full discussion of virtual memory will not be presented in this dissertation because all the details have not been completely worked out as of the time of this writing. Here, I merely assert that Synthesis does support virtual memory, but the model and interface are still in flux.

4.6 Summary

The positive experience in using quajects shows that a highly efficient implementation of an object-based system can be achieved. The main ingredients of such an implementation are:

- a procedural interface using callout and callentry references,
- explicit callback references for asynchronous return,
- run-time code generation and linking.

Chapter 7 backs this up with measurements. But now, we will look at issues involving multiprocessors.

5

Concurrency and Synchronization

The most exciting phrase to hear in science,
the one that heralds new discoveries, is not
"Eureka!" (I found it!) but "That's funny ..."
— Isaac Asimov

5.1 Synchronization in OS Kernels

In single-processor machines, the need for synchronization within an operating system arises because of hardware interrupts. They may happen in the middle of sensitive kernel data structure modifications, compromising their integrity if not properly handled. Even if the operating system supports multiprogramming, as most do, it is always an interrupt that causes the task switch, leading to inter-task synchronization problems.

In shared-memory multiprocessors, there is interference between processors accessing the shared memory, in addition to hardware interrupts. When different threads of control in the kernel need to execute in specific order (e.g., to protect the integrity of kernel data structures), they use synchronization mechanisms to ensure proper execution ordering. In this chapter, we discuss different ways to ensure synchronization in the kernel with emphasis on the Synthesis approach based on lock-free Synchronization.

5.1.1 Disabling Interrupts

In a single-processor kernel (including most flavors of UNIX), all types of synchronization problems can be solved cheaply by disabling the hardware interrupts. While interrupts are disabled the executing procedure is guaranteed to continue uninterrupted. Since disabling and enabling interrupts cost only one machine instruction each, it is orders of magnitude cheaper than other synchronization mechanisms such as semaphores, therefore its use is widespread. For example, 112 of the 653 procedures that make up version 3.3 of the Sony NEWS kernel (a BSD 4.3 derivative) disable interrupts.

But synchronization by disabling interrupts has its limitations. Interrupts cannot remain disabled for too long, otherwise frequent hardware interrupts such as a fast clock may be lost. This places a limit on the length of the execution path within critical regions protected by disabled interrupts. Furthermore, disabling interrupts is not always sufficient. In a shared-memory multiprocessor, data structures may be modified by different CPUs. Therefore, some explicit synchronization mechanism is needed.

5.1.2 Locking Synchronization Methods

Mutual exclusion protects a critical section by allowing only one process at a time to execute in it. The many styles of algorithms and solutions for mutual exclusion may be divided into two kinds: busy-waiting (usually implemented as spin-locks) and blocking (usually implemented as semaphores). Spin-locks sit in tight loops while waiting for the critical region to clear. Blocking semaphores (or monitors) explicitly send a waiting process to a queue. When the currently executing process exits the critical section, the next process is dequeued and allowed into the critical section.

The main problem with spin-locks is they waste CPU time while waiting. The justification in multiprocessors is that the process holding the lock is running and will soon clear the lock. This assumption may be false when multiple threads are mapped to the same physical processor, and results either in poor performance, or complicated scheduling to ensure the bad case does not happen. The main difficulty with blocking semaphores is the considerable overhead to maintain a waiting queue and to set and reset the semaphore. Furthermore, the waiting queue itself requires some kind of lock, resulting in a catch-22 situation that is resolved by disabling interrupts and spin-locks, Finally, having to choose between the two implementations leads to non-trivial decisions and algorithms for making it.

Besides the overhead in acquiring and releasing locks, locking methods suffer from three major disadvantages: contention, deadlock, and priority inversion. Contention occurs when many competing processes all want to access the same lock. Important global data structures are often points of contention. In Mach, for example, a single lock serializes access to the global run-queue [7]. This becomes a point of contention if several processors want to access the queue at the same time, as would occur when the scheduler clocks are synchronized. One way to reduce the lock contention in Mach relies on scheduling "hints" from the programmer. For example, hand-off hints may give control directly to the destination thread, bypassing the run queue. Although hints may decrease lock contention for specific cases, their use is difficult and their benefits uncertain.

Deadlock results when two or more processes both need locks held by the other. Typically, deadlocks are avoided by imposing a strict request order for the resources. This is a difficult solution because it requires system-wide knowledge to perform a local function; this goes against the modern programming philosophy of information-hiding.

Priority inversion occurs when when a low priority process in a critical section is preempted and causes other, higher priority processes to wait for that critical section. This can be particularly problematic for real-time systems where rapid response to urgent events is essential. There are sophisticated solutions for the priority inversion problem [8], but they contribute to make locks more costly and less appealing.

A final problem with locks is that they are *state*. In an environment that allows partial failure — such as parallel and distributed systems — a process can set a lock and then crash. All other processes needing that lock then hang indefinitely.

5.1.3 Lock-Free Synchronization Methods

It is possible to perform safe updates of shared data without using locks. Herlihy [14] introduced a general methodology to transform a sequential implementation of any data structure into a wait-free, concurrent one using the *Compare-&-Swap* primitive, which he shows is more powerful than test-and-set, the primitive usually used for locks. *Compare-&-Swap* takes three parameters: a memory location, a `compare_value`, and an `update_value`. If contents of the memory location is identical to the `compare_value`, the `update_value` is stored there and the operation succeeds; otherwise the memory location is left unchanged

```
int data_val;

AtomicUpdate(update_function)
{
retry:
    old_val = data_val;
    new_val = Update_Function(old_val);
    if(CAS(&data_val, old_val, new_val) == FAIL)
        goto retry;
    return new_val;
}
```

Figure 5.1: Atomic Update of Single-Word Data

```
CAS(mem_addr, compare_value, update_value)
{
    if(*mem_addr == compare_value) {
        *mem_addr = update_value;
        return SUCCEED;
    } else
        return FAIL;
}
```

Figure 5.2: Definition of Compare-and-Swap

and the operation fails.

Figure 5.1 shows how *Compare-&-Swap* is used to perform an arbitrary atomic update of single-word data in a lock-free manner. Initially, the current value of the word is read into a private variable, `old_val`. This value is passed to the update function which places its result in a private variable, `new_val`. *Compare-&-Swap* then checks if interference happened by testing whether the word still has value `old_val`. If it does, then the word is atomically updated with `new_val`. Otherwise, there was interference, so the operation is retried. For reference, Figures 5.2 and 5.3 shows the definition of `CAS`, the *Compare-&-Swap* function, and of `CAS2`, the two-word *Compare-&-Swap* function, which is used later.

Updating data of arbitrary-length using *Compare-&-Swap* is harder. Herlihy's general method works like this: each data structure has a "root" pointer, which points to the current version of the data structure. An update is performed by allocating new memory and copying the old data structure into the new memory, making the changes, using *Compare-&-Swap* to swing the root pointer to the new structure, and deallocating the old.

```
CAS2(mem_addr1, mem_addr2, compare1, compare2, update1, update2)
{
    if(*mem_addr1 == compare1 && *mem_addr2 == compare2) {
        *mem_addr1 = update1;
        *mem_addr2 = update2;
        return SUCCEED;
    } else
        return FAIL;
}
```

Figure 5.3: Definition of Double-Word Compare-and-Swap

He provides methods of partitioning large data structures so that not all of it needs to be copied, but in general, his methods are expensive.

Herlihy defines an implementation of a concurrent data structure to be *wait-free* if it guarantees that *each* process modifying the data structure will complete the operation in a finite number of steps. He defines an implementation to be *non-blocking* if it guarantees that *some* process will complete an operation in a finite number of steps. Both prevent deadlock. Wait-free also prevents starvation. In this paper, we use the term *lock-free* as synonymous with non-blocking. We have chosen to use lock-free synchronization instead of wait-free because the cost of wait-free is much higher and the chances of starvation in an OS kernel is low — I was unable to construct a test case where that would happen.

Even with the weaker goal of non-blocking, Herlihy's data structures are expensive, even when there is no interference. For example, updating a limited-depth stack is implemented by copying the entire stack to a newly allocated block of memory, making the changes on the new version, and switching the pointer to the stack with a *Compare-&-Swap*. This cost is much too high, and we want to find ways to reduce it.

5.1.4 Synthesis Approach

The Synthesis approach to synchronization is motivated by a desire to do each job using the minimum resources. The previous sections outlined the merits and problems of various synchronization methods. Here are the ideas that guided our search for a synchronization primitive for Synthesis:

- We wanted a synchronization method that avoids the problem of priority inversion so as to simplify support of real-time signal processing.

- We did not want to disable interrupts because we wanted to support I/O devices that interrupt at a very high rate, such as the Sound-IO devices. Also, disabling interrupts by itself does not work for multiprocessors.

- We wanted a synchronization method that does not have the problem of deadlock. The reason is that we wanted as much flexibility as possible to examine and modify running kernel threads. We wanted to be able to suspend threads to examine their state without affecting the rest of the system.

Given these desires, lock-free synchronization is the method of choice. Lock-free synchronization does not have the problems of priority inversion and deadlock. I also feel it leads to more robust code because there can never be the problem of a process getting stuck and hanging while holding a lock.

Unfortunately, Herlihy's general wait-free methods are too expensive. So instead of trying to implement arbitrary data structures lock-free, we take a different tack: We ask "what data structures can be implemented lock-free, *efficiently*?" We then build the kernel out of these structures. This differs from the usual way: typically, implementors select a synchronization method that works generally, such as semaphores, then use that everywhere. We want to use the cheapest method for each job. We rely on the quaject structuring of the kernel and on code synthesis to create special synchronization for each need.

The job is made easier because the Motorola 68030 processor supports a two-word *Compare-&-Swap* operation. It is similar in operation to the one-word *Compare-&-Swap*, except that two words are compared, and if they both match, two updates are performed. Two-word *Compare-&-Swap* lets us efficiently implement many basic data structures such as stacks, queues, and linked lists because we can atomically update both a pointer and the location being pointed to in one step. In contrast, Herlihy's algorithms, using single-word *Compare-&-Swap*, must resort to copying.

The first step is to see if synchronization is necessary at all. Many times the need for synchronization can be avoided through *code isolation*, where only specialized code that is known to be single-threaded handles the manipulation of data. An example of code isolation is in the run-queue. Typically a run-queue is protected by semaphores or spin-locks, such as in the UNIX and Mach implementations [7]. In Synthesis, only code residing in each element can change it, so we separate the run-queue traversal, which is done lock-free, safely and concurrently, from the queue element update, which is done locally by its associated thread. Another example occurs in a single-producer, single-consumer queue. As long as the queue is neither full nor empty, the producer and consumer work on different parts of it and need not synchronize.

Once it has been determined that synchronization is unavoidable, the next step is to try to encode the shared information into one or two machine words. If that succeeds, then we can use *Compare-&-Swap* on the one or two words directly. Counters, accumulators, and state-flags all fall in this category. If the shared data is larger than two words, then we try to encapsulate it in one of the lock-free quajects we have designed, explained in the next section: LIFO stacks, FIFO queues, and general linked lists. If that does not work, we try to partition the work into two pieces, one part that can be done lock-free, such as enqueueing the work and setting a "work-to-be-done" flag, and another part that can be postponed and done at a time when it is known interference will not happen (e.g., code isolation). Suspending of threads, which is discussed in Section 5.3.2, follows this idea — a thread is marked suspended; the actual removal of the thread from the run-queue occurs when the thread is next scheduled.

When all else fails, it is possible to create a separate thread that acts as a server to serialize the operations. Communication with the server happens using lock-free queues to assure consistency. This method is used to update complex data structures, such as those in the VT-100 terminal emulator. Empirically, I have found that after all the other causes of synchronization have been eliminated or simplified as discussed above, the complex data structures that remain are rarely updated concurrently. In these cases, we can optimize, dispensing with the server thread except when interference occurs. Invoking an operation sets a "busy" flag and then proceeds with the operation, using the caller's thread to do the work. If a second thread now attempts to invoke an operation on the same data, it sees the busy-flag set, and instead enqueues the work. When the first thread finishes the operation, it sees a non-empty work queue, and spawns a server thread to process the remaining work. This server thread persists as long as there is work in the queue. When the last request has been processed, the server dies.

In addition to using only lock-free objects and optimistic critical sections, we also try to minimize the length of each critical section to decrease the probability of retries. The longer a process spends in the critical section, the greater the chance of outside interference forcing a retry. Even a small decrease in length can have a profound effect on retries.

```
Insert(elem)
{
retry:
    old_first = list_head;
    *elem = old_first
    if(CAS(&list_head, old_head, elem) == FAIL)
        goto retry;
}

Delete()
{
retry:
    old_first = list_head;
    if(old_first == NULL)
        return NULL;
    second = *old_first;
    if(CAS2(&list_head, old_first, old_head, second, second, 0) == FAIL)
        goto retry;
    return old_first;
}
```

Figure 5.4: Insert and Delete at Head of Singly-Linked List

Sometimes a critical section can be divided into two shorter ones by finding a consistent intermediate state. Shifting some code between readers and writers will sometimes produce a consistent intermediate state.

5.2 Lock-Free Quajects

The Synthesis kernel is composed of *quajects*, chunks of code with data structures. Some quajects represent OS abstractions, such as threads, memory segments, and I/O devices described earlier in Chapter 4. Other quajects are instances of abstract data types such as stacks, queues, and linked lists, implemented in a concurrent, lock-free manner, This section describes them.

5.2.1 Simple Linked Lists

Figure 5.4 shows a lock-free implementation of insert and delete at the head of a singly-linked list. Insert reads the address of the list's first element into a private variable (`old_first`), copies it into the link field of the new element to be inserted, and then uses *Compare-&-Swap* to atomically update the list's head pointer if it had not been changed since the initial read. Insert and delete to the end of the list can be carried out in a similar manner, by maintaining a list-tail pointer. This method is similar to that suggested in the

```
Push(elem)
{
retry:
    old_SP = SP;
    new_SP = old_SP - 1;
    old_val = *new_SP;
    if(CAS2(&SP, new_SP, old_SP, old_val, new_SP, elem) == FAIL)
        goto retry;
}

Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if(CAS2(&SP, old_SP, old_SP, elem, new_SP, elem) == FAIL)
        goto retry;
    return elem;
}
```

Figure 5.5: Stack Push and Pop

68030 processor handbook [21].

5.2.2 Stacks and Queues

One can implement a stack using insert and delete to the head of a linked list, using the method of the previous section. But this requires node allocation and deallocation, which adds overhead. So I found a way of doing an array-based implementation of a stack using two-word *Compare-&-Swap*. This implementation also has the advantage that it works on the hardware-defined processor stacks, which is important for delivering signals to threads. I believe this is a new result, though not a “big” one.

Figure 5.5 shows a lock-free implementation of a stack. Pop is implemented in almost the same way as a counter increment. The current value of the stack pointer is read into a private variable, which is de-referenced to get the top item on the stack and incremented past that item. The stack pointer is then updated using *Compare-&-Swap* to test for interfering accesses and retry when they happen.

Push is more complicated because it must atomically update two things: the stack pointer and the top item on the stack. This needs a two-word *Compare-&-Swap*. The current stack pointer is read into a private variable and decremented, placing the result into another private variable. This decremented stack pointer contains the memory address where the new item will be put. But first, the data at this address is read into a third

```
Put(elem)
{
retry:
    old_head = Q_head;
    new_head = old_head + 1;
    if(new_head >= Q_end)
        new_head = Q_begin;
    if(new_head == Q_tail)
        return FULL;
    old_elem = *new_head;
    if(CAS2(&Q_head, new_head, old_head, old_elem, new_head, elem) == FAIL)
        goto retry;
}

Get()
{
retry:
    old_tail = Q_tail;
    if(old_tail == Q_head)
        return EMPTY;
    elem = *old_tail;
    new_tail = old_tail + 1;
    if(new_tail >= Q_end)
        new_tail = Q_begin;
    if(CAS2(&Q_tail, old_tail, old_tail, elem, new_tail, elem) == FAIL)
        goto retry;
    return elem;
}
```

Figure 5.6: Queue Put and Get

private variable, then the new item stored there and the stack pointer updated using a a two-word *Compare-&-Swap*. (The data must be read to give *Compare-&-Swap*-2 two comparison values. *Compare-&-Swap*-2 always performs two tests; sometimes one of them is undesirable.)

Figure 5.6 shows a lock-free implementation of a circular queue. It is very similar to the stack implementation, and will not be discussed further.

5.2.3 General Linked Lists

The "simple" linked lists described earlier allow operations only on the head and tail of the list. General linked lists also allow operations on interior nodes.

Deleting nodes at the head of a list is easy. Deleting an interior node of the list is much harder because the permanence of its neighbors is not guaranteed. Linked list traversal in the presence of deletes is hard for a similar reason — a node may be deleted and deallocated while another thread is traversing it. If a deleted node is then reallocated and reused for some other purpose, its new pointer values may cause invalid memory references by the

```
VisitNextNode(current)
{
    nextp = & current->next;       // Point to current node's next-node field
retry:
    next_node = *nextp;            // Point to the next node
    if(next_node != NULL) {        // If node exists...
        refp = & next_node->refcnt; // Point to next node's ref. count field
        old_ref = *refp;           // Get value of next node's ref. count
        new_ref = old_ref + 1;     // And increment
        if(CAS2(nextp, refp, next_node, old_ref, next_node, new_ref) == FAIL)
            goto retry;
    }
    return next_node;
}

ReleaseNode(current)
{
    refp = & current->refcnt;      // Point to current node's ref. count field
retry:
    old_ref = *refp;               // Get value of current node's ref. count
    new_ref = old_ref - 1;         // ... Decrement
    if(CAS(old_ref, new_ref, refp) == FAIL)
        goto retry;
    if(new_ref == 0) {
        Deallocate(current);
        return NULL;
    } else {
        return current;
    }
}
```

Figure 5.7: Linked List Traversal

other thread still traversing it.

Herlihy's solution [14] uses reference counts. The idea is to keep deleted nodes "safe." A deleted node is safe if its pointers continue to be valid; i.e., pointing to nodes that eventually take it back to the main list where the *Compare-&-Swap* will detect the change and retry the operation. Nodes that have been deleted but not deallocated are safe.

Figure 5.7 shows an implementation of Herlihy's idea, simplified by using a two-word *Compare-&-Swap*. Visiting a node loads the pointer and increments the reference count. Leaving a node decrements the reference count. A deleted node is not actually freed until the reference count reaches zero. Deleting a node still requires the permanence of its neighbors. We do this in two steps: (1) mark the nodes to be deleted and leave them in the list; (2) if the previous node is not marked for deletion, sit on it and delete the original node marked for deletion. Since step 2 may require going back through the list an arbitrary number of nodes, usually we do step 2 the next time we traverse the list to avoid the overhead of traversing the list just for deletion.

Operation	Non Sync	Locked	Lock-free$_{noretry}$	Lock-free$_{oneretry}$
null procedure call in C	1.4	—	—	—
Increment counter	0.3	2.4	1.3	2.3
Linked-list Insert	0.6	2.7	1.4	2.3
Linked-list Delete	1.0	3.2	2.1	4.1
Circular-Queue Insert	2.0	4.2	3.3	6.0
Circular-Queue Delete	2.1	4.3	3.3	6.0
Stack Push	0.7	2.8	2.0	3.5
Stack Pop	0.7	2.8	2.0	3.5
get_semaphore Sony NEWS, UNIX	8.7	—	—	—

Times in microseconds

68030 CPU, 25MHz, 1-wait-state main memory, cold cache

Table 5.1: Comparison of Different Synchronization Methods

Optimizations are possible if we can eliminate some sources of interference. In the Synthesis run queue, for example, there is only one thread visiting a TTE at any time. So we simplify the implementation to use a binary marker instead of counters. We set the mark when we enter the node using a two-word *Compare-&-Swap*. This is easier than incrementing a counter because we don't have to read the mark beforehand – it must be zero to allow entrance. Non-zero means that node is being visited by some other processor, so we skip to the next one and repeat the test.

5.2.4 Lock-Free Synchronization Overhead

Table 5.1 shows the overhead measured for the lock-free objects described in this section, and compares it with the overhead of two other implementations: one using locking and one that is not synchronized. The column labeled "Non Sync" shows the time taken to execute the operation without synchronization. The column labeled "Locked" shows the time taken by a locking-based implementation of the operation without interference. The column labeled "Lock-free$_{noretry}$" shows the time taken by the lock-free implementation when there is no interference. The column labeled "Lock-free$_{oneretry}$" shows the time taken when interference causes the first attempt to retry, with success on the second attempt.[1] For reference, the first line of the table gives the cost of a null procedure call in the C language, and the last line gives the cost of a `get_semaphore` operation in Sony's RTX kernel. (The RTX kernel runs in the I/O processor of Sony's dual-processor workstation, and is meant to be a light-weight kernel.)

The numbers shown are for in-line assembly-code implementation and assume a pointer to the relevant data structure already in a machine register. The lock-free code measured is the same as that produced by the Synthesis kernel code generator. The non-synchronized code is the best I've been able to do writing assembler by hand. The lock-based code is the same as the non-synchronized, but preceded by some code that disables interrupts and then obtains a spinlock, and followed by code to clear the spinlock and re-enable interrupts. The reasoning behind disabling interrupts is to make sure that the thread does not get preempted in its critical section, guaranteeing that the lock is cleared quickly. This represents good use of spin-locks, since any contention quickly passes.

Besides avoiding the problems of locking, the table shows that the lock-free implementation is actually *faster* than the lock-based one, even in the case of no interference. In fact, the performance of lock-free in the presence of interference is comparable to locking without interference.

Let us study the reason for this surprising result. Figures 5.8 and 5.9 show the actual code that was measured for the linked-list delete operation. Figure 5.8 shows the lock-free code, while Figure 5.9 shows the locking-based code. The lock-free code closely follows the principles of operation described earlier. The lock-based code begins by disabling processor interrupts to guarantee that the process will not be preempted. It then obtains the spin-lock; interference at this point is limited to that from other CPUs in a multiprocessor, and the lock should clear quickly. The linked-list delete is then performed, followed by clearing

[1]This case is produced by generating an interfering memory reference between the initial read and the *Compare-&-Swap*. The Quamachine's memory controller, implemented using programmable gate arrays, lets us do things like this. Otherwise the interference would be very difficult to produce and measure.

```
retry:  move.l  (head),d1              // Get head node into reg. d1
        move.l  d1,a0                  // ... copy to register 'a0'
        beq     empty                  // ... jump if list empty
        lea     (a0,next),a2           // Get address of head node's next ptr.
        move.l  (a2),d2                // Get 2nd node in list into reg. d2
        cas2.l  d1:d2,d2:d2,(head):(a2) // Update head if both nodes still same
        bne     retry                  //  ... go try again if unsucessful
```

Figure 5.8: Lock-Free Delete from Head of Singly-Linked List

```
        move.w  %sr,d0          // Save CPU status reg. in reg. d0
        or.w    #0x0700,%sr     // Disable interrupts.
spin:   tas     lock            // Obtain lock
        bne     spin            // ... busy wait
        move.l  (head),a0       // Get head node into reg. a0
        tst.l   a0 // Is there a node?
        bne     go              // ... yes, jump
        clr.l   lock            // ... no: Clear the lock
        move.w  d0,%sr          //        Reenable interrupts
        bra     empty           //        Go to 'empty' callback
go:     move.l  (a0,next),(head)// Make 'head' point to 2nd node
        clr.l   lock            // Clear the lock
        move.w  d0,%sr          // Reenable interrupts
```

Figure 5.9: Locked Delete from Head of Singly-Linked List

the lock and reenabling interrupts. (Don't be fooled by its longer length: part of the code is executed only when the list is empty.)

Accounting for the costs, the actual process of deleting the element from the list takes almost the same time for both versions, with the the lock-free code taking a few cycles longer. (This is because the *Compare-&-Swap* instruction requires its compare-operands to be in D registers while indirection is best done through the A registers, whereas the lock-based code can use whatever registers are most convenient.) The cost advantage of lock-free comes from the much higher cost of obtaining and clearing the lock compared to the cost of *Compare-&-Swap*. The two-word *Compare-&-Swap* instruction takes 26 machine cycles to execute on the 68030 processor. By comparison, obtaining and then clearing the lock costs 46 cycles, with the following breakdown: 4 to save the CPU status register; 14 to disable interrupts; 12 to obtain the lock; 6 to clear the lock following the operation; and 10 to reenable interrupts. (For reference, fetching from memory costs 6 cycles and a single word *Compare-&-Swap* takes 13 cycles.)

Some people argue that one should not disable interrupts when obtaining a lock. They believe it is better to waste time spin-waiting in the rare occasion that the process holding the lock is preempted, than to pay the disable/enable costs each time.[2] I disagree. I believe that in operating systems, it is better that an operation perhaps cost a little more, than to have it be a little faster but occasionally exhibit very high cost. Repeatability and low variance are often times as important if not more important than low average cost. Furthermore, allowing interrupts in a critical section opens the possibility that a process which has been interrupted might not return to release the lock. The user may have typed Control-C, for example, terminating the program. Recovering from this situation or preventing it from happening requires tests and more code which adds to the cost — if not to the lock itself, then somewhere else in the system.

5.3 Threads

This section describes how thread operations can be implemented so they are lock-free and gives timing figures showing their cost.

5.3.1 Scheduling and Dispatching

Section 3.3.2 described how thread scheduling and dispatching works using an executable data structure to speed context switching. Each thread is described by a *thread table entry* (TTE). The TTE contains the thread-specific procedures implementing the dispatcher and scheduler, the thread context save area, and other thread-specific data. The dispatcher is divided into two halves: the switch-out routine, which is executed from the currently running thread's TTE and which saves the thread's context; and the switch-in routine, which is executed from the new thread's TTE and loads new thread's context and installs its switch-out routine into the quantum clock interrupt handler.

In the current version of Synthesis, the TTEs are organized into multiple levels of run-queues for scheduling and dispatching. The idea is that some threads need more frequent attention from the CPU than others, and we want to accommodate this while maintaining an overall round-robin-like policy that is easy to schedule cheaply. The policy works like

[2] for the data structures of interest here, not disabling interrupts makes the cost of locking when no process is preempted very nearly identical to the cost of lock-free.

this: on every second context switch, a thread from level 0 is scheduled, in round-robin fashion. On every fourth context switch, a thread from level 1 is scheduled, also in round-robin fashion. On every eighth context switch, a thread from level 2 is scheduled. And so on, for 8 levels. Each level gets half the attention of the previous level. If there are no threads at a particular level, that level's quanta is distributed among the rest of the levels.

A global counter and a lookup table tells the dispatcher which level's queue is next. The lookup table contains the scheduling policy described above — a 0 every other entry, 1 every fourth entry, 2 every eighth entry, like this: $(0, 1, 0, 2, 0, 1, 0, 3, 0, 1, \cdots)$. Using the counter to follow the priority table, the kernel dispatches a thread from level 0 at every second context-switch, from level 1 at every fourth context-switch, level 2 at every eighth, and so on.

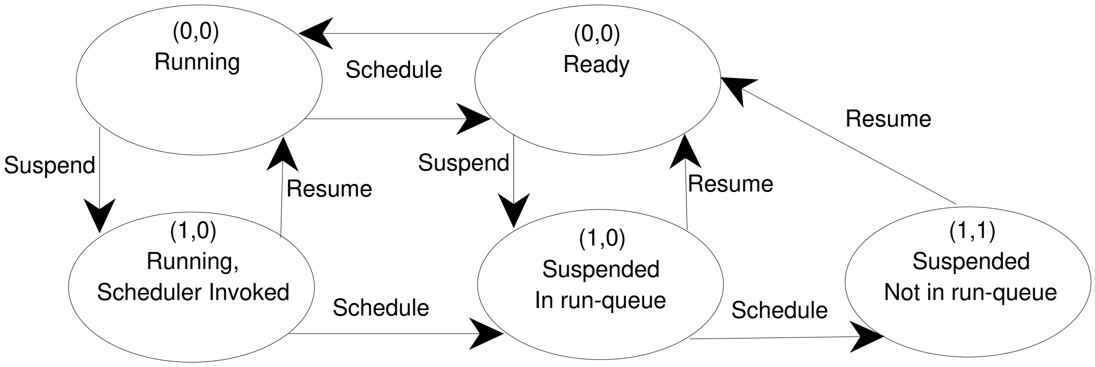
When multiple CPUs attempt to dispatch threads from the run-queues, each active dispatcher (switch-out routine) acquires a new TTE by marking it using *Compare-&-Swap*. If successful, the dispatcher branches to the switch-in routine in the marked TTE. Otherwise, some other dispatcher has just acquired the attempted TTE, so this dispatcher moves on to try to mark the next TTE. The marks prevent other dispatchers from accessing a particular TTE, but not from accessing the rest of the run queues.

5.3.2 Thread Operations

We now explain how the other thread operations are made lock-free. The general strategy is the same. First, mark the intended operation on the TTE. Second, perform the operation. Third, check whether the situation has changed. If negative, the operation is done. If positive, retry the operation. An important observation is that all state transitions and markings are done atomically through *Compare-&-Swap*.

Figure 5.10 shows the thread state-transition diagram for the suspend and resume operations.

Suspend: The thread-suspend procedure sets the STOPME flag in the target thread's TTE indicating that it is to be stopped. If the target thread is currently running on a different CPU, a hardware interrupt is sent to that CPU by writing to a special I/O register, forcing a context-switch. We optimize the case when a thread is suspending itself by directly calling the scheduler instead. Thread-suspend does not actually remove the thread from the run-queue.



Numbers in parenthesis: (STOPME,STOPPED)

Figure 5.10: Thread State Transition Diagram

When a scheduler encounters a thread with the STOPME flag set, it removes its TTE from the run-queue and sets the STOPPED flag to indicate that the thread has been stopped. This is done using the two-word compare-and-swap instruction to synchronize with other CPU's schedulers that may be operating on the adjacent queue elements. The mark on the TTE guarantees that only one CPU is visiting each TTE at any given time. This also makes the delete operation safe.

Resume: First, the STOPME and STOPPED flags are read and the STOPME flag is cleared to indicate that the thread is ready to run. If the previously-read STOPPED flag indicates that the thread had not yet been removed from the run-queue, we are done. Otherwise, we remove the TTE and insert the thread directly into the run queue. The main problem we have to avoid is the case of a neighboring TTE being deleted due to the thread being killed. To solve that problem, when a thread is killed, we mark its TTE as "killed," but do not remove it from the run-queue immediately. When a dispatcher realizes the next TTE is marked "killed" during a context switch, it can safely remove it.

Signal: Thread-signal is synchronized in a way that is similar to thread-resume. Each thread's TTE has a stack for pending signals which contains addresses of signal-handler procedures. Thread-signal uses a two-word *Compare-&-Swap* to push a new procedure address onto this stack. It then sets a signal-pending flag, which the scheduler tests. The scheduler removes procedures from the pending-signal stack, one at a time, and constructs

Thread Operation	Time (μs)
Create$_{shared\ vector\ table}$	19.2
Create$_{separate\ vector\ table}$	97
Destroy	2.2 + 6.1 in dispatcher
Suspend	2.2 + 3.7 in dispatcher
Resume	3.0 if in Q; 6.6 not in Q
Signal	4.6 + 4.4 in scheduler
Step (no FP, no VM switch)	25

Table 5.2: Thread operations

procedure call frames on the thread's runtime stack to simulate the thread having called that procedure.

Step: Thread-step is intended for instruction-at-a-time debugging; concurrent calls defeats its purpose. So we do not give any particular meaning to concurrent calls of this function except to preserve the consistency of the kernel. In the current implementation, all calls after the first fail. We implement this using an advisory lock.

5.3.3 Cost of Thread Operations

Table 5.2 shows the time taken to perform the various thread operations implemented using the lock-free synchronization methods of this chapter. They were measured on the Sony NEWS 1860 machine, a dual 68030 each at 25 MHz, with no interference from the other processor.

Thread suspend, destroy, and signal have been split into two parts: the part done by the requester and the part done by the dispatcher. The time for these are given in the form "$X + Y$," the first number is the time taken by the requester, the second number is the time taken by the dispatcher. Thread resume has two cases, the case where the thread had been stopped but the scheduler had not removed it from the run queue yet, shown by the first number, and the case where it was removed from the run queue and must be re-inserted, shown by the second number.

Type of context switch	Synthesis V.1
Integer registers only	14
Floating-point	56
Integer, change address space	$20 + 1.6 * TLB\_fill$
Floating-point, change address space	$60 + 1.6 * TLB\_fill$

Table 5.3: Overhead of Thread Scheduling and Context Switch

Thread create has been made significantly faster with a copy-on-write optimization. Recall from Section 4.3 that each thread has a separate vector table. The vector table contains pointers to synthesized routines that handle the various system calls and hardware interrupts. These include the 16 system-call trap vectors, 21 program exception vectors, 19 vectors for hardware failure detection, and, depending on the hardware configuration, from 8 to 192 interrupt vectors. This represents a large amount of state information that had to be initialized — 1024 bytes.

Newly-created threads point their vector table to the vector table of their creator and defer the creation of their own until they need to change the vector table. There are only two operations that change a thread's vector table: opening and closing quajects. If a quaject is not to be shared, `open` and `close` test if the TTE is being shared, and if so they first make a copy of the TTE and then modify the new copy. Alternatively, several threads may share the changes in the common vector table. For example, threads can now perform system calls such as `open file` and naturally share the resulting file access procedures with the other threads using the same vector table.

Table 5.3 shows the cost of context switching and scheduling. Context-switch is somewhat slower than shown earlier, in Table 3.3, because now we schedule from multiple run queues, and because there is synchronization that was not necessary in the single-CPU version discussed in Section 3.3.2. When changing address spaces, loading the memory management unit's translation table pointer and flushing the translation cache increases the context switch time. Extra time is then used up to fill the translation cache. This is the "$+1.6 * TLB\_fill$" time. Depending on the thread's locality of reference, this can be as low as 4.5 microseconds for 3 pages (code, global data, and stack) to as high as 33 microseconds to fill the entire TLB cache.

5.4 Summary

We have used only lock-free synchronization techniques in the implementation of Synthesis multiprocessor kernel on a dual-68030 Sony NEWS workstation. This is in contrast to other implementations of multiprocessor kernels that use locking. Lock-based synchronization methods such as disabling interrupts, spin-locking, and waiting semaphores have many problems. Semaphores carry high management overhead and spin-locks may waste significant amount of CPU. (A typical argument for spin-locks is that the processor would be idle otherwise. This may not apply for synchronization inside the kernel.) A completely lock-free implementation of a multiprocessor kernel demonstrates that synchronization overhead can be reduced, concurrency increased, deadlock avoided, and priority inversion eliminated.

This completely lock-free implementation is achieved with a careful kernel design using the following five-point plan as a guide:

- Avoid synchronization whenever possible.
- Encode shared data into one or two machine words.
- Express the operation in terms of one or more fast lock-free data structure operations.
- Partition the work into two parts: a part that can be done lock-free, and a part that can be postponed to a time when there can be no interference.
- Use a server thread to serialize the operation. Communications with the server happens using concurrent, lock-free queues.

First we reduced the kind of data structures used in the kernel to a few simple abstract data types such as LIFO stacks, FIFO queues, and linked lists. Then, we restricted the uses of these abstract data types to a small number of safe interactions. Finally we implemented efficient special-purpose instances of these abstract data types using single-word and double-word *Compare-&-Swap*. The kernel is fully functional, supporting threads, virtual memory, and I/O devices such as window systems and file systems. The measured numbers show the very high efficiency of the implementation, competitive with user-level thread management systems.

Two lessons were learned from this experience. The first is that a lock-free implementation is a viable and desirable alternative to the development of shared-memory multiprocessor kernels. The usual strategy — to evolve a single-processor kernel into a multiprocessor kernel by surrounding critical sections with locks — carries some performance penalty and potentially limits the system concurrency. The second is that single and double word *Compare-&-Swap* are important for lock-free shared-memory multiprocessor kernels. Architectures that do not support these instructions may suffer performance penalties if operating system implementors are forced to use locks. Other synchronization instructions, such as the Load-Linked/Store-Conditional found on the MIPS processor, may also yield efficient lock-free implementations.

6

Fine-Grain Scheduling

There's no sense in being precise when you don't even know what you're talking about.
— John von Neumann

6.1 Scheduling Policies and Mechanisms

There are two parts to scheduling: the policy and the mechanism. The policy determines when a job should run and for how long. The mechanism implements the policy.

Traditional scheduling mechanisms have high overhead that discourages frequent scheduler decision making. Consequently, most scheduling policies try to minimize their actions. We observe that high scheduling and dispatching overhead is a result of implementation, not an inherent property of all scheduling mechanisms. We call scheduling mechanisms *fine-grain* if their scheduling/dispatching costs are much lower than a typical CPU quantum, for example, context switch overhead of tens of microseconds compared to CPU quanta of milliseconds.

Traditional timesharing scheduling policies use some global property, such as job priority, to reorder the jobs in the ready queue. A scheduling policy is *adaptive* if the global property is a function of the system state, such as the total amount of CPU consumed by the job. A typical assumption in global scheduling is that all jobs are independent of each other. But in a pipeline of processes, where successive stages are coupled through their input and output, this assumption does not hold. In fact, a global adaptive scheduling algorithm may lower the priority of a CPU-intensive stage, making it the bottleneck and slowing down the whole pipeline.

To make better scheduling decisions for I/O-bound processes, we take into account local information and coupling between jobs in addition to the global properties. We call such scheduling policies *fine-grain* because they use local information. An example of interesting local information is the amount of data in the job's input queue: if it is empty, dispatching the job will merely block for lack of input. This chapter focuses on the coupling between jobs in a pipeline using as the local information the amount of data in the queues linking the jobs.

Fine-grain scheduling is implemented in the Synthesis operating system. The approach is similar to feedback mechanisms in control systems. We measure the progress of each job and make scheduling decisions based on the measurements. For example, if the job is "too slow," say because its input queue is getting full, we schedule it more often and let it run longer. The measurements and adjustments occur frequently, accurately tracking each job's needs.

The key idea in fine-grain scheduling policy is modeled after the hardware phase locked loop (PLL). A PLL outputs a frequency synchronized with a reference input frequency. Our software analogs of the PLL track a reference stream of interrupts to generate a new stable source of interrupts locked in step. The reference stream can come from a variety of sources, for example an I/O device, such as disk index interrupts that occur once every disk revolution, or the interval timer, such as the interrupt at the end of a CPU quantum. For readers unfamiliar with control systems, the PLL is summarized in Section 6.2.

Fine-grain scheduling would be impractical without fast interrupt processing, fast context switching, and low dispatching overhead. Interrupt handling should be fast, since it is necessary for dispatching another process. Context switch should be cheap, since it occurs often. The scheduling algorithm should be simple, since we want to avoid a lengthy search or calculations for each decision. Chapter 3 already addressed the first two requirements. Section 6.2.3 shows that the scheduling algorithms are simple.

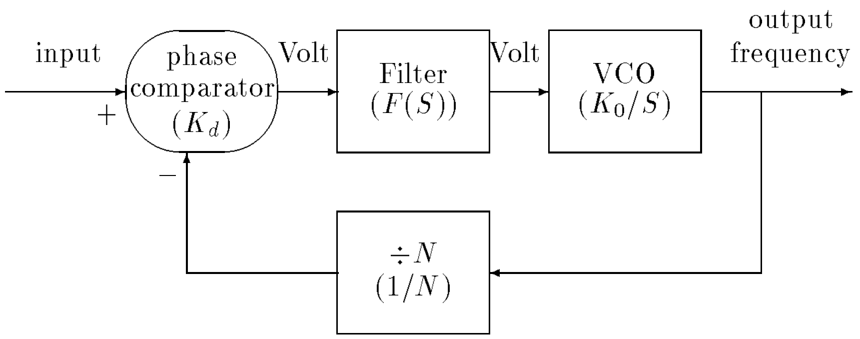


Figure 6.1: PLL Picture

6.2 Principles of Feedback

6.2.1 Hardware Phase Locked Loop

Figure 6.1 shows the block diagram of a PLL. The output of the PLL is an internally-generated frequency synchronized to a multiple of the external input frequency. The phase comparator compares the current PLL output frequency, divided by N, to the input frequency. Its output is proportional to the difference in phase (frequency) between its two inputs, and represents an error signal that indicates how to adjust the output to better match the input. The filter receives the signal from the phase comparator and tailors the time-domain response of the loop. It ensures that the output does not respond too quickly to transient changes in the input. The voltage-controlled oscillator (VCO) receives the filtered signal and generates an output frequency proportional to it. The overall loop operates to compensate the variations on input, so that if the output rate is lower than the input rate, the phase comparator, filter, and oscillator work together to increase the output rate until it matches the input. When the two rates match, the output rate tracks the input rate and the loop is said to be locked to the input rate.

6.2.2 Software Feedback

The Synthesis fine-grain scheduling policies have the same three elements as the hardware PLL. They track the difference between the running rate of a job and the reference frame in a way analogous to the phase comparator. They use a filter to dampen the oscillations in the difference, like the PLL filter. And they re-schedule the running job to

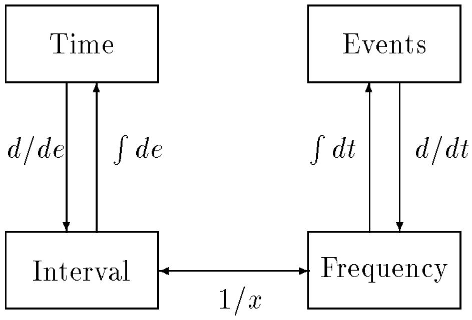


Figure 6.2: Relationship between ILL and FLL

minimize its error compared to the reference, in the same way the VCO adjusts the output frequency.

Let us consider a practical example from a disk driver: we would like to know which sector is under the disk head to perform rotational optimization in addition to the usual seek optimizations. This information is not normally available from the disk controller. But by using feedback, we can derive it from the index-interrupt that occurs once per disk revolution, supplied by some ESDI disk controllers. The index-interrupt supplies the input reference. The rate divider, N, is set to the number of sectors per track. An interval timer functions as the VCO and generates periodic interrupts corresponding to the passage of new sectors under the drive head. The phase comparator and filter are algorithms described in Section 6.2.3.

When we use software to implement the PLL idea, we find more flexibility in measurement and control. Unlike hardware PLLs, which always measure phase differences, software can measure either the frequency of the input (events per second), or the time interval between inputs (seconds per event). Analogously, we can adjust either the frequency of generated interrupts or the intervals between them. Combining the two kinds of measurements with the two kinds of adjustments, we get four kinds of software locked loops. This dissertation looks only at software locked loops that measure and adjust the same variable. We call a software locked loop that measures and adjusts frequency an FLL (frequency locked loop) and a software locked loop that measures and adjusts time intervals an ILL (interval locked loop).

In general, all stable locked loops minimize the error (feedback signal). Concretely, an FLL measures frequency by counting events, so its natural behavior is to maintain the number of events (and thus the frequency) equal to the input. An ILL measures intervals, so its natural behavior is to maintain the interval between consecutive output interrupts equal to the interval between inputs. At first, this seems to be two ways of looking at the same thing. And if the error were always zero, it would be. But when a change in the input happens, there is a period of time when the loop oscillates before it converges to the new output value. During this time, the differences between ILL and FLL show up. An FLL tends to maintain the correct number of events, although the interval between them may vary from the ideal. An ILL tends to maintain the correct interval, even though it might mean losing some events to do so.

This natural behavior can be modified with filters. The overall response of a software locked loop is determined by the kind of filter it uses to transform measurements into adjustments. A low-pass filter makes the FLL output frequency or the ILL output intervals more uniform, less sensitive to transient changes in the input. But it also delays the response to important changes in the input. An integrator filter allows the loop to track linearly changing input without error. Without an integrator, only constant input can be tracked error-free. Two integrators allows the loop to track quadratically changing input without error. But too many integrators tend to make the loop less stable and lengthens the time it takes to converge. A derivative filter improves response to sudden changes in the input, but also makes the loop more prone to noise. Like their hardware analogs, these filters can be combined to improve both the response time and stability of the SLL.

6.2.3 FLL Example

Figure 6.3 shows the general algorithm for an FLL that generates a stream of interrupts at four times the rate of a reference stream. The procedure `i1` services the reference stream of interrupts, while the procedure `i2` services the generated stream. The variable `freq` holds the frequency of i2 interrupts and is updated whenever `i1` or `i2` runs. The variable `residue` keeps track of differences between `i1` and `i2`, serving the role of the phase comparator in a hardware PLL. Each time `i1` executes, it adds 4 to `residue`. Each time `i2` executes, it subtracts 1 from `residue`. The `Filter` function determines how the residue affects the frequency adjustments.

```
int residue=0, freq=0;

/*   Master (reference frame)   */
i1()
{
    residue += 4;
    freq += Filter(residue);
        .
        .
     <do work>
        .
        .
    return;
}

/*    Slave (derived interrupt)    */
i2()
{
     residue--;
     freq += Filter(residue);
         .
      <do work>
         .
     next_time = NOW + 1/freq;
     schedintr(i2, next_time);
     return;
}
```

Figure 6.3: General FLL

```
LoPass(x)
{
    static int lopass;
    lopass = (7*lopass + x) / 8;
    return lopass;
}
```

Figure 6.4: Low-pass Filter

If `i2` and `i1` were running at the perfect relative rate of 4 to 1, `residue` would tend to zero and `freq` would not be changed. But if `i2` is slower than 4 times `i1`, `residue` becomes positive, increasing the frequency of `i2` interrupts. Similarly, if `i2` is faster than 4 times `i1`, `i2` will be slowed down. As the difference in relative speeds increases, the correction becomes correspondingly larger. As `i1` and `i2` approach the exact ratio of 1:4, the difference decreases and we reach the minimum correction with `residue` being decremented by one and incremented by four, cycling between -2 and $+2$. Since `residue` can never converge to zero — only hover around it — the `i2` execution frequency will always jitter slightly. In practice, `residue` would be scaled down by an appropriate factor so that the jitter is negligible.

Figures 6.4, 6.5, and 6.6 show some simple filters that can be used alone or in combination to improve the responsiveness and stability of the FLL. In particular, the low-pass filter shown in Figure 6.4 helps eliminate the jitter mentioned earlier at the expense of a longer settling time. The variable `lopass` keeps a "history" of what the most recent `residue`s were. Each update adds 1/8 of the new `residue` to 7/8 of the old `lopass`. This has the effect of taking a weighted average of recent `residue`s. When residue is positive

```
Integrate(x)
{
    static int accum;
    accum = accum + x;
    return accum;
}
```

Figure 6.5: Integrator Filter

```
Deriv(x)
{
    static int old_x;
    int dx;
    dx = x - old_x;
    old_x = x;
    return dx;
}
```

Figure 6.6: Derivative Filter

for many iterations, as is the case when `i2` is too slow, `lopass` will eventually be equal to `residue`. But if residue oscillates rapidly, as in the situation described in the previous paragraph, `lopass` will go to zero. The derivative is never used alone, but can be used in combination with other filters to improve response to rapidly-changing inputs.

6.2.4 Application Domains

We choose between measuring and adjusting frequency and intervals depending on the desired accuracy and application. Accuracy is an important consideration because we can measure only integer quantities: either the number of events (frequency), or the clock ticks between events (interval). We would like to measure the larger quantity of the two since it carries higher accuracy.

Let us consider a scenario that favors ILL. Suppose you have a microsecond-resolution interval timer and the input event occurs about once per second. To make the output interval match the input interval, the ILL measures second-long intervals with a microsecond resolution timer, achieving high accuracy with few events. Consequently, ILL stabilizes very quickly. In contrast, by measuring frequency (counting events), an FLL needs more events to detect and adjust the error signal. Empirically, it takes about 50 input events (in about 50 seconds) for the output to stabilize to within 10% of the desired value.

A second scenario favors FLL. Suppose you have an interval timer with the resolution of one-sixtieth of a second. The input event occurs 30 times a second. Since the FLL is independent of timer resolution, its output will still stabilize to within 10% after seeing about 50 events (in about 1.7 seconds). However, since the event interval is comparable to the resolution of the timer, an ILL will suffer loss of accuracy. In this example, the measured interval will be either 1, 2 or 3 ticks, depending on the relative timing between the clock and input. Thus the ILL's output can have an error of as much as 50%.

Generally, slow input rates and high resolution timers favor ILL, while high input rates and low resolution timers favor FLL. Sometimes the problem at hand forces a particular choice. For example, in queue handling procedures, the number of get-queue operations must equal the number of put-queue operations. This forces the use of an FLL, since the actual number of events control the actions. In another example, subdivision of a time interval (like in the disk sector finder), an ILL is best.

6.3 Uses of Feedback in Synthesis

We have used feedback-based scheduling policies for a wide variety of purposes in Synthesis. These are:

- An FLL in the thread scheduler to support real-time signal-processing applications.
- An ILL rhythm tracker for a special effects sound processing program.
- A digital oversampling filter for a CD player. An FLL adjusts the filter I/O rate to match the CD player.
- An ILL that adjusts itself to the disk rotation rate, generating an interrupt a few microseconds before each sector passes under the disk head.

6.3.1 Real-Time Signal Processing

Synthesis uses the FLL idea in its thread scheduler. This enables a pipeline of threads to process high-rate, real-time data streams and simplifies the programming of signal-processing applications. The idea is quite simple: if a thread's input queue is filling or if its output queue is emptying, increase its share of CPU. Conversely, if a thread's input queue is emptying or if its output queue is filling, decrease its share of CPU. The effect of

```
main()
{
        char    buf[100];
        int     n, fd1, fd2;

        fd1 = open("/dev/cd", 0);
        fd2 = open("/dev/speaker", 1);
        for(;;) {
                n = read(fd1, buf, 100);
                write(fd2, buf, n);
        }
}
```

Figure 6.7: Program to Play a CD

this scheduling policy is to allocate enough CPU to each thread in the pipeline so it can process its data. Threads connected to the high-speed Sound-IO devices find their input queues being filled — or their output queues being drained — at a high rate. Consequently, their share of CPU increases until the rate at which they process data equals the rate that it arrives. As these threads run and produce output, the downstream threads find that their queues start to fill, and they too receive more CPU. As long as the total CPU necessary for the entire pipeline does not exceed 100%, the pipeline runs in real-time.

The simplification in applications programming that occurs using this scheduler cannot be overstated. One no longer needs to worry about assigning priorities to jobs, or of carefully crafting the inner loops so that everything is executed frequently enough. For example, in Synthesis, reading from the CD player is no different than reading from any other device or file. Simply open "`/dev/cd`" and read from it. To listen to the CD player, one could use the program in Figure 6.7. The scheduler FLL keeps the data flowing smoothly at the 44.1 KHz sampling rate — 176 kilobytes per second for each channel — regardless of how many CPU-intensive jobs might be executing in the background.

Several music-oriented signal-processing applications have been written for Synthesis and run in real-time using the FLL-based thread scheduler. The Synthesis music and signal-processing toolkit includes many simple programs that take sound input, process it in some way, and produce sound output. These include delay elements, echo and reverberation filters, adjustable low-pass, band-pass and high-pass filters, Fourier transform, and a correlator and feature extraction unit. These programs can be connected together in a pipeline to perform more complex sound processing functions, in a similar way that text filters in UNIX can be cascaded using the shell's "`|`" notation. The thread scheduler ensures the pipeline runs in real-time.

6.3.2 Rhythm Tracking and The Automatic Drummer

Besides scheduling, the feedback idea finds use in the actual processing of music signals. In one application, a correlator extracts rhythm pulses from the music on a CD. These are fed to an ILL, which subdivides the beat interval and generates interrupts synchronized to the beat of the music. These interrupts are then used to drive a drum synthesizer, which adds more drum beats to the original music. The interrupts also adjust the delay in the reverberation unit making it equal to the beat interval of the music. You can also get pretty pictures synchronized to the music when you plot the ILL input versus output on a graphics display.

6.3.3 Digital Oversampling Filter

In another music application, an FLL is used to generate the timing information for a digital interpolation filter. A digital interpolator takes as input a stream of sampled data and creates additional samples between the original ones by interpolation. This oversampling increases the accuracy of analog reconstruction of digital signals. We use 4:1 oversampling, i.e. we generate 4 samples using interpolation from each CD sample. The CD player has a new data sample available 44,100 times per second, or one every 22.68 microseconds. The interpolated data output is four times this rate, or one every 5.67 microseconds.[1] We use an FLL to generate an interrupt source at this rate, synchronized with the CD player. This also serves as an example of just how fine-grained the timing can be: an interrupt every 5.67μs corresponds to over 175,000 interrupts per second.

6.3.4 Discussion

A formal analysis of fine-grain scheduling is beyond the scope of this dissertation. However, I would like to give readers an intuitive feeling about two situations: saturation and cheating. As the CPU becomes saturated, the FLL-based scheduler degrades gracefully. The processes closest to externally generated interrupts (device drivers) will still get the necessary CPU time. The CPU-intensive processes away from I/O interrupts will slow down first, as they should at saturation.

[1]This program runs on the Quamachine at 50 MHz clock rate.

Another potential problem is cheating by consuming resources unnecessarily to increase priority. This is possible because fine-grain scheduling tends to give more CPU to processes that consume more. However, cheating cannot be done easily from within a thread or by cooperation of several threads. First, unnecessary I/O loops within a program does not help the cheater, since they do not speed up data flow in the pipeline of processes. Second, I/O within a group of threads only shifts CPU quanta within the group. A thread that reads from itself gains quanta for input, but loses the exact amount in the self-generated output. To increase the priority of a process, it must read from a real input device, such as the CD player. In this case, it is virtually impossible for the OS kernel to distinguish the real I/O from cheating I/O.

6.4 Other Applications

6.4.1 Clocks

The FLL provides integral stability. This means the long-term drift between the reference frame and generated interrupts tends to zero, even though any individual interval may differ from the reference. This is in contrast with differential stability, in which the consecutive intervals are all the same, but any systematic error, no matter how small, will accumulate into a long-term drift. To illustrate, the interval timers found on many machines provide good differential stability: all the intervals are of very nearly the same length. But they do not provide good integral stability: they do not keep good time.

The integral stability property of the FLL lets it increase the resolution of precise timing sources. The idea is to synchronize a higher-resolution but less precise timing device, such as the machine's interval timer, to the precise one. The input to the FLL would be an interrupt derived from a very precise source of timing, for example, from an atomic clock. The output is a new stream of interrupts occurring at some multiple of the input rate.

Suppose the atomic clock ticks once a second. If the FLL's rate divider, N, is set to 1000, then the FLL will subdivide the second-long intervals into milliseconds. The FLL adjusts the interval timer so that each 1/1000-th interrupt occurs as close to the "correct" time of arrival as possible given the resolution of the interval timer, while maintaining integral stability — N interrupts out for every interrupt in. If the interval timer used exhibits good differential stability, as most interval timers do, the output intervals will be both precise and accurate.

But for this to work well, one must be careful to avoid the accumulation of round-off error when calculating successive intervals. A good rule-of-thumb to remember is: calculate based on elapsed time; not on intervals. Use differences of elapsed times whenever an interval is required. This is crucial to guaranteeing convergence. The sample FLL in figure 6.3 follows these guidelines.

To illustrate this, suppose that the hardware interval timer ticks every 0.543 microseconds.[2] Using this timer, a millisecond is 1843.2 ticks long. But when scheduling using intervals, 1843.2 is truncated to 1843 since interrupts can happen only on integer ticks. This gains time. One second later, the FLL will compensate by setting the interval to 1844. But now it loses time. The FLL ends up oscillating between 1843 and 1844, and never converging. Since the errors accumulate all in the same direction for the entire second before the adjustment occurs, the resulting millisecond subdivisions are not very accurate.

A better way to calculate is this: let the desired interval (1/frequency) be a floating-point number and accumulate intervals into an elapsed-time accumulator using floating-point addition. Interrupts are scheduled by taking the integer part of the elapsed-time accumulator and subtracting it from the previous elapsed-time to obtain the integer interval. Once convergence is reached, 4 out of 5 interrupts will be scheduled every 1843 ticks and 1 out of 5 every 1844 ticks, evenly interspersed, averaging to 1843.2. Each interrupt will occur as close to the 1-millisecond mark as possible given the resolution of the timer (e.g., they will differ by at most $\pm 0.272\mu s$). In practice, the same effect can be achieved using appropriately scaled integer arithmetic, and floating point arithmetic would not be used.

6.4.2 Real-Time Scheduling

The adaptive scheduling strategy might be improved further, possibly encompassing many hard real-time scheduling problems. Hard real-time scheduling is a harder problem than the real-time stream processing problem discussed earlier. In stream processing, each job has a small queue where data can sit if the scheduler makes an occasional mistake. The goal of fine-grain scheduling is to converge to the correct CPU assignments for all the jobs before any of the queues overflow or underflow. In contrast, hard real-time jobs must meet their deadline, every single time. Nevertheless, I believe that the feedback-based scheduling

[2] This is a common number on machines that derive timing from the baud-rate generator used in serial communications.

idea will find useful application in this area. In this section, I only outline the general idea, without offering proof or examples. For a good discussion of issues in real-time computing, see [29].

We divide hard-deadline jobs into two categories: the short ones and the long ones. A short job is one that must be completed in a time frame within an order of magnitude of interrupt and context switch overhead. For example, a job taking up to 100 microseconds would be a short job in Synthesis. Short jobs are scheduled as they arrive and run to completion without preemption.

Long jobs take longer than 100 times the overhead of an interrupt and context switch. In Synthesis this includes all the jobs that take more than 1 millisecond, which includes most of the practical applications. The main problem with long jobs is the variance they introduce into scheduling. If we always take the worst scenario, the resulting hardware requirement is usually very expensive and unused most of the time.

To use fine-grain scheduling policies for long jobs, we break down the long job into small *strips*. For simplicity of analysis we assume each strip to have the same execution time ET. We define the estimated CPU power to finish job J as:

$$Estimate(J) = \frac{(\text{strips in J}) * ET}{Deadline(J) - Now}$$

For a long job, it is not necessary to know ET exactly since the locked loop "measures" it and continually adjusts the schedule in lock step with the actual execution time. In particular, if $Estimate(J) > 1$ then we know from the current estimate that J will not make the deadline. If we have two jobs, A and B, with $Estimate(A) + Estimate(B) > 1$ then we may want to consider aborting the less important one and calling a short emergency routine to recover.

Unlike traditional hard-deadline scheduling algorithms, which either guarantee completion or nothing, fine-grain scheduling provides the ability to predict the deadline miss under dynamically changing system loads. I believe this is an important practical concern to real-time application programmers, especially in recovery from faults.

6.4.3 Multiprocessor and Distributed Scheduling

I also believe the adaptiveness of FLL promises good results in multiprocessor and distributed systems. As in the previous section, the idea can be offered at this writing, but with little support. At the risk of oversimplification, I describe an example with fixed

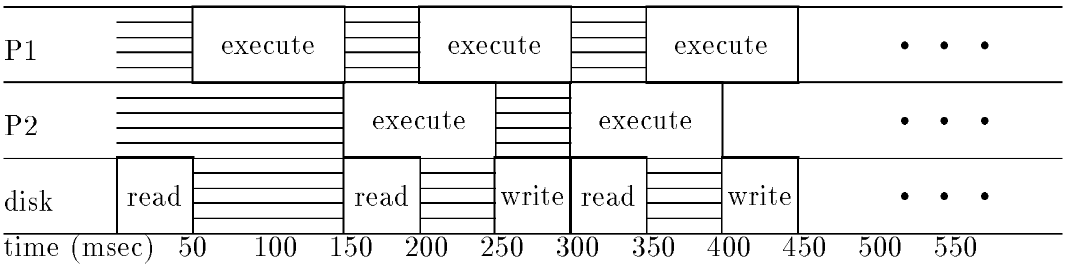


Figure 6.8: Two Processors, Static Scheduling

buffer size and execution time. Recognize that at a given a load, we can always find the optimal scheduling statically by calculating the best buffer size and CPU quantum. But I emphasize the main advantage of feedback: the ability to dynamically adjust towards the best buffer size and CPU quantum. This is important when we have a variable system load, jobs with variable demands, or a reconfigurable system with a variable number of CPUs.

Figure 6.8 shows the static scheduling for a two-processor shared-memory system with a common disk (transfer rate of 2 MByte/second). We assume that both processes access the disk drive at the full transfer rate, e.g. reading and writing entire tracks. Process 1 runs on processor 1 (P1) and process 2 runs on processor 2 (P2). Process 1 reads 100 KByte from the disk into a buffer, takes 100 milliseconds to process them, and writes 100 KByte through a pipe into process 2. Process 2 reads 100 KByte from the pipe, takes another 100 milliseconds to process them, and writes 100 KByte out to disk. In the figure, process 1 starts to read at time 0. All disk activities appear in the bottom row, P1 and P2 show the processor usage, and shaded quadrangles show idle time.

Figure 6.9 shows the fine-grain scheduling mechanism (using FLL) for the same system. We assume that process 1 starts by filling its 100 KByte buffer, but soon after it starts to write to the output pipe, process 2 starts. Both processes run to exhaust the buffer, when process 1 will read from the disk again. After some settling time, depending on the filter used in the locked loop, the stable situation is for the disk to remain continuously active, alternatively reading into process 1 and writing from process 2. Both processes will also run continuously, with the smallest buffer that maintains the nominal transfer rate.

The above example illustrates the benefits of fine-grain scheduling policies in par-

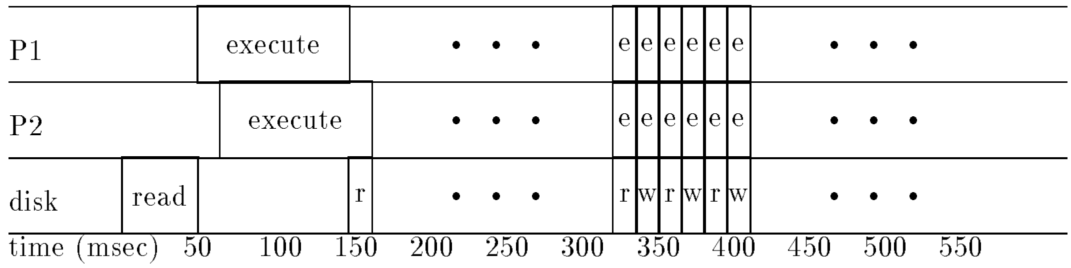


Figure 6.9: Two Processors, Fine-Grain Scheduling

allel processing. In a distributed environment, the analysis is more complicated due to network message overhead and variance. In those situations, calculating statically the optimal scheduling becomes increasingly difficult. We expect the fine-grain scheduling to show increasing usefulness as it adapts to an increasingly complicated environment.

Another application of FLL to distributed systems is clock synchronization. Given some precise external clocks, we would like to synchronize the rest of machines with the reference clocks. Many algorithms have been published, including a recent probabilistic algorithm by Christian [10]. Instead of specialized algorithms, we use an FLL to synchronize clocks, where the external clock is the reference frame, the message delays introduce the jitter in the input, and we need to find the right combination of filters to adapt the output to the varying message delays. Since an FLL exhibits integral stability, the clocks will tend to synchronize with the reference once they stabilize. We are currently collecting data on the typical message delay distributions and finding the appropriate filters for them.

6.5 Summary

We have generalized scheduling from job assignments as a function of time, to job assignments as a function of any source of interrupts. The generalized scheduling is most useful when we have fine-grain scheduling, that uses frequent state checks and dispatching actions to adapt quickly to system changes. Relevant new applications of the generalized fine-grain scheduling include I/O device management, such as a disk sector interrupt source, and adaptive scheduling, such as real-time scheduling and distributed scheduling.

The implementation of fine-grain scheduling in Synthesisis based on feedback systems, in particular the phase locked loop. Synthesis' fine-grain scheduling policy means adjustments every few hundreds of microseconds on local information, such as the number of characters waiting in an input queue. Very low overhead scheduling and context switch for dispatching form the foundation of our fine-grain scheduling mechanism. In addition, we have very low overhead interrupt processing to allow frequent checks on the job progress and quick, small adjustments to the scheduling policy.

There are two main advantages of fine-grain scheduling: quick adjustment to changing situations, and early warning of potential deadline misses. Quick adjustments make better use of system resources, since we avoid queue/buffer overflow and other mismatches between the old scheduling policy and the new situation. Early warning of deadline misses allows real-time application programmers to anticipate a disaster and attempt an emergency recovery before the disaster strikes.

We have only started exploring the many possibilities that generalized fine-grain scheduling offers. Distributed applications stand to benefit from the locked loops, since they can track the input interrupt stream despite jitters introduced by message delays. Concrete applications we are studying include load balancing, distributed clock synchronization, smart caching in memory management and real-time scheduling. To give one example, load balancing in a real-time distributed system can benefit greatly from fine-grain scheduling, since we can detect potential deadline misses in advance; if a job is making poor progress towards its deadline locally, it is a good candidate for migration.

7

Measurements and Evaluation

15. Everything should be built top-down, except the first time.
— Alan J. Perlis *Epigrams on Programming*

7.1 Measurement Environment

7.1.1 Hardware

The current implementation of Synthesis runs on two machines: the Quamachine and the Sony NEWS 1860 workstation. As described in section 1.3.4, the Quamachine is a home-brew, experimental 68030-based computer system designed to aid systems research and measurement. Its measurement facilities include an instruction counter, a memory reference counter, hardware program tracing, and a memory-mapped clock with 20-nanosecond resolution. The processor can operate at any clock speed from 1 MHz up to 50 MHz. Normally it runs at 50 MHz. But by changing the processor speed and introducing wait-states into the main memory access, the Quamachine can closely emulate the performance characteristics of common workstations, simplifying measurements and comparisons. The Quamachine also has special I/O devices that support digital music and audio signal processing: stereo 16-bit analog output, stereo 16-bit analog input, and a compact disc (CD) player digital interface.

The Sony NEWS 1860 is a commercially-available workstation with two 68030 processors. Its architecture is not symmetric. One processor is meant to be the main processor and the other is meant to be the I/O processor. Synthesis tries to treat it as if it were a symmetric multiprocessor, scheduling most tasks on either processor without preference, except those that require something that is accessible from one processor and not the other. While this is not a large number of processors, it nevertheless helps demonstrate Synthesis multiprocessor support. But for measurement purposes of this chapter, only one processor — the slower I/O processor — was used. (With the kernel's multiprocessor support kept intact.)

7.1.2 Software

A partial emulator for UNIX runs on top of the Synthesis kernel and emulates some of the SUNOS (version 3.5) kernel calls. This provides a direct way of measuring and comparing two otherwise very different operating systems. Since the executables are the same, the comparison is direct. The emulator further demonstrates the generality of Synthesis by setting the lower bound — Synthesis is at least as general as UNIX if it can emulate UNIX. It also helps with the problem of acquiring application software for a new operating system by allowing the use of SUN-3 binaries instead. Although the emulator supports a subset of the UNIX system calls — time constraints have forced an "implement-as-the-need-arises" strategy — the set supported is sufficiently rich to provide a good idea of what the relative times for the basic operations are.

7.2 User-Level Measurements

7.2.1 Comparing Synthesis with SUNOS 3.5

This section describes a comparison between Synthesis and SUNOS 3.5. The benchmark programs consist of simple loops that exercise a particular system function many times. The source code for the programs is in appendix A. All benchmark programs were compiled on the SUN 3/160, using `cc -O` under SUNOS release 3.5. The executable `a.out` was timed on the SUN, then brought over to the Quamachine and executed using the UNIX

Program	Raw Sun Data				Sun	Synthesis	Ratio	I/O Rate
	usr	sys	total	watch	usr+sys	Emulator		(MB/Sec)
1 Compute	19.8	0.5	20	20.9	20.3	21.42	0.95	—
2 R/W pipe 1	0.4	9.6	10	10.2	10.0	0.18	56.	0.1
3 R/W pipe 1024	0.5	14.6	15	15.3	15.1	2.42	6.2	8
4 R/W pipe 4096	0.7	37.2	38	38.2	37.9	9.64	3.9	8
5 R/W file	0.5	20.1	21	23.4	20.6	2.91	7.1	6
6 open null/close	0.5	17.3	17	17.4	17.8	0.69	26.	—
7 open tty/close	0.5	42.1	43	43.1	42.6	0.88	48.	—

Table 7.1: Measured UNIX System Calls (in seconds)

emulator.

Ideally, we would want to run both Synthesis and SUNOS on the same hardware. Unfortunately, we could not obtain detailed information about the Sun-3 machine, so Synthesis has not been ported to the Sun. Instead, we closely emulate the hardware characteristics of a Sun-3 machine using the Quamachine. This involves three changes: replace the 68030 CPU with a 68020, set the CPU speed to 16MHz, and introduce one wait-state into the main-memory access. To validate faithfulness of the hardware emulation, the first benchmark program is a compute-bound test. This test program implements a function producing a chaotic sequence.[1] It touches a large array at non-contiguous points, which ensures that we are not just measuring the "in-the-cache" performance. Since it does not use any operating system resources, the measured times on the two machines should be the same.

Table 7.1 summarizes the results of the measurements. The columns under "Raw SUN data" were obtained using the UNIX `time` command and verified with a stopwatch. The SUN was unloaded during these measurements and `time` reported more than 99% CPU available for them. The columns labeled "usr," "sys," and "total" give the time spent in the user's program, in the SUNOS kernel, and the total elapsed time, as reported by the

[1]Pages 137-138 in Godel, Escher, Bach: An Eternal Golden Braid, by Douglas Hofstadter.

`time` command. The column labeled "usr+sys" is the sum of the user and system times, and is the number used for comparisons with Synthesis. The Synthesis emulator data were obtained by using the microsecond-resolution real-time clock on the Quamachine, rounded to hundredths of a second. These times were also verified with stopwatch, sometimes by running each test 10 times to obtain a more easily measured time interval. The column labeled "Ratio" gives the ratio of the preceding two columns. The last column, labeled "I/O Rate", gives the overall Synthesis I/O rate in megabytes per second for those test programs performing I/O.

The first program is a compute-intensive calibration function to validate the hardware emulation.

Programs 2, 3, and 4 write and then read back data from a UNIX pipe in chunks of 1, 1024, and 4096 bytes. Program 2 shows a remarkable speed advantage — 56 times — for the single-byte read/write operations. Here, the low overhead of the Synthesis kernel calls really makes a difference, since the amount of data moved is small and most of the time is spent in overhead. But even as the I/O size grows to the page size, the difference remains significant — 4 to 6 times. Part of the reason is that the SUNOS overhead is still significant even when amortized over more data. Another reason is the fast synthesized routines that move data across address spaces. The generated code loads words from one address space into registers and stores them back in the other address space. With unrolled loops this achieves the data transfer rate of about 8MB per second.

Program 5 reads and writes a file (cached in main memory) in chunks of 1K bytes. It too shows a remarkable speed improvement over SUNOS.

Programs 6 and 7 repeatedly open and close `/dev/null` and `/dev/tty`. They show that Synthesis kernel code generation is very efficient. The `open` operations create executable code for later read and write, yet they are 20 to 40 times faster than the UNIX `open` that does not do code generation. Table 7.3 contains more details of file system operations that are discussed in the next section.

7.2.2 Comparing Window Systems

A simple measurement gives an idea of the speed of interactive I/O on various machines running different window systems. We use "`cat /etc/termcap`" to a TTY window. The local termcap file is 110620 bytes long. The window size is 80 characters wide by 24

OS, Window System	Machine	CPU	Time (Seconds)
Synthesis	Sony NEWS	68030, 25mhz	2.9
UNIX, X11 R5	Sony NEWS	68030, 25mhz	23
UNIX, console	Sony NEWS	68030, 25mhz	127
Mach, NextStep	NeXT	68030, 25mhz	55
Mach, NextStep	NeXT	68040, 25mhz	13
SUNOS, X11 R5	Sun SparcStation II	Sparc	6.5

Table 7.2: Time to "cat /etc/termcap" to a 80*24 TTY window

lines, using a 16 by 24 pixel font, and with scrollbars enabled.

Table 7.2 summarizes the times taken by the various machines and window systems. There are many good reasons why the other window systems are slow. The Sony console device driver, for example, scrolls the whole screen one line at a time, even when there are several lines of output waiting. The X window system uses RPC to communicate between client and server; no doubt this adds to the overhead. The NextStep window system is based on Postscript, which is overkill for the task at hand.

The point is not to parade Synthesis speed nor justify the other's slowness. It is to point out that that speed *is* possible through careful thought and program structuring that provides just the right level of abstraction for each application. For example, one application that runs under Synthesis reads music data from the CD player, computes its Fourier transform (1024 point), and displays the result in a window, *all in real-time*. It displays 88200 data points per second. This is impossible to do today using any other single-processor workstation and operating system because the abstractions provided are too expensive and just plain wrong for this particular task. This is true even though the newer Sparc-based workstations from SUN are more than four times faster than the machine running Synthesis. Section 7.3.3 shows detailed measurements for the Synthesis window system.

Operation	Native Time	UNIX Emulation
emulation trap	—	2
open /dev/null	43	49
open /dev/tty	62	68
open (disk file)	73	85
close	18	22
read 1 byte from file	9	10
read N bytes from file	9+N/8	10+N/8
read N from /dev/null	6	8

Table 7.3: File and Device I/O (in microseconds)

7.3 Detailed Measurements

The Quamachine's 20-nanosecond resolution memory-mapped clock enables precise measurement of the time taken by each individual system call. To obtain direct timings in microseconds, we surround the system call to be measured with two "read clock" machine instructions and subtract to find the elapsed time.

7.3.1 File and Device I/O

Table 7.3 gives the time taken by various file- and device-related I/O operations. It compares the timings measured for the native Synthesis system calls and for the equivalent call in SUNOS emulation mode. For these tests, the Quamachine was running at 25MHz using the 68030 CPU.

Worth noting is the cost of `open`. The simplest case, `open /dev/null`, takes 49 microseconds, of which about 70% are used to find the name in the directory structure and 30% for memory allocation and code synthesis to create the null read and write procedures. The additional 19 microseconds in opening `/dev/tty` come from generating more involved code to read and write the TTY device. Finally, opening a file requires synthesizing more sophisticated code and buffer allocations, costing 17 additional microseconds.

Operation	Time (μs)
Service Translation Fault	13.6
Allocate page (pre-zeroed)	2.4 + 13.6 = 16.0
Allocate page (needs zeroing)	152 + 13.6 = 166
Allocate page (none free; replace)	$154 + 13.6 + T_{replace} = 168 + T_{replace}$
Copy a page (4 Kbytes)	260 + 13.6 = 274
Free page	1.6

Table 7.4: Low-level Memory Management Overhead (Page Size = 4KB)

7.3.2 Virtual Memory

Table 7.4 gives the time taken by various basic operations related to virtual memory. The first row, labeled "Service Translation Fault," gives the time taken to service a translation fault exception. It represents overhead that is always incurred, regardless of the reason for the fault. Translation faults happen whenever a memory reference can not be completed because the address could not be translated. The reasons are manifold: the page is not present, or it is copy-on-write, or it has not been allocated, or that reference is not allowed. This number includes the time taken by the hardware to detect the translation fault, save the machine state, and dispatch to the fault handler. It includes the time taken by the Synthesis fault handler to interpret the saved hardware state, determine the reason for the fault, and dispatch to the correct sub-handler. And it includes the time to re-load the machine state and retry the reference once the sub-handler has fixed the situation.

Subsequent rows give the additional time taken by the various sub-handlers, as a function of the cause of the fault. The numbers are shown in the form "$X + 13.6 = Y$," where X is the time taken by the sub-handler alone, and Y the total time including the fault overhead. The second row of the table gives the time to allocate a zeroed page when one already exists. (Synthesis uses idle CPU time to maintain a pool of pre-zeroed pages for faster allocation.) The third row gives the time taken to allocate and zero a free page. If no page is free, one must be replaced, and this cost is given in the fourth row.

Quaject	μs to Create	μs to Write
TTY-Cooker	27	2.3 + 2.1/char
VT-100 terminal emulator	532	14.2 + 1.3/char
Text window	71	23.9 + 27.7/char

Table 7.5: Selected Window System Operations

7.3.3 Window System

A terminal window is composed of a pipeline of three quajects: a TTY-Cooker, a VT-100 Terminal Emulator, and a Text-Window. Each quaject has a fixed cost of invocation and a per-character cost that varies depending on the character being processed. These costs are summarized in Table 7.5. The numbers are show in the form "$X + Y$/char," where X is the invocation cost and Y the average per-character costs. The average is taken over the characters in `/etc/termcap`.

The numbers in Table 7.5 can be used to predict the elapsed time for the "`cat /etc/termcap`" measurement done in Section 7.2.2. Performing the calculation, we get 3.4 seconds if we ignore the invocation overhead and use only the per-character costs. Notice that this exceeds the elapsed time actually observed (Table 7.2). This unexpected result happens because Synthesis kernel can optimize the data flow, resulting in fewer calls and less actual work than a straight concatenation of the three quajects would indicate. For example, in a fast window system, many characters may be scrolled off the screen between the consecutive vertical scans of the monitor. Since these characters would never be seen by a user, they need not be drawn. The Synthesis window manager bypasses the drawing of those characters by using fine-grained scheduling. It samples the content of the virtual VT100 screen 60 times a second, synchronized to the vertical retrace of the monitor, and draws the parts of the screen that have changed since the last time. This is a good example of how fine-grain scheduling can streamline processing, bypassing I/O that does not affect the visible result. The data is not lost, however. All the data is available for review using the window's scrollbars.

7.3.4 Other Figures

Other performance figures at the same level of detail were already given in the previous chapters. In Table 5.2 on page 88, we see that Synthesis kernel threads are lightweight, with less than 20 microsecond creation time; Table 5.3 on page 89 shows that thread context switching is fast. Table 3.4 on page 41 gives the time taken to handle the high-rate interrupts from the Sound-IO devices.

7.4 Experience

7.4.1 Assembly Language

The current version of Synthesis is written in 68030 macro assembly language. This section reports on the experience.

Perhaps the first question people ask is, "Why is Synthesis written in assembler?" This is soon followed by "How much of Synthesis could be re-written in a high-level language?" and "At what performance loss?".

There are several reasons why assembler language was chosen, some of them research-related, and some of them historical. One reason is I felt that it would be an interesting experiment to write a medium-size system in assembler, which allows unrestricted access to the machine's architecture, and perhaps discover new coding idioms that have not yet been captured in a higher-level language. Later paragraphs talk about these. Another reason is that much of the early work involved discovering the most efficient way of working with the machine and its devices. It was a fast prototyping language, one in which I could write and test simple I/O drivers without the trouble of supporting a complex language runtime environment.

But perhaps the biggest reason is that in 1984, at the time the seed ideas were being developed, I could not find a good, reliable (bug-free) C compiler for the 68000 processor. I had tried the compilers on several 68000-based UNIX machines and repeatedly found that compilation was slow, that the compilers were buggy, that they produced terrible machine code, and that their runtime libraries were not reentrant. These qualities interfered with my creativity and desire to experiment. Slow compilation dampens the enthusiasm of trying new ideas because the edit-compile-test cycle is lengthened. Buggy compilers makes it that much harder to write correct code. Poor code-generation makes my optimization efforts seem meaningless. And non-reentrant runtime libraries makes it harder to write a multi-threaded kernel that can take advantage of multiprocessor architecture.

Having started coding in assembler, it was easier to continue that way than to change. I had written an extensive library of utilities, including a fully reentrant C-language runtime library and subroutines for music and signal processing. In particular, I found my signal processing algorithms difficult to express in C. To achieve the high performance necessary for real-time operation, I use fixed-point arithmetic for the calculations, not floating-point. The C language provides poor support for fixed-point math, particularly multiply and divide. The Synthesis "`printf`" output conversion and formatting function provides a stunning example of the performance improvements that result with carefully-coded fixed-point math. This function converts a floating-point number into a fully-formatted ASCII string, *1.5 times faster* than the *machine instruction* on the 68882 floating-point coprocessor converts binary floating-point to unformatted BCD (binary-coded decimal).

Overall, the experience has been a positive one. A powerful macro facility helped minimize the difficulty of writing complex programs. The Synthesis assembler macro processor borrows heavily from the C-language macro processor, sharing much of the syntax and semantics. It provides important extensions, including macros that can define macros and quoting and "eval" mechanisms. Quaject definition, for example, is a declarative macro instruction in the assembler. It creates all the code and data structures needed by the kernel code generator, so the programmer need not worry about these details and can concentrate on the quaject's algorithms. Also, the Synthesis assembler (written in C, by the way) assembles 5000 lines per second. Complete system generation takes only 15 seconds. The elapsed time from making a change to the Synthesis source to having a new kernel booted and running is less than a minute. Since the turn-around time is so fast, I am much more likely to try different things.

To my surprise, I found that there are some things that were distinctly *easier* to do using Synthesis assembler than using C. In many of these, the powerful macro processor played an important role, and I believe that the C language could be usefully improved with this macro processor. One example is the procedure that interprets receiver status code bits in the driver for the LANCE Ethernet controller chip. Interpreting these bits is a little tricky because some of the error conditions are valid only when present in conjunction with certain other conditions. One could always use a deeply-nested `if-then-else` structure to separate out the cases. It would work and also be quite readable and maintainable. But a jump-table implementation is faster. Constructing this table is difficult and error-prone. So we use macros to do it. The idea is to define a macro that evaluates the jump-address corresponding to a constant status-value passed as its argument. This macro is defined using preprocessor "`#if`" statements to evaluate the complex conditionals, which is just as readable and maintainable as regular `if` statements. The jump-table is then constructed by passing this macro to a counting macro which repeatedly invokes it, passing it 0, 1, 2, ... and so on, up to the largest status register value (128).

The VT-100 terminal emulator is another place where assembly language made the job of coding easier. The VT-100 terminal emulator takes as input a buffer of data and interprets it, making changes to the virtual terminal screen. A problem arises when the input buffer runs out while in the middle of processing an escape sequence, for example, one which sets the cursor to an (X,Y) position on the screen. When this happens, we must save enough state so that processing can resume where it left off when the emulator is called again with more data. Saving the state variables is easy. Saving the position within the program is harder. There is no way to access the program counter from the C language. This is a big problem because the VT-100 emulator is very complex, and there are many places where execution may be suspended. Using C, one must label all these places, and surround the whole piece of code with a huge `switch` statement to take execution flow to the right place when the function is called again. Using assembly language, this problem does not arise. We can encode the state machine directly, using the different program counter addresses to represent the different states.

I believe much of Synthesis could be re-written in C, or a C-like high-level language. Modern compilers now have much better code generators, and I feel that performance of the static runtime code would not degrade too much — perhaps less than 50%. Runtime code-generation could be handled by writing machine instructions into integer arrays and this code would continue to be highly efficient but still unportable. However, with the code generator itself written in a high-level language, porting it might be easier.

I feel that adding a few new features to the C language can simplify the rewriting of Synthesis and help minimize the performance loss. Features I would like to see include:

- A code-address data type to hold program-counter values, and an expanded "`goto`" to transfer control to such addresses. State machines in particular can benefit from a "`goto a[i]`" programming construct.

- A concept of a subroutine within a procedure, analogous to the "`jsr`...`rts`" instructions in assembly language. These would allow direct language model of the underlying hardware stack. They are useful to separate out into subroutines common blocks of code within a procedure, without the argument passing and procedure call overhead of ordinary functions, since subroutines implicitly inherit all local variables. Among other things, I have found that LALR(1) context-free parsers can be implemented very efficiently by representing the parser stack using the hardware, and using `jsr` and `rts` to perform the state transitions.

- Better support for fixed-point math. Even an efficient way of obtaining the full 64-bit result from a 32-bit integer multiplication would go a long way in this regard.

The inclusion of features like these does *not* mean that I encourage programmers to write spaghetti-code. Rather, these features are intended to supply the needed hooks for automatic program generators, for example, a state machine compiler, to take maximum benefit of the underlying hardware.

7.4.2 Porting Synthesis to the Sony NEWS Workstation

Synthesis was first developed for the Quamachine, and like many substantial software systems, has gone through several revisions. The early kernel had several shortcomings. While the kernel showed impressive speed gains over conventional operating systems such as UNIX, its internal structure was not clean. The quaject structuring idea had come late in kernel development, so there were many parts that had been written in an *ad hoc* manner. Furthermore, the Quamachine kernel did not support virtual memory or networking.

The goal of the Synthesis port to the Sony workstation was to alleviate the shortcomings, for example, by cleaning up the kernel structure and adding virtual memory and networking support. In particular, we wanted to show that the additional functionality would not significantly slow down the Synthesis kernel. This section reports on the experience and discusses the problems encountered while porting.

The Synthesis port happened in three stages: first, a minimal Synthesis is ported to run under Sony's native UNIX. Then we wrote drivers for the keyboard and screen, and got minimal Synthesis to run on the raw hardware. This was followed by a full port, including all the devices.

The first step went fast, taking two to three weeks. The reason is that most of the quajects do not need to run in kernel mode in order to work. The difference between Synthesis under UNIX and native Synthesis is that instead of connecting the final-stage I/O quajects to I/O device driver quajects (which are the only quajects that must be in the kernel), we connect them to UNIX `read` and `write` system calls on appropriately opened file descriptors. This is ultimate proof that Synthesis services can run in user-level as well as kernel.

Porting to the raw machine was much harder, primarily because we chose to do our own device drivers. Some problems were caused by incomplete documentation on how to program the I/O devices on the Sony NEWS workstation. It was further complicated by the fact that each CPU has a different mapping of the I/O devices onto memory addresses and not everything is accessible by both CPUs. A simple program was written to patch the running UNIX kernel and install a new system call — "execute function in kernel mode." Using this utility (carefully!), we were able to examine the running kernel and discover a few key addresses. After a bit more poking around, we discovered how to alter the page mappings so that sections of kernel and I/O memory were directly mapped into all user address spaces.[2] (The `mmap` system call on `/dev/mem` did not work.) Then using the Synthesis kernel monitor running on minimal Synthesis under a UNIX process, we were able to "hand access" the remaining I/O devices to verify their addresses and operation.

The Synthesis kernel monitor is basically a C-language parser front-end with direct access to the kernel code generators. It was crucial to both development and porting of Synthesis because it let us run and test sections of code without having the full kernel present. A typical debug cycle goes something like this: using the kernel monitor, we instantiate the quaject we want to test. We create a thread and point it at one of the quaject's callentries. We then single-step the thread and verify that the control flows where it is supposed to.

But the most difficult porting problems were caused by timing sensitivities in the various I/O devices. Some devices would "freeze" when accessed twice in rapid succession. These problems never showed up in the UNIX code because UNIX encapsulates device access in procedures. Calling a procedure to read a status value or change a control register allows enough time for the device to "recover" from the previous operation. But with

[2]Talk about security holes!

code synthesis, device access frequently consists of a single machine instruction. Often the same device is accessed twice in rapid succession by two consecutive instructions, causing the timing problem. Once the cause of the problem was found, it was easy to correct: I made the kernel code generator insert an appropriate number of "`nop`" instructions between consecutive accesses.

Once we had the minimal kernel running, getting the rest of the kernel and its associated libraries working was relatively easy. All of the code that did not involve the I/O devices ran without change. This includes the user-level shared runtime libraries, such as the C functions library and the signal-processing library. It also includes all the "intermediate" quajects that do not directly access the machine and its I/O devices, such as buffers, symbol tables (for name service), and mappers and translators (for file system mapping). Code involving I/O devices was harder, since that required writing new drivers. Finally, there are some unfinished drivers such as the SCSI disk driver.

The thread system needed some changes to support the two CPUs on the Sony workstation; these were discussed in Chapter 5. Most of the changes were in the scheduling and dispatching code, to synchronize between the processors. This involved developing efficient, lock-free data structures which were then used to implement the algorithms. The scheduling policy was also changed from a single round-robin queue to one that uses a multiple-level queue structure. This helped guarantee good response time to urgent events even when there are many threads running, making it feasible to run thousands of threads on Synthesis.

The most time-consuming part was implementing the new services: virtual memory, Ethernet driver, and window system. They were all implemented "from scratch," using all the performance-improving ideas discussed in this dissertation, such as kernel code generation. The measurements in this chapter show high performance gains in these areas as well. The Ethernet driver, for example, is fast enough to record all the packet traffic of a busy Ethernet (400 kilobytes/second, or about 4 megabits per second) into RAM using only 20% of a 25MHz, 68030 CPU's time. This is a problem that has been worked on and dismissed as impractical except when using special hardware.

Besides the Sony workstation, the new kernel runs on the Quamachine as well. Of course, each machine must use the appropriate I/O drivers, but all the new services added to the Sony version work on the Quamachine.

7.4.3 Architecture Support

Having worked very close to the hardware for so long, I have acquired some insight of what kinds of things would be useful for better operating systems support in future CPUs. Rather than pour out everything I ever thought useful for a machine to have, I will keep my suggestions to those that fit reasonably well with the "RISC" idea of processor design.

- **Better cache control** to support runtime code generation. Ideally, I would like to see fully coherent instruction caches. But I recognize the expense involved, both in silicon area and degraded signal propagation times. But full coherence is probably not necessary. A cheap, *non-privileged* instruction to invalidate changed cache lines provides very good support at minimal cost for both hardware and code-modifying software. After all, if you've just modified an instruction, you know it's address, and it is easy to issue a cache-line invalidate on that address.

- **Faster interrupt handling.** Chapter 6 discussed the advantages of fine-grained handling of computation, particularly when it comes to interrupts. Further benefits result by also reducing the hardware-imposed overhead of interrupt handing. Perhaps this can be achieved at not-too-great expense by replicating the CPU pipeline registers much like register-windows enable much faster procedure call. I expect even a single level of duplication to really help, if we assume that interrupts are handled fast enough that the chances are small of receiving a second interrupt in the middle of processing the first.

- **Hardware support for lock-free synchronization.** Chapter 5 discussed the virtues of lock-free synchronization. But lock-free synchronization requires hardware support in the form of machine instructions that are more powerful than the test-and-set instruction used to implement locking. I have found that double-word *Compare-&-Swap* is sufficient to implement an operating system kernel, and I conjecture that single-word *Compare-&-Swap* is too weak. There may also be other kinds of instructions that also work.

- **Hardware support for fast context switching.** As processors become faster and more complex, they have increasing amounts of state that must be saved and restored on every context switch. Earlier sections had discussed the cost of switching the floating-point context, which is high because of the large amount of data that must be moved: 8 registers, each 96 bits long, requires 24 memory cycles to save them, and another 24 cycles to re-load them. Newer architectures, for example, one that supports hardware matrix multiply, can have even more state. I claim that a lot of this state does not change between switch-in and switch-out. I propose hardware support to efficiently save and restore only the part of the state that was used: a modified-bit on each register, and selective disabling of hardware function units. Modified-bits on each register lets the operating system save only those registers that have been changed since switch-in. Selective disabling of function units lets the operating system defer loading that unit's state until it is needed. If a functional unit goes unused between switch-in and the subsequent switch-out, its state will not have been loaded nor saved.

- **Faster byte-operations.** Many I/O-related functions tend to be byte-oriented, whereas CPU and memory tends to be word-oriented. This means it costs no more to fetch a full 32-bit word as it does to fetch a byte. We can take advantage to this with two new instructions: "load-4-bytes" and "store-4-bytes". These would move a word from memory into four registers, one byte to a register. The program can then operate on the four bytes in registers without referencing memory again.

  Another suggestion, probably less useful, is a "carry-suppress" option for addition, to suppress carry-out at byte-boundaries, allowing four additions or subtractions to take place simultaneously on four bytes packed into a 32-bit integer. I foresee the primary use of this to be in low-level graphics routines that deal with 8-bit pixels.

- **Improved bit-wise operation support.** The current complement of bitwise-logical operations and shifts are already pretty good, what is lacking is a perfect shuffle of bits in a register. This is very useful for bit-mapped graphics operations, particularly things like bit-matrix transpose, which is heavily used when unpacking byte-wide pixels into separate bit-planes, as is required by certain framebuffer architectures.

7.5 Other Opinions

In any line of research, there are often significant differences of opinion over what assumptions and ideas are good ones. Synthesis is no exception, and it has its share of critics. I feel it is my duty to point out where differences of opinion exist, to allow readers to come to their own conclusions. In this section, I try to address some of the more frequently raised objections regarding Synthesis, and rebut those that are, in my opinion, ill-founded.

> **Objection 1:** "How much of the performance improvement is due to my ideas, and how much is due to writing in assembler, and tuning the hell out of the thing?"

This is often asked by people who believe it to be much more of the latter and much less of the former.

Section 3.3 outlined several places in the kernel where code synthesis was used to advantage. For data movement operations, it showed that code synthesis achieves 1.4 to 2.4 times better performance than the best assembly-language implementation not using code synthesis. For more specialized operations, such as context switching, code synthesis delivers as much as 10 times better performance. So, in a terse answer to the question, I would say "40% to 140%".

But those figures do not tell the whole story. They are detailed measurements, designed to compare two versions of the same thing, in the same execution environment. Missing from those measurements is a sense of how the interaction between larger pieces of a program changes when code synthesis is used. For example, in that same section, I show that a procedural implementation of "putchar" using code synthesis is slightly faster than the C-language "putchar" macro, which is in-line expanded into the user's code. The fact that enough savings could be had through code synthesis to more than amortize the cost of a procedure call — even in a simple, not-easily-optimized operation such as "putchar" — changes the nature of how data is passed between modules in a program. Many modules that process streams of data are currently written to take as input a buffer of data and produce as output a new buffer of data. Chaining several such modules involves calling each one in turn, passing it the previous module's output buffer as the input. With a fast "putchar" procedure, it is no longer necessary to pass buffers and pointers around; we can now pass the address of the downstream module for "putchar," and the address of the upstream module for "getchar." Each module makes direct calls to its neighbors to get the data, eliminating the memory copy and all consequent pointer and counter manipulations.

> **Objection 2:** "Self-modifying data structures are troublesome on pipelined machines, and code generation has problems with machines that don't allow fine-grained control of the instruction cache. In other words, Synthesis techniques are dependent on hardware features that aren't present in all machines, and, worse, are becoming increasingly scarce."

Pipelined machines pose no special difficulties because Synthesis does not modify instructions ahead of the program counter. Code modification, when it happens, is restricted to patching just-executed code, or unrelated code. In both cases, even a long instruction pipeline is not a problem.

The presence of a non-coherent and hard-to-flush instruction cache is the harder problem. By "hard-to-flush," I mean a cache that must be flushed whole instead of line-at-a-time, or one that cannot be flushed in user mode without taking a protection exception. Self-modifying code is still effective, but such a cache changes the breakeven point when it becomes more economical to interpret data than to modify code. For example, conditions that change frequently are best represented using a boolean flag, as is usually done. But for conditions that are tested much more frequently than changed, code modification remains the method of choice. The cost of flushing the cache determines at what ratio of testing to modification the decision is made.

Relief may come from advances in the design of multiprocessors. Recent studies show that, for a wide variety of workloads, software-controlled caches are nearly as effective as fully coherent hardware caches and much easier to build, as they require no hardware [23] [2]. Further extensions to this idea stem from the observation that full coherency is often not necessary, and that it is beneficial to rely on the compiler to maintain coherency in software only when required [2]. This line of thinking leads to cache designs that have the necessary control to efficiently support code-modifying programs.

But it is true that the assumption that code is read-only is increasingly common, and that hardware designs are more and more using this assumption. Hardware manufacturers design according to the needs of their market. Since nobody is doing runtime code generation, is it little wonder that it is not well supported. But then, isn't this what research is for? To open people's eyes and to point out possibilities, both new and overlooked. This dissertation points out certain techniques that increase performance. It happens that the techniques are unusual, and make demands of the hardware that are not commonly made. But just as virtual memory proved to be a useful idea and all new processors now support memory management, one can expect that if Synthesis ideas prove to be useful, they too will be better supported.

> **Objection 3:** "Does this matter? Hardware is getting faster, and anything that is slow today will probably be fast enough in two years."

Yes, it matters!

There is more to Synthesis than raw speed. Cutting the cost of services by a factor of 10 is the kind of change that can fundamentally alter the structure of those services. One example is the PLL-based process scheduling. You couldn't do that if context switch was expensive — driving the time way below one millisecond is what made it possible to move to a radically different scheduler, with nice properties, besides speed.

For another example, I want to pose a question: if threads were as cheap as procedure calls, what would you do with them? One answer is found in the music synthesizer applications that run on Synthesis. Most of them create a new thread *for every note!* Driving the cost of threads to within a few factors of the cost of procedure call changes the way applications are structured. The programmer now only needs to be concerned that the waveform is synthesized correctly. The Synthesis thread scheduler ensures that each thread gets enough CPU time to perform its job. You could not do that if threads were expensive.

Finally, hardware may be getting faster, but it is not getting faster fast enough. Look at the window-system figures given in Table 7.2. Synthesis running on 5-year-old hardware technology outperforms conventional systems running on the latest hardware. Even with faster hardware, it is not fast enough to overtake Synthesis.

> **Objection 4:** "Why is Synthesis written in assembler? How much of the reason is that you wanted no extraneous instructions? How much of the reason is that code synthesis requires assembler? How much of Synthesis could be re-written in a high-level language?"

Section 7.4.1 answers these questions in detail.

8

Conclusion

A dissertation is never finished.
You just stop writing.
— Everyone with a Ph.D.

This dissertation has described Synthesis, a new operating system kernel that provides fundamental services an order of magnitude more efficiently than traditional operating systems.

Two options define the direction in which research of this nature may proceed. Firstly, an existing system may be adopted as a platform upon which incremental development may take place. Studying a piece of an existing system limits the scope of the work, ensures that one is never far from a functioning system that can be measured to guide development, and secures a preexisting base of users, upon completion. On the down side, such an approach may necessarily limit the amount of innovation and creativity brought to the process and possibly carry along any preexisting biases built in by the originators of the environment, reducing the impact the research might have on improving overall performance.

Alternatively one can start anew. Such an effort removes the burden of preexisting decisions and tradeoffs, and allows use of knowledge and hindsight acquired from past systems to avoid making the same mistakes. The danger, however, is of making new, possibly fatal mistakes. In addition, so much valuable time can be spent building up the base and re-inventing wheels, that little innovation takes place.

I have chosen the second direction. I felt that the potential benefits of an important breakthrough far outweighed the dangers of failure. Happily, I believe that a positive outcome may be reported. I would like to summarize both the major contributions and shortcomings of this effort.

A basic assumption of this research effort has been that low overhead and low latency are important properties of an operating system. Supporting this notion is the prediction that as distributed computing becomes ubiquitous, responsiveness and overall performance will suffer at the hands of the high overhead and latency of current systems. Advances in networking technology, impressive as they are, will bear little fruit unless operating systems software is efficient enough to to make full use of the higher bandwidths. Emerging application areas such as interactive sound, video, and the future panoply of interface technologies subsumed under the umbrella of “multi-media” place strict timing requirements on operating system services — requirements that existing systems have difficulty meeting, in part, because of their high overhead and lack of real-time support.

The current leading suggestion to address the performance problems is to move function out of the kernel, thus avoiding crossing the kernel boundary and allowing customization of traditional kernel services to individual applications. Synthesis shows that it is not necessary to accept that kernel services will be slow, and to find work-arounds to them, but rather that it is possible to provide very efficient kernel services. This is important, because ultimately communications with the outside world still must go through the kernel.

With real-time support and an overhead factor ten times less than that of other systems, Synthesis may be considered a resounding success. Four key performance-improving dynamics differentiate Synthesis:

- Large scale use of run-time code generation.
- Quaject-oriented kernel structure.
- Lock-free synchronization.
- Feedback-based process scheduling.

Synthesis constitutes the first large-scale use of run time code generation to specifically improve operating system performance. Chapter 3 demonstrates that common operating system functions run five times faster when implemented using runtime generated code than a typical C language implementation, and nearly ten times faster when compared with the standard UNIX implementation. The use of run time code generation not only improves the performance of existing services, but allows for the addition of new services without incremental systems overhead.

Further differentiating Synthesis is its novel kernel structure, based around "quajects," forming the building blocks of all kernel services. In many respects, quajects resemble the objects of traditional Object-Oriented programming, including data encapsulation and abstraction. Quajects differ, however, in four important ways:

- A procedural rather than message-based interface.
- Explicit declaration of exceptions and external calls.
- Runtime binding of the external calls, and
- Implementation using runtime code generation.

By making explicit the quaject's exceptions and external calls, the kernel may dynamically link quajects. Rather than providing services monolithically, Synthesis builds them through the use of one or more quajects eventually comprising the user's thread. This binding takes place dynamically, at runtime, allowing for both the expansion of existing services and for an enhanced capability for creating new ones. The traditional distinction between kernel and user services becomes blurred, allowing for applications' direct participation in the delivery of services. This is possible because a quaject's interface is extensible across the protection boundaries which divide applications from the kernel and from each other. Such an approach enjoys a further advantage: preserving the partitioning and modularity found in traditional systems centered around user-level servers, while bettering the higher performance levels of the monolithic kernels which, while fast, are often difficult to understand and modify.

The code generation implementation and procedural interface of quajects enhances performance by reducing argument passing and enabling in-line expansion of called quajects into their caller to happen at runtime. Quaject callentries, for example, require no "self" parameter, since it is implicit in their runtime-generated code. This shows, through quajects, that a highly efficient object-based system is possible.

A further research contribution of Synthesis is to demonstrate that lock-free synchronization is a viable, efficient alternative to mutual exclusion for the implementation of multiprocessor kernels. Mutual exclusion and locking, the traditional forms of synchronization, suffer from deadlock and priority inversion. Lock-free synchronization avoids these problems. But until now, there has been no evidence that a sufficiently rich set of concurrent data structures could be implemented efficiently enough using lock-free synchronization to support a full operating system. Synthesis successfully implements a sufficient number of concurrent, lock-free data structures using one and two-word *Compare-&-Swap* instructions. The kernel is then carefully structured using only those data structures in the places where synchronization is needed. The lock-free concurrent data structures are then demonstrated to deliver better performance than locking-based techniques, further supporting my thesis for hardware that supports *Compare-&-Swap*.

New scheduling algorithms have been presented which generalize scheduling from job assignments as a function of time, to functions of data flow and interrupt rates. The algorithms are based upon feedback, drawing from control systems theory. The applications for these algorithms include support for real-time data streams and improved support for dealing with the flow of time. These applications have been illustrated by numerous descriptions of real-time sound and signal processing programs, a disk-sector finder program, and a discussing on clock synchronization.

It is often said that good research raises more questions than it answers. I now explore some open questions, point out some of Synthesis' shortcomings, and suggest directions for future work.

Clearly, we need better understanding of how to write programs that create code at run time. A more formal model of what it is and how it is used would be helpful in extending its applicability and in finding ways to provide a convenient interface to it.

Subsidiary to this, a good cost/benefit analysis of runtime code generation is lacking. Because Synthesis is the first system to apply run time code generation on a large-scale basis, a strategic goal has simply been to get it to work and show that performance benefits do exist. Accordingly, the emphasis has been in areas where the benefits have been deemed to be greatest and where code generation was easiest to do, both in terms of programming difficulty and in CPU cycles. Intuition has been an important guide in the implementation process, resulting in an end product which performs well. It is not known is how much more improvement is possible. Perhaps applying runtime code generation more vigorously or structuring things in a different way will yield even greater benefits.

Unfortunately, there is no high-level language available making programs that use run time code generation easy to write and at the same time, portable. Aside from the obvious benefit of making the technique more accessible to all members of the profession, a better understanding of the benefits of runtime code generation will sure accrue from developing such a language.

An interesting direction to explore is to extend the ideas of runtime code generation to runtime reconfigurable hardware. Chips now exist whose function is "programmed" by downloading strings of bits that configure the internal logic function blocks and the routing of signals between blocks. Although the chips are generally programmed once, upon initialization, they could be reprogrammed at other times, optimizing the hardware as the environment changes. Some PGAs could be set aside for computations purposes: functions such as permuting bit vectors can be implemented much more efficiently with PGA hardware than in software. Memory operations, such as a fast memory-zero or fast page copy could be implemented operating asynchronously with the main processor. As yet unanticipated functions could be configured as research identifies the need. A machine architecture is envisaged having no I/O device controllers at all — just a large array of programmable gate array (PGA) chips wired to the processor and to various forms of I/O connectors. Clearly, the types of I/O devices which the machine supports is a function of the bit patterns loaded into its PGAs, rather than the boards which alternatively would be plugged into its backplane. This is highly advantageous, for as new devices need to be supported, there is no need for new boards and the attendant expense and delay of acquiring them.

Currently, under Synthesis, users cannot define their own services. Quaject composition is a powerful mechanism to define and implement kernel services, but this power has not yet been made accessible to the end user. At present, all services that exist do so because located somewhere in the kernel is a piece of code which knows which quajects to create and how to link them in order to provide each service. It would be better if this were not hard coded into the kernel, but made user-accessible via some sort of service description language. To support such a language, the quaject type system would need to be extended to provide runtime type checking, which is currently lacking.

Another open question concerns the generality of lock-free synchronization. Lock-free synchronization has many desirable properties as discussed in this dissertation. Synthesis has demonstrated that lock-free synchronization is sufficient for the implementation of an operating system kernel. "Is this accomplished at the expense of required generality" is a question in need of an answer. Synthesis isolates all synchronization to a handful of concurrent data structures which have been shown to have an efficient lock-free implementation. Nonetheless, when lock-free data structures are used to implement systems policy, a loss of generality or efficiency may result. Currently, Synthesis efficiently supports a scheduling policy only if it has an efficient lock free implementation. One approach to this issue is to add to the list of efficient lock-free data structure implementations, thereby expanding the set of supportable policies. Another research direction is to determine when other synchronization methods are necessary so that a policy may be followed literally, but also when the policy can be modified to fit an efficient lock-free implementation. In addition, determining how best to support processors without a *Compare-&-Swap* instruction would be valuable.

The behavior of feedback-based, fine grained scheduling has not been fully explored. When the measurements and adjustments happen at regular intervals, the schedule can be modeled as a linear discrete time system and Z-transforms used to prove stability and convergence. In the general case, measurements and adjustments can occur at irregular intervals because they are scheduled as a function of previous measurements. It is not known whether this type of scheduler is stable for all work load conditions. While empirical observations of real-time signal processing applications indicate that the scheduler *is* stable under many interesting, real-life workloads, it would be nice if this could be formally proven.

The current 68030 assembly language implementation limits Synthesis' portability. The amount of code is not inordinately large (20,000 lines) and much of it is macro invocations, rather than machine instructions, so an assembler-level port might not be nearly as difficult as it might first appear. A high level language implementation would be better. An open question is the issue of runtime code generation. While one could create code which inserts machine-op codes into memory, the result would be no more portable than the current assembly language version. A possible approach to this problem would be to abstract as many runtime code generation techniques as possible in simple, machine-independent extensions to an existing programming language such as C. Using the prototypic language and making performance comparisons along the way would go far toward identifying which direction the final language should take.

While Synthesis holds out enormous promise, its readiness for public release is retarded by the following factors:

- Known bugs need to be removed.
- The window system is incomplete, and lacks many 2-D graphics primitives and mouse tracking.
- The virtual memory model is not fully developed, and the pager interface should be exported to the user in order to enhance its utility.
- The network driver works, but no protocols have as yet been implemented.

All of these enhancements can be made without risk to either the measurements presented in this dissertation, or to the speed and efficiency of the primitive kernel. My confidence rests partly because the significant execution paths have been anticipated and measured, and partly from past experience, when the much more significant functionality of multiprocessor support, virtual memory, ethernet driver, and the window system were added to the then primitive kernel without slowing it down.

I want to conclude by emphasizing that although this work has been done in the context of operating systems, the ideas presented in this dissertation — runtime code generation, quaject structuring, lock-free methods of synchronization, and scheduling based on feedback — can all be applied equally well to improving the performance, structuring, and robustness of user-level programs. The open questions, particularly those regarding runtime code generation, may make this difficult at times; nevertheless the potential is there.

While countless philosophers throughout Western Civilization have all proffered advice against the practice of predicting the future, most have failed to resist the temptation. While computer scientists shall most likely fare no better at this art, I believe that Synthesis brings to the surface an operating system of elegance and efficiency as to accelerate serious consideration and development of multiple microprocessor machine environments, particularly those allied with multi-media and communications. In short, Synthesis and the concepts upon which it rests are not likely to be eclipsed by any major occurrence on the horizon of technology any time soon.


Qua!

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the 1986 Usenix Conference*, pages 93–112. Usenix Association, 1986.

[2] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. In *The 18th Annual International Symposium on Computer Architecture*, volume 19, pages 298–308, 1991.

[3] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, Pacific Grove, CA, October 1991. ACM.

[4] James Arleth. A 68010 multiuser development system. Master's thesis, The Cooper Union for the Advancement of Science and Art, New York City, 1984.

[5] Brian N. Bershad, Edward D. Lazowska, Henry M. Levy, and David B. Wagner. An Open Environment for Building Parallel Programming Systems. In *Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 1–9, New Haven, Connecticut (USA), July 1988. ACM SIGPLAN.

[6] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *Proceedings of the First Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 78–86. ACM, September 1986.

[7] D.L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, May 1990.

[8] Min-Ih Chen and Kwei-Jay Lin. A Priority Ceiling Protocol for Multiple-Instance Resources. In *IEEE Real-Time Systems Symposium*, San Antonio, TX, December 1991.

[9] David Cheriton. An Experiment Using Registers for Fast Message-Based Interprocess Communication.

ACM SIGOPS Operating Systems Review, 18(4):12–20, October 1984.

[10] F. Christian.
Probabilistic Clock Synchronization.
Technical Report RJ6432 (62550) Computer Science, IBM Almaden Research Center, September 1988.

[11] H.M. Deitel.
An Introduction to Operating Systems.
Addison-Wesley Publishing Company, second edition, 1989.

[12] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean.
Using Continuations to Implement Thread Management and Communication in Operating Systems.
In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, Pacific Grove, CA, October 1991. ACM.

[13] J. Feder.
The Evolution of Unix System Performance.
AT&T Bell Laboratories Technical Journal, 63(8):1791–1814, October 1984.

[14] P.M. Herlihy.
Wait-Free Synchronization.
ACM Transactions on Programming Languages and Systems, 13(1), January 1991.

[15] Neil D. Jones, Peter Sestoft, and Harald Sondergaard.
Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation.
Lisp and Symbolic Computation, 2(9-50):10, 1989.

[16] David Keppel, Susan J. Eggers, and Robert R. Henry.
A Case for Runtime Code Generation.
Technical Report UW CS&E 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991.

[17] B.D. Marsh, M.L.Scott, T.J.LeBlanc, and E.P.Markatos.
First-Class User-Level Threads.
In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, Pacific Grove, CA, October 1991. ACM.

[18] H. Massalin and C. Pu.
Threads and Input/Output in the Synthesis Kernel.
In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Arizona, December 1989.

[19] Henry Massalin.
A 68010 Multitasking Development System.
Master's thesis, The Cooper Union for the Advancement of Science and Art, New York City, 1984.

[20] Motorola.
MC68881 and MC68882 Floating-Point Coprocessor User's Manual.
Prentice Hall, Englewood Cliffs, NJ, 07632, 1987.

[21] Motorola.
MC68030 User's Manual.
Prentice Hall, Englewood Cliffs, NJ, 07632, 1989.

[22] J. Ousterhout.
Why Aren't Operating Systems Getting Faster as Fast as Hardware.
In *USENIX Summer Conference*, pages 247–256, Anaheim, CA, June 1990.

[23] Susan Owicki and Anant Agarwal.
Evaluating the Performance of Software Cache Coherence.
In *Proceedings of the 3rd Symposium on Programming Languages and Operating Systems*. ACM, 1989.

[24] R. Pike, D. Presotto, K. Thompson, and H. Trickey.
Plan 9 from Bell Labs.
Technical Report CSTR # 158, AT&T Bell Labs, 1991.

[25] C. Pu, H. Massalin, and J. Ioannidis.
The Synthesis Kernel.
Computng Systems, 1(1):11–32, Winter 1988.

[26] J.S. Quarterman, A. Silberschatz, and J.L. Peterson.
4.2BSD and 4.3BSD as Examples of the Unix System.
ACM Computing Surveys, 17(4):379–418, December 1985.

[27] D. Ritchie.
A Stream Input-Output System.
AT&T Bell Laboratories Technical Journal, 63(8):1897–1910, October 1984.

[28] D.M. Ritchie and K. Thompson.
The Unix Time-Sharing System.
Communications of ACM, 7(7):365–375, July 1974.

[29] J.A. Stankovic.
Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems.
IEEE Computer, 21(10):10–19, October 1988.

[30] M. Stonebraker.
Operating System Support for Database Management.
Communications of ACM, 24(7):412–418, July 1981.

[31] Sun Microsystems Incorporated, 2550 Garcia Avenue, Mountain View, California 94043, 415-960-1300.
SunOS Reference Manual, May 1988.

[32] Peter Wegner.
Dimensions of Object-Based Language Design.
In Norman Meyrowitz, editor, *Proceedings of the OOPSLA'87 conference*, pages 168–182, Orlando FL (USA), 1987. ACM.

[33] Mark Weiser, Alan Demers, and Carl Hauser.
The Portable Common Runtime Approach to Interoperability.

In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114–122, Litchfield Park AZ (USA), December 1989. ACM.

[34] W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, , and F. Pollack. Hydra: The Kernel of a Multiprocessing Operating System. *Communications of ACM*, 17(6):337–345, June 1974.

Appendix A

Unix **Emulator Test Programs**

```
#define N       500000
int     x[N];
main() {
        int     i;
        for(i=5; i--; )
                g();
        printf("%d\n%d\n", x[N-2], x[N-1]);
}

g() {
        int     i;
        x[0] = x[1] = 1;
        for(i=2; i<N; i++)
                x[i] = x[i-x[i-1]] + x[i-x[i-2]];
}
```

Figure A.1: Test 1: Compute

```
#define N       1024    /* or 1 or 4096 */
char    x[N];
main()
{
        int     fd[2],i;
        pipe(fd);
        for(i=10000; i--; ) {
                write(fd[1], x, N);
                read(fd[0], x, N);
        }
}
```

Figure A.2: Test 2, 3, and 4: Read/Write to a Pipe

```
#include <sys/file.h>
#define Test_dev        "/dev/null"     /* or /dev/tty */
main()
{
        int     f,i;
        for(i=10000; i--; ) {
                f = open(Test_dev, O_RDONLY);
                close(f);
        }
}
```

Figure A.3: Test 5 and 6: Opening and Closing

```
#include <sys/file.h>
#define N 1024
char x[N];
main()
{
        int     f,i,j;
        f = open("file", O_RDWR | O_CREAT | O_TRUNC, 0666);
        for(j=1000; j--; ) {
                lseek(f, 0L, L_SET);
                for(i=10; i--; )
                        write(f, x, N);
                lseek(f, 0L, L_SET);
                for(i=10; i--; )
                        read(f, x, N);
        }
        close(f);
}
```

Figure A.4: Test 7: Read/Write to a File