An Overview of The Synthesis Operating System

Calton Pu and *Henry Massalin*

Department of Computer Science
Columbia University
New York, NY 10027

Technical Report No. CUCS-470-89

Abstract

An operating system (OS) maps a model of computation, defined by its kernel interface, onto the underlying hardware. A simple and intuitive model of computation makes it easy for programmers to write applications. An efficient implementation of the mapping makes the applications run fast. Typical OS interfaces for von Neumann hardware include processes (CPU), address spaces (memory), and I/O devices (I/O). An OS for distributed and multiprocessor systems must support parallel processing and inter-process communications.

Designed for the distributed and multiprocessor systems of today and tomorrow, Synthesis is an example of the new generation of OS's. The Synthesis model of computation based on macro dataflow makes parallel programming easy. The Synthesis implementation uses innovative ideas from areas as different as compilers and control systems. Kernel Code Synthesis, Reduced Synchronization, Fine-Grain Scheduling, and Valued Redundancy provide high performance. Emulation of guest OS's in Synthesis allow existing software written for other OS's to run on Synthesis with little or no modification. Using hardware and software emulating a SUN-3/160 running SUNOS, the current version of Synthesis kernel achieves several times to several dozen times speedup for UNIX kernel calls.

[0]This work is partially funded by the New York State Center for Advanced Technology on Computer and Information Systems under the grant NYSSTF CU-0112580, by the AT&T Foundation under a Special Purpose Grant, and by the National Science Foundation under the grant CDA-88-20754. We gladly acknowledge the hardware parts contributed by AT&T, Hitachi, IBM, and Motorola.

Contents

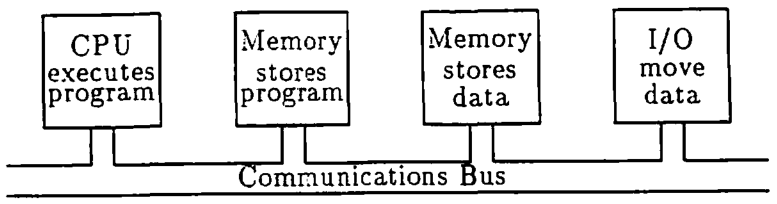

Figure 1: A von Neumann Computer

1 Models of Computation and Operating Systems

1.1 Models of Computation

We write programs using a programming language such as FORTRAN. A programming language is an example of a *model of computation*, the primary conceptual tool in computer science to describe computations. Other examples of models of computation range from the very abstract, for example automata and Turing Machines in theoretical computer science, to the very concrete, for example the instruction set of actual microprocessors.

In the late 40's John von Neumann proposed a model of computer architecture. Now known as the von Neumann architecture, the model consists of a *central processing unit* (CPU), which executes a program stored in *memory*. The overwhelming majority of the computers today are instances of the von Neumann architecture, including all the mainframes, minicomputers, and microprocessors. In figure 1 we see the main components of the von Neumann architecture, connected together with a communications bus.

The von Neumann architecture is an abstract model. A real machine such as the Intel 80386 microprocessor is a concrete instantiation of the abstract model, specified by the computer's *instruction set*, i.e., all the instructions that the CPU knows how to execute. The instruction set is also known as the machine language, since we use it to write programs that are directly executable by the CPU.

Writing machine language programs to run on a bare machine is notoriously hard. Most programmers have never handled the bits and bytes necessary to make such programs work. Instead, they use help from programming languages (either at high level as FORTRAN or at low level such as assembly language) and operating systems (OS). To bridge the gap between the bare machine and the applications, the OS presents a high-level virtual machine to the programmers, hiding the tedious details of bit handling. In analogy to the computer architecture, OS kernel calls (a.k.a. system calls) are the instructions of the high-level virtual machine. The virtual machine specified by the kernel interface is analogous to the instruction set.

Although the OS designers can choose from many different models of computation for

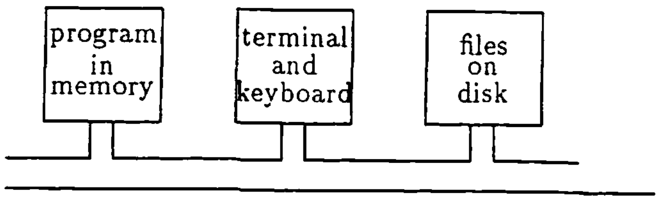

Figure 2: 1-1 Mapping (DOS)

their kernel interfaces, most prevailing OS's have virtual machines that are instances of the von Neumann model. There are many reasons for this. The most important external reason is that OS users, especially the programmers, are used to the von Neumann model embodied in procedural programming languages. The most important internal reason is the ease of implementing an efficient OS. It is usually easier to find a simple mapping between two similar models (when both are von Neumann) than very different models.

In summary, the OS maps its own model of computation (the kernel interface) into the hardware model (the instruction set). We classify the OS implementation into three important classes: from one virtual machine to one CPU (1-1 mapping), from N virtual machines to one CPU (N-1 mapping), and from N virtual machines to M central processing units (N-M mapping). We will consider each in turn.

1.2 The 1-1 Model

MS-DOS [3] is the OS with the most copies sold in history. Many thousands of application programs have been written to run on DOS, including word processors, electronic spreadsheets, graphics display packages, and database managers. It is curious that in many aspects the DOS resembles the primitive systems of the 50's, before the IBM /360 Operating System made the term OS popular.

Historically, the pioneering computers of the 50's either ran bare machine without any systems software or had very simple support systems called I/O executives. A typical executive would be a deck of punched cards that can assemble the program following it and load the resulting assembled code into the memory and execute the program. Sometimes the executive also provided limited I/O support, such as easy access to either punched or magnetic tapes. In comparison, DOS has sophisticated I/O support, including terminal I/O and the disk file system (hence its name: Disk Operating System). However, in terms of running programs, DOS can run only one program at a time, just like the early I/O executives.

We call this kind of OS, exemplified by DOS, the 1-1 mapping OS. The first 1 means that the model of computation presented by the OS contains only one single process. The second 1 means that the OS is intended to run on a single-processor machine.

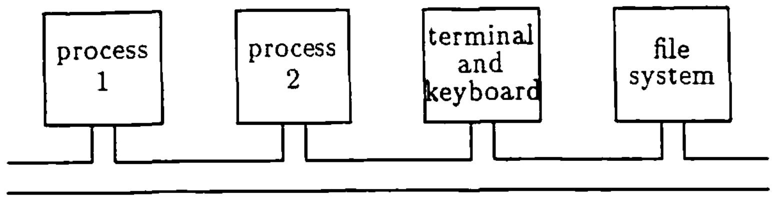

Figure 3: N-1 Mapping (UNIX)

An 1-1 mapping is simple. Figure 2 shows the similarity between the DOS model and the von Neumann model. Since there is only one process that occupies all the memory and runs on the physical CPU all the time, once the program is loaded the OS is only concerned with servicing I/O requests. All the I/O external operations and internal handling are done by subroutines that encapsulate the hardware.

1.3 The N-1 Model

The next class of OS in complexity, the N-1 mapping OS, is characterized by multiprogramming support, as in UNIX [9]. The N says that we are mapping N processes to a single processor. Figure 3 shows the UNIX N-1 model of computation. Even though the difference between the two figures (2 and 3) appears slight, i.e., the addition of one process, the impact on the system and the users is great.

Having N processes sharing the same physical machine introduces new opportunities for better hardware utilization. For example, the 1-1 mapping makes the CPU stop whenever there is an I/O activity. An N-1 mapping could run another process while the original process is waiting for the I/O operation to complete.

On the other hand, the sharing also introduces several implementation difficulties for the N-1 mapping. First, we have to isolate the memory allocated to each process from the other processes. Otherwise, their execution may interfere with each other. Second, the OS has to switch the CPU between the processes when appropriate. Deciding when the CPU should run which process is called *scheduling*, which is a problem absent in the 1-1 mapping. Like memory and CPU, access to other shared resources such as disk space also need protection and synchronization. Solving these problems and providing other services make a big difference between typical 1-1 mappings and N-1 mappings. For example, the size of a typical UNIX kernel executable is measured in hundreds of kilobytes, compared to a typical DOS kernel measured in tens of kilobytes.

The UNIX virtual machine is called a *process*, which has exactly one thread of control and one complete address space. From the model of computation point of view, the UNIX process is a uniprocessor. UNIX I/O is based on the idea of streams. Data are moved through byte streams connected I/O devices and processes. When two processes want to exchange data, they open a channel called a pipe. The writer process then sends data to the pipe and the reader process receive them. The UNIX process is probably the most elegant model of computation among the current OS's. For example, a UNIX filter, which reads from the standard input and writes to the standard output, is essentially a finite state machine (probably the best-known model of computation in computer science).

Comparng DOS with UNIX we can see the difficulty of extending an 1-1 mapping into an N-1 mapping. Several multitasking extensions to DOS (such as Desqview) exist. They work by protecting the memory locations referring to screen and I/O. When a program attempts to access these areas (to perform I/O), an exception occurs and the multitasking kernel simulates the memory mapped operation on behalf of the user program. The convenience comes at the cost of speed. Originally application programs made direct memory mapped I/O for speed, but now they are actually slower than the OS kernel calls that the programmers were trying to bypass. This case underscores an important requirement of an OS: the interface must be general and the implementation must be fast. Were the DOS I/O as fast as the best programmers could do at the user level, they would have not been tempted to access the memory mapped I/O directly. Microsoft and IBM decided to move on to OS-2 as the N-1 mapping for the PS/2 generation of personal computers, instead of trying to introduce multiprogramming into DOS.

1.4 The N-M Model

In real multiprocessing and distributed processing, however, applications require communications between the processors. The two main models of communications between processors are called *shared memory* and *message-passing*. In the shared memory model, processors have access to a common memory. So one processor may write to the shared memory, then another may read from it. In the message-passing model, processors send messages to each other when they want to exchange information.

Both models have natural analogs in OS's. An OS supporting shared memory allows different processes to access overlapping pieces of memory. This is usually implemented by mapping part of each process's memory to the same physical location. An OS supporting message-passing provides primitives for processes to exchange messages. This is usually implemented by copying the message content from the sender's memory space to the receiver's. In the OS, the communications problem is called inter-process communications (IPC).

The OS's that present a natural multiprocessing model of computation to the programmers are in the N-M class. Many efforts attempted to make UNIX an N-M mapping. Of the two classic variants of UNIX, the Berkeley Software Distribution (BSD) 4.2 and 4.3 support message-passing through socket while the AT&T System V and compatible systems support shared memory. A more modern strain - the SUNOS - implements both. However, the problem is far from solved. Both the socket and the shared memory are hard to use and carry significant system overhead. Finally, another variety - Mach [1] from CMU - introduced the light-weight threads that share the same address space within a process and have relatively low overhead. Some of the commercially available multiprocessors (such as the Sequent Symmetry and Encore Multimax) run UNIX variants that follow these solutions.

Each attempted extension of UNIX represents an incremental improvement. For example, light-weight threads make parallel processing within a process affordable, since the communications between threads through shared memory is fast. However, light-weight threads do not solve the IPC problem at the process level. Also, the introduction of a two-level process hierarchy makes the composition of parallel programs more difficult. Ultimately, it is the original uniprocessor character of the UNIX process that makes parallel processing with UNIX extensions difficult.

Before we discuss the details of OS's designed for multiprocessing and distributed processing, we first introduce their hardware base. Then we describe the desirable properties of such OS's. After that we introduce the Synthesis model of computation (an **N-M** mapping).

2 Multiprocessor and Distributed OS

2.1 Next-Generation Architectures

Although the research on multiprocessor and distributed OS's has been going on for many years, the proliferation of actual parallel and distributed systems started only recently. During the first half of the 1980's, we saw some specialized systems such as Tandem systems for transaction processing and Teradata database machines. During the second half, general-purpose multiprocessor computers with around a dozen of powerful CPUs such as Encore Multimax and Sequent Symmetry became available.

The evolution of wide-area networking happened at about the same time. ARPANET was developed and deployed towards the end of 1970's. During the 1980's, we saw the development of NSFnet and others. Network bandwidth grew slowly at the beginning. It took a decade for one-and-half order of magnitude increase (from the 56 Kilobit/second leased lines in ARPANET to 1.5 Megabit/second T-1 channels in NSFnet). Now we expect a three order of magnitude upgrade (Gigabit/second range) within a few years.

The next-generation architecture we expect to see in a few years will consist of many machines connected through a very fast network. The machines range from PC's to super-workstations to supercomputers. A typical workstation would have a fast CPU at 25 to 50 MIPS (million instruction per second), plenty of memory at 1 to 4 MB (Megabyte) per each MIPS of processing power, and both diversity and abundance in I/O devices, including disk space (20 to 100 times the memory size), color graphics displays, sound and video. A multiprocessor may be a collection of these machines with many sets of CPU and memory on a fast bus.

2.2 Next-Generation Computer Systems

We would want to run a next-generation operating system on top of the next-generation architecture described above. Many desirable properties are missing from our current systems: high performance and availability at low cost, graceful degradation when components break, interoperability with a large set of heterogeneous resources, etc. We will consider each in turn.

The very reason for building parallel computers is performance. We always need more net processing power and parallel processing is an effective way of harnessing computing power at low cost. With additional machines on the network, the next-generation system should provide plenty of computing power in an easy-to-use form. In other words, the system should help the programmers use the many CPUs, solving the communications and synchronization problems with low overhead.

With lots of hardware redundancy, a distributed system has the potential for high availability. Even though there may be no need for replicating everything in such a network, we really want graceful degradation when components break down. Ideally, we want to substitute the broken machines with idle machines, maintaining the system performance and availability. After we have reached full utilization, a node crash would decrease only its contribution to the whole system performance and availability.

An important attribute of a large system is the ability to encapsulate resources at different levels of abstraction. At the high levels, application programs can use the resources easily, regardless of their location, degree of redundancy, and other technical details. We call the high-level abstraction *transparency*. At the low levels, system programs can use the resource with the maximum economy. In Lampson's system design hints [4], this is called "do not hide power". Abstractions at several levels usually are specified as a layered interface.

Finally, connectivity in a large distributed system inevitably results in heterogeneity. Even though systems with different components may not work together as efficiently as a homogeneous system, we still want to communicate with the other systems. This ability is called *interoperability*. The main reason for interoperability is that some resources are not available elsewhere, so we have to go to a particular place to get them.

2.3 The Synthesis Model of Computation

Synthesis is a multiprocessor and distributed OS, designed to achieve the goals enumerated in the previous section for an N-M mapping. For system programmers and compiler writers, we want high performance and predictability (for real-time applications). For application programmers, we want an intuitive model of computation that is easy to use for writing parallel and distributed programs. Finally, for the general user we want the support to run existing software.

The Synthesis model of computation contains threads of execution, memory protection

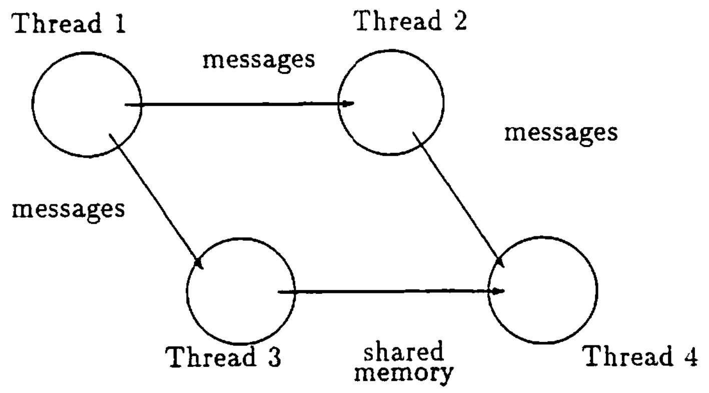

Figure 4: N-M Mapping (Synthesis)

boundaries, and I/O devices. Each thread is an instance of the von Neumann model. To support parallel processing in a shared-memory machine, threads may share memory. To support distributed processing, the threads of execution form a directed graph, in which the nodes are threads and the arcs are data flow channels.

Threads have a very simple behavior. Once created, we can `start` and `stop` the thread, or `step` through single instructions in the thread. The thread also contains the `signal` procedures, the error trap handlers, and the body of I/O system calls such as `read` and `write`. From the kernel point of view, the thread also contains its own context-switch-in and context-switch-out procedures that dispatch the thread.

Currently Synthesis supports real memory only. At its creation, memory is allocated and allows direct read and write access. When destroyed, the memory is deallocated and returned to the kernel free memory list. Some important memory management issues such as protection await the Synthesis port to the NeXT machine, which includes a memory management unit (MMU) in the 68030 CPU.

Synthesis threads are threads of execution, like UNIX processes. We should note that the Synthesis kernel also uses the same threads when concurrency in the kernel is desired. Threads execute programs in a quaspace (*quasi address space*), which also store data. Finally, I/O devices move data between threads, including files and messages.

On one physical node. all the Synthesis quaspaces are subspaces of one single address space, defined by the CPU architecture (e.g., with a 32-bit microprocessor we have a 32-bit address space). The kernel blanks out the part of the address space that each quaspace is not supposed to see. Since they are parts of the same address space, it is easy to share memory between quaspaces by setting their address mappings. The current implementation of the Synthesis kernel does not support virtual memory.

Synthesis I/O devices are byte streams. All the devices support the basic `read` and `write` operations on byte streams. The byte stream model of I/O is implemented as a pipeline of device servers. Each physical I/O device has a raw device server, which may be connected to higher level device servers upstream the pipeline. For example, the TTY device has a raw device server connected to the cooked server, which filters the byte stream according to the editing control characters such as backspace.

3 The Ideas in Synthesis

In the previous section, we have described the concept of model of computation in OS's. Now we proceed to present the innovative ideas in the design and implementation of Synthesis. We refer the interested reader to technical discussions of these ideas in other papers: kernel code synthesis [8], reduced synchronization [6], fine-grain scheduling [5], valued redundancy [7].

All these ideas improve performance, although some have other good properties as well. Before the introduction of the ideas, we make the fundamental observation in optimization: all the components of a system must be optimized for the whole system to be optimized. Otherwise, the system performance (or availability) is limited by its weakest link. It is in light of this fundamental observation that our relentless pursuit of performance becomes reasonable.

3.1 Kernel Code Synthesis

Traditional OS kernels maintain the system state in data structures such as linked lists. A typical kernel call starts by traversing the appropriate data structures to reach the starting system state, then executes the few machine instructions to perform the actual function of the kernel call. For example, Stonebraker [10] pointed out that UNIX kernel cost to fetch one byte from buffer pool is about 1800 instructions on the PDP-11/70.

The idea of kernel code synthesis is to capture frequently visited system states in small chunks of code. Instead of traversing the data structures, we branch to the synthesized code directly. The term "code synthesis" refers to the process of creating new code at run-time to capture a system state. In this section, we describe three methods to synthesize code: Factoring Invariants, Collapsing Layers, and Executable Data Structures. Following the PDP-11/70, successive generations of UNIX systems have been optimized in several ways. However, as shown by our measurements (Section 4), Synthesis kernel using code synthesis outperforms SUNOS by a factor from several times to several dozens of times. Now we describe the three code synthesis methods in turn.

Borrowing from mathematics, the Factoring Invariants method is based on the observation that a functional restriction is usually easier to calculate than the original function. When a function will be called repeatedly with a constant parameter, we can apply a process called currying, which simplifies the calculation by substituting the constant and carrying out the calculation then. If the simplified function is called many times, we compensate for the cost of currying and win.

The Factoring Invariants method is analogous to constant folding optimization in compiler code generation. But the difference is also significant. Constant folding eliminates static code at compile time. In contrast, Factoring Invariants skips dynamic data structure traversals in addition to bypassing code at execution time. Factoring Invariants can be applied whenever we synthesize code to shorten the resulting execution path.

The Collapsing Layers method is based on the observation that in a layered design, separation between layers are a part of specification, not implementation. In other words, procedure calls and context switches between functional layers can be bypassed at execution time.

For example, in a system implementing the layered OSI interface, a naive implementation of the presentation layer would call the session layer, which calls the data link layer, and so on. For execution, we can run a flat function by eliminating these procedure calls. We call this a vertical layer collapsing, which is analogous to in-line code substitution for procedure calls in compiler code generation. After a successful layer collapsing, typically we can apply Factoring Invariants to reduce the code further. These opportunities are due to the normal redundancy across layers of code. Examples include data copying and parameter marshaling.

An example of the horizontal (context switch) layer collapsing is the pipelined organization of I/O processing. Each stage is a filter in the pipeline, conceptually a thread communicating with its neighbors using queues. If two consecutive filters are finite-state machines, we can collapse the two finite-state machines and the queue into one finite-state machine, eliminating the communication and synchronization overhead. We optimize dedicated I/O devices such as TTY drivers this way.

The Executable Data Structures method is based on the observation that many data structures are traversed in a preferred order. Therefore, we insert the traversal code locally into the data structures, making them *self-traversing*. The hardwired, localized code reduces the traversal overhead to minimum.

Let us consider the simplified example of the active job queue managed by a round-robin scheduler. Each element in the queue contains two short sequences of code: context-switch-out and context-switch-in. The context-switch-out saves the registers and jumps into the next job's context-switch-in routine (in the next element in queue). The context-switch-in restores the registers, installs the address of its own context-switch-out in the timer interrupt vector table, and resumes processing.

An interrupt causing a context switch will trigger the current program's context-switch-out, which saves the current state and branches directly into the next jobs's context-switch-in. 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.

We have used kernel code synthesis in many different places in the Synthesis kernel. In particular. code synthesis helped speed up I/O processing in Synthesis. In this aspect, code synthesis is useful for all three classes of OS's. However, the more layers we have in the OS the more opportunities for Layer Collapsing we can find. Therefore, code synthesis makes the biggest difference in the most sophisticated OS's, i.e. the N-M mappings.

3.2 Reduced Synchronization

In traditional OS's, the kernel call and dispatching/scheduling overhead overshadows the kernel synchronization cost. Therefore, we see traditional kernels using powerful mutual exclusion mechanisms such as semaphores. However, in Synthesis we have used kernel code synthesis to trim kernel calls and context switches. The next bottleneck turned out to be kernel internal synchronization cost, given that the Synthesis kernel is highly parallel. Our answer to this problem consists of three methods that reduce synchronization in the Synthesis kernel: Code Isolation, Optimistic Synchronization, and Procedure Chaining.

Code isolation uses kernel code synthesis to separate and isolate fragments of the data structure manipulation program. The separation eliminates unnecessary synchronization if each fragment operates on its own piece of data. For example, in Synthesis each thread manipulates its own Thread Table Entry (equivalent to the proctable in UNIX). Since nobody else can access the thread's Thread Table Entry, there is no conflict.

Optimistic synchronization assumes that interference between threads is rare and therefore we should shorten the normal non-interfering case. The idea of optimistic validation is to go ahead and make the changes, without locking anything. A check at the end of the update tests whether the assumption of non-interference is true. If the check fails, we rollback the changes and redo the work. We have implemented an optimistic queue which is used extensively in the Synthesis kernel.

Procedure chaining avoids synchronization by serializing the execution of the conflicting threads. Instead of allowing concurrent execution that would have complicated synchronization problems, we chain the new procedure to be executed to the end of the currently running procedure. The handling of a signal in the middle of interrupt handling is a situation in which we use procedure chaining.

Since synchronization is a problem only for N-1 and N-M mappings, reduced synchronization applies only to these two classes. However, since there are more need for synchronization in parallel and distributed processing, Reduced Synchronization is more important for OS's of the N-M class.

3.3 Fine-Grain Scheduling

We call scheduling policies *fine-grain* if they take into account local information in addition to global properties. An example of interesting local information for scheduling is the size of the job's input queue: if it is empty, dispatching the job will merely block for lack of input. Fine-grain scheduling policies are more sensitive to system state changes by definition. In particular, fine-grain scheduling may smooth the data flow in a pipeline of processes, eliminating the bottlenecks.

We call scheduling mechanisms *fine-grain* if their scheduling/dispatching costs approach zero. Traditional scheduling mechanisms have high scheduling and dispatching overhead that discourages frequent scheduler decision making. Consequently, most scheduling algorithms tend 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. Fine-grain scheduling mechanisms turn out to differ significantly from traditional mechanisms.

Fine-grain scheduling policies and mechanisms together are called "fine-grain scheduling", implemented in the Synthesis operating system. Our approach to fine-grain scheduling policies is similar to feedback mechanisms in control systems. We take a job to be scheduled and measure its progress, making scheduling decisions based on the measurements. For example, if the job is "too slow", say its input queue is getting full, we schedule it more often and let it run longer.

The key idea in our fine-grain scheduling policy is based on feedback control, in particular phase locked loop (PLL). A hardware PLL outputs a frequency synchronized with a reference input frequency. Our software analogs of the PLL track a stream of interrupts to generate a new stable source of interrupts locked in step. The reference stream comes from a variety of sources, say an I/O device, e.g., disk index interrupts that occur once every disk revolution, or the interval timer, e.g., at the end of a CPU quantum.

When we use software to implement the PLL idea, we find more flexibility in measurement and control. Unlike hardware PLLs where we always measure phase differences, in software we can measure either the frequency of the input (events/second), or the time interval between inputs (seconds/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 (SLL). An example of SLL that measures and adjusts frequency is a digital oversampling filter program for a CD player, which adjusts the filter I/O rate to match the CD player output. An example of SLL that measures and adjusts time intervals is the disk sector interrupt generator, which decreases the rotational delay in disk access.

As a feedback system, the SLL generates interrupts as a function of its input. As the input interrupt stream changes its frequency or interval, the SLL adjust its output. For example, an SLL that measures intervals will have its natural behavior to maintain the interval between consecutive output interrupts equal to the interval between inputs.

Fine-grain scheduling would be impractical without fast interrupt processing, fast context switch, and low dispatching overhead. 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 (a few tens of microseconds) and context switch for dispatching (less then ten microseconds) form the foundation of the Synthesis implementation of fine-grain scheduling mechanism. In addition, we have very low overhead interrupt processing to allow frequent checks on the job progress and quick adjustments to the scheduling policy.

Like reduced synchronization, fine-grain scheduling applies only to the N-1 and N-M classes of OS's. With more processors and resources to schedule, fine-grain scheduling is more important for N-M mappings.

3.4 Valued Redundancy

The idea of *valued redundancy* increases system performance and availability at low cost by replicating the most valuable objects of the system. Since the redundancy management system explicitly maintains a replicated object's cost/performance ratio as its value, it is clear that valued redundancy maximizes system performance and availability objectives while minimizing maintenance cost.

In the calculation of an object's value, we take into account the object's costs, performance contributions, and performance goals. Important cost parameters include the object creation costs (in terms of resources consumed) and maintenance costs, such as storage, consistent update across copies and garbage collection. The performance contributions include the access patterns such as read and write frequency. Finally, the main performance goals are object access time (averaged over the copies) and the apparent object availability (calculated over the attempted accesses).

These value calculation parameters are closely related. Between the performance goal of an object and its performance contributions, we have a positive correlation. For example, to maximize system throughput we would set high performance goals for frequently accessed objects. Between the object cost and performance goals, we also have a positive correlation. For instance, we would carry more copies of an object that is read by many nodes. Between the object cost and the performance contributions, we have a trade-off. For example, replica update cost will be high for objects that are modified often, while queries do not carry additional object maintenance cost.

With valued redundancy, in principle we can handle specific performance goals for each object. In this paper, we will focus on system-wide performance goals, set in terms of the average behavior. More concretely, we want a system with high throughput, fast response-time, and high availability. Valued redundancy will let us concentrate on the objects that have high performance contributions, replicating them to the extent we can afford. As we will see from our discussion in the next section on the implementation of valued redundancy, the interaction between the components will influence the value calculations strongly. Therefore, it is unlikely that one single value calculation formula will be the best for all the different combinations of algorithms.

Since the current implementation of Synthesis kernel does not yet support distributed processing, we have not started the implementation of valued redundancy. Instead, we have written a simulation program that analyzes both the performance gains during the normal processing and the availability gains when nodes crash. The simulation model studies the interaction between local memory, remote memory, and the local disks in the next-generation architecture. With simple value management (such as read/write frequencies, creation and maintenance costs) we already observe significant performance gains over traditional buffer/cache mechanism and availability advantages over existing replication techniques.

For centralized systems, replication gains little because there are only a few modes of failure. It is only in a distributed system that we have enough hardware redundancy and independent failure modes to make data replication interesting. Therefore, valued redundancy is a technique mainly applicable to N-M mappings.

3.5 Old (but Important) Ideas

Having presented the new ideas in Synthesis, for completeness we should also include some ideas important to Synthesis that have been invented some time ago. In the first place, kernel code synthesis depends on two well-known software engineering principles. For successful code synthesis, we need a complete separation between the interface specification and the implementation. This separation depends in turn on the abstraction and encapsulation of the resources such as a file. In this aspect, the Synthesis kernel objects (threads, memory, and I/O) are specified as abstract data types (an object-oriented system in the modern jargon).

Another idea that influenced the design of the Synthesis is that OS kernels should be small. This idea has been defended by several previous small OS kernels such as Distributed V being developed at Stanford University and Amoeba at CWI and Free University of Amsterdam. We also believe that an OS kernel should remain small in order to achieve high performance.

We have also partitioned the Synthesis kernel into a small number of building blocks. This idea is shared by some other OS projects such as the *x*-Kernel being developed at the University of Arizona. The building block approach has been particularly successful in the Synthesis I/O support, introducing flexibility in the kernel code.

Finally, we are building emulators to run software written for other OS's. Several OS projects have announced their emulation of UNIX, for example, MACH at Carnegie-Mellon University and Sprite at the University of California at Berkeley. The main reason for their emulation is to import existing software, which is also one of our objectives. However, we also have two other goals with emulation.

First, although we have chosen the macro dataflow model of computation for Synthesis, we understand that future research on parallel and distributed processing may very well discover new and better models of computation. Since our choice of the macro dataflow model was made based on what we know now, we want to be able to include new and improved models when they are discovered. We will rely on emulation to maintain application program upward compatibility.

Second and more importantly, we use emulation as a scientific way to compare Synthesis performance with other OS's. Since we will run the same code for the same kernel interface on the same hardware, the comparison will be fair. The Synthesis emulation of SUNOS for performance comparison is described below. The main reason for the efficiency of Synthesis emulation is kernel code synthesis (in particular Collapsing Layers and Executable Data Structures).

4 The Implementation of Synthesis

Synthesis is an example of OS research "in practice". We have developed the ideas, applied them in an actual implementation, and refined them with the lessons learned from the implementation.

All the performance improvement techniques we used can be summarized in two software engineering principles. We call one of them the *principle of independence*, which separates the implementation from the specification. Among other examples, kernel code synthesis is made possible by a sufficiently abstract interface. We call the other the *principle of frugality*, which says that we should use the least powerful solution to a given problem. Optimistic synchronization is an example of frugality.

The principle of frugality leads us towards a different direction from the usual optimization. Normally people measure systems to find out what is being used the most, say procedure calls and IPC. Their optimization consists of making procedure calls and IPC fast. Instead of this line of work, the principle of frugality says that we should observe what is useful work in a system, e.g. calculations and algorithms, and what is overhead, e.g. procedure calls and IPC. Then we proceed to eliminate the overhead code, instead of trying to make it fast. The idea is, 1 millisecond is faster than 2 milliseconds, but you cannot beat 0 milliseconds.

4.1 Kernel Structure

The Synthesis kernel can be divided into a number of collections of procedures and data. We call these collections of procedures *quajects* that encapsulate hardware resources, like Hydra objects [11]. Important quajects in this paper are the I/O device servers and threads. I/O device servers are abstractions of the respective I/O devices and a thread is an abstraction of the CPU. Since these quajects consist only of procedures and data, they are passive. Events such as interrupts start the threads that animate the quajects and do work. The quajects do not support inheritance or any other language features.

Most quajects are implemented by combining a small number of building blocks. Some of the building blocks are well known, such as monitors, queues, and schedulers. The others are simple but somewhat unusual: switches, pumps and gauges. The unusual building blocks require some explanation. A switch is equivalent to the C switch statement. For example, switches direct interrupts to the appropriate service routines. A pump contains a thread that actively copies its input into its output. Pumps connect passive producers with passive consumers. A gauge counts events (e.g., procedure calls, data arrival, interrupts). Schedulers use gauges to collect data for scheduling decisions.

All of Synthesis I/O is implemented with these building blocks. Applying the principle of independence. each building block may have several implementations. Applying the principle of frugality, we use the most economical implementation depending on the usage. An example is the several kinds of queues in the Synthesis kernel. In general, code synthesis techniques are used to create the most efficient version of a quaject for each particular situation.

4.2 Threads

Synthesis threads are light-weight processes. Each Synthesis thread (called simply "thread" from now on) executes in a context defined by the computer architecture, including: the register save area, the vector table, the address map tables, and the context-switch-in and context-switch-out procedures synthesized by the kernel. Frequently occurring operations on a thread are context switching, blocking and unblocking. We now show how we speed up context switching.

Context switches are expensive in traditional systems like UNIX because they always do the work of a complete switch: save the registers in a system area, setup the C run-time stack. find the current proc-table and copy the registers into proc-table, start the next process, among other complications (summarized from SUNOS source code [2]). A Synthesis context-switch is shorter for two reasons. First, we use Executable Data Structures to minimize the critical path. Second, we switch only the part of the context being used, not all of it.

The key data structure in context switching is the ready queue, where we find the threads ready to run (waiting for CPU) chained in an executable circular queue. Instead of using data pointers that link the elements of the queue, we have a jump instruction at the end of each context-switch-out procedure of the preceding thread that points to the context-switch-in procedure of the following thread. When the context switch happens, we simply jump to the thread's contex-switch-out procedure, which consists of a few instructions to save the registers and ends up jumping into the next thread's context-switch-in procedure. This in turn restores the registers and starts the new thread.

Since the context switch code is custom created for each thread, we can further optimize it by moving data only when needed. Two optimization opportunities are in the handling of floating point registers and the MMU address space switch. Bypassing the floating point register saves about 50% of context switch time (table 2). Initially we assume no floating point. When the thread executes its first floating point instruction, the Synthesis kernel resynthesizes the context switch procedures to include the floating point co-processor. Similarly, we bypass MMU address space switch when the new thread shares address space with the old.

4.3 Scheduling

Currently, 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.

In our macro dataflow model of computation, a thread's "need to execute" is determined by the rate at which I/O data flows into and out of its quaspace. 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, our 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: we may raise a thread's CPU quantum, and we may reorder the ready queue when 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 way we minimize response time to events. 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 Synchronization

Since the threads do not execute in isolation, the kernel must synchronize their access to shared resources. We have three methods to reduce synchronization cost: Code Isolation, Procedure Chaining, and Optimistic Synchronization. Each method shortens the execution path in a somewhat different way. Informally speaking, Code Isolation and Procedure Chaining can be thought of as synchronization avoidance techniques. If absolutely unavoidable we use Optimistic Synchronization.

Using Optimistic Synchronization we have implemented an optimistic queue, which is an important example because most of the Synthesis kernel data structures are queues. Also, some of the control structures, such as chained interrupt and signal handlers, are implemented as queues of pointers to the routines. In other words, once we can synchronize queue operations without locking, most of the Synthesis kernel will run without locking.

The basic idea of an optimistic queue is to minimize synchronization overhead between the producer and the consumer; when the queue buffer is neither full nor empty, the consumer and the producer operate on different parts of the buffer. Therefore, synchronization is necessary only when the buffer becomes empty or full. The synchronization primitives are the usual primitives, say busy wait or blocking wait.

To avoid losing updates in the queue for a single producer and a single consumer, we use a variant of Code Isolation. The queue manipulation routines update two variables, the queue_head and the queue_tail. If only the producer writes on queue_head and only the consumer writes on queue_tail, they need not to synchronize if the queue_head and queue_tail are pointint to different parts of the queue. With some additional care the queue insert has a critical path of about a dozen MC68020 instructions protected only by a compare-and-swap instruction.

4.5 Input/Output

In Synthesis, I/O means more than device drivers. I/O includes all data flow among hardware devices and quaspaces. Data move along logical channels we call *streams*, which connect the source to the destination of data flow. Physical I/O devices are encapsulated in quajects called device servers. Typically, the device server interface supports the usual I/O operations such as `read` and `write`. In general, `write` denotes data flow in the same direction of control flow (from caller to callee), and `read` denotes data flow in the opposite direction of control flow (from callee back to caller).

High-level servers may be composed from more basic servers. At boot time, the kernel creates the servers for the raw physical devices. A simple example of composition is to pipeline the output of a raw server into a filter. Concretely, the Synthesis equivalent of UNIX cooked `tty` driver is a filter that processes the output from the raw `tty` server and interprets the erase and kill control characters. This filter reads characters from the raw keyboard server. To send characters to the screen, however, the filter writes to an optimistic queue, since output can come from both a user program or the echoing of input characters.

The default file system server is composed of several filter stages. Connected to the disk hardware we have a raw disk device server. The next stage in the pipeline is the disk scheduler, which contains the disk request queue, followed by the default file system cache manager, which contains the queue of data transfer buffers. Directly connected to the cache manager we have the synthesized code to read the currently open files. The other file systems that share the same physical disk unit would connect to the disk scheduler through a monitor and switch. The disk scheduler then will redirect the data flow to the appropriate stream.

Another implementation of queues, called the buffered queues, uses kernel code synthesis to generate several specialized queue insert operations (a couple of instructions); each moves a chunk of data into a different area of the same queue element. This way, the overhead of a queue insert is amortized by the blocking factor. For example, the A/D device server handles

44,100 (single word) interrupts per second by packing eight 32-bit words per queue element. This is orders of magnitude faster than current general-purpose OS's.

4.6 Performance Figures

The current implementation of Synthesis runs on an experimental machine (called the Quamachine), which is similar to a SUN-3: a Motorola 68020 CPU, 2.5 MB no-wait state main memory, 390 MB hard disk, $3\frac{1}{2}$ inch floppy drive. In addition, it has some unusual I/O devices: two-channel 16-bit analog output, two-channel 16-bit analog input, a compact disc player interface, and a 2Kx2Kx8-bit framebuffer with graphics co-processor.

The Quamachine is designed and instrumented to aid systems research. Measurement facilities include an instruction counter, a memory reference counter, hardware program tracing, and a microsecond-resolution interval timer. The CPU can operate at any clock speed from 1 MHz up to 50 MHz. Normally we run the Quamachine at 50 MHz. By setting the CPU speed to 16 MHz and introducing 1 wait-state into the memory access, the Quamachine can closely emulate the performance of a SUN-3/160.

The current implementation of Synthesis kernel is fully operational, supporting the threads, memory, and I/O devices. A number of application and demonstration programs use the Synthesis kernel. The Synthesis kernel is written in 680x0 assembly language. We believe that the Synthesis kernel is quite portable since it contains only about 3000 lines of assembly code. This is less than many C compiler run-time library written in assembly.

For detailed measurement numbers we refer the reader to our SOSP paper [6]. Here, we only highlight the performance of Synthesis kernel. Measurements are made on the UNIX emulator running on top of the Synthesis kernel, which is capable of servicing SUNOS kernel calls. In the simplest case, the emulator translates the UNIX kernel call into an equivalent Synthesis kernel call. Otherwise, multiple Synthesis primitives are combined to emulate a UNIX call.

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 under the UNIX emulator. To validate our emulation, the first benchmark program is a compute-bound test of similarity between the two machines. This test program implements a function producing a chaotic sequence. It touches a large array at non-contiguous points, which ensures that we are not just measuring the "in-the-cache" performance. With both hardware and software emulation, we run the same object code on equivalent hardware to achieve a fair comparison between Synthesis and SUNOS.

In table 1 we summarize and compare the results of the measurements. The columns under "Raw SUN data" were obtained with the `time` command and also with a stopwatch. The SUN was unloaded during these measurements, as `time` reported more than 99% CPU available for them. The Synthesis emulator data were obtained by using the microsecond-resolution real-time clock on the Quamachine, rounded to hundredths of a second. These

		====Raw Sun Data===				Sun	Synthesis		Synthesis
No	Descr.	user	sys	tot	watch	U+S	Emulator	Ratio	thruput
1	Validation	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.	100KB/s
3	R/W pipe 1024	0.5	14.6	15	15.3	15.1	2.42	6.2	8MB/sec
4	R/W pipe 4096	0.7	37.2	38	38.2	37.9	9.64	3.9	8MB/sec
5	R/W file	0.5	20.1	21	23.4	20.6	2.91	7.1	6MB/sec
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 1: Measured UNIX System Calls (in seconds)

operation	time (usec)
Full context switch	11 (*)
Full context switch	21 (with FP registers)
Thread create	142
Thread destroy	11

(*) If the thread does not use the Floating Point co-processor.

Table 2: Synthesis Kernel Overhead (in microseconds)

times were also verified with stopwatch (sometimes running each test 10 times to obtain a more easily measured time interval).

Program 1 is the computation-intensive calibration function to validate the hardware emulation. The 5% difference between SUN and Synthesis is due to the SUN clock running at 16.7 MHz, compared to the 16 MHz of the Quamachine. Programs 2, 3, and 4 write and then read back data from a pipe in chunks of 1, 1K and 4K bytes. They show a remarkable speed advantage (56 times) for single-byte read/write operations and significant difference (4 to 6 times) even at page size. Program 5 reads and writes a file (cached in main memory) in chunks of 1K bytes. Programs 6 and 7 open and close `/dev/null` and `/dev/tty`, showing that Synthesis kernel code generation is very efficient.

Besides I/O operations, the overhead of other Synthesis kernel services is also small. For example, a full context switch costs 11 microseconds if the thread does not use the floating point co-processor and 21 microseconds if it does. Since UNIX does not have light-weight processes, this is not directly comparable. To give an idea of the state-of-art on this kernel service, Mach thread context switch on an RT/APC (roughly comparable to SUN-3/160) takes about 130 microseconds. In contrast, in table 2 we see that Synthesis threads are light-weight - less than 150 microsecond creation time.

4.7 Applications and Demos

We have written a few demo programs to exercise the high performance capabilities of Synthesis. Most demos do real time signal processing and graphics to show the performance and predictability in Synthesis. In particular, they demonstrate the high I/O rate in Synthesis due to code synthesis and the real-time response due to fine-grain scheduling.

The first demo is a music synthesizer that reads a music score for four voices. Each voice is implemented as a separate thread that creates its waveforms, which are then pipelined to a pair of threads that simulate a reverberation chamber and produce the stereo effect. The waveforms are piped through the raw digital-to-analog (D/A) driver to the speakers. The music synthesizer illustrates the generality of the Synthesis model of computation, since it uses the OS primitives for all of its I/O as well as memory management and threads. It also demonstrates the efficiency of Synthesis implementation and the effectiveness of fine-grain scheduling: the entire pipeline (consisting of the four voice threads, the two reverberation threads and the D/A driver) produces CD-quality sound (44,100 stereo sample points per second) in real-time on a 40MHz MC68020. When running the music synthesizer, the kernel provides an I/O bandwidth of 1.4 MBytes/second through the pipes at the same time the threads make 33,000 context switches per second.

The second example is the digital interpolator filter for the CD player. A digital interpolator takes as input a stream of sampled data (from the raw CD player device driver) and creates additional samples in-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 generates 44,100 data samples per second (one every 22.68 microseconds). The interpolated data output is four times this rate, or one every 5.67 microseconds. Again, fine-grain scheduling ensures that the interpolator thread runs often enough to sustain this data rate, in real-time.

A third example is the disk driver, which speeds up disk accesses by using a software locked loop (SLL) to calculate which sector is under the disk head and re-order disk requests for minimum rotational delay. The SLL measures intervals between successive disk index interrupts (once per rotation), synchronizes to this interrupt. and creates a new source of interrupts corresponding to each sector (about once every 500 microseconds). Thus the disk driver knows which sectors are closest to the disk heads and can perform rotational optimization in addition to normal seek optimization.

The fourth demo program uses an SLL to find the beat interval of a piece of music and create as output a synthesized rhythm track which is then re-mixed with the original. This demo is implemented as four cooperating routines, each in its own thread: the correlator, the interval divider. the drum synthesizer, and the mixer. The correlator reads the music input (a string of 32-bit samples), detects beat intervals, and interrupts the interval divider at the start of each beat. The interval divider uses an SLL to subdivide the beat interval into 16 equal parts, in the same way that the disk driver subdivides a disk rotation into sectors. This information is sent to the drum synthesizer, which creates a drum pattern to fit the beat interval. Finally, the mixer reads the the original music and the drum output, mixes them using a simulated reverb chamber, and passes the result to the A/D driver.

4.8 Fitting Together

We have designed and implemented Synthesis as the basis for the development of the next-generation system. In the first place, all the important ideas in Synthesis (kernel code synthesis, reduced synchronization, fine-grain scheduling, and valued redundancy) contribute to high performance. We expect the system programmers trying to optimize application programs and compiler writers trying to optimize code generation to find Synthesis performance desirable.

The second point is availability. Valued redundancy maximizes data availability at low cost (in space). After a node crash, valued redundancy would have kept the most valuable data items on the network and available. Up to now, we have not addressed the issue of recovering aborted programs (a la Tandem NonStop).

The Synthesis model of computation (composed of threads, memory, and I/O) is simple and intuitive. Its parallel processing interface is at a level of abstraction comparable to UNIX. On top of the Synthesis native interface we have the guest OS emulators, for example, the UNIX emulator. These emulators support the existing software and interfaces familiar to the users.

Synthesis is also intended for real-time applications. We are developing the programming methodology and tools that use fine-grain scheduling to make the system performance adaptive and predictable. The combination of high performance with adaptiveness in the Synthesis kernel is most useful to real-time application programmers.

5 Conclusion

We define an OS as a mapping from a model of computation defined by its kernel interface onto the hardware. We classify the OS's into three kinds: 1-1, N-1, and N-M. The 1-1 OS maps a uniprocessor model into a uniprocessor machine. The N-1 OS supports multiprogramming, which maps several uniprocessor models into a uniprocessor machine. The N-M OS maps a multiprocessor model into a multiprocessor and distributed system.

We believe that the next-generation computer systems will consist of very fast networks connecting many fast machines with plenty of memory and disk space. The OS's for such an environment (N-M mappings) must present an intuitive and simple model of computation that supports parallel and distributed processing as its native mode. Furthermore, their implementation must be efficient for usability and simple for portability.

We describe the Synthesis OS as an example of N-M mappings designed and implemented for the multiprocessor and distributed systems. Synthesis incorporates several innovative ideas in its implementation. Traditional OS's always interpret their system calls: in contrast, Synthesis relies on Kernel Code Synthesis to "compile" frequently used kernel calls. Normal concurrent programs use synchronization methods such as semaphores; the Synthesis kernel adopts Reduced Synchronization to reduce interlocking overhead. Usual scheduling techniques consume sizable amount of CPU; the Fine-Grain Scheduling in Synthesis uses very fast context switches to provide adaptive and predictable response time. Conventional replication methods are expensive; Synthesis valued redundancy provides both availability and performance at low cost.

Synthesis is a new way of looking at operating systems. We combine new implementation techniques such as kernel code synthesis with a simple and intuitive model of computation (thread, memory, and I/O). The result is a distributed and parallel OS (N-M mapping), which is faster and easier to use than an N-1 OS even for a uniprocessor. The current version of Synthesis kernel is several times to several dozen times faster than SUNOS running the same executable code on equivalent hardware. We are porting Synthesis to the NeXT machine to enlarge our hardware base to explore parallel and distributed computing.

References

[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] Anonymous et al.
SUNOS release 3.5 source code.
SUN Microsystems Source License, 1988.

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

[4] B. Lampson.
Hints for computer system design.
IEEE Software, 1(1):11–28, January 1984.

[5] H. Massalin and C. Pu.
Fine-grain scheduling.
In *Proceedings of the Workshop on Experience in Building Distributed Systems*, Fort Lauderdale, Florida, October 1989.

[6] H. Massalin and C. Pu.
Threads and input/output in the Synthesis kernel.

In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Arizona, December 1989.

[7] C. Pu, A. Leff, S.W. Chen, F. Korz, and J. Wha.
Valued redundancy.
Technical Report CUCS-453-89, Department of Computer Science, Columbia University, August 1989.

[8] C. Pu, H. Massalin, and J. Ioannidis.
The Synthesis kernel.
Computing Systems, 1(1):11–32, Winter 1988.

[9] D.M. Ritchie and K. Thompson.
The Unix time-sharing system.
Communications of ACM, 7(7):365–375, July 1974.

[10] M. Stonebraker.
Operating system support for database management.
Communications of ACM, 24(7):412–418, July 1981.

[11] 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.