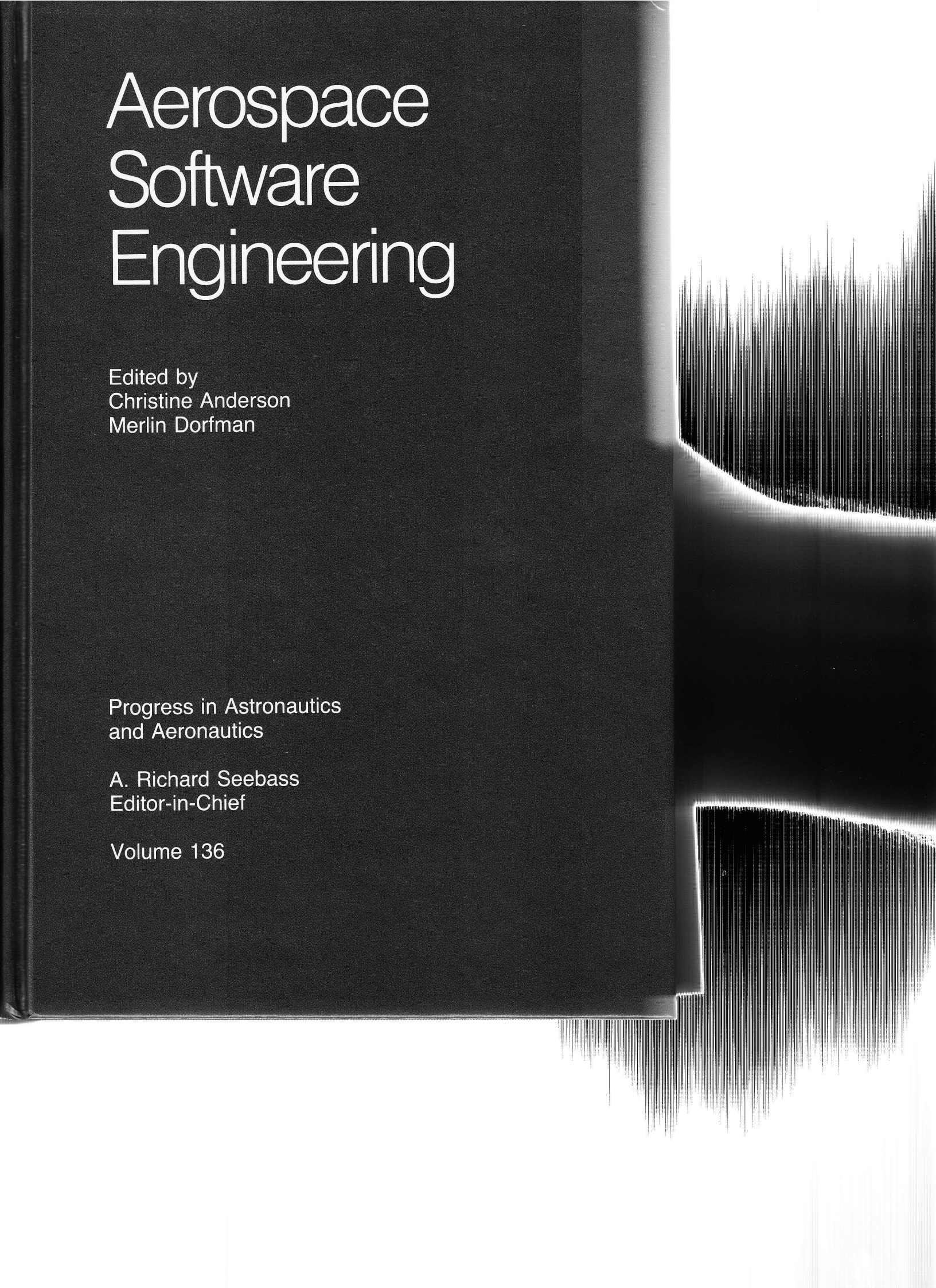
Aerospace Software Engineering

Edited by
Christine Anderson
Merlin Dorfman

Progress in Astronautics
and Aeronautics

A. Richard Seebass
Editor-in-Chief

Volume 136



Aerospace Software Engineering
A Collection of Concepts

Edited by
Christine Anderson
U.S. Air Force Wright Laboratory
Armament Directorate
Eglin Air Force Base, Florida

Merlin Dorfman
Lockheed Missiles & Space Company, Inc.
Sunnyvale, California

Volume 136
PROGRESS IN
ASTRONAUTICS AND AERONAUTICS

A. Richard Seebass, Editor-in-Chief
University of Colorado at Boulder
Boulder, Colorado

Published by the American Institute of Aeronautics and
Astronautics, Inc., 370 L'Enfant Promenade, SW,
Washington, DC 20024-2518

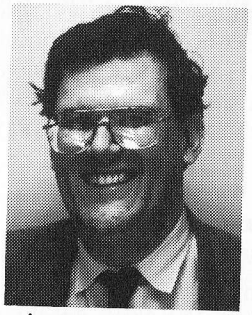
American Institute of Aeronautics and Astronautics, Inc. Washington, DC

Library of Congress Cataloging-in-Publication Data

Aerospace software engineering / edited by Christine Anderson, Merlin Dorfman.
p. cm.—(Progress in astronautics and aeronautics; v. 136)
Includes bibliographical references.
1. Aeronautics—Computer Programs. 2. Astronautics—Computer programs. 3. Software engineering. I. Anderson, Christine, 1947–
—II. Dorfman, M. (Merlin) III. Series.
TL507.P75 vol. 136 (TL563) 629.1 s—dc20 [629.13'00285'53] 91-27124
ISBN 1-56347-005-5

Progress in Astronautics and Aeronautics

[6]Dewar, R. B. K., and Smosna, M., *Microprocessors, a Programmer's View*, McGraw-Hill, 1990.
[7]Wichman, B. A., "Low-Ada: An Ada Validation Tool," National Physical Laboratory, NPL Rept. DITC 144/89, Aug. 1989.
[8]"IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754–1985 ed., 1985.
[9]Wichman, B. A., "A Note on the Use of Floating Point in Critical Systems," National Physical Lab. (to be published).
[10]Payne, M., Schaffert, C., and Wichman, B. A., "The Language Compatible Arithmetic Standard," *ACM SIGPLAN Notices*, Vol. 25, pp. 59–86; also, *ACM SIGNUM Newsletter*, Vol. 25, No. 1, pp. 2–43, Jan. 1990.
[11]"Interim Defence Standard 00–55, "Requirements for the Procurement of Safety Critical Software in Defence Equipment," UK Ministry of Defence, undated.
[12]Barrett, G., "Formal Methods Applied to a Floating-Point Number System," Oxford University Programming Research Group, TM-PRG-58, 1987.



Robert B. K. Dewar is a Professor and past Chairman of the Computer Science Department at New York University. His fields of interest include real-time software and operating systems, programming languages and compilers, and microprocessor architectures. He has extensive experience in the design and implementation of software systems, including a series of real-time operating systems for Honeywell. He wrote the backend of the Realia COBOL compiler, and has consulted for Alsys Inc. on the design and development of their compilers, particularly for the Intel line. He has been involved in the Ada language design effort almost from its inception, and most recently has served as one of the authors of the Ada 9X Requirements Document. He has also consulted in the area of embedded real-time software development, and has recently published a book with Matthew Smosna, *Microprocessors, a Programmers View*, which discusses the CISC vs RISC debate from a programmer's point of view.

Chapter 10.3

Symmetric Parallel Processing

Ilya Gertner and Ike Nassi
Encore Computer Corporation, Marlborough, Massachusetts

Symmetric parallel processing is one of the major techniques in achieving efficient utilization of multiprocessors. Those techniques have been realized in hardware for quite some time. In software, on the other hand, this symmetric design has been frequently overlooked in favor of simpler implementations. This has resulted in performance penalties that in some cases undercut even the reason for moving software to multiprocessor systems. In order to support symmetric multiprocessing, software must be symmetric at all levels, *including the system kernel, run time language support libraries, and user applications. This paper describes our experience in implementing and using the symmetric parallel software at several levels, including a UNIX kernel, Ada run time, high-level debuggers, and performance monitors. Preliminary experience indicates that many programming tools and applications can be built with the symmetric model.*

Introduction

SYMMETRIC parallel processing is a technique for implementing parallel systems where none of the distinguishable components have an identifiable serial component. All of the components run in parallel; occasionally some of the components synchronize their computations with each other when accessing shared resources. In contrast, nonsymmetric processing such as master-slave is a technique where all computing components (i.e., slaves) always ask for a permission from the master to perform computations. The master becomes the chief arbitrator, and a potential sequential bottleneck, no matter how parallel the rest of the computation.

In reality the notion of the master comes up in many environments where resources are shared for economic reasons. We find examples of the masters at different levels ranging from the multiprocessor hardware level where a single bus (i.e., the master arbitrator) is shared by multiple CPUs to a software level where users may share a high-quality laser printer (another master) in a time-sharing environment. In all of the aforementioned cases the final decision for accessing the shared resource is made by the master process. There is no way to avoid using the master if we want to share costs.

Although there is no way to completely eliminate the master server, it is extremely important to minimize the processing time of the master. In software this is usually accomplished by postponing the request to the master until the last possible moment. For example, one needs to reserve the printer only when it is actually printing, and not when the printing software is copying a file. By requesting the master too soon, one frequently wastes resources and introduces sequential bottlenecks in the system that could have been avoided through a symmetric design. This paper focuses on this "postponing" of requests to the master.

Design of a symmetric shared-memory multiprocessor begins with the design of the access to memory from all processors. There are no special processors: All processors can access memory equally well (from the points of view of both the user interface and response time). In addition, all processors are equal in handling devices: Interrupts are handled according to priority level by the first available processor; input/output (I/O) operations can be initiated by any processor. This provides the basis for a completely symmetric environment where all user requests are balanced against the system hardware resources. This is the main objective of the operating system kernel and run time libraries: to provide a balance between the user workload and available system resources. No system-level software should introduce a component that cannot satisfy those requirements.

Symmetric properties are often inherent in many applications. For example, time sharing is inherently a symmetric parallel application. Multiple users access computing resources at the same time. It can be naturally supported on a multiprocessor. Sometimes even a single user can naturally benefit from a multiprocessor. Consider a software development cycle: A user iterates through the edit, compile, link, and debug steps. The edit and compile steps typically involve multiple files. The compilation of multiple files is a naturally parallel application. In fact, today the UNIX "make" program is the most common parallel application supported on many multiprocessors.[1,2]

Inherent parallelism in an application and underlying symmetric multiprocessor hardware does not automatically ensure the best possible performance. For example, bottlenecks in the file system may prevent the previously described parallel make to scale linearly over a number of processors that suggest the focus of this paper. How can the system software be designed so that the degree of parallelism is limited only by the hardware and by the nature of applications?

Related Work

Although the topic of operating system support for multiprocessors has received considerable attention in the literature, the issues of symmetry have not been as widely reported. Many papers are concerned with the hardware issues such as the way in which each processor accesses memory or about low-level synchronization primitives.[3]

All of the preceding issues are unimportant or trivial on shared-memory multiprocessors. For example, in developing the Multimax (a shared-memory multiprocessor), hardware was designed to support uniform, efficient accesses to memory.[4] The same approach has also been extended to a cluster of multiprocessors.[5] This eliminated any need for the software to attempt to optimize local memory references. In addition, access to shared memory became the main synchronization mechanism. It was trivial to implement a variety of synchronization mechanisms, such as the mutual exclusion locks, read-write locks, and messages.

Although shared-memory multiprocessors implemented with the bus and caches may fail for some applications (in fact, given a particular cache design, one can write a trivial program that breaks that cache), in practice, we found most of the applications to have a 90–95% cache hit ratio. In other cases we found techniques for changing applications to run efficiently in this paradigm.[6] In our experiences those drawbacks are small compared to the advantages of having a simple, efficient mechanism for synchronization and device handling.

Other papers on multiprocessors deal with the issue of the multithreading systems kernel, but leave out the issues of multithreading of run time libraries and managing resources.[7,8] As a result, some of the layers are left nonsymmetric, contributing to the sequential bottleneck of the total system. It is the position of this paper that the most difficult part, and the largest performance gain, is to make all system levels multithreaded and fully symmetric; otherwise, we cannot claim to have a general-purpose parallel processing system.

Multithreading vs Master-Slave

Given parallel applications (including naturally parallel time-sharing applications), the next step is to get the systems software to function properly in the multiprocessor environment. One needs to protect access to all shared data. There are basically two techniques for achieving this: multithreading and master-slave.

The master-slave approach is the easiest way to get programs to run on a multiprocessor. It is typically applied to systems software that has been implemented on a sequential machine. On a multiprocessor the systems software can be protected from corruption (by multiple users running in parallel and using the same software) by creating an additional layer, the master of the system services, which controls access to that systems software. The simplest control discipline is that of one master, which allows one slave at a time to access the protected software.

Multithreading the systems software, albeit more difficult, is a more efficient method of implementation. To multithread means to ensure that

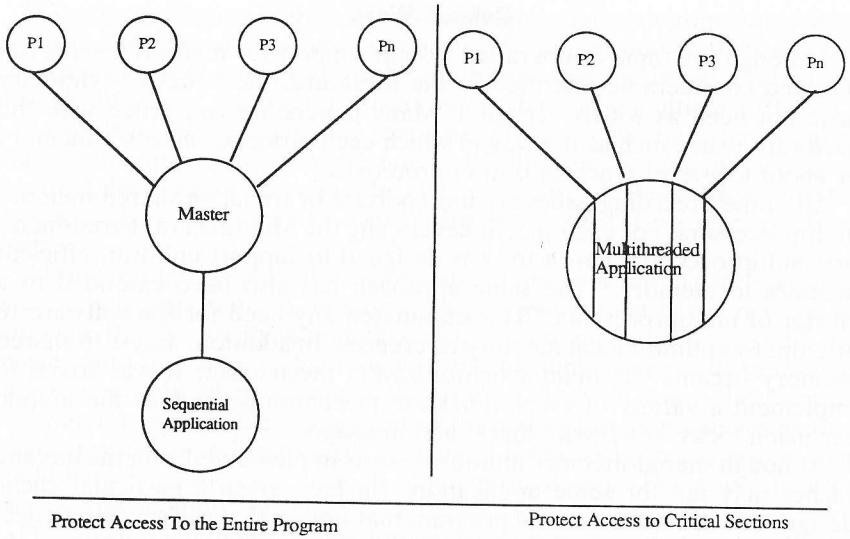


Fig. 10.3.1 Design trade-offs; master-slave vs multithreading.

multiple threads of control at the same time can execute the same program safely. It essentially requires rewriting the systems software to protect accesses to all shared data (Fig. 10.3.1).

The performance measurements described later indicate the importance of multithreading in the systems kernel. The class of user programs we are considering are those that half of the time run at the user level and half of the time at the system level. A typical example of such a program is the C compiler, which is the most commonly used program in a time-sharing program development environment.[1] To simulate a multiuser load, multiple operations are then done in the background using a UNIX shell script. The results are given in terms of the real time necessary to complete all compilations (Fig. 10.3.2).

The results in Fig. 10.3.2 clearly point out the problem: The master-slave kernel fails to scale beyond three processors, whereas the multithreaded kernel continues to scale for up to 12 processors. The anticipation of these results provided the main motivation for multithreading the UNIX kernel.

Multithreading a UNIX Kernel

The standard (sequential) UNIX code assumes that the kernel is never preempted except for processing of interrupts. Hence, kernel data structures do not need to be protected unless referenced by an interrupt routine, and if so, the data can be protected by locking out interrupts. This is normally done by raising the processor priority level high enough to prevent the type of interrupt from occurring. This protection is insufficient in multiprocessors where an arbitrary fragment of kernel code can be executed concurrently on several processors. Simultaneous updates to the same data structure can result in inconsistencies.

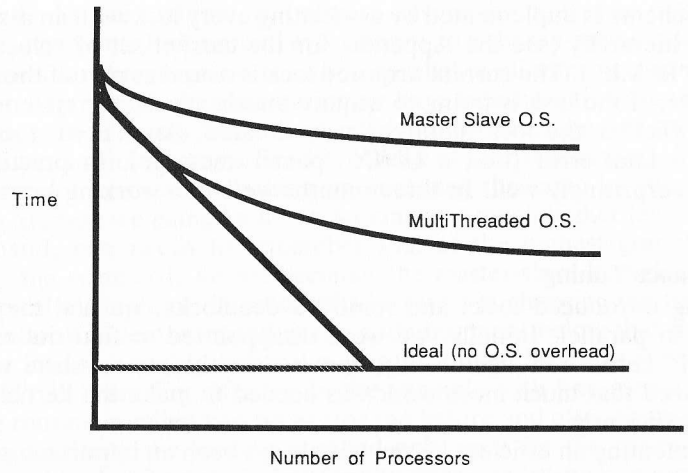


Fig. 10.3.2 Master-slave vs multithreading performance.

Ensuring consistency in a multithreading kernel is very simple: One has to identify all data structure references that have the potential of being cross referenced from different processors. Then one is going to introduce synchronizers (e.g., mutual exclusion locks, called "mutex") around all accesses to shared data. Having accomplished this, we have the first version of the multithreaded kernel. The hard problems begin later when one has to tune the performance of the kernel and debug it to be deadlock-free.

Avoiding Deadlocks

Deadlock avoidance in a kernel is a classical problem that has been described in many textbooks on operating systems. In a multithreaded kernel (implemented with mutex locks as described earlier), the deadlock occurs if a task T1 accesses data structure A using a mutex lock A followed by access to the data structure using lock B. At the same time task T2 attempts to access the data structure B followed by access to the structure A. This is a classical case of a deadlock where some tasks are waiting for each other's resources (resource in this case is a data structure), but since all are waiting, none is able to release a resource.

There are basically three approaches to avoiding such deadlocks: 1) Change the kernel code and data structures so that at no time can a task ask for more than one consecutive mutual lock. 2) Write a high-level synchronization function that ensures deadlock-free acquisition of two resources. 3) Introduce an ordering on the set of resources (and their associated mutex locks).

Our choice has been to proceed with the third choice. We felt that this would minimize the amount of changes in the system kernel.

The ordering of locks corresponds to a directed acyclic graph. This implies that resources can be accessed only in a certain order. For example, one could define a kernel resource so that access to the resource A is followed by access to the resource B. If a task accesses resource B and then attempts to access the resource A, the kernel reports an error.

This scheme is implemented by associating every lock with an associated level of hierarchy (see the Appendix for the current set of values in the header "lock.h"). The current acquired lock is stored as part of the current task state. If the task is trying to acquire another lock, the system kernel checks whether the lock hierarchy was violated. And, if so, the kernel reports a fatal error (i.e., a UNIX "panic" message). In practice, this worked surprisingly well. In three months we had a working kernel.

Performance Tuning

Having introduced locks and removed deadlocks, we had the kernel running in parallel. Initially, we were disappointed to find out that the "parallel" kernel ran significantly slower than the master-slave version. We realized that much more work was needed to make the kernel run in parallel efficiently.

Implementing an efficient kernel has always been an important issue for any architecture. Still, on sequential architectures it was considered a luxury, an afterthought, if time permitted to do some additional work. On a multiprocessor we realized that efficient kernel is as important as correct kernel. Otherwise, why move to a multiprocessor? Performance tuning was clearly the next task.

One of the major decisions for every mutex is to choose between a spinlock (busy wait) and counting semaphore (Dijkstra's scheduling primitive). How does one choose between the two seemingly conflicting primitives? The spinlock() seems to waste processor resources, whereas the semaphore "gracefully" puts a task to sleep. The performance characteristics of the spinlock() and semaphore are also contradictory. Spinlock requires a minimum of four instructions. In the worst case a blocking semaphore routine call requires about a thousand instructions to put a task A to sleep and schedule another task B that eventually wakes up task A (directly or indirectly), which again must be scheduled and returned from the call to the semaphore routine. Clearly, if the average time for resolving the conflict is on the order of milliseconds, then the advice is to use semaphores. If the time is on the order of tens of microseconds, then the desired approach is to use a spinlock. It is very difficult at the design time to decide on the optimal set of primitives.

We began with a straightforward approach: If the code looks simple (e.g., fits in a single page), then use busy-wait; otherwise, use semaphores. For quite some time we have been considering the idea of a combined primitive: Spin for a while, then block. Following some experimentation we have rejected this alternative. It turns out that OS operations are clearly divided into two classes: data manipulation and device accesses. Device access times are on the order of tens of milliseconds, whereas data manipulations are on the order of tens or hundreds of microseconds. Given a process scheduling time on the order of a few milliseconds, the choice between primitives became clear: All data manipulation should wait with a busy lock; all device accesses should suspend.

Another design decision is the granularity of the mutex lock. What data structure (or the code that manipulates that data structure) needs to be

protected? At one extreme, in order to ensure the maximum parallelism possible, one wants lock around every access to a shared-memory location. This allows for the programs to run in parallel at all unprotected memory accesses. After some experimentation we have measured that this approach would introduce a very large fixed overhead for entering and exiting the locks. Even a spinlock has a fixed overhead of four instructions. Since the probability of processors clashing on accessing the same location is very low, on average we gained a lot by locking at a much higher level. On the other hand, one needs to remember that at the highest granularity of locking, the symmetric kernel becomes the master-slave kernel!

A typical example of the trade-offs involved in choosing the right level of granularity is demonstrated in the code fragment below that inserts an element into the doubly linked list [e.g., a linked list of process control blocks (PCBs); see Fig. 10.3.3]. In the example with fine granularity the locking routine is called two times (for the before and after element); with coarse granularity the locking is called only once for the entire list. Although the probability of clashes for the once-per-list lock increases, the fixed overhead for calling lock and unlock decreases. Only by measuring a running system with real workloads were we able to make those trade-offs.

In some cases the simple locking primitive is too restrictive. In particular, the mutual exclusion primitive described earlier is a very strict primitive: Only one processor at a time can be in the protected code section. This is clearly too prohibitive for some applications. For example, in the file directory system we want a read-write lock that allows multiple readers to proceed concurrently. Many processes may need to read the same directory

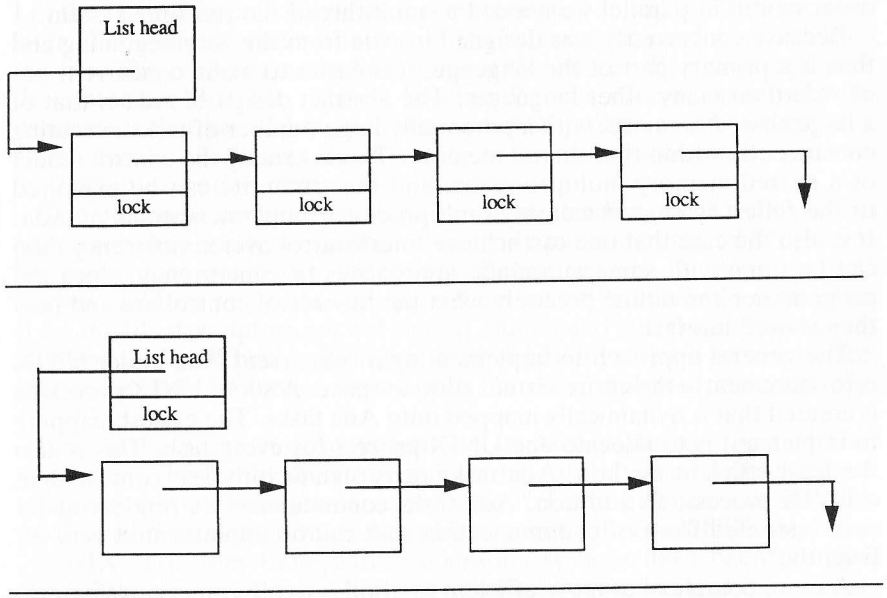


Fig. 10.3.3 Granularity of locks; inserting elements in a list. Locking elements vs locking lists.

control structure. On the other hand, the directories rarely change compared to other operations. Read-write locks were designed for this purpose. This again contributed to significant improved performance scalability of the file system.

Finally, some algorithms turned out to be very awkward to parallelize. In such situations one lock was followed by another lock, and another lock, and the system would panic (i.e., deadlock). In such cases we would say that the algorithm is not parallelizable, and we would search for higher-level locks and rewrite the algorithm completely to make it easier for it to run in parallel.

Ada Run Time

Multithreading Ada Run Time

In general, concurrent programming languages have run time systems that have characteristics that are similar to those of operating systems. They schedule threads of control, have synchronization needs, and provide access to shared resources.

A typical example is Ada, a language that uses tasking to support concurrency. Since Ada has been designed with portability in mind, a compiler for Ada comes with a run time library that supports tasking. (Both for portability and historical reasons Ada was introduced on sequential processors, with concurrency implemented at user level.)

An Ada run time system has properties similar to the UNIX kernel described earlier. It generally has been written with the single processor in mind where only that processor is executing the run time system. In order to run in parallel we needed to multithread the run time system.

Because concurrency was designed in Ada from the very beginning and thus is a primary part of the language, it is easier to make concurrent use of Ada than many other languages. The abstract design of Ada is that of a large shared memory, with a potentially large number of tasks executing concurrently within that shared memory. This is exactly the control model of a shared-memory multiprocessor, and this similarity can be exploited to the fullest in shared-memory multiprocessor implementations of Ada. It is also the case that one can achieve finer control over concurrency than can be done with some automatic approaches to concurrency, since the programmer can define precisely what the threads of control are and how they should interact.

The general approach to implementing a concurrent Ada under UNIX is to share nearly the entire virtual address space. A set of UNIX processes is created that is dynamically mapped onto Ada tasks. The easiest mapping to implement is to allocate one UNIX process for every task. This is also the least efficient method. Ada tasks carry significantly less context than a UNIX process. In addition, Ada tasks communicate via rendezvous, a very restricted form of communication that can be implemented very efficiently.[9]

A more complex but more efficient method is to allocate a certain number of UNIX processes that map to a much larger number of tasks. The processes are allocated for the lifetime of the program. The number of processes is set by the run time to express the amount of parallelism that the user wishes to exploit (i.e., it is not bound into the code of the program).

The user can thus vary the number of processes used by the program between runs and conduct experiments to find the optimal partitioning for various programs and input.

When the Ada run time system tries to run a task, it first sees if a process is available to run it. If so, the process assumes the identity of the task and executes it. If no process is available (and there is no lower priority task that must be preempted), the task is queued on a ready queue for execution by the next process that becomes available. In this way Ada programs with a potentially enormous set of tasks can be written and executed, and as processing power increases over the next few years, these programs will automatically work on the newer, massively powerful machines without major modification (Fig. 10.3.4).

Input/Output Processing

A difference between the Ada run time system and a UNIX multiprocessor kernel involves the difference between UNIX processes and Ada tasks. In general, concurrency of UNIX processes is hierarchical. The kernel controls the sequencing of processes, and they rarely interfere with each other, except through few, well-defined points, such as communications over pipes.

The marriage of Ada and UNIX is not without its problems, however. For example, the Ada tasking implementation described earlier involved a set of Ada tasks that ran on top of a set of UNIX processes. Consider a set of tasks T1 and T2 running on top of UNIX processes P1 and P2. Let the task T1 open a file while it is rescheduled on top of P1. Some time later, T1, while scheduled on top of P2, attempts to read a record from the file. The read operation fails because the process P2 has no open file descriptor for that file. This is because P1 and P2 can share only file descriptors that have been open during UNIX process creation time [i.e., UNIX fork()]. Following the creation, both processes continue on an independent execution path.

The preceding problem points to problems in the whole UNIX process paradigm when applied to multiprocessing. In particular, the UNIX process hierarchy does not match the model of Ada tasking. A parent process spawns children processes and gives them a COPY of all file descriptors (i.e., the file descriptors are not shared, but copied into the child process's address space). When a child gets control and allocates a new resource, that resource is no longer shared with the other children processes (Fig. 10.3.5).

This is the problem of implementing an Ada tasking on top of UNIX processes. This problem is not unique to Ada tasking. Any user-level scheduler suffers from the same problem.[10] The problem is not with this particular implementation of Ada threads, but with the UNIX paradigm.

UNIX and other time-sharing operating systems have been designed around the notion of protection and privacy of processes that by necessity run on a time-shared hardware. The concern with privacy and protection is clearly inappropriate for certain applications on a shared-memory mul-

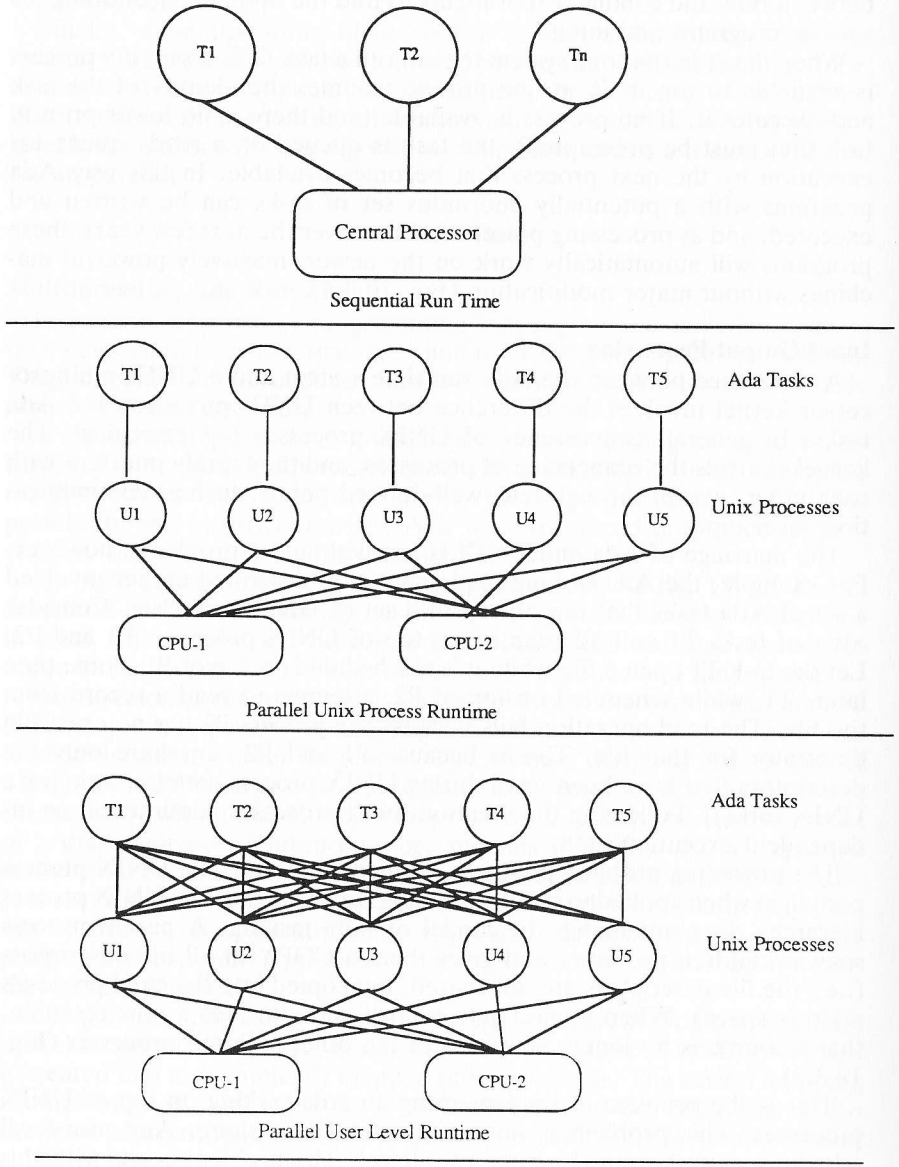


Fig. 10.3.4 Ada run time design alternatives under UNIX.

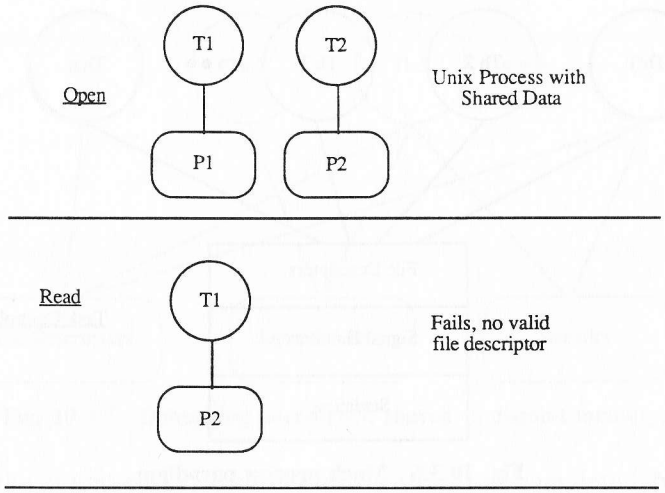


Fig. 10.3.5 Ada run time I/O problem.

tiprocessor, where often it is precisely the sharing of data that is of main interest. A parallel program that consists of a set of tightly communicating processes (or threads) is routinely allocated to a single user.

Clearly, there is a need for a new process paradigm. The first direction we have been experimenting with is the MACH threads model.[11]

MACH introduces a new structure called a task control block (TCB), which directly addresses the machine model of the kernel. The intent is to allow for the existence of "lightweight" threads of control that are, in effect, little more than virtual CPUs. These "threads" are the control units scheduled by the kernel and are grouped into "tasks," each with its own TCB and each describing the remaining elements of the machine (memory, I/O, etc.) for the entire set of its satellite threads.

In MACH the previously described I/O problem does not exist because all threads point to the same set of file descriptors. Put another way, all Ada tasks can potentially share a set of files (Fig. 10.3.6).

MACH makes the decision of sharing all resources; UNIX makes the decision of sharing in a very special hierarchical way at the process creation time. This seems to be an unnecessary limitation. Those limitations become specially visible when implementing a parallel debugger. To meet those requirements, we have been moving toward a new operating system paradigm.

Debugging Requirements

Consider the problem of an Ada debugger. The debugger wants to share memory but does not want to share exception handlers. For example, a floating-point exception in the application may cause the program to fail (if the user did not provide any exception handling code), while the debugger may only want to report the thread in which the exception occurred. Some file descriptors are shared by both the debugger and application;

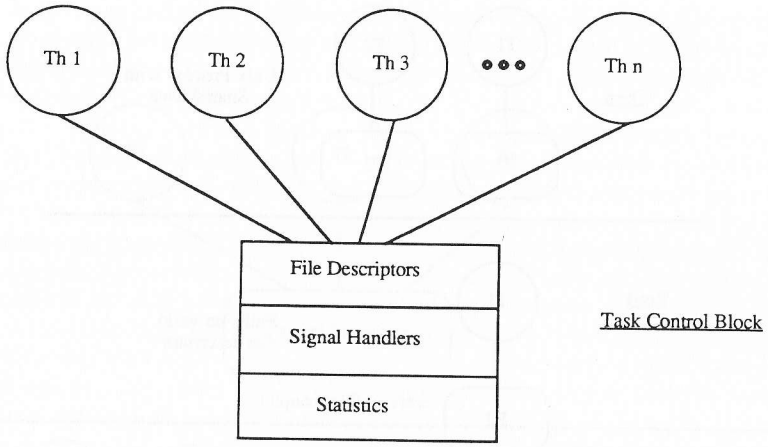


Fig. 10.3.6 Mach process paradigm.

other descriptors are private to each. In summary, we need a more flexible sharing of resources than that provided by both MACH or UNIX.

Today existing debuggers are built using the UNIX process model.[12] There the debugger is allocated a new UNIX process that uses the special system calls to manipulate the state of the application[13] (Fig. 10.3.7).

Although such debuggers adequately support sequential programs, in our experience they are very limited in a parallel environment. In fact, for quite some time we have been working on a parallel debugger that relies on the efficient sharing of resources.[14] Our experience comes from developing Parasight, a parallel programming environment that was the major force in defining the requirements for resource management for parallel programs.

To effectively support Parasight, we needed a new UNIX environment that allows one to flexibly allocate and manage resources. Some of the resources are shared between the debugger and an application; others remain private to each.

To support those requirements we have developed a new operating systems paradigm called nUNIX.[15] It is built on the notion of the resource control block (RCB), as opposed to the process control block (PCB) that is common to many operating systems.

nUNIX partitions the existing concept of a UNIX process into a new variable-weight process and several independent system resource descriptors. It defines new primitives for manipulating those resources in order to create a new thread of execution and to selectively manage system resources associated with it. Different from MACH, this new UNIX variant (nUNIX) preserves the semantics of UNIX processes by considering threads as just another variation on processes and by retaining the existing system interfaces to them. Resources themselves are arbitrarily sharable, or not, by each of these new processes. Surprisingly few changes are needed to achieve this effect, especially in comparison to the requirements of parallelizing the kernel itself, and the design that is offered can properly be considered as a simple extension to UNIX (Fig. 10.3.8).

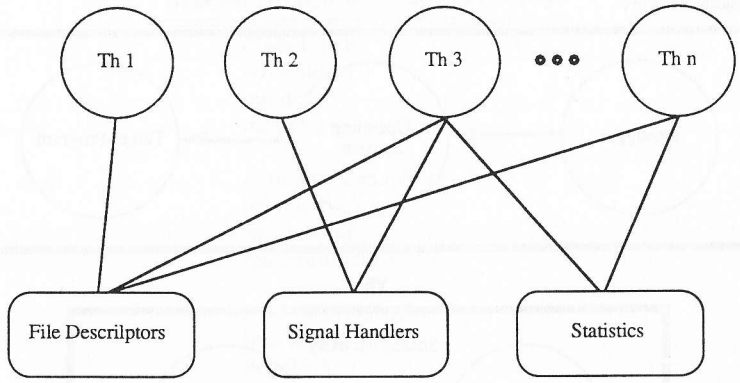


Fig. 10.3.7 Debugging paradigms: shared vs disjoint memory.

Once the aforementioned kernel restructuring was completed, backward compatible programming models were implemented. Implementation of both UNIX fork( ) and MACH-cthread_fork( ) became trivial programs that could fit in one page. In addition, the new interface provided the capability for flexible sharing resources. We are only beginning to learn the power of this flexibility.

Conclusions

Symmetric hardware is only the first and the smallest building block of a fully symmetric parallel processing system. Many layers of system software need to be implemented and modified to conform to the symmetric model. A mistake at any layer of the systems software may introduce a sequential bottleneck (i.e., nonparallelizable code fragment) that will slowdown scalability of the parallel program and may even jeopardize all parallelization efforts at other layers.

This paper describes our experiences in implementing a variety of such systems-level parallel programs. They are complex parallel programs that have been used by many users. The programs include different versions of the UNIX kernels, parallel Ada run time systems, and high-level debuggers.

In software, multithreading appears to be the most common technique for parallelizing all systems software. This technique was the starting point for parallelizing both the UNIX kernel and Ada run time system. It was only a beginning. Many more issues, such as how to avoid deadlocks, how to tune the performance of the multithreaded kernel, and how choose the granularity of the locks in order to optimize the overhead for calling the locks, are emerging.

Choosing the programming model for implementing Ada tasking also involved many trade-offs, ranging from a fully shared-memory model (as in MACH) to a completely private memory model as in standard UNIX. Those trade-offs were further highlighted in the design of the Ada tasking debugger that required a more flexible resource-sharing model that is not supported by either MACH or UNIX.

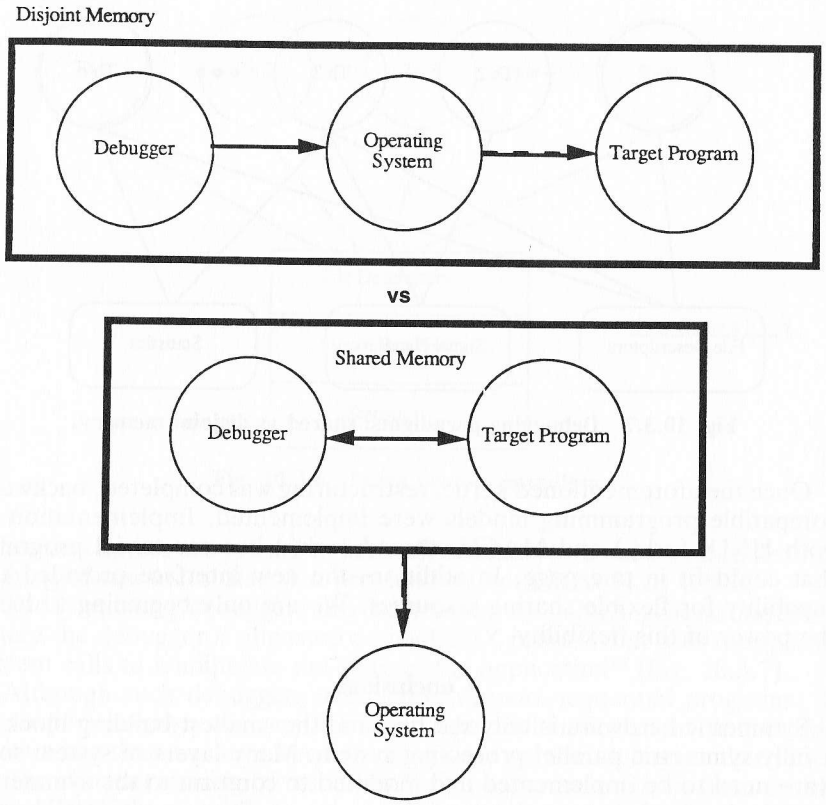


Fig. 10.3.8 nUNIX process paradigm. Flexible allocation of resources.

Although many of the techniques developed appeared to be made on an ad hoc, per case basis, we have identified some principles that occur throughout all layers. Although the implementation techniques may vary, we believe that the same set of issues will have to be addressed in other systems. We are beginning to learn the principles of symmetric parallel processing.

Acknowledgments

This research was supported by the Defense Advanced Research Projects Agency ARPA Order 5875 and was monitored by Space and Naval Warfare Systems Command under Contract N00039-86-C-0158. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government. Parasight and Multimax are trademarks of Encore Computer Corporation. UNIX is a trademark of AT&T Bell Laboratories. Ike Nassi's current address is: Apple Computer, 238 Main St., Cambridge, MA 02142.

Appendix A: Synchronization Primitives

```
/* spin locks */
initmplock( )
mplock( )
mpunlock( )

/* binary semaphores */
initmutex( )
mutexlock( )
mutexunlock( )

/* read/write locks */
initrwlock( )
deadlock( )
readunlock( )
writelock( )
```

Appendix B: Hierarchy of Locks, a code fragment from the header.

```
/** Lock level 0 */
#define PM_RWLOCK 0 /* Read/write lock mutex */
#define MS_USCLOCK 0 /* Mass store device lock */
#define NET_ENETLOCK 0 /* Ethernet device structure lock */
#define TTY_AUXLOCK 0 /* Serial line aux struct lock */
#define TTY_TABLOCK 0 /* Typeahead buffer lock */
#define NET_TCPTIMELOCK 0 /* TCP timer data lock */
#define SYSV_SEMLOCK 0 /* System V Semaphore lock (coarse) */

/** Lock level 1 */
#define TTY_DEVLOCK 1 /* Lock on serial line device regs */
#define NET_ENETCRQLOCK 1 /* Lock on Ethernet CRQ */
#define NET_MBUFLOCK 1 /* Lock of network mbuf pool */
#define NET_TCPSTATLOCK 1 /* Lock of TCP statistics */
#define NET_UDPSTATLOCK 1 /* Lock of UDP statistics */
#define NET_ICMSTATLOCK 1 /* Lock of ICMP statistics */
#define NET_IPSTATLOCK 1 /* Lock of IP statistics */
#define NET_RDPHASHLOCK 1 /* Lock of rdptable has table */
#define NET_SOCKREF 1 /* Socket reference count lock */
#define NET_CMDPOOLLOCK 1 /* eth_cmd_resp pool lock */
#define MS_TMO_QUE_LOCK 1 /* Masstore timeout message queue */
#define MS_SCANLOCK 1 /* Masstore pending command queue */
#define SYSV_IPCLOCK 1 /* System V IPC lock */

/** Lock level 2 */
#define TTY_ACILOCK 2 /* Lock on ACI service bitmask */
#define PM_WAITLOCK 2 /* Insure atomic process exit and */

/* .... */
/** Maximum lock levels */
#define MPLOCK_MAX 10
```

References

[1]*UMAX 4.2 Programmer's Reference Manual*, Encore Computer Corp., Marlborough, MA.

[2]*Balance Guide to Parallel Programming*, Sequent, Inc., Beaverton OR, 1986.

[3]Russell, C., and Waterman, P., "Variations on UNIX for Parallel-Processing Computers," *Communications of the ACM*, Dec. 1987.

[4]Bell, C. G., "Multi: A New Class of Multiprocessor Computers," *Science*, 228, April 1985.

[5]Nassi, L., "A Preliminary Report on the Ultramax: A Massively Parallel Shared Memory Multiprocessor," *DARPA Workshop on Parallel Architectures for Mathematical and Scientific Computing*, July 1987.

[6]Wilson, A. W., "Organization and Statistical Simulation of Hierarchical Multiprocessors," PhD Dissertation, Dept. of Electrical Engineering and Computer Engineering, Carnegie-Mellon University, Pittsburgh, PA, Aug. 1985.

[7]Bach, M., and Buroff, S., "Multiprocessor UNIX Operating Systems," *AT&T Bell Laboratories Technical Journal*, Vol. 63, Oct. 1984, pp. 1733–1749.

[8]Hamilton, G., and Code, D., "An Experimental Symmetric Multiprocessor Ultrix Kernel," *Proceedings of the Winter USENIX Conference*, 1988.

[9]Nassi, I., and Habermann, N., "Efficient Implementation of Ada Tasks," Carnegie-Mellon University, Pittsburgh, PA, TR-CMU-CS-80-103, 1980.

[10]Doeppner, T., "Threads—A System for the Support of Concurrent Programming," Brown University, Providence, RI, TR-CS-87-11, June 1987.

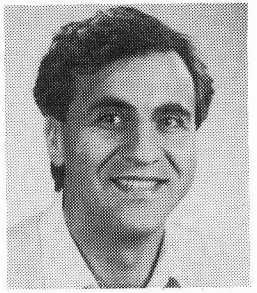
[11]Tevanian, A., Jr., Rashid, R., Young, M., Golub, D., Thompson, M., Bolosky, W., and Sanzi, R., "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach," *USENIX Conference Proceedings Summer 1987*, June 1987.

[12]Adams, E., and Muchnick, S. S., "Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations," *Software—Practice and Experience*, Vol. 16, No. 7, July 1986, pp. 653–669.

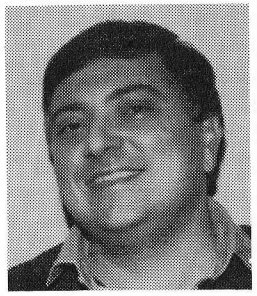
[13]"The Common Object File Format," *UMAX Support Tools Guide*, Encore Computer Corp., Marlborough, MA, Chap. 8.

[14]Aral, Z., and Gertner, I., "Efficient Debugging Primitives for Multiprocessors," *ACM/SIGPLAN ASPLOS 1989—Third International Conference on Architectural Support for Programming Languages and Operating Systems*, May 1989.

[15]Gertner, I., Langermanz, A., and Aral, Z., "Variable Weight Processes with Flexible Resources," *USENIX Conference Proceedings Winter, 1989*, June 1987.



Dr. Ilya Gertner is currently a Senior Consultant at the Parallel Systems Division of Encore Computer Corporation. He is one of the principal developers of Parasight, a parallel programming environment, and nUNIX, a variable-weight process implementation of UNIX. Dr. Gertner received a BS and MS from Technion, Israel, and a PhD from the University of Rochester, where he worked on RIG, a pioneering message-based distributed operating system that led toward the development of MACH.



Dr. Ike Nassi has extensive experience in programming languages and systems, computer architecture, and distributed systems. At Encore Computer Corporation, he held the post of Vice President of Research, where he was also the principal investigator for a research project sponsored by DARPA to develop a general purpose shared memory multiprocessor capable of delivering 1000 MIPS. An early prototype of this system was demonstrated to DARPA in May 1989. Dr. Nassi is currently the Director of Research and Technology for Apple Computer, Inc.