

Variable Weight Processes with Flexible Shared Resources ^{*}

Ziya Aral [†] James Bloom [‡] Thomas Doeppner [‡] Ilya Gertner [†]
Alan Langerman [†] Greg Schaffer [†]

Encore Computer Corporation and Brown University

Abstract

Traditional UNIX processes are inadequate for representing multiple threads of control in parallel programs. They are inflexible in their resource allocation, unable to cleanly share system resources, and they carry a heavy overhead. Some new operating systems, such as MACH, split the process into multiple light-weight threads of execution and a task which defines their resource set. This paper addresses the same problem within the confines of UNIX process semantics. It partitions the existing concept of a process into a new “variable weight” process and several independent “system resource descriptors”. Processes pay creation and maintenance costs only for resources they wish to keep private. Surprisingly few changes to the kernel are needed to achieve this effect. The design can properly be considered as a simple concurrency extension to UNIX. Variable-weight processes also enhance the effectiveness of the cooperating process paradigm and so are broadly applicable to both uniprocessor and multiprocessor UNIX implementations.

A prototype kernel has been implemented on MultimaxTM, a shared-memory multiprocessor. Preliminary experience indicates that relatively few changes to the process structures in UNIX make this strategy incrementally applicable to a range of UNIX variants.

Introduction

The design of a symmetric multi-threaded UNIX kernel on a shared memory multiprocessor is by now a problem that is well understood [2, 1, 13, 11, 8]. Much less clear are the appropriate abstractions for scheduling and resource management that should be provided by the kernel to support both sequential and multi-threaded programs. What are the characteristics of a system that gives the full capabilities of the multiprocessor hardware to the user, yet remains a system that most would agree is UNIX? In fact, why do we want a system that is UNIX, at all?

UNIX semantics are very well understood by a large number of people. There is a lot of documentation and instructional material on UNIX. There are a large number of UNIX implementations; UNIX has become something of a commercial standard for large systems. Finally, UNIX has proven to be remarkably adaptable to the demands of over a decade of advances in computer science. If support for parallelism can be integrated into UNIX in a clean way so as not to interfere with existing

^{*}This research was supported in part by the Office of Naval Research under Contract No. N00014-88-K-0406; and in part by the Defense Advanced Research Projects Agency (DoD) through ARPA Order No. 5875, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-86-C-0158; and through ARPA Order No. 6320, monitored by the Office of Naval Research under Contract No. N00014-83-K-0146.

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.

[†]Encore Computer Corporation, 257 Cedar Hill Street, Marlborough, Ma. 01752-3004.

[‡]Department of Computer Science, Brown University, Providence, RI 02912-1910.

features and yet be in the style of those existing features, the task of adopting concurrency features is made far easier. In this paper we argue that UNIX may be easily modified to support concurrent programming, both on uniprocessors and on shared-memory multiprocessors.

In both its AT&T and Berkeley incarnations, UNIX's most obvious shortcoming with respect to multiprocessing is the cost of processes. Processes are the most fundamental abstraction of UNIX, and are the only OS-supported execution units. They are, however, very expensive to create and to schedule. Thus it is costly for a programmer to make use of multiple processors, since doing so requires the use of multiple processes.

This paper assumes that concurrency support must address processes and thus must include kernel modifications. We have much experience in supporting multiple threads of control at the user level (using user-level schedulers) and do not question the usefulness of this approach [5, 10]. There will always be a need to support application-specific approaches to concurrency that go beyond what the OS can be expected to provide. The exact division of labor between user-level and kernel-level concurrency support is an important topic in its own right, which we will address in the future. However, the fact that these packages must overcome deficiencies in UNIX partly motivates this work.

Most versions of UNIX support some form of memory sharing among processes, a feature necessary for multiprocessor applications. Memory is not the only resource that must be made sharable. There is much discussion in the literature of the need for sharing other resources, such as file descriptors and signal handlers [4, 9, 11, 12].

Our work on monitoring and debugging parallel programs leads us to agree [3, 7]. Parallel programming involves the cooperation of a number of threads of control. In some cases, these threads need to share all resources, but in other cases there should be restrictions on this sharing. A debugger needs write access to the debuggee's text segment, but the debuggee should have only execute access. Timers may be associated with a single process or a group of processes. Notification of external events such as I/O completions or the typing of interrupt characters in some instances might be sent to each of many potential interested processes, in others be handled by one process on behalf of a group, and in yet others result in the creation of an ad hoc process. The main point is that the resource sharing provided by most, if not all existing UNIX and variant systems is much too inflexible.

Where support for multiple threads of control which share resources exists, it is assumed that these threads are identical. While the notion of *homogeneous* threads of control, each sharing the same system resource set and executing the same code may be adequately supported, the idea of *heterogeneous* threads of control is not. At best, such heterogeneity is achieved only by falling back on multiple processes, with added overhead, complexity, and semantic incompatibilities resulting from it.

In our experience, the typical case is that of multiple threads of control which want to share memory but not necessarily share other resources. Even when working with a homogeneous paradigm, such examples proliferate. A certain degree of support for heterogeneity is almost always useful and often crucial.

UNIX can accommodate the necessary support for parallel processing without drastic changes to the fundamental notion of process. Rather than completely divorce the thread of control from the set of program resources, the approach we describe will permit *variable-weight* processes spanning the spectrum from *homogeneous* threads of control which share all resources, to *heterogeneous* threads of control which share some, but not all resources, to *heavyweight* processes, which share no resources.

This paper focuses on issues of flexible resource sharing for parallel programs. The implementation of a symmetric multi-threaded kernel and the synchronization primitives associated with it is the starting point for our work. Our base operating system is UMAX4.2, a commercially available, multi-threaded variant of 4.2BSD.

To further narrow the domain, we start with the following assumptions:

1. The "cost" of traditional UNIX processes prohibits their use as the basic building block for all but the coarsest parallel programs. This cost is paid not only on start-up, but throughout a process's lifetime in increased kernel memory requirements, increased paging traffic, and longer process-switching times.

2. The notion that every process maps to a separate program with a distinct address space and resource set is not desirable for parallel programs.
3. Existing kernels, whether intended for uniprocessors or multiprocessors, have inadequate and inflexible mechanisms for sharing system resources and their associated overhead among processes.
4. The key to useful concurrency support is the modification of UNIX processes to reduce their cost and promote flexibly shared resources.
5. Where possible, the preservation of UNIX semantics is very important. Further, the preservation of compatible semantics and a common interface for both parallel and serial programs is equally important.

UNIX Processes

UNIX processes represent the wrong model for parallel programming [12]. They are an artifact of a timesharing environment in which processes were expected to map to programs and the priority for the operating system was the segregation of such programs from each other. Concurrency support was explicitly confined to the kernel. The interprocess communication facilities of UNIX were designed to support low-bandwidth channels among independent programs rather than efficient communication within them. Virtual memory was later implemented with the assumption of a separate address space mapped to each process. UNIX's ancillary subsystems, however, have been amenable to modification or replacement over time [14]. The concept of "process" has proven to be more fundamental; it has remained essentially unchanged.

The problems inherent in the process model for multiprocessing are twofold:

1. Traditional processes are inherently "heavyweight", requiring a complete and independent system context including an address space. As a result, they take up significant space, are time-consuming to create and to delete, and take considerable time to switch.
2. Processes are not designed to share resources. [4].

Most parallel variants of UNIX implement facilities to support resource sharing and parallel programming. In their most advanced form, such implementations create a second execution abstraction to coexist with processes and to support shared resources, lower system cost, or both. One proposal introduced *share groups*, whereby processes share a common resource set [4]. While this implementation addresses the issues of a shared address space and I/O descriptors, it takes a narrow view of what system resources should be sharable and does not address at all the "weight" of processes. In fact, it actually adds to that weight by creating the higher-level abstraction of the share group. Furthermore, arbitrary sharing is not supported: share groups do not provide for heterogeneity.

A much more radical approach is taken by MACH. MACH defines a new "lightweight" unit of control, implemented at kernel-level and explicitly designed to support parallel programs [11]. MACH splits the UNIX process into a "thread," which defines a processor context (registers and stack), and a "task," which defines a system context as a collection of system resources. Lightweight control units are implemented by instantiating multiple threads per task. A much broader view of what resources may be shared is taken. All of the threads associated with a task share all of the resources allocated to that task. Conversely, the combination of one thread plus one task is the equivalent of a process and allows emulation of traditional process semantics.

While lightweight threads thus coexist with heavyweight "processes" (literally, single-threaded tasks), some penalty is paid. Such a scheme creates a second set of semantics for threads, incompatible with process semantics, and requires that all threads automatically share all system resources defined by their associated task. In MACH, heterogeneity is supported only by recourse to multiple tasks, the equivalent of multiple "heavyweight" UNIX processes. In addition, MACH is predicated

on an entirely new kernel: to the extent that concurrency is supported, it is not UNIX; to the extent that it is UNIX, no concurrency extensions are available.

We attempt to solve the problems cited above while preserving UNIX process semantics. The motivation for this research was our experience with parallel programming environments, applications, and user-level threads systems. This experience suggested the need not only for greater flexibility in the sharing of resources but also for simplicity and consistency between interfaces for serial and parallel programming. In addition, we saw the need for a general mechanism which could be incrementally developed as further experience with multiprocessing is gained. Surprisingly few changes to UNIX are needed to achieve these goals and the design that is offered can properly be considered as a simple extension to UNIX.

We partition the existing concept of a UNIX process into a new “variable-weight” process and several independent “system resources”. We define new primitives for manipulating those resources in order to create a new thread of execution and to selectively manage system resources associated with it. Resources can be arbitrarily shared, or not, by each of these new processes. All existing system interfaces are retained. These processes are as lightweight as MACH threads when they share all resources, and as heavyweight as traditional processes when all resources are private to the process.

This mechanism supports standard UNIX processes and lightweight threads; in addition, it has the flexibility to create heterogeneous threads of execution that share some resources and leave others private.

Figure 1: UNIX PROCESS and MACH reconfiguration

nUNIX — A Resource-Oriented Kernel

A prototype kernel, which we will call nUNIX to distinguish it from more conventional variants, implements this capability. The approach may be thought of as being conceptually similar to that of MACH, although the implementation is practically the opposite. Like MACH, this kernel breaks up the process data structures. Instead of breaking out the runtime context from the process descriptor, however, it breaks out the individual resource descriptors. These in turn acquire an existence independent of any particular process.

A UNIX process is described by two data structures, PROC and USER. The partitioning of process information between the two structures is of importance only to the extent that the former contains data which must always remain resident in core while the latter is swappable. For all

Figure 2: Resource Descriptor PROCESSES

practical purposes, we can abstract a merged structure that defines processes. This structure, in turn, is an amalgam of various independent resources whose only commonality is that together they represent the system environment of the process.

The basis of the nUNIX implementation is the incorporation of an additional level of indirection in the resource descriptor fields of the process data structures. Instead of declaring process resources, such as file descriptors, in-line, the process data structures in nUNIX contain pointers to resource descriptors external to the process itself. These resource descriptors are allocated independently and are sharable across processes (Figure 2).

The resource descriptors themselves include a reference counter. Each process that attaches to a descriptor increments its reference count. Conversely, each process that terminates or detaches from a resource decrements its reference count. The descriptor persists only so long as its reference count does not reach zero. New resource descriptors are allocated dynamically from the UMAX 4.2 kernel's non-paged memory pool.

The classes of resource descriptors currently implemented in nUNIX are the six categories grouped in the USER structure: address space, file descriptors, signal management, usage control, priorities and protections, and process administration statistics.

Control of these independent resource descriptors is quite straightforward. The **fork()** system call causes the creation of new process data structures and new resource descriptors referenced by them (which are copied from the parent's resources). Very little extra overhead is implied by the addition of another level of indirection, which in any event lends itself to being optimized away by the compiler. The existence of independent resource descriptors is invisible not only to user-level applications, but to much of the kernel itself. Alternatively, two new system calls provide for customization of the resource set associated with a process.

A new interface called **sfork()** creates the equivalent of a kernel-level thread. The actual implementation falls out of the restructuring of resource descriptors. **sfork()** is merely a **fork()** in which the resource pointers of the parent's process data structure are copied and the reference count of the individual resources incremented instead of the resources themselves being copied and initialized. The resulting thread of control is extremely lightweight but remains a UNIX "process," indistinguishable from any other. All existing system calls including **wait()**, **exit()**, etc. work as usual. **fork()** and **sfork()** may be freely mixed with no incompatibilities.

A second system interface called **resctl()** implements direct control over resource descriptors. **resctl()** is used to detach from an existing descriptor and to create a new one as well as to implement resource controls not supported by the existing UNIX process interface. All existing UNIX system calls that affect process resources work as usual with the qualification that all processes sharing that particular resource descriptor are impacted similarly.

Using the nUNIX interface, arbitrarily complex combinations of private and shared resources may be constructed by processes executing the same program. Suppose, for example, that two separate sets of file descriptors are desired among a number of processes which otherwise wish to share their environment. **sfork()** is used to create the processes while **resctl()** creates and attaches the second file descriptor resource. The cost of this configuration is increased only by the expense of creating a second array of file descriptors and the overhead of the system call itself.

The new system calls are:

sfork()

Create a new execution environment that includes registers, program counter, scheduling data and pid. Attach this process to the current resources of the parent.

resctl()

General control function for system resource descriptors.

A higher-level library interface to **resctl()** is also provided:

rcreate()

Create and initialize a new resource descriptor of specified type. Attach current process to this resource and detach the previous resource.

rcopy()

Create a new resource descriptor of specified type. Copy the state of the previously attached resource of this type. Attach current process to this resource and detach the previous resource.

This handful of additional system calls is quite flexible. The existing process interface may, for example, be defined in terms of the new primitives. The following pseudo-code segment describes the implementation of **fork()** in terms of **sfork()**, **rcopy()**, and **rcreate()**. All resources are copied from the parent, except for usage statistics.

```
fork()
{
    int pid;

    if ((pid = sfork()) == 0) {
        /* I am the child */
        rcopy(SF_MEM);
        rcopy(SF_IO);
        rcopy(SF_SIG);
        rcopy(SF_PROT);
        rcopy(SF_UCTL);
        rcreate(SF_STAT);
    }
    /* Both parent and child come here. */
    return(pid);
}
```

Preliminary Experience

The nUNIX kernel is still in its early stages and is intended mainly as a platform for experimentation with variable-weight processes and independent resources. Many limitations and unresolved issues remain.

The `sfork()` primitive provides an example of these issues. In order to maintain compatibility with `fork()` and allow a thread of control to return back up the call stack after an `sfork()` call, the parent's stack is copied into the new stack area of the child. For this scheme to work, pointers into the stack cannot be absolute references but must remain relative to the stack pointer. Currently, this is enforced only by convention. For the future, the decision must be made whether to enforce this limitation through compiler support or to disallow returning up the call stack, thus sacrificing orthogonality. Other, lesser issues of this type also remain.

Furthermore, nUNIX's resource sets need work. The existing partition was adapted merely as a convenience, because it corresponded to the implied partitioning of resources in the current process data structures. Many changes to the existing design are contemplated including separate memory management and memory protection resources, the independent partition of process timers, and others. Thought has even been given to new classes of resources such as symbolic information and efficient communication areas shared between user and kernel. In addition, consideration will be given to making these resource pageable.

Semantic ambiguities such as those encountered in MACH [11, 6], also remain. For example, what does it mean to send a signal to a process which shares a signal management resource? Does only that process receive it, or is it broadcast to all other processors sharing that resource, or is the signal taken by only the first available process which shares that resource? nUNIX has no limits on processes or resources to prevent any of the above strategies from being the default, but which is the "correct" approach?

Finally, the exact characteristics of the kernel are difficult to quantify without resolving many of the issues discussed above. In future publications, we hope to report timing data on the new constructs.

Despite these limitations, preliminary experience with the nUNIX primitives has been very encouraging. nUNIX supports a range of programming models and conforms to existing UNIX semantics. The system has proven itself to be particularly suited to an incremental implementation.

One unexpected result of nUNIX was that shared resources also appear to enhance the co-operating processes paradigm. In one example, almost 3 dozen NFS daemons simultaneously execute on our system. Because these daemons are stateless, they conform to the classic process model. It has proven useful, however, to share process statistics across them. A modified version of the utility `ps` shows statistics resources rather than pids and thus makes process management more rational. Instead of 34 entries for NFS daemons, we now see one.

nUNIX supports lightweight threads and UNIX processes within the same paradigm. Further, nUNIX processes may share some resources and have others private: the additional cost is paid only for those resources which are private in a "pay for what you use" strategy. Thus we have efficient threads for parallel programming, and still lose none of the strengths of conventional processes. This is a significant result that we hope can extend the life of UNIX.

A prototype kernel is being implemented on MultimaxTM, a shared-memory multiprocessor running both MACH, a new 4.3 BSD compatible kernel, and UMAX 4.2 a conventional (parallelized) adaptation of 4.2 BSD, upon which our prototype is based.

Conclusion

UNIX can accommodate the necessary support for parallel processing without requiring drastic changes to the fundamental notion of process. Rather than completely divorce the thread of program control from the set of program resources, our solution permits variable-weight processes, spanning the spectrum from multiple threads of program control executing within the same resource space (equivalent to the MACH notion of "threads" grouped into a "task") to threads sharing a subset

of their resources (such as file descriptors) to multiple threads executing with completely disjoint resource sets (equivalent to the “traditional” notion of a UNIX process).

The benefits of this approach are several:

- the notion of resource becomes orthogonal to the notion of thread execution,
- the kernel can support a range of processes, from a large number of very lightweight processes to a much smaller number of “traditional” processes,
- user-level applications and thread packages become significantly easier to implement,
- the user interface is uniform across process of all weights,
- the implementation is simple and clean.

Making efficient use of shared-memory parallel processors does not necessarily require the use of radically new operating systems or kernel interfaces. The fundamental UNIX concepts of process and resource need only be generalized. nUNIX represents one attempt at such a generalization and underscores yet again the remarkable adaptability of the UNIX Operating System.

References

- [1] *Balance Guide to Parallel Programming*. Sequent Inc., Beaverton Oregon, 1986.
- [2] *UMAX 4.2 Programmer's Reference Manual*. Encore Computer Corporation, Marlborough, MA 01581-4003.
- [3] Z. Aral and I. Gertner. Non-intrusive and interactive profiling in Parasight. In *ACM/SIGPLAN PPEALS 1988 — Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, New Haven, Connecticut, July 1988.
- [4] J. Barton and J. Wagner. Beyond threads: resource sharing in UNIX. In *Winter 1988 Usenix Conference Proceedings*, February 1988.
- [5] T. Doeppner. *Threads - A System for the Support of Concurrent Programming*. Computer Science Technical Report CS-87-11, Brown University, June 1987.
- [6] T. Doeppner. *A Threads Tutorial*. Technical Report CS-87-06, Brown University, March 1987.
- [7] T. Doeppner and D. Johnson. A multi-threaded debugger, extended abstract. In *ACM Workshop on Parallel and Distributed Debugging*, University of Wisconsin-Madison, May 1988.
- [8] G. Hamilton and D. Code. An experimental symmetric multiprocessor Ultrix kernel. In *Proceedings of the Winter USENIX Conference*, 1988.
- [9] I. Nassi. Encore's Parallel Ada Implementation. 1988. Private Communications.
- [10] I. Nassi. A preliminary report on the Ultramax: a massively parallel shared memory multiprocessor. In *DARPA Workshop on Parallel Architectures for Mathematical and Scientific Computing*, July 1987.
- [11] R. Rashid. Threads of a new system. *Unix Review*, August 1986.
- [12] D. Ritchie. Computer Science Seminar given at Carnegie-Mellon University, 1988.
- [13] A. Tevanian, Jr., R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. Mach threads and the unix kernel: the battle for control. In *USENIX Conference Proceedings Summer 1987*, June 1987.
- [14] A. Tevanian, Jr., R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. A UNIX interface for shared memory and memory mapped files under Mach. In *USENIX Conference Proceedings Summer 1987*, June 1987.