Parallelizing Signal Handling and Process Management in OSF/1 *

Don Bolinger
bolinger@encore.com

Shashi Mangalat
shashi@encore.com

Mach Operating System Project
Encore Computer Corporation
257 Cedar Hill Street
Marlborough, MA 01752

Abstract

Release 1.0 of the OSF/1 operating system, despite its high degree of parallelization, left several dozen system calls unparallelized. The most important subsystems not converted were process management and signal handling. This paper describes the project to make these subsystems multiprocessor-efficient, and to make their system calls usable within multi-threaded tasks. After presenting background on OSF/1 and on the relevant system calls, we describe the general approach and specific changes we adopted for the parallelization, and for the adaptation of Unix process-oriented abstractions to the multi-threaded programming model of OSF/1. After providing rationales for our most important choices, and comparing them to a few discarded alternatives, we look at how some common operations are implemented in the resulting kernel, examining the resolution of races and other synchronization problems introduced by our changes. Finally, we present data on performance improvements introduced by the project, and indicate a few possibilities for useful future development.

1 Introduction

The OSF/1 operating system consists of two major parts. The core of the system is the native Mach kernel, which provides the mechanisms needed for operation in a distributed environment using either uniprocessor or multiprocessor (MP) hosts. In addition, OSF/1 (like Mach itself) contains code to emulate the programming (system call) interface of a BSD Unix operating system, in this case the 4.3BSD-Reno release.

The native Mach kernel, of course, is fully parallelized for use on tightly-coupled shared-memory multiprocessor (SMP) architectures. The original 4.3BSD-Reno code used for Unix emulation, however, was designed to run on a uniprocessor, and uses interrupt-level (spl) based synchronization mechanisms that fail in an SMP environment. Of this code, the most heavily-used subsystems — notably the networking and TTY subsystems and all filesystem code — were fully parallelized before Release 1.0 [9, 2]. The remaining emulation code, however, is unparallelized, and must run on a single master processor, which is permanently designated at boot time. Use of this code therefore

*Multimax, UMAX4.3 and UMAXV are trademarks of Encore Computer Corporation. Unix is a trademark of AT&T Bell Laboratories. OSF/1 is a trademark of the Open Software Foundation.

obliges a thread to wait for and execute on the master, an operation implemented by the function **unix_master**, and which we will refer to by that name.

A major goal for Release 1.1 of OSF/1 was to eliminate as many of the remaining uses of **unix_master** as possible, since they degrade system performance not only by serializing the execution of unrelated threads, but also by forcing otherwise-needless context switches to the master CPU. Of the kernel code that had not already been parallelized, the Unix system calls relating to process management and signal handling, along with the kernel code involved in signal generation and delivery, were seen as the most urgent candidates for conversion. Some of these calls (like *sigvec* or *sigaction*) are used very frequently. Others (like *fork* and *exit*), though less common, take a very long time to execute.

Parallelizing these system calls, and the associated signal processing code, involved both making them MP-efficient, and making them thread-safe, that is, suitable for use in a multi-threaded task. Thus we enabled the calls to be run on any processor in an SMP system by adding appropriate locks and other synchronization mechanisms. After ensuring that the calls obeyed the appropriate specification (BSD or POSIX) in single-threaded use, we then implemented correct behavior in multi-threaded tasks, as well.

We did not seek to provide conformance with any particular model of multi-threaded task behavior, but rather tried to emulate single-threaded semantics as closely as possible. Where changes in multi-threaded behavior were unavoidable in achieving this goal, we used a recent draft document (draft 5) of the POSIX 1003.4a (Pthreads) working group as a guide in our own implementation.

Though references exist in the literature to parallelization efforts involving BSD-derived Unix kernels [2, 11, 6]), and to process migration schemes to enhance the performance of process manipulation on MP architectures [5, 4], we are unaware of any detailed description of the problems involved in parallelizing Unix process management for use in an SMP environment, and of how to resolve these problems effectively. We hope that sharing our experience may be of use to others faced with similar tasks, in providing some data to simplify the undertaking.

2 Background

2.1 Related Unix system calls

The main process management system calls treated by this project were the classic Unix quadrumvirate: *fork*, *exec*, *wait* and *exit*. We assume that the reader is familiar with them, so we will not explain their operation here. As it happens, many other, less important system calls in this area (like *getpriority* and *setpriority*) were also parallelized in the course of our work, but their cases are not of sufficient interest to merit inclusion here.

It may be more useful, however, to recap the Unix signal mechanism and a few associated abstractions before going further.

2.1.1 Signals

A signal is a software interrupt, sent to a given active process to indicate some exceptional condition (either hardware- or software-related). Though they were initially meant to provide a mechanism for explicit manual intervention in resolving errors (see [8] for an example of how primitive early signal implementations were), signals in modern-day Unix systems are used for many purposes (look at [1, 7], for instance, for a full description). A few uses of signals that were particularly important to our project are:

- to indicate a resource limit has been exceeded
- to suspend or unsuspend a process
- to indicate that a process without access to the terminal wants to do input or output

A signal may be generated either by a user application, via the *kill* system call, or within the kernel. Signal generation does not directly change the execution state of the targeted process — instead the signal is recorded (marked as pending) for receipt by the process at a convenient point.

With some exceptions, a process may choose to ignore a signal, to let it retain its default behavior, or to associate a handler function with it. By default, most signals terminate the targeted process, though a few by default are ignored. A process may usually choose to mask a signal temporarily without affecting how the signal will later be handled — the signal will cause the action specified as soon as it is unmasked.

Three concepts associated with signals are particularly relevant to the changes we made in the course of our project. These are process groups, job control, and sessions.

2.1.2 Process groups

A *process group* is a mechanism for identifying a set of processes that can be signalled as a unit, as a way of ensuring that all processes in the group receive a given signal more-or-less simultaneously. (Obviously, each process in the group can still elect to handle it differently.) The kernel never changes a process' group internally — this is done only by an explicit system call. The most common use of process groups is by shell programs, which generally put each command line entered by the user into its own group.

Any signal can be posted to a process group — one common user action is to issue a `SIGSTOP` or `SIGKILL` to a group in order to suspend or terminate its execution. Within the kernel, some of the most important uses of process groups are in the terminal handling (TTY) code. This code uses process groups as the basis for multiplexing access to a terminal between potentially many different groups. This multiplexing is known as *job control.*

2.1.3 Job control

Each command line executed by a shell that implements job control is known as a *job.* The shell process itself can also be considered to be a job, distinct from any other ones. At any given point, a single job has control over input from or output to the user's controlling terminal — this job is the *foreground job*, and its group is the *foreground process group.* If a user executes commands sequentially, without any manual signal generation, then foreground status alternates between the shell (when the user is entering a command), and each command line executed.

When commands are run asynchronously — that is, "in the background" — they do not have foreground status. If such a command tries to interact with its terminal, it will be suspended, and will not continue until the user brings it to the foreground (by instructing his shell to make it the foreground job). Similarly, a user can explicitly put the current foreground job into the background by suspending it, then continuing it, disassociated from terminal.

2.1.4 Sessions

Intuitively speaking, a *session* consists of all of the processes started during a given login session. Naturally, therefore, a session may contain several process groups, which vary over time. Generally,

a session is associated with a single controlling terminal, access to which is given to one process group (or job) at a time, using the shell facilities described above.

2.2 Synchronization overview

The native Mach kernel and the parallelized sections of the emulation code rely on two varieties of multiprocessor lock to synchronize operations between different processors. The first type of lock, called a *simple lock*, is implemented as a spin lock. A thread waiting to acquire such a lock will spin in a tight loop till it gets the lock.

The other style of lock is a blocking lock. A thread waiting to acquire this kind of lock will be blocked (i.e., will no longer execute) until the lock becomes available. A blocking lock may be used either as a *read/write lock* or as a *mutual exclusion (mutex) lock*. In the first case, multiple readers, but only one writer, may hold the lock at any given time. In the second case, only one thread may ever hold the lock at a time.

Our goals in applying these primitives can be summarized as follows. Predictably enough, some of them are mutually-conflicting.

- Minimize the scope of source-level changes.
- Make the changes automatically adaptable to different hardware configurations (e.g., uniprocessor vs. MP).
- Minimize dynamic lock scope, as by using reference counts.
- Maximize parallelism in application-visible interfaces, as well as internal to the kernel, on behalf of multi-threaded tasks.

In particular, the great majority of the synchronization mechanisms we added for use on an MP system are not needed (and are not compiled into the kernel) in ordinary uniprocessor configurations — where "ordinary" means that the user has not explicitly requested the inclusion of MP locks for debugging purposes. Details on whether a given lock disappears in uniprocessor kernels will be given when it is introduced.

3 Process and Signal Management in OSF/1.0

Among the main process management system calls treated by this project, only *exec* was already MP-efficient in OSF/1.0. Even this call required some changes for safe use in multi-threaded tasks — see Section 4.1.

In the area of signal handling, OSF/1 provides the full set of 4.3BSD-derived system calls, as well as the system call interface required by POSIX. None of these calls was parallelized in OSF/1.0, nor was the related signal code internal to the kernel. Making this subsystem MP-efficient involved changing the system call entry points involved, modifying the signal generation (`psignal`) and delivery code (`issig`, `psig`, `sendsig`), and also making excursions into the TTY subsystem and the clock interrupt handler.

Since OSF/1 provides for multiple threads of execution within an emulated Unix process, OSF/1.0 already contains several internal modifications to conventional (single-threaded) signal handling, designed to make signal handling safe in multi-threaded processes. (Most of these, and in particular the two that we will discuss here, were implemented by David Black, currently of the OSF Research Institute, for an early version of Mach.)

3.1 Signal typing

The first of these modifications is that signals are divided into per-thread and per-process types. A per-thread signal type is one directly caused by the execution of a given thread (like `SIGILL` or `SIGSEGV`). It corresponds to an exception, and is delivered synchronously to the thread in question. A per-process signal is one not caused by the execution of the targeted thread, and is delivered asynchronously to the "first" (i.e., oldest active) thread in the process.

3.2 Signal delivery in multi-threaded tasks

Another feature present in OSF/1.0 ensures that all threads in a multi-threaded task interact properly with the *exit* and *ptrace* system calls — that is, that they are properly stopped and restarted (where appropriate) when a process is being traced, and are cleanly terminated when it exits. This feature is based on a set of multi-threaded signal delivery macros, organized as follows.

The macros are based on a simple lock, called **p_siglock**, and on two other pieces of process state: a flag, called **p_sigwait**; and a thread pointer, **p_end_thread**. Together, these fields define the following states:

locked – **p_siglock** acquired. A thread is in this state only for short periods, in order to query or change the state of the other fields.

unlocked – **p_siglock** not acquired, **p_waiting** and **p_sigwait** zero. Process is not being traced, and no thread is exiting.

waiting – **p_sigwait** non-zero. A thread has already suspended the current task on behalf of a parent process that is tracing the current process using *ptrace*. No further signal processing may occur till the parent continues execution of this process.

exiting – **p_end_thread** non-zero. The thread indicated in **p_end_thread** has begun to exit, and no further signal processing should occur.

The following macros in OSF/1.0 use the above state to provide multi-thread synchronization:

sig_lock_or_return takes **p_siglock** to check whether current state is **waiting** or **exiting**. If not, control passes out of the macro with the lock still held. If so, the macro releases the lock and blocks or terminates the current thread as appropriate.

In reality, this macro merely marks the thread so that it will be blocked or terminated on its return to user mode, then forces a return from the function containing the use of the macro. It thus assumes that that no unintended thread manipulations will intervene before the thread is blocked or terminated. All current uses of the macro are in system call entry points, so no significant kernel processing occurs between the return of the containing function, and the actual change to the thread state.

sig_lock_to_wait assumes that **p_siglock** is locked. It asserts **p_sigwait**, then releases **p_siglock**. Used to await continuation by parent when process is being traced with **ptrace**.

sig_wait_to_lock assumes that **p_sigwait** is asserted. It acquires **p_siglock**, then deasserts **p_sigwait**. Used to co-ordinate the continuation of all threads when parent requests it.

sig_lock_to_exit assumes that **p_siglock** is locked. It registers the current thread as **exiting**, then halts all other threads.

4 Changes to Provide Multi-Threaded Correctness

As noted above, OSF/1.0 already contained some significant changes to ensure the predictable operation of multi-threaded tasks. In this section, we describe how we modified these to apply uniformly to the full process management and signal handling subsystem.

4.1 Multi-threaded process manipulation

First off, we extended the **sig_lock** macros described in Section 3.2 so that they applied to all process manipulation system calls.

As we've seen, the existing macros caused all threads to halt when any one thread called *exit*. We extended this mechanism so that calls to *exec* would have the same effect. We also defined an interface between this mechanism and *fork*, such that any *fork*'s in progress would complete (but no new ones could start) after either *exec* or *exit* had been called. A further description of this interface is given in Section 6.2.

Finally, in order to make multi-threaded signal handling more efficient, we introduced two macros, to repackage the transition between the **unlocked** and **waiting** states:

sig_wait_lock – invoke **sig_lock_or_return**, then assert **p_sigwait** and release **p_siglock**.

sig_wait_unlock – acquire **p_siglock**, deassert **p_sigwait**, then release **p_siglock**.

These macros avoid prolonged busy-waiting to acquire **p_siglock**, which could otherwise occur at a few points in the signal delivery code. Instead, a calling thread will either acquire the lock and put the process into the **waiting** state, or it will see this state already set, and block (rather than busy-waiting) until the state changes. The new macros effectively add another meaning to the **waiting** state of the multi-threaded signal delivery macros — which is simply that another thread is delivering a signal, and the current thread must wait before doing so.

This optimization was not necessary when signal delivery had to be performed on the master processor, since waiting to run on the master would cause a thread to block in an equivalent manner.

4.2 Per-thread suspension of execution

We have also introduced a new per-thread saved signal mask, among other state, in order to permit *sigsuspend* to operate correctly on single threads within a multi-threaded task. In a similar vein, we have ensured that only one thread can successfully *wait* for a given process to terminate or stop, and that all other threads waiting for a given process (or the last of a set of processes) will return an error indication, rather than waiting fruitlessly. (This is also explained further in Section 6.2.)

These two issues (correct per-thread suspension of execution and safe multi-threaded execution of process management calls) were the only ones related to multi-threaded tasks that we felt obliged to address (because the existing implementation was incorrect). Given the rapid evolution of proposed standards in this area, we felt it best to exclude from the project other changes to aspects of multi-threaded execution. The most obvious of these would have been adding more per-thread signal state, or augmenting signal delivery semantics.

5 Changes to Provide Multiprocessor Synchronization

We made two sorts of changes in this area. To begin with, we added locking, a reference count, or both to the various data structures associated with a process. We also introduced several global locks in order to synchronize access to kernel tables or other global structures.

The global locks we added are listed in Table 1. All but the last of these do the obvious in co-ordinating access to the table with which they are associated, and will not be further described here. Of these locks, only the **pgrphash_lock** exists in uniprocessor configurations.

Lock Name	*Lock Type*	*Associated Data*
pgrphash_lock	R/W	process group hash table
uidhash_lock	simple	process uid hash table
pid_lock	simple	table of per-process "handles"
proc_relation_lock	R/W	all process relation pointers

Table 1: *New global locks*

(Note that the lock **pid_lock** is associated with a replacement for the traditional **proc** table, in which only a minimal per-process "handle" is statically allocated, each **struct proc** being allocated dynamically on demand. This scalability enhancement has nothing to do with the parallelization project, so will not be discussed further.)

The **proc_relation_lock** is more interesting. It protects the process pointers (parent, child, and sibling pointers) in *all* active processes. Though it may seem counterintuitive that a single, global lock should be needed for this purpose, we will present a case for it a bit later. First, in part to motivate that discussion, we will look at the per-**struct** changes we made to provide MP synchronization.

Changes to three structures were central to our parallelization effort. These are the process group (**pgrp**), process (**proc**), and session (**session**) structures. The changes to each are described in the sections that follow. Of the synchronization mechanisms listed, only the process group lock **pg_lock** exists in uniprocessor configurations.

5.1 Changes to process groups

As part of our work, each process group, or **pgrp**, now contains a reference count, **pg_refcnt**, to record references to the group other than from the processes in its own member list. Such references occur, for instance, from the TTY code, when a job-control signal must be sent to a group not currently in the foreground. The **pg_refcnt** member is protected by its own simple lock, **pg_refcnt_lock**, in order to ensure that a valid reference can be made to the group through one of its member processes (see Section 7.2.1).

A **pgrp** now also contains a read/write lock, **pg_lock**, taken for writing when the **pgrp** itself or its member list is modified, and for reading whenever the **pgrp** or its member list is otherwise accessed. This means, in particular, that the lock is held across calls to **pgsignal**. Since the new implementation of **psignal** can block, we initially wanted to avoid this prolonged lock retention. However, we found that strictly obeying a lock ordering scheme could eliminate the risk of deadlock in this situation (see Section 5.5).

Further, we found that our would-be replacement for the use of this lock was not robust. We had considered replacing it with a sequence of locking operations on the **proc**'s in the member list. However, since the member list is traversed using members of the **proc**'s themselves, and since (in

this scheme) a process could change groups at any time, use of the `proc`'s alone to control group traversal could result in partial traversal of n different groups by a single `pgsignal`. The idea of fixing this scheme by locking the original `pgrp` against process removal led us squarely back to the read/write lock that we have implemented.

5.2 Changes to processes

The "process", or `proc` struct posed different problems. Numerous parts of the kernel can refer to an arbitrary `proc`. In addition, the subsystem we concentrated on is filled with references to the current process. We felt we needed different, low-cost synchronization mechanisms for these two cases, for which a blocking lock was clearly inappropriate.

To ensure that a process remained valid throughout an arbitrary reference to it, we implemented a per-`proc` reference count, used whenever a process is successfully looked up by another process. A process can begin an *exit* (and terminate all threads other than the exiting one) while still referenced — it will not complete the *exit* until all references to it have been ended.

To safeguard modifications to the contents of a `proc` struct, we also added a per-`proc` simple lock, `p_lock`, which is taken only long enough to alter the members affected in a given operation.

5.3 Change to sessions

The only change we found necessary to the `session` struct was the addition of a simple lock, `s_lock`, to control access to its contents.

5.4 Global process relation lock

As noted above, the global `proc_relation_lock` protects the four pointers in each `proc` structure to other, related processes. These are are listed in Table 2:

Member Name	*Process Designated*
`p_pptr`	parent
`p_cptr`	youngest child
`p_ysptr`	next younger sibling
`p_osptr`	next older sibling

Table 2: *Per-*`proc` *process relation pointers*

We introduced this lock in order to avoid the formidable lock-ordering problems we foresaw in trying to use per-`proc` locks alone to control changes in process relations. As an example of these, consider the following points:

- When a process *fork*s, its `p_cptr` is set to point to the new child. If the child exits before the parent does, then the parent's `p_cptr` may need to be changed to point to an older sibling (if it still pointed to the exiting child).
- As a result of a *fork*, the `p_pptr` of the child process is set to point to the parent. If the parent exits before the child does, the child must be "reparented" – that is, its `p_pptr` must be changed to point to the *init* process.

- At several points in the kernel, a child process must be able to refer to its parent reliably (to send it a `SIGCHLD`, for instance, if the child stops due to a signal or because it is being traced). Similarly, a parent process must be able to refer reliably to its youngest child, in order to begin a traversal of the list of parent's children (which is maintained via the sibling pointers of each each child).
- We provide a reference count, of course, which is incremented to prevent a process from exiting while still in use by another process. However, in order to use the count in the contexts outlined here, we need to lock two processes at once, in an order determined by the operation being performed (such that we can't order the acquisition of the locks by any fixed scheme).

Thus in the absence of the `proc_relation_lock`, attempting to lock pairs of processes in the orders required could (on an MP system) easily result in deadlock. Placing all changes to these process pointers under the control of a single, global lock, however, permits us to access the pointers within related processes, and increment the reference counts of the target processes, without individual locks being taken beforehand.

5.5 Lock Ordering

More generally, in order to acquire multiple locks concurrently with no risk of deadlock, we defined a fixed order in which different kinds of locks had to be taken. A partial listing of this lock ordering (containing only the locks mentioned in this paper) follows, in Table 3. In general, blocking locks must be acquired before simple locks, and, subject to this constraint, global locks must be acquired before local locks.

Order	*Lock*	*Type*	*Scope*	*Comments*
1	`pgrphash_lock`	r/w	global	
2	`proc_relation_lock`	r/w	global	
3	`pg_lock`	r/w	per-`pgrp`	If two needed, order by address.
4	`uidhash_lock`	simple	global	
5	`pid_lock`	simple	global	
6	`p_lock`	simple	per-`proc`	If two needed, order by address.
7	`pg_ref_lock`	simple	per-`pgrp`	
8	`s_lock`	simple	per-`session`	

Table 3: *New lock Ordering*

6 Parallelized Process Management

Enabling the process management system calls to run on any processor in an MP system introduced two sorts of problems. The first concerned locking data written at high spl. The second was a variety of races that then became possible, both between related processes and between threads in a multi-threaded process.

6.1 Accessing data written at high spl

In an OSF/1.0 kernel, two pieces of process state can be written from the clock interrupt processing code, which runs with most (usually all) interrupts disabled. The first datum is a flag, used only

if user profiling is in effect, indicating that the profiling buffer should be updated on behalf of the thread interrupted by the clock tick. The second datum is the current CPU time limit for the current process. If this limit is exceeded, a `SIGXCPU` signal is sent to the process, and the limit is increased, to allow the process time to process the signal.

Both of these data must be protected from simultaneous writes by different processors. In the first case, we moved the "profiling buffer update" flag into its own word in a per-thread struct, where only the current thread would ever access it. In the second case, we used the existing per-process `u_time_lock`, which already had to be acquired with interrupts disabled, to control access to the CPU limit.

Once we introduce an operation on a lock in interrupt context, of course, we must ensure that the lock involved is always acquired at high spl. In thread context, this means disabling any interrupts during which the lock may be acquired. To see why, suppose a thread acquiring a lock that is also acquired in interrupt context fails to disable that interrupt. If the interrupt occurs and the interrupt handler tries to acquire the lock, the handler will spin waiting to obtain it. But the handler will spin forever in this case, since it interrupted the thread holding the lock (see [10] for a full discussion of this and other MP synchronization issues).

6.2 Resolving process management races

A number of races could be avoided simply by careful ordering of state changes. This was particularly relevant in contexts where such ordering had not mattered, when the code executed only on the master processor.

One such race involved the code implementing *wait*-style system calls (which enable a parent process to await or simply check for a state change in one of its children) and the code in *exit* that actually terminated a child. The child had to ensure that its termination was complete before awakening its parent, since otherwise the parent might run (and fail to see that the child had exited) before the child could complete its processing.

A more interesting race in `wait` would occur if multiple threads in a parent process each waited for the same child (or overlapping sets of children). To ensure that each parent thread would either wait for a child successfully or (correctly) return an error indication, we made the following changes in the *wait* code:

- Before traversing the child list of the calling process, take the `proc_relation_lock` for reading.
- If a child is found and is in the desired state, set a flag indicating that the current thread "owns" the child, unless of course the flag is already set.
- If a child is found, but is already owned by another thread, forget that we found it and continue scanning the child list.
- Once we have flagged the child as ours, release the `proc_relation_lock`.
- If we're removing the flagged child from its sibling list (i.e., if the child has terminated), re-acquire the `proc_relation_lock` for writing and update the list.

A further race condition existed between the *fork* system call and *exit* or *exec* when the two were used concurrently by different threads in the same process. The utility of such a combination is questionable, at best, but we did want to ensure that all kernel state would remain consistent if the combination occurred. This we did simply by adding a `p_forkcnt` member to the `proc` struct, to count *fork* operations currently in flight. Before a *fork* is allowed to proceed, the `sig_lock` is acquired, ensuring that no other thread in the process has begun an *exec* or an *exit* (see Section 4.1). Then `p_forkcnt` is incremented.

In *exec* or *exit*, on the other hand, a non-zero fork count causes the system call to block, waiting for the count to return to zero. At the end of a *fork*, the forking thread in the parent process decrements the count and awakens any blocked threads.

7 Parallelized Signal Handling

7.1 Reliable Signal Generation

One problem area in signal process is the signal generation code — **psignal**, which signals a single process, or **pgsignal**, which signals each process in a **pgrp**. In an ordinary uniprocessor kernel, this code does not acquire any locks, so does not block, and may be run at either interrupt or base level. On an MP, however, both **pgsignal** and **psignal** must acquire locks. Since the locks involved are widely used at base spl, they cannot be used at high spl as well.

Thus for configurations in which the signal generation code can block, we defined a deferred signalling mechanism, which we refined and extended from one used for an unrelated purpose in OSF/1.0. In this mechanism, a signal generation request enqueues an event containing the signal parameters, for later processing by a dedicated kernel thread. Naturally, the kernel thread runs at base spl, eliminating locking problems.

To simplify the correct use of **psignal** and **pgsignal**, we have defined the symbols as macros, which, in each kernel configuration, expand to whichever set of routines (direct or indirect) is universally safe in that configuration. Thus these macros may always be used. If a developer knows that a given call will always occur at base spl, he may explicitly call the direct version of the routines, rather than using the macros.

Function (or Macro) Name	*Usable From*	*Operation Type*	*Target*
psignal (macro)	any context	configuration-dependent	process
pgsignal (macro)	any context	configuration-dependent	process group
pgsignal_self (macro)	any context	configuration-dependent	process group + self
psignal_inthread	thread context	immediate	process
pgsignal_inthread	thread context	immediate	process group
pgsignal_inthread_self	thread context	immediate	process group + self
psignal_indirect	any context	deferred	process
pgsignal_indirect	any context	deferred	process group
pgsignal_internal	internal only	immediate	selected via flag

Table 4: *Signal generation macros and entry points*

Table 4 presents a summary of the signal generation macros, and of the underlying routines. Each macro expands to one of the corresponding routine names, according to kernel configuration. ("Thread context," of course, means base spl level. The symbols **pgsignal_self** and **pgsignal_inthread_self** are explained in Section 7.2.3.)

7.2 Process group signalling

Numerous problems arise from the fact that, in the parallelized process management code, one thread in the current process may change the **pgrp** of the process, while another thread is attempting to signal the **pgrp** (or otherwise traverse its member list).

7.2.1 Referring to a pgrp reliably

In order to signal a pgrp, the first problem a process must overcome is just to refer to the pgrp reliably. Generally, a pgrp is accessed via one of its member processes (i.e., using the p_pgrp member of a proc). To keep the process from changing groups while its p_pgrp pointer is read, the process is locked "around" the operation. As soon as the lock is released, the process can change groups. Further, if its former pgrp then becomes empty, the pgrp itself can be deleted. If the thread that read the pointer to the pgrp were interrupted for sufficiently long, both of these events might occur before it even got around to using the pgrp pointer it had acquired.

Clearly, to ensure that the pgrp remains valid, a reference should be taken on it while holding the lock of the process through which we accessed it. The reference count, however, must be incremented under lock. But taking a pgrp pg_lock while holding a proc lock would violate our locking hierarchy, introducing potential deadlocks. So we added the per-pgrp pg_ref_lock, at the bottom of the hierarchy, whose only role is to control access to the group reference count. This lock, which can be taken while the current proc is still locked, ensures that the reference count can be reliably incremented, and hence that the pgrp will remain valid after the process is unlocked.

This lock does not address the possibility that the process will change groups, once it is unlocked. We discuss that now.

7.2.2 Indeterminacy of pgrp signalled

One apparent problem that is actually not an issue in most respects is the fact that the current process may no longer belong to its original pgrp by the time the pgrp is signalled. Certainly it would be possible in some way to prevent the process from changing pgrp's while preparing to signal its (once-current) pgrp. (A process is already prevented from changing groups while group signalling is in progress — see Section 5.1.) However, the group change could always then occur as soon as the process was released — in particular, before it or any other group member had had time to act on the signal.

In general, therefore, it serves no particular purpose to delay a change of pgrp while deciding whether to generate a group signal.

7.2.3 Signalling the current process

Though, as just noted, the current pgrp of the current process is not germane when signalling the (potentially not-current) pgrp recorded for it at some previous point, we do often need to ensure that we signal the current process, whether or not we wind up signalling the pgrp to which it belongs at the time of the signalling operation. This need arises in signal generation performed by the TTY code, for instance. This poses a problem only in a multi-threaded process, in which one thread changes the process' pgrp while another thread is preparing to signal the previous pgrp.

To deal with this situation, we created the new macro pgsignal_self, and the new underlying routine pgsignal_inthread_self. These are like their ordinary pgsignal equivalents, except that they ensure the current process is always signalled, even if it is not a member of the pgrp indicated.

Since this behavior does differ from that of ordinary group signalling, of course, the entry point must be kept separate from pgsignal.

8 Observed Parallelization Gains

The parallelization changes we made to the OSF/1.1 kernel, in conjunction with other changes not described in this paper, have eliminated the use of unix_master in the course of normal kernel

usage in a BSD environment. There remain subsystems (primarily System V IPC and the System V filesystem) that are unparallelized, but, outside these, very few uses of **unix_master** remain anywhere in the kernel.

We believe that reducing the serialization induced by **unix_master** has yielded a significant increase in system-wide throughput, and has produced a measurable gain in single-threaded throughput for applications heavily dependent on the parallelized system calls. Note, in particular, that an interactive shell (whether or not a job-control shell) will issue several signal-related system calls each time it executes a command line. Now, these no longer force context switches in order to run on the master processor.

To quantify the gain in performance introduced by our changes, we will present comparisons of a kernel not containing our changes with the current OSF/1.1 kernel. These comparisons provide only a general impression of the performance enhancements involved, since the two kernels we used differ by more than just our changes. We have tried to minimize the impact of unrelated changes on the comparisons, but we will point out the results we consider especially affected by them.

The machine configuration used for these tests was an Encore Multimax containing 224 MB of memory and ten NS32332 processors, each rated at about 1.5 MIPS. All tests made use of a single SCSI disk attached via an EMC interface. The first three tests were run on an idle machine in single-user mode, while the pseudo-user session test was run on an idle machine that was up multi-user, with its network interface enabled. All times given below are elapsed (real) times, as measured either with the free-running microsecond counter provided by the Multimax (in the first two cases), or with **/bin/time** (in the last two cases).

8.1 Signal delivery throughput

We begin with a statistic very closely tied to our changes — namely the number of signals that can be delivered to a process per second. Figure 1 shows the delivery throughput in the two kernels, with from one to ten process pairs concurrently sending and receiving **SIGUSR1**.

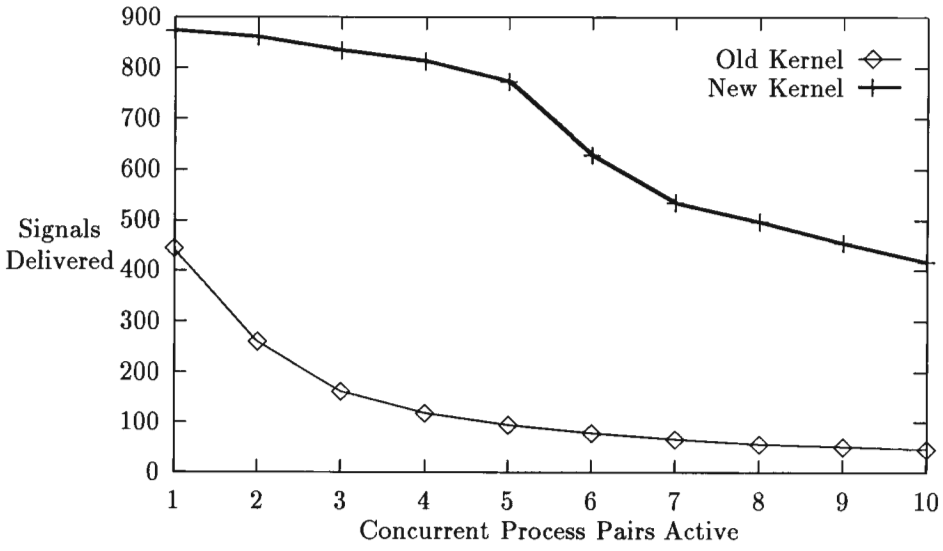

Figure 1: *Signal Deliveries per Process per Second*

In each pair of processes, a receiver process set up a handler for the signal, then looped over

calling *sigsuspend* in order to await signal receipt. The other process looped over sending the signal to the receiver using *kill*. When 1000 signals were received, the receiver exited, and the sender, when *kill* failed, did likewise. We measured the elapsed time in the receiver between the first call to *sigsuspend* and the final signal receipt.

In the old kernel, throughput per receiver process varies inversely to the number of active receivers, as would be expected. In the new kernel, the dropoff in performance with more than five sender/receiver pairs active is probably due to context switch overhead, since naturally the senders and receivers were running simultaneously and only ten processors were present.

8.2 Fork completion throughput

Another statistic of interest, somewhat more difficult to measure, is the number of forks that a process can complete per second. Figure 2 shows fork throughput in the two kernels, with from one to seven processes simultaneously calling *fork*.

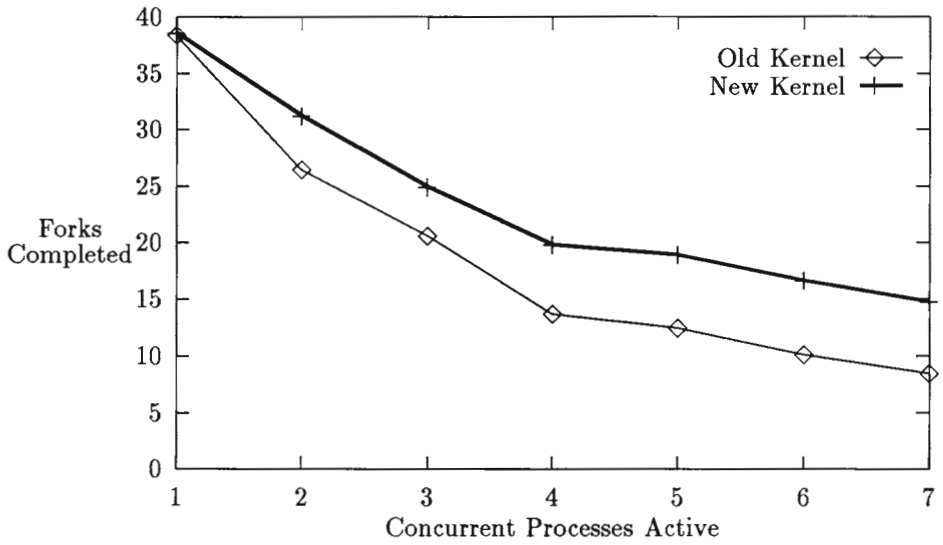

Figure 2: *Forks Completed per Process per Second*

Here, each process would loop over calling *fork* fifty times, as quickly as possible. Each child started would sleep for long enough to allow all forks to complete, then would exit. After completing all forks, each parent process would wait for its children to exit. We measured elapsed time in the parent between the first and the last fork.

Obviously, *fork* involves much more than just the process manipulation code that we parallelized — in particular, it also depends heavily on the Mach VM subsystem in creating child address spaces. Thus our changes, though still having a noticeable effect on throughput, did not produce the same magnitude of improvement as was possible in signal handling. These are the numbers that we consider the least useful for evaluating our changes, since the VM subsystems differ between the two kernels used, and since this test centers on an operation that is VM-intensive. At the very least, the single-process numbers do show that our MP locking does not degrade performance in this case.

8.3 Concurrent build throughput

To get some performance data more closely related to real-life system usage, we also measured the elapsed time needed to complete from one to eight instances of a medium-sized *make* operation, all executing simultaneously. Figure 3 plots the average elapsed time per build against the number of instances running.

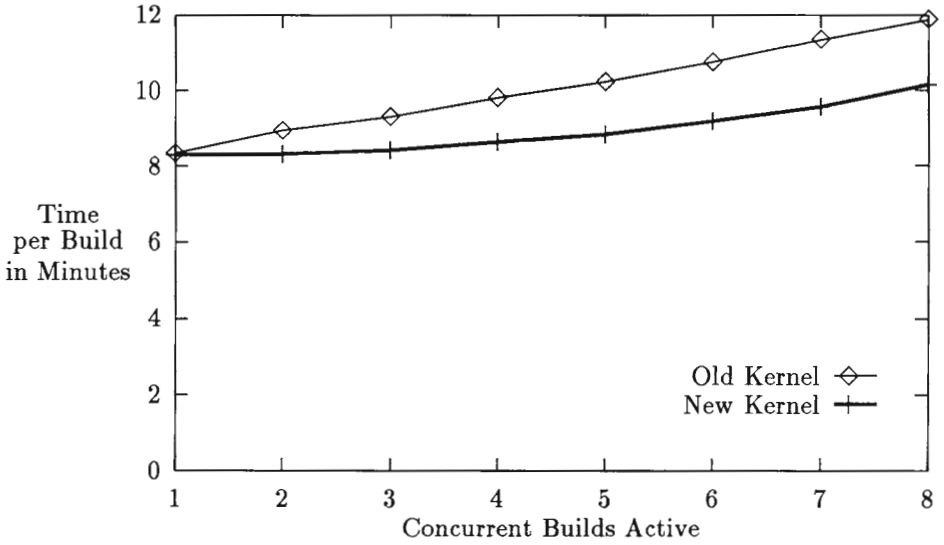

Figure 3: *Elapsed Time for Multiple Builds*

Each build was single-threaded, and compiled separate copies of the same source files, which contained roughly 10000 lines of C code. The builds were performed in sibling subdirectories of the same parent directory.

With the new kernel, it is likely that the sharper performance degradation seen to the right of the graph results more from loading on the single disk channel in use than from kernel data contention. Naturally, in this case even more than in our *fork* measurements, the improvement introduced by our changes is dominated by the time required for operations unrelated to process management.

8.4 Pseudo-user session throughput

The final data we will present here result from runs of a special script designed (very loosely) to emulate the behavior of a logged-in user, by exercising a broad subset of the OSF/1 utility set. The script is used by defining a number of pseudo-users, and giving them entries in `/etc/passwd` that specify the script as their "login shell." A remote front-end command then logs in to the test machine, specifying one of the pseudo-users as its login id. A run of this script subjects the system to significantly higher load than a real user would, since there are no delays between commands executed.

Figure 4 plots the elapsed time needed for the execution of one iteration of the pseudo-user script against the number of instances of the script being run simultaneously.

By the nature of the test, the numbers here are less well-ordered than any of the others we have presented. Though the test system was otherwise idle, it was up in multi-user mode and accessible to a local network, as we noted above. Also, in this particular case, we arranged for each concurrent

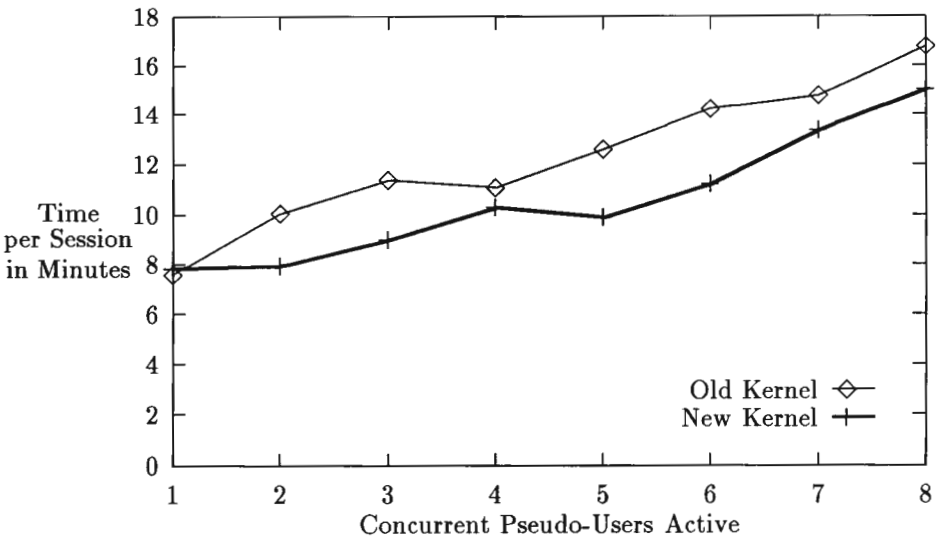

Figure 4: *Elapsed Time for Multiple Pseudo-User Sessions*

session to be started after a delay dependent on its order in the set of sessions currently being run. Thus different numbers of concurrent sessions could interact in unexpected ways, counteracting the uniform increase in per-session elapsed time one might otherwise expect to see. Really, the primary information to be gleaned from this graph is that the new kernel will, in general, execute this multi-user workload faster than the old kernel would.

9 Conclusions and Future Directions

We have successfully introduced significant performance gains into the OSF/1 kernel by parallelizing its process management and signal processing subsystems. As the previous section shows, the new kernel containing our changes in general degrades more gracefully under load than does the older kernel we compared it against. In the only test for which this was not true (the pseudo-user session test), though the shape of the performance curve is similar, our kernel's performance is still consistently better by a noticeable factor.

We believe that our implementation strikes a reasonable balance between efficiency and the complexity of the changes introduced. There is, of course, room for improvement in what we did. The most noticeable bottleneck that could be removed is the global `proc_relation_lock`, though experience with equivalent locks in other parallel kernels in the past has not shown them to be a major problem[3, 12].

We also have some data on contention for this lock in our kernel, under moderately heavy load (namely the eight-pseudo user session test running concurrently with the eight-process build test). When acquiring a lock, a thread can be obliged to wait in two different ways. First, when a lock cannot be immediately acquired, the locking code will busy-wait for a short time before blocking the calling thread. Then, if the lock remains inaccessible at the end of that time, the caller is blocked. We found that a thread had to wait at all in acquiring the `proc_relation_lock` well under two percent of the time. Of these waits, ten percent involved blocking the calling thread — meaning that 90 percent of the waits were of very short duration.

This contention rate, though above the median for locks used elsewhere in the kernel, does not make the `proc_relation_lock` one of the most heavily-contended. However, the lock might pose a scalability problem in much larger systems. It is possible that further per-`proc` locks could be used to replace the global lock, though more thought is still needed to be sure that these would be equivalent. One could imagine adding:

- a simple lock to control access to the process reference count alone, which (like the equivalent `pg_ref_lock`) would be at the bottom of the lock hierarchy, and hence could be taken without conflicting with the use of general per-process locks (`p_lock`).
- a lock, probably a simple lock, to control access to the child list of a process, that is, to the sibling pointers of the process' children.

With these in place, one could more easily synchronize changes to inter-process relationships "naturally" (i.e., by following the semantics of the code, rather than the constraints of a lock hierarchy). However, correctness concerns aside, we would be reluctant to add still more per-struct locks to our implementation without compelling evidence they were needed. Though performance measurement is still underway, we don't feel at the moment that we have such evidence.

Further analysis, particularly using lock statistics, may well also reveal other opportunities for optimization.

In conceptual terms, this project resolved some of the conflicts between traditional Unix abstractions (processes and signals in particular) and the multi-threaded execution model provided by Mach and hence by OSF/1, and began to explore the provision of a consistent and usable interface between the two. Much remains to be done, especially in promoting threads to be full-fledged participants in the Unix scheme. We hope we have laid some groundwork for future efforts, as the relevant standards mature sufficiently to become useful in guiding them.

10 Acknowledgements

We would like to thank the many persons at OSF and at Encore who supported us in the design and implementation of this project. David Black, of the OSF Research Institute, implemented the signal processing macros and other extensions for multi-threaded processes present in OSF/1.0, which served as a basis for our own changes to multi-threaded process manipulation. He also provided valuable advice and commentary throughout the project, as did several persons in the OSF engineering organization, including Jeff Carter, Jeff Collins, George Feinberg, Gary Fernandez, David Mitchell, and Tom Talpey. Finally, Alan Langerman, our project leader at Encore, provided helpful insights when we most needed them. His extensive SMP expertise significantly eased project development.

References

[1] M. Bach. *The Design of the Unix Operating System*, Englewood Cliffs, NJ, 1986. Prentice-Hall, Inc.

[2] J. Boykin and A. Langerman. The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis, in *Conference Proceedings, USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pp. 105-126, Ft. Lauderdate, FL, 1989. USENIX Association.

[3] Personal communication from Jeff Collins, who designed the parallelized process management subsystem in Encore UMAX 4.2.

[4] F. Douglis. Experience with Process Migration in Sprite, in *Conference Proceedings, USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pp. 59-72, Ft. Lauderdate, FL, 1989. USENIX Association.

[5] D. Freedman. Experience Building a Process Migration Subsystem for Unix, in *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pp. 349-356, Berkeley, CA, 1991. USENIX Association.

[6] G. Hamilton and D. Code. An Experimental Symmetric Multiprocessor Ultrix Kernel, in *Conference Proceedings, 1988 Winter USENIX Technical Conference*, Berkeley, CA, 1988. USENIX Association.

[7] S. Leffler, M. McKusick, M. Karels and J. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*, Reading, MA, 1989. Addison-Wesley Publishing Company.

[8] J. Lions. *A Commentary on the Unix Operating System*, Sydney, Australia, 1977. Univ. of New South Wales.

[9] S. LoVerso, N. Paciorek, A. Langerman and G. Feinberg. The OSF/1 Unix Filesystem (UFS), in *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pp. 207-218, Berkeley, CA, 1991. USENIX Association.

[10] N. Paciorek, S. LoVerso and A. Langerman. Debugging Multiprocessor Operating System Kernels, in *Symposium Proceedings, Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 185-202, Atlanta, GA, 1991. USENIX Association.

[11] U. Sinkewicz. A Strategy for SMP ULTRIX, in *Conference Proceedings, 1988 Summer USENIX Technical Conference*, pp. 203-212, El Toro, CA, 1988. USENIX Association.

[12] *UMAX 4.2 Programmer's Reference Manual*, Marlborough, MA, 1986. Encore Computer Corporation.