Digital Technical Journal
Volume 6, Number 3

VTX Version of the OSF Paper

DEC OSF/1 Symmetric Multiprocessing

by

Jeffrey M. Denham, Paula Long, and James A. Woodward

ABSTRACT

The primary goal for an operating system in a symmetric multiprocessing (SMP) implementation is to convert the additional computing power provided to the system, as processors are added, into improved system performance without compromising system quality. The DEC OSF/1 version 3.0 operating system uses a number of techniques to achieve this goal. The techniques include algorithmic enhancements to improve parallelism within the kernel and additional lock-based synchronization to protect global system state. Synchronization primitives include spin locks and blocking locks. An optional locking hierarchy was imposed to detect latent symmetric multiprocessor synchronization issues. Enhancements to the kernel scheduler improve cache usage by enabling soft affinity of threads to the processor on which the thread last ran; a load-balancing algorithm keeps the number of runnable threads spread evenly across the available processors. A highly scalable and stable SMP implementation resulted from the project.

INTRODUCTION

The DEC OSF/1 operating system is a Digital product based in part on the Open Software Foundation's OSF/1 operating system.[1] One major goal of the DEC OSF/1 version 3.0 project was to provide a leadership multiprocessing implementation of the UNIX operating system for Alpha server systems, such as the Digital AlphaServer 2100 product. This paper describes the goals and development of this operating system feature for the version 3.0 release.

THE DEC OSF/1 VERSION 3.0 MULTIPROCESSING PROJECT

Multiprocessing platforms like the AlphaServer 2100 product provide a cost-effective means of increasing the computing power of a server. Additional computing capacity can be obtained at a potentially significant cost advantage by simply adding CPU modules to the system rather than by adding a new system to a more loosely coupled network-server arrangement. An effective execution of this server-scaling strategy requires significant cooperation between the hardware and software components of the system. The hardware must provide symmetrical (i.e., equal) access to system resources, such as memory and I/O, for all processors; the operating system software must provide for enough parallelism in its major subsystems to allow applications to take advantage of the additional CPUs in the system. That is, the operating system cost of multiprocessing must be kept low enough to enable most of an additional CPU's computing power to be used by applications rather than by the operating system's efforts to synchronize simultaneous access to shared memory by multiple processors.

Regarding hardware, the AlphaServer 2100 product and the other Alpha multiprocessing platforms provide the shared memory and symmetric access to the system and I/O buses desired by the operating system designers.[2] The design allows all CPUs to share a single copy of the operating system in memory. The hardware also has a load-locked/store-conditional instruction sequence, which provides both a mechanism for atomic updates to shared memory by a single processor and an interprocessor interrupt mechanism.

Given these hardware features, operating system software developers have a great deal of freedom in developing a multiprocessing strategy. The approach used in DEC OSF/1 version 3.0 is called symmetric multiprocessing (SMP), in which all processors can participate fully in the execution of operating system code. This symmetric design contrasts with asymmetric multiprocessing (ASMP), in which all operating system code must be executed on a single designated "master" processor. Such an approach is undesirable because it provides inadequate utilization of additional "slave" processors for most application mixes. By contrast, for the DEC OSF/1 multiprocessing design, the concept of a master processor applies only to the keeping of the global system time and to other specialized uses (such as supporting subsystems that are not yet fully symmetric).

The SMP features in the DEC OSF/1 version 3.0 operating system are based on the joint work of Carnegie Mellon University, for the Mach version 2.5 kernel, and the Open Software Foundation and the Encore Computer Corporation, for the version 1.2 release of the OSF/1 operating system.[3-6] From this substantial technical base, the DEC OSF/1 multiprocessing project focused on achieving UNIX leadership performance on targeted commercial server applications, such as data servers (i.e., DBMS and file servers) and compute servers. These application domains tend to make heavy use of system services. Therefore, shortcomings in the multiprocessing implementation become readily apparent through the failure of these applications to gain significant performance speedups as processors are added to the server. The ideal benefit is, of course, to obtain 100 percent of each additional processor for the applications' use. In reality, a gain of 70 to 80 percent of the last CPU added is well worth the incremental cost of the processor.

From the outset of the project, the engineering team was empowered to enhance and augment the OSF/1 version 1.2 code base to obtain this level of multiprocessing performance for DEC OSF/1 version 3.0. At the same time, it was required to maintain the system's stability and reliability. The team was staffed by engineers with extensive multiprocessing and real-time operating system experience inside and outside Digital. Quality assurance (Q/A) and performance teams provided considerable feedback as the product moved through its development base levels.

The engineering team faced multiple technical issues in the SMP implementation of the DEC OSF/1 operating system, including

- Analyzing concurrency and locking issues
- Adapting the base operating system for SMP
- Supporting a comprehensive lock package
- Adapting thread scheduling for SMP
- Ensuring a quality implementation
- Benchmarking progress in SMP performance

The remainder of this paper describes the highlights of the team's efforts in these areas.

ANALYZING CONCURRENCY AND LOCKING ISSUES

Moving from a uniprocessor to a shared-memory, symmetric multiprocessing platform places new demands on an operating system. Multiple processes running independently on separate processors can access kernel data structures simultaneously. This level of true concurrency is unobtainable on uniprocessor systems, where concurrency either derives from the asynchronous execution of interrupt service routines (ISRs) or is emulated through the interleaving of processes on a time-share basis. In the first case, synchronization is required for data structures accessed by both mainline kernel code and the ISR. The technique used to achieve synchronization is to raise the processor interrupt priority level (IPL), i.e., system priority level (SPL) in UNIX parlance, in the mainline code to the level used by the competing ISR, thus blocking the interrupt that invokes the ISR. In the case of the virtual concurrency provided by process time-sharing, synchronization is achieved by allowing only one process to be in kernel context at a time. The kernel protects itself by preventing context switching (process preemption) until an executing process has reached a safe point, i.e., usually when it is about to leave kernel context. Other safe points appear when a process must voluntarily block to await the availability of some resource. These are the synchronization strategies employed by traditional UNIX-based operating systems.

One powerful feature of the OSF/1 kernel provides a further level of concurrency, which complicates the process of synchronizing access to kernel data; that feature is kernel-based threads. The Mach task/thread model allows multiple threads of execution to be active within a single task (process) address space. Therefore, whereas an unthreaded UNIX system has to protect data shared by multiple processes, e.g., the scheduling queues, a threaded kernel must protect all process-level data, which is shared by all threads in the process.

Although in many ways a traditional UNIX system from the user's point of view, the first version of the DEC OSF/1 operating system departed from typical UNIX practice by providing kernel-mode preemption in its real-time version of the kernel. This enhancement, targeted to improve the responsiveness of the system to real-time events, allows preemptive priority-based context switching between threads to occur at any point in kernel execution that meets a set of criteria for preemption safety. These criteria have an immediate relevance and applicability to the work of adapting the OSF/1 uniprocessor code to a multiprocessing environment. In the following discussion of preemption safety, each criterion for safe preemption is presented as it relates and leads to an understanding of correct multiprocessing synchronization.

Real-time thread preemption can occur only when all three of the following conditions are met:

1. The processor SPL is zero. This state indicates that all interrupts are enabled and implies that no code is executing in an ISR or is modifying kernel data shared with an ISR.

   On a nonpreempting uniprocessor kernel, SPL synchronization alone is adequate to protect shared data structures. SPL is a processorwide rather than a systemwide characteristic. Consequently, raising the SPL to interrupt level is inadequate protection on a multiprocessing system, in which one processor's SPL has no effect on another's. The classic multiprocessing solution to this problem is to combine SPL synchronization with mutual-exclusion spin locks to block out other processors as well as ISRs.

2. No simple locks (spin locks) are held. This state is represented in the Mach and OSF/1 kernel code by a call to the simple_lock() routine. This call signifies that the code has entered a critical section where shared data will be modified. On a uniprocessor, calling the simple_lock() routine actually increments a global spin lock count; unlocking decrements that count. If the count is zero, then an attempt to preempt the current process can be made. In this uniprocessor implementation, no actual spin locks exist in memory, and nothing is locked in the physical sense of a lock bit being checked for a

state of zero or one.

By contrast, on a microprocessing system, real locking, not lock counting, is required; therefore, spin locks occupy real memory. On a multiprocessing system, locking a spin lock involves testing the lock location for a value of zero and then atomically setting the value to one before continuing into the critical code section, assured of exclusive access. If another processor finds the lock bit set (i.e., nonzero), it will repeatedly test the lock location and thus "spin" until the lock value becomes zero when unlocked by its previous holder. Because processors make no progress while they attempt to obtain a spin lock, such a lock is meant to be held for bounded, hopefully brief periods. Extensive or unbounded accesses require the use of complex locks (blocking read/write locks) by which a thread will sleep until a locked resource becomes available and unlocked. (Sleeping to obtain a complex lock is by definition a preemption point.)

3. The code is not funneled to the master processor. This state is another way by which OSF/1 kernel code delineates a critical code section. Funneling forces code to run on a single processor designated as the master processor. Funneling allows device drivers and entire kernel subsystems that have not been adapted to a concurrent-execution environment with simple_lock() calls to modify kernel data safely. On a preempting uniprocessor, funneling is represented simply as a per-thread flag that prevents preemption when set; no context switching is required to cause funneling.

   By contrast, on a multiprocessing system, funneling to the master processor may involve an actual context switch from the funneling thread's current processor---an expensive form of synchronization. Prior to DEC OSF/1 version 3.0, all UNIX process subsystem components, including the fork(), exec(), wait(), and exit() routines and signal logic, were not safe for preemption and were therefore funneled. All modifications to process data structures could occur only on the master processor. This situation eliminated concerns about access to those structures from another processor but at the same time virtually eliminated the parallelism of the process subsystem. For example, for the fork system calls, the list of active processes in the system (allproc) was traversed in funneled code. Clearly, funneling this fundamental resource introduces significant latencies into the system's response to scheduling events. In multiprocessing terms, no process-level operations can execute in parallel.

The development of the DEC OSF/1 real-time kernel leveraged the existing OSF/1 SPL, locking, and funneling constructs to implement preemption on uniprocessor Alpha systems. This work provided a valuable product feature and was a preview of the effort that would be required to adapt the OSF/1 code for the DEC 2000, 4000, and 7000 multiprocessing platforms. Supporting separate preemptive kernels for three versions prior to DEC OSF/1 version 3.0, combined with the developers' experience on other multiprocessing systems (including ULTRIX version 4 and an advanced development project using MIPS multiprocessing platforms), uncovered the following challenges and problems that the team had to overcome to produce a competitive multiprocessing product:

- Supporting two complete sets of kernel binary objects---base and real-time---was burdensome for the operating system engineers and awkward for third-party developers. Therefore, the DEC OSF/1 multiprocessing product team had to strive to ship a single, unified set of kernel binaries. This set should encompass the full range of real-time features, including preemption and POSIX fixed-priority scheduling. For that to be practical, the resulting multiprocessing kernel would have to perform as well on a uniprocessor as the non-SMP kernel.

- Diagnosing locking problems in the preemptive kernel was expensive in developer time. The process required painstaking inspection of the simple-locking source code, which is often disguised in subsystem-specific macros. Locking or unlocking a spin lock multiple times or not unlocking it at all (usually in code loops) would disable preemption well beyond the end of a critical section or enable it before the end. A coherent locking architecture with automated debugging facilities was needed to ship a reliable product on time. The lock-debugging facility present in the original OSF/1 code was probably inadequate for the task.

- Experiments with the real-time kernel revealed unacceptable preemption latencies, especially in funneled code paths. This deficiency indicated that, when moved to a multiprocessing platform, the existing kernel would fail to use additional processors effectively. That is, the kernel would not exhibit adequate parallelism to scale effectively. Clearly, major work was required to significantly increase parallelism in the kernel. This task would likely involve removing most uses of funneling, eliminating some spin locks, and adding other spin locks to create a finer granularity of locking.

ADAPTING THE BASE OPERATING SYSTEM FOR SYMMETRIC MULTIPROCESSING

Making the leap from a preemptive uniprocessor kernel to an effective SMP implementation, built from a single set of kernel binaries, required contributions from the OSF/1 version 1.2 and the DEC OSF/1 version 3.0 projects. Fundamental changes were introduced into the system to support SMP.

The basic approach planned by the SMP project team was first to bootstrap the DEC OSF/1 version 1.3 kernel on the existing Alpha multiprocessing platforms. This task was accomplished by funneling all major subsystems to a single processor while stabilizing the underpinnings of the multiprocessing system (i.e., the scheduler, the virtual memory subsystem, the virtual file system, and the hardware support) in the new environment. This approach allowed the team to make progress in understanding the scope of the effort while analyzing the multiprocessing requirements of each kernel subsystem. The in-depth analysis was necessary to identify those subsystems in the kernel that required modifications to run safely and efficiently under SMP. As each subsystem was confirmed to exhibit parallelism or was made parallel, it was unfunneled and thus freed to run on any processor. This process was iterative. If incorrectly parallelized, a subsystem will reveal itself by (1) leaving data incorrectly unprotected and thus open for corruption and (2) developing a deadlock, i.e., a situation in which each of two threads holds a spin lock required by the other thread and thus neither thread can take the lock and proceed.

The efforts at parallelizing the kernel fell into two classes of modification: lock-based synchronization to ensure multiprocessing correctness and algorithmic changes to increase the level of parallelism achieved.

Lock-based Synchronization

The code base on which the DEC OSF/1 product is built, i.e., the Open Software Foundation's OSF/1 software, provides a strong foundation for SMP. The OSF further strengthened this foundation in OSF/1 versions 1.1 and 1.2, when it corrected multiple SMP problems in the code base and parallelized (and thus unfunneled) additional subsystems. As the multiprocessing bootstrap effort continued, the team analyzed and incorporated the OSF/1 version 1.2 SMP improvements into DEC OSF/1 version 3.0. As strong as this starting point was, however, some structures in the system did not receive the appropriate level of synchronization. The team corrected these problems as they were uncovered through testing and code inspection.

The DEC OSF/1 operating system uses a combination of simple locks, complex locks, elevated SPL, and funneling to guarantee synchronized access to system resources and data structures. Simple locks, SPL, and funneling were described briefly in the earlier discussion of preemption. Complex locks, like elevated SPL, are used in both uniprocessor and multiprocessor environments. These locks are usually sleep locks---threads can block while they wait for the lock---which offer additional features, including multiple-reader/single-writer access and recursive acquisition.

An example of the use of each synchronization technique follows:

- A simple lock is used to protect the kernel's callout (timer) queue. In an SMP environment, multiple threads can update the callout queue at the same time, as each of them adds a timer entry to the queue. Each thread must obtain the callout lock before adding an entry and release the lock when done. The callout simple lock is also a good example of SPL synchronization under multiprocessing because the callout queue is scanned by the system clock ISR. Therefore, before locking the callout lock, a thread must raise the SPL to the clock's IPL. Otherwise, the thread holding the callout lock at an SPL of zero can be interrupted by the clock ISR, which will in turn attempt to take the callout lock. The result is a permanent deadlock.

- A complex lock protects the file system directory structure. A blocking lock is required because the directory lock holder must perform I/O to update the directory, which itself can block. Whenever blocking can occur while a lock is held, a complex lock is required.

- Funneling is used to synchronize access to the ISO 9660 CD-ROM file system.[7] The decision to funnel this file system was largely due to limitations in the DEC OSF/1 version 3.0 schedule; however, the file system is a good choice for funneling because of its generally slow operation and light usage.

To ensure adequate performance and scaling as processors are added to the system, an SMP implementation must provide for as much parallelism through the kernel as possible. The granularity of locks placed in the system has a major impact on the amount of parallelism obtained.

During multiprocessing development, locking strategies were designed to

- Reduce the total number of locks per subsystem

- Reduce the number of locks taken per subsystem operation

- Improve the level of parallelism throughout the kernel

At times, these goals clashed: enhancing parallelism usually involves adding a lock to some structure or code path. This outcome conflicts with the goal of reducing lock counts. Consequently, in practice, the process of successfully parallelizing a subsystem involves striking a balance between lock reduction and the resulting increase in lock granularity. Often, benchmarking different approaches is required to fine-tune this balance.

Several general trends were uncovered during lock analysis and tuning. In some cases locks were removed because they were not needed; they were the products of overzealous synchronization. For example, a structure that is private to a thread may require no locking at all. Moreover, a data element that is read atomically needs no locking. An example of lock removal is the gettimeofday() system call, which is used frequently by DBMS servers. The system call simply reads the system time, a 64-bit quantity, and copies it to a buffer provided by the caller. The original OSF/1 system call, running on a 32-bit architecture, had to take a simple lock before reading the time to guarantee a consistent value. On the Alpha architecture, the system call can read the entire 64-bit time value atomically. Removing the lock resulted in a 40 percent speedup.

In other cases, analyzing how structures are used revealed that no locking was needed. For example, an I/O control block called the buf structure was being locked in several device drivers while the block was in a state that allowed only the device driver to access it. Removing these unnecessary locks saved one complex and one simple locking sequence per I/O operation in these drivers.

Another effective optimization involved postponing locking until a thread determined that it had actual work to do. This technique was used successfully in a routine frequently called in a transaction processing benchmark. The routine, which was locking structures in anticipation of following a rarely used code path, was modified to lock only when the uncommon code path was needed. This optimization significantly reduced lock overhead.

To improve parallelism across the system, the DEC OSF/1 SMP development team modified the lock strategies in numerous other cases.

Algorithm Changes

In some instances, the effective migration of a subsystem to the multiprocessing environment required significant reworking of its fundamental algorithms. This section presents three examples of this work. The first example involves the rework of the process management subsystem; the second example is a new technique for a thread to refer to its own state; and the third example deals with enhancements in translation buffer coherency or "shootdown."

Managing Processes and Process State. Early versions of the DEC OSF/1 software maintained a set of systemwide process lists, most notably proc (static proc structure array), allproc (active process list), and zomproc (zombie process list). These lists tend to be fairly long and are normally traversed sequentially. Operations involving access to these lists include process-creation time (fork()), signal posting, and process termination. The original OSF/1 code protected these process lists and the individual proc structures themselves by means of funneling. This meant that virtually every system call that involved process state, such as exit(), wait(), ptrace(), and sigaction(), was also forced into a single funnel. Experience with real-time preemption indicated that this approach would exact excessive multiprocessing costs. Although it is possible to protect these lists with locks, the development team decided that this basic portion of the kernel must be optimized for maximum multiprocessing performance. The OSF also recognized the need for optimization; they addressed the problem in OSF/1 version 1.2 by adopting a redesign of the process management developed for their Multimax systems by Encore Computer Corporation. The DEC OSF/1 team adopted and enhanced this design for handling process lists, process management system calls, and signal processing.

The redesign replaces the statically sized array of proc structures with an array of smaller process identification (PID) entry structures. Each PID entry structure potentially points to a dynamically allocated proc structure. Under this new scheme, finding the proc structure associated with a user PID has been reduced to hashing the PID value to an index into the PID entry array. The process state associated with that PID (active, zombie, or nonexistent) is maintained in the PID entry structure. This allows process structures to be allocated dynamically, as needed, rather than statically at boot time, as before. Simple locks are also added to the process structure to allow multiple threads in the process to perform process management system calls and signal handling concurrently. These changes allowed process management funneling to be removed entirely, which significantly improved the degree of parallelism in the process management subsystem.

Accessing Current Thread State. One critical design choice in implementing SMP on the DEC OSF/1 system concerned how to access the state of the currently running thread. This state includes the current thread's process, task, and virtual memory structures, and the so-called uarea, which contains the pageable UNIX state. Access to this state, which threads require frequently as they run in kernel context, must have low overhead. Further, because the DEC OSF/1 operating system supports kernel-mode preemption, the method for accessing the current thread's state must work even if a context switch to another CPU occurs during the access operation.

The original OSF/1 code used arrays indexed by the CPU number to look up the state of a running thread. One of these arrays was the U_ADDRESS array, which was used to access the currently active uarea. The U_ADDRESS array was loaded at context switch time and accessed while the thread executed. Before the advent of multiprocessing, the CPU number was a compile-time constant, so that thread-state lookup involved simply reading a global variable to form the pointer to the data. Adding multiprocessing support meant changing the CPU number from a constant to the result of the WHAMI ("Who am I?") PALcode call to get the current CPU number. (PALcode is the operating-system-specific privileged architecture library that provides control over interrupts, exceptions, context switching, etc.[8])

Using such global arrays for accessing the current thread's state presented three shortcomings:

1. The WHAMI PALcode call added a minimum overhead of 21 machine cycles on the AlphaServer 2100 server, not including further overhead due to cache misses or instruction stream stalls. The multiprocessing team felt that this was too large a performance price to pay.

2. Allowing multiple CPUs to write sequential pointers caused cache thrashing and extra overhead during context switching.

3. Indexing by CPU number was not a safe practice when kernel-mode preemption is enabled. A thread could switch processors in the middle of an array access, and the wrong pointer would be fetched. Providing additional locking to prevent this had unacceptable performance implications because the operation is so common.

These problems convinced the team that a new algorithm was required for accessing the current thread's state.

The solution selected was modeled on the way the OpenVMS VAX system uses the processor interrupt stack pointer to derive the pointer to per-CPU state.[9] In the OSF/1 system, each thread has its own kernel stack. By aligning this stack on a power-of-two boundary, a simple masking of the stack pointer yields a pointer to the per-thread data, such as the process control block (PCB) and uthread structure. Any data item in the per-thread area can be accessed with the following code sequence:

```
lda r16, MASK         # Get mask value
bic sp, r16, r0       # Mask stack pointer to point to stack base
ldq rx, OFFSET(r0)    # Add offset to base and fetch item
```

Accessing thread state using the kernel stack pointer solves all three problems with CPU-number-based indexing. First, this technique has very low overhead; accessing the current thread's data involves only a simple masking operation and a read operation. Second, using the kernel stack pointer incurs no extra overhead during context switching because the pointer has to be loaded for other uses. Third, because thread stack areas are pages, no cache conflicts exist between threads running on different processors. Finally, the data access can be preempted at any point, and the correct pointer is still fetched. No processor-dependent state is used to access the current thread state.

Interprocessor Translation Lookaside Buffer Shootdown. Alpha processors employ translation lookaside buffers (TLBs) to speed up the translation of physical-to-virtual mappings. The TLB caches page table entries (PTEs) that contain virtual-to-physical address mappings and access control information. Unlike data cache coherency, which the hardware maintains, TLB cache coherency is a task of the software. The DEC OSF/1 system uses an enhanced version of the TLB shootdown algorithm developed for the Mach kernel to maintain TLB coherency.[10] First, a modification to the original shootdown algorithm was needed to implement the Alpha architecture's address space numbers (ASNs). Second, a synchronization feature of the original algorithm was removed entirely to enhance shootdown performance. This feature provided synchronization for architectures in which the hardware can modify PTEs, such as the VAX platform; the added protection is unnecessary for the Alpha architecture.

The final shootdown algorithm is as follows. The physical map (PMAP) is the software structure that holds the virtual-to-physical mapping information. Each task within the system has a PMAP; operating system mappings have a special kernel PMAP. Each PMAP contains a list of processors currently using the associated address space. To initiate a virtual-to-physical translation change, a processor (the initiator) first locks the PMAP to prevent any other threads from modifying it. Next, the initiator updates the PTE mapping in memory and flushes the local TLB. The processor then sends an interprocessor interrupt to all other processors (the responders) that are currently active in the same address space. Each responder needs to acknowledge the initiator and invalidate its own mapping. Once all responders are accounted for, the initiator is free to continue with the knowledge that all TLBs are coherent on the system. The initiator marks nonactive processors' ASNs inactive, spins while it waits for other processors to check in, and then unlocks the PMAP. Figure 1 shows this final TLB shootdown algorithm as it progresses from the initiating processor to the potential responding processors.

Figure 1 Translation Lookaside Buffer Shootdown Algorithm

```
Initiator:                                    Responders:

Lock the PMAP.
Update the translation map (PTE).
Invalidate the processor TLB entry.
Send an interprocessor interrupt to all
   processors that are using the PMAP.
                                              Acknowledge the shootdown.
                                              Invalidate the processor TLB
entry.
                                              Return from the interrupt.
Mark the nonactive processors' ASNs
   inactive.
Spin while it waits for other
   processors to check in.
Unlock the PMAP.
```

DEVELOPING THE LOCK PACKAGE

Key to meeting the performance and reliability goals for the multiprocessing portion of the DEC OSF/1 version 3.0 release was the development of a lock package with the following characteristics:

- Low execution and memory overhead
- Flexible support for both uniprocessor and multiprocessor platforms, with and without real-time preemption
- Automated debugging facilities to detect incorrect locking practices at run time
- Statistical facilities to track the number of locks used, how many times a lock is taken, and how long threads wait to obtain locks

Of course, the overall role of the lock package is to provide a set of synchronization primitives, that is, the simple and complex locks described in earlier sections. To support kernel-mode thread preemption, DEC OSF/1 version 1.0 had extended the lock package originally delivered with OSF/1 version 1.0. Early in the DEC OSF/1 version 3.0 project, the development team extended the package again to optimize its performance and to add the desired debugging and statistical features.

As previously noted, a major goal for DEC OSF/1 version 3.0 was to ship a single version of its kernel objects, instead of the base and real-time sets of previous releases. Therefore, simple locks would have to be compiled into the kernel, even for kernels that would run only on uniprocessor systems. Achieving this goal required minimizing the size of the lock structure; it would be unacceptable to have hundreds of kilobytes (KB) of memory dedicated to lock structures in systems that did not use such structures. Further, the simple lock and unlock invocations required by the multiprocessing code would have to be present for all platforms, which would raise serious performance issues for uniprocessor systems. In fact, in the original OSF/1 lock package, the CPU overhead cost of compiling in the lock code was between 1 and 20 percent. Compute-intensive benchmarks showed the cost to be less than 5 percent, but the cost for multiuser benchmarks was greater than 10 percent, which represents an unacceptable performance degradation. To meet the goal of a single set of binaries, the development team had to enhance the lock package to be configurable at boot time. That is, the package needed to be able to tailor itself to fit the configuration and real-time requirements of the platform on which it would run.

The lock package supplied by the OSF/1 system was further deficient in that it did not support error checking when locks were asserted. This deficiency left developers open to the most common tormentor of concurrent programmers, i.e., deadlocks. Without error checking, potential system hangs caused by locks being asserted in the wrong order could go undetected for years and be difficult to debug. A formal locking order or hierarchy for all locks in the system had to be established, and the lock package needed the ability to check the hierarchy on each lock taken.

These needs were met by introducing the notion of lock mode to the lock package. Developers defined the following five modes and associated roles:

- Mode 0: No lock operations; for production uniprocessor systems

- Mode 1: Lock counting only to manage kernel preemption; for production real-time uniprocessor systems

- Mode 2: Locking without kernel preemption; for production multiprocessing systems

- Mode 3: Locking with kernel preemption; for production real-time multiprocessing systems

- Mode 4: Full lock debugging with or without preemption; for any development system

The default uniprocessor lock mode is 0; the multiprocessing default is lock mode 2. Both selections favor non-real-time production systems. The system's lock mode, however, can be selected at boot time by a number of mechanisms. Lock modes are implemented through a dynamic lock configuration scheme that essentially installs the appropriate set of lock primitives for the selected lock mode. Installation is realized by patching the compiled-in function calls, such as simple_lock(), to dispatch to the corresponding lock primitive for the selected lock mode. This technique avoids the overhead of dispatching indirectly to different sets of lock primitives for each call, based on the lock mode. The compiled-in lock function calls to the lock package are all entry points that branch to a call-patching routine called simple_lock_patch(). This routine changes the calling machine instruction to be patched out (for lock mode 0) or to branch to the corresponding primitive in the appropriate set of actual primitives, and then branches there (for lock modes 1 through 4). Thus, the overhead for dynamically switching between the versions of simple lock primitives occurs only once for each code path. In the case of lock mode 0, calls to simple lock primitives are "back patched" out. Under this model, uniprocessor systems pay a one-time cost to invoke the simple lock primitives, after which the expense of executing a lock primitive is reduced to executing a few no-op instructions where the code for the lock call once resided.

To address memory consumption issues and to provide better system debug capabilities, the developers reorganized the lock data structures around the concept of the lockinfo structure. This structure is an encapsulation of the lock's ordering (hierarchical relationship) with surrounding locks and its minimum SPL requirement. Lock debugging information and the lock statistics were decoupled from the lock structures themselves. To facilitate the expression of a lock hierarchy, the developers introduced the concept of classes and instances. A lock class is a grouping of locks of the same type. For example, the process structure lock constitutes a lock class. A lock instance is a particular lock of a given class. For example, one process structure simple lock is an instance of the class process structure lock. Error checking and statistics-gathering are performed on a lock-class basis and only in lock mode 4.

Decoupling the lock debugging information from the lock itself significantly reduced the sizes of the simple and complex lock structures to 8 and 32 bytes, respectively. Embedded in both structures is a 16-bit index into the lockinfo structure table for that particular lock class. The lockinfo structure is dynamically created at system startup in lock mode 4. All classes in the system are assigned a relative position in a single unified lock hierarchy. A lock class's position in the lockinfo table is also its position in the lock hierarchy; that is, locks must be taken in the order in which they appear in the table. Lock statistics are also maintained on a per-class basis with separate entries for each processor. Keeping lock statistics per processor and separating this information by cache blocks eliminates the need to synchronize lock-primitive access to the statistics. This design, which is illustrated in Figure 2, prevents negative cache effects that could result from sharing this data.

Figure 2   Lock Structure

```
LOCK INSTANCES     LOCK CLASS          LOCK STATISTICS
+------------+     +--------+            +---------+
|            |--+  |        |            |         |
+------------+  |  |        |            |         |
                |  |        |            +---------+
+------------+  |  |        |   +------->|  CPU 0  |
|            |--+  +--------+   |        +---------+
+------------+  |->|        |---+        |         |
                |  +--------+   |        |         |
+------------+  |  |        |   |        +---------+
|            |--+  |        |   +------->|  CPU 1  |
+------------+     |        |   |        +---------+
                   |        |   |        |         |
                   |        |   |        |         |
                   |        |   |        +---------+
                   |        |   +------->|  CPU N  |
                   |        |            +---------+
                   |        |            |         |
                   |        |            |         |
                   +--------+            +---------+
```

Once this powerful lock package was operational, developers analyzed the lock design of their kernel subsystems and attempted to place the locks used into classes in the overall system lock hierarchy. The position of a class depends on the order in which its locks are taken and released in relation to other locks in the same code path and in the system. At times, this static lock analysis revealed problems in existing lock protocols, in which locks were taken in varying orders at different points in the code. Clearly, the lock protocol needed to be reworked to produce a consistent order that could be codified in the hierarchy. Thus, the exercise of producing an overall lock hierarchy resulted in a significant cleanup of the original multiprocessing code base. To add a new lock to the system, a developer would have to determine the hierarchical position for the new lock class and the minimum SPL at which the lock must be taken.

Running the system in lock mode 4 and exercising code paths of interest provided developers with immediate feedback on their lock protocols. Using the hierarchy and SPL information stored in the run-time lockinfo table, the lock primitives aggressively check for a variety of locking errors, which include the following:

- Locking a lock out of hierarchical order
- Locking a simple lock at an SPL below the required minimum
- Locking a simple lock already held by the caller
- Unlocking an unlocked simple lock
- Unlocking a simple lock owned by another CPU
- Locking a complex lock with a simple lock held
- Locking a complex lock at interrupt level
- Sleeping with a simple lock held
- Locking or unlocking an uninitialized lock

Encountering any of these types of violation results in a lock fault, i.e., a system bug check that records the information required by the developer to quickly track down the lock error.

The reduction in lock sizes and the major enhancement of the lock package enabled the team to realize its goal of a single set of kernel binaries. Benchmarks that compare a pure uniprocessor kernel and a kernel in lock mode 0 that are both running on the same hardware show a less than 3 percent difference in performance, a cost considered by the team to be well worth the many advantages to returning to a unified kernel. Moreover, the debugging capabilities of the lock package with its hierarchical scheme streamlined the process of lock analysis and provided precise and immediate feedback as developers adapted their subsystems to the multiprocessing environment.

ADAPTING THE SCHEDULER FOR MULTIPROCESSING

The normal scheduling behavior, i.e., policy, of the OSF/1 system is traditional UNIX time-sharing. The system time-slices processes based on a time quantum and adjusts process priorities to favor interactive jobs over compute-intensive jobs. To support the POSIX real-time standard, the DEC OSF/1 system incorporates two additional fixed-priority scheduling policies: first in, first out (POLICY_FIFO) and round robin (POLICY_RR).

A time-share thread's priority degrades with CPU usage; the more recent the thread's CPU usage, the more its priority degrades. (Note that OSF/1 scheduling entities are threads rather than processes.) In contrast, a fixed-priority thread never suffers priority degradation. Instead, a POLICY_RR thread runs until it blocks voluntarily, is preempted by a higher-priority thread, or exhausts a quantum (and even then, the round robin scheduling applies only to threads of equal priority). A POLICY_FIFO thread has no scheduling quantum; it runs until it blocks or is preempted. These specialized policies are used by real-time applications and by threads created and managed by the kernel. Examples of these kernel threads include the swapper and paging threads, device driver threads, and network protocol handlers. A feature called thread binding, or hard affinity, was added to DEC OSF/1 version 3.0. Binding allows a user or the kernel to force a thread to run only on a specified processor. Binding supports the funneling feature used by unparallelized code and the bind_to_cpu() system call.

The goal of a multiprocessing operating system in scheduling threads is to run the top N priority threads on N processors at any given time. A simple way to accomplish this would be to schedule threads that are not bound to a CPU in a single, global run queue and schedule bound threads in a run queue local to its bound processor. When a processor reschedules, it would select the highest-priority thread available in the local or the global run queue.

Scheduling threads out of a global run queue is highly effective at keeping the N highest-priority threads running; however, two problems arise with this approach:

1. A single run queue leads to contention between processors that are attempting to reschedule, as they race to lock the run queue and remove the highest-priority thread.

2. Scheduling with a global run queue does not take advantage of the cache state that a thread builds on the CPU where it last ran. A thread that migrates to a different processor must reload its state into the new processor's cache. This can substantially degrade performance.

To help preserve cache state and reduce wasteful global run queue contention, the developers enhanced the multiprocessing scheduler by adding two new scheduling models: a soft-affinity scheduling model for time-share threads and a last-processor-preference model for fixed-priority threads. Under these models, each processor schedules time-share and bound threads in its local run queue, and it schedules unbound fixed-priority threads out of a global run queue.

Fixed-priority threads scheduled from a global run queue are able to run as soon as possible. This behavior is required for high-priority tasks like kernel threads and real-time applications. The last-processor-preference model gives a fixed-priority thread a preference for running on the processor where it last ran; if that processor is busy, the thread runs on the next available processor. Each time-share thread is softly bound to the processor on which it last ran; that is, the thread shows an affinity for that processor. Unlike funneling or user binding, which support hard (mandatory) affinity, soft affinity allows a thread to run elsewhere if it is advantageous, i.e., if such activity balances the load. Otherwise, the softly bound thread tries to return to the processor where it last ran and where its recent cache state may still reside.

Under load, however, a soft affinity model used alone can degenerate to a state where one processor builds up a large queue of threads, leaving the other processors with little to do and thus diminishing the performance of the multiprocessing system. To mitigate these side effects of soft affinity, developers paired the soft affinity feature with the ability to load-balance the runnable threads in the system. To keep the load of time-share jobs spread evenly across processors, the scheduler must periodically load-balance the system. In addition to distributing threads evenly across the local run queues in the system, this load-balancing activity must

o Cost no more in processing time than it saves

o Prevent excessive thread movement among processors

o Recognize and effectively accommodate changes in the job mix

To implement load balancing, each processor maintains a time-share load average, i.e., the average local run queue depth over the last five seconds. Each processor updates its own load average on each system clock tick. Processors also keep track of the time they spend handling interrupts and running fixed-priority threads, which are not accounted for in the local run queue depth. Taking a processor's total potential execution time for a scheduling period and subtracting from this time the interrupt-processing and fixed-priority run times yields the total time available on a processor (processor ticks available) to run time-share threads. In the worse case, a processor could be completely consumed by fixed-priority threads and/or interrupt processing and have no time to run time-share threads. In this extreme case, the scheduler should give no time-share load to that processor.

Adding the time-share load averages of all processors determines the aggregate time-share load for the system. Similarly, summing the processor ticks available yields the total time available on the system for handling time-share tasks. Using this data, the scheduler calculates the desired load for each processor once per second, as follows:

$$\text{Desired load} = \frac{\text{Processor ticks available} \times \text{System time-share load}}{\text{System ticks available}}$$

Load balancing is called for when at least one processor is above and one is below its desired load by a minimal amount. If this condition arises, then those processors under their desired loads are declared to be "out of balance." The next time an out-of-balance processor reschedules, it will try to take a thread from the local run queue of a processor that is above its desired load ("thread stealing"). A processor can declare itself back in balance when its current load is above its desired load or when there are no eligible threads to steal. Figure 3 shows a simplified load-balancing scenario, in which a processor below its desired load steals a thread from a processor above its desired load.

```
Figure 3   Load Balancing

             +---------+          +---------+         +---------+
             |  CPU 1  |          |  CPU 2  |   ...   |  CPU N  |
             +---------+          +---------+         +---------+

CURRENT LOAD +---------+          +---------+         +---------+
(NUMBER OF   |    5    | CPU 1 IS |    3    |         |    4    |
THREADS)     |         | OUT OF   |         |         |         |
             |         | BALANCE  |         |         |         |
DESIRED LOAD |    4    |          |    4    |         |    4    |
             |         |          |         |   ...   |         |
             |         |          |         |         |         |
             |  LOCAL  |<---------|  LOCAL  |         |  LOCAL  |
             |  RUN    |--------->|  RUN    |         |  RUN    |
             |  QUEUE  |CPU 2     |  QUEUE  |         |  QUEUE  |
             +----+----+STEALS    +----+----+         +----+----+
                  |     ONE THREAD     |                   |
                  |     FROM CPU 1     |                   |
HIGHEST PRIORITY  |                    |                   |
THREAD BETWEEN    +-----------------+  +  +----------------+
LOCAL RUN QUEUES                    |  |  |
AND GLOBAL RUN QUEUE              +-+-----+-+
WINS THE PROCESSOR                |         |
                                  |         |
                                  |         |
                                  |  GLOBAL |
                                  |  RUN    |
                                  |  QUEUE  |
                                  |         |
                                  |         |
                                  |         |
                                  +---------+
```

To help preserve the cache benefits of soft affinity, a thread is eligible for stealing only when it has not run on its current processor for some configurable number of clock ticks. After this time has elapsed without a thread running, the chance of it having significant cache state remaining has diminished sufficiently that the thread is more likely to benefit from migrating to another processor and running immediately than from waiting longer to run on its current processor.

To demonstrate that soft affinity with load balancing improves multiprocessing performance through cache benefits and the elimination of run queue contention, developers ran a simple test program. The program, which writes 128 KB of data, yields the processor, and then reads the same data back, was run on a four-processor DEC 7000 system. Table 1 shows the results of running multiple versions of this program with and without soft affinity and load balancing in operation. Performance benefits appear only when multiple copies of the program begin piling up in the run queues at the 16-job level. Prior to this point, each job keeps running on the same processor, i.e., the cache it had just filled still had its data cached when the program read it back---the ideal case. At the 16-job level, the four processors must be time-shared. The jobs that are running with soft affinity now benefit significantly because they continue to run on the same processor and thus find some of their cache state preserved from when they last ran. The systems that schedule from a global run queue provide no such benefit. Jobs take longer to complete, since they are likely to run on a different processor for each time slice and find no cache state that they can reuse.

Table 1 Benefits of Soft Affinity with Load Balancing (SA/LB)

Number of Jobs	Time with SA/LB (Seconds)	Time without SA/LB (Seconds)	Benefit from SA/LB (Percent)
1	25.9	26.0	0
4	25.9	26.0	0
16	106.5	141.9	25

The soft affinity and load-balancing features improved many other multiuser benchmarks. For example, a transaction processing benchmark showed a 17 percent performance improvement.

FOCUSING ON QUALITY

The error-checking focus of the lock package is just one example of how the DEC OSF/1 version 3.0 project focused on the quality and stability of the product. Most members of the multiprocessing team had been involved in an SMP development effort prior to their DEC OSF/1 effort. This past experience, coupled with the difficulties other vendors had experienced with their own multiprocessing implementations, reinforced the need to have a strong quality focus in the SMP project.

Developers took multiple steps to ensure that the SMP solution delivered in DEC OSF/1 version 3.0 would be production quality, including

- Code reviews
- Lock debugging
- In-line assertion checking
- Multithreaded test suite development for SMP qualification

The base kernel code was reviewed for multiprocessing correctness. During this review phase, checks were made to ensure that the proper level of synchronization was employed to protect global data structures. Numerous defects were uncovered during this process and corrected. Running code with lock checking enabled provided empirical evidence of the incremental improvements of the multiprocessing implementation.

Beyond code reviews and lock debugging, internal consistency checks (assertions) were coded into the kernel to verify correctness of operations at key points. Assertion checking was enabled during the development process to ensure that the kernel was functioning correctly; it was then compiled out for the production version of the kernel.

In parallel with the operating system development effort, new component tests were designed to force as much concurrency as possible through particular code paths. The core of the test suite is a thread-race library, which consists of a set of routines that can be used to construct multithreaded system-call exercisers. The library provides the ability to commence multiple test instances simultaneously. The individual tests are then combined to form focused subsystem tests and systemwide tests. These tests have been used to uncover multiple race conditions in the operating system.

The UNIX development organization had a four-processor DEC 7000 system deployed in its development environment for more than 7 months prior to releasing the SMP product. This system has been extremely stable, with few complaints from the user community. Extensive internal and external field testing produced similar results.

MEASURING MULTIPROCESSING PERFORMANCE OUTCOMES

The major functionality delivered with SMP is improved performance through application concurrency. The goal of the SMP project was to provide leadership performance in the areas of compute and data servers. To gauge success in this effort, several industry-standard benchmarks were utilized. These benchmarks include SPECrate_INT92, SPECrate_FP92, and AIM Suite III.

SMP performance is measured in terms of the incremental performance gained as processors are added to the system. Adding processors by no means guarantees increased system performance. Systems that have I/O or memory limitations rarely benefit from introducing additional CPUs. Systems that are compute bound tend to have the largest potential for gain from additional processors. Note that large, monolithic applications tend to see little performance improvement as processors are added because such applications employ little concurrency in their designs.

Performance tuning for SMP was an iterative process that can be characterized as follows:

1. Collect and analyze performance data.

    o CPU utilization across the processors

    o Lock statistics

    o I/O rates

    o Context switch rates

    o Kernel profiling

2. Identify areas that require improvement.

3. Prototype changes.

4. Incorporate changes that demonstrate improvement.

5. Repeat steps 1 through 4.

In reality, the process has two stages for each benchmark. The initial phase was devoted to driving the system to become compute bound. The second phase improved code efficiencies.

Figures 4 and 5 show that, as expected, the SPECrate_INT92 and

SPECrate_FP92 benchmarks scale almost linearly. Both of these benchmarks are compute intensive and make only nominal demands on the operating system.

[Figure 4 (SPECrate Integer Scaling for Four-CPU Systems) is not available in ASCII format.]

[Figure 5 (SPECrate Floating-point Scaling for Four-CPU Systems) is not available in ASCII format.]

AIM Suite III is a multiuser benchmark that stresses multiple components of an operating system, including the virtual memory system, the scheduler, UNIX pipes, and the I/O subsystem. Figure 6 shows AIM III results for one and four processors, with a resulting 3.27 to 4 scaling factor. This equates to a greater than 80 percent scaling factor, a figure well within the goals for this benchmark at first multiprocessing release. Efforts to produce still better results are under way.

[Figure 6 (AIM Suite III Scaling) is not available in ASCII format.]

AIM Suite III scaling appears to be gated by a single test in the AIM Suite III benchmark, i.e., directory search. The goal of this test is to create and remove a set of files across a limited number of directories.[10] Because these operations require updating directory information, only one thread of execution can perform these operations on a directory at a time. Some improvements have been applied to mitigate this contention, but this single test still impacts the overall scaling results.

CONCLUSION

The focus of the first release of SMP capabilities for the DEC OSF/1 operating system was to provide leadership SMP performance without compromising overall system quality. The project team accomplished this goal by carefully modifying the base operating system to take advantage of the additional processing power provided. The team paid particular attention to synchronization, parallel algorithms, and error and inconsistency detection.

Work for future releases will continue to focus on performance and quality improvements. Other areas of investigation include features such as processor sets, stopping and starting CPUs, and more flexible handling of interrupts as processors are added.

ACKNOWLEDGMENTS

Virtually every phase of this project depended on the teamwork and cooperation of multiple groups with the UNIX Software Group. The authors wish to acknowledge the tireless efforts and accomplishments of that entire organization in making DEC OSF/1 version 3.0 and SMP a reality. In particular, we would like to acknowledge the following contributors who were involved in the SMP project from its earliest stages: Tim Burke, Dan Christians, Scott Cranston, Richard Flower, Heather Gray, Gerri Harter, Tim Hoskins, Chet Juszczak, Stan Luke, Shashi Mangalat, Joe Martin, Ron Menner, Brian Nadeau, Ernie Petrides, Rajul Shah, Dave Stanley, and Tony Verhulst.

NOTE AND REFERENCES

1. The OSF/1 operating system, based on Carnegie Mellon University's Mach version 2.5 kernel, is developed and distributed by the Open Software Foundation. The DEC OSF/1 operating system, based in part on the OSF/1 system, is developed and distributed by Digital Equipment Corporation. To further clarify the relationship between the two products, DEC OSF/1 versions 1.0, 1.2, 1.3, 2.0, and 2.1 include code mainly from the OSF/1 version 1.0 software. DEC OSF/1 version 3.0 includes code from the OSF/1 version 1.1 and 1.2 software.

2. F. Hayes, "Design of the AlphaServer Multiprocessor Server Systems," Digital Technical Journal, vol. 6, no. 3 (Summer 1994, this issue): 8-19.

3. R. Rashid, "Threads of a New System (Mach: A Multiprocessor Operating System)," UNIX Review (August 1986): 37-49.

4. M. Accetta et al., "Mach: A New Kernel Foundation for Unix Development," USENIX Summer Proceedings (August 1986): 93-112.

5. Open Software Foundation, Design of the OSF/1 Operating System (Englewood Cliffs, NJ: Prentice-Hall, 1993).

6. S. Mangalat and D. Bolinger, "Parallelizing Signal Handling and Process Management in OSF/1," USENIX Symposium Proceedings (November 1991): 105-122.

7. Information Processing---Volume and File Structure of CD-ROM for Information Interchange, ISO 9660 (Geneva: International Organization for Standardization, 1988).

8. R. Sites, ed., Alpha Architecture Reference Manual (Burlington, MA: Digital Press, 1992).

9. R. Gamache and K. Morse, "VMS Symmetric Multiprocessing," Digital Technical Journal, vol. 1, no. 7 (August 1988): 57-63.

10. D. Black et al., "Translation Lookaside Buffer Consistency: A Software Approach," Proceedings of the Third International Conference on Architectural Support for Programming Languages

and Operating Systems (1989).

BIOGRAPHIES

Jeffrey M. Denham A principal software engineer in the UNIX Software Group, Jeff Denham is a contributor to the DEC OSF/1 version 3.0 symmetric multiprocessing effort. Prior to this, he helped add POSIX.1b features to the DEC OSF/1 operating system and worked on the VAXELN real-time kernel. Jeff came to Digital in 1986 from Raytheon Corporation. He holds a B.A. (1979) from Hiram College, an M.A. (1980) from Tufts University, both in English, and an M.S. (1985) in Technical Communication from Rensselaer Polytechnic Institute.

Paula Long   Since joining Digital in 1986, Paula Long has contributed to various operating system projects. Presently a principal software engineer with the UNIX Software Group, she leads the development of symmetric multiprocessing (SMP) capabilities for the DEC OSF/1 operating system. In previous positions, she led the DEC OSF/1 real-time and DECwindows on VAXELN projects. Paula received a B.S.C.S. from Westfield State College in 1983.

James A. Woodward   Principal software engineer James Woodward is a member of the UNIX Software Group. He is responsible for DEC OSF/1 symmetric multiprocessing (SMP) processor scheduling and base kernel support. In previous work, Jim led the ULTRIX SMP project and the VAX 8200, VAX 8800, and VAX 6000 ULTRIX operating system ports. He also wrote microcode for the VAX 8200 systems as a member of the Semiconductor Engineering Group. Jim joined Digital in 1981 after receiving a B.S.E.E. from the University of Michigan.