An Efficient Kernel-Based Implementation of POSIX Threads

Robert A. Alfieri
Data General Corporation

Abstract

This paper describes the kernel-based implementation of POSIX Threads (Pthreads) in the DG/UX™ operating system. The implementation achieves time efficiency by using a general-purpose trap mechanism, known as a *Kernel Function Call (KFC)*, that carries an order of magnitude less overhead than a traditional system call. On a 50 MHz Motorola MC88110, the implementation can create and exit a thread (with the associated context switch) in 8.1 microseconds and yield to another thread in 4.0 microseconds. The implementation also achieves space efficiency by paging and decoupling bulky data structures.

The advantages of a kernel-based implementation include design simplicity, less code redundancy, optimization of global (interprocess) operations, avoidance of inopportune preemption, and global semantic flexibility. The disadvantage is a monolithic design that lacks user-level flexibility.

1. Introduction

Threads provide an efficient and convenient concurrent programming paradigm for applications running on shared-memory multiprocessors. As industry-standard thread interfaces such as POSIX Threads (Pthreads) [Pos93] find their way into open systems, an increasing number of portable applications are being written (or rewritten) to exploit threads.

Support for Draft 6 of Pthreads was shipped in version 5.4R3.00 of DG/UX™, Data General's commercial UNIX® operating system. DG/UX originated in 1984 as a rewrite of the UNIX kernel in order to support Symmetric Multiprocessing (SMP), full preemption, and better modularity [Kel89].

Though this paper focuses on the techniques that DG/UX uses to implement threads efficiently in the kernel, any operating system could apply the same techniques to other thread packages or performance-critical system calls. Most importantly, none of these techniques are specific to DG/UX or POSIX Threads.

2. Design Overview

The overriding design goal is to allow standard multithreaded applications to map as efficiently as possible (both in terms of time and space) onto current and future multiprocessor architectures. In order to realize this goal, the implementation employs three main features:

Kernel Function Calls (KFCs)

KFCs are fast, general-purpose kernel traps that allow Pthreads to be implemented simply and efficiently in the kernel.

The implementation uses the same set of KFCs to optimize both local (intraprocess) and global (interprocess) thread operations. Global operations are important today and will become increasingly important as applications and machines move towards a paradigm of distributed computing.

Low Memory Overhead

By keeping the per-thread physical and virtual memory consumption to a minimum, the implementation can easily support thousands of threads in a single process.

All per-thread data structures are pageable, except a 128-byte kernel-level structure. Moreover, space-consuming kernel stacks and other *transient data* are tied only to threads that have entered the kernel for a traditional system call, page fault, or other type of full kernel entry.

Pthread Groups and Hierarchical CPU Affinity

These extensions give applications explicit control over the scheduling of threads onto the underlying multiprocessor machine.

The affinity mechanism is hierarchical in that it parallels the CPU/cache/memory hierarchy of the underlying machine. Applications may group together threads working on the same data set, then affix that thread group to a set of CPUs sharing the same cache or local memory. This feature is particularly important for improving cache locality on large SMP or NUMA (Non-Uniform Memory Access) machines. If an application does not specify its thread grouping or affinity, the operating system performs automatic thread grouping and affinity assignments.

The rest of this paper is devoted to describing the motivations and details of the implementation with respect to the first two features. The last feature is described separately [Alf94].

3. Application Requirements

There are two broad classes of thread operations:

1) *Local Operations* are those operations that occur among threads in the same process. Local operations do not involve communication with other processes and are often referred to as *intraprocess* or *intra-address-space* operations.

2) *Global Operations* are those operations that occur between threads in two different processes or between a thread and the kernel. Global operations are often referred to as *interprocess* or *inter-address-space* operations.

A thread implementation should optimize both types of operations. Because global operations require more work, they tend to be somewhat slower than local operations. However, a request should only incur a cost that is proportional to the service that must be delivered.

3.1. Local Operations

Ideally, a threaded application spends all of its time in user space performing local operations and rarely talks to the kernel or to other processes. Examples of using local-only operations include situations in which

- *Database* back-end threads perform a parallel sort of a database table stored in process memory.
- *Real-time* threads are used to prioritize certain events that are handled entirely in user space.
- *Simulation software* breaks up a problem into smaller simulations that run as threads on separate CPUs.

3.2. Global Operations

Commercial applications also make extensive use of global operations. For data integrity reasons, applications are split into multiple processes, and shared system services reside in the kernel or in privileged server processes. The kernel must always participate in (safe) global thread operations, as illustrated by the following examples in which

- *Database* front-end threads go through the kernel to talk to database back-end threads which, in turn, make kernel system calls to transfer data to or from permanent storage.
- *Real-time* threads require kernel-based global scheduling in order to minimize response time to global asynchronous events.
- *Distributed computing* services are divided into replicated multithreaded processes, which communicate using a *Local or Remote Procedure Call (RPC)* mechanism.

Clearly, global interactions play an important role in current and future multithreaded applications. Implementations should optimize both local and global cases.

4. Previous Work

This section introduces library-based implementations of threads and explains why DG/UX has not adopted the same approach. Next, the discussion focuses on previous work within DG/UX that ultimately led to an efficient kernel-based implementation.

4.1. Multiplexed Libraries

In pursuit of optimal performance, numerous multiplexed thread libraries have been developed to avoid kernel system calls, especially during performance critical operations such as thread creation and context switching [Gol90, Pow91, Ste92, Mue93].

Multiplexed libraries employ two levels of scheduling. User-level threads are multiplexed onto a typically smaller number of kernel-level entities (e.g., processes or kernel-level threads). In the simplest implementations, all user-level threads are multiplexed onto a single process. In more sophisticated implementations, the number of kernel-level entities varies with the number of CPUs that are assigned to the process. Context switches among kernel-level entities involve slower system calls. Context switches among user-level threads occur entirely in user space, thus optimizing performance. In addition, flexible multiplexed libraries [And91b, Mar91] allow applications to integrate their own thread schedulers.

Because multiplexed libraries reside in user space, highly tuned library primitives can be used only for local thread operations. Global thread operations, such as interprocess synchronization and RPCs, typically follow significantly slower system call paths or use dedicated kernel-entry mechanisms.

In addition, well-integrated multiplexed libraries require complex algorithms to bridge the wide gap between library and kernel databases. In particular, the library and kernel must continuously inform each other of the number of runnable threads in user space and the number of available CPUs in the kernel. Refer to the section on "Kernel vs. User Threads" for a more detailed discussion of this and other issues.

Despite their complexity, multiplexed libraries prevail due to the perception that system call overhead would dominate the cost of a kernel-based thread primitive.

4.2. System Call Overhead

This section explains in greater detail why multiplexed library-based systems and DG/UX have avoided using traditional system calls to implement threads.

Have System Calls Gotten Relatively Slower?

Recent research in the area of RISC processor performance [Ous90, And91a] asserts that operating system primitives such as system calls, exceptions, and process context switches have not experienced the same accelerated speedup as application integer and floating-point operations. There are two reasons why this observation may hold true for system calls in many commercial operating systems:

1) System calls are typically more memory-intensive than arithmetic operations, thereby introducing more processor stalls in the form of cache misses and degraded instruction-level parallelism.

2) System call entry has grown more complex due to the introduction of additional functionality and complexities in the kernel.

As an example, DG/UX system call overhead has increased mainly for the following reasons:

1) RISC processors have required the kernel to save and restore more registers and to do more work during system calls than previous CISC processors.

2) The desire to provide more precise execution time accounting has resulted in reading hardware timers during system call entry and exit. Historically, time accounting has been implemented using a coarse ten-millisecond clock tick.

3) An effort to make the kernel more modular and portable has introduced more inter-subsystem calls during the system call path.

Reasons (1) and (3) likely apply to many other operating systems, while reason (2) may be specific only to DG/UX and a few other commercial systems.

Regardless of the actual implementation, system call overhead is inherently significant due to user state saving and restoring, and preparing the caller for execution in a fully preemptive SMP kernel.

Impact on Thread Primitives

Most local thread primitives execute in microseconds or tens of microseconds. System call overhead, which is on the order of tens of microseconds, would contribute significantly to the overall cost of a kernel-based thread primitive. Roughly speaking, a system call imposes the equivalent of (at least) an additional thread-to-thread context switch on each kernel entry.

4.3. Down-Sizing System Calls

Fortunately, system calls are not the only way to enter the kernel. Modern RISC and CISC processors provide fast trap instructions, which can be used to invoke simpler kernel primitives that are an order of magnitude faster than traditional system calls.

For example, on a (relatively slow) 25 MHz Motorola MC88100 [Mot91], the cost of trapping into the kernel and returning from the trap, including proper Processor Status Register (PSR) and Program Counter (PC) setup and restoration, is less than one microsecond. A null system call on the same machine takes approximately 20 microseconds.

The actual trap into the kernel is inexpensive. The real issue is the costly steps that must be performed in order to set up for a traditional system call. In fact, most of these steps can be avoided by using a simpler primitive.

4.4. DG/UX Extended Operations (XOPs)

For a number of years, DG/UX has used fast kernel traps to implement certain extended operations (XOPs) that are not provided directly by the Motorola 88000 instruction set, such as atomic-increment and conditional-store to user memory locations. Though these atomic operations could have been implemented completely in user space using a lock based on the 88000's test-and-set instruction, there was a desire to avoid any unbounded delay associated with *inopportune preemption* (e.g., hardware interrupt, page fault, process abort) while holding a user-level spin lock or mutex.

XOPs prevent this inopportune preemption by using simple traps across the user-kernel protection boundary. A null XOP takes less than one microsecond and scales directly with processor speed. Atomicity among multiple CPUs is provided using a spin lock inside the kernel. In the 88000 family, there is no need to disable CPU interrupts explicitly because they are implicitly disabled by the trap. Thus, preemption is avoided while executing an XOP and holding a spin lock.

Once the XOP has acquired the spin lock, the XOP is free to perform the atomic increment or conditional store on the user memory location. However, this memory location could be paged out or invalid. Because the XOP holds a critical spin lock and has not fully entered the kernel, it must be prepared to deal with a fault when touching the user memory location. To this end, a flag is set in per-CPU data indicating that an XOP is in progress. If a fault occurs while touching user memory, the fault prehandler ignores the fault and backs out of the XOP. The failure is reported back to the user-space library routine, which is then responsible for touching the memory location. Touching the memory location causes a normal page fault to bring in the memory page. Then the library routine retries the XOP; the XOP almost always succeeds on the second try.

4.5. Fast Traps in Other Systems

Fast traps have been used in numerous other operating systems. In particular, fast traps have been successfully exploited for the purpose of speeding up *Interprocess Communication (IPC)* [Ber90, Lie92, Wal92, Lie93]. Like XOPs, these traps are specific to the primitive that they implement.

5. Kernel Function Calls (KFCs)

The powerful combination of fast kernel trap and fault interception used in XOPs is at the heart of the kernel-based thread implementation. However, XOPs do not provide the level of semantic flexibility and ease-of-use that thread primitives require. In particular, XOPs must be implemented in assembly language, the XOP back-out mechanism is cumbersome, and XOPs do not check for preemptions before leaving the kernel. For these reasons, XOPs were abandoned in favor of a more general and usable flavor of fast kernel trap, known as a Kernel Function Call (KFC).

5.1. KFC Semantics

As the name implies, a KFC is a function that resides in the kernel and is callable from user space. Calls to KFCs follow the same parameter passing and register saving conventions of the underlying processor architecture. Returns from KFCs follow the same conventions as UNIX system calls on the underlying processor architecture. KFCs carry more overhead than an XOP, but still an order of magnitude less overhead than a traditional system call. Actual KFC overhead is on the order of 20-30 machine instructions and scales directly with processor speed.

KFCs operate in a restricted environment where CPU interrupts are disabled, time is still charged to user space, and blocking with a kernel stack is not permitted. Although these restrictions could be lifted in order to implement all system calls using KFCs, only the shortest system calls would benefit noticeably. Also, complete generality would bloat KFC overhead and introduce undesirable complexities, such as the need to unwind a KFC so that a user debugger could manipulate a thread's user registers. For these reasons, as well as the desire to avoid other likely additions to the KFC entry and exit paths, DG/UX limits KFC usage to performance-critical primitives (e.g. Pthreads) that can operate in this restricted environment. Other potential uses include `getpid()`, `gettimeofday()`, and RPCs.

Appendix A gives a complete list of thread-related KFC interfaces. Table 1 illustrates the semantic differences among XOPs, KFCs, and system calls. Note that numerous costly steps are avoided during KFCs and XOPs.

Aspect of Kernel Entry/Exit	XOP	KFC	SYSCALL
Enters the kernel using a fast trap instruction and leaves the kernel using a fast return-from-trap instruction	✓	✓	✓
Uses common entry/exit code		✓	✓
Performs a full register state save on entry and full register restore on exit			✓
Switches to system time accounting on kernel entry, then back to user time accounting on exit			✓
Prepares the environment for full kernel entry and enables interrupts			✓
Executes with CPU interrupts disabled	✓	✓	
Can be (and is typically) written in C		✓	✓
Can block the calling thread and switch to another thread		✓	✓
Frees up the kernel stack for reuse when the calling thread is blocked		✓	
Can unblock other threads and check for preemption before leaving the kernel		✓	✓
Handles faults directly without explicit detection			✓
Backs out of faults that occur while touching pageable user or kernel memory	✓	✓	
Backs out of faults and *promote*s to an internal system call to handle the fault		✓	
Follows system call return conventions		✓	✓

Table 1: Semantic Differences

5.2. KFC Details

The following sections describe the general KFC mechanism in more detail, with emphasis on using KFCs to implement thread primitives. The reader may wish to browse or skip these details, then return to them after reading the rest of this paper.

5.2.1. KFC Entry

The steps in KFC entry proceed as follows for all KFCs:

1) A thread makes a call to a library routine.

2) The library routine typically performs a few steps, then decides to invoke a KFC. The routine packages some arguments and traps into the kernel through a common vector.

   How the arguments are packaged and how the trap is performed are architecture-dependent. For example, on RISC machines, the arguments are typically passed into the kernel via registers.

   On CISC machines, the processor may automatically copy the arguments into the kernel, or it may provide enough registers to hold all of the arguments.

   The amount of work that the trap instruction does depends on the underlying processor architecture. RISC traps are typically faster than CISC traps, but CISC traps may perform more steps, such as automatically switching to a kernel stack. On modern RISC and CISC processors with the same clock speed, KFC entry overhead tends to be similar.

3) Once in the kernel, common KFC entry code saves the thread's user-level return address and PSR in per-CPU data.

   Assuming that all KFCs are written in C, the entry code saves the user-level stack pointer and switches to a kernel-level stack. On CISC architectures, these operations may be performed automatically by the trap instruction.

4) Next, the entry code transfers control to the actual KFC. On RISC architectures, the arguments to the KFC are typically already in the appropriate registers. On CISC architectures, the arguments may need to be pushed onto the kernel stack before the call. However, argument pushing comes cheaply on modern CISC processors.

5) The actual KFC looks like a traditional system call written in C.

5.2.2. KFC Exit

When the thread has completed its KFC, the steps in KFC exit proceed as follows for all KFCs:

1) From within the KFC, the thread temporarily stores its primary and secondary return values to per-CPU data variables. Note that the use of per-CPU data is allowed because CPU interrupts are disabled.

2) The thread returns naturally from the KFC. The return status indicates whether the KFC completed normally.

3) The common KFC exit code reads a different per-CPU variable to determine if the calling thread should check for priority preemption. This occurs if the thread had awakened higher priority threads during the KFC. In this case, the calling thread yields to another thread and will eventually return to this exit code.

4) The KFC exit code interrogates the return status from the KFC. For a normal return, the saved primary and secondary return values are returned to the library routine. The conventions for returning these values are the same as those for system calls. For a failure return, the `errno` value is extracted from the return status, and the primary and secondary return values are ignored.

5) The KFC exit code restores the partially saved user state, and executes the appropriate instruction(s) for returning to user space.

6) The library routine interrogates the results of the KFC in the same way that a library routine would interrogate the results of a system call.

7) The library routine may perform other steps before returning to its caller.

5.2.3. Thread Reschedule

Some KFCs do more than enter the kernel, perform a few steps, and return. In numerous cases, the calling thread must suspend its execution. Examples of thread suspension include the following:

- The calling thread attempts to *join* (wait for) a thread that has not yet terminated.
- The calling thread explicitly *yields* to another thread of equal or better priority.
- The calling thread awaits the *release* of a *mutex* or the *signaling* of a *condition variable*.
- The calling thread *sleeps* for a specific amount of time.

When a thread suspends, it unwinds the KFC that it is executing so that the thread's register state is readily accessible. Then common KFC code saves the full register state in a standard `mcontext_t` structure that resides inside the kernel. Next, the thread calls into the dispatcher, which switches control to another thread.

Eventually, the original thread will be awakened and a CPU will continue the thread's execution. The thread's *continuation function* [Dra91] restores the thread's register state from its `mcontext_t` structure and resumes the KFC where it left off.

5.2.4. KFC Fault Detection

Like XOPs, KFCs must back out of faults that occur while referencing pageable kernel or user memory. There are two main reasons for backing out:

1) The KFC operates in a restricted kernel environment with CPU interrupts disabled. Full fault handling is not permitted in this environment.

2) The KFC typically holds a critical spin lock. If the KFC were to allow the fault to be processed, the spin lock could remain held for an indefinite period of time. This could lead to deadlock or extremely long latencies for other threads in the same process. *Fault detection actually eliminates long latencies associated with critical locks that are used to implement threads.*

Faulting on Kernel and User Memory

Because `mcontext_t` structures can consume several hundred bytes, depending on the processor architecture, the implementation allows these structures to be paged. KFCs must be prepared to back out of page faults that occur while saving/restoring state to/from the per-thread `mcontext_t` structure located in the kernel.

In addition, a few thread KFCs need to reference user space from inside the kernel. For example, when a thread cannot immediately obtain a Pthread mutex in user space (i.e., failing the uncontested case), the thread invokes a KFC to await the release of the mutex. In this contested case, the KFC saves the thread's state and adds the thread to a kernel-level synchronization queue associated with the mutex. However, due to the potential for certain race conditions, the KFC must re-check the user-level mutex to see if the mutex has been released before actually putting the thread to sleep.

The KFC, while attempting to touch the mutex, must be prepared to back out of two types of faults:

1) The user page holding the mutex is paged out.

2) The user mutex address passed into the KFC no longer refers to valid user memory. Although the Pthread library dereferences the mutex address and checks the validity of the corresponding mutex structure, another thread could erroneously unmap the mutex memory before the library invokes the KFC.

Detecting Faults

Whenever a KFC needs to touch kernel or user memory that could cause a fault, the KFC calls a trivial assembly language routine to attempt the memory operation. Before performing the operation, this common routine sets a per-CPU variable to the address of back-out code at the bottom of the routine. Under normal operation, this per-CPU variable holds a value of zero, indicating that faults are not being intercepted.

As with XOPs, if a fault occurs while touching the memory location, the appropriate fault prehandler checks the per-CPU variable. Since the variable has a nonzero value, the prehandler knows not to proceed normally to the full fault handler. Instead, the prehandler ignores the fault and branches to the back-out code at the bottom of the common assembly language routine. The back-out code returns to the KFC, indicating that a failure occurred while touching the memory location. If no fault had occurred, the assembly language routine would have simply returned normally after successfully manipulating the memory location.

Unlike XOPs, whenever a KFC access to kernel or user memory fails, the KFC is *promoted* directly to an internal system call without returning to user space, as discussed in the next section.

5.2.5. KFC Promotion and Demotion

In most cases, thread operations are completed entirely at the KFC level. However, whenever something complicated occurs that cannot be handled at the KFC level, the KFC must be *promoted* to an *internal system call*. These rare complications include the following:

1) Faulting on user memory, such as a user-level mutex.

2) Page faulting on kernel memory, such as the per-thread `mcontext_t` during register state save or restore.

3) Failing to allocate memory resources without blocking, for example, during thread creation when no dead thread is available for “reincarnation.”

4) Detecting the presence of a software interrupt sent to the calling thread for such events as a signal, Pthread cancellation, abort, or stop.

In each of these cases, the cost of completing the operation is high relative to system call overhead. Therefore, neither internal system call performance nor promotion overhead is critical.

Implementation

The basic idea behind KFC promotion is to make the environment appear as if the calling thread invoked a system call instead of a KFC. The promotion proceeds as follows:

1) The detecting KFC stores in per-CPU data the address of its associated internal system call and parameters for the call.

   Typically, most of the parameters that were passed to the KFC are also passed to the internal system call. Furthermore, because complications can occur during various *phases* of the KFC, an indication of the phase is also passed as an extra argument to the internal system call. This indicator tells the internal system call where to resume the operation, for previous phases have already been completed successfully.

2) The KFC returns a special status code, indicating that the KFC is being promoted to a system call. The common KFC exit code recognizes this special return status and proceeds with the promotion instead of returning from the KFC. Essentially, the KFC exit code arranges the environment so that it appears as if the calling thread has just trapped into the kernel to execute a system call. However, instead of having passed in a system call number, the thread has “passed in” the kernel address of the internal system call function. Also, unlike a failed XOP, the promoted KFC does not force the thread to return to user space; the promotion takes place completely within the kernel.

3) The calling thread follows the normal system call entry path, except that the system call entry code understands that this is an internal system call. In particular, the handler knows that the internal system call address has been “passed in” rather than a system call number.

4) The internal system call interrogates its arguments to determine the next phase that needs to be completed. Because the thread is running in a full-fledged system call, the thread is free to page fault on user memory, handle signals, etc.

5) Once the thread completes the slow phase, it may decide to complete other phases, suspend itself, or return from the system call.

   If the thread decides to suspend itself (in the same way that it would have in the KFC), it *demotes* the system call to the KFC so that it can free up its kernel stack for reuse (refer to the section on "Transient Data"). When the thread is continued, it resumes at the KFC level as if it had never been promoted. Note that the KFC may get promoted again if it encounters a complication during a later phase.

   If, on the other hand, the KFC had been promoted to handle one of the last phases of the KFC, the thread simply returns from the internal system call after completing these phases. When the thread gets back to user space, the library routine recognizes no difference because KFCs follow the same return conventions as system calls.

Note the following about KFC promotion and demotion:

- Only those KFCs that can encounter complications are promotable.
- Each promotable KFC is associated with an internal system call that mimics some of the phases of the KFC.
- Internal system calls tend to be significantly larger than their corresponding KFCs because the KFCs handle only the common, performance-critical cases. In order to reduce code redundancy, internal system calls share most of the same support code as their corresponding KFCs. With minor effort, it should be possible to condense the KFC and the internal system call into one routine.
- The system call entry code remains nearly the same for both normal system calls and internal system calls. The only differences are the specification of the actual call routine and, on some CISC architectures, where the arguments are located. The system call exit code is exactly the same for both types of system call.
- Given that promotion is rare, the only good reason for demotion is kernel stack reuse.

5.3. Performance Measurements

The following tables demonstrate that thread operations can be implemented efficiently using KFCs. All measurements were taken on a dedicated Motorola 50MHz MC88110 uniprocessor [Mot91] using Draft 6 of Pthreads as shipped in the DG/UX 5.4R3.00 operating system. Each reported measurement reflects the average elapsed time per iteration of a loop that repeatedly invokes the associated primitive(s).

Kernel Entry Overhead

Table 2 illustrates the costs of XOPs, KFCs, and system calls. All calls include the overhead of a user-level "wrapper" function in a shared library.

Note that the null system call time is accurate for the DG/UX kernel even though the time exceeds the number reported by research projects such as [And91a]. This discrepancy is explained by the additional steps that DG/UX takes in order to support SMP, precise time accounting, multiple APIs, and greater internal code modularity.

Kernel Entry Type	Time (usec)
Null XOP (basic trap overhead)	0.5
Null KFC (written in C)	1.4
Null System Call	19.5

Table 2: Kernel Entry Overhead

Local Operation Overhead

Table 3 gives times for local (intraprocess) thread operations. As in library-based systems, local threads are the fastest because they share the same CPU time accounting (and timeslice) and get scheduled onto process-local queues. During local context switches, the implementation can thus avoid time bookkeeping and global scheduling decisions.

In all cases, local thread operations outperform the null system call. The breakdown for thread create/exit was determined using a hardware logic analyzer

Local Operation	Time (usec)
Null thread yield (no context switch)	1.8
Thread Yield (with context switch)	4.0
Thread create/exit (with context switch) Breakdown: • create (with KFC entry/exit) • exit (with KFC entry) • context switch (with KFC exit) • invoke new thread's start routine	8.1 4.0 1.6 2.1 0.4
Contested mutex lock/unlock (with context switch)	12.3
Condition wait/signal (with context switch)	13.2
Thread create/exit/join (with both context switches)	14.1
Timed condition wait/signal (with context switch and one-second time-out overhead)	17.7

Table 3: Local Thread Operations

Task Creation Overhead

Table 4 gives the time required to create and wait for a thread or process to exit. Times are given for both locally and globally scheduled Pthreads. Unlike local threads, global threads have dedicated CPU time accounting and get scheduled onto system-wide queues. The local time has been copied from Table 3 for comparison. Total time includes the context switches to and from the new thread or process.

Create/Exit/Wait (with both context switches)	Time (usec)
Locally scheduled thread	14.1
Globally scheduled thread	57.1
Process (using fork)	7738.9

Table 4: Task Creation Overhead

Context Switch Overhead

Table 5 gives times for various types of context switches involving Pthread condition variables. The overhead for obtaining and releasing the associated mutex is included in the condition-variable times. In Table 3, a time was given for locally scheduled threads using local (intraprocess) condition variables. Table 5 repeats the same value and includes those for other combinations of locally vs. globally scheduled threads and local vs. global condition variables. Table 5 also includes a time for two processes using a global condition variable in shared memory. Note that two processes cannot communicate via local condition variables because local condition variables are, by definition, only available within a single process. Lastly, times are included for traditional UNIX (System V IPC) semaphores in order to illustrate the advantage of using condition variables in multitasking applications.

Sleep/Wakeup Pair (with context switch)	Time (usec)
Local threads, local condition	13.2
Local threads, global condition	13.7
Global threads, local condition	33.6
Global threads, global condition	38.4
Two processes, global condition	38.9
Local threads, UNIX semaphore	99.7
Global threads, UNIX semaphore	107.9
Two processes, UNIX semaphore	116.1

Table 5: Context Switch Overhead

Comparison with Library Implementations

Table 6 compares the local-operation performance of DG/UX threads with two library implementations that were presented at previous USENIX conferences. The first is a pure library implementation from Florida State University [Mue93], which reported best numbers for a 40MHz SPARC™ IPX. The second is the SunOS™ multiplexed implementation running on a 25MHz SPARC 1+ [Pow91]. Given the disparity in processor types and the fact that thread implementations undergo continual tuning, Table 6 does not attempt to declare one implementation superior to the others. Nonetheless, if one scales the measurements in Table 6 by processor speed, one sees that kernel-based implementations can perform local Pthread operations in roughly the same time as library-based implementations.

Local Operation	DG/UX 50MHz Moto 88110 (usec)	FSU 40MHz SPARC IPX (usec)	SunOS 25MHz SPARC 1+ (usec)
Thread Create (no context switch)	4.0	12.0	56.0
Mutex Lock/Unlock (with context switch)	12.3	51.0	not given
Condition wait/signal (with context switch)	13.2	55.0	158.0

Table 6: Comparative Measurements (not scaled)

Summary

KFCs make both local and global thread operations efficient. In particular, the performance of local and global operations roughly matches the performance of local operations cited by previous library implementations.

6. Minimizing Memory Consumption

In order to meet the design goal of minimizing per-thread memory consumption, the implementation pages and decouples data structures wherever possible. Since DG/UX already pages kernel data, the paging of thread structures did not require special work in the virtual memory subsystem.

6.1. Kernel-Level Thread Structure

The only per-thread data that cannot be paged is the 128-byte main control block structure that resides in the kernel address space. An application that uses as many as 1,024 threads consumes only 128KB. Given current trends in memory capacities, this amount of overhead is insignificant. Even on a 16MB workstation, swap space for 1,024 threads (and their associated user stacks) is a more limited resource than 128KB of physical memory. On a medium-sized server (the primary design target), the impact of this 128KB is even more negligible.

6.2. User-Level Thread Structure

Every thread has a small user-level structure that resides at the base of the thread's stack. This structure and its enclosing thread stack are entirely pageable. The following information is stored in this structure:

- a cached copy of the thread's ID
- the per-thread errno variable
- a pointer to the thread's most recently pushed Pthread cleanup handler
- the values for the Pthread thread-specific data variables

6.3. Kernel-Level mcontext_t

Every thread has a kernel-level `mcontext_t` structure that is used to store the thread's register state when it is suspended in a KFC. This standard UNIX structure can consume several hundred bytes and is entirely pageable. As discussed earlier, KFCs must be prepared to back out of page faults on this structure while saving or restoring register state.

6.4. Transient Data

When a thread enters the kernel for a traditional system call, hardware interrupt, page fault, or other type of exception, the thread must run on a kernel stack. The kernel stack, along with other variables that are needed for one trip inside the kernel, make up *transient data*. The size of transient data is approximately 8KB; thus, it is too expensive to provide each thread with a dedicated transient data area. In addition, transient data must be made temporarily non-pageable while a thread is running on a CPU.

In order to reduce the amount of pageable and non-pageable transient data, the implementation *decouples* transient data from threads that are no longer using it. As the word "transient" implies, transient data are needed for only one trip inside the kernel and can migrate from thread to thread. The following rules summarize transient data allocation requirements:

- When a thread is actually running on a CPU, the thread must have non-pageable transient data because the thread must be able to enter fully into the kernel for any reason (e.g. to process an exception).
- Once a thread enters fully into the kernel, its transient data remain assigned to the thread until the thread leaves the kernel or exits. However, once the thread is descheduled (e.g. while sleeping for a long time in a system call or when the system is loaded), the kernel may decide to page out the thread's transient data.

- In contrast, when a thread suspends in a KFC, it relinquishes its transient data for reuse by other threads in the process. This is possible because the thread has not entered fully into the kernel, has saved its state in its kernel-level `mcontext_t` structure, and no longer needs its kernel stack. This rule applies also to global (interprocess) operations that are implemented using KFCs.
- When a suspending thread demotes an internal system call to a KFC, the suspending thread relinquishes its transient data. Similarly, when a thread is preempted directly out of user space as a result of taking a timeslice or other hardware interrupt, the suspending thread demotes to the KFC level and frees up its transient data.
- The kernel regulates the number of transient data areas in the process based on the current number of threads that are making system calls, page faults, etc. The kernel regulates the number of non-pageable transient data areas based on system-wide load and process priorities.
- *In a typical application, it is common to have many threads sharing a small number of transient data areas.*

7. Kernel vs. User Threads

Given that threads can be implemented efficiently in the kernel, what are the advantages and disadvantages of kernel-based threads vs. library-based threads?

7.1. Design Simplicity

Kernel-based implementations are inherently simpler than multiplexed implementations because they employ only one level of thread scheduling; they need not treat both user-level threads and kernel-level entities. This eliminates the communication gap between library and kernel databases that is common in multiplexed implementations. The kernel sees all threads in the system and can simply schedule runnable threads directly onto available CPUs with minimum latency. Furthermore, because critical data structures (including sleep queues) reside in the kernel address space, an errant application cannot corrupt them.

A kernel-based implementation also reduces the amount of code redundancy found in multiplexed implementations. For example, there is no need to have two sets of user synchronization primitives, one for user-level threads and one for kernel-level entities. The same holds true for thread creation, signal handling, thread timeslicing, user debugging support, etc.

On the other hand, many Pthread semantics creep into the kernel. If the thread library does not provide support for a particular thread package or function, the application writer may need to wait until the kernel provides the desired functionality. Naturally, many thread packages can be emulated efficiently on top of existing Pthread or KFC semantics. Nonetheless, as a kernel-based implementation accumulates more semantics, its monolithic design increases in complexity.

7.2. Performance Focus

Library-based systems focus their performance effort in the user-space thread library. Thus, highly-tuned thread primitives can be exploited only for local thread operations.

Kernel-based systems focus their performance effort in the kernel address space. Highly-tuned KFCs can be used for both local and global thread operations. For example, one KFC handles thread suspensions for both local and global condition variables. As a result, when the same type of thread is involved, the overhead for a global condition variable is only slightly greater than that for a local condition variable.

7.3. Avoiding Inopportune Preemption

Library-based implementations protect thread data structures with critical user-level locks. If a preemption or page fault occurs while a critical lock is held, other thread operations could be delayed for tens of milliseconds (or more) until the critical lock is finally released. Such latencies hurt performance, scale badly with increased numbers of CPUs, and can cause real-time applications to miss deadlines.

While complex schemes exist to work around this problem, a kernel-based implementation provides a natural solution. During KFCs, interrupts are disabled, so no timeslice or other preemption can occur. Furthermore, if a page fault occurs while touching user space (e.g., retrying a mutex before going to sleep), the page fault is intercepted, any critical kernel spin lock is released, and the KFC is promoted to an internal system call to handle the page fault as if it had occurred from user space.

7.4. Semantic Flexibility

Generally speaking, library-based implementations provide more local semantic flexibility, while kernel-based implementations provide more global semantic flexibility.

Unlike the most advanced multiplexed implementations [And91b, Mar91], kernel-based implementations do not allow applications to directly manipulate the internal scheduling mechanism or integrate their own scheduling policies. Though POSIX provides a rich set of basic mechanisms and policies on which more complex constructs can be built, a flexible library implementation will allow the same constructs to be expressed, while maintaining excellent integration with the rest of the library.

On the other hand, a kernel-based implementation can efficiently accommodate additional types of global semantics. For example, DG/UX allows applications to establish affinity relationships between groups of threads and sets of CPUs sharing the same cache or local memory [Alf94]. The implementation maintains local-operation performance within Pthread groups, while minimizing penalties for intergroup operations. Library-based implementations typically inflict greater penalty for CPU affinity because they require that user threads be "bound" to slower kernel-level entities.

7.5. User Tools

In a kernel-based implementation, the kernel sees all threads in the system. This simplifies and enhances the development of user tools that support threads. For example, user debuggers need only converse with the kernel, and the kernel always supplies a consistent set of information. In library-based implementations, the debugger must talk to both the kernel and the user-level library, and the library could be in an inconsistent state.

DG/UX provides a **ps** command option that displays the detailed status of all threads in the system. This status information includes, for example, the user address of the mutex or condition variable on which a thread is blocked. Most library-based implementations show information only for the underlying kernel entities, which have little correlation with library-level threads.

7.6. The Best of Both Approaches

Library-based systems could implement their kernel-level entities more efficiently using KFCs. Such a hybrid would improve the performance of global operations, while retaining flexibility in user space.

Conversely, kernel-based systems could provide additional mechanisms found in flexible multiplexed implementations. For example, a kernel-based implementation could provide Pthread extensions that allow a parallel language runtime library to better regulate the number of active worker threads based on the current number of assigned processors.

8. Conclusions

This work demonstrates how challenging basic assumptions (e.g., that threads cannot be implemented efficiently using system calls) can favorably redirect the course of a design. This work also illustrates the trade-offs involved in implementing operating system primitives in kernel space vs. user space.

For example, when KFCs are combined with the conservative use of memory, the result is a simple and efficient kernel-based implementation of POSIX Threads. While such an implementation forfeits some semantic flexibility in user space, it optimizes both local and global operations, and can easily accommodate additional global semantics.

Finally, any operating system could use the same general KFC mechanism to implement its own thread package or other types of fast kernel primitives, including the following global operations:

- Local or remote procedure call
- Global asynchronous event notification in the form of a newly created thread
- Interprocess thread creation or migration
- Quick `gettimeofday()`, `getpid()`, etc.

9. Acknowledgments

Eric Hamilton and Michael Kelley developed XOPs. Steve Daniel suggested using fast kernel traps to implement thread primitives. Jeff Kimmel provided a number of other useful design suggestions. Many thanks to the following persons who contributed to the DG/UX implementation of POSIX Threads: Richard Barnette, Jeff Beneker, Gina Couch, Ming Hwang, David Lakey, Etta LeBlanc, Ed McAdams, Earle MacHardy, Bill McGrath, Mark O'Connell, Cyril Sagan, Lee Sanders, Ed Savage, Guy Simpson, Eric Vook. Special thanks to Tom Ash, Dean Herington, and Andy Huber for their careful review of this paper.

10. References

[Alf94] R.A. Alfieri, "Pthread Groups and Hierarchical CPU Affinity," Data General Corporation, in progress.

[And91a] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska, "The Interaction of Architecture and Operating System Design," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 108-120, April 1991.

[And91b] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 95-109, October 1991.

[Ber90] B.N. Bershad, T.E. Anderson, E.D. Lazowska, H.M. Levy, "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systems*, volume 8, number 1, pp. 37-55, February 1990.

[Dra91] R.P. Draves, B.N. Bershad, R.F. Rashid, R.W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 122-136, October 1991.

[Gol90] D. Golub, R. Dean, A. Florin, R. Rashid, "UNIX as an Application Program," *Proceedings of the Summer 1990 USENIX Conference*, pp. 87-95, June 1990.

[Kel89] M.H. Kelley, "Multiprocessor Aspects of the DG/UX Kernel," *Proceedings of the Winter 1989 USENIX Conference*, pp. 85-99, January 1989.

[Lie92] J. Liedtke, "Fast Thread Management and Communication Without Continuations," *Proceedings of Microkernel and Other Kernel Architectures*, pp. 213-221, April 1992.

[Lie93] J. Liedtke, "Improving IPC by Kernel Design," *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 175-188, December 1993.

[Mar91] B.D. Marsh, T.J. LeBlanc, M.L. Scott, and E.P. Markatos, "First-Class User-Level Threads," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 110-121, October 1991.

[Mot91] *MC88110: Second Generation RISC Microprocessor User's Manual*, Motorola Inc., 1991.

[Mue93] F. Mueller, "A Library Implementation of POSIX Threads under UNIX," *Proceedings of the 1993 Winter USENIX Conference*, pp. 29-41, *January 1993.*

[Ous90] J.K. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proceedings of the Summer 1990 USENIX Conference*, pp. 247-256, June 1990.

[Pos93] POSIX P1003.4a/D8, *Threads Extension for Portable Operating Systems*, IEEE Computer Society, October 1993.

[Pow91] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, "SunOS Multi-thread Architecture," *Proceedings of the 1991 Winter USENIX Conference*, pp. 65-80, January 1991.

[Ste92] D. Stein, D. Shah, "Implementing Lightweight Threads," *Proceedings of the 1992 Summer USENIX Conference*, pp. 1-9, June 1992.

[Wal92] J. Walpole, J. Inouye, R. Konuru, "Modularity and Interfaces in Micro-kernel Design and Implementation: A Case Study of Chorus on the HP PA-Risc," *Proceedings of the 1992 Workshop on Micro-kernel and Other Kernel Architectures*, pp. 71-82, April 1992.

Robert A. Alfieri works for Data General's UNIX kernel group in Research Triangle Park, NC, where he has developed threads, hierarchical affinity scheduling, virtual memory, and other kernel functionality. He received a B.S. degree in Computer Science from Northwestern University in 1985. His technical interests include operating systems, molecular nanotechnology, and Italian cooking. He can be reached at Data General Corporation, 62 T.W. Alexander Drive, Research Triangle Park, NC 27709, or electronically at alfieri@dg-rtp.dg.com.

Appendix A: Thread KFC Interfaces

This section lists all KFC interfaces for Pthreads and Pthread Groups. Like other UNIX systems [Pow91], DG/UX refers to kernel-level threads as LWPs (Light Weight Processes). However, DG/UX LWP interfaces differ and, as discussed earlier, DG/UX dedicates a very efficient LWP to each thread.

In all cases, a trivial "wrapper" function in the Pthread library invokes the appropriate KFC. Upon completion, the KFC returns values and status in the same way as a traditional system call. Because DG/UX ships the Pthread library only in shared form and does not publish the KFC interfaces, these interfaces can change from revision to revision without requiring that users relink their applications.

The following KFCs create, terminate, join, detach, and manipulate the scheduling of threads. In the case of `__lwp_create()`, the new thread begins execution at a pre-registered library function that invokes the thread's start routine on the thread's user stack.

```
int __lwp_create(
   lwp_sched_info_t   sched_info,
   lwp_stack_info_t   stack_info,
   void              (*start_rtn)(),
   void *             start_arg,
   lwp_group_id_t     group_id);

int __lwp_join(
   lwpid_t            lwpid);

int __lwp_exit(
   void *             exit_status);

int __lwp_detach(
   lwpid_t            lwpid);

int __lwp_set_sched(
   lwpid_t            lwpid,
   lwp_sched_info_t   sched_info);

int __lwp_get_sched(
   lwpid_t            lwpid);

int __lwp_yield(
   int                to_peers,
   int                is_global);

int __lwp_sleep(
   unsigned           seconds,
   unsigned           nseconds);
```

The following KFCs manipulate synchronization queues that are used to implement the contested cases for mutexes and condition variables. The Pthread library does not invoke KFCs in the uncontested cases.

```
int __lwp_sq_alloc(
   lwp_sq_info_t     creation_info);

int __lwp_sq_dealloc(
   lwp_sqid_t        sq_id);

int __lwp_sq_sleep(
   lwp_cond_t *      cond_ptr,
   lwp_mutex_t *     mutex_ptr,
   struct timespec * abstime_ptr,
   lwp_sqid_t        sq_id);

int __lwp_sq_wakeup(
   lwp_sqid_t        sq_id,
   int               is_broadcast);
```

The following KFCs create, destroy, and manipulate DG/UX Pthread Groups, which are discussed separately [Alf94].

```
int __lwp_group_create(
   lwp_sched_info_t  sched_info);

int __lwp_group_destroy(
   lwp_group_id_t    lwp_group_id);

int __lwp_group_set_sched(
   lwp_group_id_t    lwp_group_id,
   lwp_sched_info_t  sched_info);

int __lwp_group_get_sched(
   lwp_group_id_t    lwp_group_id);

int __lwp_group_get_times(
   lwp_group_id_t    lwp_group_id,
   struct timespec * user_time_ptr,
   struct timespec * sys_time_ptr);
```