Notes on Thread Models in Mach 3.0

Bryan Ford
Mike Hibler
Jay Lepreau

UUCS-93-012

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

April, 1993

Abstract

During the Mach In-Kernel Servers work, we explored two alternate thread models that could be used to support traps to in-kernel servers. In the "migrating threads" model we used, the client's thread temporarily moves into the server's task for the duration of the call. In the "thread switching" model, an actual server thread is dispatched to handle client traps. Based on our experience, we find that the migrating threads model is quite complex and difficult to implement in the context of the current design of Mach and the Unix single server. The thread switching model would fit more naturally and would probably be much simpler and more robust than migrating threads, making it a valuable approach to explore in the near future. However, we believe migrating threads inherently to be faster than thread switching, and ultimately to be the best long term direction.[1]

[1]This research was sponsored by the Hewlett-Packard Research Grants Program.

Introduction

In our Mach In-Kernel Servers (INKS) work[2], one important implementation issue we had to confront is the thread semantics employed during traps to in-kernel servers. Should the client's thread "move" into the server task for the duration of the RPC, or should the kernel switch to a "real" thread created and owned by the server? The latter adheres to existing Mach semantics, and therefore would be more straightforward. However, for our initial implementation we chose the former option, for several reasons:

- It offers the greatest potential for performance improvement from INKS.
- It tests the feasibility of such a stretching of Mach semantics.
- It offers the flexibility to support other specialized communication mechanisms.
- We did not realize the extent to which the Unix server relied on the traditional implementation of server threads.

We now discuss the benefits and drawbacks of the "migrating threads" approach we took, as well as the alternate "thread-switching" method.

Migrating Threads

In our implementation of INKS, when a client takes a trap to an in-kernel server, its thread temporarily moves from the client task to the server task. The thread's `task` pointer is changed to indicate the server task, and a new server stack is attached to the thread, allocated from a pool of free server stacks allocated specifically for the use of INKS traps. When the server work function returns, the thread moves back to the client's task and the original client stack is re-attached. While this approach sounds reasonably clean and natural, there are many complications that showed up in actual implementation.

- Allowing threads to migrate to the server task creates a major security issue. Mach threads can normally be manipulated at any time by any code that has access to the thread's control port. Generally, this control port is not highly sensitive and is made available to client tasks. However, if other threads in a client task were allowed to perform operations like `thread_set_state` on a thread while executing in the server as an INKS trap, a serious security breach would occur. We have not yet attempted to handle this problem, but it probably could be solved in a backward-compatible way by blocking any operations on such a thread until it returns from the INKS trap to its "home" task.

In the future, to provide a server the ability to change the state of *any* thread running in its own space, even if it is a "visiting" thread, alternate versions of the thread control operations could be introduced to which the task control port is specified as a parameter as well as the thread control port. These calls would then operate on a thread *only* while it is running in the specified task. They would block if the thread has an execution context in the specified task but is temporarily executing in another one, or fail if the thread has no execution context in that task.

- It may seem at first that, in a migrating thread model, it would be difficult to implement things like interruptible system calls and Unix signals. In fact, however, the interruptible call problem is mostly unrelated to the thread model and must be dealt with specially in either case. When an RPC client, such as a Unix program, blocks in a `mach_msg` call, technically another thread in the client has the option of forcibly aborting that `mach_msg` call by sending a command to the thread's control port. In a migrating thread model, this option does not exist, which is what intuitively makes it seem like a problem exists with this model. However, in actual practice, it is a bad idea anyway to "blindly" abort a `mach_msg` call, as the server would still continue to process the request, eventually sending back an "orphan" reply message, and undesirable complications could easily result. To correctly implement interruptible system calls and signals, the server *must* be involved—it must provide specific mechanisms for detecting interruption requests and aborting operations when possible. It is just as easy for a server thread, recognizing an "interrupt" request from the client, to respond by returning the thread to the client through a migrating thread mechanism, as to respond by sending a reply message to an RPC client.

- If more than one in-kernel server is to be supported at once, there is the possibility that one server, running on an INKS trap, will call the other through a nested INKS trap. It is even possible that a set of servers could be mutually recursive, making it necessary to support an arbitrary number of "levels" of traps, each with its own user stack. Because this is not a problem in the case of the OSF/1 server, we do not yet deal with it. However, multi-server systems stand to benefit the most from INKS, so this is an important issue.

- There is a thread analog to the "task identity" problem discussed in the INKS paper. Just as server code may make the assumption that `mach_task_self` refers to its own task rather than the user task, it may also assume that `mach_thread_self` refers to one of its own threads and not a client's thread. An example of this in the OSF/1 server is the `exit` system call which attempted to terminate the currently active user task and threads, one of which is running the exit code.

- Twice the amount of virtual memory in the server as usual is used for stacks, because *two* sets of stacks must be maintained: one for the threads receiving messages, and the other for the use of INKS traps. However, in practice this is not a major problem, because INKS trap stacks are used for almost all system calls. This means "real" threads are used only rarely, allowing most of their stacks to remain paged out indefinitely.

- Cthreads, the lightweight user-level thread package employed by the single server, maintains a special data structure at the bottom of every user-level stack which it may reference at any time during execution of server code. Therefore, the stacks used for INKS trap execution cannot be produced by simply allocating blocks of memory. Instead, the server must create and supply to the kernel, extra "pseudo-Cthreads" with correctly initialized stacks.

  In addition, some of Cthreads' automatic thread management operations would cause serious problems if they were allowed on these pseudo-Cthreads. For example, when OSF/1 server code blocks (i.e. `sleeps`), it uses Cthread primitives which normally will schedule another Cthread on the current kernel thread rather than actually blocking the kernel thread and switching to another. This would not work, as the kernel thread being used "belongs" to the client task and must be returned to the client immediately when the RPC work function is complete. We solved this problem by marking every pseudo-Cthread "wired" to its kernel thread.

- Determining how many stacks should be allocated for INKS traps can be tricky, and it may be necessary to "grow" and "shrink" the stack pool dynamically to adjust for system load and other conditions. This problem is basically equivalent to the problems already handled by the server, in maintaining its thread pool. However, it does introduce additional complications and potential for performance problems if not handled well. We currently do not handle it well. We have a fixed-sized pool of stacks allocated at server initialization that is large enough to handle the maximum number of concurrent system calls in any of the benchmarks we ran. The stacks are recorded in a list from which the kernel selects the first available, using a linear scan.

- If this INKS mechanism is to be applied to a server that uses only a single service thread instead of a thread pool, it must be ensured that that service thread and an INKS trap do not execute simultaneously. This could be handled in several ways. For small servers, the easiest way might be to use a `mutex` lock in the server, which is acquired on entry to every server work function and released on return. A more transparent method would be for the kernel to check the server port's queue of threads waiting for a message, and if the service thread is on it, remove it for the duration of the INKS trap. If the thread queue is empty, the kernel would have to abort the INKS trap with a MACH_SEND_INTERRUPTED error, forcing the client to queue a message instead. Since we know of no single-threaded servers worth running in the kernel, we currently do not deal with this problem.

In general, there may also be other unanticipated problems from violating Mach semantics by allowing allowing threads to migrate to other tasks. For example, does our revectoring of the thread's task pointer cause problems we have not yet seen, or not yet recognized? There are probably a few surprises yet in store.

Many of the complexities and shortcomings cited could be avoided by altogether eliminating the notion of a separate server task. By viewing the server code as down-loadable kernel code running as part of the "kernel task" we no longer have to worry about thread migration or service threads. User threads trapping into the server are just trapping into an extended kernel, and the user thread is just "running in kernel mode". The semantics of thread operations applied to user threads running in the kernel are well-defined, so threads running in server code introduce no new problems. Coupling this with short-circuiting server/kernel calls, we have essentially re-created a monolithic kernel.

There are drawbacks, however. One is the kernel/server stack issue. User threads entering the kernel will still need to switch to a service stack or "prepare" the kernel stack so it appears as the server expects. For the latter, the size and non-pageable nature of kernel stacks would again be a concern. In either case, the kernel would have to be modified to avoid using the continuation and stack handoff optimizations, or else those optimizations would have to be modified extensively. This approach also does not address the preemptability problem. Since server threads would now be true kernel threads, they would no longer be preemptible, leading to all the attendant latency problems.

Thread-Switching

In the alternate model, actual server threads could be used for the execution of server work functions invoked by INKS traps. One possible implementation of this model is described here.

When an INKS trap is caught and the destination port and server are found, the kernel would first "steal" a thread from the port's list of server threads waiting to receive messages (the `ip_blocked` field of the `ipc_port` structure). It would fetch the thread's user stack pointer and push onto that stack the stolen thread's continuation state (i.e., the message receive parameters). Then it would handoff to the stolen thread, leaving an appropriate continuation state in the original thread. Finally, the stolen thread would be used to call the server work function on the server's user stack. After the work function returns, the process is reversed: the stolen thread switches back to the original client thread, the stolen thread's continuation parameters are popped from its stack and restored, the thread is returned to the waiting queue on the `ipc_port`, and the client's thread exits from the trap.

This implementation has several advantages over the "migrating threads" approach:

- Since server threads already contain user stacks appropriate for executing server RPC work functions, very little special stack-handling code would be needed. In particular, it is not necessary for servers to explicitly allocate a pool of stacks for the use of INKS traps.

- Since the client's thread is not used for execution of server code, there would be no security problems from external parties attempting to manipulate client threads while they are running in the server.
- Single-threaded servers would be handled implicitly, because INKS trap stacks would be allocated from the same "resource pool" (the waiting-thread queue on the server's receive port).

There are some disadvantages to this approach, however:

- The Cthreads library can no longer know exactly how many threads are waiting on the server's request port at a particular time, because the kernel may steal threads without notice. This might cause serious performance problems related to Cthreads' normal handling of the thread pool.
- If *all* of the server's threads are removed by the kernel for INKS trap execution at a given time, the server becomes unable to process any additional requests until one of the traps returns. In the case of multiple in-kernel servers that call each other, this problem could even result in deadlock. The kernel could be instructed never to steal the last thread (i.e., abort with MACH_SEND_INTERRUPTED if there is only one thread on the port's waiting queue). However, this policy would have to be disabled for single-threaded servers; otherwise no INKS traps would ever be made to the server.
- In general, this approach is likely to be slower, since switching from one thread to another entails a significant amount of processing overhead associated with handling the kernel stack and other necessary operations.

Conclusion

In summary, a thread switching model would have resulted in a cleaner, safer, more robust implementation and would have fit better with Mach's current thread semantics. It would probably be slower than a migrating threads implementation, but perhaps not significantly overall, especially for a single server system. Therefore, for practical purposes, we believe an implementation of INKS based on the thread switching model would be highly valuable in the short term.

In the long term, however, we believe that thread-switching is only a temporary way to work around a problem deeply woven into Mach. If a general-purpose migrating threads model such as that exploited in LRPC[1] were to be introduced in the future, this approach to in-kernel server traps would become both faster *and* cleaner than thread switching. Our INKS work suggests such a model may be feasible, and we expect to explore it.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[2] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeff Law. In-kernel servers on Mach 3.0: Implementation and performance. In *Proc. of the Third Usenix Mach Symposium*, Santa Fe, NM, April 1993.