Evolving Mach 3.0 to a Migrating Thread Model

Bryan Ford
Jay Lepreau

Center for Software Science
Department of Computer Science
University of Utah

{baford, lepreau}@cs.utah.edu
801-581-{4280,4285}

January 17, 1994
Terminology

Thread: A sequential flow of control.

RPC: A control transfer modeling a procedure call, that crosses the protection barrier between two processes.

Static thread model: RPC is implemented by two separate threads, each confined to one process, passing messages.

Migrating thread model: RPC is implemented by one thread temporarily crossing the protection barrier and “doing its own work” in the server’s code.
RPC is *supposed* to model a procedure call.

So, what’s all the gunk on the left?!
Research Overview

Questions: Is the migrating thread model a good idea? Does it make much difference?

Project: Design and implement a version of Mach based on migrating threads; compare the two versions.

Result: Migrating threads are better in speed, code simplicity, and functionality; do not sacrifice protection or backward compatibility.
Primary Issues

- What is the execution environment of a client thread in a server?

- How can clients and servers be protected from each other?

- How can good controllability be provided?
Key Design Element: Split Up the Thread

• “Thread” (migrating part)
  * Logical flow of control
  * Schedulable entity (priority, policy, time quantum)
  * Resource accounting statistics

• “Activation” (static part)
  * Execution context (PC, registers, stack)
  * Exported point of control (Mach “control port”)

Threads and Activations

![Diagram showing relationships between tasks and activations]

- **Task** (e.g. Unix app)
- **Task** (e.g. file system)
- **Task** (e.g. device driver)

- Activation
- Migration
- Kernel
Threads and Activations

![Diagram of threads and activations](image-url)
And the winner is...

Migrating threads provide better:

• Performance

• Simplicity of kernel code

• Functionality
Performance Comparison

- Static RPC
- Migrating RPC

- Null RPC
- 32 bytes in
- 1K in

HP700 cycles

5000
4000
3000
2000
1000
0
Why is it faster?

- Upcall semantics on server side
- Kernel-visible RPC (no “reply ports”)
- No scheduler involvement
Simpler

**RPC paths:**
Approx 2000 lines of code replaced by 400.

**Control mechanisms:**
Approx 3000 lines of code replaced by 1700.
Macrobenchmark Results

- “make” of gas: 2% faster

- link of HP-UX linker: 14% faster

*Note*: without additional leverage of migrating RPC yet.
What made the RPC path simpler?

Answer: the same things that made it faster!

- Reversed call semantics on server side:
  146 instructions down to 33

- Kernel-visible RPC:
  206 instructions down to 12

- No scheduler involvement:
  408 instructions down to 30
More Functional

- Inherited thread attributes:
  - Priority (real-time)
  - Resource limits and accounting

- RPC call chain provides client-server communication channel
Related Work

- Amoeba: exports RPC to user
- Alpha: real time, remote migration
- LRPC on Taos
- Spring: confronted Unix issues
- “Passive Object” Systems: Emerald, Clouds, Psyche
LRPC Comparison

- LRPC addressed data transfer also
- We attempted to isolate control transfer
- We examined simplicity
- We fully addressed controllability issues (Taos has weak protection semantics)
Issues/Future Work

- Leverage to further optimize Mach RPC
- Use improved functionality to simplify servers
- Leverage for real-time support
- Extend to cross-node RPC
- Show another OS can be evolved
Conclusions

• A migrating thread model is faster, simpler, and more powerful.

• Exported RPC abstraction a prerequisite; synchronous invocation beats async.

• Existing systems can be adapted without sacrificing backward compatibility.
Why We Don’t Use Spring’s Terminology

• High-level concept of “thread” fits the migrating entity better

• Encourages us to think in the new model

• Precedent: Alpha, LRPC

• Momentum