Systems@ETH Zürich

Week 6:
Microkernels and fast local IPC
(263-3800-00L)

Timothy Roscoe

Herbstsemester 2014

http://www.systems.ethz.ch/courses/fall2014/aos

Overview

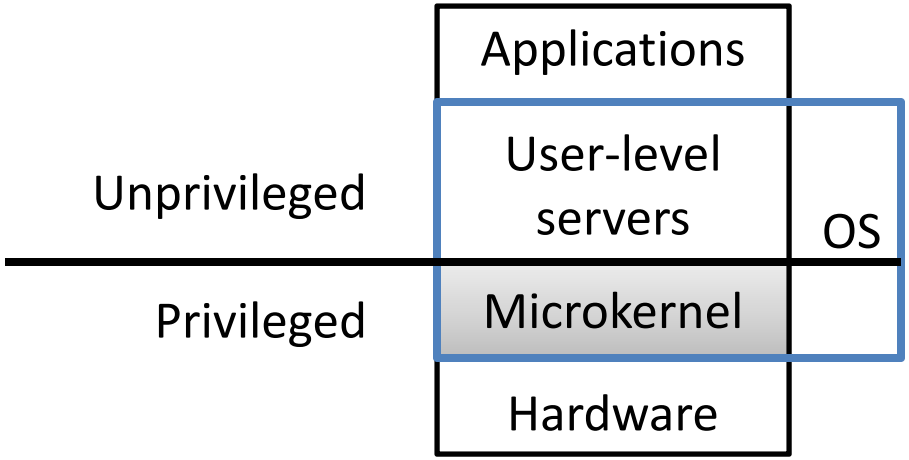
Systems@ETH Zürich

- The microkernel idea
  - Mach (and others)
- The great microkernel debate
  - Bershad and Chen vs. Liedtke
- The design of L3 and L4
  - Performance and size are everything
- Lightweight RPC (LRPC)
  - Making interprocess calls fast
- L4 RPC
  - Making interprocess calls even faster

Approaches to tackling OS complexity

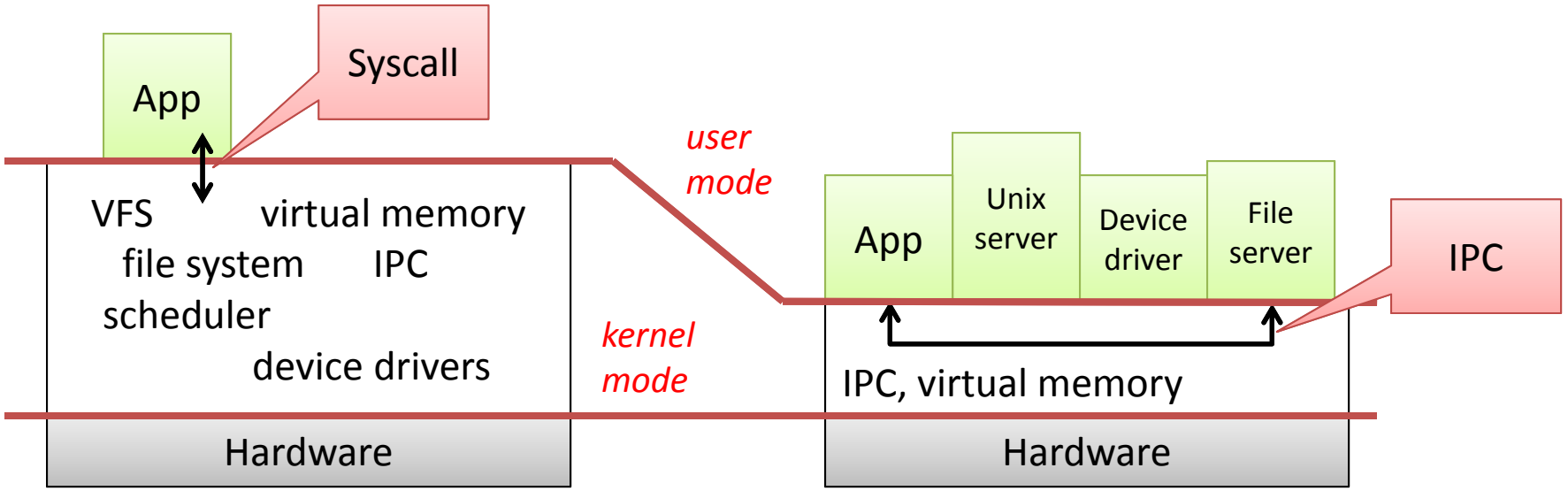
- Classical software engineering approach: modularity
  - Relatively small, self-contained components
  - Well-defined interfaces
  - Enforcement of interfaces
  - Containment of faults
- Doesn't work with monolithic kernels
  - All kernel code executes in privileged mode
  - Faults aren't contained
  - Interfaces cannot be enforced
  - Performance takes priority over structure

Microkernel



Based on ideas of the “Nucleus” [Brinch Hansen, 1970].

Monolithic vs. microkernel OS structure



- Monolithic OS
  - lots of privileged code
  - services invoked by syscall
- Microkernel OS:
  - little privileged code
  - services invoked by IPC
  - “horizontal” structure

Microkernel OS

Systems@ETH Zürich

Kernel:

- Contains code which must run in privileged mode
- Isolates hardware dependence from higher levels
- Small and fast extensible system
- Provides **mechanisms**

User-level servers:

- Are hardware independent/portable
- Provide the “OS environment/personality”
- May be invoked:
  - From application (via message-passing IPC)
  - From kernel (via upcalls)
- Implement **policies**

Popular example: Mach

- Developed at CMU by Rashid and others from 1984 [Rashid et al., 1988]

Goals:

- **Tailorability**: support different OS interfaces
- **Portability**: almost all code H/Windependent
- **Real-time** capability
- **Multiprocessor** and **distribution** support
- **Security**
- Coined term **microkernel**

Basic features of Mach kernel

- Task and thread management
- Inter-process communication
  - asynchronous message-passing
- Memory object management
- System call redirection
- Device support
- Multiprocessor support

Mach = µkernel?

Systems@ETH Zürich

- Most OS services implemented at user level
  - Using memory objects and external pagers
  - Provides mechanisms, not policies
- Mostly hardware independent
- Big!
  - 140 system calls (300 in later versions), >100 kLOC
    - Unix 6th edition had 48 system calls, 10kLOC without drivers
  - 200 KiB text size (350 KiB in later versions)
- Poor performance
  - Tendency to move features into kernel

Overview

Systems@ETH Zürich

- The microkernel idea
  - Mach (and others)
- The great microkernel debate
  - Bershad and Chen vs. Liedtke
- The design of L3 and L4
  - Performance and size are everything
- Lightweight RPC (LRPC)
  - Making interprocess calls fast
- L4 RPC
  - Making interprocess calls even faster

Critique of microkernel architectures

*"Personally, I'm **not** interested in making device drivers look like user-level. They aren't, they shouldn't be, and microkernels are just stupid."*

-- Linus Torvalds

Microkernel performance

- First generation microkernel systems (’80s, early ’90s)
  - Exhibited poor performance when compared to monolithic UNIX implementations
  - Particularly Mach, the best-known example
- Typical results:
  - Move OS services back into the kernel for performance
  - Move complete OS personalities into kernel
    - Chorus Unix
    - Mac OS X (Darwin): complete BSD kernel linked to Mach
    - OSF/1
- Some spectacular failures
  - IBM Workplace OS
  - GNU Hurd

Microkernel performance

Systems@ETH Zürich

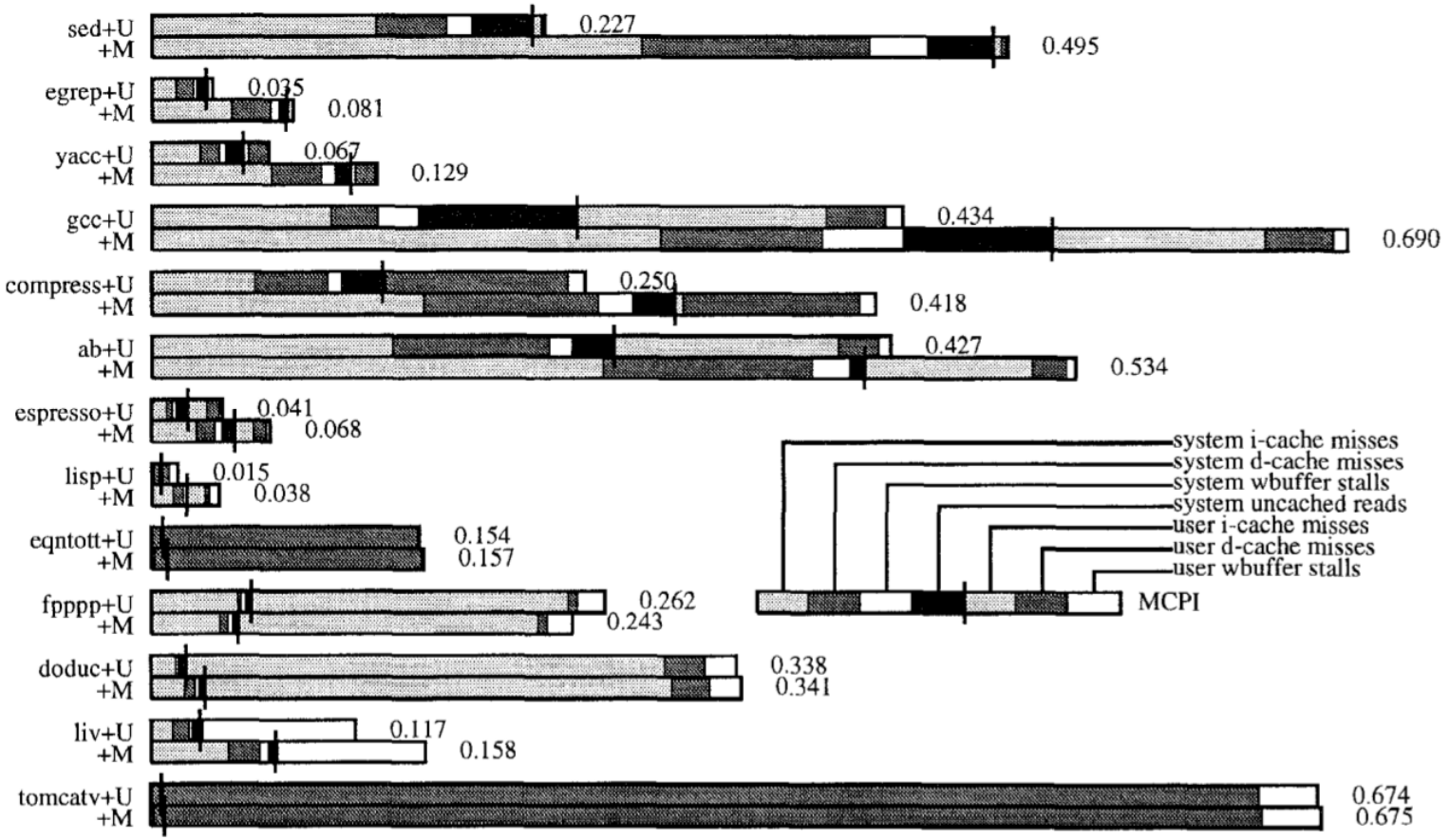
Reasons investigated [Chen and Bershad, 1993]:

- Instrumented user & system code to collect execution traces
- Run on DECstation 5000/200 (25MHz MIPS R3000)
- Run under Ultrix and Mach with Unix server
- Traces fed to memory system simulator
- Analysed memory cycles per instruction:

$$MCPI = \frac{\text{stall cycles due to memory system}}{\text{instructions retired}}$$

- Baseline MCPI (i.e. excluding idle loops)

Ultrix vs. Mach+Unix MCPI



Interpretation

Systems@ETH Zürich

Observations:

- Mach memory penalty higher
  - i.e. cache misses or write stalls
- Mach VMsystem executes more instructions than Ultrix
  - but is portable and has more functionality

Claim:

- Degraded performance is result of OS structure
- IPC cost is not a major factor:

"...the overhead of Mach's IPC, in terms of instructions executed, is responsible for a small portion of overall system overhead. This suggests that microkernel optimizations focusing exclusively on IPC, without considering other sources of system overhead such as MCPI, will have a limited impact on overall system performance."

Conclusions

- System instruction and data locality is measurably worse than user code
  - Higher cache and TLB miss rates
  - Mach worse than Ultrix
- System execution is more dependent than user on instruction cache behaviour
  - MCPI dominated by system Icache misses
- Competition between user and system code not a problem
  - Few conflicts between user and system cache

"The impact of Mach's microkernel structure on competition is not significant."

Conclusions

- Self-interference, especially on instructions, is a problem for system code
  - Ultrix would benefit more from higher cache associativity (direct-mapped cache was used)
- Block memory operations are responsible for a large component of overall MCPI
  - IO and copying
- Write buffers less effective for system
- Page mapping strategy has significant effect on cache

"The locality of system code and data is inherently poor"

Other experience with µkernel performance

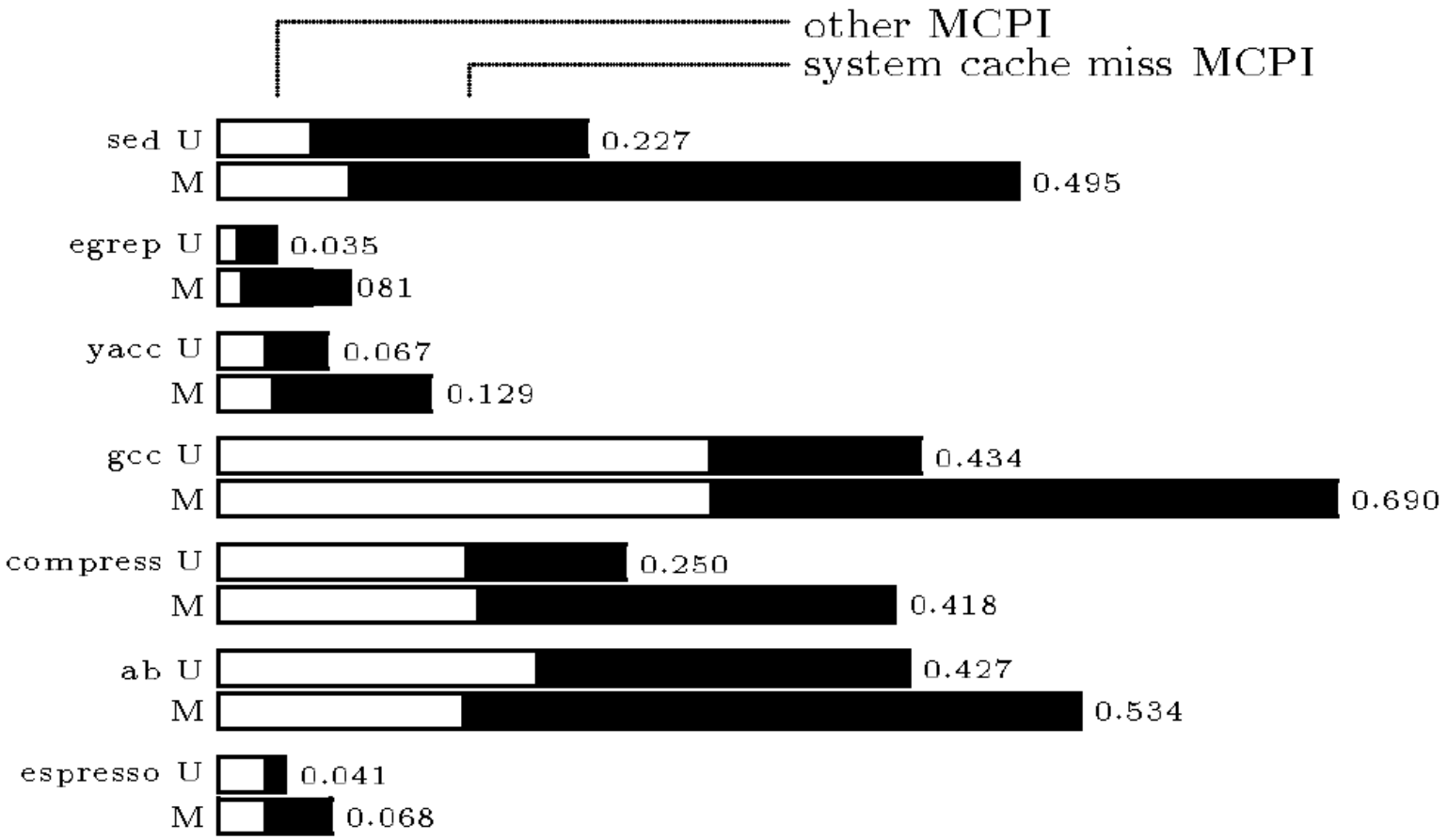
- System call costs are high
- Context switching costs are high
  - Getting worse with increasing CPU/memory speed ratios and lengthening pipelines

⇒ IPC (system call + context switch) expensive

- Microkernels depend heavily on IPC
  - Is the microkernel idea inherently flawed?

A Critique of the critique

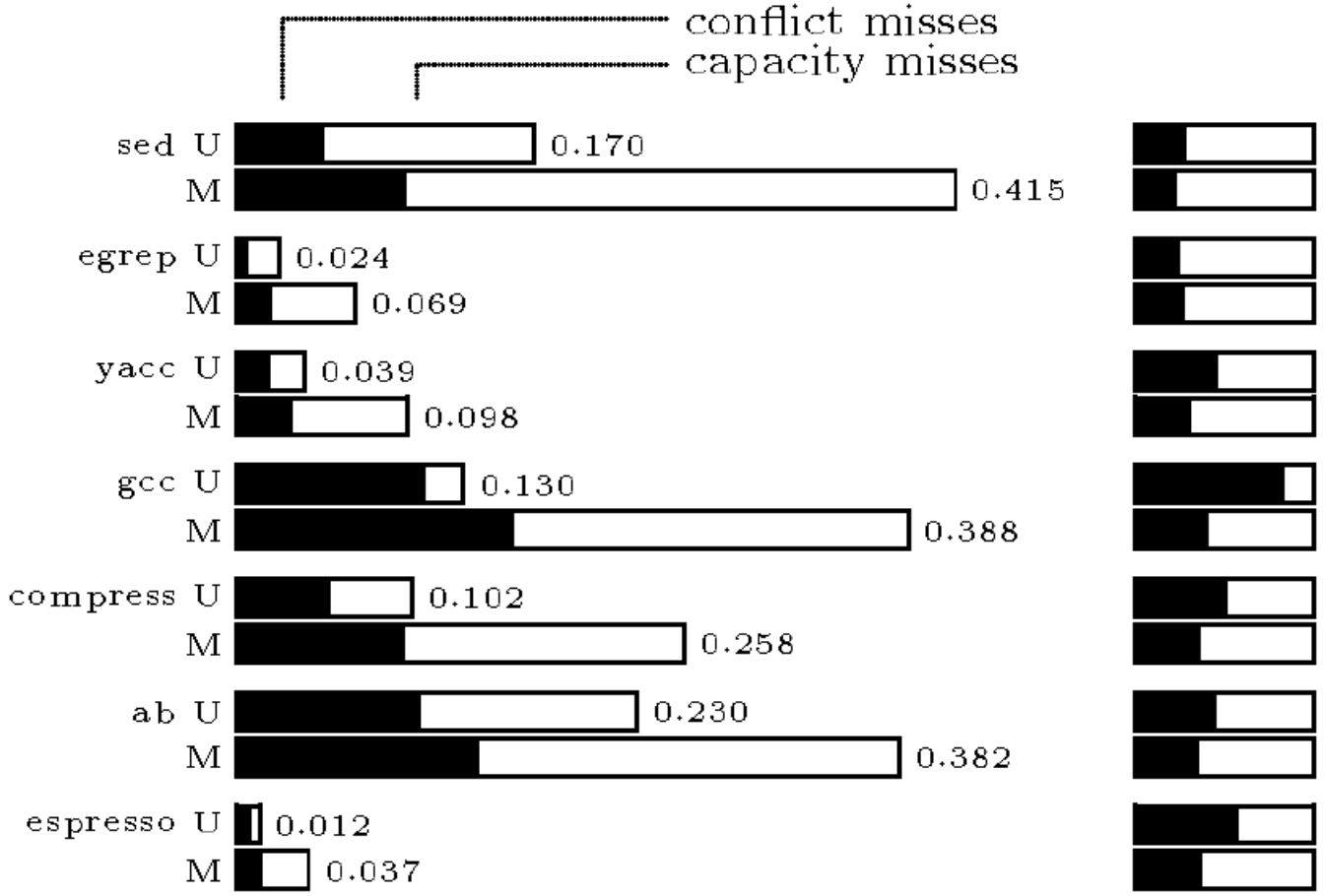
MCPI for Ultrix and Mach:



[Liedtke, 1995]

A Critique of the critique

MCPI caused by cache misses:



[Liedtke, 1995]

Conclusion

Systems@ETH Zürich

- Mach system is too big
  - Kernel + Unix server + emulation library
- Unix server is essentially the same as Unix
- Emulation library irrelevant [Chen and Bershad, 1993]
- Conclusion:

Mach kernel working set is too big

- Can we build microkernels which avoid these problems?

Overview

Systems@ETH Zürich

- The microkernel idea
  - Mach (and others)
- The great microkernel debate
  - Bershad and Chen vs. Liedtke
- The design of L3 and L4
  - Performance and size are everything
- Lightweight RPC (LRPC)
  - Making interprocess calls fast
- L4 RPC
  - Making interprocess calls even faster

Improving IPC by kernel design

[Liedtke, 1993]

- IPC is the most important operation in a microkernel
- The way to make IPC fast is to design the whole system around it
- Design principle: aim at a concrete performance goal
  - Hardware-dictated costs are 172 cycles (3.5µs) for a 486
  - Aimed at 350 cycles for the implementation
- Applied to the L3 kernel

L3/L4 implementation techniques

- Minimise number of system calls
  - Combined operations: `Call`, `ReplyWait`
  - Complex messages
    - Combines multiple messages into one operation
    - As many arguments as possible in registers
- Copy messages only once
  - via direct mapping, not user → kernel → user
- Fast access to thread control blocks (TCBs)
  - TCBs accessed via VMaddress determined from thread ID
  - Invalid threads caught via a page fault
  - Separate kernel stack for each thread in TCB
  - Avoids extra TLB misses on fast path

L3/L4 implementation techniques

- Lazy scheduling
  - Don’t update scheduling queues until you need to schedule
- Direct process switch to receiver
- Short messages in registers
- Reducing cache and TLB misses
  - Frequently-used TCB data near the beginning (single-byte displacement)
  - Frequently-used TCB data co-located (for cache locality)
  - IPC code and kernel tables in a single page (to reduce TLB pressure and refills)
- Use x86 alias registers (ax = al,ah) to pack arguments
- Avoid jumps and checks on fast path
- and more...

Results (L3)

Systems@ETH Zürich

- A short cross address space IPC (user to user) takes 5.2µs
  - compared to 115µs for Mach
- Code and data together use 592 bytes (7%) of on-chip cache
  - kernel must be small to be fast

On µ-Kernel Construction

[Liedtke, 1995]

What primitives should a microkernel implement?

"...a concept is tolerated inside the µ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality."

- Recursively-constructed address spaces
  - Required for protection
- Threads
  - As execution abstraction
- IPC
  - For communication between threads
- Unique identifiers
  - For addressing threads in IPC

What should a microkernel not provide?

- Memory management
- Page-fault handler
- File system
- Device drivers
- …

Rationale:

few features ⇒ small size ⇒ low cache use ⇒ fast

Non-portability

Systems@ETH Zürich

- Liedtke argues that microkernels must be constructed per-processor and are inherently unportable
- Eg. major changes made between 486 and Pentium:
  - Use of segment registers for small address spaces
  - Different TCB layout due to different cache associativity
    - Changes user-visible bit structure of thread identifiers!

Overview

Systems@ETH Zürich

- The microkernel idea
  - Mach (and others)
- The great microkernel debate
  - Bershad and Chen vs. Liedtke
- The design of L3 and L4
  - Performance and size are everything
- Lightweight RPC (LRPC)
  - Making interprocess calls fast
- L4 RPC
  - Making interprocess calls even faster

Lots of Unix IPC mechanisms

- Pipes
- Signals
- Unix-domain sockets
- POSIX semaphores
- FIFOs (named pipes)
- Shared memory segments
- System V semaphore sets
- POSIX message queues
- System V message queues
- etc.

And many, many more in Windows!

IPC is usually heavyweight

IPC mechanisms in conventional systems tend to combine:

- Notification: (telling the destination process that something has happened)
- Scheduling: (changing the current runnable status of the destination, or source)
- Data transfer: (actually conveying a message payload)

Unix doesn't have a *lightweight* IPC mechanism

IPC in Unix is usually polled

- Blocking `read()`/`recv()` or `select()`/`poll()`
- Signals are the nearest thing to upcalls, but...
  - Dedicated (small) stack
  - Limited number of syscalls available (e.g. semaphores)
  - Calling out with `longjmp()` problematic, to say the least
- Unix lacks a good upcall / event delivery mechanism

The Problem

Systems@ETH Zürich

- How to perform has cross-domain invocations?
- Does the calling domain/process block?
- Is the scheduler involved?
- Is more than one thread involved?
- What happens across physical processors?

Lightweight RPC (LRPC): Basic concepts

- Simple control transfer: client's thread executes in server's domain
- Simple data transfer: shared argument stack, plus registers
- Simple stubs: i.e. highly optimized marshalling
- Design for concurrency: Avoids shared data structures

High overhead of previous efforts

- Stubs copy lots of data (not an issue for the network)
- Message buffers usually copied through the kernel (4 copies!)
- Access validation
- Message transfer (queueing/dequeuing of messages)
- Scheduling: programmer sees thread crossing domains, system actually rendezvous's two threads in different domains
- Context switch (x 2)
- Dispatch: find a receiver thread to interpret message, and either dispatch another thread, or leave another one waiting for more messages

Most messages are short

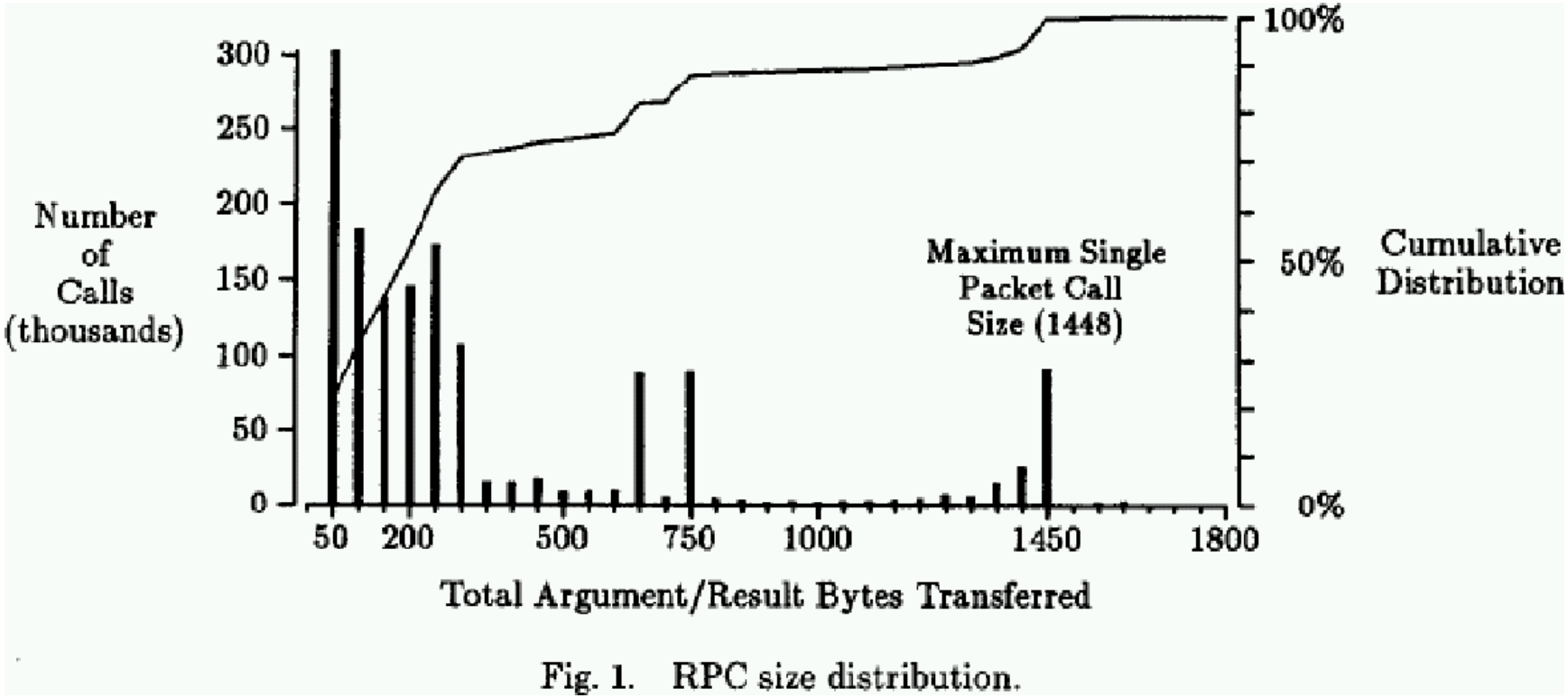


Fig. 1. RPC size distribution.

LRPC Binding: connection setup phase

- *Procedure Descriptors* (PDs) registered with kernel for each procedure in the called interface
- For each PD, *argument stacks* (A-stacks) are preallocated and mapped read/write in both domains
- Kernel preallocates linkage records for return from A-stacks
- Returns A-stack list to client as (unforgeable) Binding Object

Calling sequence (all on client thread)

1. Verify Binding Object, find correct PD
2. Verify A-Stack, find corresponding linkage
3. Ensure no other thread using that A-stack/linkage pair
4. Put caller's return addr and stack pointer in linkage
5. Push linkage on to thread control block's stack (for nested calls)
6. Find an execution stack (E-stack) in server's domain
7. Update thread's SP to run off E-stack
8. Perform address space switch to server domain
9. Upcall server's stub at address given in PD

LRPC discussion

Systems@ETH Zürich

- Main kernel housekeeping task is allocating A-stacks and E-stacks
- Shared A-stacks reduce copying of data while still safe
- Stubs incorporated other optimizations (see paper)
- Address space switch is most of the overhead (no TLB tags)
- For multiprocessors:
  - Check for processor idling on server domain
  - If so, swap calling and idling threads
    - (note: thread migration was very cheap on the Firefly!)
  - Same trick applies on return path

Overview

Systems@ETH Zürich

- The microkernel idea
  - Mach (and others)
- The great microkernel debate
  - Bershad and Chen vs. Liedtke
- The design of L3 and L4
  - Performance and size are everything
- Lightweight RPC (LRPC)
  - Making interprocess calls fast
- L4 RPC
  - Making interprocess calls even faster

L4 synchronous RPC

- L4 pushed this idea further (for uniprocessor case)
- No kernel-allocated A-stack: server must have waiting thread (no upcalls possible)
- RPC just exchanges register contents with calling thread
- *Synchronous RPC:* calling thread blocks, waits for reply
- Scheduler bypassed completely
  - The infamous “null RPC” microbenchmark
  - Latency of a single call, nothing else happening
- Design couples notification, transfer, scheduling

IPC overview

Systems@ETH Zürich

- L4 provides a single system call for all IPC
  - Synchronous and unbuffered *(apart from async notify)*
  - Has a send and a receive component
  - Either send or receive may be omitted
- Receive may specify:
  - A specific thread ID from which to receive (“closed receive”)
  - Willingness to receive from any thread (“open wait”)

Logical IPC operations

- **`Send`** sends a message to a specific thread
- **`Receive`** “closed” receive from a specific sender
- **`Wait`** “open” receive from any sender
- **`Call`** send to and wait for reply from specific thread
  - Typical client RPC operation
- **`ReplyWait`** send to specific thread, “open” receive
  - Typical server operation

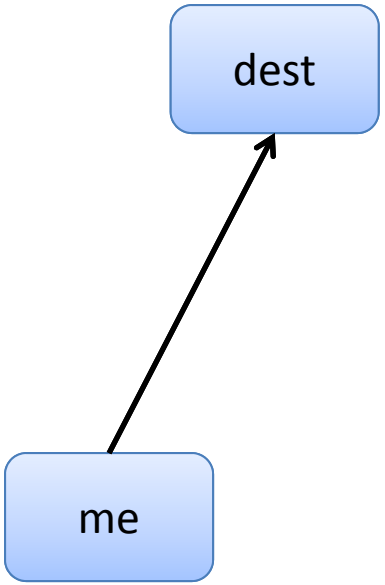
Passing device interrupts to a user-space driver

- Basic idea: an interrupt is a message.
  - Driver registers an IPC endpoint with the kernel
  - 1st-level IRQ handler in kernel
    - Masks the interrupt
    - Creates a message
    - Makes the driver runnable
  - Driver is dispatched
    - Handles interrupt
    - Replies to message to acknowledge the interrupt
  - Kernel unmasks the interrupt

IPC message registers (MRs)

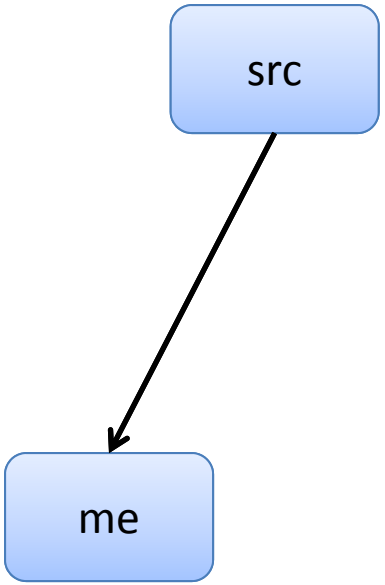
- Virtual registers
  - Not necessarily backed by CPU registers
  - Part of thread state
- On ARM: 6 physical registers, rest in UTCB
- Actual number is a system configuration parameter
  - At least 8, no more than 64
- Contents of MRs form message
  - First MR stores the message tag defining message size etc.
  - Rest are untyped words, not normally interpreted by the kernel
  - Kernel protocols define semantics in some cases
- IPC just copies data from sender's to receiver's MRs

IPC Send



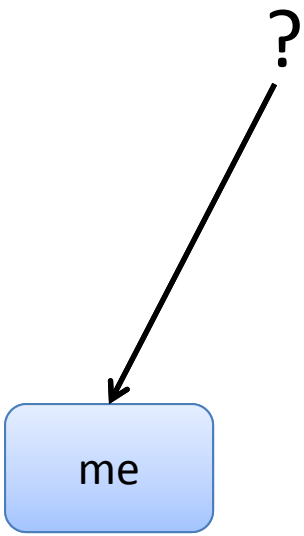
to:	dest
FromSpecifier:	*nil*

IPC Receive (closed)



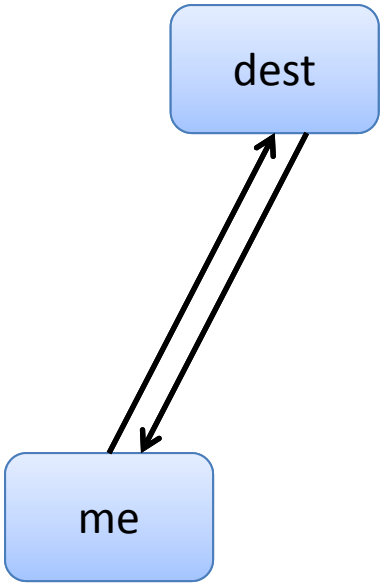
to:	*nil*
FromSpecifier:	src

IPC Wait (open)



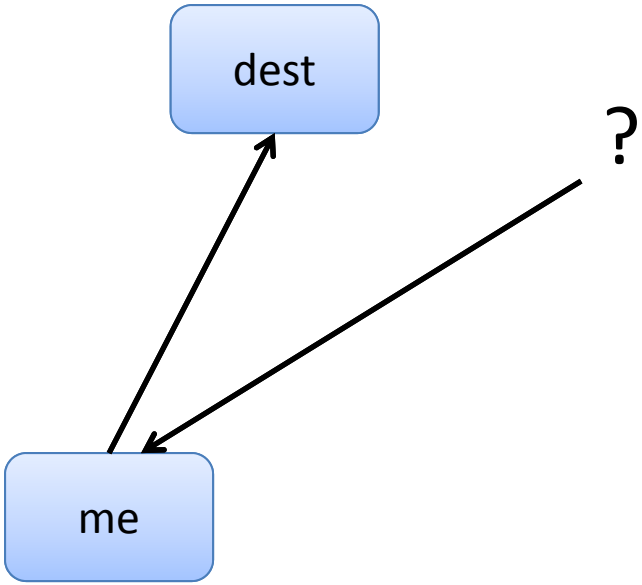
to:	*nil*
FromSpecifier:	*any*

IPC Call



to:	dest
FromSpecifier:	dest

IPC ReplyWait



to:	dest
FromSpecifier:	*any*

Asynchronous Notification

Systems@ETH Zürich

- Very restricted form of asynchronous IPC
  - Delivered without blocking sender
  - Delivered immediately, directly to receiver's UTCB
  - Message consists of a bit mask ORed to the receiver:
    - receiver.NotifyBits |= sender.MR1
  - No effect if receiver's bits are already set
  - Receiver can prevent asynchronous notification by setting a flag in its UTCB

LMP: Barrelfish local RPC

Systems@ETH Zürich

- On a single core:
  - IPC is asynchronous: one-way messaging only
  - RPC implemented at higher level in stubs
- Message is queued at destination, may cause an upcall
  - L4-style fast path: thread can optionally wait for a message
- Unlike L4, can decouple notification & transfer
  - Scheduler is always involved (but . . .)
- Between cores: later in this course…