Improving IPC by Kernel Design & The Performance of Micro-Kernel Based Systems
The IPC Dilemma

IPC is very important in μ-kernel design
  - Increases modularity, flexibility, security and scalability

Past implementations have been inefficient
  - Message transfer takes 50 - 500μs
The L3 (μ-kernel based) OS

A task consists of:

Threads
- Communicate via messages that consist of strings and/or memory objects

Dataspaces
- Memory objects

Address space
- Where dataspaces are mapped
Redesign Principles

IPC performance is the Master!

- All design decisions require a performance discussion
- If something performs poorly, look for new techniques
- Synergetic effects have to be taken into considerations
- Design must cover all levels from architecture to coding
- The design must be made on a concrete basis
- The design must aim at a concrete performance goal
Achievable Performance

A simple scenario
- Thread A sends a null message to thread B
- Minimum of 172 cycles

Will aim at 350 cycles (7 µs)
- Will actually achieve 250 cycles (5 µs)
Redesign Levels

Architectural
- System Calls, Messages, Direct Transfer, Strict Process Orientation, Control Blocks

Algorithmic
- Thread Identifier, Virtual Queues, Timeouts/Wakeups, Lazy Scheduling, Direct Process Switch, Short Messages

Interface
- Unnecessary Copies, Parameter passing

Coding
- Cache Misses, TLB Misses, Segment Registers, General Registers, Jumps and Checks, Process Switch.
Architectural Level

System Calls
- Expensive! So, require as few as possible.
- Implement two calls:
  - Call
  - Reply & Receive Next
- Combines sending an outgoing message with waiting for an incoming message
  - Schedulers can handle replies the same as requests
Messages

Complex Messages:
- Direct String, Indirect Strings (optional)
- Memory Objects
- Used to combine sends if no reply is needed
- Can transfer values directly from sender’s variable to receiver’s variables.
Direct Transfer

Address spaces have a fixed kernel accessible part
- Messages transferred via the kernel part
- User A space -> Kernel -> User B space
- Requires 2 copies.
- Larger Messages lead to higher costs
Shared User Level Memory

Shared User Level memory (LRPC, SRC RPC)
- Security can be penetrated
- Cannot check message’s legality
- Long messages -> address space becoming a critical resource
- Explicit opening of communication channels
- Not application friendly
L3 uses a *Communication Window*

- Only kernel accessible, and exists per address space
- Target region is temporarily mapped there
- Then the message is copied to the communication window and ends up in the correct place in the target address space
Temporary Mapping

Must be fast!

2 level page table need only copy one word
- pdir A -> pdir B

TLB must be clean of entries relating to the use of the communication window by other operations
- One thread
  - TLB is always “window clean”.
- Multiple threads
  - Interrupts – TLB is flushed
  - Thread switch – Invalidate Communication window entries
Strict Process Orientation

One kernel stack per thread, unlike Mach’s use of continuations and a single kernel thread per CPU

May lead to a large number of stacks
  - Minor problem if stacks are objects in virtual memory
Thread Control Blocks (tcb’s)

Hold kernel, hardware, and thread-specific data
Stored in a virtual array in shared kernel space

User area

Kernel area

tcb

Kernel stack
Tcb Benefits

Fast tcb access using thread id
- tcb contains the thread’s kernel stack
- saves 3 TLB misses per IPC

Threads can be locked by unmapping the tcb
- Checking for tcb allocation and residence is moved to the page fault handler
Algorithmic Level

Thread ID’s

- L3 uses a 64 bit unique identifier (uid) containing the thread number
- Tcb address is easily obtained
  - anding the lower 32 bits with a bit mask and adding the tcb base address
Algorithmic Level

Lazy Scheduling
- Only a thread state variable is changed (ready/waiting)
- Deletion from queues happens when queues are parsed
  - Reduces delete operations
  - Reduces insert operations when a thread needs to be inserted that hasn’t been deleted yet!
Algorithmic Level

Short messages via registers
- Register transfers are fast
- 50-80% of messages \( \geq 8 \) bytes
- Up to 8 byte messages can be transferred by registers with a decent performance gain
- May not pay off for other processors
Interface Level

Parameter Passing

- Use registers whenever possible.
  - Far more efficient
  - Give compilers better opportunities to optimize code.
Coding Level

Cache Misses
- Cache line fill sequence should match the usual data access sequence

TLB Misses
- Try and pack in one page:
  - IPC related kernel code
  - Processor internal tables
  - Start/end of Larger tables
  - Most heavily used entries
Coding Level

Registers
- Segment register loading is expensive
- One flat segment covering the complete address space

Jumps and Checks
- Basic code blocks should be arranged so that as few jumps are taken as possible

Process switch
- Save/restore of stack pointer and address space only
- Other registers (floating point etc) saved/restored only when really necessary
L3 IPC Performance vs Mach IPC

Figure 7: 486-DX50, L3 versus Mach IPC Times

Figure 8: 486-DX50, L3 versus Mach IPC Times
But Is That Enough?

What is the impact on overall system performance?

Haertig et al explore performance of
- L4-based Linux
- Mach-based Linux
- native Linux
L^4Linux – micro-kernel based Linux

Full binary compatibility with Linux/X86
No changes to architecture-independent Linux code
No Linux-specific modifications to the L4 kernel
L4Linux – Design & Implementation

Linux implemented as a single Linux server in a µ-kernel task
- µ-kernel tasks used for Linux user processes
- A single L4 thread handles system calls and page faults
- This thread is multiplexed (treated as a virtual CPU)

On booting, the Linux server requests memory from its pager
- L4 maps physical memory into the server’s address space

The Linux server is then the pager for the processes it creates
- L4 converts user-process page faults into an RPC
- sent to the Linux server
- it then maps pages from its address space to user process
Interrupt Handling

Linux top halves run as one server thread per interrupt source
  - L4 converts an interrupt to a message to the appropriate thread

Linux bottom halves all execute in a single high priority thread

Linux interrupt threads are high priority
  - avoids concurrent execution of Linux code on a uniprocessor
System Calls

System calls implemented as IPC
- user process send IPC to the Linux server

Modified libc.so or libc.a avoid trap instructions
- use L4 IPC instead to call the Linux server

User-level exception handler (trampoline) emulates the native system call ‘trap’ instruction for binary compatibility
- L4 redirects trap to emulation library which then used L4 IPC to call the Linux server
Signals

Each user process has a separate signal-handler thread

Linux server delivers a signal by sending a message to the user process’s signal-handler thread

The signal-handler causes the user process’s main thread to save its state and enter Linux by manipulating the main thread’s SP and PC
Scheduling

All thread scheduling is done by the L4 kernel

The Linux server’s schedule() routine is only used for multiplexing the Linux server’s Main thread across concurrent Linux system calls
Experiment

What is the penalty of using L4Linux?
- Compare L4Linux to native Linux

Does micro-kernel performance matter?
- Compare L4Linux to MkLinux

Is co-location critical for good performance?
- Compare L4Linux to an in-kernel version of MkLinux
Microbenchmarks

System call overhead on shortest call “getpid()”

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>223 cycles</td>
</tr>
<tr>
<td>L4Linux</td>
<td>526 cycles</td>
</tr>
<tr>
<td>L4Linux (trampoline)</td>
<td>753 cycles</td>
</tr>
<tr>
<td>MkLinux (in kernel)</td>
<td>2050 cycles</td>
</tr>
<tr>
<td>MkLinux (user)</td>
<td>14710 cycles</td>
</tr>
</tbody>
</table>
Microbenchmarks (cont.)

Performance of specific system calls

- write /dev/null [lat]
- null process [lat]
- simple process [lat]
- /bin/sh process [lat]
- mmap [lat]
- 2-proc context switch [lat]
- 8-proc context switch [lat]
- pipe [lat]
- UDP [lat]
- RPC/UDP [lat]
- TCP [lat]
- RPC/TCP [lat]
- pipe [bw⁻¹]
- TCP [bw⁻¹]
- file reread [bw⁻¹]
- mmap reread [bw⁻¹]
Macrobenchmarks

Time to recompile Linux server

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Percentage Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>476 s</td>
<td></td>
</tr>
<tr>
<td>L4Linux</td>
<td>506 s</td>
<td>(+6.3%)</td>
</tr>
<tr>
<td>L4Linux (trampo)</td>
<td>509 s</td>
<td>(+6.9%)</td>
</tr>
<tr>
<td>MkLinux (kernel)</td>
<td>555 s</td>
<td>(+16.6%)</td>
</tr>
<tr>
<td>MkLinux (user)</td>
<td>605 s</td>
<td>(+27.1%)</td>
</tr>
</tbody>
</table>

Figure 7: *Real time for compiling the Linux Server.* (133 MHz Pentium)
Macrobenchmarks (cont.)

Commercial test suite simulates system under load
Performance Analysis

L4Linux averages 8.3% slower than native Linux
  - Only 6.8% slower at maximum load

MkLinux averages 49% slower than native Linux
  - 60% slower at maximum load

Co-located MkLinux averages 29% slower than native Linux
  - 37% at maximum load
Conclusions

Using the L4 micro-kernel imposes a 5-10% slowdown to native Linux
- Much faster than previous micro-kernels

Is hardware-based protection inherently inefficient?

Have we explored fine grained protection?
Extensibility Performance

A micro-kernel must provide more than just the features of the OS running on top of it.

Specialization – improved implementation of Os functionality

Extensibility – permits implementation of new services that cannot be easily added to a conventional OS.
Pipes and RPC

First five (1) use the standard pipe mechanism of the Linux kernel.
(2) Is asynchronous and uses only L4 IPC primitives. Emulates POSIX standard pipes, without signalling. Added thread for buffering and cross-address-space communication.
(3) Is synchronous and uses blocking IPC without buffering data. (4) Maps pages into the receiver’s address space.
Virtual Memory Operations

The “Fault” operation is an example of extensibility – measures the time to resolve a page fault by a user-defined pager in a separate address space.

“Trap” – Latency between a write operation to a protected page, and the invocation of related exception handler.

“Appel1” – Time to access a random protected page. The fault handler unprotects the page, protects some other page, and resumes.

“Appel2” – Time to access a random protected page where the fault handler only unprotects the page and resumes.

<table>
<thead>
<tr>
<th></th>
<th>L4</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault</td>
<td>6.2 µs</td>
<td>n/a</td>
</tr>
<tr>
<td>Trap</td>
<td>3.4 µs</td>
<td>12 µs</td>
</tr>
<tr>
<td>Appel1</td>
<td>12 µs</td>
<td>55 µs</td>
</tr>
<tr>
<td>Appel2</td>
<td>10 µs</td>
<td>44 µs</td>
</tr>
</tbody>
</table>