Port of the Linux Kernel to the L4 Microkernel
Work in Progress

Slide 1

Michael Hohmuth <*hohmuth@inf.tu-dresden.de*>

September 30th, 1996

Linux port to L4 microkernel — why?

Slide 2

- focus on μK-based real-time systems
- need platform to do development and work on
- port promises fast results
- Linux is portable (i386, Alpha, m68k, ...)
- OSF's Mklinux serves as model

- research target: develop a mk-based multi-server OS which supports apps with QoS requirements

Linux port to L4 microkernel — why interesting?

Slide 3

Unix emulations on μKs usually have bad performance

previous approaches:

1. link server code into μK's address space (Mklinux, SPIN)
2. "lean" servers for providing only limited Unix functionality for special apps (QNX)

our approach:

lean/fast μK, "traditional" Unix server

- usually bad performance because of many address space/ context switches

1. less task switches, RPCs become local calls;
   but less security; only for trusted servers
2. e.g. no virtual memory

The L4 Microkernel

Slide 4

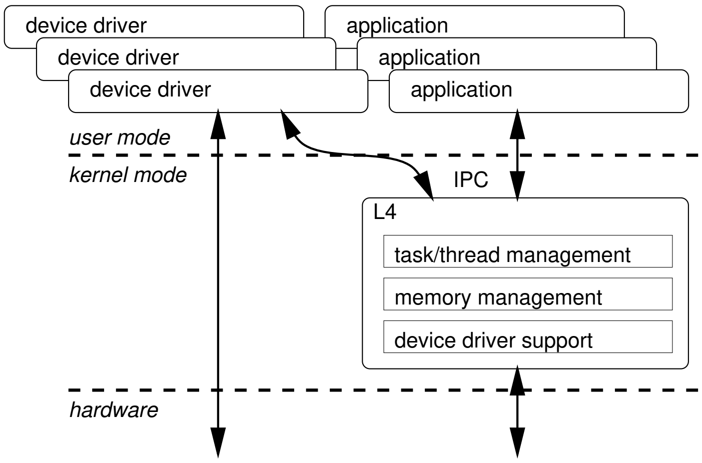
- minimalistic design: implements only basic mechanisms for
  - IPC
  - address space management
  - scheduling
- doesn't implement policies — policies must be implemented on user level
- IPC:
  - powerful: message transfer, scheduling, paging and address space management, I/O and interrupts
  - but very fast: optimized i486/Pentium versions available

- policies are for examples: paging strategies, scheduling strategies, device drivers
- fast IPC:
  - fast kernel entry/exit: CPU needs 100 cycles for entering/leaving kernel mode; L4 needs ca. 50 cycles kernel overhead for a null IPC (Mach: 800)
  - fast address space switches: Pentium version uses segmentation hardware for switching to prevent reloading page table root register
  - small cache working set; fast IPC fath fits entirely into first level cache and is about 1K code.

see Liedtke: *On μ-Kernel Construction,* SOSP 1995.

Slide 5

The L4 Microkernel

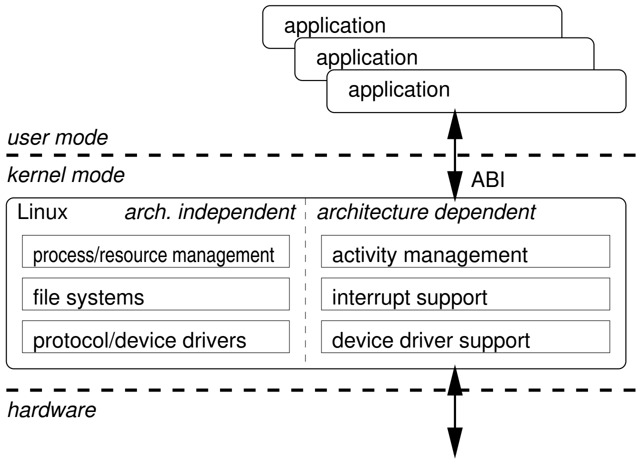


Slide 6

- task/thread management:
  - tasks consist of an address space plus threads
  - direct, synchronous IPC between threads
  - supports external schedulers: preemption = IPC
- memory management:
  - virtual address spaces: pages can be shared between them (flexpages via IPC).
  - supports external pagers: page faults = IPC
- device driver support:
  - I/O address space can be accessed/shared using flexpages (just as memory address space)
  - interrupts = IPC messages

The Linux kernel

Slide 7



- monolithic design
- supports many traditional OS applications: X11, TCP/IP, Emacs, ...
- portable, using architecture-independent layer

ABI binary interface between user processes and kernel

Activity = kernel-internal abstraction of Linux processes = thread + context; we distinguish between

1. user a., context = user address space
2. kernel a., context = user address space, kernel stack, kernel address space

manages/supports

- processor (scheduling = switching between activities)
- memory; Copy-In/Out
- kernel entry/exit
- signal delivery

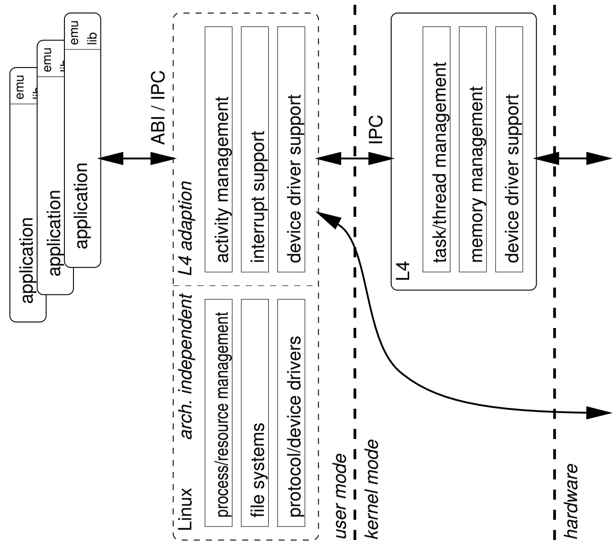
interrupt support arch-independent interface to interrupt hardware

device driver support arch-independent interface to Floppy, DMA, BIOS32

⇒ device drivers are portable

Slide 8

The Linux-Kern on L4



- copy ABI definition from Linux/i386
- copy device driver support (Floppy, DMA), because of identical hardware
- and all device drivers

⇒ activities + interrupts left

- Emulib required to catch traps (Syscall = int 80)
- (Linux dashed because not only one task)

Activity Management

Activity = kernel-internal abstraction of Linux processes = thread + context; we distinguish between

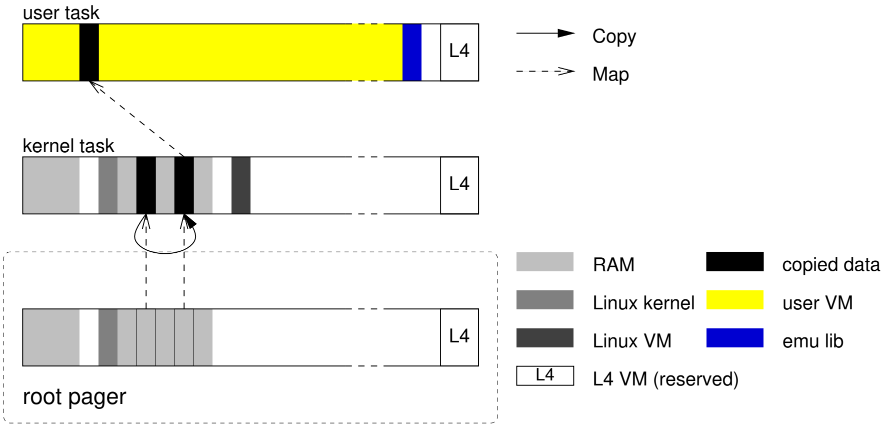
1. user a., context = user address space
2. kernel a., context = user address space, kernel stack, kernel address space

manages/supports

- processor (scheduling = switching between activities)
- memory; Copy-In/Out
- kernel entry/exit
- signal delivery

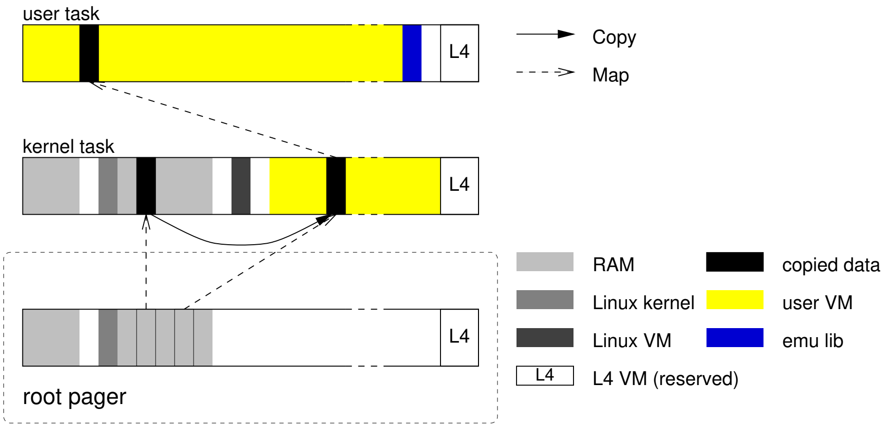
Copy-In/Out in physical frames

Slide 10



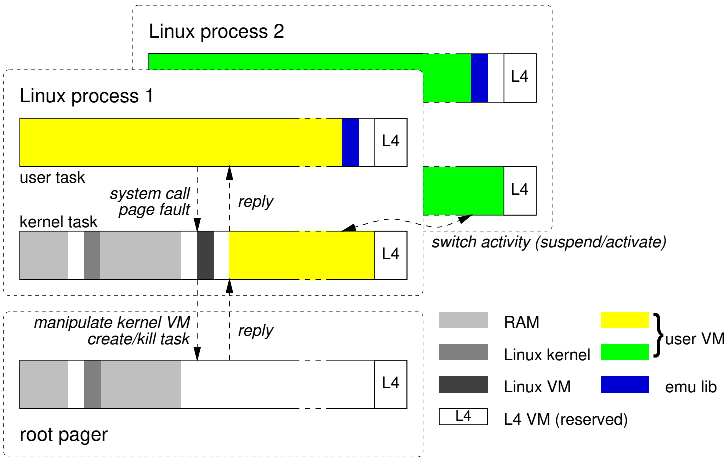
Copy-In/Out using mapping

Slide 11



main architecture

Slide 12



square boxes = tasks with address space

⇒ 2 tasks per Linux process: user task + kernel task

easy Copy-In/Out (user address space always mapped and not cleared when switching)

⇒ additional “root pager” task:

required because kernel uses VM internally, and mapping is defined only between tasks);

(also for task creation so that all tasks are members of the same clan)

⇒ RAM mapped in at 0

because Linux device drivers expect virtual = physical address

⇒ emu library mapped in user task

catches exceptions and notifies kernel via IPC

⇒ suspending and activation of another kernel activity via IPC (in `schedule()`)

Not in this picture:

⇒ interrupt threads in task 0 (Linux: `idle`)

⇒ 2 threads in user task

(emulated process + signal thread)

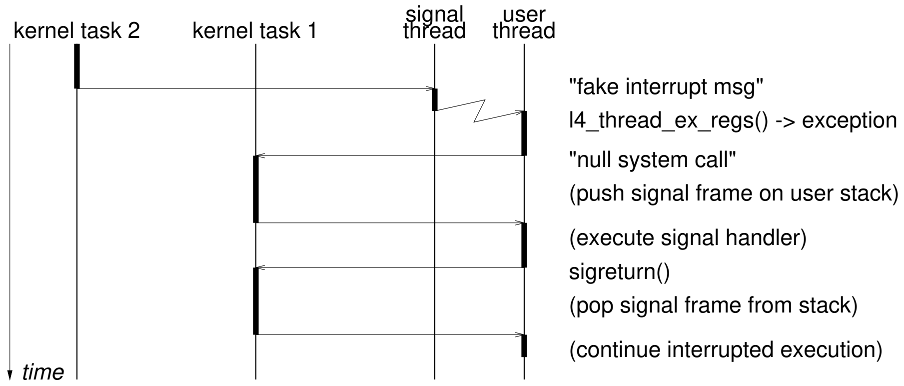
⇒ 2 threads in kernel task

server + pager (activated during copy-in/out and when touching kernel VM)

Signal Delivery

process 2 sends a signal to process 1 which has installed a signal handler

Slide 13



- (see “main architecture”:) P1 doesn’t enter the kernel automatically when a signal is being raised

⇒ need signal thread which takes care of entering the kernel

Interrupt Management

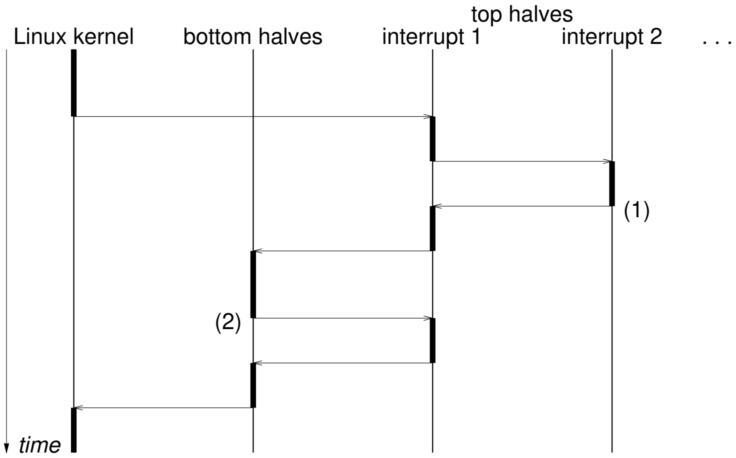
Slide 14

Operations:

- attach/detach of interrupt service routines
  "top half" and "bottom half"
- allow/deny interrupts (`cli()`)
  (also being used to protect critical regions)

interrupt = hardware interrupt, not i386 software interrupt
provides to device drivers an arch-independent interface to the irq hardware

Slide 15



Control flows

- top half — executed immediately after the irq has been raised

  L4 can attach at most one interrupt to a thread ⇒ one thread per interrupt

  interruptible by other top halves

- bottom half — executed only when all executable top halves have been processed; used for irq-triggered but non-time-critical operations

  interruptible by top halves

- no preemption by “normal” kernel code

Synchronization problems

- no “normal” kernel code during top/bottom half execution

  global lock (Linux/SMP) or prios

  ⇒ prios because of lower latency

  problems when switching to SMP; switch to lock then or solve general problems of Linux

- no bottom halves during top half execution

  guaranteed by Linux lock `intr_count`

Bottom halves can be executed in

1. extra thread; lower prio than IRQ threads

2. in IRQ thread (as in René Stange's Linux drivers on L3)

Option 1 yields higher IRQ latency (see (1) and (2)) $\Rightarrow$ extra thread

Implementation

Slide 16

- complete implementation of the design presented

Adaption to L4	12000	line C source code;

  in comparison:

i386 adaption	18000	lines (w/o FPU emulation);
Linux kernel	167000	lines (w/o architecture adaptions);
drivers	343600	lines.

- ABI compatible with Linux/i386 (runs with any standard PC Linux distribution)
- boots multi-user
- important application software runs (X11, networking, compiler etc.)

Performance

- didn't profile or optimize yet
- first clues: "micro benchmarks", comparison with Linux/i386

  ⇒ latencies: larger by factor 1.8 – 19

Benchmark	i386 (ms)	L4 (ms)
start `/bin/sh` process	73	130
system call	.003	.053

  ⇒ throughput: 99% – 44%

Benchmark	i386 (MB/s)	L4 (MB/s)
reread file blocks	22	19
pipe throughput	22	10

⇒ not great but within expected range

⇒ first clues for optimization