The L4 Microkernel

Developed by Jochen Liedtke, GMD (Germany), now IBM T.J. Watson, since early 90's, successor of Eumel (1979) and L3 (1987)

Features:

- 7(!) system calls
- ≈ 5k lines assembler
  ≈ 4k lines C
  ≈ 70k bytes total (MIPS R4k version)
  ≈ 20k lines C of support code (drivers, libc)
- recursive address spaces
- user-level device drivers

Address spaces in L4

Memory management in L4 is greatly simplified by the idea of recursive address spaces

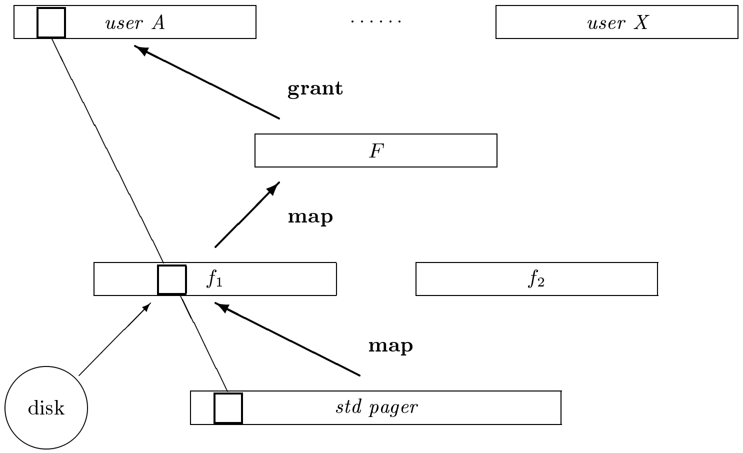
- "magic" initial address space σ_0
  maps physical memory
- sections of σ_0 are mapped by pagers into task address spaces using three mapping primitives:

  Grant: permanently map a page from sender's to receiver's AS
  page is removed from caller

  Map: map a page from sender's to receiver's AS
  caller retains page

  Flush: undoes a Map

Granting



Threads

A thread is the basic active entity (unit of execution and scheduling).

Threads communicate via message-passing IPC.

Each thread has

- a register set (IP, SP, etc)
- an associated address space
- a page fault handler (pager)
  this is a thread which receives page faults (via IPC)
- an exception handler (pager)
  this is a thread which receives exceptions (via IPC)

Each thread belongs to a *task*.

Tasks

A task essentially provides an address space
(plus a *clan boundary*).

An (active) task contains one or more (active) threads.

The number of threads in a task is limited (128 on R4k).
Upon task creation, the full set of (128) threads is created with the task, but all but one are *inactive*
(i.e., they do nothing and consume no resources).

Similarly, upon system initialisation, the full set of tasks (2048 on R4k) is created, but in an *inactive* state.

IPC

Message-passing IPC is the basic communication mechanism between threads.

All IPC is

- synchronous (i.e., blocking), and
- unbuffered

⇒ key to high IPC performance

IPC requires an *agreement* between sender and receiver (i.e., receiver must be expecting IPC, must provide buffers, etc.).

- IPC also used to perform *map* and *grant* operations.
- IPC supports in-line and out-of-line data.

Indefinite blocking can be prevented by *timeouts*.

Interrupts

- Interrupts are converted to IPC message to interrupt handler by L4.

⇒ All interrupt handling is essentially done at user level.
⇒ Device drivers are user-level threads.

- Receiver of interrupt:
  Whoever registered to receive it (first).

Clans and Chiefs

Clans and chiefs are L4's answer to two kinds of protection problems:

- message transfer:
  may want to restrict who can talk to whom and what data can be transferred
- task control:
  who has the right to kill a particular task?

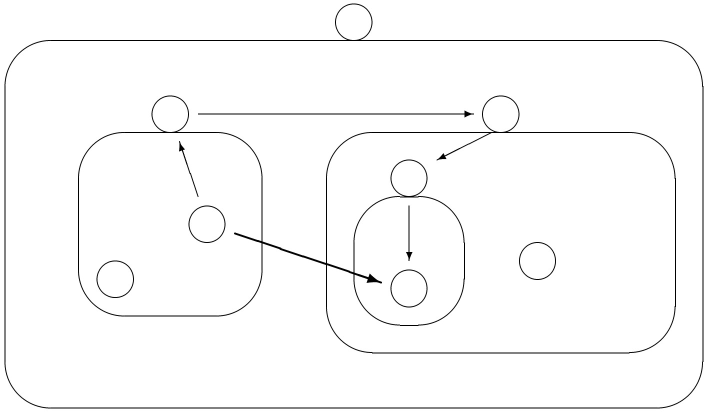
As usual, clans and chiefs present a *mechanism* which allows a large variety of policies to be implemented.

Clans and Chiefs: Basic Idea

There is a *hierarchy* of tasks:

- a task creating another task becomes the *chief* of that task
- the set of tasks created by a chief is that chief's *clan*
- a task can only be killed (directly) if it is inside one's clan or (indirectly) by killing its chief
- intra-clan messages are delivered directly
- inter-clan messages are routed via chiefs
- chief can inspect, modify, suppress messages

Inter-Clan IPC



Clans & Chiefs: Cost

Cost of *supporting* clans is about 4 cycles per IPC (of about 50–250 cylces).

Cost of *using* clans is multiplying number of IPC operations by number of chiefs involved.
⇒ should keep hierarchy flat

There is an implementation limit on the depth of the hierarchy (16 on R4k).

Usages of Clans & Chiefs

network proxies (as in Mach):
master clan for each node receives all inter-node IPC and forwards it

system upgrade:
clan per *system version*,
chiefs translate message format between old and new version

user-based access permissions:
encapsulate tasks in per-user clans
⇒ chiefs can implement arbitrary access protocols

confinement:
encapsulating a task in its own clan
may be used to prevent it from leaking data
or mounting denial-of-service attacks

L4 System Calls

ipc lots of different IPC operations, incl. mappings, timer

ip_nearest ID of rearest IPC routing point (incl. self)

fpage_unmap unmapping memory

thread_switch release CPU (to specific or any thread)

lthread_ex_regs create/delete/modify thread

thread_schedule allows implementation of user-level schedulers

task_new create/delete/donate task

Generally perform lots of different functions depending on parameters.

Use very densly packed parameter structures.

ipc System Call

Several variants (accessible via separate C library entry points):

send: send a message (blocking) to a specific thread

receive: wait for message from specific sender (might be interrupt)
includes sleeping (if specify invalid sender)

call: send and wait for reply (usual "RPC" operation)

reply_and_wait: send message and wait for any new message
typical server operation

wait: wait for message from any thread (incl. interrupt)

send_deceiving: like **send** but substituting sender ID

reply_deceiving_and_wait:
like **reply_and_wait** but substituting sender ID

Exampe: send call (C interface)

int l4_mips_ipc_send (**l4_threadid_t** dest,	*UID of dest. thread*
const void *snd_msg,	*msg descriptor*
l4_ipc_reg_msg_t *snd_reg,	*initial part of message*
l4_timeout_t timeout,	*timeout spec*
l4_msgdope_t *result)	*result code*

Message is divided into two parts:

- initial part (*snd_reg*, 64 bytes on R4k) is passed in registers
- remainder (possibly empty) is passed as a by-value string

Operation will not block longer than indicated by *timeout*
(can be 0 or ∞).

Exampe: receive call (C interface)

int l4_mips_ipc_receive (**l4_threadid_t** src,	*UID of sender thread*
const void *rcv_msg,	*msg descriptor*
l4_ipc_reg_msg_t *rcv_reg,	*initial part of message*
l4_timeout_t timeout,	*timeout spec*
l4_msgdope_t *result)	*result code*

Will only accept message from specified sender.

result contains result code and description of received message (i.e. in-line vs. out-of-line data).

L4 IPC Messages

L4 IPC operations take **two kinds of message parameters**:

1. *snd_reg* or *rcv_reg*: in-register ("short") part of message (first 8 words on MIPS R4k)
2. *snd_msg* or *rcv_msg*: in-memory ("long") part of message

Messages consist of **3 kinds of data**:

1. by-value *in-line data* (directly in registers or message buffer)
2. by-value "*string*" data (message buffer contains pointer to data)
3. by-reference "*fpages*" (mapped by IPC into the receiver's address space)

Register message format

Registers (**s0** ... **s7** on MIPS R4k) contain:

- some (possibly zero) 2-word *fpage descriptors*,
- followed by data

fpage descriptors	*by-value data*

- presence of fpages is indicated by the *m*-bit in *snd_msg* or *rcv_msg*.
- fpage processing stops if invalid fpage descriptor found
- remainder (or all) of register data is simply copied

Size is specified in `l4/ipc.h:l4_ipc_reg_msg_t`.

Send message descriptor format

Format of *snd_msg* parameter:

$*snd\ msg/4_{(62)}$	m	d

- ***snd msg/4** — *message descriptor address*
  $= 0$**:** *short message* (no memory data)
  $\neq 0$**:** address of *message descriptor*
- **m** — *mapping bit*
  $= 0$**:** *by-value send operation*
  message string contains only by-value data
  $\neq 0$**:** *by-reference (mapping) send operation*
  beginning of message string contains *fpage descriptors*
  (potentially followed by more by-value data)
- **d** — *deceiting bit* (used for clans & chiefs)

Receive message descriptor format

Format of *msg_rcv* parameter:

$*rcv\ msg/4_{(62)}$	m	o

$m = 0$**:** *by-value receive operation*
message string contains only by-value data

****rec msg**$= 0$:* receive register data only

****rec msg**$\neq 0$:* message descriptor address
first 8 words of message are in registers

$m = 1$**:** *fpage receive operation*
**rcv msg* is not a pointer but contains the fpage describing where to map incoming fpages
registers pass sender's register values (incl. fpage descriptors)

o=0 receive only from sender specified in *dest id*

o=1 receive from any sender (open wait)

Memory message format

message header	fpage descriptors	by-value data	string dopes

- fpages are expected only if
  - the *m*-bit is set in the message descriptor
  - all register data consists of valid fpages
- fpage processing stops if invalid fpage descriptor found
- remainder of data (up to size given in header) is simply copied
- string descriptors ("dopes") immediately follow in-line data in memory

Note: present R4k implementation allows no more than 3 fpages (in particular, no fpage descriptors in memory).

This is not a problem in practice.

Message header format

Data structure `l4/types.h:l4_msghdr_t`

w2	$0_{(32)}$	$words_{(19)}$	$str_{(5)}$	$\tilde{}_{(8)}$
w1	$0_{(32)}$	$words_{(19)}$	$str_{(5)}$	$\tilde{}_{(8)}$
w0	$fpage_{(64)}$			

w0 *receive fpage*: describes how to map any incoming fpages

w1 *message size dope*: specifies the total buffer space available
words: size of buffer **in words** (total for fpages and in-line data)
strings: number of string dopes

w2 *message send dope*: buffer space used on sending i.e., buffer size used (*words*) and string dopes used (*strings*) must be less than or equal specifications of message size dope

Note: specified buffer/message size is *in addition* to registers

String dope format

Data structure `l4/types.h:l4_strdope_t`

w3	$*rcv\ string_{(64)}$
w2	$rcv\ string\ size_{(64)}$
w1	$*snd\ string_{(64)}$
w0	$snd\ string\ size_{(64)}$

w0 size **in bytes** of string to be sent

w1 address of string to be sent

w2 size **in bytes** of buffer for string to be received

w3 address of buffer for string to be received

Flex-pages (fpages)

A flexpage is a contiguous region of virtual address space

- of size 2^s, ($\geq$ hardware page size)
- aligned to 2^s
- containing all the *mapped* pages of that region.

E.g. the fpage of size 2^{64} and starting at address 0 is the whole address space.

Fpages are the basis of *mapping* and *granting* virtual memory.

Fpage format (as used in *rec fpage* field in message header):

$base/4096_{(52)}$	$0_{(3)}$	$s_{(7)}$	~	~

Data structure `l4/types.h:l4_fpage_t`.

Fpage descriptor

An fpage descriptor (as used for send fpages in message) occupies two words:

w1	*snd fpage*$_{(62)}$	*w*	*g*
w0	*snd base*$_{(64)}$		

snd fpage: *flex-page* in above format

***w*:** *write-permission bit*, unset ⇒ fpage will be mapped read-only

***g*:** *grant bit*, set ⇒ fpage will be granted

snd base: "*hot spot*" for mapping
if sender and receiver specify different size fpages

Fpages are typically used by page-fault handlers,
the "hot spot" is then the faulting address.

Fpage mapping

- sender specifies fpage as b, s: fpage $[b \times 2^s, (b+1) \times 2^s]$,
- sender specifies hotspot h,
- receiver specifies fpage as b', s': fpage $[b' \times 2^{s'}, (b'+1) \times 2^{s'}]$.

$s = s'$: mapping is $b \times 2^s \mapsto b' \times 2^s$
hot spot specification is not needed

$s < s'$: mapping is $b \times 2^s \mapsto b'_{[63,s']} h_{[s'-1,s]} 0_{(s)}$
sender's fpage is aligned around hot spot

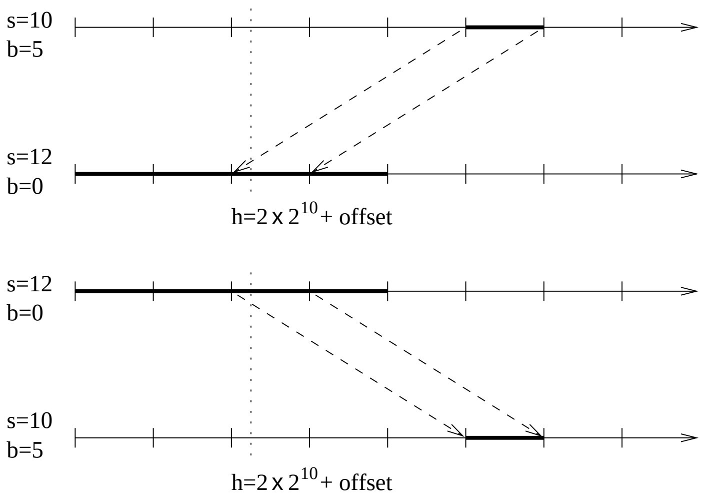
$s > s'$: mapping is $b_{[63,s]} h_{[s-1,s']} 0_{(s')} \mapsto b' \times 2^{s'}$
receiver's fpage is aligned around hot spot

In other words, the smaller fpage is mapped to/from the larger one so that:

- it is aligned according to its size, and
- it contains the hot spot.

Note: The actual value of h is not important, only $h \bmod \max(2^s, 2^{s'})$

Fpage mapping examples



IPC Result Status

The result of the IPC operation is returned in a *message dope*:

$0_{(32)}$	*words*$_{(19)}$	*str*$_{(5)}$	*cc*$_{(8)}$

words: size **in words** of in-line data received (other than in registers)

str: number of strings received
(in buffers pointed to by string dopes in message header supplied to call)

cc: condition code

IPC Result Condition Code

Condition code format:

$ec_{(4)}$	i	r	m	d

ec error code: ec$\neq 0 \Rightarrow$ IPC failed
consult the manual for the meaning of error codes
frequent reason: *Cut message* — receiver's buffer was too small, not enough strings, etc.
Note: this value is also delivered as the return value of the call when using the C library interface

m map bit: $m = 1 \Rightarrow$ fpages were received

Other bits have to do with clans and chiefs, consult the manual for explanation.

IPC Timeout Specifications

An IPC operation specifies 4 timeout values:

$m_{r(8)}$	$m_{s(8)}$	$p_{r(4)}$	$p_{s(4)}$	$e_{s(4)}$	$e_{r(4)}$

$\boldsymbol{m_r, e_r}$**:** receive timeout is $m_r 4^{15-e_r}\,\mu s$

$\boldsymbol{m_s, e_s}$**:** receive timeout is $m_s 4^{15-e_s}\,\mu s$

$\boldsymbol{p_r}$**:** receive page fault timeout is $4^{15-p_r}\,\mu s$

$\boldsymbol{p_s}$**:** send page fault timeout is $4^{15-p_s}\,\mu s$

Zero values of e or p mean ∞, i.e., no timeout
Zero values of m (with $e > 0$ mean 0, i.e., never block
Max values of p ($p = 15$) mean 0, i.e., fail on page fault

- IPC timeouts can be specified in the between 1μs and 19h
- page fault timeouts can be specified between 4μs and 256s

Data structure `l4/types.h:l4_timeout_t, L4_IPC_TIMEOUT()`

[1] Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *Proc. 6th W. Hot Topics Operating Syst. (HotOS)*, pages 73–79, Cape Cod, MA, USA, May 1997. IEEE.

[2] Jochen Liedtke. On μ-kernel construction. In *Proc. 15th ACM Symp. OS Princip.*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[3] Jochen Liedtke. Towards real microkernels. *Comm. ACM*, 39(9):70–77, September 1996.

[4] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual — MIPS R4x00*. School Comp. Science and Engin., University NSW, Sydney 2052, Australia, December 1997. UNSW-CSE-TR-9709. Latest version available from http://www.cse.unsw.edu.au/~disy/.

[5] Alan Au and Gernot Heiser. *L4 User Manual*. School Comp. Science and Engin., University NSW, Sydney 2052, Australia, January 1998. UNSW-CSE-TR-9801. Latest version available from http://www.cse.unsw.edu.au/~disy/.