The L4 Microkernel

- Developed and implemented on i86 by Jochen Liedtke, GMD (Germany) ≈ 1992–95 (“Version 2”).
- Successor of Eumel (1979) and L3 (1987).
- Implementations at Dresden Uni of Technology and UNSW.
The L4 Microkernel

- Developed and implemented on ix86 by Jochen Liedtke, GMD (Germany) \( \approx 1992–95 \) (“Version 2”).
- Successor of Eumel (1979) and L3 (1987).
- Implementations at Dresden Uni of Technology and UNSW.

Features

- 7(!) system calls (Version 2)
- recursive address spaces
- user-level page fault handlers
- user-level device drivers
- user-level scheduling
- real-time capable (sort-of)
L4 Implementations

→ ix86:
  → Liedtke’s kernel, 100% assembler, Versions 1–X
  → Hohmut, Dresden, called *Fiasco*, C++, Version 2
  → Dannowski, L4Ka “hazelnut” kernel, C++, Version X
  → L4Ka Team “pistachio” kernel, C++, portable, Version X.2

→ MIPS R4x00:
  → Elphinstone, UNSW, 64-bit, assembler&C, Version 2
  → Calypso kernel by Chris Szmajda (VM management)
  → UNSW pistachio port (Potts & Winwood), in progress

→ Alpha:
  → Schönberg (Dresden), 96, 64-bit, PALcode&C (rudimentary)
  → Potts, Winwood (UNSW), completed, SMP, Version 2
  → UNSW pistachio port (Potts & Winwood), in progress

→ StrongARM:
  → L4Ka hazelnut kernel

→ PowerPC: pistachio (LeVasser, Karlsruhe), partially complete

→ IA-64: pistachio (Skoglund, Karlsruhe), partially complete

→ SPARC: planned (UNSW)
L4 Implementation Example: MIPS

**History**

- Written by Kevin Elphinstone (UNSW PhD student), 1995–7
- First 64-bit version of L4
- Used in OS research projects at UNSW since 1996
- Used in teaching at UNSW since 1997
- New VM management (multiple page size, shared page tables, new PT structure) by Szmajda (calypso, not yet released)
L4 Implementation Example: MIPS

**HISTORY**

- Written by Kevin Elphinstone (UNSW PhD student), 1995–7
- First 64-bit version of L4
- Used in OS research projects at UNSW since 1996
- Used in teaching at UNSW since 1997
- New VM management (multiple page size, shared page tables, new PT structure) by Szmajda (calypso, not yet released)

**STATISTICS (KERNEL VERSION 79)**

- 6k lines assembler source (*.S)
- 5k lines C source (*.c)
- 1.7k lines C and assembler header files (*.h)
- 80kB kernel text and static data
- 1MB kernel data (mostly TCBs and page tables)
  - kernel footprint could be reduced to $\approx 200kB$
- fast (details later)
Main L4 Abstractions

**threads**: execution abstraction and UIDs

**tasks**: address spaces and resources

**IPC**: message-based communication, incl VM mappings

**flexpages**: VM page abstraction, superpages

**clans and chiefs**: task hierarchy for (arbitrary) security models

**pagers, excepters, preempters, interrupt handlers**: exceptions

- Mostly strict separation of:
  - *mechanisms* (provided by kernel) and
  - policy (implemented by user-level servers).
- Minimality achieved by orthogonality of mechanisms.
L4 Threads

- A thread is the basic active entity (execution and scheduling).
- Threads communicate via message-passing IPC.
- Each thread has
  - a register set (IP, SP, user-visible registers, processor state)
  - an associated task/address space
  - a page fault handler (pager)
    - this is a thread which receives page faults (via IPC)
  - an exception handler (dependent on architecture)
    - this is a thread which receives exceptions (via IPC)
  - preempters (not implemented)
    - thread which receives preemption messages
  - scheduling parameters (priority, time slice)
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 fixed (128 on R4k).
  - The full set of (128) threads is created with the task,
  - all but one are *inactive* (i.e., they do nothing and consume no resources).
  - further threads can be activated via a system call
    (lthread_ex_regs())
  - *Model will change in future versions!*
- Upon system initialisation, the full set of tasks (2048 on R4k) is created, but in an *inactive* state.
- A task has a *chief* (parent/owner).
  - *Chiefs will vanish in future versions!*
IPC

- Message-passing IPC provides communication between threads.
- All IPC is
  - synchronous (i.e., blocking), and
  - unbuffered
  This is key to high IPC performance
- IPC requires an *agreement* between sender and receiver
  (i.e., receiver must be expecting IPC, must provide buffers, etc.).
  - supports in-line and out-of-line *by-value* data.
  - supports *map* and *grant* VM operations for *by-reference* data.
- Blocking can be limited by *timeouts*.

Note the interaction between IPC and threads:
- threads need efficient IPC to talk,
- blocking IPC needs low-cost threads to be efficient.
Flexpages

- Describe virtual memory regions for use in *mapping* operations.
- Generalisation of pages by abstracting over page size.
- Usually called *fpages*.

**Properties**

- size $2^s$, ($\geq$ hardware page size),
- aligned to $2^s$
- kernel should try to map whole fpage as a single super-page
- partially populated fpages: an fpage refers to all mapped pages within the region designated by it.
Address Spaces

Address spaces are recursively constructed from mappings into other address spaces:

- A “magic” initial address space $\sigma_0$ maps physical memory;
- each address space (pager) can map portions of its own address space into another (cooperating) address space.
Address Spaces

Address spaces are recursively constructed from mappings into other address spaces:

- A “magic” initial address space $\sigma_0$ maps physical memory;
- each address space (pager) can map portions of its own address space into another (cooperating) address space.

**Done with three mapping primitives**

**Map:** a page to receiver: sender retains page

**Grant:** a page to receiver: sender loses page

**Flush:** undoes a Map (removes page from receiver)
Clans & Chiefs

Clans & Chiefs are a security mechanism for:

- **task control**: define ownership 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*
  - task can be killed only
    1. directly by its chief
    2. indirectly when its chief is killed

- **communication control**: restrict the flow of messages
  - intra-clan messages are delivered directly
  - inter-clan messages are redirected to chief
  - chief can forward the message transparently

Depth of hierarchy is limited (16 on MIPS).

**Note**: Clans & chiefs will be replaced by a better model!
INTER-CLAN IPC
Page Faults and Pagers

- L4 maintains kernel page tables containing mappings *explicitly established* by user threads (via IPC).
- On a page fault, the kernel invokes the thread’s pager by
  ➀ sending an IPC message to the pager on the faulter’s behalf
  ➁ catching the pager’s reply and continue the faulter
Pager’s is expected to send a mapping for the missing page.

```
client      RPC      pager
  ↓        └─ page fault └─
  ↑                        ↓
               L4
```
Exceptions and Excepters

L4/MIPS EXCEPTION HANDLING

Totally analogous to page faults:

- Each thread has an excepter
- If a thread triggers an exception, the kernel invokes the thread’s excepter by:
  ① sending an IPC message to the pager on the faulter’s behalf,
  ② catching the pager’s reply and continue the faulter.
- The excepter may chose not to reply, leaving the excepting thread blocked forever.
L4/ix86 EXCEPTION HANDLING

Virtualisation of hardware:

• A thread installs its own interrupt vector, using (kernel-emulated) processor features.

The kernel handles some exceptions internally (TLB miss, system call).
Preemptions and Preempters

• Basic idea:
  ➔ treat a preemptions like a page fault, i.e., a time faults,
  ➔ have a preempter as a time-fault handler,
  ➔ preempter is invoked via IPC.

• Model is, at present, not matured, and preemtpers are not implemented in any L4 version.
Interrupts and Interrupt Handlers

- Each hardware interrupt is modelled as a virtual (hardware) thread.
- (At most) one user-level interrupt-handler thread is associated with each hardware interrupt.
- If an interrupt occurs, the kernel generates an IPC from the interrupt thread to the interrupt handler.
- The interrupt handler is in general (part of) a device driver.
  - It will need access to device registers.
  - Done via a special mapping protocol of the root pager.

Kernel handles some interrupts internally (e.g., timer).
Device Interfaces

- Devices are controlled via special *device registers*, typically:
  - *status register*, to obtain device status,
  - *control register*, to send commands to device
    status and control registers are at the same address,
  - *data register(s)* to pass data/command parameters.
- Number of registers is normally small,
  data and parameter buffers are passed in memory, address
  specified in data registers
- Device registers are either *memory mapped* or accessed via *I/O instructions*
- Devices access only *physical memory*, i.e. bypass the MMU.
Device Driver

- Interface between hardware (device controller) and OS.
- Processes OS device requests and controls device by writing to data & command registers.
- Monitors device by reading status & data registers and handling device interrupts.
- Transfers data between OS buffers and device.
L4 device driver

- Runs at user level.
- Has mappings for device registers (MIPS) and physical memory.
- Typically consists of **top half** and **bottom half**.
L4 device driver

- Runs at user level.
- Has mappings for device registers (MIPS) and physical memory.
- Typically consists of top half and bottom half.

**TOP HALF HANDLER** processes device interrupts:

- receive L4 interrupt IPC,
- check success (status register),
- make data available (copying or mapping),
- initiate next request (if one),
- notify/reply to user (IPC).
L4 device driver

- Runs at user level.
- Has mappings for device registers (MIPS) and physical memory.
- Typically consists of **top half** and **bottom half**.

**Top half handler** processes device interrupts:

- receive L4 interrupt IPC,
- check success (status register),
- make data available (copying or mapping),
- initiate next request (if one),
- notify/reply to user (IPC).

**Bottom half handler** processes user requests:

- initiate I/O (set up parameter buffer, write device registers) or queue request if device busy,
- reply to user (if asynchronous)
L4 device driver...

- Top half must be fast (to avoid missing interrupts):
  - runs at high priority,
  - generally runs with interrupts disabled,
  - does minimal amount of work,
  - longer tasks left to to top half
    (copying buffers, replying to user).
- Concurrency control required between top and bottom half.
- Top half must not be blocked by bottom half.
L4-Based OS Design

client

OS

devices

tty

SCSI

pager

except

handler

syscall

handler

sub

system

other OS

L4
OS Structure

- OS server is chief of client processes.
- Client’s “system calls” are library stubs performing RPC to OS server.
- “OS” may consist of many server threads in same or separate tasks.
- OS server may redirect client requests to other servers
  - within OS server task,
  - outside OS server task.
- Clans & chiefs mechanism:
  - prevents direct client access to separate servers tasks,
  - supports easy redirection of “system call” RPC.
The L4 Interface

- The L4 Reference Manual [EHL97] defines the L4 ABI:
  - making maximal use of registers
  - assembler interface
  - very architecture-specific

- `libl4` provides a C API:
  - still somewhat architecture specific (e.g. size of register message)
  - interface is defined in header files in
    `http://~cs9242/include/,$L4/include/l4/`

- Usage is explained in the L4 User Manual [AH98]
L4 System Calls

1. **ipc()**
   - Message passing, combining send and receive
2. **fpage_unmap()**
   - Revoke mappings
3. **id_nearest()**
   - Determination own and target TID
4. **task_new()**
   - Create/delete task/address space
5. **lthread_ex_regs()**
   - Create/manipulate thread
6. **thread_switch()**
   - Explicit time-slice donation
7. **thread_schedule()**
   - Setting/enquiring scheduling parameters
IPC System Call Overview

Variants of syscall accessible via separate C bindings

**CLIENT FUNCTIONS**

- **send()** send a message (blocking) to a specific thread
- **receive()** “closed” receive from specific sender — includes sleeping (if specify invalid sender)
- **wait()** “open” receive from any thread (incl. interrupt)
- **call()** send & wait for reply — *usual “RPC” operation*
- **reply_and_wait()** send & wait for any message — *typical server operation*
- **send_deceiving()** like **send()** but substituting sender ID
- **reply_deceiving_and_wait()** similar
C BINDINGS FOR CHIEFS

• Support transparent forwarding by chief:

→ chief_send()
  identical to send_deceiving

→ chief_wait()
  like wait() but returns intended destination

→ chief_receive()
  like receive() but returns intended destination

→ chief_call()
  like call(); substitutes sender & returns intended destination

→ chief_reply_and_wait()
  like reply_deceiving_and_wait(); returns intended destination
Deceiving IPC

**Clans & Chiefs Mechanism** supported by *deceiving*:

- If "deceit" bit is set by sender, L4 will deliver the message with sender-specified *virtual sender ID*
- The receiver is alerted by the *deceit bit*
- Deceiving only works if *direction preserving*
  - real sender must be along the redirection chain from the virtual sender to the receiver, i.e.:
    - message goes out of clan & virtual sender ID is within clan, or
    - message goes to subclan & virtual sender ID is outside clan.
- Supports transparent inter-clan IPC interception
Example: Send Call

```c
int l4_mips_ipc_send (l4_threadid_t dest,
                       const void *snd_msg,
                       l4_ipc_reg_msg_t *snd_reg,
                       l4_timeout_t timeout,
                       l4_msgdope_t *result)
```

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 ∞.
Example: Receive Call

```c
int l4_mips_ipc_receive (l4_threadid_t src,
                        const void *rcv_msg,
                        l4_ipc_reg_msg_t *rcv_reg,
                        l4_timeout_t timeout,
                        l4_msgdope_t *result)
```

- 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

Two kinds of message parameters:

1. \texttt{snd\_reg} or \texttt{rcv\_reg}: in-register (“short”) part of message (first 8 words on MIPS R4k)
2. \texttt{snd\_msg} or \texttt{rcv\_msg}: in-memory (“long”) part of message
L4 IPC Messages

**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 contain 3 kinds of data:**

① by-value *in-line data* (directly in registers or message buffer)

② by-value “*string*” (out-of-line) data (message buffer contains pointer to data)

③ by-reference “*fpages*” (describing mappings)
Register message format

- Registers (s0 ... s7 on MIPS R4k) contain:
  - some (possibly zero) 2-word *fpage descriptors*,
  - followed by data

<table>
<thead>
<tr>
<th><em>fpage descriptors</em></th>
<th><em>by-value data</em></th>
</tr>
</thead>
</table>

- 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 14/ipc.h:14_ipc_reg_msg_t.
Send message descriptor format

**FORMAT OF snd_msg PARAMETER:**

\[ *\text{snd msg}/4\ (62) \]

\[ m \]

\[ d \]

→ *\text{snd msg}/4 — message descriptor address
  = 0: short message (no memory data)
  ≠ 0: address of message descriptor

→ m — mapping bit
  = 0: by-value send operation (no mappings)
  ≠ 0: by-reference (mapping) send operation
    → beginning of message string contains \textit{fpage descriptors}

→ d — deceiving bit: lie about sender
  turned on automatically by using deceiving send binding.
Receive message descriptor format (MIPS)

FORMAT OF $msg_{\text{rcv}}$ PARAMETER:

\[
*rcv \ msg/4 (62) \quad m \sim
\]

\( \rightarrow m \) — mapping bit

\( m = 0 \):
- \( *\text{rec \ msg} = 0 \): receive register data only (no mappings)
- \( *\text{rec \ msg} \neq 0 \): message descriptor address
  \( \rightarrow \) first 8 words of message are in registers
  \( \rightarrow \) rest as specified in message buffer
  \( \rightarrow \) may accept mappings (if receive fpage option is used)

\( m = 1 \): fpage receive operation
- \( *\text{rcv \ msg} \) is not a pointer
- \( \rightarrow \) contains the fpage describing the mapping window
Memory message format

- Fpages are expected only if
  - the $m$-bit is set in the send message descriptor, and
  - register data starts with a valid fpage descriptor.
  - Fpage processing stops if invalid fpage descriptor found.
  - Fpages are only recognised in register part of message (MIPS)
  - all data is copied to receive (incl. fpage descriptors)
- Out-of-line data ("strings")
  - described by "dopes", immediately after in-line data.
Message format: Logical

snd_msg

msg hdr

fpages in-line

string dopes

string

string

string
Message format: Physical

```
msg hdr in-line part 2
string dopes

snd_msg

registers:  fpages  in-line part 1

string
string
string
```

---

cse/UNSW
## Message header format

**Data structure** `l4/types.h:l4_msghdr_t`

<table>
<thead>
<tr>
<th></th>
<th>(0_{(32)})</th>
<th>(\text{words}_{(19)})</th>
<th>(\text{str}_{(5)})</th>
<th>(\sim_{(8)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(w_2)</td>
<td>(0_{(32)})</td>
<td>(\text{words}_{(19)})</td>
<td>(\text{str}_{(5)})</td>
<td>(\sim_{(8)})</td>
</tr>
<tr>
<td>(w_0)</td>
<td>(0_{(32)})</td>
<td>(\text{words}_{(19)})</td>
<td>(\text{str}_{(5)})</td>
<td>(\sim_{(8)})</td>
</tr>
</tbody>
</table>

\(fpage_{(64)}\)

- **\(w_0\)** *receive fpage*: describes how to map any incoming fpages
- **\(w_1\)** *message size dope*: specifies the total buffer space available
  - \(\text{words}\): size of buffer *in words* (total for fpages and in-line data)
  - \(\text{strings}\): number of string dopes
- **\(w_2\)** *message send dope*: buffer space used on sending i.e., buffer size used (\(\text{words}\)) and string dopes used (\(\text{strings}\)) 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:**

<table>
<thead>
<tr>
<th>w3</th>
<th><em>rcv string</em>(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>w2</td>
<td>rcv string size*(64)</td>
</tr>
<tr>
<td>w1</td>
<td><em>snd string</em>(64)</td>
</tr>
<tr>
<td>w0</td>
<td>snd string size*(64)</td>
</tr>
</tbody>
</table>

- **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
IPC Result Status

MESSAGE DOPE returns status word:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$0_{32}$</td>
<td>$\text{words}_{19}$</td>
<td>$\text{str}_5$</td>
</tr>
</tbody>
</table>

**words**: size in words of in-line data received
- in addition to registers

**str**: number of strings received
- in buffers pointed to by string dopes
- dopes contained in message header

**cc**: condition code
IPC Result Condition Code

**CONDITION CODE FORMAT:**

\[
\begin{array}{c}
ec_{(4)} \ i \ r \ m \ d
\end{array}
\]

**ec** error code: \( ec \neq 0 \Rightarrow \text{IPC failed} \)
- result codes in manual
- frequent reason: *Cut message* — receiver’s buffer was too small, not enough strings, etc.
- **Note:** this value is also delivered as C return value

**m** map bit: \( m = 1 \Rightarrow f\text{pages were received} \)

Other bits are for clans and chiefs, consult the manual.
IPC Timeout Specifications

- An IPC operation specifies 4 timeout values:

<table>
<thead>
<tr>
<th>$m_r(8)$</th>
<th>$m_s(8)$</th>
<th>$p_r(4)$</th>
<th>$p_s(4)$</th>
<th>$e_s(4)$</th>
<th>$e_r(4)$</th>
</tr>
</thead>
</table>

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

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

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

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

- $e = 0$ or $p = 0$ mean $\infty$, i.e., no timeout
- $m = 0$ & $e > 0$ mean 0, i.e., never block
- $p = 15$ means 0, i.e., fail on page fault
Timeouts...

- IPC timeouts can be specified between $1\mu s$ and $19h$
- page fault timeouts can be specified between $4\mu s$ (1ms) and 256s
- actual timeout resolution is more coarse
  ➔ 1ms timeout resolution on MIPS
- Data structure `l4/types.h:l4_timeout_t, L4_IPC_TIMEOUT()`
- Utilities (`time.h`):
  ```c
  void l4_mips_encode_timeout(
      dword_t msecs, byte_t *mant,
      byte_t *exp, byte_t round);
  dword_t l4_mips_decode_timeout(byte_t mant, byte_t exp);
  ```
Specification of Mappings

- Source and destination specified as fpages.
- Sender specifies a set of fpages which are to be mapped/granted to the receiver.
- Receiver specifies the *receive window* ➔ kernel will not map outside this window!
Specification of Mappings

- Source and destination specified as fpages.
- Sender specifies a set of fpages which are to be mapped/granted to the receiver.
- Receiver specifies the *receive window* ➔ kernel will not map outside this window!

**FPAGE REPRESENTATION (MIPS)**

\[
\text{base}/4096_{(52)} \quad 0_{(3)} \quad s_{(7)} \quad \sim \sim
\]

- Data structure `l4/types.h:l4_fpage_t`.
- MIPS kernel presently only supports 4kB pages
- Bigger (send) fpages are handled by mapping individual pages
Fpage Mapping

- Send and receive fpages may be of different size:
  - page fault receive fpage covers full address space,
  - there may be several send fpage, but only one receive fpage,
  - fpage may need to be mapped at different addresses in sender and receiver (e.g., on page fault).
Fpage Mapping

- Send and receive fpages may be of different size:
  - page fault receive fpage covers full address space,
  - there may be several send fpage, but only one receive fpage,
  - fpage may need to be mapped at different addresses in sender and receiver (e.g., on page fault).
- Need ability to specify where an fpage gets mapped
- Send fpage is accompanied with a *hotspot* address (send base)
  - Determines mapping address if receive fpage is big enough.
Fpage Mapping

- Send and receive fpages may be of different size:
  - Page fault receive fpage covers full address space,
  - There may be several send fpage, but only one receive fpage,
  - Fpage may need to be mapped at different addresses in sender and receiver (e.g., on page fault).
- Need ability to specify where an fpage gets mapped
- Send fpage is accompanied with a *hotspot* address (send base)
  - Determines mapping address if receive fpage is big enough.

**Send Fpage Information**

<table>
<thead>
<tr>
<th></th>
<th>snd fpage&lt;sub&gt;62&lt;/sub&gt;</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>w&lt;sub&gt;1&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w&lt;sub&gt;0&lt;/sub&gt;</td>
<td>snd base&lt;sub&gt;64&lt;/sub&gt;</td>
<td></td>
</tr>
</tbody>
</table>

\(w\): *write-permission bit*, unset \(\Rightarrow\) fpage will be mapped read-only

\(g\): *grant bit*, set \(\Rightarrow\) fpage will be granted
Fpage Mapping Rules

• Rules for disambiguating fpage mapping:
  * Hotspot address is taken modulo receive fpage size.
    ➔ Uniquely determines a page in the receiver’s address space which will receive a mapping.
  * Hotspot address is also taken modulo send fpage size.
    ➔ Uniquely determines a page in the send fpage which will be mapped.

• 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.
Fpage Mapping Examples

- s=10, b=5
- s=12, b=0
- s=10, b=5

h = 2 \times 2^{10} + \text{offset}
Formal Fpage Mapping Rules

- 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 $b \times 2^s \leftrightarrow b' \times 2^{s'}$
  - hot spot specification is not needed

$s < s'$: mapping is $b \times 2^s \leftrightarrow b'[63,s']h[s'-1,s]0(s)$
  - sender’s fpage is aligned around hot spot

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

- **Note:** Only $h \mod \max(2^s, 2^{s'})$ is relevant.
Page fault IPC (manufactured by kernel) specifies whole address space for receiver \((b' = 0, s' = 64)\).

Present MIPS implementation only uses smallest hardware page size \((s = 12)\), but that is transparent to user

→ *Changed in next release.*

Present MIPS implementation does not support granting

→ *Changed in next release.*

An attempt to map over an existing mapping is silently ignored (except if the mappings only differ in the write permission). This is a **bug** in *all* present L4 implementations!

→ *Changed in next release.*
Revoke Mappings: fpage_unmap()

- Unmaps pages directly or indirectly mapped from caller’s address space.
- Unmapping may be:
  - partial (revert to read-only, “remap”), or
  - complete (pages vanish from other address spaces, “flush”).
- The fpage argument defines a region in the caller’s address space.
- All pages within that region which are mapped into other address spaces will be remapped or flushed.
- Mappings in the caller’s address space are
  - unaffected (“other”), or
  - also unmapped (“all”).
Obtain Thread IDs: id_nearest System Call

- Returns the ID of the thread which would really receive a message sent to a specified destination thread.
- Also returns a type field, indicating the direction of the IPC with respect to the clan boundary.
- If destination is:
  - inside own clan: returns destination thread ID, type = same;
  - outside own clan: returns own chief’s ID, type = outer;
  - in subclan of own clan: returns ID of chief (within own clan) of subclan, type = inner;
  - nil: returns own thread ID.
Task Creation and Deletion: task_new()

- System has fixed number of tasks, initially all *inactive*.
- Inactive task is essentially a capability to create an active one.
- Task is *active* iff it has a valid pager.
- `task_new` deletes an (active or inactive) task and creates a new one (with task same number but different *version*, hence different ID).
- New task can be
  - *active*: syscall parameters specify start address, stack pointer, pager, exception handler, scheduling priority;
  - initially runs single thread (lthread 0);
  - *inactive*: does not consume any resources, can optionally be donated to new chief.
Details of task_new Operation

- Donation of inactive tasks allows passing of creation right.
- Deleting an inactive task does not affect version number.
- Deleting a task implicitly deletes all tasks in its clan or subclans.
- Only a task’s chief can execute a task_new syscall for it.
- **Exception** (MIPS): Anyone can call task_new for a task which has never been active.
  → Means of allocating task creation rights at system startup.
Thread Manipulation: lthread_ex_regs()

- Task has a fixed number (128) of threads, initially all but one inactive.
- Thread is activated by supplying a valid IP and SP.
  ➜ Thread inherits pager, excepter from activating thread.
- lthread_ex_regs() sets new and returns previous values for instruction pointer (IP), stack pointer (SP), exception handler, pager.
- Supplying invalid value (-1) to any of those retains original setting. Can be used for:
  ➜ performing a user-level thread switch
    (exchanging registers of running thread with saved ones);
  ➜ saving thread’s context (by supplying only invalid parameters).
- Call terminates any pending or ongoing IPC.
- **Note:** A thread cannot be “deleted”, only blocked.
Release CPU: thread_switch()

The calling thread voluntarily releases the CPU.

- May specify another thread to continue immediately
  ("time slice donation").
  ➔ Destination thread gets remaining time slice "for free".
  ➔ Normal scheduling taken at expiry.
- May *yield* CPU by not specifying valid destination thread.
  ➔ Remaining time thread is forfeit.
  ➔ Normal scheduling action taken immediately
    (possibly re-scheduling caller thread).
Scheduling Parameters: thread_schedule()

- Allows setting/inquiring the priority and timeslice length of a thread.
- Also returns thread state (running/IPC-ing/dead).
  ➔ *Partially implemented on MIPS.*
- Scheduling parameters can *only* be changed for a thread running at a lower priority than the caller’s *maximum controlled priority* (MCP).
- If setting priority, cannot exceed caller’s MCP.
- MCP is task attribute, specified in task_create system call (child MCP cannot exceed parent’s).
Task and thread management

- Management of task and thread IDs is left to user-level code.
- L4’s `task_create` and `lthread_ex_regs` system calls will happily destroy running tasks/threads if requested to do so.

⇒ It is up to the user code (i.e., OS server) to manage them properly.
L4 Scheduling

• Every L4 thread has a timeslice length and a priority.
• These are:
  ➔ inherited from parent,
  ➔ changeable via thread_schedule().
• L4 implements hard priorities:
  ➔ scheduler will always select highest priority runnable thread,
  ➔ within priority scheduler uses round-robin.
User-level scheduling in L4

Two ways to control scheduling:

- Can use controller thread (with high MCP):
  ➜ uses thread\_schedule to manipulate other threads,
  ➜ controlled threads run with zero MCP.
- Can use user-level scheduler thread running at highest priority:
  ➜ L4 will always schedule this scheduler thread,
  ➜ Scheduler thread uses thread\_switch() to give a time slice to some thread.
- Preempters (unimplemented) would be used to inform scheduler of preemptions.

Obviously combinations of these are possible.
The Root Pager $\sigma_0$

- $\sigma_0$ is the *initial address space*.
- $\sigma_0$ contains a mapping for each available frame of physical memory.
- $\sigma_0$ is also a pager (and chief) for *original servers* (tasks contained in boot image and marked as automatically run).
- $\sigma_0$ maps any frame (writable) to the first task requesting it (and ignores any further requests for the same frame).
- Pages can be requested implicitly (by touching) or explicitly (by RPC according to paging protocol).
  - First job of “OS personality” is to request all available frames.
  - Server has then control over memory.
- Some special pages are used for *kernel information page* and memory-mapping *devices*.
- On MIPS $\sigma_0$ runs in kernel space for no good reason.
Kernel information page

- Lives in kernel reserved space
- Mapped by $\sigma_0$ upon requesting a particular invalid page (address -3 on MIPS).
- Mapped read-only to anyone requesting it at any time.
- Contains information about L4 and machine:
  - L4 version etc,
  - size of physical memory,
  - size and address of L4 reserved memory,
  - millisecond real-time clock,
  - address of *DIT header page* (MIPS).
DIT header page (MIPS)

- Lives in kernel reserved space.
- Address is contained in kernel info page.
- Mapped read-only to anyone requesting it at any time.
- Contains for each file in the boot image:
  - name,
  - size and location in physical memory,
  - entrypoint address (zero if not executable image),
  - flag indicating whether it’s to be started by $\sigma_0$. 
Devices (MIPS)

- Are memory-mapped to addresses outside RAM range.
- Device pages are mapped upon requesting a particular invalid page with page address as second parameter.
- Mapped writable and uncacheable to anyone requesting it at any time.
  - **Note:** only tasks in $\sigma_0$’s clan can IPC directly to $\sigma_0$!
- “Device” mappings within RAM are used for DMA-able memory.
- Present MIPS $\sigma_0$ does not check whether address really refers to a device.
- Cacheability attribute is passed on when mapping to subtasks → supports device drivers not directly in $\sigma_0$’s clan.
Bootstrap (MIPS Specific)

Boot image:

| header | L4       | initial server... | other stuff |

On boot:

① Load image into RAM.
② L4 bootstrap starts $\sigma_0$ as first task (in kernel mode).
③ $\sigma_0$ starts all *initial servers* (as user tasks)
   registering as their pager, exception handler (and chief).

Initial servers are marked as such in boot image set up by DIT ("downloadable image tool").
OS Startup Code

① Register itself for all free interrupts.
② Grab all memory from $\sigma_0$ (without interfering with other initial tasks).
③ Set up data structures for memory management:
  ➔ reserved space for own tables,
  ➔ free lists/frame table/page tables for client memory.
④ Grab all (inactive) tasks.
⑤ Start device drivers (maybe separate initial server tasks).
  ➔ Drivers map device pages.
⑥ Start other server threads (if multi-server implementation).
⑦ Set up data structures for services (TCBs, file system, ...)
⑧ Set up task management.
⑨ Start up initial “user” task(s).
⑩ Possibly donate tasks to subtasks.
Device drivers

- maps device registers (device mapping from $\sigma_0$)
- maps uncached RAM (device mapping from $\sigma_0$)
- sets up interrupt handler thread for device
- initialises device (by writing to device registers)
- process user requests and device interrupts
Implication of server architecture

“OS” is just a user-level L4 task.

⇒ OS can be interrupted & unscheduled.

⇒ Need concurrency control on all OS data structures!
What to run first?

OS always

→ handles a client request, or
→ waits for a client request

Q: Where does it get the clients from?
What to run first?

OS always

→ handles a client request, or
→ waits for a client request

Q: Where does it get the clients from?

A: define your own startup convention, e.g.:

- OS starts up first non-initial server executable in boot image,
- First non-executable item in boot image contains list of initial client tasks,
- OS looks for program of a certain name in boot image.
Where’s the OS?

How does a client know where to send syscall RPCs?

1. First action of new task could be an “open receive”:
   - parent sends message (thereby disclosing its identity)
   - hide in startup code (“crt0”)

2. Client could send all system call RPCs to $\sigma_0$:
   - clans & chiefs mechanism ensures that parent receives message,
   - parent replies by “deceiving send” pretending to be $\sigma_0$.

$\Rightarrow$ It’s a matter of convention.
References
