

## 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/`,  
`$L4/include/l4/`
  - documented in Unix man pages `http://~cs9242/man/`,  
`$L4/man/man2/`
- Usage is explained in the L4 User Manual [AH98]

## L4 SYSTEM CALLS

- ① **ipc()**
  - Message passing, combining sending and receiving operations
- 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

One system call, variants accessible via separate C library entry points

### CLIENT FUNCTIONS:

- **send()** send a message (blocking) to a specific thread
- **receive()** “closed” receive from specific sender (might be interrupt) — includes sleeping (if specify invalid sender)
- **wait()** “open” receive from any thread (incl. interrupt)
- **call()** send and wait for reply (usual “RPC” operation)
- **reply\_and\_wait()** send message and wait for any new message  
typical server operation
- **send\_deceiving()** like **send()** but substituting sender ID
- **reply\_deceiving\_and\_wait()**  
like **reply\_and\_wait()** but substituting sender ID

**C BINDINGS FOR CHIEFS:** Alternative bindings which return the ID of the intended destination in the case of redirection:

→ **chief\_send()**

identical to **send\_deceiving**

→ **chief\_wait()**

like **wait()** but returns ID of intended destination

→ **chief\_receive()**

like **receive()** but returns ID of intended destination

→ **chief\_call()**

like **call()** but substitutes sender ID and returns ID of intended destination

→ **chief\_reply\_and\_wait()**

like **reply\_deceiving\_and\_wait()** but returns ID of *intended destination*

The ID of the *intended destination* supports transparent forwarding by chief.

## DECEIVING IPC

Clans & chiefs mechanism is supported by a feature called *deceiving*:

- If “*deceit*” *bit* is set by sender, L4 will lie to the receiver about the sender, delivering the sender-specified *virtual sender ID*
- The receiver is alerted to the deceit by the *deceit bit* in the return status
- Deceiving only works if *direction preserving*, i.e., if the real sender is along the redirection chain from the virtual sender to the receiver:
  - message goes out of clan and virtual sender ID is within clan (or subclan), or
  - message goes to subclan and virtual sender ID is outside clan.
- Used by chief to make inter-clan IPC interception transparent to clients

## EXAMPLE: SEND CALL

```
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  $\infty$ ).

## EXAMPLE: RECEIVE CALL

```
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*” (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

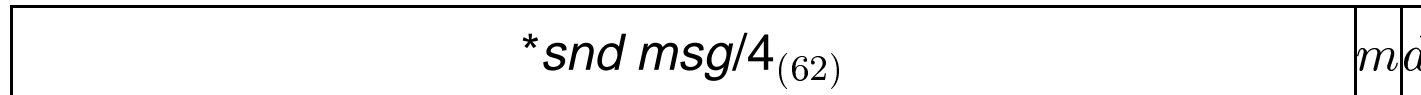


- presence of fpages is indicated by the *m*-bit in *snd\_msg* or *rcv\_msg*.
- fpage processing stops if invalid fpage descriptor found
- remainder (or all) of register data is simply copied

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

## SEND MESSAGE DESCRIPTOR FORMAT

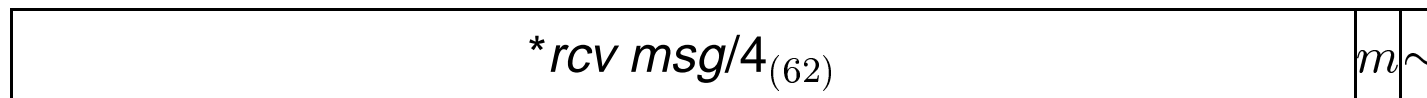
Format of *snd\_msg* parameter:



- **\*snd msg/4** — *message descriptor address*
  - = 0: *short message* (no memory data)
  - ≠ 0: *address of message descriptor*
- **m** — *mapping bit*
  - = 0: *by-value send operation*
    - message string contains only by-value data
  - ≠ 0: *by-reference (mapping) send operation*
    - beginning of message string contains *fpage descriptors*
    - (potentially followed by more by-value data)
- **d** — *deceiving bit*: lie about sender
  - turned on automatically by using *deceiving send* function.

## RECEIVE MESSAGE DESCRIPTOR FORMAT (MIPS)

Format of *msg\_rcv* parameter:



*m* = 0: *by-value receive operation*

message string contains only by-value data

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

***\*rec msg*** ≠ 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)

## MEMORY MESSAGE FORMAT

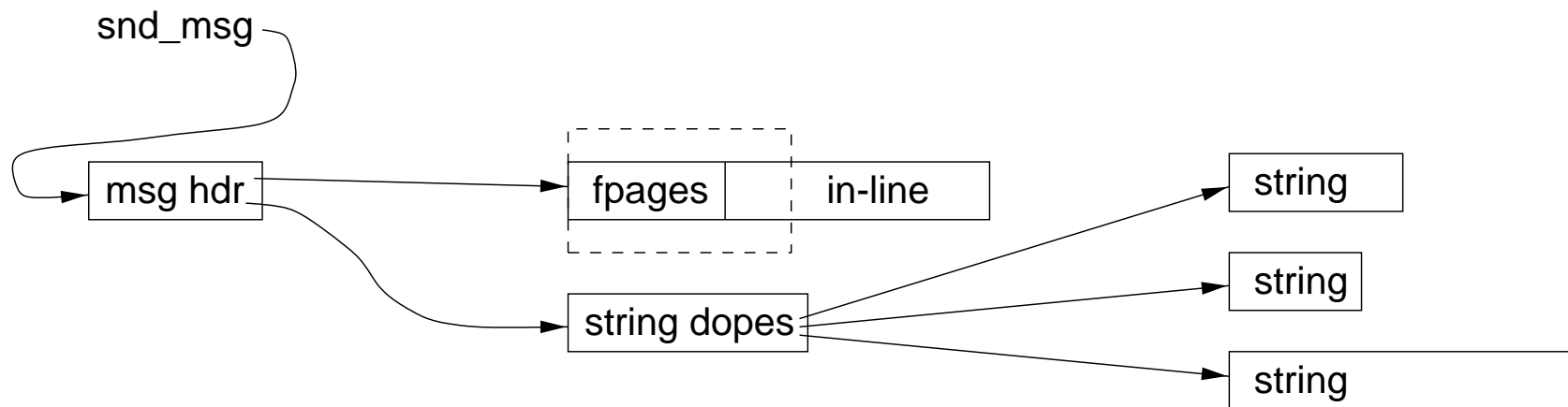


- Fpages are expected only if
  - the *m*-bit is set in the message descriptor, **and**
  - all register data consists of valid fpages.
- Fpage processing stops if invalid fpage descriptor found.
- Remainder of data (up to size given in header) is simply copied.
- String descriptors (“dopes”) immediately follow in-line data in memory.

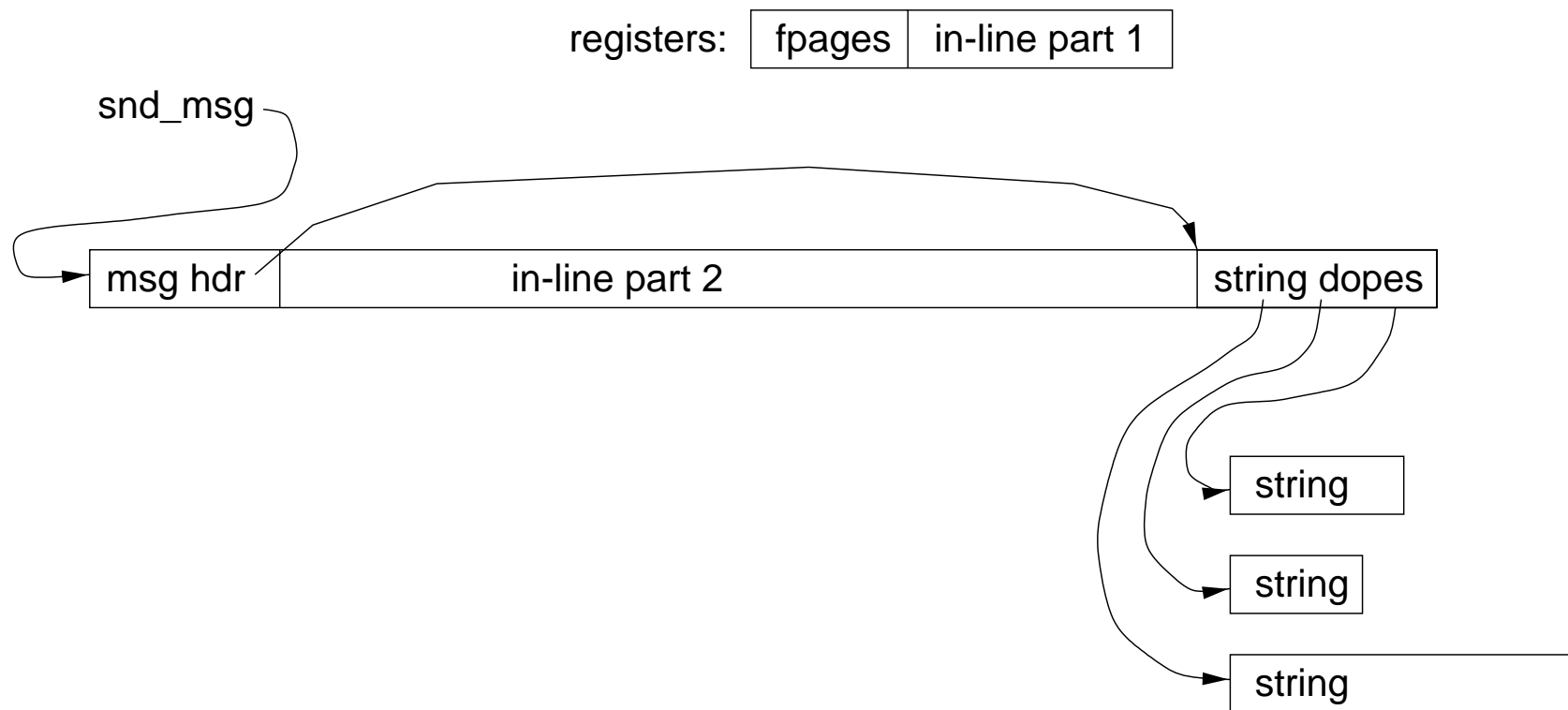
**Note:** present R4k implementation allows no more than 4 fpages (no fpage descriptors in memory).

This is not a problem in practice.

## MESSAGE FORMAT: LOGICAL



## MESSAGE FORMAT: PHYSICAL



## MESSAGE HEADER FORMAT

Data structure `l4/types.h:l4_msghdr_t`

w2	$0_{(32)}$	<i>words</i> <sub>(19)</sub>	<i>str</i> <sub>(5)</sub>	$\sim_{(8)}$
w1	$0_{(32)}$	<i>words</i> <sub>(19)</sub>	<i>str</i> <sub>(5)</sub>	$\sim_{(8)}$
w0	<i>fpage</i> <sub>(64)</sub>			

**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	<i>*rcv string</i> <sub>(64)</sub>
w2	<i>rcv string size</i> <sub>(64)</sub>
w1	<i>*snd string</i> <sub>(64)</sub>
w0	<i>snd string size</i> <sub>(64)</sub>

**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

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

$0_{(32)}$	<i>words</i> <sub>(19)</sub>	<i>str</i> <sub>(5)</sub>	<i>cc</i> <sub>(8)</sub>
------------	------------------------------	---------------------------	--------------------------

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

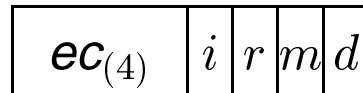
**str:** number of strings received

(in buffers pointed to by string dopes in message header supplied to call)

**cc:** condition code

## IPC RESULT CONDITION CODE

Condition code format:



**ec** 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:

$m_r(8)$	$m_s(8)$	$p_r(4)$	$p_s(4)$	$e_s(4)$	$e_r(4)$
----------	----------	----------	----------	----------	----------

$m_r, e_r$ : 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  $4^{15-p_s} \mu s$

Zero values of  $e$  or  $p$  mean  $\infty$ , 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

## TIMEOUTS...

- IPC timeouts can be specified between  $1\mu\text{s}$  (MIPS: 1ms) and 19h
- page fault timeouts can be specified between  $4\mu\text{s}$  (1ms) and 256s

However, actual timeout resolution is generally more coarse:

1ms timeout resolution on MIPS

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

Utilities (`time.h`):

```
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);
```

And similar for page-fault timeouts.

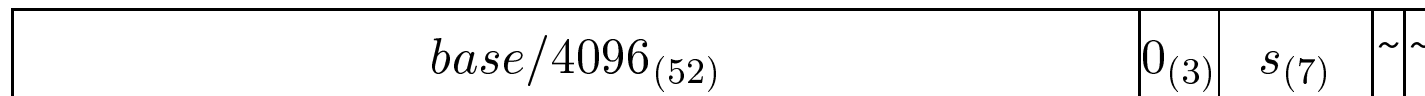
## SPECIFICATION OF MAPPINGS

- Source and destination of mappings are specified as fpages.
- Sender specifies a set of fpages which are to be mapped/granted to the receiver.
- Receiver specifies the window where mappings will be accepted as a *receive fpage*.

## SPECIFICATION OF MAPPINGS

- Source and destination of mappings are specified as fpages.
- Sender specifies a set of fpages which are to be mapped/granted to the receiver.
- Receiver specifies the window where mappings will be accepted as a *receive fpage*.

### FPAGE REPRESENTATION (MIPS):



- Data structure `l4/types.h:l4_fpage_t`.
- MIPS kernel presently only supports 4kB pages
- Bigger (send) fpages are handled as if all component pages were listed individually.

## FPAGE MAPPING

- In general send and receive fpages will 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).

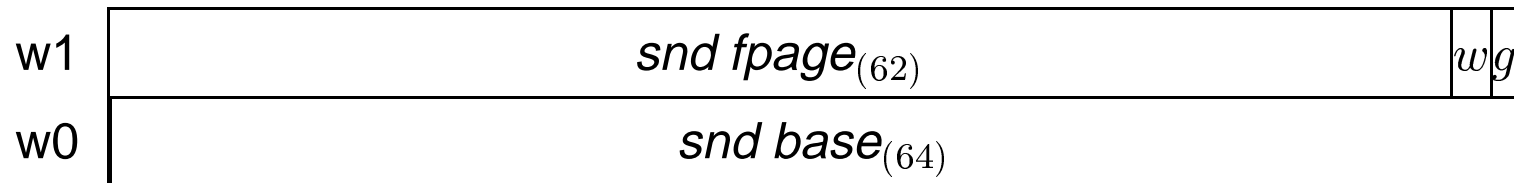
## FPAGE MAPPING

- In general send and receive fpages will 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
- Each send fpage is accompanied with a *hotspot* address (send base)
  - Determines mapping address if receive fpage is big enough.

## FPAGE MAPPING

- In general send and receive fpages will 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
- Each send fpage is accompanied with a *hotspot* address (send base)
  - Determines mapping address if receive fpage is big enough.

### SEND FPAGE INFORMATION:



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

*g*: grant bit, set  $\Rightarrow$  fpage will be granted

## FPAGE MAPPING RULES

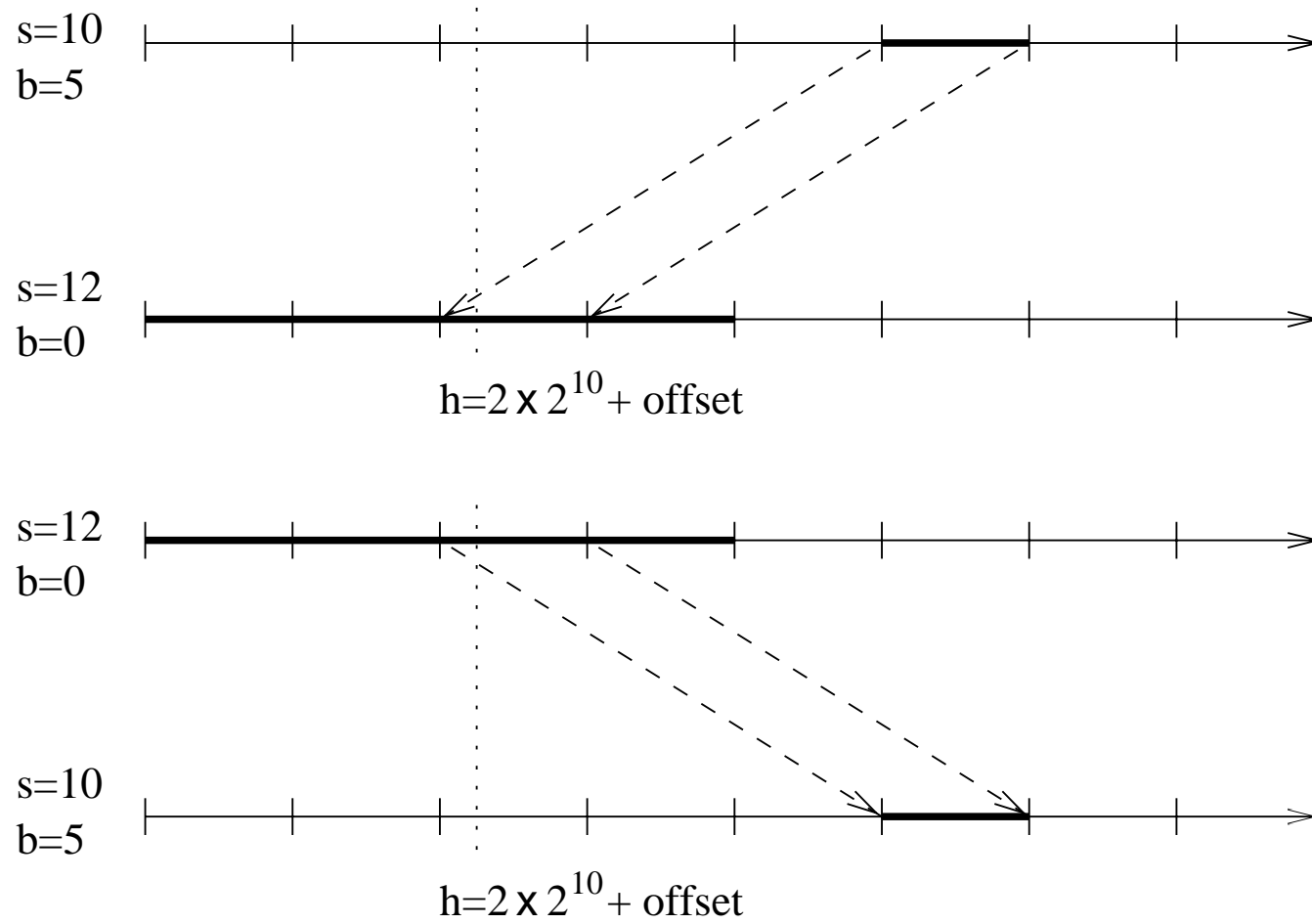
In order to disambiguate fpage mapping, the following rules apply:

- 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



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

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

## FPAGE NOTES

- 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
- Present MIPS implementation does not support *granting*
- 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!

## REVOKE MAPPINGS: FPAGE\_UNMAP SYSTEM CALL

- 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 CALL

- 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 SYSTEM CALL

- 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”, it is stopped by blocking.

## RELEASE CPU: THREAD\_SWITCH SYSTEM CALL

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 destination thread.
  - Remaining time thread is forfeit.
  - Normal scheduling action taken immediately (possibly re-scheduling caller thread).

## MANIPULATE SCHEDULING PARAMETERS: THREAD\_SCHEDULE SYSTEM CALL

- Allows setting/inquiring the priority and timeslice length of a thread.
- Also returns thread state (running/IPC-ing/dead).
- 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.

Obvioulsy 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 (may be started as 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.

## 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  $\sigma_0$ .

⇒ It's a matter of convention.

## REFERENCES

- [AH98] Alan Au and Gernot Heiser. *L4 User Manual*. School Comp. Science & Engin., University NSW, Sydney 2052, Australia, January 1998. UNSW-CSE-TR-9801. Latest version available from <http://www.cse.unsw.edu.au/~disy/L4/>.
- [EHL97] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual — MIPS R4x00*. School Comp. Science & Engin., University NSW, Sydney 2052, Australia, December 1997. UNSW-CSE-TR-9709. Latest version available from <http://www.cse.unsw.edu.au/~disy/L4/>.