More on L4 IPC

- sending to a non-existing thread fails immediately
- receiving from non-existing thread blocks (until timeout)
  ⇒ use for timed sleep
- receive can be
  - "open" (`l4_mips_ipc_wait` — receive from any)
  - "closed" (`l4_mips_ipc_receive` — receive from specific sender)
- receive send can be combined as:
  - send&wait (`l4_mips_ipc_call` — "normal" RPC operation)
  - reply&wait (`l4_mips_ipc_reply_and_wait`) — like "call",
    but wait is open (typical server operation)

Deceiving IPC

- If *"deceit" bit* is set in send descriptor parameter,
  L4 will deliver to the receiver instead of the real sender ID the *virtual sender ID* supplied by the sender
- the receiver is alerted to the deceit by the *deceit bit* in the return status
- deceiving only works if *direction preserving*, i.e.:
  - 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

Note: L4 does not tell interceping chief the intended destination!
⇒ some user-level protocol required to allow chief to forward message

Deceiving IPC...

- used by chief to make inter-clan IPC interception transparent to clients
- can also be used for transparent RPC redirection [later]
- C library: `l4_mips_ipc_send_deceiving`,
  `l4_mips_ipc_reply_deceiving_and_wait`

Revoke mapping: fpage_unmap

Unmaps pages directly or indirectly mapped from caller's address space.

Unmapping may be

- partial (revert to read-only, "**remap**")
- 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**").

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 a (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

task_new...

- 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:** anyone can call `task_new` for a task which has never been active

⇒ initial servers reserve all tasks by calling `task_new` for all possible tasks

- donation of inactive tasks allows passing of creation right.

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
- `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 thread switch
    (exchanging registers of running thread with saved ones)
  - saving thread's context (by supplying only invalid parameters)
- call aborts any pending IPC

Note: A thread cannot be "deleted", it is stopped by blocking.

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

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
- No destination thread specified
  - ⇒ remaining time thread is forfeit normal scheduling action taken immediately (possibly re-scheduling caller thread)

Typical yield operation as in non-prepemtive scheduling

L4 Scheduling

- every L4 thread has a *timeslice length* and a *priority*
- these are inherited from parent
- L4 implements hard priorities:
  - schedule will always select highest priority runable thread
  - within priority uses round-robin
- priority and time slice can be set with thread_schedule system call

Manipulate scheduling parameters: thread_schedule

- 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)

User-level scheduling in L4

Two ways to control scheduling:

- can use controller thread (with high MCP) which uses thread_schedule to manipulate other threads (all running with zero MCP)
- can use user-level scheduler thread running at highest priority:
  - L4 will always schedule this thread
  - it will determine which user thread it wants to run
  - uses thread_switch to donate its time slice to that thread

Obvioulsy combinations of these are possible

Obtain thread IDs: id_nearest

Returns the ID of the thread which would **really** receive a message sent to a specified destination thread.

If destination is

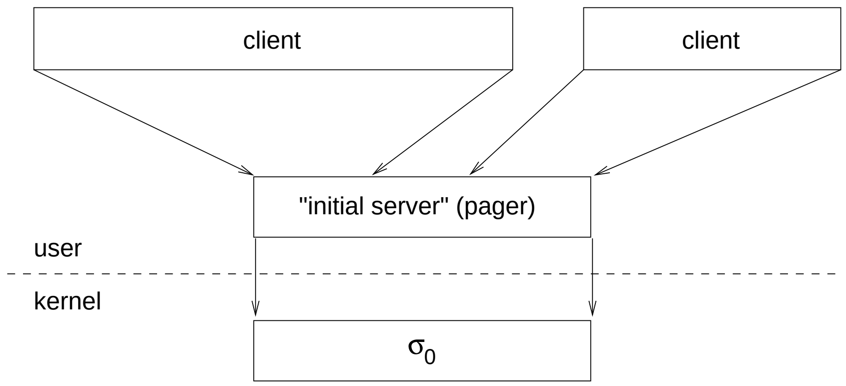
inside own clan: returns destination thread ID

outside own clan: returns own chief's ID

in subclan of own clan: returns ID of chief (within own clan) of subclan

***nil*:** returns own thread ID

L4 Use



The root pager σ_0

- σ_0 is the *initial address space*
- σ_0 runs in kernel space (L4/MIPS) or user space (L4/x86, L4/Alpha)
- σ_0 maps physical memory (except for some kernel-reserved regions)
- σ_0 is also a pager (and chief) for *original servers*
- σ_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*

Kernel information page

- lives in kernel reserved space
- mapped 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 clock
  - address of *DIT header page*

DIT header page

- lives in kernel reserved space
- address is contained in kernel info page
- mapped read-only to anyone requesting it at any time
- contains information about the downloaded 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 σ_0

  of each file loaded into the boot image.

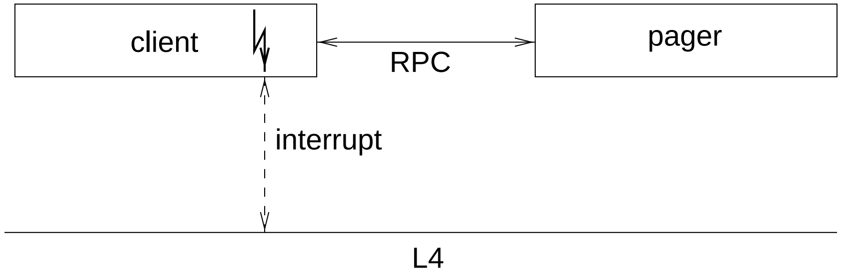
Devices

- identified by address outside RAM range
- mapped upon requesting a particular invalid page (address -2 on MIPS) with device address as second parameter
- mapped writable and *uncacheable* to anyone requesting it at any time
  Note: only tasks in σ_0's clan can IPC directly to σ_0
- present MIPS implementation does not check whether address really refers to a device

⇒ this is a **security hole** in the present MIPS implementation

- cacheability attribute is passed on when mapping to subtasks

Page fault handling



- client triggers page fault
- L4's interrupt handler sends RPC to pager on client's behalf detailing faulting address and instruction pointer
- pager replies with an fpage mapping
- kernel receives mapping on client's behalf and continues client

Exception handling

Handled similar to page faults (on L4/MIPS):

- every thread has an exception handler thread (defined at thread/task creation)
- L4's interrupt handler sends RPC to exception handler, detailing faulting address, instruction pointer and exception code
- exception handler can choose to continue thread by replying to message

Interrupt handling

- A thread can install itself as a handler for a particular interrupt ("associate" itself with interrupt)
- association is done by a special form of *receive* IPC operation specifying the interrupt number as the *sender* ID and a zero timeout
- to wait for interrupt handler receives with non-zero timeout
- "associating" with NULL interrupt *dissociates* handler from interrupt
- interrupt is associated with *first thread asking*
- attempts to associate with already associated interrupt will fail

Bootstrap

Boot image:

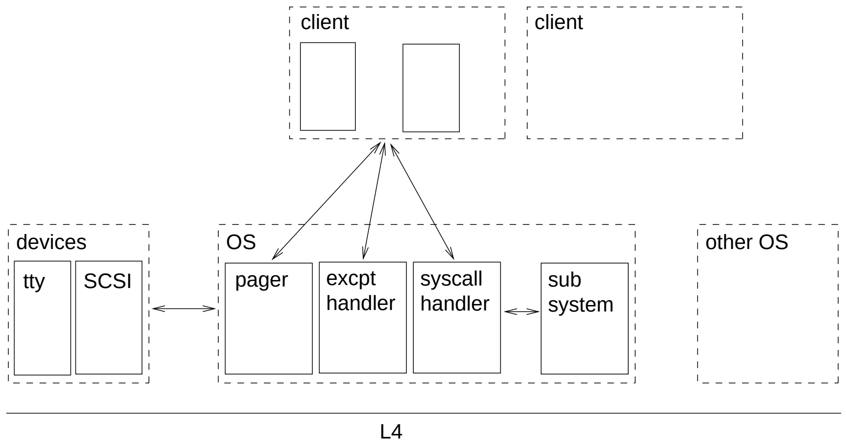
header	L4	initial server...	other stuff

On boot:

- load image into RAM
- L4 bootstrap starts σ_0 as first process (in kernel mode)
- σ_0 starts all *initial servers* (in user mode) 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 "personalities"

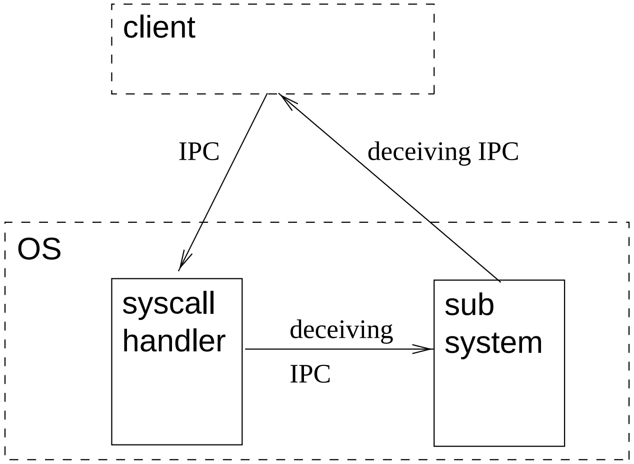


Typical L4-based OS structure

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 may 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 servers running as separate tasks
  - allows for easy redirection of "system call" RPC

RPC redirection



RPC redirection...

- client (via syscall library stub) sends RPC to syscall server
  - waits for reply from syscall server
- syscall server decides to forward to another server
  - uses "deceiving send" IPC operation giving client as virtual sender
- subsystem replies directly back to client
  - uses "deceiving send" giving syscall server as virtual sender
- sender receives (apparently) from syscall server

  ⇒ satisfies RPC semantics

OS startup code

- register itself for all free interrupts
- grab all memory from σ_0 (without interfering with other initial tasks)
- set up data structures for memory management:
  - reserved space for own tables
  - free lists/frame table/page tables for client memory
- grab all (inactive) tasks
- start device drivers (may be started as separate initial server tasks) these map device pages
- start other server threads (if multi-server implementation)
- set up data structures for services (TCBs, file system, ...)
- set up task management
- start up initial "user" task(s)
- possibly donate tasks to subtasks

Implication of server architecture

"OS" is just a user-level L4 task

⇒ 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?

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 σ_0
   - clans & chiefs mechanism ensures that parent receives message
   - parent replies by "deceiving send" pretending to be σ_0

$\Rightarrow$ It's a matter of convention.

