L4 Reference Manual

MIPS R4x00

Version 1.0

Kernel Version 70

Kevin Elphinstone, Gernot Heiser
Department of Computer Systems
School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia
{kevine,gernot}@cse.unsw.edu.au

Jochen Liedtke
IBM T. J. Watson Research Center
30 Saw Mill River Road, Hawthorne, NY 10532, USA
jochen@watson.ibm.com

UNSW-CSE-TR-9709 — December 1997

Department of Computer Systems
School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Note

This document describes release 1.0 of the L4 microkernel for the MIPS R4x00 microprocessor family. It is based on the L4/x86 reference manual Version 2.0 by Jochen Liedtke and has been modified to describe the MIPS implementation. Some material has been added to clarify the L4 message structure. Comments and critiques, as well as proposed additions and alternatives for future versions are most welcome.

The source code for L4/R4x00 is available free of charge under the terms of the GNU General Public License. To obtain the source contact disy@cse.unsw.edu.au. Future versions of this document, as well as related documents and tools, will be available from URL http://www.cse.unsw.edu.au/~disy/.

How To Read This Manual

This reference manual consists of two parts, (1) a processor-independent description of the principles and mechanisms of L4 and (2) a more detailed processor-specific description. Part 2 refers to the IDT R4x00.

Where L4/MIPS differs from L4/x86 significantly, or something is partially or completely unimplemented, then an implementation note appears as below. There is also a summary of various implementation details in section 2.1.

MIPS Implementation Note: This is what an implementation note looks like.

Acknowledgements

The original L4 reference manual was written by Jochen Liedtke, who would like to thank many people for their helpful contributions for improving the reference manual and the L4 interface. Particular thanks go to Bryan Ford, Hermann Härtig, Michael Hohmuth, Sebastian Schönberg and Jean Wolter. For the MIPS version we would like to thank in particular Jerry Vochteloo for testing the kernel, Alan Au for contributions to the manual, as well as the 1997 class of UNSW COMP9242 "guinea pigs" who built their operating systems on top of the MIPS version of the kernel.

Contents

Chapter 1

L4 in General

1.1 Basic Concepts

The following section contains excerpts from [Lie93b, Lie93a, Lie95].

We reason about the minimal concepts or "primitives" that a μ-kernel should implement.[1] The determining criterion used is functionality, not performance. More precisely, a concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

We assume that the target system has to support interactive and/or not completely trustworthy applications, i.e. it has to deal with protection. We further assume that the hardware implements page-based virtual memory.

One inevitable requirement for such a system is that a programmer must be able to implement an arbitrary subsystem S in such a way that it cannot be disturbed or corrupted by other subsystems S'. This is the principle of independence: S can give guarantees independent of S'. The second requirement is that other subsystems must be able to rely on these guarantees. This is the principle of integrity: there must be a way for S_1 to address S_2 and to establish a communication channel which can neither be corrupted nor eavesdropped by S'.

Provided hardware and kernel are trustworthy, further security services, like those described by [GGKL89], can be implemented by servers. Their integrity can be ensured by system administration or by user-level boot servers. For illustration: a key server should deliver public-secret RSA key pairs on demand. It should guarantee that each pair has the desired RSA property and that each pair is delivered only once and only to the demander. The key server can only be realized if there are mechanisms which (a) protect its code and data, (b) ensure that nobody else reads or modifies the key and (c) enable the demander to check whether the key comes from the key server. Finding the key server can be done by means of a name server and checked by public key based authentication.

1.1.1 Address Spaces

At the hardware level, an *address space* is a mapping which associates each virtual page to a physical page frame or marks it 'non-accessible'. For the sake of simplicity, we omit access attributes like read-only and read/write. The mapping is implemented by TLB hardware and page tables.

[1] Proving minimality, necessarity and completeness would be nice but is impossible, since there is no agreed-upon metric and all is Turing-equivalent.

The μ-kernel, the mandatory layer common to all subsystems, has to hide the hardware concept of address spaces, since otherwise, implementing protection would be impossible. The μ-kernel concept of address spaces must be tamed, but must permit the implementation of arbitrary protection (and non-protection) schemes on top of the μ-kernel. It should be simple and similar to the hardware concept.

The basic idea is to support recursive construction of address spaces outside the kernel. By magic, there is one address space σ_0 which essentially represents the physical memory and is controlled by the first subsystem S_0. At system start time, all other address spaces are empty. For constructing and maintaining further address spaces on top of σ_0, the μ-kernel provides three operations:

Grant. The owner of an address space can *grant* any of its pages to another space, provided the recipient agrees. The granted page is removed from the granter's address space and included into the grantee's address space. The important restriction is that instead of physical page frames, the granter can only grant pages which are already accessible to itself.

Map. The owner of an address space can *map* any of its pages into another address space, provided the recipient agrees. Afterwards, the page can be accessed in both address spaces. In contrast to granting, the page is not removed from the mapper's address space. Comparable to the granting case, the mapper can only map pages which itself already can access.

Flush. The owner of an address space can *flush* any of its pages. The flushed page remains accessible in the flusher's address space, but is removed from all other address spaces which had received the page directly or indirectly from the flusher. Although explicit consent of the affected address-space owners is not required, the operation is safe, since it is restricted to own pages. The users of these pages already agreed to accept a potential flushing, when they received the pages by mapping or granting.

Reasoning

The described address-space concept leaves memory management and paging outside the μ-kernel; only the grant, map and flush operations are retained inside the kernel. Mapping and flushing are required to implement memory managers and pagers on top of the μ-kernel.

The grant operation is required only in very special situations: consider a pager F which combines two underlying file systems (implemented as pagers f_1 and f_2, operating on top of the standard pager) into one unified file system (see figure 1.1). In this example, f_1

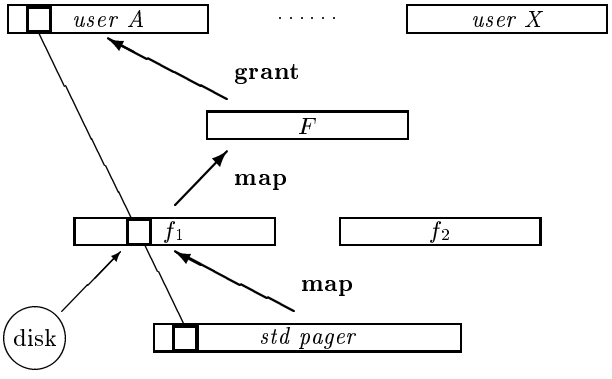


Figure 1.1: *A Granting Example.*

maps one of its pages to F which grants the received page to *user A*. By granting, the page

disappears from F so that it is then available only in f_1 and *user A*; the resulting mappings are denoted by the thin line: the page is mapped in *user A*, f_1 and the standard pager. Flushing the page by the standard pager would affect f_1 and *user A*, flushing by f_1 only *user A*. F is not affected by either flush (and cannot flush itself), since it used the page only transiently. If F had used mapping instead of granting, it would have needed to replicate most of the bookkeeping which is already done in f_1 and f_2. Furthermore, granting avoids a potential address-space overflow of F.

In general, granting is used when page mappings should be passed through a controlling subsystem without burdening the controller's address space by all pages mapped through it.

The model can easily be extended to access rights on pages. Mapping and granting copy the source page's access right or a subset of them, i.e., can restrict the access but not widen it. Special flushing operations may remove only specified access rights.

I/O

An address space is the natural abstraction for incorporating device ports. This is obvious for memory mapped I/O, but I/O ports can also be included. The granularity of control depends on the given processor. The 386 and its successors permit control per port (one very small page per port) but no mapping of port addresses (it enforces mappings with $v{=}v'$); the PowerPC uses pure memory mapped I/O, i.e., device ports can be controlled and mapped with 4K granularity. Controlling I/O rights and device drivers is thus also done by memory managers and pagers on top of the μ-kernel.

An Abstract Model of Address Spaces

We describe address spaces as mappings. $\sigma_0 : V \to R \cup \{\phi\}$ is the initial address space, where V is the set of virtual pages, R the set of available physical (real) pages and ϕ the nilpage which cannot be accessed. Further address spaces are defined recursively as mappings $\sigma : V \to (\Sigma \times V) \cup \{\phi\}$, where Σ is the set of address spaces. It is convenient to regard each mapping as a one column table which contains $\sigma(v)$ for all $v \in V$ and can be indexed by v. We denote the elements of this table by σ_v.

All modifications of address spaces are based on the replacement operation: we write $\sigma_v \leftarrow x$ to describe a change of σ at v, precisely:

$$\text{flush } (\sigma, v) \quad ; \quad \sigma_v \;:=\; x \quad .$$

A page potentially mapped at v in σ is flushed, and the new value x is copied into σ_v. This operation is internal to the μ-kernel. We use it only for describing the three exported operations.

A subsystem S with address space σ can *grant* any of its pages v to a subsystem S' with address space σ' provided S' agrees:

$$\sigma'_{v'} \;\leftarrow\; \sigma_v \quad , \quad \sigma_v \;\leftarrow\; \phi \quad .$$

Note that S determines which of its pages should be granted, whereas S' determines at which virtual address the granted page should be mapped in σ'. The granted page is transferred to σ' and removed from σ.

A subsystem S with address space σ can *map* any of its pages v to a subsystem S' with address space σ' provided S' agrees:

$$\sigma'_{v'} \;\leftarrow\; (\sigma, v) \quad .$$

In contrast to grant, the mapped page remains in the mapper's space σ and *a link to the page in the mapper's address space* (σ, v) *is stored in the receiving address space* σ', instead of transferring the existing link from σ_v to $\sigma'_{v'}$. This operation permits to construct address spaces recursively, i.e. new spaces based on existing ones.

Flushing, the reverse operation, can be executed without explicit agreement of the mappees, since they agreed implicitly when accepting the prior map operation. S can *flush* any of its pages:

$$\forall_{\sigma'_{v'}=(\sigma,v)} : \ \sigma'_{v'} \ \leftarrow \ \phi \quad .$$

Note that $\leftarrow$ and *flush* are defined recursively. Flushing recursively affects also all mappings which are indirectly derived from σ_v.

No cycles can be established by these three operations, since $\leftarrow$ flushes the destination prior to copying.

Implementing the Model

At a first glance, deriving the physical address of page v in address space σ seems to be rather complicated and expensive:

$$\sigma(v) = \begin{cases} \sigma'(v') & \text{if } \sigma_v{=}(\sigma', v') \\ r & \text{if } \sigma_v{=}r \\ \phi & \text{if } \sigma_v{=}\phi \end{cases}$$

Fortunately, a recursive evaluation of $\sigma(v)$ is never required. The three basic operations guarantee that the physical address of a virtual page will never change, except by flushing. For implementation, we therefore complement each σ by an additional table P, where P_v corresponds to σ_v and holds either the physical address of v or ϕ. Mapping and granting then include

$$P'_{v'} \ := \ P_v$$

and each replacement $\sigma_v \leftarrow \phi$ invoked by a flush operation includes

$$P_v \ := \ \phi \quad .$$

P_v can always be used instead of evaluating $\sigma(v)$. In fact, P is equivalent to a hardware page table. μ-kernel address spaces can be implemented straightforward by means of the hardware-address-translation facilities.

The recommended implementation of σ is to use one mapping tree per physical page frame which describes all actual mappings of the frame. Each node contains (P, v), where v is the according virtual page in the address space which is implemented by the page table P.

Assume that a grant-, map- or flush-operation deals with a page v in address space σ to which the page table P is associated. In a first step, the operation selects the according tree by P_v, the physical page. In the next step, it selects the node of the tree that contains (P, v). (This selection can be done by parsing the tree or in a single step, if P_v is extended by a link to the node.) Granting then simply replaces the values stored in the node and map creates a new child node for storing (P', v'). Flush lets the selected node unaffected but parses and erases the complete subtree, where $P'_v := \phi$ is executed for each node (P', v') in the subtree.

1.1.2 Threads and IPC

A *thread* is an activity executing inside an address space. A thread τ is characterised by a set of registers, including at least an instruction pointer, a stack pointer and a state information. A thread's state also includes the address space $\sigma^{(\tau)}$ in which τ currently executes. This dynamic or static association to address spaces is the decisive reason for including the thread concept (or something equivalent) in the μ-kernel. To prevent corruption of address spaces, all changes to a thread's address space ($\sigma^{(\tau)} := \sigma'$) must be controlled by the kernel. This implies that the μ-kernel includes the notion of some τ that represents the above mentioned activity. In some operating systems, there may be additional reasons for introducing threads as a basic abstraction, e.g. preemption. Note that choosing a concrete thread concept remains subject to further OS-specific design decisions.

Consequently, cross-address-space communication, also called inter-process communication (IPC), must be supported by the μ-kernel. The classical method is transferring messages between threads by the μ-kernel.

IPC always enforces a certain agreement between both parties of a communication: the sender decides to send information and determines its contents; the receiver determines whether it is willing to receive information and is free to interpret the received message. Therefore, IPC is not only the basic concept for communication between subsystems but also, together with address spaces, the foundation of independence.

Other forms of communication, remote procedure call (RPC) or controlled thread migration between address spaces, can be constructed from message-transfer based IPC.

Note that the *grant* and *map* operations (section 1.1.1) need IPC, since they require an agreement between granter/mapper and recipient of the mapping.

Interrupts

The natural abstraction for hardware interrupts is the IPC message. The hardware is regarded as a set of threads which have special thread ids and send empty messages (only consisting of the sender id) to associated software threads. A receiving thread concludes from the message source id, whether the message comes from a hardware interrupt and from which interrupt:

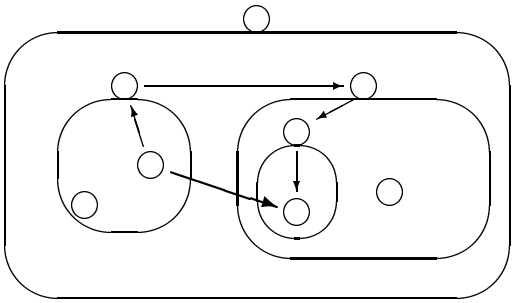
```
driver thread:
    do
        wait for (msg, sender) ;
        if sender = my hardware interrupt
            then  read/write io ports ;
                  reset hardware interrupt
            else  ...
        fi
    od .
```

Transforming the interrupts into messages must be done by the kernel, but the μ-kernel is not involved in device-specific interrupt handling. In particular, it does not know anything about the interrupt semantics. On some processors, resetting the interrupt is a device specific action which can be handled by drivers at user level. The `iret`-instruction then is used solely for popping status information from the stack and/or switching back to user mode and can be hidden by the kernel. However, if a processor requires a privileged operation for releasing an interrupt, the kernel executes this action implicitly when the driver issues the next IPC operation.

1.1.3 Clans & Chiefs

Within all systems based on direct message transfer, protection is essentially a matter of message control. Using access control lists (acl) this can be done at the server level, but maintenance of large distributed acls becomes hard when access rights change rapidly. So [HKK93] have proposed that object (passive entity) protection be complemented by subject (active entity) restrictions. In this approach the kernel is able to restrict the outgoing messages of a task (the subject) by means of a list of permitted receivers.

The clan concept [Lie92] is an algorithmic generalisation of this idea:



A *clan* (denoted as an oval) is a set of tasks (denoted as a circle) headed by a *chief* task. Inside the clan all messages are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless of whether it is outgoing or incoming, it is redirected to the clan's chief. This chief may inspect the message (including the sender and receiver ids as well as the contents) and decide whether or not it should be passed to the destination to which it was addressed. As demonstrated in the figure above, these rules apply to nested clans as well. Obviously subject restrictions and local reference monitors can be implemented outside the kernel by means of clans. Since chiefs are tasks at user level, the clan concept allows more sophisticated and user definable checks as well as active control. Typical clan structures are

Clan per machine: In this simple model there is only one clan per machine covering all tasks. Local communication is handled directly by the kernel without incorporating a chief, whereas cross machine communication involves the chief of the sending and the receiving machine. Hence, the clan concept is used for implementing remote ipc by user level tasks.

Clan per system version: Sometimes chiefs are used for adapting different versions. The servers of the old or new versions are encapsulated by a clan so that its chief can translate the messages.

Clan per user: Surrounding the tasks of each user or user group by a clan is a typical method when building security systems. Then the chiefs are used to control and enforce the requested security policy.

Clan per task: In the extreme case there are single tasks each controlled by a specific chief. For example these one-task-clans are used for debugging and supervising suspicious programs.

In the case of intra-clan communication (no chief involved), the additional costs of the clan concept are negligible (below 1% of minimal ipc time). Inter-clan communication

however multiplies the ipc operations by the number of chiefs involved. This can be tolerated, since (i) L4 ipc is very fast (see above) and (ii) crossing clan boundaries occurs seldom enough in practice. Note that many security policies can be implemented simply by checking the client id in the server and do not need clans.

1.1.4 Unique Identifiers

A μ-kernel must supply unique identifiers (uid) for something, either for threads or tasks or communication channels. Uids are required for reliable and efficient local communication. If S_1 wants to send a message to S_2, it needs to specify the destination S_2 (or some channel leading to S_2). Therefore, the μ-kernel must know which uid relates to S_2. On the other hand, the receiver S_2 wants to be sure that the message comes from S_1. Therefore the identifier must be unique, both in space and time.

In theory, cryptography could also be used. In practice, however, enciphering messages for local communication is far too expensive and the kernel must be trusted anyway. S_2 can also not rely on purely user-supplied capabilities, since S_1 or some other instance could duplicate and pass them to untrusted subsystems without control of S_2.

1.1.5 Flexibility

To illustrate the flexibility of the basic concepts, we sketch some applications which typically belong to the basic operating system but can easily be implemented on top of the μ-kernel.

Memory Manager. A server managing the initial address space σ_0 is a classical main memory manager, but outside the μ-kernel. Memory managers can easily be stacked: M_0 maps or grants parts of the physical memory (σ_0) to σ_1, controlled by M_1, other parts to σ_2, controlled by M_2. Now we have two coexisting main memory managers.

Pager. A Pager may be integrated with a memory manager or use a memory managing server. Pagers use the μ-kernel's grant, map and flush primitives. The remaining interfaces, pager – client, pager – memory server and pager – device driver, are completely based on IPC and are user-level defined.

Pagers can be used to implement traditional paged virtual memory and file/database mapping into user address spaces as well as unpaged resident memory for device drivers and/or real time systems. Stacked pagers, i.e. multiple layers of pagers, can be used for combining access control with existing pagers or for combining various pagers (e.g. one per disk) into one composed object. User-supplied paging strategies [LCC94, CFL94] are handled at the user level and are in no way restricted by the μ-kernel. Stacked file systems [KN93] can be realized accordingly.

Multimedia Resource Allocation. Multimedia and other real-time applications require memory resources to be allocated in a way that allows predictable execution times. The above mentioned user-level memory managers and pagers permit e.g. fixed allocation of physical memory for specific data or locking data in memory for a given time.

Note that resource allocators for multimedia and for timesharing can coexist. Managing allocation conflicts is part of the servers' jobs.

Device Driver. A device driver is a process which directly accesses hardware I/O ports mapped into its address space and receives messages from the hardware (interrupts) through the standard IPC mechanism. Device-specific memory, e.g. a screen, is handled by means of appropriate memory managers. Compared to other user-level processes, there is nothing special about a device driver. No device driver has to be integrated into the μ-kernel.[2]

Second Level Cache and TLB. Improving the hit rates of a secondary cache by means of page allocation or reallocation [KH92, RLBC94] can be implemented by means of a pager which applies some cache-dependent (hopefully conflict reducing) policy when allocating virtual pages in physical memory.

In theory, even a software TLB handler could be implemented like this. In practice, the first-level TLB handler will be implemented in the hardware or in the μ-kernel. However, a second-level TLB handler, e.g. handling misses of a hashed page table, might be implemented as a user-level server.

Remote Communication. Remote IPC is implemented by communication servers which translate local messages to external communication protocols and vice versa. The communication hardware is accessed by device drivers. If special sharing of communication buffers and user address space is required, the communication server will also act as a special pager for the client. The μ-kernel is not involved.

Unix Server. Unix[3] system calls are implemented by IPC. The Unix server can act as a pager for its clients and also use memory sharing for communicating with its clients. The Unix server itself can be page-able or resident.

Conclusion. A small set of μ-kernel concepts lead to abstractions which stress flexibility, provided they perform well enough. The only thing which cannot be implemented on top of these abstractions is the processor architecture, registers, first-level caches and first-level TLBs.

[2] In general, there is no reason for integrating boot drivers into the kernel. The booter, e.g. located in ROM, simply loads a bit image into memory that contains the micro-kernel and perhaps some set of initial pagers and drivers (running in user mode and *not* linked but simply appended to the kernel). Afterwards, the boot drivers are no longer used.

[3] Unix is a registered trademark of UNIX System Laboratories.

1.2 Data Types

1.2.1 Unique Ids

Unique ids identify tasks, threads and hardware interrupts. They are also unique in time. Unique ids are 64-bit values.

1.2.2 User-Level Operations on Uids

a = b :	a = b
task(a) = task (b) :	(a AND NOT lthread mask) = (b AND NOT lthread mask)
chief(a) = chief (b) :	(a AND NOT chief mask) = (b AND NOT chief mask)
site(a) = site (b) :	(a AND NOT site mask) = (b AND NOT site mask)
lthread no(a) :	(a AND lthread mask) SHR lthread shift *extract lthread no from thread id* a
thread(a,n) :	(a AND NOT lthread mask) + (n SHL lthread shift) *construct thread id from task id* a *and lthread no* n
task no(a) :	(a AND task mask) SHR task shift
chief no(a) :	(a AND chief mask) SHR chief shift
site no(a) :	(a AND site mask) SHR site shift

1.2.3 Fpages

Fpages (Flexpages) are regions of the virtual address space. An fpage consists of all pages actually mapped in this region. The minimal fpage size is the minimal hardware-page size.

An fpage of size 2^s has a 2^s-aligned base address b, i.e. $b \bmod 2^s = 0$. An fpage with base address b and size 2^s is denoted by the 64-bit value

$$b + 4s.$$

On R4x00 processors, the smallest possible value for s is 12, since the hardware page size is 4K.

1.2.4 Messages

S :: snd ; EMPTY .

R :: rcv ; EMPTY .

EMPTY :: .

S R message: rcv fpage option ,
size dope ,
S R msg dope ,
S R mwords ,
S R string dopes .

rcv fpage option: rcv fpage:fpage ;
zero:word.

size dope: reserved:byte ,
string dope number:5bits , $= S$
mwords number:19bits . $= W$

snd R msg dope: undefined:byte ,
string dope number:5bits , $= s \quad s \leq S$
mwords number:19bits . $= w \quad w \leq W$

rcv msg dope: undefined:word .

snd R mwords: $w \times$ send receive word ,
$(W - w) \times$ receive word ;

$m \times$ snd fpage receive double word , $2m \leq w$
$w - 2m \times$ send receive words ,
$(W - w) \times$ receive word .

rcv mwords: $W \times$ receive word .

snd R string dopes: $s \times$ snd R string dope ,
$(S - s) \times$ R string dope .

rcv string dopes: $S \times$ rcv string dope .

snd rcv string dope: snd addr:word ,
snd size:word , $\leq$ 4MB
rcv addr:word ,
rcv size:word . $\leq$ 4MB

snd string dope: snd addr:word ,
snd size:word , $\leq$ 4MB
undefined:word ,
undefined:word .

rcv string dope: undefined:word ,
undefined:word ,
rcv addr:word ,
rcv size:word . $= s_r \qquad s_r \leq$ 4MB

snd map fpage: grant flag:1bit ,
write flag:1bit ,
snd base:30bits ,
snd fpage:fpage .

1.3 μ-Kernel Calls

ipc (dest option, snd descriptor option, rcv descriptor option, timeouts)
$\rightarrow$ (source option, result code)

CALL
(dest, snd descriptor, closed rcv descriptor, timeouts)
$\rightarrow$ (dest option, result code)

SEND/RECEIVE
(dest, snd descriptor, open rcv descriptor, timeouts)
$\rightarrow$ (source option, result code)

SEND
(dest, snd descriptor, –nil– , timeouts) $\rightarrow$ ($\sim$, result code)

RECEIVE FROM
(source, –nil– , closed rcv descriptor, timeouts)
$\rightarrow$ (source option, result code)

RECEIVE
($\sim$, –nil– , open rcv descriptor, timeouts)
$\rightarrow$ (source option, result code)

RECEIVE INTR
(intr, –nil– , closed rcv descriptor, timeouts)
$\rightarrow$ (source option, result code)

SLEEP
(–nil– , –nil– , closed rcv descriptor, timeouts) $\rightarrow$ ($\sim$, result code)

id_nearest (dest id) $\rightarrow$ (nearest id)

fpage_unmap (fpage, map mask) → ()

thread_switch (dest) → ()

lthread_ex_regs (lthread no, SP, IP, excpt, pager)
→ (FLAGS, SP, IP, excpt, pager)

MIPS Implementation Note: Added exception handler identifier (see section 2.5), and removed preempter which is currently not supported in L4/MIPS

thread_schedule (dest, prio, timeslice, ext preempter)
→ (prio, timeslice, state, ext preempter, partner, time)

MIPS Implementation Note: thread_schedule is not currently implemented in L4/MIPS

task_new (dest task id, mcp/new chief, SP, IP, pager id, excpt id) → (new task id)

MIPS Implementation Note: Added an exception handler identifier (see section 2.5)

Chapter 2

L4/MIPS

L4/R4x00

2.1 Implementation Notes

What follows is a list of implementation details of the current L4/MIPS implementation. It is here to serve as a quick reference as to what may or may not be implemented for those that are familiar with L4/x86.

2.1.1 Cache

The R4600 has 16KB data cache and 16KB instruction cache. Both are two-way associative, virtually indexed with physical tags. The data cache has either a write-through or write-back policy.

To avoid aliasing problems, shared memory regions must lie at the same offset from a 8KB boundary in the virtual address space.

Write-back caching is not used in the current version (other than in the L4 kernel itself) as there is potential for the cache to write-back data from a recycled page no longer used at its original address, ie when it appears elsewhere in the cache.

I envisage adding a MIPS specific system call in the future to perform cache management functions. This will allow write-back caching to once again be re-enabled.

2.1.2 IPC

- Granting is not supported.
- Sending multiple fpages is supported in registers only, i.e. up to 3 valid fpages plus the terminating nil fpage. Sending fpages in memory based messages is not supported.
- Dwords sent in memory based messages are 64-bit, not 32-bit as in L4/x86. This allows sending direct messages of up to 4MB in size.
- Indirect strings can be up to 4MB in size.

2.1.3 Scheduling

- thread_schedule is not implemented.
- The current scheduler uses a simple round robin scheme with no priorities.
- Internal and external preempters are not supported.
- Constant interrupts will prevent other threads from running.

2.1.4 σ_0

- Multiple mappings of the same physical frame is not supported.
- The RPC protocol is slightly different, see section 2.8 for details.

2.1.5 Exceptions

Exceptions are handled using IPC. Each thread has it's own exception handling thread, see section 2.5 for details.

2.2 Notational conventions

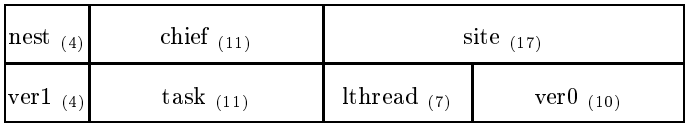
$\sim$	If this refers to an input parameter, its value is meaningless. If it refers to an output parameter, its value is undefined.
a0,**a1**...	denote the processor's general registers. Note that the SGI 64-bit ABI register names are used.

2.3 Data Types

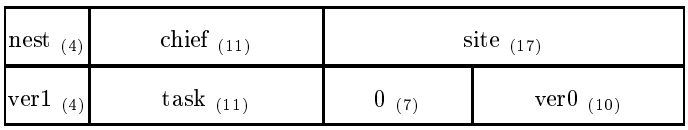
2.3.1 Unique Ids

Unique ids identify tasks, threads and hardware interrupts. Each unique id is a 64-bit value which is unique in time. An unique id in R4x00 format consists of a single 64-bit word:

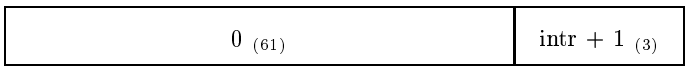
thread id



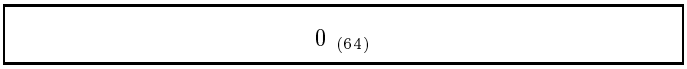
task id



interrupt id



nil id



invalid id

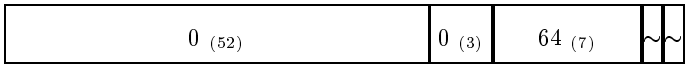
2.3.2 Fpages

Fpages (Flexpages) are regions of the virtual address space. An fpage consists of all pages actually mapped in this region. The minimal fpage size is 4 K, the minimal hardware-page size.

An fpage of size 2^s has a 2^s-aligned base address b, i.e. $b \bmod 2^s{=}0$. On the R4x00 processors, the smallest possible value for s is 12, since hardware pages are at least 4K. The complete user address space (base address 0, size $2^{64} - K$, where K is the size of the kernel area) is denoted by $b = 0, s = 64$. An fpage with base address b and size 2^s is denoted by a 64-bit word:

fpage$(b, 2^s)$

fpage$(0, 2^{64} - K)$



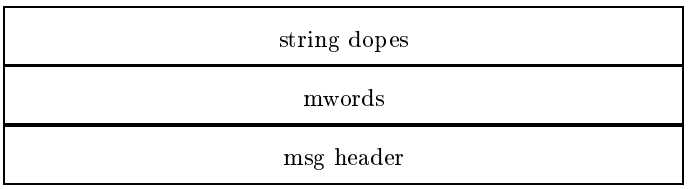
MIPS Implementation Note: The user address space on the R4600 is one terabyte (2^{40}) beginning at 0x0. Values of $s \geq 40$ are equivalent, however if the intention is to specifiy the whole address space one should of course use $s = 64$ for future compatibility.

2.3.3 Messages

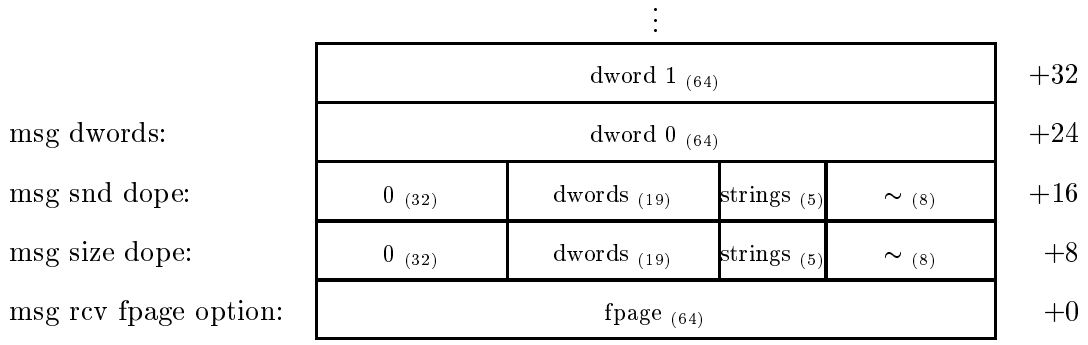
A message contains between 2^6 and $2^{22} + 2^6$ bytes of in-line data (**mwords**). The first 64 bytes (eight dwords) are transfered via registers, the (optional) remainder is contained in a dword-aligned memory buffer pointed to by a *message descriptor*. Every successful IPC operation will always copy at least eight dwords to the receiver.

The buffer pointed to by the optional message descriptor contains a 3 dword message header, followed by a number of mwords, followed by a number of *string dopes*. The number of mwords (in 64-bit dwords, excluding those copied in registers) and string dopes is specified in the message header.

message:



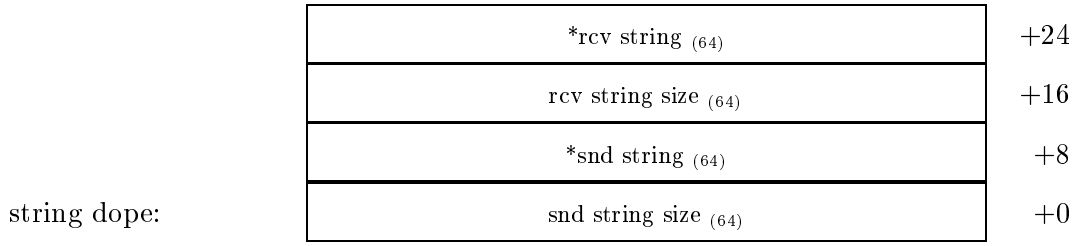
The beginning of the message buffer has the following format:



The *receive fpage* describes the address range in which the caller is willing to accept fpage mappings or grants in the receive part (if any) of the IPC. The *size dope* defines the size (in dwords) of the mword buffer (and hence the offset of the string dopes from the end of the header), and the number of string dopes.

The *send dope* specifies how many dwords and strings are actually to be sent. (Specifying send dope values less than the size dope values makes sense when the caller is willing to receive more data than sending.)

Strings are out-of-line by-value data. Their size and location is specified by the corresponding string dopes. The string dope format is:

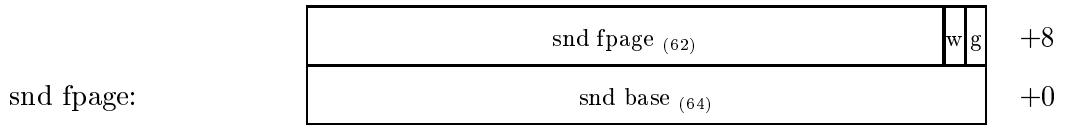


The first part of the string dope specifies the size and location of the string the caller wants sent to the destination, while the second part specifies the size and location of a buffer where the caller is willing to receive a string. **Note** that strings do not have to be aligned, and that their size is specified in *bytes*.

The in-line part of the message consists of the eight dwords passed in registers followed by any dwords specified by the message descriptor. This part consists of optional *fpage descriptors* followed by by-value data. If the receiver of an IPC has specified a valid *receive fpage*, the kernel will interpret each pair of dwords of the in-line part as fpage descriptors, until an invalid descriptor is encountered. This and any further dwords are then passed by value.

MIPS Implementation Note: Presently at most three fpages can be passed on the R4600.

The format of an fpage descriptor is:



The first word contains the address of the *hot spot*, while the second word describes the sender's fpage in the format given in Sect. 2.3.2. The g-bit, if set, indicates that the fpage is to be *granted* to the receiver, otherwise it is just *mapped*. The w-bit indicates whether the receiver will be given write or read-only access to the address-space region.

Each fpage specified by the sender is mapped individually into the address-space window specified by the receiver's *receive fpage*. If the sender and receiver specify different fpage sizes, the hot-spot specification is used to determine how the mapping between the two different size fpages occurs: If 2^s is the size of the larger, and 2^t the size of the smaller fpage, then the larger fpage can be thought as being tiled by 2^{s-t} fpages of the smaller size. One of these is uniquely identified as containing the hot spot address (mod 2^s). This is the fpage which will actually be mapped.

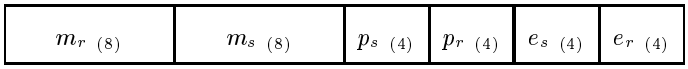
MIPS Implementation Note: The current message headers are not as compact as they could be. They will be optimised in a future version.

2.3.4 Timeouts

Timeouts are used to control ipc operations. The *send timeout* determines how long ipc should try to send a message. If the specified period is exhausted without that message transfer could start, ipc fails. The *receive timeout* specifies how long ipc should wait for an incoming message. Both timouts specify the maximum period of time *before message transfer starts*. Once started, message transfer is no longer influenced by send or receive timeout.

Pagefaults occuring during ipc are controlled by *send* and *receive pagefault timeout*. A pagefault is translated to an RPC by the kernel. In the case of a pagefault in the receiver's address space, the corresponding RPC to the pager uses *send pagefault timeout* (specified by the sender) for both send and receive timeout. In the case of a pagefault in the sender's address space, *receive pagefault timeout* specified by the receiver is taken.

Besides the special timeouts 0 (do not wait at all) and ∞ (wait forever), periods from 1 μs up to approximately 19 hours can be specified. The complete quadruple is packed into one 32-bit word:



Note that for efficiency reasons the highest bit of any mantissa m *must* be 1, except for m=0.

$$\text{snd timeout} = \begin{cases} \infty & \text{if } e_s{=}0 \\ 4^{15-e_s} m_s \ \mu s & \text{if } e_s{>}0 \\ 0 & \text{if } m_s{=}0,\ e_s{\neq}0 \end{cases}$$

$$\text{rcv timeout} = \begin{cases} \infty & \text{if } e_r{=}0 \\ 4^{15-e_r} m_r \ \mu s & \text{if } e_r{>}0 \\ 0 & \text{if } m_r{=}0,\ e_r{\neq}0 \end{cases}$$

$$\text{snd pagefault timeout} = \begin{cases} \infty & \text{if } p_s{=}0 \\ 4^{15-p_s}\,\mu s & \text{if } 0{<}p_s{<}15 \\ 0 & \text{if } p_s{=}15 \end{cases}$$

$$\text{rcv pagefault timeout} = \begin{cases} \infty & \text{if } p_r{=}0 \\ 4^{16-p_r}\,\mu s & \text{if } 0{<}p_r{<}15 \\ 0 & \text{if } p_r{=}15 \end{cases}$$

approximate timeout ranges		
e_s, e_r, p_s, p_r	snd/rcv timeout	pf timeout
0	∞	∞
1	256 s ... 19 h	256 s
2	64 s ... 55 h	64 s
3	16 s ... 71 m	16 s
4	4 s ... 17 m	4 s
5	1 s ... 4 m	1 s
6	262 ms ... 67 s	256 ms
7	65 ms ... 17 s	64 ms
8	16 ms ... 4 s	16 ms
9	4 ms ... 1 s	4 ms
10	1 ms ... 261 ms	1 ms
11	256 μs ... 65 ms	256 μs
12	64 μs ... 16 ms	64 μs
13	16 μs ... 4 ms	16 μs
14	4 μs ... 1 ms	4 μs
15	1 μs ... 255 μs	0
$m{=}0, e{>}0$	0	—

2.4 μ-Kernel Calls

System calls are implemented using the `syscall` instruction in conjunction with the **AT** register which is set to the system call number prior to the call. All registers, unless otherwise stated, are returned undefined after the system call except for the stack pointer **sp**.

This section describes the 7 system calls of L4:

- ipc **AT** 0
- id_nearest **AT** 2
- fpage_unmap **AT** 1
- thread_switch **AT** 4
- thread_schedule **AT** 5
- lthread_ex_regs **AT** 6
- task_new **AT** 7

MIPS Implementation Note: The system call numbers will be cleaned up some time in the future.

ipc

snd descriptor	**a0**		**a0**	$\sim$		
rcv descriptor	**a1**		**a1**	$\sim$		
timeouts	**a2**		**a2**	$\sim$		
dest id	**a4**	— **AT** 0x0⟶	**a4**	$\sim$		
waiting for id / 0	**a5**		**a5**	$\sim$		
virtual sender id / $\sim$	**a6**		**a6**	$\sim$		
msg.w0	**s0**		**s0**	*msg.w0*	/	$\sim$
msg.w1	**s1**		**s1**	*msg.w1*	/	$\sim$
msg.w2	**s2**		**s2**	*msg.w2*	/	$\sim$
msg.w3	**s3**		**s3**	*msg.w3*	/	$\sim$
msg.w4	**s4**		**s4**	*msg.w4*	/	$\sim$
msg.w5	**s5**		**s5**	*msg.w5*	/	$\sim$
msg.w6	**s6**		**s6**	*msg.w6*	/	$\sim$
msg.w7	**s7**		**s7**	*msg.w7*	/	$\sim$
$\sim$	**v0**		**v0**	*msgdope + cc*	/	*cc*
$\sim$	**v1**		**v1**	*source id*		

This is the basic system call for inter-process communication and synchronisation. It may be used for intra- as inter-address-space communication. All communication is synchronous and unbuffered: a message is transferred from the sender to the recipient if and only if the recipient has invoked a corresponding ipc operation. The sender blocks until this happens or a period specified by the sender elapsed without that the destination became ready to receive.

Ipc can be used to copy data as well as to *map* or *grant* fpages from the sender to the recipient. For the description of messages see section 2.3.3.

64-byte messages (plus 64-bit sender id) can be transferred solely via the registers and are thus specially optimised. If possible, short messages should therefore be reduced to 64-byte messages.

A single ipc call combines an optional send operation with an optional receive operation. Whether it includes a send and/or a receive is determined by the actual parameters. If the send or receive address is specified as *nil* (0xFFFFFFFFFFFFFFFF), the corresponding operation is skipped.

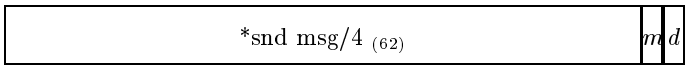
No time is required for the transition between send and receive phase of one ipc operation (i.e., the destination can reply with a timeout of zero).

Parameters

snd descriptor *"nil"*

Ipc does not include a send operation.

"mem"



Ipc includes sending a message to the destination specified by *dest id*. *snd msg must point to a valid message. The first 8 64-bit words of the message (*msg.w0* to *msg.w7*) are *not* taken from the message data structure but must be contained in registers **s0** through **s7**.

snd descriptor *"reg"*

Ipc includes sending a message to the destination specified by *dest id*. The message consists solely of the 8 64-bit words *msg.w0* to *msg.w7* in registers **s0** through **s7**.

m=0 Value-copying send operation; the dwords of the message are simply copied to the recipient.

m=1 Fpage-mapping send operation. The dwords of the message to be sent are treated as 'send fpages'. The described fpages are mapped or granted (depending on the *g* bit in the fpage descriptor) /cbend into the recipient's address space. Mapping/granting stops when either the end of the dwords is reached or when an invalid fpage denoter is found, in particular 0. The send fpage descriptors and all potentially following words are also transferred by simple copy to the recipient. Thus a message may contain some fpages and additional value parameters. The recipient can use the received fpage descriptors to determine what has been mapped or granted into its address space, including location and access rights.

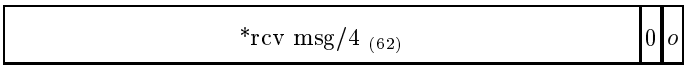
d=0 Normal send operation. The recipient gets the true sender id.

d=1 Deceiving send operation. A chief can specify the *virtual sender id* which the recipient should get instead of the chief's id. The *virtual sender id* parameter contained in **a6** is only required if *d*=1. Recall that deceiving is secure, since only *direction-preserving deceit* is possible. If the specified *virtual-sender id* does not fulfil this constraint, the send operation works like *d*=0.

rcv descriptor *"nil"*

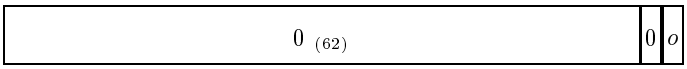
Ipc does not include a receive operation.

"mem"



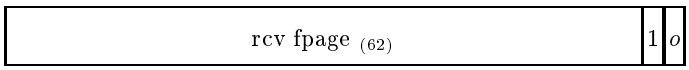
Ipc includes receiving a message or waiting to receive a message. *rcv msg must point to a valid message. The 8 64-bit words of the received message (*msg.w0* to *msg.w7*) are *not* stored in the message data structure but are returned in registers **s0** through **s7**.

"reg"



Ipc includes receiving a message or waiting to receive a message. However, only messages up to 8 64-bit words *msg.w0* to *msg.w7* are accepted. The received message is returned in registers **s0** through **s7**.

rcv descriptor *"rmap"*



Ipc includes receiving a message or waiting to receive a message. However, only send-fpage messages or up to 8 64-bit words *msg.w0* to *msg.w7* are accepted. The received message is returned in registers **s0** through **s7**. If a map message is received, "rcv fpage" describes the receive fpage (instead of "rcv fpage option" in a memory message buffer). Thus fpages can also be received without a message buffer in memory.

o **MIPS Implementation Note:** *o* is currently not used in L4/MIPS. See *waiting for id* below.

o=0 Only messages from the thread specified as *dest id* are accepted ("closed wait"). Any send operation from a different thread (or hardware interrupt) will be handled exactly as if the actual thread would be busy.

o=1 Messages from any thread will be accepted ("open wait"). If the actual thread is associated to a hardware interrupt, also messages from this hardware interrupt can arrive.

waiting for id "Open" and "closed" waits are not specified in bit zero of *rcv descriptor* as in L4/x86. Instead if *waiting for id*= 0 the receive is an open wait, else the wait is for the thread id specified. Note that operation when waiting for any thread id other than *dest id* is undefined.

MIPS Implementation Note: This will likely change back to the way L4/x86 does it in the future using *o*.

dest id $\neq nil$ Sending is directed to the specified thread, if it resides in the sender's clan. If the destination is outside the sender's clan, the message is sent to the sender's chief. If the destination is in an inner clan (a clan whose chief resides in the sender's clan), it is redirected to that chief. (See also 'nchief' operation.) If no send part was specified (*snd descriptor* $=nil$), *dest id* specifies the source from which messages can be received. (However recall that the receive restriction is only effective if $o = 0$.)

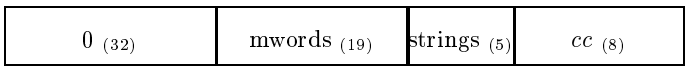
$=nil$ (nil=0) Although specifying nil as the destination for a send operation is illegal (error: 'destination not existent'), it can be legally specified for a receive-only operation. In this case, ipc will not receive any message but will wait the specified *rcv timeout* and then terminate with error code 'receive timeout'. (However recall that the receive restriction is only effective if $o = 0$.)

source id If a message was received this is the id of its sender. (If a hardware interrupt was received this is the interrupt id.) The parameter is undefined if no message was received.

msg.w0 ... w7 *"snd"* First 8 64-bit words of message to be sent. These message words are taken directly from registers **s0** through **s7**. *They are not read from the message data structure.*

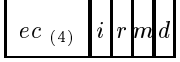
"rcv" First 8 64-bit words of received message, undefined if no message was received. *These message words are available only in registers **s0** through **s7**.* The μ-kernel does not store it in the receive message buffer. The user program may store it or use it directly in the registers.

msg.dope + cc



Message dope describing received message. If no message was received, only *cc* is delivered. *The dope word of the received message is available only in register **v0**.* The μ-kernel does not store it in the receive message buffer. The user program may store it or use it directly in the register. (Note that the lowermost 8 bits of msg dope and size dope in the message data structure are undefined. So it is legal to store **v0** in the msg-dope field, even if $cc \neq 0$.)

cc



d=0 The received message is transferred directly ("undeceived") from *source id*.

d=1 The received message is "deceived" by a chief. *source id* is the virtual source id which was specified by the sending chief.

m=0 The received message did not contain fpages.

m=1 The sender mapped or granted fpages. The sender's fpage descriptors were also (besides mapping/granting) transferred as mwords.

r=0 The received message was directed to the actual recipient, not redirected to a chief. I.e. sender and receiver a part of the same clan. The i-bit has no meaning in this case and is zero.

r=1 The received message was redirected to the chief which was next on the path to the true destination. Sender and addressed recipient belong to different clans.

i=0 If r=1: the received message comes from outside the own clan.

i=1 If r=1: the received message comes from an inner clan.

ec

= 0 *ok*: the optional send operation was successful, and if a receive operation was also specified (*rcv descriptor* $\neq$*nil*) a message was also received correctly.

$\neq$ 0 If ipc fails the completion code is in the range 0x10...0xF0. If the send operation fails, ipc is terminated without attemtping any receive operation. s specifies whether the error occurred during the receive ($s = 0$) operation or during the send ($s = 1$) operation:

1 *Non-existing* destination or source.

$2 + s$ *Timeout*.

$4 + s$ *Cancelled* by another thread (system call lthread_ex_regs).

$6 + s$ *Map failed* due to a shortage of page tables.

$8 + s$ *Send pagefault timeout*.

$A + s$ *Receive pagefault timeout*.

$C + s$ *Aborted* by another thread (system call lthread_ex_regs or task_new).

$E + s$ *Cut* message. Potential reasons are (a) the recipient's mword buffer is too small; (b) the recipient does not accept enough strings; (c) at least one of the recipient's string buffers is too small.

1...5 The respective operation was terminated before a real message transfer started. No partner was directly involved.

6...F The respective operation was terminated while a message transfer was running. The message transfer was aborted. The current partner (sender or receiver) was involved and received the corresponding error code. It is not defined which parts of the message are already transferred and which parts are not yet transferred. The source id returned to the receiver is also undefined.

timeouts This 32-bit word specifies all 4 timeouts, the quadruple (*snd, rcv, snd pf, rcv pf*). For A detailed description see section 2.3.4. Frequently used values are

	snd	rcv	snd pf	rcv pf
0x00000000	∞	∞	∞	∞
0x00000001	0	∞	∞	∞
0x00000011	0	0	∞	∞

"snd" If the required send operation cannot start transfer data within the specified time, ipc is terminated and fails with completion code 'send timeout' (0x18). If ipc does not include a send operation, this parameter is meaningless.

"rcv" If ipc includes a receive operation and no message transfer starts within the specified time, ipc is terminated and fails with completion code 'receive timeout' (0x20). If ipc does not include a receive operation, this parameter is meaningless.

"spf" If during sending data a pagefault *in the receiver's address space* occurs, *snd pf* specified by the sender is used as send and receive timeout for the pagefault RPC.

"rpf" If during receiving data a pagefault *in the sender's address space* occurs, *rcv pf* specified by the receiver is used as send and receive timeout for the pagefault RPC.

Basic Ipc Types

CALL

**snd msg / 0*	**a0**		**a0**	$\sim$
**rcv msg / 0*	**a1**		**a1**	$\sim$
timeouts	**a2**		**a2**	$\sim$
dest id	**a4**	**– AT** 0x0$\longrightarrow$	**a4**	$\sim$
dest id	**a5**		**a5**	$\sim$
msg.w0	**s0**		**s0**	*msg.w0*
msg.w1	**s1**		**s1**	*msg.w1*
msg.w2	**s2**		**s2**	*msg.w2*
msg.w3	**s3**		**s3**	*msg.w3*
msg.w4	**s4**		**s4**	*msg.w4*
msg.w5	**s5**		**s5**	*msg.w5*
msg.w6	**s6**		**s6**	*msg.w6*
msg.w7	**s7**		**s7**	*msg.w7*
$\sim$	**v0**		**v0**	*msgdope + cc*
$\sim$	**v1**		**v1**	*dest id*

This is the usual blocking RPC. *snd msg* is sent to *dest id* and the invoker waits for a reply from *dest id*. Messages from other sources are not accepted. Note that since the send/receive transition needs no time, the destination can reply with *snd timeout* = 0.

This operation can also be used for a server with one dedicated client. It sends the reply

to the client and waits for the client's next order.

SEND/RECEIVE

**snd msg / 0*	**a0**			**a0**	∼
**rcv msg / 0*	**a1**			**a1**	∼
timeouts	**a2**			**a2**	∼
dest id	**a4**	**−AT** 0x0→		**a4**	∼
0	**a5**			**a5**	∼
msg.w0	**s0**			**s0**	*msg.w0*
msg.w1	**s1**			**s1**	*msg.w1*
msg.w2	**s2**			**s2**	*msg.w2*
msg.w3	**s3**			**s3**	*msg.w3*
msg.w4	**s4**			**s4**	*msg.w4*
msg.w5	**s5**			**s5**	*msg.w5*
msg.w6	**s6**			**s6**	*msg.w6*
msg.w7	**s7**			**s7**	*msg.w7*
∼	**v0**			**v0**	*msgdope + cc*
∼	**v1**			**v1**	*source id*

snd msg is sent to *dest id* and the invoker waits for a reply from any source. This is the standard server operation: it sends a reply to the actual client and waits for the next order which may come from a different client.

SEND

**snd msg / 0*	**a0**			**a0**	∼
0xFFFFFFFFFFFFFFFF	**a1**			**a1**	∼
timeouts	**a2**			**a2**	∼
dest id	**a4**	**−AT** 0x0→		**a4**	∼
msg.w0	**s0**			**s0**	∼
msg.w1	**s1**			**s1**	∼
msg.w2	**s2**			**s2**	∼
msg.w3	**s3**			**s3**	∼
msg.w4	**s4**			**s4**	∼
msg.w5	**s5**			**s5**	∼
msg.w6	**s6**			**s6**	∼
msg.w7	**s7**			**s7**	∼
∼	**v0**			**v0**	*msgdope + cc*

snd msg is sent to *dest id*. There is no receive phase included. The invoker continues

working after sending the message.

RECEIVE FROM

0xFFFFFFFFFFFFFFFF	**a0**		**a0**	∼
**rcv msg / 0*	**a1**		**a1**	∼
timeouts	**a2**		**a2**	∼
dest id	**a4**	**– AT** 0x0⟶	**a4**	∼
dest id	**a5**		**a5**	∼
∼	**s0**		**s0**	*msg.w0*
∼	**s1**		**s1**	*msg.w1*
∼	**s2**		**s2**	*msg.w2*
∼	**s3**		**s3**	*msg.w3*
∼	**s4**		**s4**	*msg.w4*
∼	**s5**		**s5**	*msg.w5*
∼	**s6**		**s6**	*msg.w6*
∼	**s7**		**s7**	*msg.w7*
∼	**v0**		**v0**	*msgdope + cc*
∼	**v1**		**v1**	*dest id*

This operation includes no send phase. The invoker waits for a message from *source id*. Messages from other sources are not accepted. Note that also a hardware interrupt might be specified as source.

RECEIVE

0xFFFFFFFFFFFFFFFF	**a0**		**a0**	∼
**rcv msg / 0*	**a1**		**a1**	∼
timeouts	**a2**		**a2**	∼
dest id	**a4**	**– AT** 0x0⟶	**a4**	∼
0	**a5**		**a5**	∼
∼	**s0**		**s0**	*msg.w0*
∼	**s1**		**s1**	*msg.w1*
∼	**s2**		**s2**	*msg.w2*
∼	**s3**		**s3**	*msg.w3*
∼	**s4**		**s4**	*msg.w4*
∼	**s5**		**s5**	*msg.w5*
∼	**s6**		**s6**	*msg.w6*
∼	**s7**		**s7**	*msg.w7*
∼	**v0**		**v0**	*msgdope + cc*
∼	**v1**		**v1**	*source id*

This operation includes no send phase. The invoker waits for a message from any source

(including a hardware interrupt).

RECEIVE INTR

0xFFFFFFFFFFFFFFFF	**a0**			**a0**	∼
**rcv msg / 0*	**a1**			**a1**	∼
timeouts	**a2**			**a2**	∼
intr + 1	**a4**	**−**	**AT** 0x0→	**a4**	∼
intr + 1	**a5**			**a5**	∼
∼	**s0**			**s0**	∼
∼	**s1**			**s1**	∼
∼	**s2**			**s2**	∼
∼	**s3**			**s3**	∼
∼	**s4**			**s4**	∼
∼	**s5**			**s5**	∼
∼	**s6**			**s6**	∼
∼	**s7**			**s7**	∼
∼	**v0**			**v0**	*msgdope + cc*
∼	**v1**			**v1**	*intr + 1*

This operation includes no send phase. The invoker waits for an interrupt message coming from interrupt source *intr*. Note that interrupt messages come *only* from the interrupt which is currently associated with this thread.

The *intr* parameter is only evaluated if *rcv timeout* = 0 is specified, see 'associate intr'.

ASSOCIATE INTR

0xFFFFFFFFFFFFFFFF	**a0**			**a0**	∼
**rcv msg / 0*	**a1**			**a1**	∼
rcv timeout = 0	**a2**			**a2**	∼
intr + 1	**a4**	**−**	**AT** 0x0→	**a4**	∼
intr + 1	**a5**			**a5**	∼
∼	**s0**			**s0**	∼
∼	**s1**			**s1**	∼
∼	**s2**			**s2**	∼
∼	**s3**			**s3**	∼
∼	**s4**			**s4**	∼
∼	**s5**			**s5**	∼
∼	**s6**			**s6**	∼
∼	**s7**			**s7**	∼
∼	**v0**			**v0**	*msgdope + cc*
∼	**v1**			**v1**	*intr + 1*

The *intr* parameter is evaluated if *rcv timeout* = 0 is specified. If no (currently associated) interrupt is pending, the current thread is (1) detached from its currently associated interrupt (if any) and (2) associated to the specified interrupt provided that this one is free, i.e. not associated to another thread. If the association succeeds, the completion code is *receive timeout* (0x20) and no interrupt is received.

If an interrupt from the currently associated interrupt was pending, this one is delivered together with completion code *ok* (0x00); the interrupt association is *not* modified. If the

requested new interrupt is already associated to another thread or is not existing, completion code *non existing* (0x10) is delivered and the interrupt association is not modified.

Getting rid of an associated interrupt without associating a new one is done by issuing a receive from *nilthread* (0) with *rcv timeout* = 0.

SLEEP

0xFFFFFFFFFFFFFFFF	**a0**			**a0**	∼
0	**a1**			**a1**	∼
timeouts	**a2**			**a2**	∼
0	**a4**	**−**	**AT** 0x0→	**a4**	∼
0	**a5**			**a5**	∼
∼	**s0**			**s0**	∼
∼	**s1**			**s1**	∼
∼	**s2**			**s2**	∼
∼	**s3**			**s3**	∼
∼	**s4**			**s4**	∼
∼	**s5**			**s5**	∼
∼	**s6**			**s6**	∼
∼	**s7**			**s7**	∼
∼	**v0**			**v0**	*cc = 0x20*
∼	**v1**			**v1**	∼

This operation includes no send phase. Since *nil* (0) is specified as source, no message can arrive and the ipc will be terminated with 'receive timeout' after the time specified by the *rcv-timeout* parameter is elapsed.

id_nearest

dest id	**a0**			**a0**	$\sim$
$\sim$	**v0**			**v0**	*type*
$\sim$	**v1**			**v1**	*nearest id*
		— **AT** 0x2→			

If *nil* is specified as destination, the system call delivers the uid of the current thread. Otherwise, it delivers the nearest partner which would be engaged when sending a message to the specified destination. If the destination does not belong to the invoker's clan, this call delivers the chief that is nearest to the invoker on the path from the invoker to the destination.

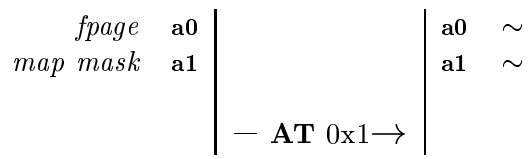
- If the destination resides outside the invoker's clan, it delivers the invoker's own chief.
- If the destination is inside a clan or a clan nesting whose chief C is direct member of the invoker's clan, the call delivers C.
- If the destination is a direct member of the invoker's clan, the call delivers the destination itself.
- If the destination is *nil*, the call delivers the current thread's id.

Concluding: nchief (*dest id* $\neq$ *nil*) delivers exactly that partner to which the kernel would physically send a message which is targeted to *dest id*. On the other hand, a message from *dest id* would physically come from exactly this partner.

Parameters

dest id		Id of the destination.
type		Note that the *type* values correspond exactly to the completion codes of ipc.
	=0	Destination resides in the same clan. *dest id* is delivered as *nearest id*.
	=C	Destination is in an inner clan. The chief of this clan or clan nesting is delivered as *nearest id*.
	=4	Destination is outside the invoker's clan. The invoker's chief is delivered as *nearest id*.
nearest id		Either the current thread's id or the id of the nearest partner towards *dest id*.

fpage_unmap

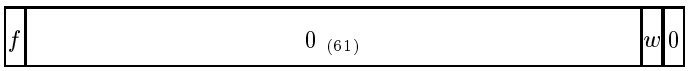


The specified *fpage* is unmapped in all address spaces into which the invoker mapped it directly or indirectly.

Parameters

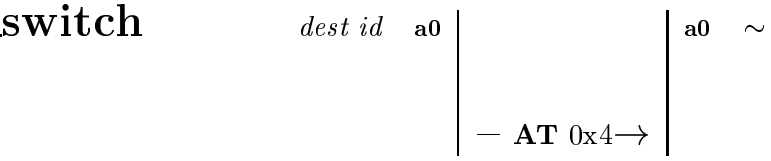
fpage Fpage to be unmapped.

map mask



- w=0 Fpage will partially unmapped. Already read/write mapped parts will be set to read only. Read only mapped parts are not affected.
- w=1 Fpage will be completely unmapped.
- f=0 Unmapping happens in all address spaces into which pages of the specified fpage have been mapped directly or indirectly. The *original* pages in the own task remain mapped.
- f=1 Additionally, also the original pages in the own task are unmapped (flushing).

thread_switch

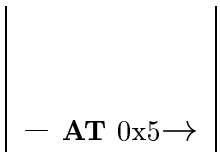


The invoking thread releases the processor (non-preemtively) so that another ready thread can be processed.

Parameters

dest id	$=nil$	(=0) Processing switches to an undefined ready thread which is selected by the scheduler. (It might be the invoking thread.) Since this is "ordinary" scheduling, the thread gets a new timeslice.
	$\neq nil$	If *dest id* is ready, processing switches to this thread. In this "extraordinary" scheduling, the invoking thread donates its remaining timeslice to the destination thread. (This one gets the donation additionally to its ordinary scheduled timeslices.) If the destination thread is not ready, the system call operates as described for *dest id* $=nil$.

thread_schedule



MIPS Implementation Note: thread_schedule is yet to be implemented in L4/MIPS

The system call can be used by schedulers to define the *priority*, *timeslice length* and *external preempter* of other threads. Furthermore, it delivers thread states. Note that due to security reasons thread state information must be retrieved through the appropriate scheduler.

The system call is only effective if the current priority of the specified destination is less or equal than the current task's *maximum controlled priority (mcp)*.

Parameters

dest id — Destination thread id. The destination thread must currently exist and run on a priority level less than or equal to the current thread's *mcp*. Otherwise, the destination thread is not affected by this system call and all result parameters except *old param word* are undefined.

param word

valid

m_t (8)	e_t (4)	0 (4)	small (8)	prio (8)

prio — New priority for destination thread. Must be less than or equal to current thread's *mcp*.

small — (Only effective for Pentium.) Sets the *small address space number* for the addressed *task*. On Pentium, small address spaces from 1 to 127 currently available. A value of 0 or 255 in this field does not change the current setting for the task. This field is currently ignored for 486 and PPro.

m_t, e_t — New timeslice length for the destination thread. The timeslice quantum is encoded like a timeout: $4^{15-e_t} m_t\ \mu s$.
The kernel rounds this value up towards the nearest possible value. Thus the timeslice granularity can be determined by trying to set the timeslice to 1 μs. However note that the timeslice granularity may depend on the priority.
Timeslice length 0 ($m_t = 0, e_t \neq 0$) is always a possible value. It means that the thread will get no ordinary timeslice, i.e. is blocked. However, even a blocked thread may execute in a timeslice donated to it by ipc.

"inv" — (0xFFFFFFFF) The current priority and timeslice length of the thread is not modified.

ext preempter valid — Defines the external preempter for the destination thread. (Nilthread is a valid id.)

"inv" — (0xFFFFFFFF,∼) The current external preempter of the thread is not changed.

old param word valid

m_t (8)	e_t (4)	ts (4)	∼ (8)	prio (8)

prio — Old priority of destination thread.

m_t, e_t — Old timeslice length of the destination thread: $4^{15-e_t} m_t\ \mu s$.

$ts =$ — Thread state:

- $0 + k$ — *Running.* The thread is ready to execute at user-level.
- $4 + k$ — *Sending.* A user-invoked ipc send operation currently transfers an outgoing message.
- $8 + k$ — *Receiving.* A user-invoked ipc receive operation currently receives an incoming message.
- C — *Waiting* for receive. A user-invoked ipc receive operation currently waits for an incoming message.
- D — *Pending* send. A user-invoked ipc send operation currently waits for the destination (recipient) to become ready to receive.
- E — Reserved.
- F — *Dead.* The thread is unable to execute.

$k =$

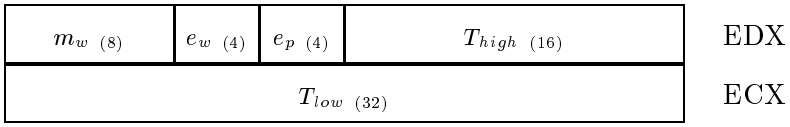
- 0 — *Kernel inactive.* The kernel does not execute an automatic RPC for the thread.
- 1 — *Pager.* The kernel executes a pagefault RPC to the thread's pager.
- 2 — *Internal preempter.* The kernel executes a preemption RPC to the thread's internal preempter.
- 3 — *External preempter.* The kernel executes a preemption RPC to the thread's external preempter.

"inv" — (0xFFFFFFFF) The addressed thread does either not exist or has a priority which exceeds the current thread's *mcp*. All other return parameters are undefined (∼).

old ext preempter — Old external preempter of the destination thread.

partner — Partner of an active user-invoked ipc operation. This parameter is only valid, if the thread's user state is *sending, receiving, pending* or *waiting* (4...D). An invalid thread id (0xFFFFFFFF,∼) is delivered if there is no specific partner, i.e. if the thread is in an open receive state.

time



T Cpu time (48-bit value) in microseconds which has been consumed by the destination thread.

m_w, e_w Current user-level wakeup of the destination thread, encoded like a timeout. The value denotes the still remaining timeout interval. Valid only if the user state is *waiting* (C) or *pending* (D).

e_p Effective pagefault wakeup of the destination thread, encoded like a 4-bit pagefault timeout. The value denotes the still remaining timeout interval. Valid only if the kernel state is *pager* ($k = 1$).

lthread_ex_regs

lthread no	**a0**			**a0**	$\sim$
IP	**a1**			**a1**	*old IP*
SP	**a2**			**a2**	*old SP*
excpt id	**a3**	— **AT** 0x6→		**a3**	*old excpt id*
pager id	**a4**			**a4**	*old pager id*

This function reads and writes some register values of a thread in the current task.

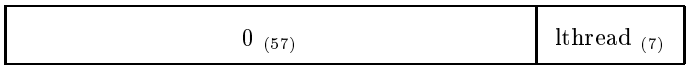
It also creates threads. Conceptually, creating a task includes creating all of its threads. Except lthread 0, all these threads run an idle loop. Of course, the kernel does neither allocate control blocks nor time slices etc. to them. Setting stack and instruction pointer of such a thread to valid values then really generates the thread.

Note that this operation reads and writes the *user-level* registers (SP and IP). Ongoing kernel activities are not affected. However an ipc operation is cancelled or aborted. If the ipc is either waiting to send a message or waiting to receive a message, i.e. a message transfer is not yet running, ipc is cancelled (completion code 0x40 or 0x50). If a message transfer is currently running, ipc is aborted (completion code 0xC0 or 0xD0).

MIPS Implementation Note: The L4/x86 *int preempter* is currently not supported in L4/MIPS and has been removed from the arguments. *excpt id* has been added to L4/MIPS to specify the exception handling thread for the thread, see section 2.5 for details.

Parameters

lthread no



Number of addressed lthread (0...127) inside the current task.

SP	valid	New stack pointer (SP) for the thread. It must point into the user-accessible part of the address space.
	"inv"	(0xFFFFFFFFFFFFFFFF) The existing stack pointer is not modified.
IP	valid	New instruction pointer (IP) for the thread. It must point into the user-accessible part of the address space.
	"inv"	(0xFFFFFFFFFFFFFFFF) The existing instruction pointer is not modified.
excpt id	valid	Defines the exception handling thread used by the thread.
	"inv"	(0xFFFFFFFFFFFFFFFF) The existing excpt id is not modified.
pager	valid	Defines the pager used by the thread.
	"inv"	(0xFFFFFFFFFFFFFFFF) The existing pager id is not modified.
old SP		Old stack pointer (SP) of the thread.
old SP		Old instruction pointer (IP) of the thread.
old excpt id		Id of the thread's old exception handler.

old pager	Id of the thread's old pager.

Example

Signalling can be implemented as follows:

```
signal (lthread) :
    sp := receive signal stack ;
    ip := receive signal ;
    mem [sp – –] := 0 ;
    lthread ex regs (lthread, sp, ip, –, –) ;
    mem [sp – –] := ip ;
    mem [idle stack – wordlength] := sp .

receive signal :
    push all regs ;
    while mem [sp + 8 × wordlength] = 0 do
        thread switch (nilthread)
    od ;
    pop all regs ;
    pop (sp) ;
    jmp (signal ip) .
```

task_new

IP	**a0**			**a0**	∼
pager	**a1**			**a1**	∼
SP	**a2**			**a2**	∼
dest task	**a3**	— **AT** 0x7→		**a3**	∼
mcp / new chief	**a4**			**a4**	∼
excpt id	**a5**			**a5**	∼
∼	**v0**			**v0**	*new task id*

This function deletes and/or creates a task. Deletion of a task means that the address space of the task and all threads of the task disappear. The cputime of all deleted threads is added to the cputime of the deleting thread. If the deleted task was chief of a clan, all tasks of the clan are deleted as well.

Tasks may be created as *active* or *inactive*, as defined by the *pager* attribute. For an active task, a new address space is created together with 128 threads. Lthread 0 is started, the other ones wait for a "real" creation by lthread_ex_regs. An inactive task is empty. It occupies no resources, has no address space and no threads. Communication with inactive tasks is not possible. Loosely speaking, inactive tasks are not really existing but represent only the right to create an active task.

A newly created task gets the creator as its chief, i.e. it is created inside the creator's clan. Symmetrically, a task can only be deleted either directly by its chief (its creator) or indirectly by a higher-level chief.

Parameters

dest task
: Task id of an *existing* task (active or inactive) whose chief is the current task. If one of these preconditions is not fulfilled, the system call has no effect. Simultaneously, a new task *with the same task number* is created. It may be active or inactive (see next parameter).

pager

≠ *nil*
: The new task is created as *active*. The specified pager is associated to lthread 0.

= *nil*
: (0) The new task is created as *inactive*. Lthread 0 is not created.

SP
: Initial stack pointer for lthread 0 if the new task is created as an active one. Ignored otherwise.

IP
: Initial instruction pointer for lthread 0 if the new task is created as an active one. Ignored otherwise.

mcp
: **MIPS Implementation Note:** *mcp* is currently ignored on L4/MIPS.
Maximum controlled priority (mcp) defines the highest priority which can be ruled by the new task acting as a scheduler. The new task's effective mcp is the minimum of the creator's mcp and the specified mcp.
a4 contains this parameter, if the newly generated task is an *active* task, i.e. has a pager and at least lthread 0.

new chief		Specifies the chief of the new inactive task. This mechanism permits to transfer inactive ("empty") tasks to other tasks. Transferring an inactive task to the specified chief means to transfer the related right to create a task. Note that the task number remains unchanged. **a4** contains this parameter, if the newly generated task is an *inactive* task, i.e. has no pager and no threads. The lthread no of the chief id is ignored, the effective chief (as far as ipc delivery is concerned) is the chief tasks' lthread 0.
new task id	$\neq nil$	Task creation succeeded. If the new task is active, the new task id will have a new version number so that it differs from all task ids used earlier. Chief and task number are the same as in *dest task*. If the new task is created inactive, the chief is taken from the *chief* parameter; the task number remains unchanged. The version is undefined so that the new task id might be identical with a formerly (but not currently and not in future) valid task id. This is safe since communication with inactive tasks is impossible.
	$=nil$	(0) The task creation failed.
excpt id		Specifies the default exception handler thread id for new task.

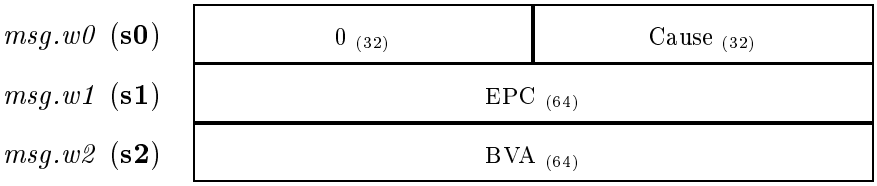
2.5 Exception Handling

Exceptions in L4/MIPS are handled with IPC, unlike L4/x86 which handles exceptions via x86 mirrored exception handling using a per thread IDT.

In L4/MIPS a thread which raises an exception which is caught by L4, has a RPC done on it's behalf to an exception handling thread, which can be specified per thread.

The thread that took the exception is left waiting, the exception handling thread can either shut down the offending thread, or generate a signal, or implement any other model the OS designer chooses.

When a thread takes an exception, the kernel sends the following to the thread's exception handler.



Cause Contents of the R4000 Cause register. It describes what type of exception was taken.

EPC Contents of the R4000 EPC register. It contains the address of the instruction that caused the exception, except when the instruction is in the branch delay slot, in which case it contains the address of the preceding branch instruction.

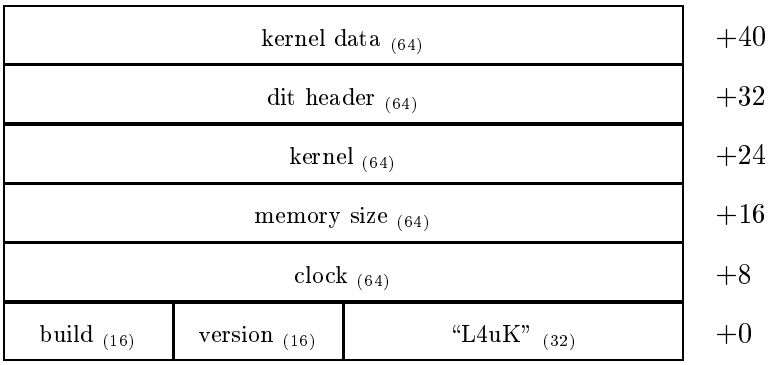
BVA Contents of the R4000 BVA register. It contains the virtual address that caused the exception.

See the R4600 processor manual[Int95] for more details of these registers including how and when they are set.

2.6 The Kernel-Info Page

The kernel-info page contains kernel-version data, memory descriptors *and the clock*. The remainder of the page is undefined. The kernel-info page is mapped *read-only* in the σ_0-address space. σ_0 can use the memory descriptors for its memory management. σ_0 can map the page read-only to other address spaces.

The kernel information page contains information useful for the initial servers to find out about the environment they were started in. Its layout is as follows.



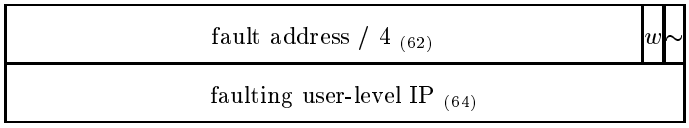
version	L4/R4600 version number.
build	L4/R4600 build number of above version
clock	Number of 1 milliseconds ticks since L4 booted.
memory size	The amount of RAM installed on machine L4 is running on.
kernel	The address + 1 of last byte reserved by the kernel of low physical memory.
dit header	The address of the DIT header which maps out what was loaded with the kernel image.
kernel data	The address of the start of kernel reserved memory in the upper physical memory region.

The physical memory initially available for applications lies between *kernel* and *kernel data*.

2.7 Page-Fault and Preemption RPC

Page Fault RPC

kernel sends:



$w = 0$ Read page fault.
$w = 1$ Write page fault.

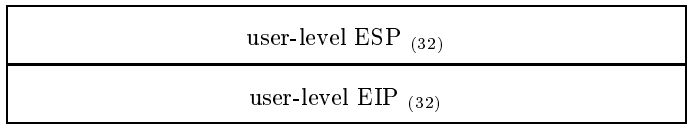
kernel receives: The kernel provides a receive fpage covering the complete user address space. The kernel accepts mappings or grants into this region. Only a short (i.e., register-only) message is accepted, and its contents are ignored.

timeouts used for pagefault RPC	PF at user level	PF at ipc in receiver's space	PF at ipc in sender's space
snd	∞	sender's snd pf	receiver's rcv pf
rcv	∞	sender's snd pf	receiver's rcv pf
snd pf	∞	sender's snd pf	receiver's rcv pf
rcv pf	∞	sender's snd pf	receiver's rcv pf

Preemption RPC

MIPS Implementation Note: Preemption RPC is yet to be implemented in L4/MIPS.

kernel sends:



ESP and EIP are the R4x00's exception stack pointer and exception instruction pointer registers, respectively.

kernel receives: The kernel only accepts a short (in-register) message, whose contents are ignored.

timeouts used for preemption RPC	
snd	∞
rcv	∞
snd pf	∞
rcv pf	∞

2.8 σ_0 RPC protocol

σ_0 is the initial address space. Although σ_0 may not be part of the kernel its basic protocol is defined by the μ-kernel. Special σ_0 implementations may extend this protocol.

The address space σ_0 is idempotent, i.e. all virtual addresses in this address space are identical to the corresponding physical address. Note that pages requested from σ_0 continue to be mapped idempotently if the receiver specifies its complete address space as receive fpage.

σ_0 gives pages to the kernel and to arbitrary tasks, but only once. The idea is that all pagers request the memory they need in the startup phase of the system so that afterwards σ_0 has spent all its memory. Further requests will then automatically denied (by sending a null reply).

MIPS Implementation Note: L4/MIPS σ_0 behaves similar to L4/x86 σ_0, however the actual RPC protocol is slightly modified and defined below. σ_0 handles device mappings via a special case in the page fault protocol only recognised in general by σ_0. This special case when used to map a page, will map it with "uncacheable" attributes suitable for doing device I/O.

A page mapped as above, when passed on to another task via fpage ipc, will continue to retain its uncacheable attributes.

General Memory Mapping

Physical memory

msg.w0 (**s0**)	address $_{(64)}$
msg.w1 (**s1**)	$\sim$ $_{(64)}$

If *address* is in the available memory range and not previously mapped, σ_0 sends a writable mapping to the requester.

Unlike L4/x86, multiple mappings of the same physical frame is not supported, any frame is only mapped once.

Kernel information page

msg.w0 (**s0**)	0xFFFFFFFFFFFFFFFD $_{(64)}$
msg.w1 (**s1**)	$\sim$ $_{(64)}$

Maps the kernel info page to the requester read-only. The requester receives the address of the info page in **s0** assuming a one to one mapping. Multiple mappings to multiple requesters are supported. Note that the address of the dit header can be found in the kernel information page (Sect. 2.6).

Dit header page

msg.w0 (**s0**)	dit header address $_{(64)}$
msg.w1 (**s1**)	$\sim$ $_{(64)}$

Maps the dit header page read-only. Multiple mappings to multiple requesters are supported.

Devices

msg.w0 (**s0**)	0xFFFFFFFFFFFFFFFE $_{(64)}$
msg.w1 (**s1**)	address $_{(64)}$

If address is not in the normal memory range, σ_0 maps *address* writable and uncacheable so as to enable access to device registers etc. The current implementation supports multiple mappings to any of the initial servers started (that is anyone in σ_0's clan).

It is expected that initial servers protect devices from untrusted access via the clans and chiefs mechanism. Device mappings retain their cacheability attributes if passed on via mapping IPC.

2.9 DIT header

DIT is the tool used to build kernel images for download. Like the L4 kernel itself, it has an information page describing the layout of various programs and data that were part of the downloaded kernel image. It consists of two parts, the initial header followed by zero of more file headers as specified by the initial header. The initial headers format follows below.

vaddr end $_{(32)}$	+20
file end $_{(32)}$	+16
phdr num $_{(32)}$	+12
phdr size $_{(32)}$	+8
phdr off $_{(32)}$	+4
"dhdr" $_{(32)}$	+0

phdr off — The offset from the beginning of this header to where the file headers start.

phdr size — The size of each of the file headers.

phdr num — The number of file headers that follow.

file end — The offset to the end of the kernel image file. For DIT internal use only.

vaddr end — The end of currently used physical memory space. This includes the L4 kernel and all other programs and data in the downloaded kernel image.

Each of the file headers is laid out as follows.

flags $_{(32)}$	+28
entry $_{(32)}$	+24
size $_{(32)}$	+20
base $_{(32)}$	+16
name string $_{(32)}$	+12
name string $_{(32)}$	+8
name string $_{(32)}$	+4
name string $_{(32)}$	+0

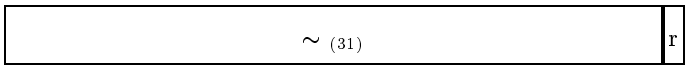
name string — Null terminated string containing name of program or data file (truncated to 16 characters).

base The base address of the program or data file .

size The size of the program or data.

entry The start address of the program if it is executable, zero otherwise.

flags Miscellaneous flags defined below.



r If set the kernel runs this program as part of the initial servers upon startup. If not set, the program or data has simply been loaded into memory and has not been invoked.

Appendix A

DIT

Downloadable Image Tool (DIT) is used to construct downloadable images that contain several parts. It exists as the boot monitor on several systems we use only supports the download of a single 32-bit ELF file. The L4-kernel is 64-bit ELF and we also needed to append various initial servers, so dit was created to achieve this.

DIT does this by "massaging" the L4 64-bit elf header into something that fools the boot monitor into thinking it is 32-bit elf. It also appends an ELF program segment containing the dit header.

Appending arbitrary files to the image is achieved by copying the file into the newly added program segment and noting it in the dit header.

DIT has the following arguments.

`-i l4_kernel kernel_image` Transforms the initial 64-bit kernel `l4_kernel` into a 32-bit downloable image `kernel_image`. Also adds the dit header.

`-l kernel_image` Prints out the next available address in the physical memory space that a program to be appended should be linked at. This is also the default address that unstructured data is appended at.

`-m kernel_image` Prints out a map of what is in the current kernel image.

`-a file kernel_image` Appends `file` to `kernel_image`. If the `file` is 64-bit elf, 32-bit elf, or 32-bit ecoff, then dit acts as a program loader laying out the contents of the executable (.text, .data, .bss etc.) inside the kernel image such that it forms a runnable image once downloaded.

If the file is not of the above format, it is simply appended as is. Note that dit enforces page alignment of 4 kbytes and rounds file size up accordingly.

`-h addr` Used in conjunction with `-a`. Instead of the image being appended at the default address, it is appended at address `addr` in the physical address space.

`-n` Used in conjunction with `-a`, it unsets the run flag for the image being appended so it is not started by L4 as one of the initial servers. This is default for unstructured data.

`-f` Used in conjunction with `-a`. Forces the the file to be appended as unstructured data even if it is a coff or elf format executable.

`-z` Does not include the bss section in the image. Expects bss to be allocated and zeroed at load time.

Appendix B

Serial Port Server

This is a description of the simple serial port server we are currently using on L4/MIPS. It could be described as "focused on fast implementation", or in other words "a quick hack".

We use a serial port server, rather than access the hardware directly as it provides reliable output unaffected by unreliable servers that are under development. Any "real" system would use a more efficient method than the simple method described below.

B.1 Output

The server waits for IPC (short, 64-byte message only) and upon receiving it, converts the received message into a string buffer and sends the null terminated string out the serial port. A simple code fragment to print "hello world" follows.

```
const l4_threadid_t SERIAL_TID = {0x10020000000060001LL};
l4_ipc_reg_msg_t msg;
l4_msgdope_t result;
char *c;

c = (char *) &msg.reg[0];
sprintf(c,"Hello World\n");
r = l4_mips_ipc_send(SERIAL_TID,
                     L4_IPC_SHORT_MSG,&msg,
                     L4_IPC_NEVER, &result);
```

The maximum string length is 64 bytes, which is the maximum amount of data that can be transfered in registers. Strings of less than 64 bytes are null-terminated. Also note the threadid assumes it is the first task loaded after σ_0.

B.2 Input

The simple server also supports receiving input from the serial port. Upon receiving a character from the serial port, the server will IPC the single character to whoever is registered as the receiver.

To register as the receiver, one should send a message to the server with the first 64-bit word being zero, and the second word being the thread id of the thread that is to receive the input characters.

A sample code fragment follows.

```
/* register to receive serial input */
id = l4_myself();
msg.reg[0] = 0;
msg.reg[1] = id.ID;
r = l4_mips_ipc_send(SERIAL_TID, L4_IPC_SHORT_MSG, &msg,
                     L4_IPC_NEVER, &result);

/* loop receiving input */
  while (1)
  {
    r = l4_mips_ipc_wait(&id, L4_IPC_SHORT_MSG, &msg,
                         L4_IPC_NEVER, &result);
    rcv_buffer[i++] = (char) msg.reg[0];
  }
```

Note that thread 0 is used to receive IPC for output and registration, and thread 1 is used to send input characters to the registered receiver.

Appendix C

Kernel Debugger

The L4 kernel debugger, as it's name suggests, is used for debugging the L4 kernel itself. It is not intended to be used for debugging applications, though it can be used to if one understands enough of the internals of L4.

The debugger is very primitive in functionality. It allows basic exploration of kernel data (both global and task specific data), memory, and R4600 registers including co-processor registers.

The kernel debugger is contantly evolving with extra features added when required to assist in debugging new problems. A description of the current list of commands follows.

`?` Print out a short help message.

`rbt` Reboot the system.

`bl` Print out the current busy list in the scheduler.

`ct` Change the debuggers "current task control block" to the one specified by the given `address`.

`pm` Print out `number` of 64-bit memory locations starting at `address`. The `number` argument is optional.

`pt` Print out the state of the "current TCB".

`pr` Print out the general register set. If `register` argument is given, then print only the register specified.

`pk` Print out the kernel data.

`pc` Print out co-processor `register` specified. Valid register names are `bva`, `epc`, `ehi`, `prid`, `tlb`, `xc`, `st`, `cs`.

`pgpt` Print out the page table associated with the current TCB.

`bon` Switch on the compiled in kernel break points.

`version` Print out the version and build number of the current kernel.

Note when in the debugger, a `cntrl-D` will exit the debugger and continue L4. This is only sensible to do when the debugger was invoked via a kernel breakpoint, and application assertion, or by pressing the interrupt key.

C.1 assert

The supplied library `libl4.a` and header file `assert.h`, implement the usual `assert()` function, ie if the assertion fails, the application stops and the file and line number of the failed assertion is printed. However, unlike a normal assert, after printing the above message the kernel debugger is called and the whole system is stopped. The system can be continued as mentioned above by entering `cntrl-D`.

Appendix D

L4 C Library Headers

The following are the self documented header files for the C library interface to L4 on the MIPS R4x00 platform. The headers contain useful constants, macros, and function prototypes for programming in the L4 environment.

D.1 types.h

```
#ifndef __L4TYPES_H__
#define __L4TYPES_H__


/*****************************************************************************
 * Define the basic types upon which to build other types
 *****************************************************************************/
#if defined(_LANGUAGE_C)

#if _MIPS_SZPTR == 64    /* SGI 64-bit compiler */
typedef unsigned char       byte_t;       /* 8-bit int */
typedef unsigned short int  hword_t;      /* 16-bit int */
typedef unsigned int        word_t;       /* 32-bit int */
typedef unsigned long       dword_t;      /* 64-bit int */
typedef long                cpu_time_t;

#else                    /* gcc or SGI 32-bit compiler */

typedef unsigned char       byte_t;
typedef unsigned short int  hword_t;
typedef unsigned int        word_t;
typedef unsigned long long  dword_t;
typedef long long           cpu_time_t;

#endif

/* define struct to access upper
   and lower 32-bits of 64-bit int */
typedef struct {
```

```
  word_t high, low;
} l4_low_high_t;


/*****************************************************************************
 * Structures for accessing L4 thread identifiers
 *****************************************************************************/

/* the basic layout of a tid */
typedef struct {
  unsigned nest:4;
  unsigned chief:11;
  unsigned site:17;
  unsigned version_high:4;
  unsigned task:11;
  unsigned lthread:7;
  unsigned version_low:10;
} l4_threadid_struct_t;

/* the general purpose thread id type giving access as */
typedef union {
  dword_t ID;                       /* 64-bit int */
  l4_low_high_t lh;                 /* two 32-bit ints */
  l4_threadid_struct_t id;          /* individual fields in struct */
} l4_threadid_t;

typedef l4_threadid_t l4_taskid_t;   /* task id is same as thread id */

/* the layout of an interrupt id */
typedef struct {
  unsigned  _pad:32;
  unsigned  _pad2:29;
  unsigned intr:3;
} l4_intrid_struct_t;

/* the general purpose interrupt id */
typedef union {
  dword_t ID;
  l4_low_high_t lh;
  l4_intrid_struct_t id;
} l4_intrid_t;


/*****************************************************************************
 * useful thread id constants, macros and functions
 *****************************************************************************/

#ifdef __GNUC__ /* for gcc */
```

```
#define L4_NIL_ID               ((l4_threadid_t)0ULL)
#define L4_INVALID_ID           ((l4_threadid_t)0xffffffffffffffffULL)
#define l4_is_nil_id(id) ((id).ID == 0uLL)
#define l4_is_invalid_id(id) ((id).ID == 0xffffffffffffffffuLL)

#else  /* for SGI 64-bit compiler */

extern const l4_threadid_t _l4_nil_tid;  /* constants defined in libl4.a */
extern const l4_threadid_t _l4_invalid_tid;

#define L4_NIL_ID               _l4_nil_tid
#define L4_INVALID_ID           _l4_invalid_tid
#define l4_is_nil_id(id) ((id).ID == 0ul)
#define l4_is_invalid_id(id) ((id).ID == 0xfffffffffffffffful)

#endif

#define thread_equal(t1, t2) ((t1).ID == (t2).ID)

/* test if two threads are in same task */
extern int task_equal(l4_threadid_t t1, l4_threadid_t t2);

/* get the task id of given thread, ie thread id of lthread 0 */
extern l4_threadid_t get_taskid(l4_threadid_t t);

#else /* for SGI 64-bit assembly */

#define L4_NIL_ID               0
#define L4_INVALID_ID           0xffffffffffffffff

#endif



/*****************************************************************************
 * L4 flex pages
 *****************************************************************************/

#if defined(_LANGUAGE_C)

/* layout of an fpage */
typedef struct {
  unsigned pageh:32; /* upper 32-bits */
  unsigned page:20;  /* lower 32-bits */
  unsigned zero:3;
  unsigned size:7;
  unsigned write:1;
  unsigned grant:1;
} l4_fpage_struct_t;
```

```
/* general purpose fpage type allowing access as */
typedef union {
  dword_t fpage;         /* a 64-bit int */
  l4_fpage_struct_t fp;  /* fields in struct */
} l4_fpage_t;

/* a send page */
typedef struct {
  dword_t snd_base;
  l4_fpage_t fpage;
} l4_snd_fpage_t;

#ifdef __GNUC__
#define L4_PAGESIZE     (0x1000uLL)   /* L4/MIPS page size */
#else
#define L4_PAGESIZE     (0x1000ul)
#endif
#else /* assembler */
#define L4_PAGESIZE     0x1000
#endif

/*****************************************************************************
 * useful constants, macros and functions to manipulate fpages
 *****************************************************************************/

#define L4_PAGEMASK     (~(L4_PAGESIZE - 1))
#define L4_LOG2_PAGESIZE (12)
#define L4_WHOLE_ADDRESS_SPACE (64)
#define L4_FPAGE_RO     0           /* read-only fpage */
#define L4_FPAGE_RW     1           /* read-write fpage */
#define L4_FPAGE_MAP    0           /* map fpage */
#define L4_FPAGE_GRANT  1           /* grant fpage */

#define L4_FPAGE_GRANT_MASK 1 /* masks for manipulations as integer */
#define L4_FPAGE_RW_MASK    2

#if defined(_LANGUAGE_C)

/* function to build fpage descriptors */
extern  l4_fpage_t l4_fpage(dword_t address, /* address of fpage */
                            unsigned int size, /* size of fpage in 'bits' */
                            unsigned char write, /* read-only / read-write */
                            unsigned char grant); /* map or grant */

/*****************************************************************************
 * L4 message dopes
 *****************************************************************************/
```

```
/* layout of a message dope */
typedef struct {
  unsigned pad:32;    /* upper 32-bits zero */
  unsigned dwords:19;
  unsigned strings:5;
  unsigned error_code:3;
  unsigned snd_error:1;
  unsigned src_inside:1;
  unsigned msg_redirected:1;
  unsigned fpage_received:1;
  unsigned msg_deceited:1;
} l4_msgdope_struct_t;

/* general purpose msgdope type allowing access as */
typedef union {
  dword_t msgdope;   /* 64-bit int */
  l4_msgdope_struct_t md; /* fields in struct */
} l4_msgdope_t;

/*****************************************************************************
 * L4 string dopes
 *****************************************************************************/

typedef struct {
  dword_t snd_size;    /* size of string to send */
  dword_t snd_str;     /* pointer to string to send */
  dword_t rcv_size;    /* size of receive buffer */
  dword_t rcv_str;     /* pointer to receive buffer */
} l4_strdope_t;


/*****************************************************************************
 * L4 message header
 *****************************************************************************/

typedef struct {
  l4_fpage_t rcv_fpage;     /* rcv fpage option */
  l4_msgdope_t size_dope;   /* size dope of message */
  l4_msgdope_t snd_dope;    /* send dope of message */
} l4_msghdr_t;


/*****************************************************************************
 * L4 timeouts
 *****************************************************************************/

/* layout of a timeout */
typedef struct {
  unsigned pad:32;
```

```
  unsigned rcv_man:8;     /* receive mantissa */
  unsigned snd_man:8;     /* send mantissa */
  unsigned rcv_pfault:4;  /* receive pagefault timeout */
  unsigned snd_pfault:4;  /* send pagefault timeout */
  unsigned snd_exp:4;     /* send exponent */
  unsigned rcv_exp:4;     /* receive exponent */
} l4_timeout_struct_t;

/* general purpose timeout type that allows access as */
typedef union {
  dword_t timeout;            /* timeout as 64-bit int */
  l4_timeout_struct_t to;     /* timeout as fields in struct */
} l4_timeout_t;

#endif


/*****************************************************************************
 * useful constants, macros and functions for manipulating timeouts
 *****************************************************************************/

/* masks for manipulating timeouts as integers */
#define L4_RCV_EXP_MASK        0x0000000f
#define L4_SND_EXP_MASK        0x000000f0
#define L4_SND_PFLT_MASK       0x00000f00
#define L4_RCV_PFLT_MASK       0x0000f000
#define L4_SND_MAN_MASK        0x00ff0000
#define L4_RCV_MAN_MASK        0xff000000

#if defined(_LANGUAGE_C)

/* function to build timeout descriptor */
extern  l4_timeout_t L4_IPC_TIMEOUT(byte_t snd_man, /* send mantissa */
                                    byte_t snd_exp, /* send exponent */
                                    byte_t rcv_man, /* receive mantissa */
                                    byte_t rcv_exp, /* receive exponent */
                                    byte_t snd_pflt, /* send pageflt timeout */
                                    byte_t rcv_pflt); /* rcv pageflt timeout */

/* constant to specify to never timeout during ipc */
#ifdef __GNUC__
#define  L4_IPC_NEVER ((l4_timeout_t) {timeout: 0})
#else
#define  L4_IPC_NEVER _l4_ipc_never
extern const l4_timeout_t _l4_ipc_never;
#endif
#else /* assembler */
#define L4_IPC_NEVER      0
#endif
```

```
/*****************************************************************************
 * l4_schedule param word: NOT USED in current version
 *****************************************************************************/

#if defined(_LANGUAGE_C)
typedef struct {
  unsigned time_man:8;
  unsigned time_exp:4;
  unsigned zero:12;
  unsigned prio:8;
} l4_sched_param_struct_t;

typedef union {
  dword_t sched_param;
  l4_sched_param_struct_t sp;
} l4_sched_param_t;

#endif

#endif /* __L4TYPES_H__ */
```

D.2 syscalls.h

```
#ifndef __L4_SYSCALLS_H__
#define __L4_SYSCALLS_H__

#include <l4/types.h>

/*****************************************************************************
 * system call numbers for assembly hackers
 *****************************************************************************/

#if defined(_LANGUAGE_ASSEMBLY)
#define SYSCALL_IPC              0
#define SYSCALL_FPAGE_UNMAP      1
#define SYSCALL_ID_NEAREST       2
#define SYSCALL_ID_NCHIEF        3
#define SYSCALL_THREAD_SWITCH    4
#define SYSCALL_THREAD_SCHEDULE  5
#define SYSCALL_LTHREAD_EX_REG   6
#define SYSCALL_TASK_CREATE      7
#define MAX_SYSCALL_NUMBER       7
#endif

/*****************************************************************************
 * prototypes and constants for system calls other than ipc
 *****************************************************************************/

#ifdef _LANGUAGE_C
extern void l4_fpage_unmap(l4_fpage_t fpage, dword_t map_mask);

/* valid values for mask */
#define L4_FP_REMAP_PAGE        0x00    /* Page is set to read only */
#define L4_FP_FLUSH_PAGE        0x02    /* Page is flushed completely */
#define L4_FP_OTHER_SPACES      0x00    /* Page is flushed in all other */
                                        /* address spaces */
#ifdef __GNUC__
#define L4_FP_ALL_SPACES        0x8000000000000000LL
                                        /* Page is flushed in own address */
                                        /* space too */
#else
#define L4_FP_ALL_SPACES        0x8000000000000000ul
#endif

extern l4_threadid_t l4_myself(void);


extern int l4_id_nearest(l4_threadid_t destination,
                         l4_threadid_t *next_chief);
```

```
#endif

/* return values of l4_id_nearest */
#define L4_NC_SAME_CLAN         0x00    /* destination resides within the */
                                        /* same clan */
#define L4_NC_INNER_CLAN        0x0C    /* destination is in an inner clan */
#define L4_NC_OUTER_CLAN        0x04    /* destination is outside the */
                                        /* invoker's clan */

#if defined(_LANGUAGE_C)
extern void l4_thread_ex_regs(l4_threadid_t destination,
                              dword_t eip, dword_t esp,
                              l4_threadid_t *excpt, l4_threadid_t *pager,
                              dword_t *old_eip, dword_t *old_esp);

extern void l4_thread_switch(l4_threadid_t destination);

/* l4_thread_schedule not implemented */
extern cpu_time_t
l4_thread_schedule(l4_threadid_t dest, l4_sched_param_t param,
                   l4_threadid_t *ext_preempter, l4_threadid_t *partner,
                   l4_sched_param_t *old_param);

extern l4_taskid_t
l4_task_new(l4_taskid_t destination, dword_t mcp_or_new_chief,
            dword_t esp, dword_t eip, l4_threadid_t pager,
            l4_threadid_t excpt);
#endif
#endif
```

D.3 ipc.h

```
#ifndef __L4_IPC_H__
#define __L4_IPC_H__

#include <l4/types.h>

/*****************************************************************************
 * L4 IPC
 *****************************************************************************/

#ifdef _LANGUAGE_C

/*****************************************************************************
 * For backward compatibility only - do not use!
 *****************************************************************************/
#define L4_IPC_MSG_DECEITED L4_IPC_MSG_DECEIVED
#define l4_mips_ipc_reply_deceiting_and_wait l4_mips_ipc_reply_deceiving_and_wait
#define l4_mips_ipc_send_deceiting l4_mips_ipc_send_deceiving
#define l4_ipc_sleep l4_mips_ipc_sleep


/*****************************************************************************
 * L4 registered message. Structure used to pass registered message to the
 * libl4 C library which loads/stores messages to be sent/received into/from
 * registers from/into this structure. If you get my gist :-)
 *****************************************************************************/

#define L4_IPC_MAX_REG_MSG 8

typedef struct {
  dword_t reg[L4_IPC_MAX_REG_MSG];
} l4_ipc_reg_msg_t;

#endif

/*****************************************************************************
 * Defines used for constructing send and receive descriptors
 *****************************************************************************/

#define L4_IPC_SHORT_MSG        0    /* register only ipc */

#ifdef _LANGUAGE_C
#define L4_IPC_STRING_SHIFT 8 /* shift amount to get strings
                                 from message dope */
#define L4_IPC_DWORD_SHIFT 13 /* shift ammount to get dwords
                                 from message dope */

#define L4_IPC_SHORT_FPAGE ((void *)2ul) /* register only ipc including
```

```
                                        sending fpages in registers */

/* macro for creating receive descriptor that receives register
   only ipc that includes fpages */
#define L4_IPC_MAPMSG(address, size)  \
     ((void *)(dword_t)( ((address) & L4_PAGEMASK) | ((size) << 2) \
                        | (unsigned long)L4_IPC_SHORT_FPAGE))
#else /* assembly */

#define L4_IPC_SHORT_FPAGE          2
#define L4_IPC_NIL_DESCRIPTOR   (-1)
#define L4_IPC_DECEIT           1
#define L4_IPC_OPEN_IPC         1

#endif

/*****************************************************************************
 * Some macros to make result checking easier
 *****************************************************************************/

#define L4_IPC_ERROR_MASK       0xF0
#define L4_IPC_DECEIT_MASK      0x01
#define L4_IPC_FPAGE_MASK       0x02
#define L4_IPC_REDIRECT_MASK    0x04
#define L4_IPC_SRC_MASK         0x08
#define L4_IPC_SND_ERR_MASK     0x10

#ifdef _LANGUAGE_C
#define L4_IPC_IS_ERROR(x)              (((x).msgdope) & L4_IPC_ERROR_MASK)
#define L4_IPC_MSG_DECEIVED(x)          (((x).msgdope) & L4_IPC_DECEIT_MASK)
#define L4_IPC_MSG_REDIRECTED(x)        (((x).msgdope) & L4_IPC_REDIRECT_MASK)
#define L4_IPC_SRC_INSIDE(x)            (((x).msgdope) & L4_IPC_SRC_MASK)
#define L4_IPC_SND_ERROR(x)             (((x).msgdope) & L4_IPC_SND_ERR_MASK)
#define L4_IPC_MSG_TRANSFER_STARTED \
     ((((x).msgdope) & L4_IPC_ERROR_MASK) < 5)


/*****************************************************************************
 * Prototypes for IPC calls implemented in libl4.a
 *****************************************************************************/

extern int
l4_mips_ipc_call(l4_threadid_t dest,
                 const void *snd_msg,
                 l4_ipc_reg_msg_t *snd_reg,
                 void *rcv_msg,
                 l4_ipc_reg_msg_t *rcv_reg,
                 l4_timeout_t timeout,
                 l4_msgdope_t *result);
```

```
extern int
l4_mips_ipc_reply_and_wait(l4_threadid_t dest,
                           const void *snd_msg,
                           l4_ipc_reg_msg_t *snd_reg,
                           l4_threadid_t *src,
                           void *rcv_msg,
                           l4_ipc_reg_msg_t *rcv_reg,
                           l4_timeout_t timeout,
                           l4_msgdope_t *result);

extern int
l4_mips_ipc_reply_deceiving_and_wait(l4_threadid_t dest,
                                     l4_threadid_t vsend,
                                     const void *snd_msg,
                                     l4_ipc_reg_msg_t *snd_reg,
                                     l4_threadid_t *src,
                                     void *rcv_msg,
                                     l4_ipc_reg_msg_t *rcv_reg,
                                     l4_timeout_t timeout,
                                     l4_msgdope_t *result);

extern 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);

extern int
l4_mips_ipc_send_deceiving(l4_threadid_t dest,
                           l4_threadid_t vsend,
                           const void *snd_msg,
                           l4_ipc_reg_msg_t *snd_reg,
                           l4_timeout_t timeout,
                           l4_msgdope_t *result);

extern int
l4_mips_ipc_wait(l4_threadid_t *src,
                 void *rcv_msg,
                 l4_ipc_reg_msg_t *rcv_reg,
                 l4_timeout_t timeout,
                 l4_msgdope_t *result);

extern int
l4_mips_ipc_receive(l4_threadid_t src,
                    void *rcv_msg,
                    l4_ipc_reg_msg_t *rcv_reg,
                    l4_timeout_t timeout,
```

```
                    l4_msgdope_t *result);

extern int l4_mips_ipc_sleep(l4_timeout_t t,
     l4_msgdope_t *result);

/*****************************************************************************
 * some functions to examine fpages
 *****************************************************************************/
extern int l4_ipc_fpage_received(l4_msgdope_t msgdope); /* test if fpages
                                                           received */
extern int l4_ipc_is_fpage_granted(l4_fpage_t fp);
extern int l4_ipc_is_fpage_writable(l4_fpage_t fp);


/*****************************************************************************
 * Symbolic constants for error codes, see reference manual for details
 *****************************************************************************/

#define L4_IPC_ERROR(x)                 (((x).msgdope) & L4_IPC_ERROR_MASK)
#endif
#define L4_IPC_ENOT_EXISTENT            0x10
#define L4_IPC_RETIMEOUT                0x20
#define L4_IPC_SETIMEOUT                0x30
#define L4_IPC_RECANCELED               0x40
#define L4_IPC_SECANCELED               0x50
#define L4_IPC_REMAPFAILED              0x60
#define L4_IPC_SEMAPFAILED              0x70
#define L4_IPC_RESNDPFTO                0x80
#define L4_IPC_SESNDPFTO                0x90
#define L4_IPC_RERCVPFTO                0xA0
#define L4_IPC_SERCVPFTO                0xB0
#define L4_IPC_REABORTED                0xC0
#define L4_IPC_SEABORTED                0xD0
#define L4_IPC_REMSGCUT                 0xE0
#define L4_IPC_SEMSGCUT                 0xF0


/*****************************************************************************
 * Size limitations on memory based IPC
 *****************************************************************************/

#define L4_MAX_DMSG_SIZE       (4*1024*1024) /* max direct message size */
#define L4_MAX_STRING_SIZE     (4*1024*1024) /* max indirect message size */

#endif /* __L4_IPC__ */
```

D.4 sigma0.h

```
#ifndef __L4_SIGMA0_H__
#define __L4_SIGMA0_H__

#include <l4/types.h>

/*****************************************************************************
 * define some constants relevent to sigma0
 *****************************************************************************/

#ifdef _LANGUAGE_C
#ifdef __GNUC__
#define SIGMA0_DEV_MAP           (0xfffffffffffffffeULL)
#define SIGMA0_KERNEL_INFO_MAP   (0xfffffffffffffffdULL)
#define SIGMA0_TID               ((l4_threadid_t) {ID: (1 << 17)})
#else
#define SIGMA0_DEV_MAP           (0xfffffffffffffffeul)
#define SIGMA0_KERNEL_INFO_MAP   (0xfffffffffffffffdul)
#define SIGMA0_TID           _l4_sigma0_tid
extern const l4_threadid_t _l4_sigma0_tid;
#endif
#else
#define SIGMA0_DEV_MAP           0xfffffffffffffffe
#define SIGMA0_KERNEL_INFO_MAP   0xfffffffffffffffd
#define SIGMA0_TID               (1 << 17)
#endif

/*****************************************************************************
 * define format of kernel info page
 *****************************************************************************/

#ifdef _LANGUAGE_C
typedef struct {
  word_t magic; /* L4uK */
  hword_t version;
  hword_t build;
  dword_t clock;
  dword_t memory_size;
  dword_t kernel;
  dword_t dit_hdr;
  dword_t kernel_data;
} l4_kernel_info;
#else
#define LKI_MAGIC           0
#define LKI_VERSION         4
#define LKI_BUILD           6
#define LKI_CLOCK           8
#define LKI_MEMORY_SIZE    16
```

```
#define LKI_KERNEL          24
#define LKI_DIT_HDR         32
#define LKI_KERNEL_DATA     40

#endif
#endif
```

D.5 dit.h

```
#ifndef DIT_H
#define DIT_H

#define DIT_NIDENT  4
#define DIT_NPNAME 16

#define DHDR_SEG_SIZE 4096
#define DHDR_ALIGN 4096

#define DIT_RUN         1


#if defined(_LANGUAGE_ASSEMBLY)

#define D_D_IDENT       0
#define D_D_PHOFF       4
#define D_D_PHSIZE      8
#define D_D_PHNUM      12
#define D_D_FILEEND    16
#define D_D_VADDREND   20

#define D_P_BASE        0
#define D_P_SIZE        4
#define D_P_ENTRY       8
#define D_P_FLAGS      12
#define D_P_NAME       16

#else /* assume C */

typedef unsigned int Dit_uint;

typedef struct {
  unsigned char d_ident[DIT_NIDENT];
  Dit_uint d_phoff;
  Dit_uint d_phsize;
  Dit_uint d_phnum;
  Dit_uint d_fileend;
  Dit_uint d_vaddrend;
} Dit_Dhdr;

typedef struct {
  Dit_uint p_base;
  Dit_uint p_size;
  Dit_uint p_entry;
  Dit_uint p_flags;
  unsigned char p_name[DIT_NPNAME];
} Dit_Phdr;
```

```
#endif
#endif
```

Bibliography

[CFL94] P. Cao, E. W. Felton, and K. Li. Implementation and performance of application-controlled file caching. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–178, Monterey, CA, 1994.

[GGKL89] M. Gasser, A. Goldstein, C. Kaufmann, and B. Lampson. The Digital distributed system security architecture. In *12th National Computer Security Conference (NIST/NCSC)*, pages 305–319, Baltimore, 1989.

[HKK93] H. Härtig, O. Kowalski, and W. Kühnhauser. The Birlix security architecture. *Journal of Computer Security*, 2(1):5–21, 1993.

[Int95] Integrated Device Technology. *IDT79R4600 and IDT79R4700 RISC Processor Hardware User's Manual*, rev 2.0 edition, April 1995.

[KH92] R. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):11–22, November 1992.

[KN93] Y. A. Khalidi and M. N. Nelson. Extensible file systems in spring. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 1–14, Asheville, NC, 1993.

[LCC94] C. H. Lee, M. C. Chen, and R. C. Chang. HiPEC: high performance external virtual memory caching. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–164, Monterey, CA, 1994.

[Lie92] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, 1992. Springer.

[Lie93a] J. Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, 1993.

[Lie93b] J. Liedtke. A persistent system in real use – experiences of the first 13 years. In *3rd International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 2–11, Asheville, NC, 1993.

[Lie95] J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, 1995.

[RLBC94] T. H. Romer, D. L. Lee, B. N. Bershad, and B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–164, Monterey, CA, 1994.