# — L4Env —
# An Environment for L4 Applications

Operating Systems Research Group
Technische Univerität Dresden
drops@os.inf.tu-dresden.de

June 30, 2003

## Preface

This document gives an introduction into L4Env, a programming environment for application development on top of the L4 microkernel family. L4Env was developed as a part of the Dresden Real-Time Operating System (DROPS) and is still under construction.

The purpose of this document is to give an overview over the concepts of L4Env. More information about building and using L4Env can be found in the *Building DROPS HOWTO* and the *Building Infrastructure for DROPS (BID) Tutorial*. Further on, most components of L4Env provide a reference manual.

We assume that the reader is familiar with the basic L4 concepts (tasks, threads, IPC).

## Contents

## Notations

| | |
|---|---|
| DICE | Development tools which can be found in l4/tool |
| `thread` | L4Env packages which can be found in l4/pkg |

## 1 Motivation

Prior to L4Env, most L4 applications had their very own idea about the environment (libraries, interfaces and so on) in which they were executed. Almost every programmer had his own set of libraries he used to build his applications, which results in huge problems if someone tries to combine components developed by different authors. Frequent problems were conflicting implementations of common functions (like `printf`) or conflicts caused by the lack of a central management of resources like threads or virtual memory.

The intention of L4Env is to define a set of functions which describe a minimal environment. This minimal environment is available for every L4 application. Hence, all applications and especially all libraries can use these functions. Libraries which are intended to be used by many different applications should only use these functions to avoid dependencies to other libraries.

Such an environment also decreases the dependencies to a certain L4 API or hardware architecture, making applications more portable.

## 2 L4Env Concepts

As a microkernel-based system L4Env is built of a set of L4 servers which manage basic system resources like memory, tasks or I/O resources. Various L4Env libraries make use of these servers to give access to the resources and to provide further functionality like thread management or synchronization primitives. Together with the L4Env servers these libraries form the programming environment for application development on L4.

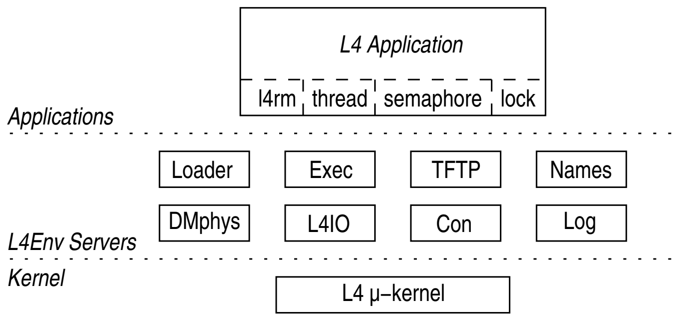

Figure 1: Main L4Env components

Figure 1 shows the main components of L4Env. In addition to these servers and libraries L4Env contains various development tools, most notably the DICE IDL compiler. The following sections will give a short overview of the concepts behind the L4Env components.

### 2.1 Memory Management: DMphys, `l4rm`

L4Env uses the concept of *dataspaces* [1] to provide memory to L4 applications. A dataspace is a container which can contain arbitrary types of memory (e.g. phys. memory, paged memory, files and so on) and is managed by a *dataspace manager*. L4Env contains DMphys (`dm_phys`) as the dataspace manager which manages the available phys. memory of the system. The address space of an application is constructed of several dataspaces, these dataspaces can be managed by different dataspace managers. The *region mapper* (`l4rm`) manages an address space. The region mapper has two major tasks, first to allocate virtual memory regions of the address space and second to invoke the appropriate dataspace manager to resolve a pagefault on a virtual address.

#### Dataspace Manager

The interface to dataspace managers is split into several parts:

`dm_generic` provides the interface common for all dataspace managers of the system. This interface must be implemented by all dataspace managers to ensure a basic functionality regardless of the type of memory the dataspace manager handles. It contains for instance the function invoked by the region mapper to resolve a pagefault or the functions to manage the access rights to dataspaces.

`dm_mem` extends `dm_generic` with functions that are useful with dataspaces specifically containing memory.

`dm_phys` further extents the dataspace manager interface with functions to manage physical memory.

`dm_generic` provides the interface definition and a client library to use the interface, it does not provide a dataspace manager implementation. However, `dm_generic` contains a library (`dsmlib`) which provides various support functions to build a dataspace manager. `dm_mem` just provides the interface definition and a client library. `dm_phys` provides in addition to the interface definition and client library the implementation of the dataspace manager DMphys.

DMphys manages the available phys. memory of the system. It grabs all the available memory from Sigma0 and provides this memory to applications (or other dataspace managers) in terms of dataspaces. DMphys supports the allocation of memory to different memory pools, the usage of superpages and special requirements like contiguous or aligned memory allocation.

Both DMphys and all the dataspace manager interfaces are described in the *L4Env Physical Memory Dataspace Manager Reference Manual*.

### Region Mapper

The region mapper `l4rm` is a library which is linked to all L4Env applications. It provides the pager for all other threads of the task. The pager forwards pagefaults caused by any other thread of the task to the dataspace manager which manages the dataspace which is associated to the address of the pagefault. Figure 2 shows the relationship between the region mapper and the dataspace managers.

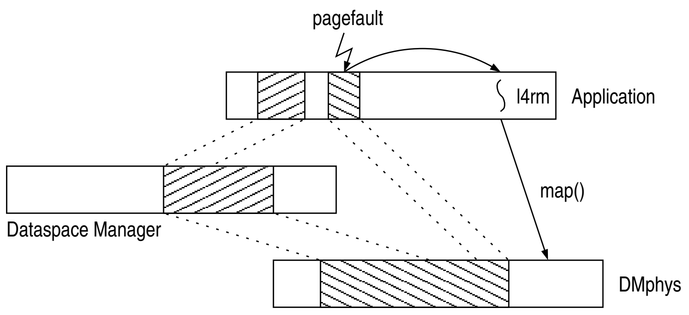

Figure 2: Dataspaces and region mapper

The region mapper provides functions to associate dataspaces with areas of the address space of an application, to map dataspace pages to the address space an to reserve areas of the address space. The usage of the region mapper is described in the *L4 Region Mapper Reference Manual*.

## 2.2 Threads: `thread`

The L4Env thread library provides a basic abstraction to native L4 threads. The thread library provides

- functions to create and destroy threads,
- shutdown callbacks,
- management of thread local data,
- priority handling, and
- stack handling.

The intention of the thread library is not to provide a fully featured thread API like Pthreads, it rather is the basis for native L4 applications and higher level thread abstractions.

The thread library provides a local thread ID which is independent from the underlying L4 implementation. Stacks are allocated using the default dataspace manager provided by L4Env. The stacks are placed in the address space of a task in a way that the ID of a thread can be derived from the stack pointer, thus avoiding expensive system calls (`l4_myself()`). In special cases the stack can be still specified manually. For more information about the usage of the thread library see the *L4 Thread Library Reference Manual*.

## 2.3 IPC: DICE

Since writing IPC code from hand over and over is a tedious task, we adopted a mechanism from distributed computing to hide the communication code: the IDL compiler. It translates an abstract interface description of a service into the communication code to talk to the service. Our IDL compiler is DICE and it generates L4 IPC communication code. It provides the programmer with a C interface to the server and hides all the L4 API specific code inside the communication stubs.

DICE allows you to include C header files into the IDL file to use your existing C types in the interface declaration. It generates code for L4 version 2 and version X.0 (version X.2 support is under way). DICE generates assembler inline code for IPC syscalls at the client side (this will be extended to generate assembler inline code at the server side as well). With former IDL compilers you had to write the server loop by hand. This code is generated by DICE as well. To provide the flexibility of hand-written server loops, you can specify hooks which are called at the critical points in the server code.

For more information about DICE please consult the DICE user's manual.

## 2.4 Synchronisation: `semaphore`, `lock`

L4Env provides basic, task-local synchronization primitives. Synchronization primitives for inter-task synchronization are not supported, such primitives should be part of the application API.

#### Semaphore

The semaphore library provides a simple counting semaphore implementation. The implementation causes nearly no overhead in case of no contention on the semaphore. If a thread must be blocked, a thread provided by the semaphore library is called. This thread manages the wait queues for all semaphores, avoiding the use of `cli/sti` by serializing the accesses to these queues. For details about the usage see the *L4 Semaphore Reference Manual*.

#### Locks

The lock library provides simple locks using the semaphore library. For details about the usage see the *L4 Lock Reference Manual*.

## 2.5 Console Input/Output: `con`, `log`

### 2.5.1 `log`

`log` is package providing a basic textual console *logserver* as well as a set of macros useful for generating logging output during program execution. The logserver and the macros can be used independent of each other.

The *logserver* serializes the output of programs to avoid them to intermix with each other on the output media, i.e. the screen, the serial console or a network connection. The server offers buffering of the data to minimze the impact to timing caused by the output. The logserver comes in two flavours. The simple one uses the L4 kernel debugger for printing text. A network-flavoured version allows data to be transfered over the network using a tcp-based protocol with a telnet at the other end.

The *log macros* allow to easily genearate verbose logging messages which may contain the position (file and line) in the program code. The macros are also available in a conditional flavor, with a condition expression as their first argument. If the condition expression is constant and false at compile-time, the whole logging statement will be ignored by the compiler.

### 2.5.2 `con`

`con` is a graphical console server. It provides virtual consoles which are multiplexed to one visible graphics screen. Only the server can directly access the physical graphics memory. However, the graphics memory can be mapped to a client's address space as long he owns the virtual console which is currently in the foreground. This feature was added due to performance reasons with the XFree86 stub.

Writing to a virtual console can only be performed by its owner. The user can select one virtual console to be actually displayed. There is a small area at the bottom of the screen which cannot be changed by any client. This area is reserved for displaying the ID of the current foreground console. The owner of that virtual console receives input events from mouse and keyboard. A client can open several virtual consoles. Each virtual console acquires one communication thread of the `con` server.

The client uses a simple protocol to communicate with `con`: *pSLIM* ("p" = pseudo) as an extension of the ***S****tateless* ***L****ow level* ***I****nterface* ***M****achine* protocol – proposed by SUN at SOSP 1999 [4]. The original protocol supports the following commands:

**`SET`** Set literal pixels of a rectangular region.

**`BITMAP`** Expand a bitmap to a fill rectangular region with a foreground color where the bitmap contains 1's and background color where the bitmap contains 0's.

**`FILL`** Fill a rectangular region with one pixel value.

**`COPY`** Copy a rectangular region of the framebuffer to another location.

**`CSCS`** Color-space convert rectangular region from YUV to RGB with optional bilinear scaling.

Due to several reasons we have extended the protocol by functions for

- fast font rendering
- mapping a framebuffer from the client to the server
- mapping the physical graphics memory from the server to the client

Input events (mouse, keyboard) are received using an L4 port of the Linux input system.

Furthermore, `con` has limited support for hardware acceleration of some graphics cards (fast copy, fast fill, backend scaler for converting YUV to RGB.

## 2.6 Starting Applications: `loader,exec,tftp`

### 2.6.1 Program Loader

The loader is responsible to load L4 applications at runtime. It is supported by the executable interpreter which interprets the binary and resolves unbound symbols (linking). The current implementation of the interpreter `exec` is able to interpret ELF binaries [2] as produced by the BID system (*Building infrastructure for DROPS*).

The loader is able to load binaries which depend on shared libraries. Those binaries have a special dynamic section listing the shared objects they depend of. Shared libraries should only contain position independent code (*PIC*) which allows to share the `text` section between multiple address spaces and thus allowing to share the same cache and TLB entries on some architectures.

### 2.6.2 File Provider

The file provider interface defines a straight-forward approach to access files. A file can be opened and its image is provided as a dataspace. Currently, we have two implementations:

- `tftp` is an L4 server which requests the file from the network using the `tftp` protocol. The network library was ported from GRUB. Several current network adapters and an interface to an external L4 network server (`oshkosh`, distributed separately) are supported.
- `fprov-l4` (example of the loader package) is an L$^4$Linux program which requests files from the local file system of an L$^4$Linux machine. It is implemented as a Linux task which is aware of the L4 environment interface behind Linux.

Both implementations read the whole image of the file at once and copy it into a new dataspace they allocated at a dataspace manager. Other implementations could read parts of the file on demand.

## 2.7 Tasks: `simple_ts`

The task server `simple_ts` is responsible for creating and destroying L4 tasks. Tasks are created in three steps: Firstly, the client asks the task server to allocate a free task ID. Secondly, the client registers this ID at the pager of the new task. Finally, the client asks the task server to create the task. This protocol ensures that the task doesn't start until its pager knows about it.

To be compatible with $L^4$Linux the task server allows to transmit tasks creation rights to a client.

## 2.8 I/O Resources: `l4io`

`l4io` manages I/O resources and provides an interface to access I/O memory regions, I/O ports, ISA DMA channels and PCI configuration space. All hardware specific PCI code is ported from Linux 2.4. Furthermore `l4io` enables the implementation of user-level interrupt handling based on *Omega0* described in [3] and supports interrupts sharing.

## 2.9 Miscellaneous

### 2.9.1 C-Library

Currently, we use the OSKit C-library for the L4 environment. There are several backend libraries (`oskit*_support*`) to map low-level functionality to L4 primitives.

We plan to move to another or own implementation in the future due to several reasons (see open issues).

### 2.9.2 Name Service: `names`

A simple root name service is part of the L4 environment. `names` provides a mapping of names (strings) to L4 `thread_id`s and vice versa. L4 servers register a unique string plus the thread ID of their worker thread. The belonging thread ID to a name can be requested by other servers to find the services they depend on and to synchronize at startup.

### 2.9.3 Utilities

The `l4util` library is a collection of useful functions not found elsewhere. It includes

- atomic operations (`cmpxchg`, ...),
- bit operations,
- low-level hardware access (port I/O, PIC/APIC programming, performance counter),
- command line parsing,
- base64 encoding, and
- pseudo-random number generator.

# 3 Open Issues

- Most servers are single threaded (vulnerable by *denial-of-service-attacks*).
- Semaphore library and thread priorities.
  - It is not ensured that we wakeup the thread with the highest priority. Using the *not-switch-if-deceit* mechanism of Fiasco does not fully solve that problem.
  - Lack of task-overlapped synchronization
- The loader/linker mechanism is quite complex.
  - We use an executable interpreter to interpret and link the binaries. Another approach would be a linker library which links the binary in its address space. The design would be a little bit more straight forward. It also would be possible to do *lazy linking*, that is resolve symbols at the first time they are accessed. This is probably not important for real-time applications.

  - The loader is still the pager for some program sections (`libloader.s.so` for *Loader-style* applications) or all program sections (for *Sigma0-style* applications). It would be better to use an own server which is only responsible for this job.
  - *L4Env-infopage* mechanism for applications not fully used.

- Still no working solution for task termination:
  - A task can only be killed explicit. A possible solution is to use a notification server (*work-in-progress*). The `exit()` function of the task could send an appropriate event to someone.
  - When a task should be killed, the loader asks all participating L4 servers (`names`, `rmgr`, `con`, `exec`, `simple_ts`) to kill all resources of that task. This could also be solved by an event server.

- No naming hierarchy in the name server.

- No resource reservation. For example, each client can get as much memory from dm_phys as we have physical memory.

- Some open DICE issues (optimization, error handling)
  - more optimization necessary
  - better error handling at client and server side

- We don't have a copy-on-write dataspace manager. A dataspace manager which could hand out cache-coloured pages would also be useful.

- We don't have a file provider for hard disks.

- We still use the heavy-weight OSKit as lib-c.
  - code bloat
  - no support for different output channels (`fprintf(stderr, ...)`)
  - No support for "command line tools". Such a tool would not open an own graphical console but it would use the console of the starter application for input and output.
  - several hacks (e.g. `log` library implements own `sprintf()`)

- We currently support L4 *version 2* and *version X.0*. Support of L4 *version X.2* alias *version 4* may require some internal restructuring and maybe even API changes (e.g. threads can be only created by privileged threads).

- We currently only support one hardware platform (x86) and only single-processor systems.

- We still depend on the `rmgr`.

- No support for resource withdrawing (e.g. IRQs).

- Support for C++ is not fully tested.

- `l4io` doesn't know about other buses than PCI and there is no bus hierarchy.

## References

[1] Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke, Kevin Elphinstone, and Luke Deller. The SawMill Framework for VM Diversity. In *Proc. 6th Australasian Computer Architecture Conference*, January 2001. Available at `ftp://ftp.cse.unsw.edu.au/pub/users/disy/papers/Aron_PJLED_01.ps.gz`.

[2] *Tool Interface Standard (TIS). Portable Formats Specification.* Intel Corporation, Literature Center, P.O. Box 7641, Mt. Prospect, IL 60056-7641, 1993.

[3] J. Löser and M. Hohmuth. Omega0 – a portable interface to interrupt hardware for L4 systems. In *Proceedings of the First Workshop on Common Microkernel System Platforms*, Kiawah Island, SC, USA, December 1999. Available at `http://os.inf.tu-dresden.de/~hohmuth/prj/omega0.ps.gz`.

[4] Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt. The interactive performance of SLIM: a stateless, thin-client architecture. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP), volume 34, pages 32-47*, December 1999.