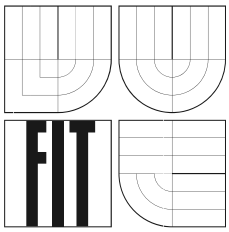
Brno University of Technology

Faculty of Information Technology



System GNU/Hurd on Architecture x86

Bachelor Project

2006

Zbyněk Michl

System GNU/Hurd on Architecture x86

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Information Technology at Brno University of Technology, Faculty of Information Technology,
April 27, 2006

© Zbyněk Michl, 2006

The author thereby grants persmission to reproduce and distribute copies of this document in whole or in part to Brno University of Technology, Faculty of Information Technology.

Declaration

I hereby declare that I have created this project on my own and under the supervision of Ing. Tomáš Kašpárek. I have listed all information sources which I used.

. . . . . .
Zbyněk Michl
April 27, 2006

Abstract

This project focuses on the GNU Hurd operating system. The first part presents concept of the Hurd, Mach and L4 microkernels that create a base of the whole system. The second part describes a few installation procedures of the Hurd on x86 architecture. Current status of the Hurd and presumption of the future development is also mentioned.

Keywords

Coyotos, Debian, GNU, GPL, Grub, Hurd, Hurd-NG, kernel, kernel space, L4, L4.Sec, Mach, microkernel, ngHurd, operating system, Pistachio, POSIX, server, translator, UNIX, user space.

Acknowledgements

First, I would like to thank my supervisor Ing. Tomáš Kašpárek for the opportunity to work on this project and for the time to discuss problems with him. My special thanks go to my family for the possibility to study in Brno and for their constant support all the time as well as to my friends for their encouragement.

Abstrakt

Tato práce se zabývá operačním systémem GNU Hurd. První část práce je zaměřena na koncept Hurdu, dále na mikrojádra Mach a L4, která tvoří základ celého systému. Druhá část popisuje několik instalačních postupů Hurdu na architektuře x86. Práce také sleduje aktuální stav Hurdu a snaží se odhadnout další budoucnost vývoje.

Klíčová slova

Coyotos, Debian, GNU, GPL, Grub, Hurd, Hurd-NG, jádro, L4, L4.Sec, Mach, mikrojádro, ngHurd, operační systém, Pistachio, POSIX, prostor jádra, server, translátor, UNIX, uživatelský prostor.

Contents

Chapter 1

Prologue

The most important program that runs on a computer is an *operating system* (OS). Every general-purpose computer have an operating system to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

Operating systems provide a software platform on the top of which other programs, called application programs, can run. The application programs must be written to run on the top of a particular operating system. Your choice of the operating system, therefore, determines a great extent of applications you can run.

The GNU Hurd is a set of servers (or daemons, in UNIX-speak) that work on the top of a microkernel (μ-kernel for short). Together, they form a base of the GNU operating system. It has been under development since 1991 by the GNU Project and is distributed as free software under the GPL. The Hurd aims to surpass UNIX kernels in functionality, security and stability, while remaining largely compatible with them. This is done by having the Hurd track the POSIX specification, while avoiding arbitrary restrictions on the user.

Chapter 2 — describes monolithic kernel based systems and μ-kernel based systems. Especially differences between mono-server and mutli-server systems are explained.

Chapter 3 — presents the GNU Hurd system. It covers its history, Hurd's goals and description of active and passive translators in detail. Hurd's security infrastructure is mentioned as well.

Chapter 4 — focuses on the Mach μ-kernel. The Mach key features, interprocess communication, memory handling, device drivers and others are mentioned there.

Chapter 5 — describes the L4 μ-kernel in detail. You can find table with various implementations of the L4 there, sections about address spaces, threads, memory mapping and interprocess communication.

Chapter 6 — presents GNU Mach and L4 μ-kernel differences. Mainly interprocess communication, virtual memory management and device drivers differences are described.

Chapter 7 — focuses on Debian GNU/Hurd installation on the x86 architecture. We describe native installation procedure (on computer without OS) and cross installation (on computer with Debian GNU/Linux installed).

Chapter 8 — describes current status of the GNU Hurd. There is shown Hurd's μ-kernel usage history and Hurd's releases overview. Next, you can find in this chapter some flaws and limitations of current Hurd system. Some Linux-related projects are mentioned as well.

Chapter 2

Monolithic Kernels and μ-kernels

2.1 What is a Kernel?

Since the operating system provides an hardware abstraction layer, and since it must provide resource sharing, every access to the hardware from user programs should be done through the operating system. To enforce this in a secure way, and prevent malicious or buggy applications to mess up with the hardware, a protection is needed at the hardware level.

Hardware must provide at least two execution levels:

Kernel mode — In this mode, the software has access to all the instructions and every piece of hardware.

User mode — In this mode, the software is restricted and cannot execute some instructions, and is denied access to some hardware (like some area of the main memory, or direct access to the IDE bus).

So, we define two spaces at software level:

Kernel space — Code running in the kernel mode is said to be inside the kernel space.

User space — Every other programs, running in user mode, is said to be in user space.

2.1.1 Monolithic Kernel Based Systems

This is the traditional design of UNIX systems. Every part which is to be accessed by most programs which cannot be put in a library is in the kernel space:

- Device drivers
- Scheduler
- Memory handling
- File systems
- Network stacks

Many system calls are provided to applications (more than 250 for Linux 2.4), to allow them to access all those services.

This design has several flaws and limitations:

- Coding in kernel space is hard, since you cannot use common libraries (like a full-featured libc), debugging is harder (it is hard to use a source-level debugger like gdb), rebooting the computer is often needed, ... Remember this is not just a problem of convenience to the developer: as debugging is harder, as difficulties are stronger, it is likely that code is "buggier".
- Bugs in one part of the kernel have strong side effects, since every function in the kernel has all the privileges, a bug in one function can corrupt data structure of another, totally unrelated part of the kernel, or of any running program.
- Kernels often become very huge (more than 100 MB of source code for Linux 2.6), and difficult to maintain.
- The presence of strong side effects makes the whole kernel less modular (remember the VM issues of Linux 2.4, some people suggested to make the VM modular, allowing the system administrator to choose one VM at compile time, but the impact of the VM over the other parts of the code was so huge that the idea was dropped). And since kernel code runs with all the privileges at hardware level, only the system administrator can be allowed to load a module inside the kernel space.

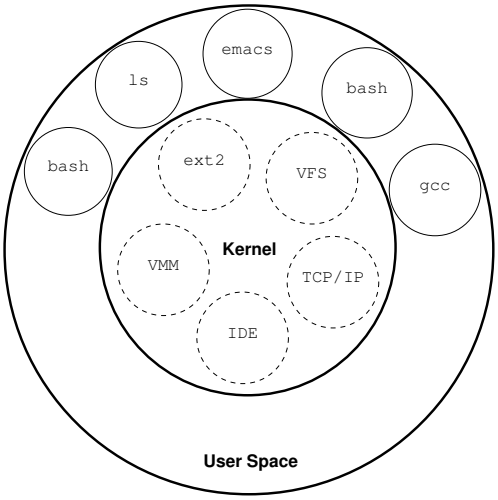


Figure 2.1: Design of monolithic kernel based system

2.1.2 μ-kernel Based Systems

Principles

Only parts which really require to be in a privileged mode are in kernel space:

- IPC (Inter-Process Communication)
- Basic scheduler, or scheduling primitives
- Basic memory handling
- Basic I/O primitives

Many critical parts are now running in user space:

- The main part of scheduler
- Memory handling
- File systems
- Network stacks

Mono-server Systems

A single user-space program (server) handles everything that belonged to the kernel. The two most common examples are MachOS and L^{4}Linux (a port of Linux as a user-space server on top of the L4 microkernel). This split allows better hardware independence of the operating system itself, a slightly easier development, and a limited improvement in overall security (since the code running in the user-space, the server cannot directly access the hardware).

But this design still has most drawbacks of monolithic systems: if the monolithic server crashes, the whole system crashes; it is impossible to add code to it without being root, changing most of the code requires a reboot, etc.

Multi-server Systems

All features are now split into a set of communicating processes, with each of them handling only a very specific task (like a TCP/IP server or an ext2fs server). This modularity allows to replace components easily, an easier developement, a far better fault-tolerance (since a crash of one of servers cannot corrupt the internal state of any other), and far more flexibility for the end user.

But, like always in computer science, all those benefits come with several drawbacks: the communication between all servers can slow down the whole system, and the definition of a strict set of interfaces and protocols for communication between those servers is an extra work to do.

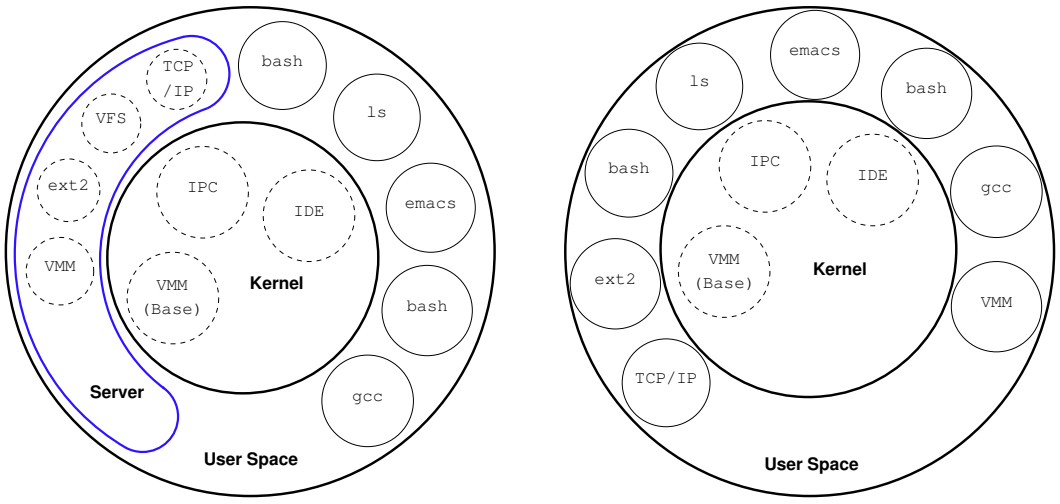


Figure 2.2: Schemes of mono-server and multi-server based system

Chapter 3

The GNU Hurd

3.1 The GNU Project

The GNU Project was started by Richard M. Stallman in 1983. The goal of the project is to create a full operating system and all applications required to allow any user to be able to use only Free Software to perform whatever task he needs to use a computer for. GNU means *GNU is Not Unix*, since GNU takes many ideas from UNIX, but does not intend to be "only" a Free implementation of UNIX, but to correct UNIX flaws and weaknesses at the same time.

3.2 The Hurd's Definition

The GNU Hurd is a set of servers, libraries and interfaces, running on top of a μ-kernel[1], and providing the services which used to be inside the kernel.

3.2.1 Vocabulary

The Hurd — The Hurd, or the GNU Hurd, is the set of servers, it is not an operating system, and since it runs in user space, it is not what we call a kernel. *Hurd* means *Hird of Unix-Replacing Daemons* and *Hird* means *Hurd of Interfaces Representing Depth*, but all of them are spelled like the word *herd*, which is the real meaning of this name: the GNU Hurd is a herd of gnus.

GNU/Hurd — GNU/Hurd, or GNU, is the full operating system, including the μ-kernel, the Hurd, the dynamic linker (GNU ld), the C library (GNU libc) and other applications.

3.3 The Hurd's Goals

The Hurd is core of the GNU project. Every program running on the GNU system rely upon the Hurd to perform most of operating system related tasks. The philosophy of the GNU project, as defined in the GNU manifesto, was the philosophy leading to the Hurd design. The goal of the GNU project is to give back the freedom to the users of computer

[1] Hurd has been originally developed on the top of GNU Mach, currently there is an effort to replace Mach with another μ-kernel (see section 8.1).

Figure 3.1: The GNU Hurd's logo

systems; but not only at the license level. Every technical limitation which is not strictly necessary is a reduction of user freedom. The fact that non-privilegied user in a UNIX system cannot mount an ISO image (setuid does not count), or test his own file system implementation, is a reduction of user freedom. With its highly modular design, the Hurd gives back some additional freedom to the users. This is a major idea of the whole GNU project, displayed even in the name of the project: we want to stay compatible with UNIX programs as much as possible, but we want to overthrow as much limitations as we can.

Interfaces between components of the GNU Hurd were clearly defined and fixed as soon as possible during the development. Fixed interfaces are very important to allow users to design their own replacement of some part of the system, without breaking the other parts. It also suppress compatibility problems.

Interfaces of the GNU Hurd were designed to fill the flaws of UNIX systems. For example, there is no (standard) way in a UNIX system to create a file with no name. The commonly used method is to create it in `/tmp`, and then unlink it without closing the file descriptor. The kernel will wait for the file descriptor to be closed before deleting it. But during a short amount of time, the file has a name, and this causes many security holes. Another example is notification: the most common way to ask the kernel of UNIX system to be informed when a file is changed (SIGIO) is non-portable, and not very flexible (in recent UNIX kernels, other ways can exist, but still lack flexibility).

3.4 The Hurd's History

1983 Richard Stallman starts the GNU project
1988 Mach 3 is chosen as μ-kernel
1991 Mach 3 is released under a Free license
1991 Thomas Bushnell, BSG, founds the Hurd
1994 GNU/Hurd boots for the first time
1997 The Hurd version 0.2 is released
1998 Marcus Brinkmann creates Debian GNU/Hurd
2002 GNU Mach version 1.3 is released
2002 Debian GNU/Hurd has 4 CDs
2002 Port of the Hurd to L4 is started
2002 POSIX threads are now supported
2003 L4Ka::Pistachio 0.1 is released
2003 Ext2fs without the 2GB limit in alpha stage
2004 Ext2fs without the 2gb limit reach release candidate
2005 Ext2fs without the 2gb in Debian GNU/Hurd
2005 First program running on L4Hurd
2005 Initial Gnome port
2005 Debian GNU/Hurd K10 series is released

3.5 Servers and Translators

3.5.1 Translator Concept

Before we take a closer look at translators, let us consider regular filesystems. A filesystem is a store for a hierarchical tree of directories and files. You access directories and files by a special character string, the path. Furthermore, there are symbolic links to refer to one file at several places in the tree, there are hard links to give one and the same file several names. There are also special device files for communication with the hardware device drivers of the kernel, and there are mount points to include other stores in the directory tree. Then there are obscure objects like FIFOs.

Although these objects are very different, they share some common properties, for example, they have all an owner and a group associated with them as well as access rights (permissions). This information is written in inodes. This is actually a further commonality: Every object has exactly one inode associated with it (hard links are somewhat special as they share one and the same inode). Sometimes, the inode has further information stored in it. For example, the inode can contain the target of a symbolic link.

However, these commonalities are usually not exploited in the implementations, despite the common programming interface to them. All inodes can be accessed through the standard POSIX calls, for example `read()` and `write()`. For example, to add a new object type (for example a new link type) to a common monolithic UNIX kernel, you would need to modify the code for each filesystem separately.

In the Hurd, things work differently. Although in the Hurd a special filesystem server can exploit special properties of standard object types like links (in the ext2 filesystem with fast links, for example), it has a general interface to add such features without modifying existing code.

The trick is to allow a program to be inserted between the actual content of a file and the user accessing this file. Such a program is called a translator, because it is able to process the incoming requests in many different ways. In other words, a translator is a Hurd server which provides the basic filesystem interface.

Translators have very interesting properties. From the kernel's point of view, they are just another user process. This means, translators can be run by any user. You don't need root privileges to install or modify a translator, you only need the access rights for the underlying inode the translator is attached to. Many translators don't require an actual file to operate, they can provide information by their own means. This is why the information about translators is stored in the inode.

Translators are responsible to serve all file system operations that involve the inode they are attached to. Because they are not restricted to the usual set of objects (device file, link etc), they are free to return anything that makes sense to the programmer. One could imagine a translator that behaves like a directory when accessed by `cd` or `ls` and at the same time behaves like a file when accessed by `cat`.

3.5.2 Translator Examples

Mount Points

A mount point can be seen as an inode that has a special translator attached to it. Its purpose would be to translate filesystem operations on the mount point in filesystem operations on another store, let's say, another partition.

Indeed, this is how filesystems are implemented under the Hurd. A filesystem is a translator. This translator takes a store as its argument, and is able to serve all filesystem operations transparently.

Device Files

There are many different device files, and in systems with a monolithical kernel, they are all provided by the kernel itself. In the Hurd, all device files are provided by translators. One translator can provide support for many similar device files, for example all hard disk partitions. This way, the number of actual translators needed is quite small. However, note that for each device file accessed, a separate translator task is started. Because the Hurd is heavily multi threaded, this is very cheap.

When hardware is involved, a translator usually starts to communicate with the kernel to get the data from the hardware. However, if no hardware access is necessary, the kernel does not need to be involved. For example, `/dev/zero` does not require hardware access, and can therefore be implemented completely in user space.

Symbolic Links

A symbolic link can be seen as a translator. Accessing the symbolic link would start up the translator, which would forward the request to the filesystem that contains the file the link points to.

However, for better performance, filesystems that have native support for symbolic links can take advantage of this feature and implement symbolic links differently. Internally, accessing a symbolic link would not start a new translator process. However, to the user, it would still look as if a passive translator is involved (see below for an explanation what a passive translator is).

Because the Hurd ships with a symlink translator, any filesystem server that provides support for translators automatically has support for symlinks (and firmlinks, and device files etc)! This means, you can get a working filesystem very fast, and add native support for symlinks and other features later.

3.5.3 Passive and Active Translators

There are two types of translators, *passive* and *active*. They are really completely different things, so don't mix them up, but they have a close relation to each other.

Active Translators

An active translator is a running translator process, as introduced above. You can set and remove active translators using the `settrans -a` command. The `-a` option is necessary to tell `settrans` that you want to modify the active translator.

The `settrans` command takes three kind of arguments. First, you can set options for the `settrans` command itself, like `-a` to modify the active translator. Then you set the inode you want to modify. Remember that a translator is always associated with an inode in the directory hierarchy. You can only modify one inode at a time. If you do not specify any more arguments, `settrans` will try to remove an existing translator. How hard it tries depends on the force options you specify (if the translator is in use by any process, you will get "device or resource busy" error message unless you force it to go away).

But if you specify further arguments, it will be interpreted as a command line to run the translator. This means, the next argument is the filename of the translator executable. Further arguments are options to the translator, and not to the `settrans` command.

For example, to mount an ext2fs partition, you can run `settrans -a -c /mnt /hurd/ext2fs /dev/hd2s5`. The `-c` option will create the mount point for you if it doesn't exist already. This does not need to be a directory, by the way. To unmount, you would try `settrans -a /mnt`.

Passive Translators

A passive translator is set and modified with the same syntax as the active translator (just leave away the `-a`), so everything said above is true for passive translators, too. However, there is a difference: passive translators are not yet started.

This makes sense, because this is what you usually want. You don't want the partition mounted unless you really access files on this partition. You don't want to bring up the network unless there is some traffic and so on.

Instead, the first time the passive translator is accessed, it is automatically read out of the inode and an active translator is started on top of it using the command line that was stored in the inode. This is similar to the Linux automounter functionality. However, it does not come as an additional bonus that you have to set up manually, but an integral part of the system. So, setting passive translators defers starting the translator task until you really need it. By the way, if the active translator dies for some reason, the next time the inode is accessed the translator is restarted.

There is a further difference: active translators can die or get lost. As soon as the active translator process is killed (for example, because you reboot the machine) it is lost forever. Passive translators are not transient and stay in the inode during reboots until you modify them with the `settrans` program or delete the inodes they are attached to. This means, you don't need to maintain a configuration file with your mount points.

One last point: Even if you have set a passive translator, you can still set a different active translator. Only if the translator is automatically started because there was no active translator the time the inode was accessed the passive translator is considered.

3.5.4 Managing Translators

As mentioned above, you can use `settrans` to set and alter passive and active translators. There are a lot of options to change the behaviour of `settrans` in case something goes wrong, and to conditionalize its action. Here are some common usages:

- `settrans -c /mnt /hurd/ext2fs /dev/hd2s5` mounts a partition, the translator will stay across reboots.
- `settrans -a /mnt /hurd/ext2fs ~/dummy.fs` mounts a filesystem inside a data file, the translator will go away if it dies.
- `settrans -fg /nfs-data` forces a translator to go away.

You can use the `showtrans` command to see if a translator is attached to an inode. This will only show you the passive translator though.

You can change the options of an active (filesystem) translator with `fsysopts` without actually restarting it. This is very convenient. For example, you can do what is called

"remounting a partition read-only" under Linux simply by running `fsysopts /mntpoint --readonly`. The running active translator will change its behaviour according to your request if possible. `fsysopts /mntpoint` without a parameter shows you the current settings.

3.5.5 The GNU Hurd's Translators and Servers

auth — The authentication server which passes credentials when two mutually untrusting servers communicate. In a sense, each *auth* server establishes a domain of trust.

console — A translator that provides virtual consoles.

crash — The server gets active whenever a task gets a fatal error signal, for example because it violates memory boundaries (segmentation fault). The *crash* server has three modes of operation: suspending the process group (pgrp) of the offending task, killing it or dumping a core file.

exec — The execute server that manages the creation of a new process image from the image file.

ext2fs — A server that manages ext2-type filesystems. It does the same as *ext2fs.static*, only that *ext2fs.static* is a statically linked executable.

fakeroot — A translator for faking privileged access to an underlying filesystem.

fatfs — A translator that manages FAT-type filesystems.

fifo — A translator that implements named pipes.

firmlink — The firmlink translator is sort of halfway between a symbolic link and a hard link.

ftpfs — A translator for FTP filesystems.

fwd — When accessed, the *fwd* translator forwards requests to another translator. It is used in the *fifo* and *symlink* translators. The idea is so that you don't get a jillion servers for such trivial things; *fwd* is used to coordinate having one server handle several different nodes conveniently.

hello — A translator providing a warm greeting. *hello-mt* is the same, but multi-threaded.

hostmux — A translator for invoking host-specific translators.

ifsock — A server which only handles `S_IFSOCK` filesystem nodes for filesystems which don't do it themselves, acting as a hook upon which to hang UNIX domain socket addresses. *pfinet* and *pflocal* implement the socket API.

init — The initialisation server for system boot procedures and basic runtime configurations.

iso9660fs — A server for iso9660-type filesystems, commonly used on compact disks. Server does the same as *iso9660fs.static*, only that *iso9660fs.static* is a statically linked executable.

mach-defpager – A standalone version of the Mach default pager.

magic — A translator that returns the magic retry result `MAGIC`. Normal end users probably need not to know much about it since it is used, for example, to facilitate terminal I/O.

new-fifo — Alternative translator for named pipes.

nfs — A translator that supports Sun's Network File System.

null — The kitchen sink. A server for lots of free space and countless numbers of zeroes, implements `/dev/null` and `/dev/zero`.

password — The Hurd standard password server.

pfinet — A server for TCP/IP, which implements the (IPv4) `PF_INET` protocol family.

pflocal — This server implements UNIX domain sockets. Needed for pipes, for example.

proc — The process server that assigns PID's and process structures to tasks, and manages all the process level stuff like wait, bits of fork, C library support.

proxy-defpager — This translator allows access to control interfaces of Mach default pager.

storeio — The storage translator for devices and other stores.

streamio — A translator for stream devices.

symlink — A translator for symbolic links for filesystems which don't support it themselves.

term — The terminal translator that implements a POSIX terminal.

tmpfs — A translator which provides tmp filesystem.

ufs — A server for ufs-type filesystems. It does the same as *ufs.static*, only that *ufs.static* is a statically linked executable.

usermux — A translator for invoking user-specific translators.

3.6 Security Infrastructure

3.6.1 Authentication Tokens

The security on GNU/Hurd is based on authentication tokens. A token is the right for an application to perform a specific set of tasks. Tokens are handled by the *auth* server, a server trusted by all other pograms, and enforcing that no one lies on which token they have. Through auth, tokens can be given, destroyed, or created. Tokens can exist for anything, for example you can imagine a token "is able to bind ports below 1024".

Tokens are comparable to Kerberos tickets, or POSIX capatibilities.

3.6.2 POSIX Compatibility

POSIX compatibility, with regard to authentication, is implemented via specific tokens: UIDs and GIDs are only a kind of tokens that auth can deliver. It is therefore possible for an application to have several UIDs tokens, and as such being able to act with different POSIX identities at the same time. Programs can lose or give UIDs at any time during their execution — they just need to contact auth, and the password server (or any other auth-trusted server that can manage authentication, for example a server that matches a user-given and a sysadmin-provided certificates).

A specific `addauth` command (a normal, non-suided program) can give tokens to programs. If you are running a shell as *kilobug*, you can give the *UID kilobug* security token to any currently running application with `addauth -u kilobug -p [PID]`. The same way, programs can drop their tokens, lowering their permissions.

Suid-ed Binaries

The translator in charge of a filesystem is the one enforcing the suid-bit. If you run `/bin/ping` as a normal user, the ext2fs translator managing the `/` filesystem will give the root security token to the ping program, before starting it. Since a translator cannot give a security token it doesn't have, a translator ran by a normal user would not be able to enfore the suid-bit, and this way there is no security risk in allowing normal users to run translators.

3.6.3 Some Applications

The Password Server

The password server is a very simple (around 200 code lines) program, which can give UID and GID security tokens in exchange of a login/password pair. A ftp server, or an ssh server, could then run without any permission, and gives the user-provided login/password to the password server, in order to gain privileges and be able to answer to the user. The huge difference with UNIX systems is that the ftp or ssh server never has root privileges (or only at bind-time, and then drop it completly), and never has even the privileges of a normal user before someone gives it a valid login/password pair. A flaw inside ftp or ssh would only give a shell with very few rights.

No Auth Programs

Programs can discard all the security tokens they have and become *noauth* programs. This can be used to process untrusted contents, like a ghostscript interpreter running on contents coming from an untrusted source. A security flaw inside the interpreter could not allow a malicious postscript file to damage your own files, since the interpreter discarded the *UID* security token before processing the data.

Chapter 4

The Mach μ-kernel

4.1 About Mach

Mach was one of the first μ-kernels. It was a project of Carnegie-Mellon university to implement a fairly new theory. It came with a lot of new concepts:

- A complex and powerful IPC layer,
- It was designed for multiprocessor and even clusters,
- It use external pagers,
- It was the first system to clearly define the notions of *task*, *thread*

Mach was the first μ-kernel to be successful, taken and improved by OSF/1 and other research groups. It was the base of MachOS, a mono-server operating system (with the UX server running on top of the Mach μ-kernel and providing BSD compatibility).

4.2 Mach Key Features

Mach does quite a lot of things for a μ-kernel:

- It handles tasks as containers (a task contains memory areas, threads, IPC rights),
- A complex IPC system,
- The virtual memory layer, with an LRU decision algorithm,
- A basic scheduler,
- Device drivers.

4.3 GNU Mach

GNU Mach was derived from OSF Mach. The last stable version is 1.3, and maybe it is the last of the 1.x branch. The new 2.0 version, might be released sometime in the future, uses the OSKit framework for device drivers (allowing nearly all drivers from Linux 2.2 to be ported easily).

4.4 Interprocess Communication (IPC)

IPC with Mach is based on ports. A *port* is kernel managed entity which can be seen as a message queue, with one task having a *receive right* on the port, and any number of tasks having *send rights* to the port. The task having the receive right can read the messages sent by the tasks having send rights. Receive rights can be given to another task, and send rights can be given, destroyed or duplicated (rights are sent across tasks using IPCs). There is also a special *send once* right, which can be used only once, and is often used when waiting for an answer.

Communication across ports is asynchronous: a sending thread is not blocked until the receiving task received the message. This implies that all IPC messages must be copied (either logically or physically) inside the kernel and queued there, which slows down the IPC a lot.

4.5 Memory Handling in GNU/Hurd with Mach

4.5.1 Paging with Mach

Mach takes the decision of which page to keep and which page to discard (move to back-end storage) in time of memory pressure, using a Least Recently Used algorithm (LRU); but the pagers are in user-space. When a page has to be evicted from memory, Mach sends an IPC to the user-space pager associated with the page; it is up to the pager to save it somewhere. When a page fault occurs, Mach asks to the pager to bring back the page to memory, and then resumes the faulting application.

4.5.2 Some Applications

User space pagers can be used to implement different kind of backing stores: hard disks, compressed memory, a remote computer through networking, ... Several pagers can handle different parts of the address space of the same task.

The Use in diskfs

The library used by *regular* filesystem translators (like ext2fs, iso9660fs, fatfs, ...) works by mapping in memory all the metadata, and then using only pointer indirection to access to the metadata. The GNU Mach VM handles all the caching, and a user-space pager is used to load and write pages to/from the backing store.

The problem is that, for filesystems with metadata spread all over the partition (like ext2fs), the whole partition has to be mapped into memory. That is why many diskfs translators are limited to 2GB partitions on IA-32 (and that's why fatfs isn't limited to 2 GB — the FAT is located at the beginning of the partition, so the only limitation is to have the FAT be less than 2 GB).

Two possible solutions for this problem are either to set-up a tree of intelligent specialized pagers, or to use a cache of mappings, creating and destroying mappings of meta-data as needed. The current implementation of ext2fs in Debian GNU/Hurd (see section 7.1) contains a patch from Ognyan to cover this problem. It implements a caching of mappings, with static mapping of fixed metadata (since in ext2, some metadata are in fixed places, and others can be anywhere in the filesystem).

4.6 Device Drivers

The Mach kernel provides all low-level device support. Each device is represented as a port to which messages can be sent to transfer data or control the device. Data is transferred through read and write operations; the request and reply messages are exported separately, allowing both synchronous and asynchronous styles of I/O. The external memory object protocol allows a user to map the frame buffer for a graphics device directly into its address space.

Chapter 5

The L4 μ-kernel

5.1 About L4

Originally, L4 is the name of a second-generation μ-kernel designed and implemented around 1995 by Jochen Liedtke, running on i486 and Pentium CPUs.

However, there are numerous implementations of the L4 API on several hardware architectures. You can see overview of various implementations in table 5.1 (the last one is Liedtke's original implementation).

Note that this chapter is focused on the L4Ka::Pistachio implementation, which GNU Hurd uses.

Name	**CPU**	**Impl. Lng.**
L4Ka::Pistachio	Pentium or better, IA64, PowerPC, Alpha, 64-bit MIPS	C++
NICTA::L4-embedded	ARM, x86 and MIPS	C++
Fiasco	i486 or better, StrongARM, Linux (user-mode L4 emulation)	C++
P4	x86, MIPS, PowerPC, ARM	C
L4 for PowerPC	PowerPC 603e	C
L4Ka::Hazelnut	Pentium or better, StrongARM	C++
L4/MIPS	MIPS R4x00	C
L4/Alpha	Alpha AXP 21264	Assembler
L4/x86	i486 or better	Assembler

Table 5.1: Various implementations of the L4 μ-kernel

5.2 Design Philosophy

The most fundamental task of an operating system is to provide secure sharing of resources. A μ-kernel should to be as small as possible. Hence, the main design criterion of the μ-kernel is minimality with respect to security: *A service (feature) is to be included in the μ-kernel if and only if it impossible to provide that service outside the kernel without loss of security.* The idea is that once we make things small (and do it well), performance will look after itself.

A strict application of this rule has some surprising consequences. Take, for example, device drivers. Some device drivers access physical memory (e.g. DMA) and can therefore break security. They need to be trusted. This does not, however, mean that they need to be in the kernel. If they do not need to execute privileged instructions and if the kernel can provide sufficient protection to run them at user level, then this is what should be done. Consequently, this is what L4 does.

According to Liedtke, and based on such reasoning, a μ-kernel must provide:

Address spaces — the basis of protection,

Threads — an abstraction of program execution,

Interprocess communication (IPC) — a mechanism to transfer data between address spaces,

Unique identifiers (UIDs) — providing context-free addressing in IPC operations.

All these concepts will be explain in this chapter later.

5.3 Registers and Address Spaces

5.3.1 Registers

L4 μ-kernel provides programs with access to virtual registers. Virtual registers offer a fast interface to exchange data between the μ-kernel and user threads. They are registers in the sense that they are static per-thread objects. Depending on the specific processor type, they can be mapped to hardware registers or to memory locations. Mixed mappings, where some virtual registers map to hardware registers while others map to memory, are also possible.

There are three classes of virtual registers: *Thread Control Registers* (TCRs), *Message Registers* (MRs) and *Buffer Registers* (BRs).

In general, virtual registers can only be addressed directly, not indirectly through pointers. The L4 API provides specific functions for accessing the three different classes of registers. Loading illegal values into virtual registers, overwriting read-only virtual registers, or accessing virtual registers of other threads in the same address space (which may be physically possible if some are mapped to memory locations) is illegal and can have undefined effects on all threads of the current address space.

5.3.2 Address Space

An address space contains all the data that is directly accessible by a thread. It consists of a set of mappings from virtual to physical memory. This set of mappings is *partial* in the sense that many mappings may be undefined, making the corresponding virtual memory inaccessible. Figure 5.1 shows an example of how the virtual memory in an address space may map onto physical memory. The regions of virtual memory that are not mapped are inaccessible to threads running in that address space.

Typically an address space will contain a number of standard regions. These include the text, data, heap, and stack regions. The text region contains program code, the data region contains preinitialised data used by the program, the heap region is used for dynamically allocated data, and the stack region is used for storing temporary data during execution.

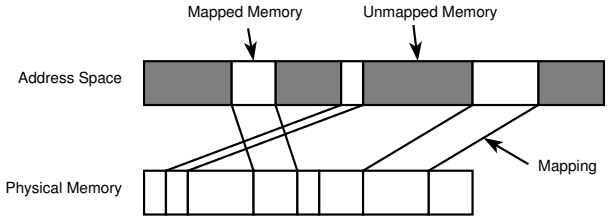


Figure 5.1: Mapping of virtual memory to physical memory

The text and data regions usually have a fixed size and do not change during execution of a program. The heap and stack regions, on the other hand, can grow or shrink while the program is executing. Note that by convention on most architectures the heap grows up toward high memory, while the stack grows down toward lower memory.

Figure 5.2 shows a typical arrangement of these regions in an address space. Note that L4 does not enforce any specific layout of address spaces. The layout is generally determined by a combination of the compiler and higher-level operating system services built on top of L4.

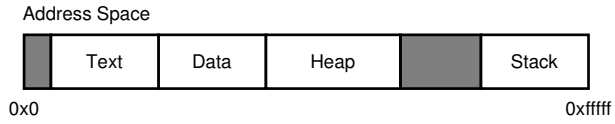


Figure 5.2: A typical address space layout

5.4 Threads

A *thread* is the basic execution abstraction in L4. L4 threads are light-weight and cheap to manage. The light-weight thread concept together with fast IPC are the keys to the efficiency of L4 and OS personalities running on top of L4.

A thread is always associated with a specific address space. The relationship between threads and address spaces is $1 : n$, that is, a thread is associated with exactly one address space, while any address space can have many threads associated with it.

Each thread has its own set of virtual registers called the *thread control registers* (TCRs). These virtual registers are static (they keep their values until explicitly modified) and store the thread's private state (e.g., parameters for IPCs, scheduling information, the thread's identifiers, etc.). They are stored in an area of the address space called the *thread control block* (TCB). The TCB is actually split into two parts, there is the *user TCB* (UTCB), which is accessible by the thread, and the *kernel TCB* (KTCB) which is accessible only by the kernel.

Each thread is further associated with a *page-fault handler* and an *exception handler*. These are separate threads that are set up to handle page faults and other exceptions caused by the thread.

Each thread in an address space has its own stack. A thread's stack address is explicitly specified during thread creation. It is up to the thread to determine where to place its heap

— this is a design issue, but typically several threads in the same address space will share a heap.

L4 also distinguishes between *privileged* and *non-privileged* threads. Any thread belonging to the same address space as one of the initial threads created by the kernel upon boot-time is treated as privileged. Some system calls can only be executed by privileged threads.

Threads can be created as *active* or *inactive* threads. Inactive threads do not execute but can be activated by active threads that execute in the same address space. They are typically used during the creation of address spaces. A thread created active starts executing immediately after it is created. The first thing it does is execute a short receive operation waiting for a message from its pager. This message will provide the thread with an instruction and stack pointer and allow it to start executing its code.

5.4.1 Tasks

A *task* is the set of of threads sharing an address space. The terms task and address space are often used interchangeably, although strictly speaking they are not the same. We will try to avoid using the term task where possible, prefering to specifically refer to an address space or thread.

5.4.2 Identifying Threads and Address Spaces

A thread is identified by its unique identifier (UID). A thread can actually have two identifiers, a *local* identifier and a *global* identifier. While a global identifer is valid in any address space, a local identifier is only valid within the thread's address space. That is, a global identifier can be used by any thread, while a local identifier can only be used by threads that are part of the same task. In different address spaces, the same local thread ID may identify different and unrelated threads.

Unlike threads, address spaces do not have identifiers. Instead, an address space is identified by the UID of any thread associated with that address space. This means that an address space must always have at least one thread associated with it. That thread does not, however, have to be active.

5.5 Communication

One of the main activities that threads engage in is to communicate with other threads (for example, in order to request services from each other, in order to share results of computations, etc.). There are two ways that threads communicate: using shared memory (see section 5.6), or using L4's Interprocess Communication (IPC) facilities (see section 5.7).

5.5.1 Communication Within an Address Space

When communicating with threads in the same address space, it is easiest (and most efficient) to use shared memory. Threads in the same address space automatically share memory, so they do not have to make use of L4's memory mapping facilities. As long as both threads agree on which shared memory region (or variables) to use, they are free to communicate in this way.

When threads communicate using shared memory it is necessary to avoid race conditions. This is best done by enforcing mutual exclusive access to shared memory. Note that L4 does not provide any mutual exclusion primitives (such as semaphores) to do this, it is expected that these are provided at user level (possibly using mechanisms provided by the underlying hardware). Various implementations of user level mutual exclusion primitives are available.

It is possible for threads in the same address space to communicate using IPC. The main use for this is thread synchronisation. Addressing a thread in the same address space can be done using local or global thread Ids.

5.5.2 Communication Between Address Spaces

When communicating between address spaces (i.e., when threads in different address spaces communicate with each other) both the use of shared memory and IPC are valid approaches. IPC is generally used for smaller messages and synchronisation, while shared memory is used to exchange larger amounts of data.

Communicating between address spaces using shared memory requires that all (communicating) threads have access to the same memory region. This is achieved by having one thread map a region of its address space into the address spaces of the other threads. The concept of mapping memory is explained in section 5.6. Once a shared region of memory has been established, the threads communicate by simply reading from or writing to the particular memory region. Note that, as mentioned previously, when communicating using shared memory it is necessary to avoid race conditions.

Communication using IPC requires that the threads send messages to one another. A message is sent from a sender thread and addressed to a particular receiver thread. Messages can be used to directly share data (by sending it back and forth), to indirectly share data (by sending memory mappings), or as a control mechanism (e.g., to synchronise).

5.6 Memory Mapping

Address spaces can be recursively constructed. A thread can *map* parts of its address space into another thread's address space and thereby share data. Figure 5.3 shows an example of two address spaces with a region of address space A mapped into address space B. Both the thread running in address space A and the thread running in address space B can access the shared region of memory. Note, however, that the memory may have different virtual addresses in the different address spaces. Thus, in the example, a thread running in address space A accesses the shared memory region using virtual address `0x1000000` while a thread in address space B uses virtual address `0x2001000`.

A *mapper* (that is, the thread making the memory available) retains full control of the mapped region of memory. In particular, the mapper is free to revoke a mapping at any time. Revoking a mapping is called *unmapping*. After a mapping has been revoked, the receiver of that mapping (the *mappee*) can no longer access the mapped memory. This is shown in figure 5.4. Here a thread in address space B can no longer access the region of virtual memory that used to contain the mapped memory.

Access to mapped memory is limited by the access permissions set by the mapper. These access permissions specify read permission, write permission, and execute permission and determine how the mappee can access the mapped memory. Note that a mapper cannot grant access rights that it does not itself have. Thus, if a thread does not have write access

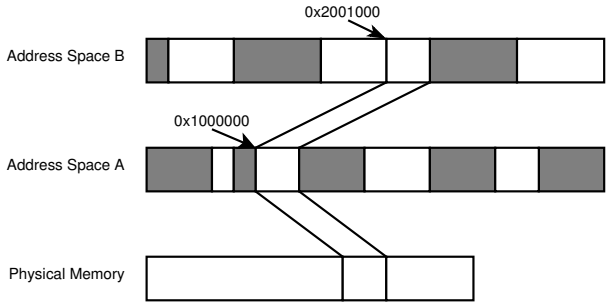


Figure 5.3: Two address spaces sharing a region of memory

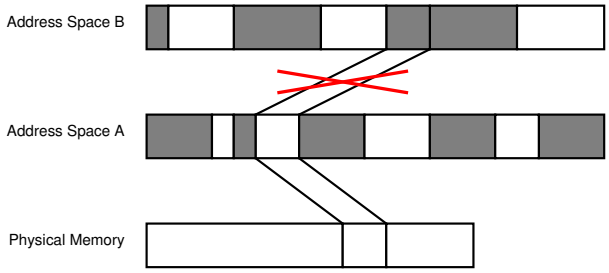


Figure 5.4: A shared region of memory is unmapped

to a particular region of memory, it cannot map that memory with write permission into another address space.

It is also possible for a region of one address space to be *granted* to another address space. Granting differs from mapping in that after the grant has succeeded, the granter loses access to that region of its address space (i.e., it no longer has a valid mapping for that region). Figure 5.5 shows an example of the situation before and after a region of address space A has been granted to address space B. Unlike revocation of mappings, a granter cannot revoke a grant, as it no longer has access to the page.

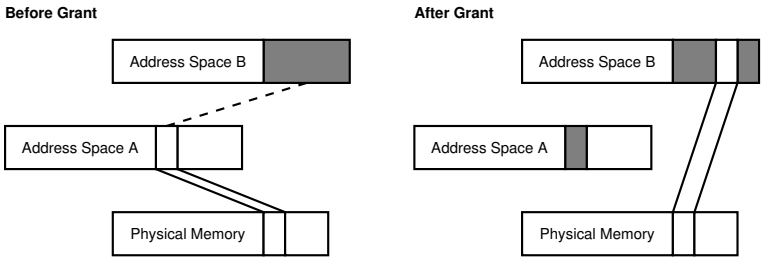


Figure 5.5: A region of one address space is granted to another

Note that, for security reasons, when mapping or granting memory the receiver must always explicitly agree to accept maps or grants.

Mapping and granting of memory are implemented using L4's IPC mechanism. In order to map memory, a mapper sends the intended mappee a message containing a *map item* specifying the region of memory to be mapped. The mappee must explicitly specify that it

is willing to receive mappings. It also specifies where, in its own address space, the memory should be mapped. The receiver does not actually have to do anything with the received map item (or grant item). The mapping is performed as a side effect of the IPC.

Note that although mappings apply to an address space, map messages are sent via IPC to a thread in that address space.

5.7 Interprocess Communication (IPC)

Message passing is the basic *interprocess communication* (IPC) mechanism in L4. It allows L4 threads in separate address spaces to communicate by sending messages to each other. This message-passing IPC is the heart of L4. It is used to pass data between threads (either by value, with the μ-kernel copying the data between two address spaces, or by reference, through mapping or granting). L4 IPC is blocking, so it is used for synchronisation (each successful IPC operation results in a *rendez-vouz*) and wakeup-calls (timeouts can be specified, so IPC can be used for timed sleeps). It is even used for memory management (the μ-kernel converts a page fault into an IPC to a user-level pager), exception handling (the μ-kernel converts an exception fault into an IPC to a user-level exception handler), and interrupt handling (the μ-kernel converts an interrupt into an IPC from a pseudo-thread to a user-level interrupt-handler).

5.7.1 Messages

A message consists of one mandatory and two optional sections. The mandatory *message tag* is followed by the optional *untyped-words* section which is followed by the optional *typed-items* section. The message tag contains message control information and a message label. The message control information specifies the size of the message and the kind of data contained in it. The μ-kernel associates no semantics with the message label; it allows threads to identify a message and is often used to encode a request key or to identify the function to be invoked upon reception of the message.

The untyped-words section holds arbitrary data that is untyped from the μ-kernel's point of view. The data is simply copied to the receiver. The μ-kernel associates no semantics with it. The typed-items section contains typed data such as string items, map items, and grant items. Map items and grant items were introduced earlier and are used to map and grant memory. String items are used for passing string data by reference.

5.7.2 Message Registers

IPC messages are transferred using *message registers* (MRs). A sender writes a message into the message registers associated with its own thread and a receiver reads the message out of the message registers associated with its thread. Each thread has 64 MRs, numbered MR_0 to MR_{63} (inclusive). A message can use some or all MRs to transfer untyped words and typed items. The message tag is always transfered in MR_0.

MRs are transient read-once virtual registers. Once an MR has been read, its value is undefined until the MR is written again. The send phase of an IPC implicitly reads all MRs; the receive phase writes the received message into MRs.

MRs can be implemented as either special purpose (hardware) registers, general memory locations, or general purpose (hardware) registers. It is generally up to the particular L4 implementation (and the hardware that it is implemented on) whether any MRs are

implemented as hardware registers and if so which ones. For example, in the MIPS implementation MR_0 to MR_9 are implemented as hardware registers.

5.7.3 Acceptor and Buffer Registers

In order to be able to handle a received message, the receiver must explicitly agree to accept messages of that type. The *acceptor* is used to specify which typed items will be accepted when a message is received. If an acceptor specifies that map or grant items are accepted, then it also specifies where the associated memory will be mapped in the receiver's address space. If an acceptor specifies that string items are accepted, then any string items received are placed in the *buffer registers* (BRs).

BRs are registers in the sense that they are per-thread objects and can only be addressed directly, not indirectly through pointers. BRs are static objects like TCRs, i.e., they keep their values until explicitly modified. BRs can be mapped to either special registers or to memory locations. There are 34 BRs numbered from BR_0 to BR_{33} (inclusive).

The acceptor is always placed in BR_0. Any string items received are placed in the BRs starting at BR_1.

5.7.4 Send and Receive

Messages are sent and received through the *IPC* system call. IPC is the fundamental operation for inter-process communication and synchronization. It can be used for intra- and 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 until a period specified by the sender has elapsed without the destination becoming ready to receive. Similarly, the reciever blocks until a message has been recieved. The receiver must also supply all necessary buffers, as the kernel does not provide any such storage. This unbuffered operation reduces the amount of copying involved and is the key to high-performance IPC.

A single IPC call combines an optional send phase and an optional receive phase. Which phases are included is determined by specific parameters to the IPC call. It is also possible to specify timeouts for IPCs. Different combinations of timeouts and send and receive phases lead to fundamentally different kinds of IPCs. For example, including both a send and a receive phase with no timeout implements a synchronous IPC that blocks until a reply is received, while an IPC that includes only a receive phase and no (or rather infinite) timeout implements a blocking call that waits for a message to arrive. The L4 API provides convenience interfaces to these different kinds of IPC calls.

To enable implementation-specific optimizations, two variants of the IPC system call exist: normal IPC and *lightweight IPC* (LIPC). Functionally, both variants are identical, but LIPC may be optimized for sending messages to local threads. Transparently to the user, a kernel implementation can unify both variants or implement differently optimized functions. *IPC* is the default IPC function. It must always be used, except if all criteria for using *LIPC* are fulfilled: both a send and receive phase are included, the destination thread is specified as a local thread ID, the destination thread runs on the same processor, there is a short receive timeout, and the IPC includes no map/grant operations. These restrictions allow LIPC to be up to an order of magnitude faster than regular IPC.

Chapter 6

GNU Mach and L4 Differences

As you can see in previous chapters 4 and 5, there are some differences between these two μ-kernels. Currently the Hurd runs on Mach, but as mentioned in chapter 3, there is an effort to replace Mach with L4 or another μ-kernel, because Mach has some serious lacks.

6.1 GNU Mach

– GNU Mach is big and slow (more than 100 system calls).

– Mach runs only on IA32 systems.

IPC

- A Mach IPC message is sent to a task instead of a thread. This is because ports are shared by all the threads in a task.
- IPC is asynchronous (messages sent by IPCs are queued in kernel-level buffers).

\+ Mach acts as a central object server, providing not only the facility but also defining the policy.

– This policy makes IPC convenient, but slow.

– Message buffering requires in-copy and out-copy.

– Flexible data types require costly message analysis.

– The port right system is very heavyweight.

VMM

- Mach acts as a central virtual memory management server, not only providing access to the physical memory but also defining the policy of virtual memory management.

– It doesn't know enough about how the memory is used to make correct pageout decisions.

Device Drivers

- Mach has built-in device drivers.

– That means a bug in a driver can crash the whole system.

6.2 L4

+ L4 is small and very fast (Pistachio provides only 11 system calls).

+ There are L4 implementations for Alpha, AMD64, ARM, IA32, IA64, MIPS and PowerPC architecture.

IPC

- In L4, a message is always sent to a thread.
- Only primitive security mechanism.
- Every server will manage itself the objects it provides and the handles to them.
- L4 IPC is synchronous (messages are sent to target threads directly), the sending thread blocks until the receiving thread received the message.

+ No copying necessary if all arguments fit into the registers.

+ Limited support for data types makes message analysis easy.

VMM

- L4 provides only access to the physical pages.
- Instead of implementing a single VMM server, distributed approach was proposed.
- Every task enters a contract with the memory server about guaranteed physical pages and some extra pages.
- Guaranteed pages are renegotiated from time to time.
- Extra pages have to be returned on short notice.
- Tasks manage their own virtual memory.

Device Drivers

- L4 has no built-in device drivers — they are implemented in user-space.

– An interface for drivers is needed.

Chapter 7

GNU/Hurd Installation

This installation guide provides a basic instructions for *native* and *cross* installation of the Debian GNU/Hurd to x86 computer. It is to date March 31, 2006 and the guide expects K10 version of the distribution.

7.1 About Debian GNU/Hurd

The GNU Hurd is a totally new operating system being put together by the GNU group. In fact, the GNU Hurd is the final component which makes it possible to built an entirely GNU OS — and Debian GNU/Hurd is going to be one such (possibly even the first) GNU OS.

Debian is a free operating system. Currently, stable version of Debian is only available with Linux kernel, but there are a non-Linux ports such as Debian GNU/NetBSD, Debian GNU/kFreeBSD and Debian GNU/Hurd as well. Because Debian GNU/Hurd is still under active development, therefore hasn't been officially released yet, and won't be for some time.

7.2 Native Installation

7.2.1 Introduction

This section is focused to the Debian GNU/Hurd installation on x86 computer without any operating system. You can have already some OS installed[1], but you need about 2 GB (it depends whatever you want to install) of free space on your hard drive.

There are many ways how to install Hurd. You can use:

- a set of four CD's,
- a one DVD,
- or a mini CD for network installation.

The best way is to use the first CD from the set for the first part of install process and then choose network install for the second one part.[2] That is because Hurd is still unstable and

[1] If you have some, it is highly recommended to do complete backup of your system.

[2] We don't use mini CD, because it doesn't include boot loader and some other utilities that we need for comfortable installation.

a lot of packages have a new version then that in released distribution on the CDs. So it is useless to install all packages from the CDs and then install new versions of these packages from the internet.

For installation you need besides 2 GB of free space some supported network card. A relatively complete hardware compatibility guide can be found at http://www.nongnu.org/thug/gnumach_hardware.html. Don't forget that the GNU Hurd currently runs only on IA32 machines!

7.2.2 CDs Preparation

CD1 of the latest Debian GNU/Hurd distribution can be downloaded from ftp://ftp.gnuab.org/pub/debian-cd/current/hurd-i386/.

Now we burn the downloaded CD1 and look up the file `./install/grub-94.iso` on it. This image includes Grub boot loader that we will need for boot Hurd after install to hard drive. So we burn it as well. Now we have got two CDs (installation CD1 and the Grub CD) and we can start installation of the Debian GNU/Hurd.

7.2.3 Base System Installation — Stage One

When you boot from the CD1, then shows a welcome screen and `boot:` command line. We don't need any special parameters, so press *Enter*. After boot shows Debian GNU/Hurd Installation Main Menu.

You can setup keyboard in 1st menu entry if you want or use default `qwerty/us`.

Next is *Partition a Hard Disk*. We select device and then starts `cfdisk`. I chose this variant:

```
 Name    Flags   Part Type   FS Type     Size (MB)
-------------------------------------------------
 hda1    Boot    Primary     Linux       1800.00
 hda2            Primary     Linux swap   128.00
```

If you have already swap partition, you can share it with another OS (e.g. Linux) and you don't have to create another one.

Note that partition names in `cfdisk` are in the Linux format (because installer uses Linux kernel). GNU Mach (that Debian GNU/Hurd uses) enumerates disks starting at zero. IDE drives are prefixed with `hd`, while SCSI disks are prefixed with `sd`. Like Linux, drives are number by their position on the controller. For instance, the primary master is `hd0` and the secondary slave is `hd3`. Partitions use the BSD slice naming convention and append `s`*`M`* to the drive name to indicate a given partition. *M* is a one, not zero, based index. For example Mach's `hd0s1`, `hd0s2` corresponds to Linux's `hda1`, `hda2`.

The next step is *Initialize and Activate a Swap Partition*. We can skip bad blocks scan and then choose `/dev/hda2` as a swap partition.

Then *Initialize a GNU/Hurd Partition*. Again skip bad blocks scan, use `/dev/hda1` and then mount this partition as root.

The last step is *Install the Base System*. We select CD-ROM installation method and *Choose from the list*. Then the installer copy base system to your hard drive.

And finally *Reboot the system*.

7.2.4 Base System Installation — Stage Two

Now we replace CD1 with the Grub CD and boot from it. Then shows *Grub boot menu*. We choose fourth item:

```
GNU Hurd (IDE 1st partition - hd0s1 single-user)
```

Before you boot the system, you should check boot parameters of this item. It should be like this:

```
root   (hd0,0)
kernel /boot/gnumach.gz -s root=device:hd0s1
module /hurd/ext2fs.static \
       --multiboot-command-line=${kernel-command-line} \
       --host-priv-port=${host-port} \
       --device-master-port=${device-port} \
       --exec-server-task=${exec-task} -T typed ${root} \
         $(task-create) $(task-resume)
module /lib/ld.so.1 /hurd/exec $(exec-task=task-create)
```

First command indicates `/` filesystem; second command path to the kernel on that root fs with single-user flag and device name for `/` fs after boot.; Last two commands loads the *root filesystem* server and the *exec* server. This is done using Grub's boot module capability. The `${var}` are filled in by GNU Mach. For more informations about server's options try: *servername* `--help`.

Note that we need to choose single-user mode, because Hurd installation is incomplete and system need to be set up before boot to multi-user mode.

Once you are presented with a shell prompt after boot, and any time that the Hurd is in single-user mode, it is necessary to set the terminal type:

```
# export TERM=mach
```

Be warned that *Ctrl+C* and family will not work in single-user mode.

We can now run the `native-install` script. This will configure the packages and set up several important translators and device files:

```
# ./native-install
```

Before the script terminates, it need to choose timezone, so we can choose `8` for *Europe* and type e.g. `Prague`. Then the script will indicate that it needs to be run a second time. Follow its instructions and reboot using the `reboot` command. Again, go into single-user mode and put into the command line:

```
# export TERM=mach
# ./native-install
```

In *debconf* configuration you can choose default values *Dialog* and then *Medium* priority.

Maybe installer ask you for another questions, but everything is nicely described.

Installation of the base system and configuration of basic packages is now complete.

7.2.5 System Configuration

Other File Systems

Next, edit `/etc/fstab` to add any additional filesystems as well as swap space. So we add at least our swap space partition that we already created. After that, `/etc/fstab` will look like this:

```
# <file system> <mount point>   <type>  <options>  <dump>  <pass>
/dev/hd0s1      /               ext2    rw         1       1
/dev/hd0s2      none            swap    none       0       0
```

Note that `nano` is the only editor installed by the the base distribution. You can use simply `cat` as well.

Remember to create any devices using the `MAKEDEV` command:

```
# cd /dev
# ./MAKEDEV hd0 hd0s1 hd0s2 hd2
```

`hd0` is whole hard drive (primary master); `hd0s1` is our root filesystem; `hd0s2` is our swap partition; and `hd2` is CD-ROM drive (you can replace it with your CD-ROM device name).

To mount an NFS filesystem, `/hurd/nfs` translator is used. When run as non-root, the translator will connect to the server using a port above 1023. By default, GNU/Linux will reject this. To tell GNU/Linux to accept connections originating from a non-reserved port, add the `insecure` option to the export line. Here is an example `/etc/exports` file assuming the client's IP address is '192.168.1.2':

```
/home   192.168.1.2(rw,insecure)
```

To mount this from a Hurd box and assuming that NFS server's IP address is '192.168.1.1':

```
# settrans -cgap /mount/point /hurd/nfs 192.168.1.1:/home
```

Rebooting

Finally, we can reboot into multi-user mode. It means that we choose in the Grub boot menu third item:

```
GNU Hurd (IDE 1st partition - hd0s1 multi-user)
```

Note that multi-user entry is the same as singe-user that has just one more `-s` option when loading the kernel.

When the system starts up in multi-user mode, you can login as *root* with:

```
login> l root
```

The Network

To configure the network, the *pfinet* translator must be configured. This is done using the `settrans` command to attach a translator to a given filesystem node. When programs access the node by, for example sending an RPC, the operating system will transparently start the server to handle the request. For example, we can set up network like this:

```
# settrans -fgap /servers/socket/2 /hurd/pfinet -i eth0 \
  -a 192.168.1.2 -m 255.255.255.0 -g 192.168.1.1
```

Here, `settrans` is passed several options. The first two, `fg`, force any existing translator to go away. The next two, `ap`, make both active and passive translators. By making the translator active, we will immediately see any error messages on *stderr*. The latter, saves the translator and arguments in the node so it can be transparently restarted later (i.e. making the setting persistent across reboots). The options are followed by the node to which the translator is to be attached[3], then the program (i.e. translator) to run and any arguments to give it. The `-i` option is the interface[4] *pfinet* will listen on, `-a` is the IP address, `-m` is the network mask and `-g` is the gateway.

Be sure to add name servers to the `/etc/resolv.conf` file, in our case:

```
nameserver 192.168.1.1
```

To test the configuration we can ping the gateway:

```
# ping -c3 192.168.1.1
```

Help on `settrans` can be obtained by passing it the `--help` option. Help on a specific translator can be gotten by invoking it from the command line with the same argument, e.g.:

```
# /hurd/pfinet --help
```

7.2.6 Final Words

The Hurd Console

Besides the Mach console you encountered during installation, the GNU/Hurd features a powerful user-space console providing virtual terminals. Currently, you have to start the Hurd console manually with the following command:

```
# console -d vga -d pc_kbd -d pc_mouse -d generic_speaker -c /dev/vcs
```

Inside the Hurd console, you can switch between virtual terminals via *Alt+F1*, *Alt+F2* and so on. *Ctrl+Alt+Backspace* detachs the Hurd console and brings you back to the Mach console, from where you can reattach again with the above command.

The Grub Installation

It is a good idea to install the Grub boot loader to MBR on the hard drive. We need for it the Grub CD, that we have in CD-ROM drive (in my case it is device `/dev/hd2`):

```
# mkdir /cdrom
# settrans -a /cdrom /hurd/iso9660fs /dev/hd2
# cp -R /cdrom/boot/grub /boot
# settrans -g /cdrom
```

[3] `/servers/socket/1` is used for loopback device. Setting for network card is always placed in `/servers/socket/2`, because that is where *glibc* will look for it.

[4] If you would like configure multiple network interfaces, for example eth0 and eth1, you can use `settrans -fgap /servers/socket/2 /hurd/pfinet -i eth0 -a ... -i eth1 -a ...`

The first one command creates direcory for mount CD-ROM; the second one sets the active translator (mounts CD-ROM filesystem); third one copies Grub files to the hard drive; and the last one asks the active translator to go away (unmounts CD-ROM).

Now you can edit `/boot/grub/menu.lst` and change `timeout`, `default` entry to boot, etc.

The last step that we need is write the Grub to MBR. The easiest way is reboot the system, again boot from the Grub CD and choose the fifth item:

```
Install GRUB into hd0 MBR
```

And that is all. Now you can boot from the hard drive.

Adding Devices

By default, only a few devices are created in the `/dev` directory. Use the `MAKEDEV` script to create any needed device nodes.

Installing More Packages

There are several ways to add packages. Downloading and using `dpkg -i` works but is very inconvenient. The easiest method is to use `dselect`, but first of all it is necessary configure `apt`. We use internet for upgrading and installing new packages as I mentioned in 7.2.1. So we need to edit `/etc/apt/sources.list`. The file should look like this:

```
deb http://ftp.cz.debian.org/debian unstable main contrib
deb http://ftp.gnuab.org/debian unreleased main
```

To use a local Debian mirror, visit http://www.debian.org/mirror/list. http://ftp.gnuab.org and its mirrors contain packages that have hacks or patches that have not yet been integrated upstream or in Debian.

Now we can run:

```
# dselect
```

and choose the first item *Access*, then *apt*. Source list we have already configured, so we can skip this step.

Update in the main menu will update list of available packages. Note that during this process you can see some GPG errors. Don't worry about it, it is fine.

When you choose *Select*, the `dselect` automatically marks common packages to install (e.g. *cron*, *manpages*, *gcc*, *wget*, etc.). You can select another packages that you need (*mc*, *bzip2*, ...).

Now select *Install*. You can see that a lot of common packages will be upgraded or newly installed. Before downloading files some authentication warning could appear. Don't worry and continue without verification.

That is all and henceforward the Debian GNU/Hurd should be quite usable.

Useful Tips

If you want to:

- change hostname, edit `/etc/hostname`.

- use Linux-like `ifconfig` for network configuration, install *inetutils-tools* package and use `inetutils-ifconfig` command.
- run the Hurd console automatically on bootup, follow instructions in the file `/etc/default/hurd-console`.
- use `df` command, you have to specify filesystem, e.g. `df /`.
- find out memory info, try `vmstat`.
- read kernel log, use `cat /dev/klog > kernel.log` and read the file. It is because log is readable only once.
- change the options or show the current settings of an active (filesystem) translator, use `fsysopts`.

More tips and FAQs you can find at http://www.gnu.org/software/hurd/faq.en.html.

7.3 Cross Installation

7.3.1 Introduction

This section explains the Debian GNU/Hurd installation on x86 computer with installed GNU/Linux system. We expect configured Debian GNU/Linux 3.1r1 "Sarge" with the Grub boot loader and 2 GB of free space on the hard drive (like in the native installation, it depends whatever you want to install).

We need also some supported network card. Like in the first case, check your hardware at http://www.nongnu.org/thug/gnumach_hardware.html. Don't forget that the GNU Hurd currently runs only on IA32 machines!

7.3.2 Install Preparation

First of all, we need to install `crosshurd` package:

```
# apt-get install crosshurd
```

Note that `crosshurd` can now be used to install not only GNU/Hurd, but also GNU/Linux, GNU/kFreeBSD and GNU/kNetBSD, but it is outside this guide.

Next it is necessary to create a partition. You can use e.g. `cfdisk` for that. Partition table should look like this:

```
 Name    Flags   Part Type   FS Type     Size (MB)
-------------------------------------------------
 hda1    Boot    Primary     Linux ext3  1800.00
 hda2            Primary     Linux swap   128.00
 hda3            Primary     Linux       1800.00
```

On `hda1` our GNU/Linux is installed; on `hda2` there is swap, that we can share with both systems; and `hda3` will be used for GNU/Hurd.

Now we can put *ext2* file system on created partition:

```
# mke2fs -b 4096 -o hurd /dev/hda3
```

The `-o hurd` is necessary so that the system know that it is safe to setup translators.

Review `/etc/crosshurd/sources.list/gnu`. Package sources should be the same as for native installation (see content of the `/etc/apt/sources.list` file in 7.2.6 *Installing More Packages*).

Because the list of required packages for the GNU/Hurd in `crosshurd` package is quite outdated, it is necessary to edit `/etc/crosshurd/packages/common` file and replace these names of packages:

libdb2	→	libdb3
libreadline4	→	readline-common
slang1	→	libslang1

Now we need to mount formatted partition:

```
# mkdir /hurd
# mount -t ext2 /dev/hda3 /hurd
```

7.3.3 Base System Installation

Finally, we can start GNU/Hurd installation:

```
# cd /hurd
# crosshurd
```

Choose the target directory to be `/hurd`, the target system to be *gnu*, leave the target CPU at *i386* and say *yes* for a `/usr -> .` symlink.

7.3.4 Grub Boot Loader Setup

It is necessary to add GNU/Hurd to Grub boot menu after installation. We can use prepared record from the `/usr/share/doc/grub/examples/menu.lst` file[5] and copy it to the `/boot/grub/menu.lst` file. The record should be the same as in 7.2.4 (only with another root device name, in our case `hd0s3`). Note that the record is prepared for mutli-user mode, so it is recommended leave it as is. When you need to boot to single-user mode, you can manually add `-s` option to the kernel in the boot menu.

It is time to reboot the system. In the boot menu we choose GNU/Hurd, that we added, and boot to single-user mode (using `-s` as mentioned above).

Next procedure is the same as for native installation, only with a few differences. We can go to 7.2.4 *Base System Installation — Stage Two* in *Native Installation* section. Before doing so, please read next lines!

Pay attention that in *Native Installation* GNU/Hurd is on `hd0s1`. But now we have GNU/Linux on `hd0s1` and GNU/Hurd is on `hd0s3`. So in the next installation procedure you need substitute `hd0s1` by `hd0s3`.

Because Grub boot loader is already installed and configured on the hard drive, you don't need the Grub CD (that is required in *Native Installation* section) and you can certainly skip 7.2.6 *The Grub Installation*.

[5] If you don't known, it is item with *GNU (also known as GNU/Hurd)* title.

Chapter 8

Current Status of the GNU/Hurd

8.1 Hurd μ-kernel Usage History

The first μ-kernel, that GNU Hurd has used, was GNU Mach, which is based on the Mach 3 μ-kernel. Developers has choosen it this way, because no other μ-kernel had been more suitable for the Hurd. But GNU Mach has had some lacks and mainly performance problems, so core developers have created new experimental branch of the Hurd based on L4Ka::Pistachio μ-kernel. Although L4Ka::Pistachio has been much faster then GNU Mach, it still hasn't got new features to offer.

That is why Hurd on L4Ka::Pistachio has been stalled and core developers think about new μ-kernel again. It will be probably Coyotos[1] which is a successor to the EROS[2] system, or L4.Sec[3], which is another implementation of the L4 μ-kernel. But both μ-kernels are still in the design stage and can't run, so it is quite difficult to make a decision which one is better for the Hurd now. New experimental branch with new μ-kernel should be called ngHurd (or Hurd-NG).

Developers Marcus Brinkmann and Neal H. Walfield have started work on a proposition paper and a technical specification of the ngHurd[4]. These documents should make a decision of a new μ-kernel easier.

One of the design goals of GNU Hurd was to be platform (architecture, μ-kernel) independent, but currently there are experimental ports only, as mentioned above. Porting Hurd to another μ-kernel for the first time will help to be able to do that in the future again.

Development of the Hurd on L4Ka::Pistachio will probably continue only in the case when Coyotos and L4.Sec fail. Probably the Hurd on GNU Mach 1.x will continue its development until more stable kernel is developed. The GNU/Hurd's μ-kernel usage history with Hurd's releases overview is shown in figure 8.1.

About OSKit-Mach

OSKit-Mach began as a branch of the GNUMach 1.2 kernel, but since the release of GNU Mach 1.3, OSKit-Mach has been merged as the new GNUMach 2.x mainline.

[1] Coyotos homepage: http://www.coyotos.org.

[2] EROS homepage: http://www.eros-os.org (research effort has ended).

[3] L4.Sec homepage: http://os.inf.tu-dresden.de/L4/L4.Sec.

[4] A draft specification of the ngHurd iterfaces can be found at http://hurd.gnufans.org/bin/view/Hurd/NextHurd.

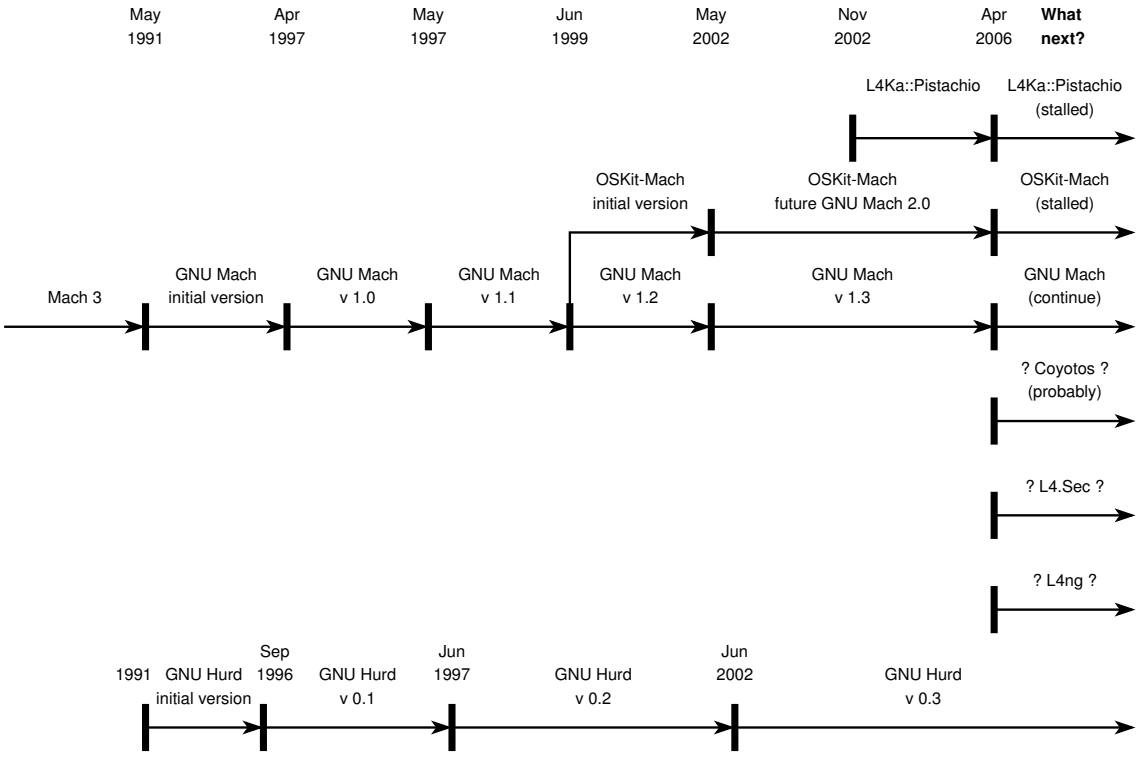


Figure 8.1: Hurd's μ-kernel usage history and Hurd's releases overview

GNU Mach 2.0 makes use of the drivers provided by the OSKit from the Flux Research Group. The OSKit provided a neat driver base where both FreeBSD and Linux (2.2.12) drivers are made available to Mach and thus the Hurd. However, OSKit isn't maintained anymore.

The OSKit-Mach version of GNU Mach is more or less defunct today (2006). Nobody is working on it. Few people ever got it running, and by now there are also problems building it with recent toolchains.

Homepage: http://hurd.gnufans.org/bin/view/Mach/OskitMach

About Coyotos

Coyotos is a secure, microkernel-based operating system that builds on ideas and experiences of EROS (Extremely Reliable Operating System) project. Much of the code developed for EROS will migrate directly to Coyotos.

The Coyotos project has several goals:

- Correct some of the shortcomings of the earlier EROS design.
- Demonstrate that an atomic kernel design scales up as well as down. They are planning to bring up versions of Coyotos on large-scale multiprocessors.
- Provide an efficient Linux compatibility environment for use as a transitional runtime system, so that we can explore adapting applications to a more secure API foundation.

- Construct the kernel and key utilities in a new system programming language (BitC) with a well-defined, mechanically-specified semantics. This will allow us to formally verify security and correctness properties of the system and its key utilities.
- Develop the proving technology necessary to do useful verification about a project of this sort.

System is being developed on AMD-64 and Pentium platforms. A port is also underway to recent Coldfire processors. Support for PowerPC and ARM processors is expected as well.

Coyotos is still in the design stage and no release is available today.

Homepage: http://www.coyotos.org

About L4.Sec

L4.Sec is a specification of next generation μ-kernel interface targeted to co-host user-level servers implementing security-critical functionality next to untrusted and potentially malicious software.

L4.Sec is still in the design stage and no release is available today.

Homepage: http://os.inf.tu-dresden.de/L4/L4.Sec

8.2 Known Flaws and Limitations

Note that the following list of known limitations is for the GNU/Hurd with the GNU Mach μ-kernel (that is used in the Debian GNU/Hurd distribution).

- 1.5+ GB ext2 partition size limit.
  - The problem is fixed in the Debian GNU/Hurd distribution.
- Missing device drivers.
  - Current GNU Mach drivers are from Linux 2.0.
  - OSKit-Mach currently has Linux 2.2 drivers.
- Swap is highly recommended, like on UNIX systems.
- Minimum of 8 MB RAM has been reported to work, but 32 MB is more realistic. Don't forget to use plenty of swap space.
- 768 MB RAM maximum approximately. If you have got too much use Grubs's `uppermem` command to limit the amount seen by Mach.
- Random devices, `/dev/random` and `/dev/urandom`, are not in the main distribution yet. These are needed by, for instance, OpenSSH.
- There are some system API limits.
- No interface for userspace non-critical drivers.
- Sound support missing.

  - OSKit-Mach can provide, but no work yet.

- PPP is currently supported, but is not stable.
- USB support is missing.
- No scanners are currently supported.
- No ACPI support.
- No WLAN support.
- XFree86 is stabilizing.
- X.Org is available in the Debian GNU/Hurd distribution.
- *fatfs* translator is available, and it doesn't have partition limits that get in the way.
- Ext3 implementation is very unstable.
- Portability between alternate microkernels, e.g. L4Ka::Pistachio
- Stability issues.

8.3 Linux-related L4 Based Projects

L^{4}Linux

L^{4}Linux is a port of the Linux kernel to the L4 μ-kernel API. It is a (para-)virtualized Linux running on top of a hypervisor, completely without privileges if wanted.

L^{4}Linux runs in user-mode on top of the μ-kernel, side-by-side with other μ-kernel applications such as real-time components. It is binary-compatible with the normal Linux/x86 kernel and can be used with any PC-based Linux distribution.

Several releases of L^{4}Linux are available, based on different Linux versions (currently 2.6.16, 2.4.30, 2.2.26 and 2.0.21). Only 2.6.x branch is currently mantained.

Homepage: http://os.inf.tu-dresden.de/L4/LinuxOnL4

Fiasco-UX μ-kernel

Fiasco-UX is a port of the Fiasco microkernel to the Linux system-call interface. Due to its special design, it runs without kernel-level privileges, despite the fact that it is a fully functional L4 microkernel. Because it uses GNU/Linux as host system, it can run on any x86-based system as a normal user-mode application.

Supported Linux kernels are from 2.2.x up to 2.6.x (earlier kernels untested).

Homepage: http://os.inf.tu-dresden.de/fiasco/ux

Chapter 9

Epilogue

This project describes the GNU Hurd operating system. The Hurd project exists over fifteen years, but its development is very very slow and stable version hasn't been released yet. That is because main goals of the Hurd are especially stability, security and freedom, but it is not so easy to implement all these features together. The Hurd is really different from other UNIX and UNIX-like systems. You can find a lot of stable and secure UNIXes, but if you take the last mentioned goal, freedom, it is unique just for the Hurd. The freedom here is understood as freedom of user, so he is limited as less as possible, so he can really experiment under OS as he wants, for example to write his own filesystem server or device driver and he does not need administrator permitions and even do not need to reboot his computer.

On the other side, the Hurd's concept of servers causes the whole system to be a bit slower. The interprocess communication is in a trouble here and optimising it can greatly help. Therefore there is an effort to replace original GNU Mach μ-kernel. L4Ka::Pistachio μ-kernel has solved the interprocess communication problem, but core developers furthermore need new features that are required for successful implementation of the Hurd according to system vision, so there is a discussion about another μ-kernel that will be more suitable for the Hurd. There are thoughts about Coyotos, the successor of the EROS or L4.Sec (both not complete yet). They are also indications of completely new μ-kernel that should be created especially for the Hurd.

It is a pity that only a few developers currently work on the Hurd and furthermore they have not got too much time, so development is practically suspended nowadays. For sure the GNU Hurd is a good idea and I hope that this great system will be released as a stable version one day.

Bibliography

[1] Introduction to the translator concept. *DEBIAN.ORG — The Universal Operating System*, [online], [cit. 2006-04-23].
URL: http://www.debian.org/ports/hurd/hurd-doc-translator

[2] Preliminary GNU/Hurd User Interface Description. *DEBIAN.ORG — The Universal Operating System*, [online], [cit. 2006-04-23].
URL: http://www.debian.org/ports/hurd/hurd-doc-server

[3] Debian GNU/Hurd Installation. *DEBIAN.ORG — The Universal Operating System*, [online], [cit. 2006-04-23].
URL: http://www.debian.org/ports/hurd/hurd-install

[4] Hurd Installation Guide. *University of Warwick Hurd User Group*, [online], Last update 2006-03-01, [cit. 2006-04-23].
URL: http://uwhug.org.uk/index.pl?Hurd_Installation_Guide

[5] WALFIELD, N. H.: The Hurd Installation Guide. [online], Last update 2003-01-21, [cit. 2006-04-23].
URL: http://web.walfield.org/pub/people/neal/papers/hurd-installation-guide/english/hurd-install-guide.html

[6] BRINKMANN, M.: How to Install Hurd. In *OpenWeekend 2004 — Video archive*, Prague, Czech Republic, 2004, [online], [cit. 2006-04-23].
URL: http://avc.sh.cvut.cz/archiv/index.php?id=1041&rid=39&offset=0&select=OpenWeekend

[7] Known Limitations and Fixes of Hurd. *The Hurd Wiki*, [online], [cit. 2006-04-23].
URL: http://hurd.gnufans.org/bin/view/Hurd/KnownHurdLimits

[8] GOLUB, D. B.; DEAN, R. W.; FORIN, A.; RASHID, R. F.: Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, June 1990, [online], [cit. 2006-04-23].
URL: http://citeseer.ist.psu.edu/golub90unix.html

[9] KUZ, I.: *L4 User Manual — API Version X.2*. ©2004 National ICT Australia, [online], Version 1.9 (September 23, 2004), [cit. 2006-04-23].
URL: http://ertos.nicta.com.au/publications/papers/L4UM_x.2.pdf

[10] The L4 Microkernel Family. *Operating Systems Group TU Dresden*, [online], [cit. 2006-04-28].
URL: http://os.inf.tu-dresden.de/L4

[11] MIGNOT, G. L.: The GNU Hurd. In *Libre Software Meeting July 5–9*, Dijon, France, 2005, [online], [cit. 2006-04-23].
URL: http://kilobug.free.fr/hurd/pres-en/abstract/abstract.pdf
URL: http://kilobug.free.fr/hurd/pres-en/slides/slides.html

[12] BRINKMANN, M.: The GNU Hurd — Lessons and Perspective. In *OpenWeekend 2004 — Collection of Talks*, Prague, Czech Republic, 2004, p. 37–40, [online], [rev. 2004-10-03], [cit. 2006-04-23].
URL: http://www.openweekend.cz/download/sbornik_ow2004.pdf