The GNU Hurd

[Extended Abstract]

Gaël Le Mignot

ABSTRACT

This is an abstract of the talk given at LSM 2005, about the GNU Hurd, its goals, its design, its features and its relationship with the GNU project. This is a general presentation of the GNU Hurd, mostly on technical aspects.

1. INTRODUCTION

1.1 What is an operating system ?

1.1.1 Historical point of view

The first computers were so expensive that only one existed for a whole university or research department; and so slow that it was unthinkable to interact with it in real time. This was the era of batch processing, when programmers created self-running programs called "batch". Those programs usually took several hours to run, printing huge listing as output. Programs were run one after the other.

Then, computers became more efficient. They were still too expensive to allow each user to have his own computer, but there were powerful enough to allow several users to use the same computer at once. The concept of "time-sharing" appeared then. The "time-sharing" concept consists in having a specific program (the scheduler) which quickly gives the CPU to one program, and then to another, and so on, to simulate simultaneous execution. Those computers had to allow people to run several programs from different users at once, without allowing one user to crash the computer or read confidential data of another. The need of a "supervisor" program, controlling time-slice attribution and enforcing security rules was natural.

1.1.2 What does an operating system do ?

An operating system is the "supervisor" program (or set of programs) of a computer. An operating system runs other programs, providing services to them:

A hardware abstraction layer. There is a large choice of hardware in the computer world. Even if we just speak of, say, storage devices, there are IDE disks, SCSI disks, floppy disks, tapes, USB devices, FireWire devices, parallel port devices, ... should every program know how to put into motion the floppy disk engine ? Providing an hardware abstraction layer is one of the major goals of operating systems.

Resource sharing. Resources on a computer are limited. Be it main memory, CPU power, hard disk storage, screen area, networking connection, or sound output, those resources must be controlled by a supervisor program, allowing each of the programs to access to a part of it as needed.

Security infrastructure. All modern operating systems can enforce several kinds of security. Security can be used to prevent a user from accessing confidential or private data of another user, to prevent accidental removal of critical files, or to limit the damages that a deficient program can do.

1.2 The GNU Project

1.2.1 What is Free Software ?

Free Software is software whose license gives users the four essential freedoms. Those freedoms are:

Freedom 0 Freedom to use the software without any constraints, for whatever purpose the user intent to.

Freedom 1 Freedom to understand the software, and to modify it to make it fit your needs.

Freedom 2 Freedom to distribute the software, since sharing and giving is not a crime, but a social act beneficial for the whole society.

Freedom 3 Freedom to distribute modified versions, giving you the ability to give back source code to the community.

We believe that those freedoms are essential, and that denying someone those freedoms is immoral. Software is knowledge, and knowledge belongs to the whole mankind and should be Free.

Please remember that Free Software doesn't deal with price or money, but with freedom from an ethical point of view.

Libre Software Meeting July 5-9, 2005, Dijon, France.

1.2.2 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.

2. MONOLITHIC KERNELS AND MICRO-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.

2.2 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's 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 chose 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.

2.3 Micro-kernel based systems

2.3.1 Principles

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

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

Many critical parts are now running in user space :

- The complete scheduler
- Memory handling
- File systems
- Network stacks

2.3.2 Monoserver systems

A single user-space program (server) handles everything that belonged to the kernel. The two most common examples are MachOS and L4Linux (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's impossible to add code to it without being root, changing most of the code requires a reboot, ...

2.3.3 Multiserver 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 the 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 the 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.

3. TECHNICAL INTERLUDE: RPCS

3.1 Inter-Process Communication

An IPC is a way for a user-space program to communicate with another one. In Unix, the most common IPC forms are a pipe and a signal.

3.2 Remote Procedure Call

A remote procedure call, or RPC, is a specific set of IPCs, where a client asks a server to perform a task, and then the server acts accordingly, providing an answer in most cases. An HTTP request can be seen as an RPC between a web browser (client) and an httpd (server).

A typical RPC is composed of two IPCs: one from the client to the server, with the type of the request (name of the procedure to call for example) and the parameters of the request; and then another one from the server to the client, indicating the result of request, or at least that the request was perfomed with success or with an error.

To make servers and clients easier to write, most of the programs using a lot of RPCs (CORBA distributed systems, or multi-server operating systems) use stubs. A "client stub" is called natively (like a normal C function if the program is in C), and then performs the marshalling (encoding of the parameters according to the protocol used, being IIOP for CORBA or Mach IPC for the GNU Hurd), the actual request, waiting for the result, and then demarshalling it, retuning it again as a native value. A server stub performs similar functions for the server.

Stubs are usually generated by a stub-code generator, like CORBA's "idl compilers", Mach's "MiG" or L4's "idl4". Those stubs are generated from interface files, which look a bit like C .h files: they contain the prototypes of all RPCs, with the datatype of the argument(s) and return value(s). Those interface files are used both by the stub code generator, and as an API reference for the RPCs.

4. THE GNU HURD

4.1 Definition

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

4.1.1 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 micro-kernel, the Hurd, the dynamic linker (GNU ld) the C library (GNU libc), ...

4.2 The Hurd's goals

The Hurd is the core of the GNU project. Every program running on the GNU system will rely upon the Hurd to perform most of all 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 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).

All this is possible, because we have the experience of Unices and many other operating systems.

4.3 The Hurd history

1983 Richard Stallman starts the GNU project
1988 Mach 3 is chosen as micro-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 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

5. THE MACH MICRO-KERNEL

5.1 History

Mach was one of the first micro-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 used external pagers
- It was the first system to clearly define the notions of "task", "thread", ...

Mach was the first micro-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 microkernel and providing BSD compatibility).

5.2 What does Mach do ?

Mach does quite a lot of things for a micro-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

5.3 GNU Mach

GNU Mach was derived from OSF Mach. The last stable version is 1.3, and should be 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).

5.4 The port concept

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.

6. TRANSLATORS

6.1 The translator concept

6.1.1 The naming service problem

After reading the previous parts, a question should arise in your head: how to get a port (a send right) to a server ? Say, how to get a send right to the TCP/IP stack ? The commonly used method is a naming service: a special server, on which you can get a port right in exchange from a name (like `port = ns_get_port (ns, "pfinet");` to get a port to the TCP/IP stack). The port to the naming service can be given a creation time. But this has several problems:

- The naming service must have its own permission mechanism,
- Every server has to register to the naming service,
- It is not very flexible, and can often cause name conflicts.

6.1.2 The solution used in GNU/Hurd

The idea of the Hurd consists in using the whole VFS (virtual filesystem) as the naming service. Every node in the filesystem is in fact an access point to an underlying server, called a translator.

The function "`file_name_lookup`" of the C library does a recursive lookup of the specified name, and then gives a send right to the translator listening (if allowed by the node permissions). For example, if you have a `/home` partition and you do a request for "`/home/kilobug/plop`", the `file_name_lookup` will ask for `/home/kilobug/plop` to the root translator (known by every application), which will anwser with "well, I don't know this, but you can ask for kilobug/plop to that server", and join a send right to the server handling the filesystem of `/home`. Then, `file_name_lookup` will ask for `kilobug/plop` to this send right, and will get a send right corresponding to this file. RPCs like `io_read` or `io_write` can then be done.

If you ask for `/servers/socket/pfinet` you'll get a port right to the TCP/IP translator, and you'll be able to do RPC to open sockets, or send a "ping" request.

The VFS also provides the permission mechanism, through standard Unix permissions.

6.2 Properties of a translator

A translator is a normal program, running in its own address space, with the rights and the identity of the user who started it. It is not a privileged process (even if many of them, like the system-wide TCP/IP stack, were started by the system administrator during the boot process, any user can run a translator).

A translator is usually a highly multi-threaded program, which can answer to different, even simultaneous, RPCs. The RPCs can be related to file handling (like `io_*` or `dir_*`), but can be completely unrelated if needed, like the `proc_*` RPC family of the proc server (the server handling Unix-like processes, processes groups, signals, ...)

6.3 Translator examples

6.3.1 ftpfs

The most common translator example is the ftpfs translator; you can see a case of use in the figure 6.3.1.

6.3.2 crash

Another example, showing what can be done by using the filesystem as the naming service, is the "crash" translator. The "crash" translator is invoked when a program performs a fatal error (like a segmentation fault). On the Hurd, there are three available crash servers: one which just freezes the process in memory, one which kills the process, and a last one which kills the the process and drop a core file. The "`/server/crash`" file is in fact a symlink to the selected crash server. Switching to another crash server can be done just by changing the link target to point to the node where the crash server you want to use lives. symlink - which is also a translator - redirects RPCs done to "`/server/crash`" to, for example "`/server/crash-suspend`" which does the expected job.

7. SECURITY INFRASTRCUTURE

7.1 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.

7.2 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.

7.2.1 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.

7.3 Some applications

7.3.1 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.

7.3.2 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.

8. TECHNICAL INTERLUDE: VIRTUAL MEMORY

8.1 The concept of virtual address space

On modern computers, programs run inside a virtual address space: the memory addresses a program uses in its instructions are not real physical addresses, but only virtual addresses. The hardware (MMU, which stands for Memory Management Unit) does a translation between virtual address and physical address before issuing queries on the memory bus.

The goals of virtual address spaces are many:

- It allows kernel to implement memory protection easily: a physical memory zone with no corresponding virtual address is automatically unreachable for the program,
- It allows code and data to be loaded at arbitrary position inside memory, which is crucial for multitasking systems,

```
(mmenal@drizzt, 42) ~ $ id
uid=1004(mmenal) gid=1004(mmenal) groups=1004(mmenal),40(src),
50(staff),100(users),518(friends),642(hurdfr)

(mmenal@drizzt, 43) ~ $ settrans -cgap ftp /hurd/hostmux /hurd/ftpfs /
(mmenal@drizzt, 44) ~ $ cd ftp
(mmenal@drizzt, 45) ~/ftp $ ls
(mmenal@drizzt, 46) ~/ftp $ cd ftp.fr.debian.org
(mmenal@drizzt, 47) ~/ftp/ftp.fr.debian.org $ ls
debian  debian-cd  debian-non-US
(mmenal@drizzt, 48) ~/ftp/ftp.fr.debian.org $ ls debian/
README      README.mirrors.html    README.non-US       dists
indices     ls-lR.gz               pool                tools
README.pgp  README.CD-manufacture  README.mirrors.txt  doc
ls-lR       ls-lR.patch.gz         project

(mmenal@drizzt, 49) ~/ftp/ftp.fr.debian.org $ head -n 2 debian/README
See http://www.debian.org/ for information about Debian GNU/Linux.
Three Debian releases are available on the main site:

(mmenal@drizzt, 50) ~/ftp/ftp.fr.debian.org $ cd ..
(mmenal@drizzt, 51) ~/ftp $ ls
ftp.fr.debian.org

(mmenal@drizzt, 52) ~/ftp $
```

Figure 1: A translator: ftpfs

- It allows easy sharing of memory between application,
- It allows easy use of backing stores (like hard disk) to free rarely used memory in time of memory pressure (swapping)

8.2 Segmentation

In the segmentation scheme, a virtual memory address is a SEG:OFFS pair. Each segment has a base, a size, and a protection mode (read-only or not, ...). The physical address is computed by adding the offset to the base of the segment, and checking if it does not overflow the segment size. This is a very simple operation for the hardware.

Usually, segments can be omitted, using a default one (or a default one for the code, a different default one for the stack, and a third one for data), but programs wanting to use some features like memory sharing have to be aware of segments.

Additionnaly, the granularity of segmentation is usually low, and moving a full segment to back storage can be very slow. Another drawback is that a segment must be contiguous in physical memory, creating fragmentation problems.

8.3 Paging

Paging is amore flexible VM design than segmentation. The virtual addess space is linear and contiguous (going from 0 to $2^{32} - 1$ on ia-32), and divided in small pages (of 4096 bytes on ia-32). Each virtual page is mapped to a physical page, or marked as invalid in a page table. This scheme allows better granularity, and is completely transparent for programs. Each process has is own mapping between virtual memory and physical memory, and memory sharing can be done by mapping pages to the same physical frames. This scheme requires more processing from the MMU, and such conversion between virtual addresses and physical addresses are cached in a special cache called TLB (Translation Look-aside Buffer).

When a page marked as invalid is accessed, a "page fault" exception occurs, and the control is given to the kernel. The operating system can then reload the page if it was transferred to back-end storage, and allow the faulting program to go on.

9. MEMORY HANDLING IN GNU/HURD

9.1 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 LRU (Least Recently Used) algorithm; 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's 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.

9.2 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.

9.2.1 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's why many diskfs translators are limited to 2GB partitions on IA-32 (and that's why fatfs isn't limited to 2GB - the FAT is located at the beginning of the partition, so the only limitation is to have the FAT be less than 2GB - I'm not sure it's even possible to have a FAT filesystem that has a 2GB FAT).

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 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).

10. CURRENT STATE OF GNU/HURD

Currently, GNU/Hurd works. You can boot it, you can run XFree86, you can run Emacs or the Gimp, play a tetrinet game, host a web server, ... The Debian GNU/Hurd now fills 4 CDs wide (of working binaries), and we have a support for POSIX threads thanks to Neal !

But, it is true that many features are missing, and many limitations are still present. The whole system is not stable yet, nor optimized (and therefore is very slow). In addition, Mach brings many limitations, performance costs, and GNU Mach is not stable either. We are still far from a "1.0" release, but it's up to you to help us !

11. THE FUTURE

11.1 The L4 microkernel

L4 is family of microkernels. There are several versions of the L4 specification (covering both API and ABI), each of them having one or more implementation. The Hurd will be based, at first, on the Pistachio version (the reference implementation of the newest L4 specifications, X.2).

11.1.1 L4Ka philosophy

The philosophy behind the whole project can be summarized in a few points:

- Defining basic and orthogonal concepts.
- Providing only the most basic mechanisms inside the kernel.
- Make the smallest possible micro-kernel (nano-kernel). Hazelnut (an implementation of L4 version X.0) is only 12K once booted !
- Always keep performance issues in mind.

To achieve that, one of the main focus of the L4Ka work was to provide very fast IPCs:

- Synchronous IPCs, with no need to buffer or copy data. We can still do asynchronous RPCs on top of synchronous IPCs (the data transfer is synchronous, but not necessary the processing)
- Smaller code: less pollution of cache lines. This was one of the major problem of Mach IPC.
- Optimizing technics like address space multiplexing (enables us to do a context switch without flushing the TLB).

Very few things are still inside the kernel (the whole Pistachio provides 11 system calls):

- IPC primitives
- Scheduling primitives (but no policy)
- Memory handling primitives (but no policy)
- I/O primitives (but no device drivers)

11.2 L4 security

With L4, every delicate operation is performed using RPCs. Therefore, controlling the IPCs an application can do, allows the system (or a specific application like a debugger) to control the application completely, from a security point of view at least.

11.2.1 Clans & Chiefs

This was the security model of Hazelnut. In this model, a clan is composed of all the tasks created by the same one task. The creator is the chief of the clan. In the Clans & Chiefs model, an IPC is only allowed from a task to:

- a member of his own clan (a brother)
- his chief (his father)
- a member of a clan he created (a direct son)

All other IPCs are directed through the shortest path of chiefs. Each chief can drop or modify the message.

11.2.2 IPC redirect

The new security model is called IPC redirect: with each thread is associated a redirector (another thread of the same task, or more often another task) controling incoming and/or outgoing IPCs. Redirectors can be changed at run-time, and can be stacked (setting a redirector to a thread already acting as a redirector).

This new system was designed for two reasons: first Clans & Chiefs was too complex and too slow (the chain could be long, and even if a single IPC is fast, a huge amount of them will take some time); but mostly it was a decision upon the OS policy, and in the L4Ka philosophy such decisions must be kept outside of the kernel.

It is possible to implement Clans & Chiefs on top of IPC redirect with the proper redirectors.

11.2.3 Some possible usages

The IPC redirect mechanism can be very useful, for example to monitor applications (for debugging, for profiling, for security, just to keep logs of what the program did, ...); or to allow sand-boxing. Sand-boxing is running untrusted code inside a sand-box, the sand-box preventing the code to interact directly with the operating system. With sand-boxing, you could even run untrusted binary code directly on the main CPU without any need of virtual machine, and without taking any risk from the security point of view. This can be very useful for interactive web, or even to run the rendering engine of a web browser (and this way protect the system from a security flaw inside the browser).

APPENDIX
A. REFERENCES

GNU `http://www.gnu.org`

The GNU Hurd `http://hurd.gnu.org`

Debian GNU/Hurd `http://www.debian.org/ports/hurd`

HurdFr `http://hurdfr.org` or `#hurdfr` on `irc.freenode.net`

L4 `http://www.l4ka.org`

The GNU manifesto `http://www.gnu.org/gnu/manifesto.html`