

The GNU Hurd

Marcus Brinkmann

Copyright © 2002 Marcus Brinkmann

Permission is granted to make and distribute verbatim copies of this paper provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this paper under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this paper into another language, under the above conditions for modified versions.

1 History

- 1983 Richard Stallman founds the GNU project.
- 1988 Decision is made to use Mach 3.0 as the kernel.
- 1991 Mach 3.0 is released under compatible license.
- 1991 Thomas Bushnell, BSG, founds the Hurd project.
- 1994 The Hurd boots the first time.
- 1997 Version 0.2 of the Hurd is released.
- 1998 Debian hurd-i386 archive is created.
- 2002 Debian GNU/Hurd snapshot fills four CD images.

2 GNU/Linux is like Unix

The GNU/Linux system consists of the GNU userland running on top of the Linux kernel. The GNU programs are written in a portable way. They are based on the POSIX.1 programming interface, which is implemented by the GNU C library. Some areas of the API are implemented by the C library exclusively, or only with little help from the kernel. In some other areas, the C library functions are just small wrappers around the kernel system calls. A system call is a trap into the kernel. The program continues execution in the kernel while doing some privileged operation. After that operation is finished, the program continues to run at the location right after the system call.

Linux contains around 200 system calls, which implement several huge and important parts of the POSIX API almost directly:

- The filesystem interface (open, read, write, close, mount, ...)

- The process handling (getpid, fork, exec, ...)

- Signal handling (sigaction)

- Pipes

- Sockets

- Network configuration

- Scheduler policy

- SysV shared memory and semaphores interface

Linux also contains some other, non-standard interfaces, like clone (used to implement thread libraries) or the modules interface to load and unload kernel modules at run time.

All these features are implemented by one large program, the kernel. This program runs in a privileged mode of the processor, and has full control over the hardware. A bug in the network stack can potentially result in random changes to the filesystem. Any random bug can crash the whole system or cause other serious malfunction. More code in the kernel results also in more bugs.

Still, a lot of code is included in the kernel, and this is often done for maximum efficiency. One prominent example is the webserver in Linux, motivated by a proprietary solution beating the GNU/Linux + Apache combo performance wise. Another example is the network stack itself, which in the early days of Unix was in userland, until some people optimized it by moving it into the Unix kernel.

However, putting a lot of functionality into the kernel has several disadvantages:

- Kernel code often runs with overly broad privileges. Running software in user space has similar advantages over running it in kernel space as running software as a normal user instead as the superuser.

- The kernel environment is static and inflexible. Although Linux provides some flexibility by giving the administrator the option to load and unload some components as modules, this is a far cry from the flexibility the user space offers, where all components can be replaced at run time, and a full tool set is available. For example, a user can set up a different library search path and use his own versions of all the system libraries (for non-suid/sgid programs). But the only thing he can not replace is the actual kernel. Changing the kernel usually involves a reboot.

Any change to the kernel configuration is completely reserved to the superuser. Because all kernel code must be trusted, the superuser can not let users load and unload kernel modules themselves.

A decision has to be made if a feature belongs into the kernel or into the userland. Depending on the usage pattern, different people will argue for different features. As patching and recompiling the kernel is difficult and destabilizing, most people will not profit from features that are not distributed with the standard version of the kernel.

3 GNU is not Unix

The GNU/Hurd system replaces the monolithic kernel by a more flexible design that solves all these problems and brings further advantages.

At the core of the GNU system is a microkernel. A microkernel runs in the same privileged mode of the process as the Unix like kernels, but it does not implement most of its features. The idea is that the microkernel provides only primitive operations that are sufficient to implement the operating system on top of it. The Hurd currently runs on GNU Mach, which provides the following concepts:

- Virtual memory management, with an external pager interface.

- Scheduling, tasks and threads abstractions (but tasks are much less than a POSIX process).

- Inter-process communication using Mach ports

- Device drivers

Mach does not contain any of the following:

- A network stack

- Signal handling

- Pipes

- Sockets

- POSIX processes, process groups and session groups

- terminal support (line discipline etc)

Mach is actually rather big for a microkernel, and quite dated. It has only a few system calls, but it uses its own IPC mechanism, and the full kernel API contains over 150 functions. In the future, the Hurd might be ported to other, more modern microkernels. A good candidate seems to be L4, a tiny microkernel with a size of about 50kb and only 7 system calls. Those 7 system calls provide enough functionality to create a whole POSIX compatible operating system on top of it.

The microkernel is providing primitive operations that make it possible to share the hardware resources (processor time, memory, ...) and make different tasks cooperate (IPC). However, it doesn't define the actual policy by which these things should happen. This has to be done by user space programs running on top of the microkernel.

4 Breaking it up

The Hurd servers, which run on top of the Mach microkernel, implement the functionality that is usually found in a Unix-like kernel. But instead having one large monolithic server, it was decided to split up the system into many individual server processes, each server implementing a separate and distinct function.

The process server maps Mach tasks to Unix processes. This server also stores other metadata about processes (for example process group and session group relationships).

The file system servers implement the file and directory operations. There is one file system server process for each mounted filesystem.

The network stack is implemented by the pfinet server.

Pipes and Unix domain sockets are implemented by the pflocal server.

Authentication is handled by the auth server, which provides users with authentication tokens that are mapped to the Unix user and group IDs.

All the Hurd servers communicate with each other and cooperate in implementing the features of the system. For example, if a user tries to open a file, the file system server has to rely on information provided by the authentication server to check if the user can be allowed access to the file. Another example: If a process wants to send a signal to another process, it has to look up the message port (to which the signal is then sent) of this process by consulting the proc server.

5 And putting it together again

The interfaces by which the servers communicate are at the core of the Hurd system. Although you can write programs specifically to talk directly with the Hurd servers, the usual way to add programs to the Hurd system is by basing them on the POSIX.1 interfaces. Except for bugs, the Hurd is a POSIX compatible operating system. However, as in GNU/Linux, the actual POSIX layer is provided by the GNU C library. Sometimes, POSIX functions map directly to corresponding Hurd server interfaces. For example:

```
read (fd, ...) <=> io_read (ioport, ...)
write (fd, ...) <=> io_write (ioport, ...)

getpid ()          <=> proc_getpids (procport, ...)
```

However, a lot of the POSIX functionality can actually be implemented by the user itself, rather than punting it into the system code. The C library does this for the user, too. For example, `fork()` is not implemented by any Hurd server, but in the C library directly, which uses Mach and Hurd server primitives to create a mirror image of the current process. The user is free to use his own implementation of `fork()`, if he desires so. For the standard case if POSIX.1 behaviour is desired, the default implementations in the GNU C library provides exactly that.

Although the GNU C library provides a POSIX personality on top of the Hurd, and this is the main personality of the Hurd today, there is no reasons why other libraries shouldn't use the Hurd server's functionality to provide a compatibility layer to some other API. If this is done carefully enough, then programs from both worlds can take advantage of the common Hurd base, for example by sending signals to each other.

By putting the lot of the system's functionality into userland, the Hurd has several advantages:

It is robust. If a non-critical server crashes, or has bugs, it can be terminated and restarted without affecting the rest of the system. Every server process runs in its own encapsulated environment with kernel enforced barriers between the user space processes. The servers run with the privileges they need. For example, the filesystem provides file system services to the users, but there is no need for it to have control over the user's task or authentication handle.

It is flexible. Servers can be added, removed, upgraded or replaced individually, and without affecting the operation of the rest of the system. For example, a second pfinet server can be started to provide a second network stack, running in parallel to the default one. Selected processes could be made to use the new network stack for testing. Server processes can be debugged with `gdb` like all other user space programs.

As the whole system is in user space, and splits up into separate components, it is much easier to add new system components that are not included in the Hurd by default. Actually, the process of compiling and installing a new Hurd server is very much like the process of installing any other user program. Incompatibilities like with Linux modules are not possible, because the Hurd server interfaces (the RPCs) are stable.

6 Accessing the servers

Accessing the various Hurd servers is not done via a nameserver as traditionally has been done in Mach. The simple reason is that POSIX.1 compliant systems already have a name service we can use, and this is the file system. Every mounted filesystem is provided by a different server. Open files and directories are actually represented by ports to the servers providing these files. So it makes sense to just use the filesystem servers as name services to access other Hurd servers.

For example, the Hurd password server hands out authentication handles in exchange for a password. It is used by programs like `login`, so those programs don't need to be run `suid root`. The password server has the canonical place `/servers/password` in the filesystem. If a user wants to communicate with the password server, he looks up the `/servers/password` node, and sends RPCs to the port he gets returned. The filesystem server has intimate knowledge about the translators attached to its nodes, so it can redirect the user to the translator's service on each such file name lookup.

There is no real difference between filesystem servers like `ext2fs` and special purpose servers like `password`. They implement different protocols, but they are all part of an hierarchical tree of Hurd servers, each server connected to the node in the parent filesystem it is attached to. Even a single file node like `/servers/password` is implementing a small filesystem.

The Hurd provides a way to store a translator command into a node of the filesystem. When an access on the node is made, and there is no active translator yet, the passive translator setting will be read out and the filesystem server will make an attempt to start up the filesystem using the command line from the passive translator setting. Because the information is written into the filesystem, it remains there across reboot, so there is no need to set up the translators after every boot. They will be started dynamically on demand and transparently to the user.

7 Inviting the user

One problem in Unix-like systems mentioned earlier is that a lot of useful functionality is reserved to the superuser, because it frequently involves privileged operations. For example, only the superuser can mount filesystems by default, although the mounting of filesystem image files is useful, for example when preparing a CD image. And even if users are allowed to mount filesystems, they can only mount filesystems of the supported types for which modules exist in the kernel. In general, there is a barrier around the system code that restricts the users freedom in using the system.

The Hurd eliminates the system code barrier as far as possible. This is achieved by designing the Hurd servers interfaces in a way that allows for communication between untrusted servers and users, without compromising the security of the system. This makes it safe to allow users to start their own Hurd servers and attach them to the filesystems at nodes that are owned by this user.

A central role is taken by the auth server, which is a central authority giving out authentication handles to users. Only root can request arbitrary authentication handles, other users are restricted to merge and subset operations on existing handles. Beside support routines for managing auth handles, the auth server provides a handshake protocol to establish a trusted channel of communication between an untrusted server and an untrusted client. As the result of the protocol, the server can associate the messages from the client with the user and group ids in the clients authentication handle, and thus implement appropriate access control.

For this to work, the server must trust the auth server, and the client must use the auth server trusted by the server. This means that if you want to communicate with a server, you will have to authenticate yourself with the auth server it trusts. All system servers will use the system auth server as the trusted one by default. Nothing forces the user to register itself with the system wide auth server, but nothing else will allow the user to communicate with the other system servers, so it is usually unavoidable for a user to use the system auth server.

However, it is possible and useful for a user to start his own authentication server in addition to the system server and use that as the trusted authserver for the user's own servers.

For other servers, the system often provides a standard way to use a user provided server rather than the system default (where it doesn't do that yet it probably should). For example, the crash server takes control over crashing tasks. It can suspend them for later debugging, dump a core file, or just kill the task without leaving a trace of the crash. However, if a user implements his own crash server that sends a message to a mobile phone when the tasks crashes, the user just needs to set the environment variable CRASHSERVER to point to his own server when starting the program to take advantage of that feature.

Similarly, the exec server is responsible to parse an executable file and take the appropriate action to start the program, for example by starting the interpreter. A user could provide his own exec server that understands a new object format, and set the EXECSERVER variable before starting it. This feature is disabled currently, though, because of security concerns. We are not sure yet that the current code covers all possible corner cases in a safe manner.

Another way for a user to extend the system is to add new filesystems and attach them to nodes in his home directory. With the ftpfs server, everyone can add a remote ftp repository to their filesystem hierarchy. Other filesystems might provide transparent access to other hierarchically organized data, as LDAP servers, or even tar files. Of course, beside the full blown filesystem servers, small servers only providing one file can also do a useful job. One program I wrote, the run translator, provides a file that on each `open()` spawns the program given on its argument line and provides its output as the content of the translated file. This can very easily be used to randomize `~/.signature` for example, without worrying about mail user agent configuration at all.

If replacing or adding individual servers is not sufficient, a user can boot a second Hurd system from a filesystem image with the program `boot`. This will start another Hurd system that runs in parallel to the first one. Because this Hurd system contains its own auth server, the user will have root permissions in his Hurd system, but despite this appearance there is no actual permission leak, as the system defaults server in the original Hurd system will continue to use the trusted system auth server. And that won't be fooled by any process related to the second Hurd system to have any different permission as the user running the second Hurd system has.

8 Solving some problems in Unix

A common problem in using computer systems is interoperability between individual programs. The output of a program can serve as the input for another, for example by using a pipe. In general, one program will use the services of some programs, and provides services itself to other programs in the system.

In Unix, pipes and sockets are powerful features to connect programs, but they are limited to the communication between user space processes. If you want to affect the communication between a user space process and the system code (eg the kernel), Unix does not provide you with any standard facility to do that. Usually a program is linked to the C library, which directly invokes the relevant Unix kernel system calls, for example to rename a file.

But, there is a real need for users and developers to intercept or redirect communication with the kernel. Here are some examples where it is attempted to do that in Unix, and how it is achieved:

strace: strace is a program to log all communication between the kernel and a program. It shows which system calls are invoked with which arguments, and what they return. The program is available for SunOS, Solaris, Irix and Linux, and uses the `ptrace()` function to make the child stop at each system call and inspects the arguments and return values. The `ptrace()` interface does not feasibly allow to modify what the child and kernel exchange, but is mostly only usable to observe all syscalls.

So although Unix provides sometimes a facility to trace system calls in a child process, the solution is specifically tailored to this and only this application.

fakeroot: fakeroot is a program used to give an unprivileged user a faked root personality. To the user and all (non-suid) programs it invokes it appears as if the user has a user and group ID of 0, and file operations like `chmod()`, `chown` etc always succeed. It is for example useful to create an installation tree that is then packaged into an archive, where files are supposed to have certain ownerships and permissions.

The fakeroot program consists of two parts: A server process, which stores the faked metadata information about files (ownership etc), and a library that is `LD_PRELOAD`'ed and that inserts wrapper function for common functions in the C library that should information about the faked nodes stored in the server process rather than the real information stored in the filesystem.

This is a very clever solution that has a lot of problems. First, it is not transparent to the applications. The fakeroot library needs to be linked into all programs that are expected to see the faked information, so it doesn't work for existing static binaries. The fakeroot library needs to find the server port, so there are some environment variables reserved to this use. Of course, to make programs use the fakeroot library, the `LD_LIBRARY_PATH` and `LD_PRELOAD` libraries must remain intact. Furthermore, the fakeroot library is specific to the C library used. Only wrapped functions will see the faked metadata. A program not using the C library functions, but the kernel system calls directly, will see the real data. A program not using the same C library as fakeroot will not start up at all.

The problem here is that we can not go down to the system code level, but have to intercept filesystem operations before they reach the C library (which usually provides

simple wrappers around kernel system calls for such functions). Any such solution has the same problems above.

Gnome VFS, KDE ioslave and libferris: The Gnome virtual filesystem, KDE ioslaves, and libferris serve the same purpose. They are intended to give applications transparent, filesystem-like access to various filesystem-like resources. This includes the normal filesystem as well as FTP archives, web sites accessed over HTTP, LDAP servers, the help documentation repository of the desktop environment and various other things.

This is achieved by providing a Unix filesystem like interface in a helper library. All programs that want to make use of the extra features need to use this interface rather than the Unix interface. The library provides backends for the various protocols and virtual filesystems supported. Once a program is ported to use the new interface, the various resources provided by the backends are transparently provided to the applications.

The problem is of course, that there are multiple of such abstraction libraries, and that applications have to be ported to use them before they can take advantage of the virtual filesystem layer. This is often unacceptable, and not desirable. Furthermore, the decision which abstraction library to use has to be made by the programmer in advance, and can not be easily changed by the user. Multiple users who want to use different middle layers can not share the same binaries.

So despite the fact that Unix makes it difficult, tedious and sometimes downright impossible to implement these features in their cleanest form, the need for them has forced people to implement them anyway and cope with the limitations. How can the situation be improved?

9 Solving the problems in the Hurd

The Hurd avoids this problem in two ways. First, it moves the system code that implements the filesystem and other functionality provided by the Unix kernel into the user space. But more importantly, it breaks up the monolithic system core into many individual servers, and uses a challenge-response protocol to establish a communication between an untrusted server and an untrusted client. This makes it possible to allow users to plug their own servers into the system.

The GNU C library implements the POSIX environment that applications expect on top of this. It ensures that all server communication happens transparently to the user. Filesystem services provided by the user's server are transparently started and accessed. At startup, a program inherits some basic server ports from its parent. Among these are the root filesystem port, the current directory filesystem port, a port to the authentication server establishing its identity to the system and others.

This is all very abstract, so let's take a look at how we can solve some real world problems this way:

rpctrace: rpctrace is a program to intercept all messages from and to a program. It does this by replacing every send right with a send right to a port it owns, and when it receives a message on this ports, it records the message content and then forwards it to the original port. All messages are intercepted, but the child process has no way to tell that it is not talking to the original recipient directly, so this is completely transparent to the user.

Furthermore, it is completely generic. Because all communication between processes and between a process and the microkernel is performed by RPCs, rpctrace sees all of the communication.

Because RPC does the forwarding of the message itself, it is not limited to observing and forwarding the messages. It could also change the message parameters, or react on a message by taking some more complex action. This happens in the rpctrace process without affecting the state of the child process.

fakeroot: The Hurd provides an implementation of fakeroot that acts at a level lower than the C library. Actually, fakeroot in the Hurd consists of two parts:

A filesystem server called fakeroot, which passes through most filesystem operations directly to the underlying filesystem, and provides faked responses to filesystem operations that involve the faked metadata of a file. If fakeroot runs on top of the root filesystem, and a port to fakeroot is used as the root directory and current directory port of a program, then this program will make all file accesses through the fakeroot server, regardless which version of the C library they use, what their environment variables are, or if they are linked dynamically or statically. The runtime environment of the program does not need to be changed at all (beside giving it a different root and current directory port).

This solves all issues regarding to filesystem operations, but the user still has his real user and group IDs. To give the user a faked root identity, the program fakeauth can be used. fakeauth is invoked like sudo. But instead starting a program with root privileges, it will start the program with faked root privileges. To do this, fakeauth acts itself as an authentication server, and starts the program with the authentication

server port pointing to the fakeauth program. Again, most operations, in particular those operations that establish a server<->client communication, are passed through to the real auth server process. But the operation that returns the user and group IDs on an authentication handle simply returns root IDs. This has the effect that programs like whoami will run with an authentication handle that *looks* like root, but *acts* like the user who started fakeauth (eg, it won't fool other servers the programs communicate with).

This is how the two programs are combined to make a fakeroot program (this is actually the only line of code in /bin/fakeroot, except `-help/-version` option handling).

```
exec /bin/settrans --chroot \  
  /bin/fakeauth /bin/sh -c "cd `pwd`; exec $*" \  
-- / /hurd/fakeroot
```

This starts the fakeroot translator on the root ("/") node, and then runs, with the root and current directory port pointing to the root of this fakeroot translator, the fakeauth program. This fakeauth program starts a shell, `cd`'s to the current directory and runs the program as specified on the command line. This program will then have three faked ports: The root and current directory port will point to the fakeroot server, the authentication port will point to fakeauth.

Gnome VFS, kioslave, and libferris: The Hurd allows any user to implement and use the filesystem they want. This can also be done on a per-process basis. The advantage of implementing the virtual filesystem representation of a data repository by a server in the Hurd layer rather than by a library in an application layer is that Hurd servers operate below the C library. So a new Hurd server intergrates itself seamlessly into the file system, and all applications can make immediate use of the features it provides. The programs don't need to be ported or recompiled, and all programs automatically use the same new filesystem backends. This also allows for sharing caches and other run time data.

Short Contents

1	History	1
2	GNU/Linux is like Unix	2
3	GNU is not Unix	4
4	Breaking it up	5
5	And putting it together again	6
6	Accessing the servers	7
7	Inviting the user	8
8	Solving some problems in Unix	10
9	Solving the problems in the Hurd	12

Table of Contents

1	History	1
2	GNU/Linux is like Unix.....	2
3	GNU is not Unix	4
4	Breaking it up.....	5
5	And putting it together again	6
6	Accessing the servers	7
7	Inviting the user	8
8	Solving some problems in Unix	10
9	Solving the problems in the Hurd.....	12