The Hurd Hacking Guide

Edition 0.2˙1, for GNU Hurd Version 0.2.
Mar 25, 2002

Wolfgang Jährling

"The GNU Hurd is the GNU project's replacement for the Unix kernel. The Hurd is a collection of servers that run on the Mach micro-kernel to implement file systems, network protocols, file access control, and other features that are implemented by the Unix kernel or similar kernels (such as Linux)."

— `http://hurd.gnu.org`

1 About this document

1.1 Conventions

The Version of this document follows the convention <hurd version>_<document release>. This means that 0.2_7 is the seventh release since Hurd 0.2 and a version like 0.4_1 would be the first release for Hurd 0.4 (which, of course, is not yet available :)).

`$(HURD)` means the top directory of the 'Hurd' source tree. `$(GNUMACH)` means the top directory of the 'GNU Mach' source tree. `$(MIG)` means the top directory of the 'MIG' source tree. `$(GLIBC)` means the top directory of the 'GNU Libc' source tree.

Single shell commands start with a `$` for user commands and a `#` for root commands:

```
$ diff -u libtrivfs.old/open.c libtrivfs/open.c
# reboot
```

Prompts of other programs look as they do in the respective application. For example, the GDB prompt is indicated by `(gdb)`:

```
(gdb) break trivfs_S_io_write
```

I will try to obey the GNU Coding Standards in my C examples. http://www.gnu.org/prep/standards_toc.html

1.2 Topic

This document is an introduction to GNU Hurd and Mach programming. The purpose of this guide is to help interested people start hacking the Hurd or extending it (by writing translators). It gives lots of references to the Hurd or GNU Mach source files. It is recommended that you read through some of these sources. Indeed the Hurd sources are very well written and commented and you can learn a lot by reading them.

The Hurd looks very complex and hard to learn — at a first glance. But it isn't, because you don't need to understand everything at once, you may do it slowly and step-by-step and can apply your existing knowledge. There are also libraries that make hacking of certain common kinds of translators easy. I think that the only problem is the absence of nice documentation like the "Linux Module Programming Guide" and such, which makes it possible to get into it step by step. This document tries to fill that gap.

Mach and MIG are not handled in depth here, so if you want specific information on them, I recommend reading the GNU Mach Reference Manual[1] and the documentation about MIG available on the internet.

The Hurd Hacking Guide is not intended to be a complete reference, but it ought to help you getting started. The only real reference at the time of writing is the Hurd source code.

The order of chapters in this document was originally partly based on a mail from Farid Hajji[2], which in turn seems to be based on `$(HURD)/doc/navigating`.

---

[1] http://www.gnu.org/software/hurd/gnumach-doc/mach.html

[2] http://lists.debian.org/debian-hurd-0012/msg00149.html

1.3 Feedback

Don't hesitate to send me improvements, corrections and extensions. You are of course welcome to correct my lousy english — I'm not a native speaker/writer. (Thanks to Alfred M. Szmidt for his corrections, BTW.)

I would be happy to hear about how understandable this text is. Did you "get" everything? Was some part confusing? Send me feedback!

Actually, I wrote this to document everything I learned about the Hurd, so that I later could quickly lookup a detail that I had forgotten. This means that if you send me extensions, I will also profit from them.

2 Requirements

1. You should know at least basic things about the Hurd[1].
2. More specific knowledge is of course welcome[2].
3. It would be good to know what a tranlators is[3].
4. The sources of Hurd and GNU Mach are also useful:

```
mkdir ~/hurd-cvs
cd ~/hurd-cvs/

cvs -d:pserver:anonymous@cvs.savannah.gnu.org:/sources/hurd \
  co hurd
cvs -d:pserver:anonymous@cvs.savannah.gnu.org:/sources/hurd \
  co -r gnumach-1-branch gnumach
```

5. A GNU/Hurd installation certainly will help you, too[4].
6. Diving into the header files of the Hurd's libraries is dangerous, because you can drown very easily, because you won't find the way out of all these data structures. Maybe a list of where to find what might be helpful, e.g. the output of

```
$ cd $(HURD) && egrep '^struct [^;*]+$' */*.h
```

7. If you know the principles of Mach, what MIG is, etc., then this will of course help you a lot, but it should not be necessary. Knowing about the Linux kernel might also help to some degree.
8. Oh, and you should know the C programming language. :-)

[1] http://hurd.gnu.org/
[2] http://www.gnu.org/software/hurd/hurd-paper.html
[3] http://www.debian.org/ports/hurd/hurd-doc-translator
[4] http://www.gnu.org/software/hurd/devel.html#machinery

3 A Short Overview of Hurd and Mach

> "We're way ahead of you here. The Hurd has always been on the cutting edge of not being good for anything." (Roland McGrath[1])

> "In short: just say NO TO DRUGS, and maybe you won't end up like the Hurd people." (Linus Torvalds[2])

Every seasonable piece of software needs a method for communication between components. Nowadays, things like CORBA or Mozilla's XPCOM are used for that. The advantage of the Hurd over other systems is, that it provides such a facility *and* does not require existing applications to be modified to take advantage of its communication framework. How does the Hurd reach this goal?

> "The Hurd, at its most central core, is just the protocols that the cooperating servers use." (Thomas Bushnell[3])

In a Mach environment, communication between programs is mostly done by sending messages through so-called "ports", which are a kind of message queue. For each port, there is one task with receive-permission (i.e. this task receives the messages someone sends to this port). Other tasks might have a send-permission or a send-once-permission (which is used for getting a reply from a server, because ports are one-way channels) for this port, or even no permission at all.

If you read through `$(GNUMACH)/include/mach/port.h`, you may notice that there are more port rights: A send or send-once right is turned into a "dead name" if the receive right is destroyed. So you can't use the dead name right for anything, it's merely a place-holder. Another port right is "port set", which Marcus Brinkmann explains as follows[4]:

> "A port set is a set of ports. It is useful to combine ports into a port set if you just want the next message on any of the ports you have a receive right for. In the Hurd, we use port classes and buckets provided by libports, though.
>
> Well, we are not exactly strict in our wording when talking about ports. [...] You can think of port rights as capabilities associated with ports and port sets, if you want."

How can a process find a specific port? It's really simple: through the file system. For example, the services of an ext2-server are available through the node where the file system he handles was "mounted" (please note that there is no such thing as mounting in the Hurd world, this is what it would be called under Unix; the correct term for the GNU/Hurd operating system would be "setting a translator").

Your favourite email client doesn't support random signatures? Write a random signature translator[5] (which returns a new signature each time you read from it). And the best thing is: now *all* email clients may use this feature! Do you see how the Hurd encourages

---

[1] REFERENCE missing

[2] REFERENCE missing

[3] `http://lists.debian.org/debian-hurd/2002/05/msg00493.html`

[4] see `http://lists.gnu.org/archive/html/help-hurd/2001-07/msg00018.html` for the complete discussion

[5] Or better use the already existing 'run' translator, which was written my Marcus Brinkmann and is much more flexible; using the existing filemux translator would also be possible

code reuse? Do you see how GNU/Hurd does not require programs to be modified to take advantage of most of the nifty features it provides?

If you would like to learn more about the Hurd filesytem, I highly recommend reading the presentation "The Hurd"[6].

We can say that the file system is the name space for services, this also true in the other direction: the name-space for services *is* the file system. This a very important thing to understand. While the file system is the canonical way to get a port, there are other ways as well; for example, you can get a port in a message.

If you are wondering why I compared this kind of communication with CORBA, the following quote from the paper "Towards a New Strategy of OS Design" might help you understand the reason:

> "With translators, the filesystem can act as a rendezvous for interfaces which are not similar to files. Consider a service which implements some version of the X protocol, using Mach messages as an underlying transport. For each X display, a file can be created with the appropriate program as its translator. X clients would open that file. At that point, few file operations would be useful (read and write, for example, would be useless), but new operations (XCreateWindow or XDrawText) might become meaningful. In this case, the filesystem protocol is used only to manipulate characteristics of the node used for the rendezvous. The node need not support I/O operations, though it should reply to any such messages with a message_not_understood return code."

[6] `http://www.gnu.org/software/hurd/hurd-talk.html`

4 Basics of Mach and MIG

4.1 Mach ports

Now we will take a look at some Mach details. Yes, I know that you would like to start writing translators as soon as possible, but you really should know at least the basics of Mach. Mach ports are used extensively throughout the Hurd, so let's start with them.

First, let's make the distinction between ports, port rights and port names clear. Marcus Brinkmann once wrote (on IRC):

> "`mach_port_t` is a port name, the port name denotes an entry in the tasks port name space, which is associated with either a dead name, a send-once right, or a combination of a receive right and a send right with potentially many user references.
>
> Assume you have `mach_port_t 5`, and want to send a message to it. To send a message, you pass the task `mach_task_self()`, the port name, the msgid and the arguments. The task is used to get the ipc name space, the port name is used to find the entry in this name space. The entry tells Mach about the port rights you have for the associated port with this port name.
>
> You have a single port name for all receive/send rights you might have for a port. But you have distinct port names for send once rights, because this is easier for Mach to manage — and for user programs, too."

Mach defines the type `natural_t`, which is the native type of the machine, e.g. 32 bits on a 32-bit processor. `natural_t` is always unsigned, but there is a signed variant called `integer_t`. The definitions for the i386 platform can be found in `$(GNUMACH)/i386/include/mach/i386/vm_types.h`. In this file you can find other interesting types as well, but those are not important for us right know.

Like Unix file descriptors, Mach port names are plain, boring integers. In the file `$(GNUMACH)/include/mach/port.h` you can see the following definitions:

```
typedef natural_t mach_port_t;
typedef mach_port_t *mach_port_array_t;
```

(For most data types in Mach, an additional `*_array_t` type is defined.) The port number identifies a unique port (in the namespace of the task), thus a `mach_port_t` value is often refered to as "port name".

A value of `MACH_PORT_DEAD` (i.e. `~0`) represents a port right that has died, while `MACH_PORT_NULL` (quoting `port.h`) "indicates the absence of any port or port right."

You may check with `MACH_PORT_VALID (port)` if a port is neither of these two values.

For the port rights, we have the macros with the names `MACH_PORT_RIGHT_RECEIVE`, `MACH_PORT_RIGHT_SEND`, `MACH_PORT_RIGHT_SEND_ONCE`, `MACH_PORT_RIGHT_PORT_SET`, `MACH_PORT_RIGHT_DEAD_NAME` and `MACH_PORT_RIGHT_NUMBER` which are of type `mach_port_right_t`:

```
typedef natural_t mach_port_right_t;
```

This type is used whenever we need to act on a particular port right. Often, however, we want to carry around a set of rights, because we may have multiple rights on a single port. For this, we use another type:

```
typedef natural_t mach_port_type_t;
typedef mach_port_type_t *mach_port_type_array_t;
```

A `mach_port_type_t` variable may either carry the value `MACH_PORT_TYPE_NONE`, which of course represents an empty set of rights, or a set of the macros `MACH_PORT_TYPE_SEND`, `MACH_PORT_TYPE_RECEIVE` etc. combined with a bitwise or. There are also several predefined combinations:

Macro	Combination
MACH_PORT_TYPE_SEND_RECEIVE	MACH_PORT_TYPE_SEND, MACH_PORT_TYPE_RECEIVE
MACH_PORT_TYPE_SEND_RIGHTS	MACH_PORT_TYPE_SEND, MACH_PORT_TYPE_SEND_ONCE
MACH_PORT_TYPE_PORT_RIGHTS	MACH_PORT_TYPE_SEND_RIGHTS, MACH_PORT_TYPE_RECEIVE
MACH_PORT_TYPE_PORT_OR_DEAD	MACH_PORT_TYPE_PORT_RIGHTS, MACH_PORT_TYPE_DEAD_NAME
MACH_PORT_TYPE_ALL_RIGHTS	MACH_PORT_TYPE_PORT_OR_DEAD, MACH_PORT_TYPE_PORT_SET

Don't confuse the `MACH_PORT_RIGHT_*` with the `MACH_PORT_TYPE_*` macros. They have similar names, but different meanings as well as different values.

More details on Mach IPC can be found in the GNU Mach Reference Manual, Chapter 4 ("Inter Process Communication")[1].

4.2 Threads and Tasks

4.3 MIG

Making RPCs (Remote Procedure Calls) by sending Mach messages is not trivial. Making it easier is the purpose of MIG (Mach Interface Generator). You must write an interface definition file, feed it into MIG and then it outputs two C sources and a header file, which do the Mach port magic for you. Then you can send messages with simple function calls.

Of course, you only need to do that if you want to define your own interfaces. The Hurd already contains various interfaces. The apropriate functions are in glibc, thus you don't have to specify any special flags if you want to use them.

The syntax of MIG files is similar to Pascal and should not be hard to understand. We will talk about some details later.

[1] `http://www.gnu.org/software/hurd/gnumach-doc/Inter-Process-Communication.html`

5 The Hurd Interfaces

Understanding concepts is a very important thing, but this alone will not enable you to get any work done. You also need to know about the interfaces which make it possible to use those concepts.

You can find the interface definitions in the `$(HURD)/hurd/*.defs` files. Important are `io.defs`, `password.defs`, `fsys.defs`, `fs.defs` and `auth.defs`. The `login.defs` interface is nice, but not implemented so far and maybe it will never be. The `*_reply.defs` interfaces are — of course — for reply messages.

You definitively should take a look at the Hurd interfaces now. In the next chapter, we will see how one uses those interfaces.

6 How does this look in practice?

6.1 Writing a file to standard output

Let's take a closer look at a program that dumps a file to its standard output in a "hurdish" way. Please note that the non-Hurd way may still be used. Glibc still provides functions like `write ()` on GNU/Hurd systems. The Hurd generic parts of glibc are in `$(GLIBC)/hurd/`, Mach dependent parts are in `$(GLIBC)/sysdeps/mach/hurd/`.

```
/* dump.c - Dump a file to stdout in a "hurdish" way.
 *
 * Copyright (C) 2001, 2002, 2007 Free Software Foundation, Inc.
 *
 * Written by Wolfgang Jährling <wolfgang@pro-linux.de>.
 *
 * Distributed under the terms of the GNU General Public License.
 * This is distributed "as is".  No warranty is provided at all.
 */

#define _GNU_SOURCE 1

#include <hurd.h>
#include <hurd/io.h>

#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <error.h>

int
main (int argc, char *argv[])
{
  file_t f;
  mach_msg_type_number_t amount;
  char *buf;
  error_t err;

  if (argc != 2)
    error (1, 0, "Usage: %s <filename>", argv[0]);

  /* Open file */
  f = file_name_lookup (argv[1], O_READ, 0);
  if (f == MACH_PORT_NULL)
    error (1, errno, "Could not open %s", argv[1]);

  /* Get size of file (buggy! See below) */
  err = io_readable (f, &amount);
```

```
  if (err)
    error (1, err, "Could not get number of readable bytes");

  /* Create buffer */
  buf = malloc (amount + 1);
  if (buf == NULL)
    error (1, 0, "Out of memory");

  /* Read */
  err = io_read (f, &buf, &amount, -1, amount);
  if (err)
    error (1, errno, "Could not read from file %s", argv[1]);
  buf[amount] = '\0';
  mach_port_deallocate (mach_task_self (), f);

  /* Output */
  printf ("%s", buf);
  return 0;
}
```

You may compile this with:

```
$ gcc -g -o dump dump.c
```

But let's look at the interesting pieces of this program:

```
#define _GNU_SOURCE 1
```

You should always define this macro for GNU/Hurd specific programs. Otherwise, you will not even be able to compile them. Define it before including any headers. Alternatively, you might want to pass `-D_GNU_SOURCE` to gcc, by adding it to `CPPFLAGS`, for example.

```
file_t f;
```

A `file_t` is actually a `mach_port_t`, but we use `file_t` to make clear we use this to open a file.

```
char *buf;
```

You may have noticed that we will use this buffer in a situation where we should pass a `data_t` (which is typedef'ed as a `char *`), but `$(HURD)/hurd/hurd_types.h` states about `data_t` and several other types:

> "These names exist only because of MIG deficiencies. You should not use them in C source; use the normal C types instead."

```
error_t err;
```

There is also the Mach type `kern_return_t`, but in the Hurd, `error_t` is the better choice. Marcus Brinkmann explains:

> "kern_return_t is the mach error type from mach_msg for example, so if you call an RPC, and want to do it in a Mach compatible fashion, use kern_return_t. BUT on the Hurd, we use error_t, because that is compatible with the glibc error types. You can always cast from kern_return_t to error_t on GNU systems."

```
if (argc != 2)
  error (1, 0, "Usage: %s <filename>", argv[0]);
```

Just in case you are not familiar with the `error ()` function, I will quote `/include/error.h` (Remember we don't use /usr in the Hurd :-)):

```
/* Print a message with ‘fprintf (stderr, FORMAT, ...)’;
   if ERRNUM is nonzero, follow it with ": " and strerror (ERRNUM).
   If STATUS is nonzero, terminate the program with ‘exit (STATUS)’.  */
extern void error (int status, int errnum, const char *format, ...);
```

Now the actual action starts. We try to open the file with the glibc function `file_name_lookup ()`. This function returns `MACH_PORT_NULL` if the attempt failed. `O_READ` is a GNU extension and is the same as the POSIX constant `O_RDONLY`. In the same way, `O_WRITE` is identical to `O_WRONLY`.

```
/* Open file */
f = file_name_lookup (argv[1], O_READ, 0);
if (f == MACH_PORT_NULL)
  error (1, errno, "Could not open %s", argv[1]);
```

Of course we could read the file character-by-character, but in this example we assume that it is a “normal” file and may be read at once, so we use `io_readable ()` to find out about the size of the file (this won't work for files like `/dev/random`! `io_readable ()` only tells us how much data is available right now). Then we allocate a buffer for the whole file:

```
/* Get size of file */
err = io_readable (f, &amount);
if (err)
  error (1, err, "Could not get number of readable bytes");

/* Create buffer */
buf = malloc (amount + 1);
if (buf == NULL)
  error (1, 0, "Out of memory");
```

Now all we have to do is reading the file into the buffer. We put a 0-byte at the end, so we will be able to use the file content as a string (we assume that the file does not contain any 0-bytes, which is bad style, but in this case, we don't care).

```
/* Read */
err = io_read (f, &buf, &amount, -1, amount);
if (err)
  error (1, errno, "Could not read from file %s", argv[1]);
buf[amount] = '\0';
```

Note that we pass `&buf`, which is a pointer to a pointer. `buf` itself might get modified, but this decision is up to the receiver of the `io_read` message.

If you look at `$(HURD)/hurd/io.defs`, you will probably wonder that `io_read` only has four arguments, while we passed five. `io_object`, `data`, `offset` and `amount` are written down in the `.defs` file, while `/include/hurd/io.h` has an additional argument (called `dataCnt`) after data, which makes perfectly sense: we also need to get the information which amount of data we actually got. Adding this argument is done automatically by MIG.

We are finished using the file, so we can close it. This is done by deallocating the port:

```
mach_port_deallocate (mach_task_self (), f);
```

That's all.

6.2 Creating a copy of a file

Now we will try to copy a file. But this time, we will do it right: if the translator that provides the file takes while to deliver new data, we will wait that while. So we will read until we reach a real EOF. We know that we reached the end of the file if our call to `io_read ()` gives us zero bytes of data.

```
/* copy.c - Copy a file in a "hurdish" way.
 *
 * Copyright (C) 2001, 2007 Free Software Foundation, Inc.
 *
 * Written by Wolfgang Jährling <wolfgang@pro-linux.de>.
 *
 * Distributed under the terms of the GNU General Public License.
 * This is distributed "as is".  No warranty is provided at all.
 */

#define _GNU_SOURCE 1

#include <hurd.h>
#include <hurd/io.h>

#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <error.h>

#define BUFLEN 10  /* Arbitrary */

int
main (int argc, char *argv[])
{
  file_t in, out;
  mach_msg_type_number_t rd_amount, wr_amount;
  char *buf, *ptr;
  error_t err;

  if (argc != 3)
    error (1, 0, "Usage: %s <inputfile> <outputfile>", argv[0]);

  /* Create buffer */
  buf = malloc (BUFLEN + 1);
  if (buf == NULL)
    error (1, 0, "Out of memory");

  /* Open files */
```

```
  in = file_name_lookup (argv[1], O_READ, 0);
  if (in == MACH_PORT_NULL)
    error (1, errno, "Could not open %s", argv[1]);
  out = file_name_lookup (argv[2], O_WRITE | O_CREAT | O_TRUNC, 0640);
  if (out == MACH_PORT_NULL)
    error (1, errno, "Could not open %s", argv[2]);

  /* Copy */
  while (1)
    {
      /* Read */
      err = io_read (in, &buf, &rd_amount, -1, BUFLEN);
      if (err)
        error (1, err, "Could not read from file %s", argv[1]);

      if (rd_amount == 0)
        break;

      /* Write */
      ptr = buf;
      do
        {
          err = io_write (out, ptr, rd_amount, -1, &wr_amount);
          if (err)
            error (1, err, "Could not write to file %s", argv[2]);
          rd_amount -= wr_amount;
          ptr += wr_amount;
        }
      while (rd_amount);
    }

  mach_port_deallocate (mach_task_self (), in);
  mach_port_deallocate (mach_task_self (), out);
  return 0;
}
```

Interesting parts are:

```
out = file_name_lookup (argv[2], O_WRITE | O_CREAT | O_TRUNC, 0640);
```

Here we open the output file and create it, if it does not exist, with permissions being 0640 (which is of course 'rw-r—-') minus the umask.

```
/* Read */
err = io_read (in, &buf, &rd_amount, -1, BUFLEN);
if (err)
  error (1, err, "Could not read from file %s", argv[1]);

if (rd_amount == 0)
  break;
```

As we said above: if we couldn't read any bytes, this indicates that we reached the end of the file. It does *not* mean that there is no data available at the moment: if we would read from `/dev/random` for example, and there would be no data to read at the moment, this call would not tell us that there is no data, but would block until new data is available.

```
/* Write */
ptr = buf;
do
  {
    err = io_write (out, ptr, rd_amount, -1, &wr_amount);
    if (err)
      error (1, err, "Could not write to file %s", argv[2]);
    rd_amount -= wr_amount;
    ptr += wr_amount;
  }
while (rd_amount);
```

There is no guarantee that `io_write ()` accepts all data we attempt to feed into it immediately, so we might need to retry, but only with the data that was not accepted until now.

6.3 Final notes

Finaly I would like to note that on the GNU/Hurd system there is no reason not to use the nice non-standard extensions of GCC. For example, nested functions are used frequently throughout the Hurd sources. It might be helpful to know about these extensions, so you should probably do

```
$ info gcc "C Extensions"
```

A nice example for "hurdish" code is `$(HURD)/init/init.c`, so you should also take a look at that. You maybe won't understand all of it, but that doesn't matter. More sources you might want to read are in `$(HURD)/utils/` and `$(HURD)/sutils/`.

7 The Hurd Libraries (Overview)

There are several libraries which make writing translators of various kinds easier.

libtrivfs is used for "trivial" translators. In this case, trivial translators means all translators that provide only a single file (node), as opposed to a complete directory or even a file system. As the first translators any new Hurd hacker will develop are simple single-file translators, this library is the first you should learn about.

libnetfs is the library for complete file systems where the translator does not directly control the underlying data, as is the case in ftpfs, nfs and unionfs[1], for example. libnetfs will probably be renamed to libfsserver.

libdiskfs is also for complete filesystems, but it is used in the case where the translator controls the underlying data. Examples are ext2fs, UFS and tmpfs.

libtreefs is defunct. It was never finished. Nobody uses it. Neither should you.

libports provides functions for working with ports. It can also be seen as an abstraction of what functionality the Hurd expects from a message-passing system.

libstore: the following explanation can be found in the libstore header file: "A 'store' is a fixed-size block of storage, which can be read and perhaps written to. This library implements many different backends which allow the abstract store interface to be used with common types of storage — devices, files, memory, tasks, etc. It also allows stores to be combined and filtered in various ways."

libiohelp: "Library providing helper functions for io servers."

libthreads is the cthreads library. This library comes from the Mach microkernel and was developed before the POSIX threads standard existed. We will have POSIX threads in the future, but currently this library is used for multithreading.

libihash provides integer-keyed hash table functions.

libps: "Routines to gather and print process information."

libshouldbeinlibc: Nomen est omen. :-)

---

[1] unionfs is not yet part of the Hurd, but a partly working implementation exists.

8 An Example using trivfs

8.1 GNU/Linux and GNU/Hurd

Before we take a closer look at how to use trivfs, let's see how this would be done on a GNU/Linux system. I won't explain the GNU/Linux example in much detail, because it is not very important for us, but since lots of people are familar with Linux (kernel) coding, it might be helpful to compare how things are done on GNU/Linux as opposed to GNU/Hurd.

Writing a Linux kernel module for a device file like `/dev/one` (which, of course, gives you infinite ones if you read from it is easy in theory. A module for kernel 2.4.x providing a special file might look like this:

```
/* linux-one.c - Linux kernel module for /dev/one.
 *
 * Copyright (C) 2000, 2001, 2007 Free Software Foundation, Inc.
 *
 * Written by Wolfgang Jährling <wolfgang@pro-linux.de>.
 *
 * Distributed under the terms of the GNU General Public License.
 * This is distributed "as is".  No warranty is provided at all.
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/wrapper.h>
#include <asm/uaccess.h>

#define ONE_NAME "one"
#define ONE_MAJOR 100 /* Major device file number */

static int is_opened = 0;

/* Someone wants to open the file */
static int
device_open (struct inode *inode, struct file *file)
{
  /* We allow only one simultaneous usage */
  if (is_opened)
    return -EBUSY;

  is_opened = 1;
  MOD_INC_USE_COUNT; /* Module can't be unloaded now */

  return 0; /* Could be opened */
}
```

```
/* The file is closed again */
static int
device_release (struct inode *inode, struct file *file)
{
  is_opened = 0;
  MOD_DEC_USE_COUNT; /* Module may be unloaded now */

  return 0;
}

/* Somebody wants to have lots of one's */
static ssize_t
device_read (struct file *file, char *buf, size_t len, loff_t *offset)
{
  int i;
  static char one = '1';

  for (i = 0; i < len; i++)
    if (copy_to_user (&buf[i], &one, 1))
      return -EFAULT;

  return len;
}

/* Now he/she wants to write something...  */
static ssize_t
device_write (struct file *file, const char *buf, size_t len,
              loff_t *offset)
{
  /* ...but we don't care */
  return len;
}

/* Let's put the supported operations in a structure */
struct file_operations one_operations =
  {
    NULL, /* Owner module...  wonder what this means :-) */
    NULL, /* seek */
    device_read,
    device_write,
    NULL, /* readdir */
    NULL, /* poll */
    NULL, /* ioctl */
    NULL, /* mmap */
    device_open
    NULL, /* flush */
    device_release,
```

```
    NULL, NULL, NULL, NULL, NULL /* Some others */
  };

/* This is automatically called when the module is loaded */
int
init_module (void)
{
  int result = register_chrdev (ONE_MAJOR, ONE_NAME, &one_operations);

  if (result < 0) /* Could not register character device */
    {
      printk (KERN_ERR "Couldn't register device: %d.\n", result);
      return result;
    }
  printk (KERN_INFO "Loading the %s module.\n", ONE_NAME);
  return 0;
}

/* This gets called when unloading the module */
void
cleanup_module (void)
{
  int result = unregister_chrdev (ONE_MAJOR, ONE_NAME);

  if (result < 0)
    printk (KERN_ERR "Couldn't unregister device: %d.\n", result);
  else
    printk (KERN_INFO "Unloading the %s module.\n", ONE_NAME);
}
```

We simply implement the usual operations like `read ()`, `close ()` and `open ()`, put pointers to these functions in a struct and register this as a character device.

In the next step, we would compile this into an object file and load it as root with insmod(8) and create the device file with

```
# mknod /dev/one c 100 1
```

This is simple to understand — but hard to put into practice, for (at least) four reasons: first, a small mistake in the code might cause a kernel panic; second, you need root privileges to do this at all; third, you can't use functions provided by the GNU C library, let alone other helpful libraries like the GLib; fourth, you need an unused device number (I used 100 above and hoped that nobody else used that before).

With the Hurd, things work different though. Of course, the superstructure looks quite different, but also (and more importantly) the environment of the code is much more friendly: it's the 'normal' user space we all know and love. This effectively means that you can develop a translator almost like any other program. The only disadvantage is that the programming interface is a bit more complex than the one of the Linux kernel. But if you understood the GNU/Linux example above, you won't have problems with the following translator, which implements the same functionality.

In fact, it provides much more functionality, because libtrivfs forces us to implement more; when writing a real world translator, you should also do option parsing, because sometimes users only ask for '--help', but I tried to keep this example translator as simple as possible. When writing programs for the GNU system, we recommend parsing options with the argp functions[1].

8.2 Implementing trivfs callback functions

In the GNU/Hurd system, the usual Unix system calls are provided by the GNU C Library. The GNU C Library wrapps them to messages and sends them to the respective ports. This means that you won't write direct implementations for functions like `read ()`, but one needs functions with the arguments of, say, `io_read ()`. When using the trivfs library, we have to implement routines with slightly different arguments. For example, the arguments of `trivfs_S_io_read ()`, as the name of such a function would be when using libtrivfs, are:

Type	Name	Description
`struct trivfs_protid *`	`cred`	Credentials
`mach_port_t`	`reply`	The port where the reply will be sent
`mach_msg_type_name_t`	`reply_type`	The rights we have on the above port
`vm_address_t *`	`data`	Pointer to the place where you should write you reply data to
`mach_msg_type_number_t`	`data_len`	Here you should store, how much data you actually return. Initialy, this is set to the size of the already available memory at `*data`.
`loff_t`	`offs`	Seek a position. If `offs` is -1, use the internal file pointer. Ignore it if the object is not seekable.
`mach_msg_type_number_t`	`amount`	How much data you should write

The `trivfs_S_io_read ()` function of the "Hello, world" translator (see `$(HURD)/trans/hello.c`) is a nice example for how to implement such a function. The implementation of our "one" translator will be a less complete example.

This is how our function looks like:

```
error_t
trivfs_S_io_read (struct trivfs_protid *cred,
                  mach_port_t reply, mach_msg_type_name_t reply_type,
                  data_t *data, mach_msg_type_number_t *data_len,
                  loff_t offs, mach_msg_type_number_t amount)
{
  /* Deny access if they have bad credentials.  */
  if (!cred)
    return EOPNOTSUPP;
  else if (! (cred->po->openmodes & O_READ))
```

[1] see "info libc Argp"

```
      return EBADF;

    if (amount > 0)
      {
        int i;

        /* Possibly allocate a new buffer.  */
        if (*data_len < amount)
          *data = mmap (0, amount, PROT_READ|PROT_WRITE, MAP_ANON, 0, 0);

        /* Copy the constant data into the buffer.  */
        for (i = 0; i < amount; i++)
          (*data)[i] = '1';
      }

    *data_len = amount;
    return 0;
  }
```

This is the most complex callback function of our translator. The others are much simpler.

You should always return `EOPNOTSUPP` (Operation not supported) if `cred` is faulty and `EBADF` (Bad file descriptor) if the necessary bit is not set in the open mode.

If the user wants to read more bytes (the number in `amount`) than the buffer can hold, we have to allocate some more memory. Do you remember that we had to pass a pointer to the pointer to our buffer, when calling `io_read ()`? This was done exactly for this reason. We are allocating the memory with `mmap ()` here, because we want a page aligned memory block. Now we made sure that we've got enough space where we can write the data into. This means we can begin filling in the ones. Note that `*data` is a `vm_address_t` and we have to cast it into a pointer before we can use it as such.

If you understand the above function, I doubt you will have troubles with the following write routine, which does almost nothing:

```
kern_return_t
trivfs_S_io_write (struct trivfs_protid *cred,
                   mach_port_t reply, mach_msg_type_name_t replytype,
                   data_t data, mach_msg_type_number_t datalen,
                   loff_t offs, mach_msg_type_number_t *amout)
{
  if (!cred)
    return EOPNOTSUPP;
  else if (!(cred->po->openmodes & O_WRITE))
    return EBADF;
  *amout = datalen;
  return 0;
}
```

Apart from the usual error checking it only claims that all data the user of our translator (i.e. the program which opened the file our translator implements) wanted to write was successfully written — which makes perfect sense, because we ignore all data someone writes into our file.

There are several callbacks we will implement in a similar way like the write function. You can find these functions in the complete source code of our translator.

Another callback routine is `trivfs_S_io_readable ()`. It will be called if somebody wants to know how much data we can deliver immediately. Of course we can provide an infinite number of bytes directly. And as Marcus Brinkmann told me[2]:

> "If you can deliver an unlimited number of bytes without blocking, I think the highest possible value that fits in mach_msg_type_number_t seems to be appropriate. (I hope applications can deal with that)."

Well, if something can go wrong, it will go wrong. For example, our example program `dump.c` above would handle such a situation in a very ungraceful way. This is why I wrote the following implementation, which is a bit paranoid and does not cause problems if an application is unable to handle a huge value in a sane way:

```
kern_return_t
trivfs_S_io_readable (struct trivfs_protid *cred,
                      mach_port_t reply, mach_msg_type_name_t replytype,
                      mach_msg_type_number_t *amount)
{
  if (!cred)
    return EOPNOTSUPP;
  else if (!(cred->po->openmodes & O_READ))
    return EINVAL;
  else
    *amount = 10240; /* Dummy value: 10k */
  return 0;
}
```

The last interesting callback function is `trivfs_S_io_select ()`, which is well commented in the complete source bellow.

8.3 Other trivfs callbacks

We have to do some more work before we have a complete trivfs translator: we must define some symbols that hold general information about our translator.

```
/* Trivfs hooks.  */
int trivfs_fstype = FSTYPE_MISC;  /* Generic trivfs server */
int trivfs_fsid = 0;              /* Should always be 0 on startup */
```

In most cases, you might want to set `trivfs_fstype` to `FSTYPE_MISC`. Other possible values are (descriptions from `$(HURD)/hurd/hurd_types.h`):

1. `FSTYPE_IFSOCK`: `PF_LOCAL` socket naming point
2. `FSTYPE_DEV`: GNU Special file server
3. `FSTYPE_TERM`: GNU Terminal driver

[2] for the complete mail see `http://lists.gnu.org/archive/html/help-hurd/2001-08/msg00003.html`

In `trivfs_allow_open`, you specify the initial permissions for your translator:

```
int trivfs_allow_open = O_READ | O_WRITE;
```

And with the following three variables, you specify what kinds of accesses are actually implemented:

```
/* Actual supported modes: */
int trivfs_support_read  = 1;
int trivfs_support_write = 1;
int trivfs_support_exec  = 0;
```

8.4 The main function

Translators are normal programs, and as such, they need a `main ()` function. The trivfs library does not define such a function, so we have to do this on our own. Our program may be started as a normal program or as a translator, so we have to distinguish between these two cases. This can be done by testing if our bootstrap port is `MACH_PORT_NULL`. If it is, the program was not started as a translator. In most cases, a translator might simply abort in this case. If, however, our bootstrap port is not `MACH_PORT_NULL`, we should initialize libtrivfs and deallocate the bootstrap port. In the last step, we will launch the translator. Our `main ()` function (which does not process any command line arguments) looks like this:

```
int
main (void)
{
  error_t err;
  mach_port_t bootstrap;
  struct trivfs_control *fsys;

  task_get_bootstrap_port (mach_task_self (), &bootstrap);
  if (bootstrap == MACH_PORT_NULL)
    error (1, 0, "Must be started as a translator");

  /* Reply to our parent */
  err = trivfs_startup (bootstrap, 0, 0, 0, 0, 0, &fsys);
  mach_port_deallocate (mach_task_self (), bootstrap);
  if (err)
    error (1, err, "trivfs_startup failed");

  /* Launch.  */
  ports_manage_port_operations_one_thread (fsys->pi.bucket,
                                           trivfs_demuxer, 0);

  return 0;
}
```

You may have wondered why our functions had such disgusting names like `trivfs_S_io_read ()`. At least by now you should know the answer: We don't need to register our functions anywhere, we only have to give them the apropriate names and libtrivfs will do the rest for us. Of course, the function names actually have a meaning, as Marcus Brinkmann explains:

> "The io_read is the name of the RPC as in the .defs file. The S_ prefix means it is the _S_erver stub implemented here, rather than the message packaging/unpacking functions. The trivfs_ prefix means that this is not the bare RPC, but the mach_port_t is actually converted to a credential struct cred (or so). This is done at the INTRAN.
>
> Usually, you get the mach_port_t as first argument, but in libtrivfs stubs, you get a different. Grep for intran in `$(HURD)/libtrivfs/*` and you will see how ports are mapped to credentials."

8.5 The complete source

Okay, that's all, folks. I left out some unimportant details, which you may look up in the following complete listing of `hurd-one.c`. You can compile this file with

```
$ gcc -g -o one hurd-one.c -ltrivfs -lports
```

Other sources you might want to look at are `$(HURD)/trans/hello.c` and `$(HURD)/trans/null.c`.

```
/* hurd-one.c - A trivial single-file translator

   Copyright (C) 1995, 1996, 1997, 1998, 1999, 2001, 2002, 2007
     Free Software Foundation, Inc.

   Written by Wolfgang Jährling <wolfgang@pro-linux.de>, 2001

   This is based on hurd/trans/hello.c.  The hello.c source says:
     Copyright (C) 1998, 1999, 2001 Free Software Foundation, Inc.
     Gordon Matzigkeit <gord@fig.org>, 1999
   It also uses parts of hurd/trans/null.c.  The null.c source says:
     Copyright (C) 1995,96,97,98,99,2001 Free Software Foundation, Inc.
     Written by Miles Bader <miles@gnu.org>

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2, or (at
   your option) any later version.

   This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place, Suite 330,
   Boston, MA  02111-1307  USA */
```

```
#define _GNU_SOURCE 1

#include <hurd/trivfs.h>

#include <stdlib.h>   /* exit () */
#include <error.h>    /* Error numers */
#include <fcntl.h>    /* O_READ etc.  */
#include <sys/mman.h> /* MAP_ANON etc.  */

/* Trivfs hooks.  */
int trivfs_fstype = FSTYPE_MISC;  /* Generic trivfs server */
int trivfs_fsid = 0;              /* Should always be 0 on startup */

int trivfs_allow_open = O_READ | O_WRITE;

/* Actual supported modes: */
int trivfs_support_read  = 1;
int trivfs_support_write = 1;
int trivfs_support_exec  = 0;

/* May do nothing...  */
void
trivfs_modify_stat (struct trivfs_protid *cred, io_statbuf_t *st)
{
  /* ..  and we do nothing */
}

error_t
trivfs_goaway (struct trivfs_control *cntl, int flags)
{
  exit (EXIT_SUCCESS);
}

error_t
trivfs_S_io_read (struct trivfs_protid *cred,
                  mach_port_t reply, mach_msg_type_name_t reply_type,
                  data_t *data, mach_msg_type_number_t *data_len,
                  loff_t offs, mach_msg_type_number_t amount)
{
  /* Deny access if they have bad credentials.  */
  if (!cred)
    return EOPNOTSUPP;
  else if (!(cred->po->openmodes & O_READ))
    return EBADF;

  if (amount > 0)
```

```
    {
      int i;

      /* Possibly allocate a new buffer.  */
      if (*data_len < amount)
        *data = mmap (0, amount, PROT_READ|PROT_WRITE, MAP_ANON, 0, 0);

      /* Copy the constant data into the buffer.  */
      for (i = 0; i < amount; i++)
        ((char *) *data)[i] = 1;
    }

  *data_len = amount;
  return 0;
}

kern_return_t
trivfs_S_io_write (struct trivfs_protid *cred,
                   mach_port_t reply, mach_msg_type_name_t replytype,
                   data_t data, mach_msg_type_number_t datalen,
                   loff_t offs, mach_msg_type_number_t *amout)
{
  if (!cred)
    return EOPNOTSUPP;
  else if (!(cred->po->openmodes & O_WRITE))
    return EBADF;
  *amout = datalen;
  return 0;
}

/* Tell how much data can be read from the object without blocking for
   a "long time" (this should be the same meaning of "long time" used
   by the nonblocking flag.  */
kern_return_t
trivfs_S_io_readable (struct trivfs_protid *cred,
                      mach_port_t reply, mach_msg_type_name_t replytype,
                      mach_msg_type_number_t *amount)
{
  if (!cred)
    return EOPNOTSUPP;
  else if (!(cred->po->openmodes & O_READ))
    return EINVAL;
  else
    *amount = 10000; /* Dummy value */
  return 0;
}
```

```
/* Truncate file.  */
kern_return_t
trivfs_S_file_set_size (struct trivfs_protid *cred, mach_port_t reply,
                        mach_msg_type_name_t reply_type, loff_t size)
{
  if (!cred)
    return EOPNOTSUPP;
  else
    return 0;
}

/* Change current read/write offset */
error_t
trivfs_S_io_seek (struct trivfs_protid *cred, mach_port_t reply,
                  mach_msg_type_name_t reply_type, loff_t offs, int whence,
                  loff_t *new_offs)
{
  if (! cred)
    return EOPNOTSUPP;
  else
    return 0;
}

/* SELECT_TYPE is the bitwise OR of SELECT_READ, SELECT_WRITE, and
   SELECT_URG.  Block until one of the indicated types of i/o can be
   done "quickly", and return the types that are then available.
   TAG is returned as passed; it is just for the convenience of the
   user in matching up reply messages with specific requests sent.  */
kern_return_t
trivfs_S_io_select (struct trivfs_protid *cred,
                    mach_port_t reply, mach_msg_type_name_t replytype,
                    int *type, int *tag)
{
  if (!cred)
    return EOPNOTSUPP;
  else
    if (((*type & SELECT_READ) && !(cred->po->openmodes & O_READ))
        || ((*type & SELECT_WRITE) && !(cred->po->openmodes & O_WRITE)))
      return EBADF;
    else
      *type &= ~SELECT_URG;
  return 0;
}

/* Well, we have to define these four functions, so here we go: */

kern_return_t
```

```
trivfs_S_io_get_openmodes (struct trivfs_protid *cred, mach_port_t reply,
                           mach_msg_type_name_t replytype, int *bits)
{
  if (!cred)
    return EOPNOTSUPP;
  else
    {
      *bits = cred->po->openmodes;
      return 0;
    }
}

error_t
trivfs_S_io_set_all_openmodes (struct trivfs_protid *cred,
                               mach_port_t reply,
                               mach_msg_type_name_t replytype,
                               int mode)
{
  if (!cred)
    return EOPNOTSUPP;
  else
    return 0;
}

kern_return_t
trivfs_S_io_set_some_openmodes (struct trivfs_protid *cred,
                                mach_port_t reply,
                                mach_msg_type_name_t replytype,
                                int bits)
{
  if (!cred)
    return EOPNOTSUPP;
  else
    return 0;
}

kern_return_t
trivfs_S_io_clear_some_openmodes (struct trivfs_protid *cred,
                                  mach_port_t reply,
                                  mach_msg_type_name_t replytype,
                                  int bits)
{
  if (!cred)
    return EOPNOTSUPP;
  else
    return 0;
}
```

```
int
main (void)
{
  error_t err;
  mach_port_t bootstrap;
  struct trivfs_control *fsys;

  task_get_bootstrap_port (mach_task_self (), &bootstrap);
  if (bootstrap == MACH_PORT_NULL)
    error (1, 0, "Must be started as a translator");

  /* Reply to our parent */
  err = trivfs_startup (bootstrap, 0, 0, 0, 0, 0, &fsys);
  mach_port_deallocate (mach_task_self (), bootstrap);
  if (err)
    error (1, err, "trivfs_startup failed");

  /* Launch.  */
  ports_manage_port_operations_one_thread (fsys->pi.bucket,
                                           trivfs_demuxer, 0);

  return 0;
}
```

9 Debugging a translator

This chapter requires you to know how to use GDB, the GNU Debugger. If you did not use GDB before, I recommend reading the sample session chapter in the GDB Texinfo documentation. If info and the documentation are installed on your system, simply do

```
$ info gdb "Sample Session"
```

Ok, so now how does one debug a translator? It's pretty obvious, but I will explain it anyway. :-) The easiest way is to start your program as an active translator:

```
$ gcc -g -o one one.c -ltrivfs -lports
$ settrans -ac foo one
```

Now the translator is up and running. You can see it in the process list:

```
$ ps Aux
```

(We don't have POSIX 'ps' at the moment, so 'ps aux' won't work, sorry.) Now we need to attach to the running (respective waiting) process. For example, if the PID was 357, we would do:

```
$ gdb one 357
```

At the gdb prompt, we can now set breakpoints, then let the translator continue:

```
(gdb) break trivfs_S_io_read
(gdb) c
```

Now, you should switch to another screen window, xterm or similar. Enter a command like

```
$ cat foo
```

there and switch back to the terminal where you are running GDB. You will see that it stopped at the breakpoint. Now you can debug as usual. That's easy, isn't it?

If you are done, enter

```
(gdb) quit
```

and say that you want to detach the process. We can conclude by saying that you don't need any special technique for debugging a translator — at least such a simple one.

At this point, you probably have an idea about how to develop Hurd servers (translators). Often, you will need more than libtrivfs provides. For outdated information on other libraries, see the Hurd Reference Manual ('`info hurd`'), also available in `$(HURD)/doc/hurd.texi`, for up-to-date information read the appropriate header files.

10 Comprehensive trivfs example

TODO: Maybe a ‘cat’ translator?

11 An example using netfs

TODO

12 An example using diskfs

TODO

13 Frequently Asked Questions

Q: How can a translator access it's underlying node? A gzip translator must do this, for example.

A: The underlying node is returned by fsys_startup(). See `$(HURD)/hurd/fsys.defs`.

Q: Can one stack translators?

A: Yes, stacking active translators is possible, but you can't do it with passive translators.

Q: What is a 'protid'?

A: 'prot' means protection. Every protid structure denotes a unique user (i.e. client) of our translator. You can access per-open information via the 'po' field of the structure.

Q: Which kind of threads should I use if I want to write a program that will run on both GNU/Linux and GNU/Hurd?

A: Use pthreads. We will have pthreads eventually.

Q: How can we claim to be POSIX compliant without having pthreads?

A: First of all, pthreads are optional. Second, we do have pthreads, for example GNU Portable Threads (pth) does provide a non-preemtive pthreads emulation. This seems to be standard compliant, but of course it's not that useful, as most programs assume preemtive multi-threading. Jeroen Dekkers is working on "real" pthreads, which partially work, as of now.

Q: In the GNU Manifesto, RMS wrote that both C and LISP will be system languages. What about that?

A: The Scheme interpreter Guile is part of the GNU project. As of now, it provides only POSIX functionality, but there's no reason why nobody should add GNU specific stuff. Adding support for GNU functionality to various languages would be nice indeed. That's certainly not an urgent issue, however.

Q: Why should I learn about Mach if the Hurd switches to L4 soon?

A: As of now, the Hurd uses Mach and you need to know Mach basics to do Hurd work. Maybe the Hurd will run on L4 in the future, but currently it's very, very far away from doing so.

14 Appendices

14.1 Stuff to do

This document is a work in progress. There are several things that should be added. If you want to help, please contact me.

```
- Use consistent formating.  often, @code{} should be used but isn't
- Correct the remaining FIXMEs (ok, this one was obvious)
- Take OSKit-Mach into account
- Add Moritz' Mach device access example and link to mailing list
  archive with Daniel Wagners Mach device code
- Write about netfs and diskfs, add a longer trivfs example
- Describe more Mach and esp. MIG details
```