The GNU Hurd Reference Manual

Thomas Bushnell, BSG
Gordon Matzigkeit

1 Introduction

The GNU Hurd[1] is the GNU Project's replacement for the Unix kernel. The Hurd is a collection of servers that run on the Mach microkernel to implement file systems, network protocols, file access control, and other features that are normally implemented by the Unix kernel or similar kernels (such as Linux).

1.1 Audience

This manual is designed to be useful to everybody who is interested in using, administering, or programming the Hurd.

If you are an end-user and you are looking for help on running the Hurd, the first few chapters of this manual describe the essential parts of installing, starting up, and shutting down a Hurd workstation. If you need help with a specific program, the best way to use this manual is to find the program's name in the index and go directly to the appropriate section. You may also wish to try running `program --help`, which will display a brief usage message for *program* (see Chapter 3 [Foundations], page 14).

The rest of this manual is a technical discussion of the Hurd servers and their implementation, and would not be helpful until you want to learn how to modify the Hurd.

This manual is organized according to the subsystems of the Hurd, and each chapter begins with descriptions of utilities and servers that are related to that subsystem. If you are a system administrator, and you want to learn more about, say, the Hurd networking subsystem, you can skip to the networking chapter (see Chapter 11 [Networking], page 71), and read about the related utilities and servers.

Programmers who are interested in learning how to modify Hurd servers, or write new ones, should begin by learning about a microkernel to which the Hurd has been ported (currently only GNU Mach) and reading Chapter 3 [Foundations], page 14. You should then familiarize yourself with a subsystem that interests you by reading about existing servers and the libraries they use. At that point, you should be able to study the source code of existing Hurd servers and understand how they use the Hurd libraries.

The final level of mastery is learning the about the RPC interfaces which the Hurd libraries implement. The last section of each chapter describes any Hurd interfaces used in that subsystem. Those sections assume that you are perusing the referenced interface definitions as you read. After you have understood a given interface, you will be in a good position to improve the Hurd libraries, design your own interfaces, and implement new subsystems.

1.2 Features

The Hurd is not the most advanced operating system known to the planet (yet), but it does have a number of enticing features:

it's free software
: Anybody can use, modify, and redistribute it under the terms of the GNU General Public License (see Section 1.5 [Copying], page 3). The Hurd is part

[1] The name *Hurd* stands for "Hird of Unix-Replacing Daemons." The name *Hird* stands for "Hurd of Interfaces Representing Depth."

of the GNU system, which is a complete operating system licensed under the GPL.

it's compatible
: The Hurd provides a familiar programming and user environment. For all intents and purposes, the Hurd is a modern Unix-like kernel. The Hurd uses the GNU C Library, whose development closely tracks standards such as ANSI/ISO, BSD, POSIX, Single Unix, SVID, and X/Open.

it's built to survive
: Unlike other popular kernel software, the Hurd has an object-oriented structure that allows it to evolve without compromising its design. This structure will help the Hurd undergo major redesign and modifications without having to be entirely rewritten.

it's scalable
: The Hurd implementation is aggressively multithreaded so that it runs efficiently on both single processors and symmetric multiprocessors. The Hurd interfaces are designed to allow transparent network clusters (*collectives*), although this feature has not yet been implemented.

it's extensible
: The Hurd is an attractive platform for learning how to become a kernel hacker or for implementing new ideas in kernel technology. Every part of the system is designed to be modified and extended.

it's stable
: It is possible to develop and test new Hurd kernel components without rebooting the machine (not even accidentally). Running your own kernel components doesn't interfere with other users, and so no special system privileges are required. The mechanism for kernel extensions is secure by design: it is impossible to impose your changes upon other users unless they authorize them or you are the system administrator.

it exists
: The Hurd is real software that works Right Now. It is not a research project or a proposal. You don't have to wait at all before you can start using and developing it.

1.3 Overview

An operating system kernel provides a framework for programs to share a computer's hardware resources securely and efficiently. This framework includes mechanisms for programs to communicate safely, even if they do not trust one another (see Section 3.2 [Ports Library], page 14).

The GNU Hurd divides up the work of the traditional kernel, and implements it in separate programs, or *kernel servers*. The Hurd formally defines the communication protocols that each of the servers understands, so that it is possible for different servers to implement the same interface.

The GNU C Library provides a POSIX environment on the Hurd, by translating standard POSIX system calls into interactions with the appropriate Hurd server.

1.4 History

Richard Stallman (RMS) started GNU in 1983, as a project to create a complete free operating system. In the text of the GNU Manifesto, he mentioned that there is a primitive kernel. In the first GNUsletter, Feb. 1986, he says that GNU's kernel is TRIX, which was developed at the Massachusetts Institute of Technology.

By December of 1986, the Free Software Foundation (FSF) had "started working on the changes needed to TRIX" [Gnusletter, Jan. 1987]. Shortly thereafter, the FSF began "negotiating with Professor Rashid of Carnegie-Mellon University about working with them on the development of the Mach kernel" [Gnusletter, June, 1987]. The text implies that the FSF wanted to use someone else's work, rather than have to fix TRIX.

In [Gnusletter, Feb. 1988], RMS was talking about taking Mach and putting the Berkeley Sprite filesystem on top of it, "after the parts of Berkeley Unix. . . have been replaced."

Six months later, the FSF is saying that "if we can't get Mach, we'll use TRIX or Berkeley's Sprite." Here, they present Sprite as a full-kernel option, rather than just a filesystem.

In January, 1990, they say "we aren't doing any kernel work. It does not make sense for us to start a kernel project now, when we still hope to use Mach" [Gnusletter, Jan. 1990]. Nothing significant occurs until 1991, when a more detailed plan is announced:

> "We are still interested in a multi-process kernel running on top of Mach. The CMU lawyers are currently deciding if they can release Mach with distribution conditions that will enable us to distribute it. If they decide to do so, then we will probably start work. CMU has available under the same terms as Mach a single-server partial Unix emulator named Poe; it is rather slow and provides minimal functionality. We would probably begin by extending Poe to provide full functionality. Later we hope to have a modular emulator divided into multiple processes." [Gnusletter, Jan. 1991].

RMS explains the relationship between the Hurd and Linux in `http://www.gnu.org/software/hurd/hurd-` where he mentions that the FSF started developing the Hurd in 1990. As of [Gnusletter, Nov. 1991], the Hurd (running on Mach) is GNU's official kernel.

1.5 GNU General Public License

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software

which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

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
of the License, 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.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

2 Bootstrap

Bootstrapping[1] is the procedure by which your machine loads the microkernel and transfers control to the Hurd servers.

2.1 Bootloader

The *bootloader* is the first software that runs on your machine. Many hardware architectures have a very simple startup routine which reads a very simple bootloader from the beginning of the internal hard disk, then transfers control to it. Other architectures have startup routines which are able to understand more of the contents of the hard disk, and directly start a more advanced bootloader.

GNU *GRUB*[2] is the GNU bootloader. GNU GRUB provides advanced functionality, and is capable of loading several different kernels (such as Linux, DOS, and the *BSD family).

From the standpoint of the Hurd, the bootloader is just a mechanism to get the microkernel running and transfer control to the Hurd servers. You will need to refer to your bootloader and microkernel documentation for more information about the details of this process.

2.2 Server Bootstrap

As it can be seen from the GNU GRUB configuration, the Hurd is bootstrapped by starting the GNU Mach microkernel and two programs: the root filesystem and the exec server.

The '`--multiboot-command-line`' option tells the file system server that it is a root filesystem, which triggers it to run `/hurd/init` as PID 1. `/hurd/init` starts the `/hurd/proc` and `/hurd/auth` servers. After the servers are launched `/hurd/init` starts the `/libexec/runsystem.sh` script to finish booting.

After the Hurd has been booted, other sets of core Hurd servers can be started in parallel. They're called sub-Hurds or neighbor Hurds (see Section 2.2.1 [Recursive Bootstrap], page 10).

2.2.1 Recursive Bootstrap

The `boot` program can be used to start a set of core Hurd servers while another Hurd is already running. You will rarely need to do this, and it requires superuser privileges to control the new Hurd (or allow it to access certain devices), but it is interesting to note that it can be done.

Usually, you would make changes to only one server, and simply tell your programs to use it in order to test out your changes. This process can be applied even to the core servers. However, some changes have far-reaching effects, and so it is nice to be able to test those effects without having to reboot the machine.

Here are the steps you can follow to test out a new set of servers:

---

[1] The term *bootstrapping* refers to a Dutch legend about a boy who was able to fly by pulling himself up by his bootstraps. In computers, this term refers to any process where a simple system activates a more complicated system.

[2] The GRand Unified Bootloader, available from `http://www.gnu.org/software/grub/`.

1. Create a pseudo-root device. Usually, you would do this by creating a new partition under your old Hurd, and initializing it with your favorite filesystem format. `boot` understands the regular `libstore` options (FIXME xref), so you may use a file or other store instead of a partition.

```
$ dd if=/dev/zero of=my-partition bs=1024k count=400
400+0 records in
400+0 records out
$ mke2fs ./my-partition
mke2fs 1.18, 11-Nov-1999 for EXT2 FS 0.5b, 95/08/09
my-partition is not a block special device.
Proceed anyway? (y,n) y
Filesystem label=
OS type: GNU/Hurd
Block size=1024 (log=0)
Fragment size=1024 (log=0)
102400 inodes, 409600 blocks
20480 blocks (5.00%) reserved for the super user
First data block=1
50 block groups
8192 blocks per group, 8192 fragments per group
2048 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729, 204801, 221185, 401409

Writing inode tables: done
Writing superblocks and filesystem accounting information: done
$
```

2. Copy the core servers, C library, your modified programs, and anything else you need onto the pseudo-root.

```
$ settrans -c ./my-root /hurd/ext2fs -r `pwd`/my-partition
$ fsysopts ./my-root --writable
$ cd my-root
$ tar -zxpf /pub/debian/FIXME/gnu-20000929.tar.gz
$ cd ..
$ fsysopts ./my-root --readonly
```

3. Create a new boot script (FIXME xref).
4. Run `boot`.

```
$ boot -D ./my-boot ./my-boot/boot/servers.boot ./my-partition
[...]
```

5. Here is an example using a hard drive that already has a GNU/Hurd system installed on an ext2 filesystem on '`/dev/hd2s1`'.

```
$ settrans /mnt /hurd/ex2fs --readonly /dev/hd2s1
$ boot -d -D /mnt -I /mnt/boot/servers.boot /dev/hd2s1
```

6. See see Section 2.2.3 [Invoking boot], page 12 for help with boot.

Note that it is impossible to share microkernel devices between the two running Hurds, so don't get any funny ideas. When you're finished testing your new Hurd, then you can run the `halt` or `reboot` programs to return control to the parent Hurd.

If you're satisfied with your new Hurd, you can arrange for your bootloader to start it, and reboot your machine. Then, you'll be in a safe place to overwrite your old Hurd with the new one, and reboot back to your old configuration (with the new Hurd servers).

2.2.2 Boot Scripts

Boot Scripts are used to boot further Hurd systems in parallel to the first, and are parsed by `boot` to boot a sub-Hurd.

In that script, the variables *host-port* and *device-port* are integers which represent the microkernel host and device ports, respectively (and are used to initialize the `${host-port}` and `${device-port}` boot script variables). If these ports are not specified, then it is assumed that the Hurd is already running, and the current ports will be fetched from the procserver.

root-name is the name of the microkernel device that should be used as the Hurd bootstrap filesystem.

FIXME: finish

2.2.3 Invoking boot

Usage: boot [*option*...] *boot-script device*...

`--kernel-command-line=`*`command line`*
`-c` Simulated multiboot command line to supply.

`--pause`
`-d` Pause for user confirmation at various times during booting.

`--boot-root=`*`dir`*
`-D` Root of a directory tree in which to find the files specified in *boot-script*.

`--interleave=`*`blocks`*
Interleave in runs of length *blocks*.

`--isig`
`-I` Do not disable terminal signals, so you can suspend and interrupt the boot program itself, rather than the programs running in the booted system.

`--layer`
`-L` Layer multiple devices for redundancy.

`--single-user`
`-s` Boot into single user mode.

`--store-type=`*`type`*
`-T` Each *device* names a store of type *type*.

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

If neither '`--interleave`' or '`--layer`' is specified, multiple *device*s are concatenated.

2.3 Shutdown

FIXME: finish

3 Foundations

Every Hurd program accepts the following optional arguments:

'--help' Display a brief usage message, then exit. This message is not a substitute for reading program documentation; rather, it provides useful reminders about specific command-line options that a program understands.

'--version'
Output program version information and exit.

The rest of this chapter provides a programmer's introduction to the Hurd. If you are not a programmer, then this chapter will not make much sense to you... you should consider skipping to descriptions of specific Hurd programs (see Section 1.1 [Audience], page 1).

The Hurd distribution includes many libraries in order to provide a useful set of tools for writing Hurd utilities and servers. Several of these libraries are useful not only for the Hurd, but also for writing microkernel-based programs in general. These fundamental libraries are not difficult to understand, and they are a good starting point, because the rest of the Hurd relies upon them quite heavily.

3.1 Threads Library

All Hurd servers and libraries are aggressively multithreaded in order to take full advantage of any multiprocessing capabilities provided by the microkernel and the underlying hardware. The Hurd threads library, `libthreads`, contains the default Hurd thread implementation, which is declared in <cthreads.h>.

Currently (April 1998), the Hurd uses cthreads, which have already been documented thoroughly by CMU. Eventually, it will be migrated to use POSIX pthreads, which are documented in a lot of places.

Every single library in the Hurd distribution (including the GNU C library) is completely thread-safe, and the Hurd servers themselves are aggressively multithreaded.

3.2 Ports Library

Ports are communication channels that are held by the kernel.

A port has separate send rights and receive rights, which may be transferred from task to task via the kernel. Port rights are similar to Unix file descriptors: they are per-task integers which are used to identify ports when making kernel calls. Send rights are required in order to send an RPC request down a port, and receive rights are required to serve the RPC request. Receive rights may be aggregated into a single *portset*, which serve as useful organizational units.

In a single-threaded RPC client, managing and categorizing ports is not a difficult process. However, in a complex multithreaded server, it is useful to have a more abstract interface to managing portsets, as well as maintaining server metadata.

The Hurd ports library, `libports`, fills that need. The `libports` functions are declared in <hurd/ports.h>.

3.2.1 Buckets and Classes

The `libports` *bucket* is simply a port set, with some metadata and a lock. All of the `libports` functions operate on buckets.

`struct port_bucket * ports_create_bucket` (*void*) [Function]

Create and return a new, empty bucket.

A port *class* is a collection of individual ports, which can be manipulated conveniently, and have enforced deallocation routines. Buckets and classes are entirely orthogonal: there is no requirement that all the ports in a class be in the same bucket, nor is there a requirement that all the ports in a bucket be in the same class.

`struct port_class ports_create_class` [Function]
(*void* (**clean_routine*) (*void* **port*),
void (**dropweak_routine*) (*void* **port*))

Create and return a new port class. If nonzero, *clean_routine* will be called for each allocated port object in this class when it is being destroyed. If nonzero, *dropweak_routine* will be called to request weak references to be dropped. (If *dropweak_routine* is null, then weak references and hard references will be identical for ports of this class.)

Once you have created at least one bucket and class, you may create new ports, and store them in those buckets. There are a few different functions for port creation, depending on your application's requirements:

`error_t ports_create_port` (*struct port_class* **class*, [Function]
struct port_bucket **bucket*, *size_t* `size`, *void* **result*)

Create and return in *result* a new port in *class* and *bucket*; *size* bytes will be allocated to hold the port structure and whatever private data the user desires.

`error_t ports_create_port_noinstall` (*struct port_class* **class*, [Function]
struct port_bucket **bucket*, *size_t* `size`, *void* **result*)

Just like `ports_create_port`, except don't actually put the port into the portset underlying *bucket*. This is intended to be used for cases where the port right must be given out before the port is fully initialized; with this call you are guaranteed that no RPC service will occur on the port until you have finished initializing it and installed it into the portset yourself.

`error_t ports_import_port` (*struct port_class* **class*, [Function]
struct port_bucket **bucket*, *mach_port_t* `port`, *size_t* `size`, *void* **result*)

For an existing *receive* right, create and return in *result* a new port structure; *bucket*, *size*, and *class* args are as for `ports_create_port`.

3.2.2 Port Rights

The following functions move port receive rights to and from the port structure:

`void ports_reallocate_port` (*void* **port*) [Function]

Destroy the receive right currently associated with *port* and allocate a new one.

`void ports_reallocate_from_external` (*void *port*, *mach_port_t* *receive*) [Function]

Destroy the receive right currently associated with *port* and designate *receive* as the new one.

`void ports_destroy_right` (*void *port*) [Function]

Destroy the receive right currently associated with *port*. After this call, `ports_reallocate_port` and `ports_reallocate_from_external` may not be used.

`mach_port_t ports_claim_right` (*void *port*) [Function]

Return the receive right currently associated with *port*. The effects on *port* are the same as in `ports_destroy_right`, except that the receive right itself is not affected. Note that in multi-threaded servers, messages might already have been dequeued for this port before it gets removed from the portset; such messages will get `EOPNOTSUPP` errors.

`error_t ports_transfer_right` (*void *topt*, *void *frompt*) [Function]

Transfer the receive right from *frompt* to *topt*. *frompt* ends up with a destroyed right (as if `ports_destroy_right` were called) and *topt*'s old right is destroyed (as if `ports_reallocate_from_external` were called).

`mach_port_t ports_get_right` (*void *port*) [Function]

Return the name of the receive right associated with *port*. The user is responsible for creating an ordinary send right from this name.

3.2.3 Port Metadata

It is important to point out that the *port* argument to each of the `libports` functions is a `void *` and not a `struct port_info *`. This is done so that you may add arbitrary meta-information to your `libports`-managed ports. Simply define your own structure whose first element is a `struct port_info`, and then you can use pointers to these structures as the *port* argument to any `libports` function.

The following functions are useful for maintaining metadata that is stored in your own custom ports structure:

`void * ports_lookup_port` (*struct port_bucket *bucket*, *mach_port_t port*, *struct port_class *class*) [Function]

Look up *port* and return the associated port structure, allocating a reference. If the call fails, return zero. If *bucket* is nonzero, then it specifies a bucket to search; otherwise all buckets will be searched. If *class* is nonzero, then the lookup will fail if *port* is not in *class*.

`error_t ports_bucket_iterate` (*struct port_bucket *bucket*, *error_t (*fun) (void *port)*) [Function]

Call *fun* once for each port in *bucket*. No guarantee is made about the order of iteration, which might vary from call to call. If FUN returns an error, then no further calls to FUN are made for any remaining ports, and the return value of FUN is returned from ports_bucket_iterate.

3.2.4 Port References

These functions maintain references to ports so that the port information structures may be freed if and only if they are no longer needed. It is your responsibility to tell `libports` when references to ports change.

`void ports_port_ref` (*void* *`port`) [Function]
: Allocate a hard reference to *port*.

`void ports_port_deref` (*void* *`port`) [Function]
: Drop a hard reference to *port*.

`void ports_no_senders` (*void* *`port`, *mach_port_mscount_t* `mscount`) [Function]
: The user is responsible for listening for no senders notifications; when one arrives, call this routine for the *port* the message was sent to, providing the *mscount* from the notification.

`int ports_count_class` (*struct port_class* *`class`) [Function]
: Block creation of new ports in *class*. Return the number of ports currently in *class*.

`int ports_count_bucket` (*struct port_bucket* *`bucket`) [Function]
: Block creation of new ports in *bucket*. Return the number of ports currently in *bucket*.

`void ports_enable_class` (*struct port_class* *`class`) [Function]
: Permit suspended port creation (blocked by `ports_count_class`) to continue.

`void ports_enable_bucket` (*struct port_bucket* *`bucket`) [Function]
: Permit suspended port creation (blocked by `ports_count_bucket`) to continue.

Weak references are not often used, as they are the same as hard references for port classes where *dropweak_routine* is null. See Section 3.2.1 [Buckets and Classes], page 15.

`void ports_port_ref_weak` (*void* *`port`) [Function]
: Allocate a weak reference to *port*.

`void ports_port_deref_weak` (*void* *`port`) [Function]
: Drop a weak reference to *port*.

3.2.5 RPC Management

The rest of the `libports` functions are dedicated to controlling RPC operations. These functions help you do all the locking and thread cancellations that are required in order to build robust servers.

`typedef int` (**ports_demuxer_type*) (*mach_msg_header_t* *`inp`, *mach_msg_header_t* *`outp`) [Typedef]
: Type of MiG demuxer routines.

`error_t ports_begin_rpc` (*void* *`port`, *mach_msg_id_t* `msg_id`, *struct rpc_info* *`info`) [Function]
: Call this when an RPC is beginning on *port*. *info* should be allocated by the caller and will be used to hold dynamic state. If this RPC should be abandoned, return `EDIED`; otherwise we return zero.

`void ports_end_rpc` (*void *`port`*, *struct rpc_info *`info`*) [Function]
Call this when an RPC is concluding. The arguments must match the ones passed to the paired call to `ports_begin_rpc`.

`void ports_manage_port_operations_one_thread` (*struct port_bucket *`bucket`*, *ports_demuxer_type `demuxer`*, *int `timeout`*) [Function]
Begin handling operations for the ports in *bucket*, calling *demuxer* for each incoming message. Return if *timeout* is nonzero and no messages have been received for *timeout* milliseconds. Use only one thread (the calling thread).

`void ports_manage_port_operations_multithread` (*struct port_bucket *`bucket`*, *ports_demuxer_type `demuxer`*, *int `thread_timeout`*, *int `global_timeout`*, *void* (**`hook`*) (*void*)) [Function]
Begin handling operations for the ports in *bucket*, calling *demuxer* for each incoming message. Return if *global_timeout* is nonzero and no messages have been received for *global_timeout* milliseconds. Create threads as necessary to handle incoming messages so that no port is starved because of sluggishness on another port. If *thread_timeout* is nonzero, then individual threads will die off if they handle no incoming messages for *local_timeout* milliseconds. If non-null, *hook* will be called in each new thread immediately after it is created.

`error_t ports_inhibit_port_rpcs` (*void *`port`*) [Function]
Interrupt any pending RPC on *port*. Wait for all pending RPCs to finish, and then block any new RPCs starting on that port.

`error_t ports_inhibit_class_rpcs` (*struct port_class *`class`*) [Function]
Similar to `ports_inhibit_port_rpcs`, but affects all ports in *class*.

`error_t ports_inhibit_bucket_rpcs` (*struct port_bucket *`bucket`*) [Function]
Similar to `ports_inhibit_port_rpcs`, but affects all ports in *bucket*.

`error_t ports_inhibit_all_rpcs` (*void*) [Function]
Similar to `ports_inhibit_port_rpcs`, but affects all ports whatsoever.

`void ports_resume_port_rpcs` (*void *`port`*) [Function]
Reverse the effect of a previous `ports_inhibit_port_rpcs` for this *port*, allowing blocked RPCs to continue.

`void ports_resume_class_rpcs` (*struct port_class *`class`*) [Function]
Reverse the effect of a previous `ports_inhibit_class_rpcs` for *class*.

`void ports_resume_bucket_rpcs` (*struct port_bucket *`bucket`*) [Function]
Reverse the effect of a previous `ports_inhibit_bucket_rpcs` for *bucket*.

`void ports_resume_all_rpcs` (*void*) [Function]
Reverse the effect of a previous `ports_inhibit_all_rpcs`.

`void ports_interrupt_rpcs` (*void *`port`*) [Function]
Cancel (with `thread_cancel`) any RPCs in progress on *port*.

`int ports_self_interrupted (void)` [Function]
: If the current thread's RPC has been interrupted with `ports_interrupt_rpcs`, return nonzero and clear the interrupted flag.

`error_t ports_interrupt_rpc_on_notification (void *object, struct rpc_info *rpc, mach_port_t port, mach_msg_id_t what)` [Function]
: Arrange for `hurd_cancel` to be called on *rpc*'s thread if *object* gets notified that any of the things in *what* have happened to *port*. *rpc* should be an RPC on *object*.

`error_t ports_interrupt_self_on_notification (void *object, mach_port_t port, mach_msg_id_t what)` [Function]
: Arrange for `hurd_cancel` to be called on the current thread, which should be an RPC on *object*, if *port* gets notified with the condition *what*.

`error_t ports_interrupt_self_on_port_death (void *object, mach_port_t port)` [Function]
: Same as calling `ports_interrupt_self_on_notification` with *what* set to `MACH_NOTIFY_DEAD_NAME`.

`void ports_interrupt_notified_rpcs (void *object, mach_port_t port, mach_msg_id_t what)` [Function]
: Interrupt any RPCs on *object* that have requested such.

`void ports_dead_name (void *object, mach_port_t port)` [Function]
: Same as calling `ports_interrupt_notified_rpcs` with *what* set to `MACH_NOTIFY_DEAD_NAME`.

3.3 Integer Hash Library

`libihash` provides integer-keyed hash tables, for arbitrary element data types. Such hash tables are frequently used when implementing sparse arrays or buffer caches.

The following functions are declared in `<hurd/ihash.h>`:

`error_t ihash_create (ihash_t *ht)` [Function]
: Create an integer hash table and return it in *ht*. If a memory allocation error occurs, `ENOMEM` is returned, otherwise zero.

`void ihash_free (ihash_t ht)` [Function]
: Free *ht* and all resources it consumes.

`void ihash_set_cleanup (ihash_t ht, void (*cleanup) (void *value, void *arg), void *arg)` [Function]
: Sets *ht*'s element cleanup function to *cleanup*, and its second argument to *arg*. *cleanup* will be called on every element *value* to be subsequently overwritten or deleted, with *arg* as the second argument.

`error_t ihash_add (ihash_t ht, int id, void *item, void ***locp)` [Function]
: Add *item* to the hash table *ht* under the integer key *id*. *locp* is the address of a pointer located in *item*; If non-null, *locp* should point to a variable of type `void **`, and will be filled with a pointer that may be used as an argument to `ihash_locp_remove`.

The variable pointed to by *locp* may be overwritten sometime between this call and when the element is deleted, so you cannot stash its value elsewhere and hope to use the stashed value with `ihash_locp_remove`. If a memory allocation error occurs, `ENOMEM` is returned, otherwise zero.

`void * ihash_find` (*ihash_t* **`ht`**, *int* **`id`**) [Function]

Find and return the item in hash table *ht* with key *id*. Returns null if the specified item doesn't exist.

`error_t ihash_iterate` (*ihash_t* **`ht`**, *error_t* (*****`fun`**) (*void ****`value`**)) [Function]

Call function *fun* on every element of *ht*. *fun*'s only arg, *value*, is a pointer to the value stored in the hash table. If *fun* ever returns nonzero, then iteration stops and `ihash_iterate` returns that value, otherwise it (eventually) returns 0.

`int ihash_remove` (*ihash_t* **`ht`**, *int* **`id`**) [Function]

Remove the entry with a key of *id* from *ht*. If there was no such element, then return zero, otherwise nonzero.

`void ihash_locp_remove` (*ihash_t* **`ht`**, *void *****`ht_locp`**) [Function]

Remove the entry at *locp* from the hashtable *ht*. *locp* is as returned from an earlier call to `ihash_add`. This call should be faster than `ihash_remove`. *ht* can be null, in which case the call still succeeds, but no cleanup is done.

3.4 Misc Library

The GNU C library is constantly developing to meet the needs of the Hurd. However, because the C library needs to be very stable, it is irresponsible to add new functions to it without carefully specifying their interface, and testing them thoroughly.

The Hurd distribution includes a library called `libshouldbeinlibc`, which serves as a proving ground for additions to the GNU C library. This library is in flux, as some functions are added to it by the Hurd developers and others are moved to the official C library.

These functions aren't currently documented (other than in their header files), but complete documentation will be added to *The GNU C Library Reference Manual* when these functions become part of the GNU C library.

3.5 Bug Address Library

`libhurdbugaddr` exists only to define a single variable:

`char * argp_program_bug_address` [Variable]

`argp_program_bug_address` is the default Hurd bug-reporting e-mail address, `bug-hurd@gnu.org`. This address is displayed to the user when any of the standard Hurd servers and utilities are invoked using the '`--help`' option.

4 Input and Output

There are no specific programs or servers associated with the I/O subsystem, since it is used to interact with almost all servers in the GNU Hurd. It provides facilities for reading and writing I/O channels, which are the underlying implementation of file and socket descriptors in the GNU C library.

4.1 Iohelp Library

The `<hurd/iohelp.h>` file declares several functions which are useful for low-level I/O implementations. Most Hurd servers do not call these functions directly, but they are used by several of the Hurd filesystem and networking helper libraries. `libiohelp` requires `libthreads`.

4.1.1 I/O Users

Most I/O servers need to implement some kind of user authentication checking. In order to facilitate that process, `libiohelp` has some functions which encapsulate a set of idvecs (FIXME: xref to C library) in a single `struct iouser`.

`struct iouser * iohelp_create_iouser` (*struct idvec *`uids`*, *struct idvec *`gids`*) [Function]

Create a new *iouser* for the specified *uids* and *gids*.

`struct iouser * iohelp_dup_iouser` (*struct iouser *`iouser`*) [Function]

Return a copy of *iouser*.

`void iohelp_free_iouser` (*struct iouser *`iouser`*) [Function]

Release a reference to *iouser*.

I/O reauthentication is a rather complex protocol involving the authserver as a trusted third party (see Section 14.5.1 [Auth Protocol], page 74). In order to reduce the risk of flawed implementations, I/O reauthentication is encapsulated in the `iohelp_reauth` function:

`struct iouser * iohelp_reauth` (*auth_t `authserver`*, *mach_port_t `rend_port`*, *mach_port_t `newright`*, *int `permit_failure`*) [Function]

Conduct a reauthentication transaction, and return a new *iouser*. *authserver* is the I/O server's auth port. The rendezvous port provided by the user is *rend_port*.

If the transaction cannot be completed, return zero, unless *permit_failure* is nonzero. If *permit_failure* is nonzero, then should the transaction fail, return an *iouser* that has no ids. The new port to be sent to the user is *newright*.

4.1.2 Conch Management

The *conch* is at the heart of the shared memory I/O system. Several Hurd libraries implement shared I/O, and so `libiohelp` contains functions to facilitate conch management.

Everything about shared I/O is undocumented because it is not needed for adequate performance, and the RPC interface is simpler (see Section 4.3 [I/O Interface], page 25). It is not useful for new libraries or servers to implement shared I/O.

4.2 Pager Library

The *external pager* (*XP*) microkernel interface allows applications to provide the backing store for a memory object, by converting hardware page faults into RPC requests. External pagers are required for memory-mapped I/O (see Section 4.3.6 [Mapped Data], page 28) and stored filesystems (see Chapter 8 [Stored Filesystems], page 55).

The external pager interface is quite complex, so the Hurd pager library contains functions which aid in creating multithreaded external pagers. `libpager` is declared in `<hurd/pager.h>`, and requires only the threads and ports libraries.

4.2.1 Pager Management

The pager library defines the `struct pager` data type in order to represent a multi-threaded pager. The general procedure for creating a pager is to define the functions listed in Section 4.2.2 [Pager Callbacks], page 24, allocate a `libports` bucket for the ports which will access the pager, and create at least one new `struct pager` with `pager_create`.

`struct pager * pager_create` (*struct user_pager_info* *`u_pager`, *struct port_bucket* *`bucket`, *boolean_t* `may_cache`, *memory_object_copy_strategy_t* `copy_strategy`) [Function]

Create a new pager. The pager will have a port created for it (using `libports`, in *bucket*) and will be immediately ready to receive requests. *u_pager* will be provided to later calls to `pager_find_address`. The pager will have one user reference created. *may_cache* and *copy_strategy* are the original values of those attributes as for `memory_object_ready`. Users may create references to pagers by use of the relevant ports library functions. On errors, return null and set `errno`.

Once you are ready to turn over control to the pager library, you should call `ports_manage_port_operations_multithread` on the *bucket*, using `pager_demuxer` as the ports *demuxer*. This will handle all external pager RPCs, invoking your pager callbacks when necessary.

`int pager_demuxer` (*mach_msg_header_t* *`inp`, *mach_msg_header_t* *`outp`) [Function]

Demultiplex incoming `libports` messages on pager ports.

The following functions are the body of the pager library, and provide a clean interface to pager functionality:

`void pager_sync` (*struct pager* *`pager`, *int* `wait`) [Function]
`void pager_sync_some` (*struct pager* *`pager`, *vm_address_t* `start`, *vm_size_t* `len`, *int* `wait`) [Function]

Write data from pager *pager* to its backing store. Wait for all the writes to complete if and only if *wait* is set.

`pager_sync` writes all data; `pager_sync_some` only writes data starting at *start*, for *len* bytes.

`void pager_flush` (*struct pager* *`pager`, *int* `wait`) [Function] void **pager_flush_some** (*struct pager *pager, vm_address_t start, vm_size_t len, int wait*) [Function]

Flush data from the kernel for pager *pager* and force any pending delayed copies. Wait for all pages to be flushed if and only if *wait* is set.

`pager_flush` flushes all data; `pager_flush_some` only flushes data starting at *start*, for *len* bytes.

void **pager_return** (*struct pager *pager, int wait*) [Function]
void **pager_return_some** (*struct pager *pager, vm_address_t start, vm_size_t len, int wait*) [Function]

Flush data from the kernel for pager *pager* and force any pending delayed copies. Wait for all pages to be flushed if and only if *wait* is set. Have the kernel write back modifications.

`pager_return` flushes and restores all data; `pager_return_some` only flushes and restores data starting at *start*, for *len* bytes.

void **pager_offer_page** (*struct pager *pager, int precious, int writelock, vm_offset_t page, vm_address_t buf*) [Function]

Offer a page of data to the kernel. If *precious* is set, then this page will be paged out at some future point, otherwise it might be dropped by the kernel. If the page is currently in core, the kernel might ignore this call.

attributes

void **pager_change_attributes** (*struct pager *pager, boolean_t may_cache, memory_object_copy_strategy_t copy_strategy, int wait*) [Function]

Change the attributes of the memory object underlying pager *pager*. The *may_cache* and *copy_strategy* arguments are as for `memory_object_change_`. Wait for the kernel to report completion if and only if *wait* is set.

void **pager_shutdown** (*struct pager *pager*) [Function]

Force termination of a pager. After this returns, no more paging requests on the pager will be honoured, and the pager will be deallocated. The actual deallocation might occur asynchronously if there are currently outstanding paging requests that will complete first.

error_t **pager_get_error** (*struct pager *p, vm_address_t addr*) [Function]

Return the error code of the last page error for pager *p* at address *addr*.[1]

error_t **pager_memcpy** (*struct pager *pager, memory_object_t memobj, vm_offset_t offset, void *other, size_t *size, vm_prot_t prot*) [Function]

Try to copy `*size` bytes between the region *other* points to and the region at *offset* in the pager indicated by *pager* and *memobj*. If *prot* is `VM_PROT_READ`, copying is from the pager to *other*; if *prot* contains `VM_PROT_WRITE`, copying is from *other* into the pager. `*size` is always filled in with the actual number of bytes successfully copied.

[1] Note that this function will be deleted when the Mach pager interface is fixed to provide this information.

Returns an error code if the pager-backed memory faults; if there is no fault, returns zero and `*size` will be unchanged.

These functions allow you to recover the internal `struct pager` state, in case the `libpager` interface doesn't provide an operation you need:

`struct user_pager_info * pager_get_upi` (*struct pager* *`p`*) [Function]

Return the `struct user_pager_info` associated with a pager.

`mach_port_t pager_get_port` (*struct pager* *`pager`*) [Function]

Return the port (receive right) for requests to the pager. It is absolutely necessary that a new send right be created from this receive right.

4.2.2 Pager Callbacks

Like several other Hurd libraries, `libpager` depends on you to implement application-specific callback functions. You *must* define the following functions:

`error_t pager_read_page` (*struct user_pager_info* *`pager`*, *vm_offset_t* `page`, *vm_address_t* *`buf`*, *int* *`write_lock`*) [Function]

For pager *pager*, read one page from offset *page*. Set `*buf` to be the address of the page, and set `*write_lock` if the page must be provided read-only. The only permissible error returns are `EIO`, `EDQUOT`, and `ENOSPC`.

`error_t pager_write_page` (*struct user_pager_info* *`pager`*, *vm_offset_t* `page`, *vm_address_t* `buf`) [Function]

For pager *pager*, synchronously write one page from *buf* to offset *page*. In addition, `vm_deallocate` (or equivalent) *buf*. The only permissible error returns are `EIO`, `EDQUOT`, and `ENOSPC`.

`error_t pager_unlock_page` (*struct user_pager_info* *`pager`*, *vm_offset_t* `address`) [Function]

A page should be made writable.

`error_t pager_report_extent` (*struct user_pager_info* *`pager`*, *vm_address_t* *`offset`*, *vm_size_t* *`size`*) [Function]

This function should report in `*offset` and `*size` the minimum valid address the pager will accept and the size of the object.

`void pager_clear_user_data` (*struct user_pager_info* *`pager`*) [Function]

This is called when a pager is being deallocated after all extant send rights have been destroyed.

`void pager_dropweak` (*struct user_pager_info* *`p`*) [Function]

This will be called when the ports library wants to drop weak references. The pager library creates no weak references itself, so if the user doesn't either, then it is all right for this function to do nothing.

4.3 I/O Interface

The I/O interface facilities are described in `<hurd/io.defs>`. This section discusses only RPC-based I/O operations.[2]

4.3.1 I/O Object Ports

The I/O server must associate each I/O port with a particular set of uids and gids, identifying the user who is responsible for operations on the port. Every port to an I/O server should also support either the file protocol (see Section 5.4 [File Interface], page 40) or the socket protocol (see Section 11.4 [Socket Interface], page 71); naked I/O ports are not allowed.

In addition, the server associates with each port a default file pointer, a set of open mode bits, a pid (called the "owner"), and some underlying object which can absorb data (for write) or provide data (for read).

The uid and gid sets associated with a port may not be visibly shared with other ports, nor may they ever change. The server must fix the identification of a set of uids and gids with a particular port at the moment of the port's creation. The other characteristics of an I/O port may be shared with other users. The I/O server interface does not generally specify the way in which servers may share these other characteristics (with the exception of the deprecated `O_ASYNC` interface); however, the file and socket interfaces make further requirements about what sharing is required and what sharing is prohibited.

In general, users get send rights to I/O ports by some mechanism that is external to the I/O protocol. (For example, fileservers give out I/O ports in response to the `dir_lookup` and `fsys_getroot` calls. Socket servers give out ports in response to the `socket_create` and `socket_accept` calls.) However, the I/O protocol provides methods of obtaining new ports that refer to the same underlying object as another port. In response to all of these calls, all underlying state (including, but not limited to, the default file pointer, open mode bits, and underlying object) must be shared between the old and new ports. In the following descriptions of these calls, the term "identical" means this kind of sharing. All these calls must return send rights to a newly-constructed Mach port.

The `io_duplicate` call simply returns another port which is identical to an existing port and has the same uid and gid set.

The `io_restrict_auth` call returns another port, identical to the provided port, but which has a smaller associated uid and gid set. The uid and gid sets of the new port are the intersection of the set on the existing port and the lists of uids and gids provided in the call.

Users use the `io_reauthenticate` call when they wish to have an entirely new set of uids or gids associated with a port. In response to the `io_reauthenticate` call, the server must create a new port, and then make the call `auth_server_authenticate` to the auth server. The rendezvous port for the `auth_server_authenticate` call is the I/O port to which was made the `io_reauthenticate` call. The server provides the *rend_int* parameter to the auth server as a copy from the corresponding parameter in the `io_reauthenticate`

[2] The latter portion of `<hurd/io.defs>` and all of `<hurd/shared.h>` describe how to implement shared-memory I/O operations. However, shared I/O has been deprecated. See Section 4.1.2 [Conch Management], page 21, for more details.

call. The I/O server also gives the auth server a new port; this must be a newly created port identical to the old port. The authserver will return the set of uids and gids associated with the user, and guarantees that the new port will go directly to the user that possessed the associated authentication port. The server then identifies the new port given out with the specified ID's.

4.3.2 Simple Operations

Users write to I/O ports by calling the `io_write` RPC. They specify an *offset* parameter; if the object supports writing at arbitrary offsets, the server should honour this parameter. If -1 is passed as the offset, then the server should use the default file pointer. The server should return the amount of data which was successfully written. If the operation was interrupted after some but not all of the data was written, then it is considered to have succeeded and the server should return the amount written. If the port is not an I/O port at all, the server should reply with the error `EOPNOTSUPP`. If the port is an I/O port, but does not happen to support writing, then the correct error is `EBADF`.

Users read from I/O ports by calling the `io_read` RPC. They specify the amount of data they wish to read, and the offset. The offset has the same meaning as for `io_write` above. The server should return the data that was read. If the call is interrupted after some data has been read (and the operation is not idempotent) then the server should return the amount read, even if it was less than the amount requested. The server should return as much data as possible, but never more than requested by the user. If there is no data, but there might be later, the call should block until data becomes available. The server indicates end-of-file by returning zero bytes. If the call is interrupted after some data has been read, but the call is idempotent, then the server may return `EINTR` rather than actually filling the buffer (taking care that any modifications of the default file pointer have been reversed). Preferably, however, servers should return data.

There are two categories of objects: seekable and non-seekable. Seekable objects must accept arbitrary offset parameters in the `io_read` and `io_write` calls, and must implement the `io_seek` call. Non-seekable objects must ignore the offset parameters to `io_read` and `io_write`, and should return `ESPIPE` to the `io_seek` call.

On seekable objects, `io_seek` changes the default file pointer for reads and writes. (See section "File Positioning" in *The GNU C Library Reference Manual*, for the interpretation of the *whence* and *offset* arguments.) It returns the new offset as modified by `io_seek`.

The `io_readable` interface returns the amount of data which can be immediately read. For the special technical meaning of "immediately", see Section 4.3.4 [Asynchronous I/O], page 27.

4.3.3 Open Modes

The server associates each port with a set of bits that affect its operation. The `io_set_all_openmodes` call modifies these bits and the `io_get_openmodes` call returns them. In addition, the `io_set_some_openmodes` and `io_clear_some_openmodes` do an atomic read/modify/write of the openmodes.

The `O_APPEND` bit, when set, changes the behaviour of `io_write` when it uses the default file pointer on seekable objects. When `io_write` is done on a port with the `O_APPEND` bit set, is must set the file pointer to the current file size before doing the write (which would then increment the file pointer as usual). The *current file size* is the smallest offset which returns end-of-file when provided to `io_read`. The server must atomically bind this update to the actual data write with respect to other users of `io_read`, `io_write`, and `io_seek`.

The `O_FSYNC` bit, when set, guarantees that `io_write` will not return until data is fully written to the underlying medium.

The `O_NONBLOCK` bit, when set, prevents read and write from blocking. They should copy such data as is immediately available. If no data is immediately available they should return `EWOULDBLOCK`.

The definition of "immediately" is more or less server-dependent. Some servers, notably stored filesystem servers (see Chapter 8 [Stored Filesystems], page 55), regard all data as immediately available. The one criterion is that something which must happen *immediately* may not wait for any user-synchronizable event.

The `O_ASYNC` bit is deprecated; its use is documented in the following section. This bit must be shared between all users of the same underlying object.

4.3.4 Asynchronous I/O

Users may wish to be notified when I/O can be done without blocking; they use the `io_async` call to indicate this to the server. In the `io_async` call the user provides a port on which will the server should send `sig_post` messages as I/O becomes possible. The server must return a port which will be the reference port in the `sig_post` messages. Each `io_async` call should generate a new reference port. (FIXME: xref the C library manual for information on how to send sig_post messages.)

The server then sends one `SIGIO` signal to each registered async user everytime I/O becomes possible. I/O is possible if at least one byte can be read or written immediately. The definition of "immediately" must be the same as for the implementation of the `O_NONBLOCK` flag (see Section 4.3.3 [Open Modes], page 26). In addition, every time a user calls `io_read` or `io_write` on a non-seekable object, or at the default file pointer on a seekable object, another signal should be sent to each user if I/O is still possible.

Some objects may also define "urgent" conditions. Such servers should send the `SIGURG` signal to each registered async user anytime an urgent condition appears. After any RPC that has the possibility of clearing the urgent condition, the server should again send the signal to all registered users if the urgent condition is still present.

A more fine-grained mechanism for doing async I/O is the `io_select` call. The user specifies the kind of access desired, and a send-once right. If I/O of the kind the user desires is immediately possible, then the server should return so indicating, and destroy the send-once right. If I/O is not immediately possible, the server should save the send-once right, and send a `select_done` message as soon as I/O becomes immediately possible. Again, the definition of "immediately" must be the same for `io_select`, `io_async`, and `O_NONBLOCK` (see Section 4.3.3 [Open Modes], page 26).

For compatibility with 4.2 and 4.3 BSD, the I/O interface provides a deprecated feature (known as *icky async I/O*). The calls `io_mod_owner` and `io_get_owner` set the "owner" of the object, providing either a pid or a pgrp (if the value is negative). This implies that only one process at a time can do icky I/O on a given object. Whenever the I/O server is sending `sig_post` messages to all the `io_async` users, if the `O_ASYNC` bit is set, the server should also send a signal to the owning pid/pgrp. The ID port for this call should be different from all the `io_async` ID ports given to users. Users may find out what ID port the server uses for this by calling `io_get_icky_async_id`.

4.3.5 Information Queries

Users may call `io_stat` to find out information about the I/O object. Most of the fields of a `struct stat` are meaningful only for files. All objects, however, must support the fields *st_fstype*, *st_fsid*, *st_ino*, *st_atime*, *st_atime_usec*, *st_mtime_user*, *st_ctime*, *st_ctime_usec*, and *st_blksize*.

st_fstype, *st_fsid*, and *st_ino* must be unique for the underlying object across the entire system.

st_atime and *st_atime_usec* hold the seconds and microseconds, respectively, of the system clock at the last time the object was read with `io_read`.

st_mtime and *st_mtime_usec* hold the seconds and microseconds, respectively, of the system clock at the last time the object was written with `io_write`.

Other appropriate operations may update the *atime* and the *mtime* as well; both the file and socket interfaces specify such operations.

st_ctime and *st_ctime_usec* hold the seconds and microseconds, respectively, of the system clock at the last time permanent meta-data associated with the object was changed. The exact operations which cause such an update are server-dependent, but must include the creation of the object.

The server is permitted to delay the actual update of these times until stat is called; before the server stores the times on permanent media (if it ever does so) it should update them if necessary.

st_blksize gives the optimal I/O size in bytes for `io_read` and `io_write`; users should endeavor to read and write amounts which are multiples of the optimal size, and to use offsets which are multiples of the optimal size.

In addition, objects which are seekable should set *st_size* to the current file size as in the description of the `O_APPEND` flag (see Section 4.3.3 [Open Modes], page 26).

The *st_uid* and *st_gid* fields are unrelated to the "owner" as described above for icky async I/O.

Users may find out the version of the server they are talking to by calling `io_server_version`; this should return strings and integers describing the version number of the server, as well as its name.

4.3.6 Mapped Data

Servers may optionally implement the `io_map` call. The ports returned by `io_map` must implement the external pager kernel interface (see Section 4.2 [Pager Library], page 22) and be suitable as arguments to `vm_map`.

Seekable objects must allow access from zero up to (but not including) the current file size as described for `O_APPEND` (see Section 4.3.3 [Open Modes], page 26). Whether they provide access beyond such a point is server-dependent; in addition, the meaning of accessing a non-seekable object is server-dependent.

5 Files

A file is traditionally thought of as a quantity of disk storage. In the Hurd, files are an extension of the I/O interface, but they do not necessarily correspond to disk storage.

Every file in the Hurd is represented by a port, which is connected to the server that manages the file. When a client wants to operate on a file, it makes RPC requests via a file port to its server process, which is commonly called a *translator*.

5.1 Translators

The Hurd filesystem allows you to set translators on any file or directory that you own. A *translator* is any Hurd server which provides the basic filesystem interface. Translated nodes are somewhat like a cross between Unix symbolic links and mount points.

Whenever a program tries to access the contents of a translated node, the filesystem server redirects the request to the appropriate translator (starting it if necessary). Then, the new translator services the client's request. The GNU C library makes this behaviour seamless from the client's perspective, so that standard Unix programs behave correctly under the Hurd.

Translators run with the privileges of the translated node's *owner*, so they cannot be used to compromise the security of the system. This also means that *any* user can write their own translators, and provide other users with arbitrary filesystem-structured data, regardless of the data's actual source. Other chapters in this manual describe existing translators, and how you can modify them or write your own.

The standard Hurd filesystem servers are constantly evolving to provide innovative features that users want. Here are a few examples of existing translators:

- Disk-based filesystem formats, such as `ext2fs`, `ufs`, and `iso9660fs` (see Chapter 8 [Stored Filesystems], page 55).
- Network filesystems, such as `nfs` and `ftpfs` (see Chapter 10 [Distributed Filesystems], page 70).
- Single files with dynamic content, such as FIXME: we need a good example.
- Hurd servers which translate rendezvous filesystem nodes in standard locations, so that other programs can easily find them and use server-specific interfaces. For example, `pflocal` implements the filesystem interfaces, but it also provides a special Unix-domain socket RPC interface (FIXME xref). Programs can fetch a port to this translator simply by calling `file_name_lookup` (FIXME xref) on '`/servers/socket/1`'[1], then use Unix socket-specific RPCs on that port, rather than adhering to the file protocol.

This section focuses on the generic programs that you need to understand in order to use existing translators. Many other parts of this manual describe how you can write your own translators.

5.1.1 Invoking `settrans`

The `settrans` program allows you to set a translator on a file or directory. By default, the passive translator is set (see the '`--passive`' option).

The `settrans` program has the following synopsis:

---

[1] The number 1 corresponds to the `PF_LOCAL` C library socket domain constant.

```
settrans [option]... node [translator arg...]
```

where *translator* is the absolute filename of the new translator program. Each *arg* is passed to *translator* when it starts. If *translator* is not specified, then `settrans` clears the existing translator rather than setting a new one.

`settrans` accepts the following options:

'`-a`'
'`--active`'
: Set *node*'s active translator. *Active translators* are started immediately and are not persistent: if the system is rebooted then they are lost.

'`-c`'
'`--create`'
: Create *node* as a zero-length file if it doesn't already exist.

'`-L`'
'`--dereference`'
: If *node* is already translated, stack the new translator on top of it (rather than replacing the existing translator).

'`--help`' Display a brief usage message, then exit.

'`-p`'
'`--passive`'
: Set *node*'s passive translator. *Passive translators* are only activated by the underlying filesystem when clients try to use the *node*, and they shut down automatically after they are no longer active in order to conserve system resources.

Passive translators are stored on the underlying filesystem media, and so they persist between system reboots. Not all filesystems support passive translators, due to limitations in their underlying media. Consult the filesystem-specific documentation to see if they are supported.

If you are setting the passive translator, and *node* already has an active translator, then the following options apply:

'`-g`'
'`--goaway`'
: Tell the active translator to go away. In this case, the following additional options apply:

'`-f`'
'`--force`' If the active translator doesn't go away, then force it.

'`-S`'
'`--nosync`'
: Don't flush its contents to disk before terminating.

'`-R`'
'`--recursive`'
: Shut down all of the active translator's children, too.

'-k'
'--keep-active'
: Leave the existing active translator running. The new translator will not be started unless the active translator has stopped.

'-P'
'--pause'
: When starting an active translator, prompt and wait for a newline on standard input before completing the startup handshake. This is useful when debugging a translator, as it gives you time to start the debugger.

'-t *sec*'
'--timeout=*sec*'
: If the translator does not start up in *sec* seconds (the default is 60), then return an error; if *sec* is 0, then never timeout.

'--version'
: Output program version information and exit.

'-x'
'--exclusive'
: Only set the translator if there is none already.

5.1.2 Invoking `showtrans`

The `showtrans` program allows you to show the passive translator setting on a file system node.

The `showtrans` program has the following synopsis:

```
showtrans [option]... file...
```

`showtrans` accepts the following options:

-p
--prefix
: Always display *filename*: before translators.

-P
--no-prefix
: Never display *filename*: before translators.

-s
--silent
: No output; useful when checking error status.

-t
--translated
: Only display files that have translators.

5.1.3 Invoking `mount`

5.1.4 Invoking `fsysopts`

The `fsysopts` program allows you to retrieve or set command line options for running translator *filesys*.

The `fsysopts` program has the following synopsis:

```
fsysopts [option...] filesys [fs_option...]
```

`fsysopts` accepts the following options:

`-L`
`--dereference`
If *filesys* is a symbolic link, follow it.

`-R`
`--recursive`
Pass these options to any child translators.

The legal values for *fs_option* depends on *filesys*, but some common ones are:

`--readonly`
`--writable`
`--remount`
`--sync[=interval]`
`--nosync`

If no options are supplied, *filesys*' current options are printed.

The options passed as *fs_option*s are meant to augment or change those which are already set, they're not meant to completely replace the existing command line options. For example, passing `--readonly` to a file system server will change the server from writable to read-only, but will not touch the used backing store. Passing `--address=new_IP` to '`/servers/socket/2`' will change the local IP address to *new_IP*, but will not touch the interface, netmask and gateway settings.

5.2 Trivfs Library

Certain translators do not need to be very complex, because they represent a single file rather than an entire directory hierarchy. The trivfs library, which is declared in `<hurd/trivfs.h>`, does most of the work of implementing this kind of translator. This library requires the iohelp and ports libraries.

5.2.1 Trivfs Startup

In order to use the trivfs library, you will need to define the appropriate callbacks (see Section 5.2.2 [Trivfs Callbacks], page 34). As with all Hurd servers, your trivfs-based translator should first parse any command-line options, in case the user is just asking for help. Trivfs uses argp (see section "Argp" in *The GNU C Library Reference Manual*) for parsing command-line arguments.

Your translator should redefine the following functions and variables as necessary, and then call `argp_parse` with the relevant arguments:

`extern struct argp * trivfs_runtime_argp` [Variable]

If this is defined or set to an argp structure, it will be used by the default `trivfs_set_options` to handle runtime options parsing. Redefining this is the normal way to add option parsing to a trivfs program.

error_t **trivfs_set_options** (*struct trivfs_control *fsys, char *argz, size_t argz_len*) [Function]

Set runtime options for *fsys* to *argz* and *argz_len*. The default definition for this routine simply uses *trivfs_runtime_argp* (supplying *fsys* as the argp input field).

error_t **trivfs_append_args** (*struct trivfs_control *fsys, char **argz, size_t *argz_len*) [Function]

Append to the malloced string `*argz` of length `*argz_len` a NUL-separated list of the arguments to this translator.

After your translator parses its command-line arguments, it should fetch its bootstrap port by using `task_get_bootstrap_port`. If this port is `MACH_PORT_NULL`, then your program wasn't started as a translator. Otherwise, you can use the bootstrap port to create a new control structure (and advertise its port) with `trivfs_startup`:

error_t **trivfs_startup** (*mach_port_t bootstrap, int flags, struct port_class *control_class, struct port_bucket *control_bucket, struct port_class *protid_class, struct port_bucket *protid_bucket, struct trivfs_control **control*) [Function]

error_t **trivfs_create_control** (*mach_port_t bootstrap, struct port_class *control_class, struct port_bucket *control_bucket, struct port_class *protid_class, struct port_bucket *protid_bucket, struct trivfs_control **control*) [Function]

`trivfs_startup` creates a new trivfs control port, advertises it to the underlying node *bootstrap* with `fsys_startup`, returning the results of this call, and places its control structure in `*control`. `trivfs_create_control` does the same thing, except it doesn't advertise the control port to the underlying node. *control_class* and *control_bucket* are passed to `libports` to create the control port, and *protid_class* and *protid_bucket* are used when creating ports representing opens of this node; any of these may be zero, in which case an appropriate port class/bucket is created. If *control* is non-null, the trivfs control port is returned in it. *flags* (a bitmask of the appropriate `O_*` constants) specifies how to open the underlying node.

If you did not supply zeros as the class and bucket arguments to `trivfs_startup`, you will probably need to use the trivfs port management functions (see Section 5.2.4 [Trivfs Ports], page 36).

Once you have successfully called `trivfs_startup`, and have a pointer to the control structure stored in, say, the *fsys* variable, you are ready to call one of the `ports_manage_port_operations_*` functions using `fsys->pi.bucket` and `trivfs_demuxer`. This will handle any incoming filesystem requests, invoking your callbacks when necessary.

int **trivfs_demuxer** (*mach_msg_header_t *inp, mach_msg_header_t *outp*) [Function]

Demultiplex incoming `libports` messages on trivfs ports.

The following functions are not usually necessary, but they allow you to use the trivfs library even when it is not possible to turn message-handling over to `trivfs_demuxer` and `libports`:

`struct trivfs_control * trivfs_begin_using_control` (*mach_port_t* `port`) [Function]
`struct trivfs_protid * trivfs_begin_using_protid` (*mach_port_t* `port`) [Function]

These functions can be used as `intran` functions for a MiG port type to have the stubs called with either the control or protid pointer.

`void trivfs_end_using_control` (*struct trivfs_control* *`port`) [Function]
`void trivfs_end_using_protid` (*struct trivfs_protid* *`port`) [Function]

These can be used as 'destructor' functions for a MiG port type, to have the stubs called with the control or protid pointer.

`error_t trivfs_open` (*struct trivfs_control* *`fsys`, *struct iouser* *`user`, *unsigned* `flags`, *mach_port_t* `realnode`, *struct trivfs_protid* **`cred`) [Function]

Return a new protid (that is, a port representing an open of this node) pointing to a new peropen in *cred*, with *realnode* as the underlying node reference, with the given identity, and open flags in *flags*. *cntl* is the trivfs control object.

`error_t trivfs_protid_dup` (*struct trivfs_protid* *`cred`, *struct trivfs_protid* **`dup`) [Function]

Return a duplicate of *cred* in *dup*, sharing the same peropen and hook. A non-null protid *hook* indicates that *trivfs_peropen_create_hook* created this protid (see Section 5.2.3 [Trivfs Options], page 35).

`error_t trivfs_set_atime` (*struct trivfs_control* *`cntl`) [Function]
`error_t trivfs_set_mtime` (*struct trivfs_control* *`cntl`) [Function]

Call these to set atime or mtime for the node to the current time.

5.2.2 Trivfs Callbacks

Like several other Hurd libraries, `libtrivfs` requires that you define a number of application-specific callback functions and configuration variables. You *must* define the following variables and functions:

`extern int trivfs_fstype` [Variable]
`extern int trivfs_fsid` [Variable]

These variables are returned in the *st_fstype* and *st_fsid* fields of `struct stat`. *trivfs_fstype* should be chosen from the `FSTYPE_*` constants found in `<hurd/hurd_types.h>`.

`extern int trivfs_allow_open` [Variable]

Set this to some bitwise OR combination of `O_READ`, `O_WRITE`, and `O_EXEC`; trivfs will only allow opens of the specified modes.

`extern int trivfs_support_read` [Variable]
`extern int trivfs_support_write` [Variable]
`extern int trivfs_support_exec` [Variable]

Set these to nonzero if trivfs should allow read, write, or execute of the file. These variables are necessary because *trivfs_allow_open* is used only to validate opens, not actual operations.

`void trivfs_modify_stat (`*struct trivfs_protid* ***cred**, struct stat* ***stbuf**`)` [Function]

This should modify a `struct stat` (as returned from the underlying node) for presentation to callers of `io_stat`. It is permissible for this function to do nothing, but it must still be defined.

`error_t trivfs_goaway (`*struct trivfs_control* ***cntl**, *int* **flags**`)` [Function]

This function is called when someone wants the filesystem *cntl* to go away. *flags* are from the set `FSYS_GOAWAY_*` found in `<hurd/hurd_types.h>`.

5.2.3 Trivfs Options

The functions and variables described in this subsection already have default definitions in `libtrivfs`, so you are not forced to define them; rather, they may be redefined on a case-by-case basis.

`extern struct port_class * trivfs_protid_portclasses []` [Variable]
`extern int trivfs_protid_nportclasses` [Variable]
`extern struct port_class * trivfs_cntl_portclasses []` [Variable]
`extern int trivfs_cntl_nportclasses` [Variable]

If you define these, they should be vectors (and the associated sizes) of port classes that will be translated into control and protid pointers for passing to RPCs, in addition to those passed to or created by `trivfs_create_control` (or `trivfs_startup`), which will automatically be recognized.

`error_t (*trivfs_check_open_hook) (`*struct trivfs_control* ***cntl**, *struct iouser* ***user**, *int* **flags**`)` [Variable]

If this variable is non-zero, it will be called every time an open happens. *user* and *flags* are from the open; *cntl* identifies the node being opened. This call need not check permissions on the underlying node. This call can block as necessary, unless `O_NONBLOCK` is set in *flags*. Any desired error can be returned, which will be reflected to the user and will prevent the open from succeeding.

`error_t (*trivfs_protid_create_hook) (`*struct trivfs_protid* ***prot**`)` [Variable]
`error_t (*trivfs_peropen_create_hook) (`*struct trivfs_peropen* ***perop**`)` [Variable]

If these variables are non-zero, they will be called every time a new protid or peropen structure is created and initialized.

`void (*trivfs_protid_destroy_hook) (`*struct trivfs_protid* ***prot**`)` [Variable]
`void (*trivfs_peropen_destroy_hook) (`*struct trivfs_peropen* ***perop**`)` [Variable]

If these variables is non-zero, they will be called every time a protid or peropen structure is about to be destroyed.

`error_t (*trivfs_getroot_hook)` (*struct trivfs_control* **`*cntl`**, *mach_port_t* **`reply_port`**, *mach_msg_type_name_t* **`reply_port_type`**, *mach_port_t* **`dotdot`**, *uid_t* **`*uids`**, *u_int* **`nuids`**, *uid_t* **`*gids`**, *u_int* **`ngids`**, *int* **`flags`**, *retry_type* **`*do_retry`**, *char* **`*retry_name`**, *mach_port_t* **`*node`**, *mach_msg_type_name_t* **`*node_type`**) [Variable]

If this variable is set, it will be called by `trivfs_S_fsys_getroot` before any other processing takes place. If the return value is `EAGAIN`, normal trivfs getroot processing continues, otherwise the RPC returns with that return value.

5.2.4 Trivfs Ports

If you choose to allocate your own trivfs port classes and buckets, the following functions may come in handy:

`error_t trivfs_add_port_bucket` (*struct port_bucket* **`**bucket`**) [Function]

Add the port bucket `*bucket` to the list of dynamically- allocated port buckets; if `*bucket` is zero, an attempt is made to allocate a new port bucket, which is then stored in `*bucket`.

`void trivfs_remove_port_bucket` (*struct port_bucket* **`*bucket`**) [Function]

Remove the previously added dynamic port bucket *bucket*, freeing it if it was allocated by `trivfs_add_port_bucket`.

`error_t trivfs_add_control_port_class` (*struct port_class* **`**class`**) [Function]

`error_t trivfs_add_protid_port_class` (*struct port_class* **`**class`**) [Function]

Add the port class `*class` to the list of control or protid port classes recognized by trivfs; if `*class` is zero, an attempt is made to allocate a new port class, which is stored in `*class`.

`void trivfs_remove_control_port_class` (*struct port_class* **`*class`**) [Function]

`void trivfs_remove_protid_port_class` (*struct port_class* **`*class`**) [Function]

Remove the previously added dynamic control or protid port class *class*, freeing it if it was allocated by `trivfs_add_control_port_class` or `trivfs_add_protid_port_class`.

Even if you do not use the above allocation functions, you may still be able to use the default trivfs cleanroutines:

`void trivfs_clean_cntl` (*void* **`*port`**) [Function]

`void trivfs_clean_protid` (*void* **`*port`**) [Function]

These functions should be installed as `libports` cleanroutines for control port classes and protid port classes, respectively.

5.3 Fshelp Library

The fshelp library implements various things that are useful to most implementors of the file protocol. It presumes that you are using the iohelp library as well. `libfshelp` is divided into separate facilities which may be used independently. These functions are declared in `<hurd/fshelp.h>`.

5.3.1 Passive Translator Linkage

These routines are self-contained and start passive translators, returning the control port. They do not require multithreading or the ports library.

`typedef error_t` (**fshelp_open_fn_t*) (*int* **`flags`**, *file_t **`node`**, *mach_msg_type_name_t **`node_type`**) [Typedef]

A callback used by the translator starting functions. Given some open flags, opens the appropriate file, and returns the node port.

`error_t fshelp_start_translator_long` (*fshelp_open_fn_t* **`underlying_open_fn`**, *char **`name`**, *char **`argz`**, *int* **`argz_len`**, *mach_port_t **`fds`**, *mach_msg_type_name_t* **`fds_type`**, *int* **`fds_len`**, *mach_port_t **`ports`**, *mach_msg_type_name_t* **`ports_type`**, *int* **`ports_len`**, *int **`ints`**, *int* **`ints_len`**, *int* **`timeout`**, *fsys_t **`control`**) [Function]

Start a passive translator *name* with arguments *argz* (length *argz_len*). Initialize the initports to *ports* (length *ports_len*), the initints to *ints* (length *ints_len*), and the file descriptor table to *fds* (length *fds_len*). Return the control port in `*control`. If the translator doesn't respond or die in *timeout* milliseconds (if *timeout* is greater than zero), return an appropriate error. If the translator dies before responding, return `EDIED`.

`error_t fshelp_start_translator` (*fshelp_open_fn_t* **`underlying_open_fn`**, *char **`name`**, *char **`argz`**, *int* **`argz_len`**, *int* **`timeout`**, *fsys_t **`control`**) [Function]

Same as `fshelp_start_translator_long`, except the initports and ints are copied from our own state, *fd*[2] is copied from our own stderr, and the other fds are cleared. For full-service filesystems, it is almost always wrong to use `fshelp_start_translator`, because the current working directory of the translator will not then be as normally expected. (Current working directories of passive translators should be the directory they were found in.) In fact, full-service filesystems should usually start passive translators as a side-effect of calling `fshelp_fetch_root` (see Section 5.3.2 [Active Translator Linkage], page 37).

5.3.2 Active Translator Linkage

These routines implement the linkage to active translators needed by any filesystem which supports them. They require the threads library and use the passive translator routines above, but they don't require the ports library at all.

This interface is complex, because creating the ports and state necessary for `start_translator_long` is expensive. The caller to `fshelp_fetch_root` should not need to create them on every call, since usually there will be an existing active translator.

`void fshelp_transbox_init` (*struct transbox **`transbox`**, *struct mutex **`lock`**, *void **`cookie`**) [Function]

Initialize a transbox, which contains state information for active translators.

`typedef error_t` (**fshelp_fetch_root_callback1_t*) (*void **`cookie1`**, *void **`cookie2`**, *uid_t **`uid`**, *gid_t **`gid`**, *char ***`argz`**, *size_t **`argz_len`**) [Typedef]

This routine is called by `fshelp_fetch_root` to fetch more information. Return the owner and group of the underlying translated file in `*uid` and `*gid`; point `*argz` at the entire passive translator specification for the file (setting `*argz_len` to the length). If there is no passive translator, then return `ENOENT`. *cookie1* is the cookie passed in `fshelp_transbox_init`. *cookie2* is the cookie passed in the call to `fshelp_fetch_root`.

typedef error_t (*fshelp_fetch_root_callback2_t) (void *`cookie1`, void *`cookie2`, int `flags`, mach_port_t *`underlying`, mach_msg_type_name_t *`underlying_type`) [Typedef]

This routine is called by `fshelp_fetch_root` to fetch more information. Return an unauthenticated node for the file itself in `*underlying` and `*underlying_type` (opened with *flags*). *cookie1* is the cookie passed in `fshelp_transbox_init`. *cookie2* is the cookie passed in the call to `fshelp_fetch_root`.

error_t fshelp_fetch_root (struct transbox *`transbox`, void *`cookie`, file_t `dotdot`, struct iouser *`user`, int `flags`, fshelp_fetch_root_callback1_t `callback1`, fshelp_fetch_root_callback2_t `callback2`, retry_type *`retry`, char *`retryname`, mach_port_t *`root`) [Function]

Fetch the root from *transbox*. *dotdot* is an unauthenticated port for the directory in which we are looking; *user* specifies the ids of the user responsible for the call. *flags* are as for `dir_lookup` (but `O_CREAT` and `O_EXCL` are not meaningful and are ignored). The transbox lock (as set by `fshelp_transbox_init`) must be held before the call, and will be held upon return, but may be released during the operation of the call.

int fshelp_translated (struct transbox *`box`) [Function]

Return true if and only if there is an active translator on this box.

error_t fshelp_set_active (struct transbox *`box`, fsys_t `newactive`, int `excl`) [Function]

Atomically replace the existing active translator port for this box with *newactive*. If *excl* is non-zero then don't modify an existing active transbox; return `EBUSY` instead.

error_t fshelp_fetch_control (struct transbox *`box`, mach_port_t *`control`) [Function]

Fetch the control port to make a request on it. It's a bad idea to use `fsys_getroot` with the result; use `fshelp_fetch_root` instead.

void fshelp_drop_transbox (struct transbox *`box`) [Function]

Clean transbox state so that deallocation or reuse is possible.

5.3.3 Fshelp Locking

The `flock` call is in flux, as the current Hurd interface (as of version 0.2) is not suitable for implementing the POSIX record-locking semantics.

5.3.4 Fshelp Permissions

These functions are designed to aid with user permission checking. It is a good idea to use these routines rather than to roll your own, so that Hurd users see consistent handling of file and directory permission bits.

`error_t fshelp_isowner` (*struct stat* *`*st`*, *struct iouser* *`*user`*) [Function]

Check to see whether *user* should be considered the owner of the file identified by *st*. If so, return zero; otherwise return an appropriate error code.

`error_t fshelp_access` (*struct stat* *`*st`*, *int* *`op`*, *struct iouser* *`*user`*) [Function]

Check to see whether the user *user* can operate on the file identified by *st*. *op* is one of `S_IREAD`, `S_IWRITE`, and `S_IEXEC`. If the access is permitted, return zero; otherwise return an appropriate error code.

`error_t fshelp_checkdirmod` (*struct stat* *`*dir`*, *struct stat* *`*st`*, *struct iouser* *`*user`*) [Function]

Check to see whether *user* is allowed to modify *dir* with respect to existing file *st*. If there is no existing file, then *st* should be set to zero. If the access is permissible, return zero; otherwise return an appropriate error code.

5.3.5 Fshelp Misc

The following functions are completely standalone:

`error_t fshelp_delegate_translation` (*char* *`*server_name`*, *mach_port_t* *`requestor`*, *char* *`**argv`*) [Function]

Try to hand off responsibility from a translator to the server located on the node *server_name*. *requestor* is the translator's bootstrap port, and *argv* is the command line. If *server_name* is null, then a name is concocted by prepending `_servers` to `argv[0]` .

`error_t fshelp_exec_reauth` (*int* *`suid`*, *uid_t* *`uid`*, *int* *`sgid`*, *gid_t* *`gid`*, *auth_t* *`auth`*, *error_t* (*`*get_file_ids`*) (*struct idvec* *`*uids`*, *struct idvec* *`*gids`*), *mach_port_t* *`*ports`*, *mach_msg_type_number_t* *`num_ports`*, *mach_port_t* *`*fds`*, *mach_msg_type_number_t* *`num_fds`*, *int* *`*secure`*) [Function]

If *suid* or *sgid* is true, adds *uid* and/or *gid* respectively to the authentication in *`ports`*`[INIT_PORT_AUTH]`, and replaces it with the result. All the other ports in *ports* and *fds* are then reauthenticated, using any privileges available through *auth*. If the auth port in *`ports`*`[INIT_PORT_AUTH]` is bogus, and *get_file_ids* is non-null, it is called to get a list of uids and gids from the file to use as a replacement. If *secure* is non-null and any added ids are new, then the variable it points to is set to nonzero, otherwise zero. If either the uid or gid case fails, then the other may still apply.

`error_t fshelp_get_identity` (*struct port_bucket* *`*bucket`*, *ino_t* *`fileno`*, *mach_port_t* *`*pt`*) [Function]

Return an identity port in *`*pt`* for the node numbered *fileno*, suitable for returning from `io_identity`; exactly one send right must be created from the returned value. *fileno* should be the same value returned as the *fileno* out-parameter in `io_identity`, and in the enclosing directory (except for mount points), and in the `st_ino` stat field. *bucket* should be a `libports` port bucket; fshelp requires the caller to make sure port operations (for no-senders notifications) are used.

`error_t` **fshelp_return_malloced_buffer** (*char* *`buf`, *size_t* `len`, *char* **`rbuf`, *mach_msg_type_number_t* *`rlen`) [Function]

Put data from the malloced buffer *buf*, *len* bytes long, into *rbuf* (which is *rlen* bytes long), suitable for returning from an RPC. If *len* is greater than zero, *buf* is freed, regardless of whether an error is returned or not.

`error_t` **fshelp_set_options** (*struct argp* *`argp`, *int* `flags`, *char* *`argz`, *size_t* `argz_len`, *void* *`input`) [Function]

Invoke `argp_parse` in the standard way, with data from *argz* and *argz_len*.

`void` **fshelp_touch** (*struct stat* *`st`, *unsigned* `what`, *volatile struct mapped_time_value* *`maptime`) [Function]

Change the stat times of *node* as indicated by *what* to the current time. *what* is a bitmask of one or more of the `TOUCH_ATIME`, `TOUCH_MTIME`, and `TOUCH_CTIME` constants.

5.4 File Interface

This section documents the interface for operating on files.

5.4.1 File Overview

The file interface is a superset of the I/O interface (see Section 4.3 [I/O Interface], page 25). Servers which provide the file interface are required to support the I/O interface as well. All objects reachable in the filesystem are expected to provide the file interface, even if they do not contain data. (The `trivfs` library makes it easy to do so for ordinary sorts of cases. See Section 5.2 [Trivfs Library], page 32.)

The interface definitions for the file interface are found in `<hurd/fs.defs>`.

Files have various pieces of status information which are returned by `io_stat` (see Section 4.3.5 [Information Queries], page 28). Most of this status information can be directly changed by various calls in the file interface; some of it should vary implicitly as the contents of the file change.

Many of these calls have general rules associated with them describing how security and privilege should operate. The `diskfs` library (see Section 8.5 [Diskfs Library], page 55) implements these rules for stored filesystems. These rules have also been implemented in the fshelp library (see Section 5.3 [Fshelp Library], page 36). Trivfs-based servers generally have no need to implement these rules at all.

In special cases, there may be a reason to implement a different security check from that specified here, or to implement a call to do something slightly different. But such cases must be carefully considered; make sure that you will not confuse innocent user programs through excessive cleverness.

If some operation cannot be implemented (for example, `chauthor` over FTP), then the call should return `EOPNOTSUPP`. If it is merely difficult to implement a call, it is much better to figure out a way to implement it as a series of operations rather than to return errors to the user.

5.4.2 Changing Status

There are several RPCs available for users to change much of the status information associated with a file. (The information is returned by the `io_stat` RPC; see Section 4.3.5 [Information Queries], page 28.)

All these operations are restricted to root and the owner of the file. When attempted by another user, they should return `EPERM`.

The `file_chown` RPC changes the owner and group of the file. Only root should be able to change the owner, and changing the group to a group the caller is not in should also be prohibited. Violating either of these conditions should return `EPERM`.

The `file_chauthor` RPC changes the author of the file. It should be legitimate to change the author to any value without restriction.

The `file_chmod` RPC changes the file permission mode bits.

The `file_chflags` RPC changes the flags of the file. It should be legitimate to change the flags to any value without restriction. No standard meanings have been assigned to the flags yet, but we intend to do so. Do not assume that the flags format we choose will map identically to that of some existing filesystem format.

The `file_utimes` RPC changes the *atime* and *mtime* of the file. Making this call must cause the *ctime* to be updated as well, even if no actual change to either the *mtime* or the *atime* occurs.

The `file_set_size` RPC is special; not only does it change the status word specifying the size of the file, but it also changes the actual contents of the file. If the file size is being reduced it should release secondary storage associated with the previous contents of the file. If the file is being extended, the new region added to the file must be zero-filled. Unlike the other RPCs in this section, `file_set_size` should be permitted to any user who is allowed to write the file.

5.4.3 Program Execution

Execution of programs on the Hurd is done through fileservers with the `file_exec` RPC. The fileserver is expected to verify that the user is allowed to execute the file, make whatever modifications to the ports are necessary for setuid execution, and then invoke the standard execserver found on '`/servers/exec`'.

This section specifically addresses what fileservers are expected to do, with minimal attention to the other parts of the process. See Chapter 13 [Running Programs], page 73, for more general information.

The file must be opened for execution; if it is not, `EBADF` should be returned. In addition, at least one of the execute bits must be on. A failure of this check should result in `EACCES`—not `ENOEXEC`. It is not proper for the fileserver ever to respond to the `file_exec` RPC with `ENOEXEC`.

If either the setuid or setgid bits are set, the server needs to construct a new authentication handle with the additional new ID's. Then all the ports passed to `file_exec` need to be reauthenticated with the new handle. If the fileserver is unable to make the new authentication handle (for example, because it is not running as root) it is not acceptable to return an error; in such a case the server should simply silently fail to implement the setuid/setgid semantics.

If the setuid/setgid transformation adds a new uid or gid to the user's authentication handle that was not previously present (as opposed to merely reordering them), then the `EXEC_SECURE` and `EXEC_NEWTASK` flags should both be added in the call to `exec_exec`.

The server then needs to open a new port onto the executed file which will not share any file pointers with the port the user passed in, opened with `O_READ`. Finally, all the information (mutated appropriately for setuid/setgid) should be sent to the execserver with `exec_exec`. Whatever error code `exec_exec` returns should returned to the caller of `file_exec`.

5.4.4 File Locking

The `flock` call is in flux, as the current Hurd interface (as of version 0.2) is not suitable for implementing the POSIX record-locking semantics.

You should ignore the `file_lock` and `file_lock_stat` calls until the new record-locking interface is implemented.

5.4.5 File Frobbing

FIXME: Other active calls on files

`file_sync`

`file_getfh`

`file_getlinknode`

`file_check_access`

These manipulate meta-information:

`file_reparent`

`file_statfs`

`file_syncfs`

`file_getcontrol`

`file_get_storage_info`

`file_get_fs_options`

5.4.6 Opening Files

FIXME: Looking up files in directories

`dir_lookup`

`dir_readdir`

5.4.7 Modifying Directories

`kern_return_t dir_mkfile` (*file_t* `directory`, *int* `flags`, *mode_t* `mode`, *mach_port_t* `*newnode`) [Function]

Create a new file in *directory* without linking it into the filesystem. You still must have write permission on the specified directory, even though it will not actually be written.

The function returns a port to the new file in **newnode*. Flags are the same as for `dir_lookup`, but `O_CREAT` and `O_TRUNC` are assumed even if not specified.

`kern_return_t dir_mkdir` (*file_t* ***directory***, *char* ****name***, *mode_t* ***mode***) [Function]

Create a new directory named *name* in *directory* with permission specified by *mode*.

`kern_return_t dir_rmdir` (*file_t* ***directory***, *char* ****name***) [Function]

Remove the directory named *name* from *directory*.

`kern_return_t dir_unlink` (*file_t* ***directory***, *char* ****name***) [Function]

Remove the non-directory node *name* from *directory*.

`kern_return_t dir_link` (*file_t* ***directory***, *file_t* ***file***, *char* ****name***, *int* ***excl***) [Function]

Create a hard link in *directory*. If *excl* is set and *name* already exists in *directory*, then this function will fail. If *excl* is not set and *name* already exists the old file named *name* will be unlinked. If *directory* and *file* are not on the same filesystem, then `dir_link` might fail with `EXDEV`.

`kern_return_t dir_rename` (*file_t* ***olddirectory***, *char* ****oldname***, *file_t* ***newdirectory***, *char* ****newname***, *int* ***excl***) [Function]

Move the node *oldname* in *olddirectory* to the node *newname* in *newdirectory*. If *excl* is set and *newname* already exists in *newdirectory*, then this function will fail. If *excl* is not set and *newname* already exists, the old file named *newname* will be unlinked. If *olddirectory* and *newdirectory* are not on the same filesystem, then `dir_rename` might fail with `EXDEV`.

5.4.8 Notifications

FIXME: File and directory change callbacks

File change notifications are not yet implemented, but directory notifications are.

`file_notice_changes`

`dir_notice_changes`

5.4.9 File Translators

FIXME: How to set and get translators

`file_set_translator`

`file_get_translator`

`file_get_translator_cntl`

5.5 Filesystem Interface

The filesystem interface (described in `<hurd/fsys.defs>`) is supported by translator control ports.

FIXME: finish

6 Special Files

In Unix, any file that does not act as a general-purpose unit of storage is called a *special file*. These are FIFOs, Unix-domain sockets, and device nodes. In the Hurd, there is no need for the "special file" distinction, since they are implemented by translators, just as regular files are.

Nevertheless, the Hurd maintains this distinction, in order to provide backward compatibility for Unix programs (which do not know about translators). Studying the implementation of Hurd special files is a good way to introduce the idea of translators to people who are familiar with Unix.

This chapter does not discuss '`/dev/zero`' or any of the microkernel-based devices, since these are translated by the generalized storeio server (FIXME xref).

FIXME: finish

6.1 fifo

6.2 ifsock

6.3 magic

6.4 null

FIXME: a chapter on libtreefs and libdirmgt will probably go here

7 Stores

A *store* is a fixed-size block of storage, which can be read and perhaps written to. A store is more general than a file: it refers to any type of storage such as devices, files, memory, tasks, etc. Stores can also be representations of other stores, which may be combined and filtered in various ways.

7.1 storeinfo, storecat, storeread

7.2 storeio

FIXME: finish

7.3 Store Library

The store library (which is declared in `<hurd/store.h>`) implements many different backends which support the store abstraction. Hurd programs use `libstore` so that new storage types can be implemented with minimum impact.

7.3.1 Store Arguments

FIXME: describe startup sequence

`struct store_parsed` [Structure]

The result of parsing a store, which should be enough information to open it, or return the arguments.

`struct store_argp_params {` *struct store_parsed* *`result`; *const char* *`default_type`; *const struct store_class* **const* *`classes`; `}` [Structure]

This is the structure used to pass args back and forth from *store_argp*. *result* is the resulting parsed result. If '`--store-type`' isn't specified, then *default_type* should be used as the store type; zero is equivalent to `"query"`. *classes* is set of classes used to validate store types and argument syntax.

`extern struct argp store_argp` [Variable]

This is an argument parser that may be used for parsing a simple command line specification for stores. The accompanying input parameter must be a pointer to a `struct store_argp_params`.

`void store_parsed_free` (*struct store_parsed* *`parsed`) [Function]

Free all resources used by *parsed*.

`error_t store_parsed_open` (*const struct store_parsed* *`parsed`, *int* `flags`, *struct store* **`store`) [Function]

Open the store specified by *parsed*, and return it in *store*.

`error_t store_parsed_append_args` (*const struct store_parsed* *`parsed`, *char* **`argz`, *size_t* *`argz_len`) [Function]

Add the arguments used to create *parsed* to *argz* and *argz_len*.

`error_t` **`store_parsed_name`** (*const struct store_parsed ****parsed***, *char *****name***) [Function]

Make an option string describing *parsed*, and return it in malloced storage in *name*.

7.3.2 Store Management

The following functions provide basic management of stores:

`error_t` **`store_create`** (*file_t* ***source***, *int* ***flags***, *const struct store_class *const ****classes***, *struct store *****store***) [Function]

Return a new store in *store*, which refers to the storage underlying *source*. *classes* is used to select classes specified by the provider; if zero, *store_std_classes* is used. *flags* is set with `store_set_flags`, with the exception of `STORE_INACTIVE`, which merely indicates that no attempt should be made to activate an inactive store; if `STORE_INACTIVE` is not specified, and the store returned for SOURCE is inactive, an attempt is made to activate it (failure of which causes an error to be returned). A reference to *source* is created (but may be destroyed with `store_close_source`).

It is usually better to use a specific store open or create function such as `store_open` (see Section 7.3.4 [Store Classes], page 48), since they are tailored to the needs of a specific store. Generally, you should only use `store_create` if you are defining your own store class, or you need options that are not provided by a more specific store creation function.

`void` **`store_close_source`** (*struct store ****store***) [Function]

If *store* was created using `store_create`, remove the reference to the source from which it was created.

`void` **`store_free`** (*struct store ****store***) [Function]

Clean up and deallocate *store*'s underlying stores.

`struct` **`store_run`** `{` *store_offset_t* ***start***, ***length***; `}` [Structure]

A `struct store_run` represents a contiguous region in a store's address range. These are used to designate active portions of a store. If *start* is -1, then the region is a *hole* (it is zero-filled and doesn't correspond to any real addresses).

`error_t` **`store_set_runs`** (*struct store ****store***, *const struct store_run ****runs***, *size_t* ***num_runs***) [Function]

Set *store*'s current runs list to (a copy of) *runs* and *num_runs*.

`error_t` **`store_set_children`** (*struct store ****store***, *struct store *const ****children***, *size_t* ***num_children***) [Function]

Set *store*'s current children to (a copy of) *children* and *num_children* (note that just the vector *children* is copied, not the actual children).

`error_t` **`store_children_name`** (*const struct store ****store***, *char *****name***) [Function]

Try to come up with a name for the children in *store*, combining the names of each child in a way that could be used to parse them with `store_open_children`. This is done heuristically, and so may not succeed. If a child doesn't have a name, `EINVAL` is returned.

error_t **store_set_name** (*struct store* ***store**, *const char* ***name**) [Function]
Sets the name associated with *store* to a copy of *name*.

error_t **store_set_flags** (*struct store* ***store**, *int* **flags**) [Function]
Add *flags* to *store*'s currently set flags.

error_t **store_clear_flags** (*struct store* ***store**, *int* **flags**) [Function]
Remove *flags* from *store*'s currently set flags.

error_t **store_set_child_flags** (*struct store* ***store**, *int* **flags**) [Function]
Set *flags* in all children of *store*, and if successful, add *flags* to *store*'s flags.

error_t **store_clear_child_flags** (*struct store* ***store**, *int* **flags**) [Function]
Clear *flags* in all children of *store*, and if successful, remove *flags* from *store*'s flags.

int **store_is_securely_returnable** (*struct store* ***store**, *int* **open_flags**) [Function]
Returns true if *store* can safely be returned to a user who has accessed it via a node using *open_flags*, without compromising security.

error_t **store_clone** (*struct store* ***from**, *struct store* ****to**) [Function]
Return a copy of *from* in *to*.

error_t **store_remap** (*struct store* ***source**, *const struct store_run* ***runs**, *size_t* **num_runs**, *struct store* ****store**) [Function]
Return a store in *store* that reflects the blocks in *runs* and *runs_len* from source; *source* is consumed, but not *runs*. Unlike the `store_remap_create` function, this may simply modify *source* and return it.

7.3.3 Store I/O

The following functions allow you to read and modify the contents of a store:

error_t **store_map** (*const struct store* ***store**, *vm_prot_t* **prot**, *mach_port_t* ***memobj**) [Function]
Return a memory object paging on *store*.

error_t **store_read** (*struct store* ***store**, *store_offset_t* **addr**, *size_t* **amount**, *void* ****buf**, *size_t* ***len**) [Function]
Read *amount* bytes from *store* at *addr* into *buf* and *len* (which follows the usual Mach buffer-return semantics) to *store* at *addr*. *addr* is in *blocks* (as defined by `store->block_size`). Note that *len* is in bytes.

error_t **store_write** (*struct store* ***store**, *store_offset_t* **addr**, *void* ***buf**, *size_t* **len**, *size_t* ***amount**) [Function]
Write *len* bytes from *buf* to *store* at *addr*. Returns the amount written in *amount* (in bytes). *addr* is in *blocks* (as defined by `store->block_size`).

error_t **store_set_size** (*struct store* ***store**, *store_offset_t* **newsize**) [Function]
Set *store*'s size to *newsize* (in bytes).

7.3.4 Store Classes

The store library comes with a number of standard store class implementations:

`extern const struct store_class *const store_std_classes []` [Variable]

This is a null-terminated vector of the standard store classes implemented by `libstore`.

If you are building your own class vectors, the following function may be useful:

`error_t store_concat_class_vectors` (*struct store_class* **`cv1`, *struct store_class* **`cv2`, *struct store_class* ***`concat`) [Variable]

Concatenate the store class vectors in *cv1* and *cv2*, and return a new (malloced) vector in *concat*.

7.3.4.1 `query` store

`extern const struct store_class store_query_class` [Variable]

This store is a virtual store which queries a filesystem node, and delegates control to an appropriate store class.

`error_t store_open` (*const char* *`name`, *int* `flags`, *const struct store_class *const* *`classes`, *struct store* **`store`) [Function]

Open the file *name*, and return a new store in *store*, which refers to the storage underlying it. *classes* is used to select classes specified by the provider; if it is zero, then *store_std_classes* is used. *flags* is set with `store_set_flags`. A reference to the open file is created (but may be destroyed with `store_close_source`).

7.3.4.2 `typed_open` store

`extern const struct store_class store_typed_open_class` [Variable]

This store is special in that it doesn't correspond to any specific store functions, rather it provides a way to interpret character strings as specifications for other stores.

`error_t store_typed_open` (*const char* *`name`, *int* `flags`, *const struct store_class *const* *`classes`, *struct store* **`store`) [Function]

Open the store indicated by *name*, which should consist of a store type name followed by a '`:`' and any type-specific name, returning the new store in *store*. *classes* is used to select classes specified by the type name; if it is zero, *store_std_classes* is used.

`error_t store_open_children` (*const char* *`name`, *int* `flags`, *const struct store_class *const* *`classes`, *struct store* ***`stores`, *size_t* *`num_stores`) [Function]

Parse multiple store names in *name*, and open each individually, returning all in the vector *stores*, and the number in *num_stores*. The syntax of *name* is a single non-alphanumeric separator character, followed by each child store name separated by the same separator; each child name is '`type:name`' notation as parsed by `store_typed_open`. If every child uses the same '`type:`' prefix, then it may be factored out and put before the child list instead (the two notations are differentiated by whether or not the first character of *name* is alphanumeric).

7.3.4.3 `device` store

`extern const struct store_class store_device_class` [Variable]

This store is a simple wrapper for a microkernel device driver.[1]

`error_t store_device_open` (*const char* *****name***, *int* ***flags***, *struct store* ******store***) [Function]

Open the device named *name*, and return the corresponding store in *store*.

`error_t store_device_create` (*device_t* ***device***, *int* ***flags***, *struct store* ******store***) [Function]

Return a new store in *store* referring to the microkernel device *device*. Consumes the *device* send right.

7.3.4.4 `file` store

`extern const struct store_class store_file_class` [Variable]

This store reads and writes the contents of a Hurd file.

`error_t store_file_open` (*const char* *****name***, *int* ***flags***, *struct store* ******store***) [Function]

Open the file *name*, and return the corresponding store in *store*.

`error_t store_file_create` (*file_t* ***file***, *int* ***flags***, *struct store* ******store***) [Function]

Return a new store in *store* referring to the file *file*. Unlike `store_create`, this will always use file I/O, even it would be possible to be more direct. This may work in more cases, for instance if the file has holes. Consumes the *file* send right.

7.3.4.5 `task` store

`extern const struct store_class store_task_class` [Variable]

This store provides access to the contents of a microkernel task.

`error_t store_task_open` (*const char* *****name***, *int* ***flags***, *struct store* ******store***) [Variable]

Open the task *name* (*name* should be the task's pid), and return the corresponding store in *store*.

`error_t store_task_create` (*task_t* ***task***, *int* ***flags***, *struct store* ******store***) [Variable]

Return a new store in *store* referring to the task *task*, consuming the *task* send right.

---

[1] It is important to note that device drivers are not provided by the Hurd, but by the underlying microkernel. Hurd 'devices' are just storeio-translated nodes which make the microkernel device drivers obey Hurd semantics. If you wish to implement a new device driver, you will need to consult the appropriate microkernel documentation.

7.3.4.6 `zero` store

`extern const struct store_class store_zero_class` [Variable]

Reads to this store always return zero-filled buffers, no matter what has been written into it. This store corresponds to the Unix '`/dev/zero`' device node.

`error_t store_zero_create` (*store_offset_t* **`size`**, *int* **`flags`**, *struct store* ****`store`**) [Function]

Return a new zero store *size* bytes long in *store*.

7.3.4.7 `copy` store

`extern const struct store_class store_copy_class` [Variable]

This store provides a temporary copy of another store. This is useful if you want to provide writable data, but do not wish to modify the underlying store. All changes to a copy store are lost when it is closed.

`error_t store_copy_open` (*const char* ***`name`**, *int* **`flags`**, *const struct store_class *const* ***`classes`**, *struct store* ****`store`**) [Function]

Open the copy store *name* (which consists of another store class name, a '`:`', and a name for the store class to open) and return the corresponding store in *store*. *classes* is used to select classes specified by the type name; if it is zero, *store_std_classes* is used.

`error_t store_copy_create` (*struct store* ***`from`**, *int* **`flags`**, *struct store* ****`store`**) [Function]

Return a new store in *store* which contains a snapshot of the contents of the store *from*; *from* is consumed.

`error_t store_buffer_create` (*void* ***`buf`**, *size_t* **`buf_len`**, *int* **`flags`**, *struct store* ****`store`**) [Function]

Return a new store in *store* which contains the memory buffer *buf*, of length *buf_len*. *buf* must be allocated with `vm_allocate`, and will be consumed.

7.3.4.8 `gunzip` store

`extern const struct store_class store_gunzip_class` [Variable]

This store provides transparent GNU zip decompression of a substore. Unfortunately, this store is currently read-only.

`error_t store_gunzip_open` (*const char* ***`name`**, *int* **`flags`**, *const struct store_class *const* ***`classes`**, *struct store* ****`store`**) [Variable]

Open the gunzip store *name* (which consists of another store class name, a '`:`', and a name for that store class to open), and return the corresponding store in *store*. *classes* is used to select classes specified by the type name; if it is zero, *store_std_classes* is used.

`error_t store_gunzip_create` (*struct store* ***`from`**, *int* **`flags`**, *struct store* ****`store`**) [Variable]

Return a new store in *store* which contains a snapshot of the uncompressed contents of the store *from*; *from* is consumed. *block_size* is the desired block size of the result.

7.3.4.9 `concat` store

`extern const struct store_class store_concat_class` [Variable]

This class provides a linear concatenation storage mode. It creates a new virtual store which consists of several different substores appended to one another.

This mode is designed to increase storage capacity, so that when one substore is filled, new data is transparently written to the next substore. Concatenation requires robust hardware, since a failure in any single substore will wipe out a large section of the data.

`error_t store_concat_open` (*const char* ***`name`**, *int* **`flags`**, *const struct store_class* **const* ***`classes`**, *struct store* ****`store`**) [Function]

Return a new store that concatenates the stores created by opening all the individual stores described in *name*; for the syntax of *name*, see `store_open_children`.

`error_t store_concat_create` (*struct store* * *const* ***`stores`**, *size_t* **`num_stores`**, *int* **`flags`**, *struct store* ****`store`**) [Function]

Return a new store in *store* that concatenates all the stores in *stores* (*num_stores* of them). The stores in *stores* are consumed; that is, they will be freed when this store is freed. The *stores array*, however, is copied, and so should be freed by the caller.

7.3.4.10 `ileave` store

`extern const struct store_class store_ileave_class` [Variable]

This class provides a RAID-0[2] storage mode (also called *disk striping*). It creates a new virtual store by interleaving the contents of several different substores.

This RAID mode is designed to increase storage performance, since I/O will probably occur in parallel if the substores reside on different physical devices. Interleaving works best with evenly-yoked substores... if the stores are different sizes, some space will be not be used at the end of the larger stores; if the stores are different speeds, then I/O will have to wait for the slowest store; if some stores are not as reliable as others, failures will wipe out every *n*th storage block, where *n* is the number of substores.

`error_t store_ileave_create` (*struct store* * *const* ***`stripes`**, *size_t num_stripes*, *store_offset_t* **`interleave`**, *int* **`flags`**, *struct store* ****`store`**) [Function]

Return a new store in *store* that interleaves all the stores in *stripes* (*num_stripes* of them) every *interleave* bytes; *interleave* must be an integer multiple of each stripe's block size. The stores in *stripes* are consumed; that is, they will be freed when this store is freed. The *stripes array*, however, is copied, and so should be freed by the caller.

---

[2] "RAID" stands for *Redundant Array of Independent Disks*: several disks used in parallel to achieve increased capacity, redundancy and/or performance.

7.3.4.11 `mvol` store

`extern const struct store_class store_mvol_class` [Variable]

This store provides access to multiple volumes using a single-volume device. One use of this store would be to provide a store which consists of multiple floppy disks when there is only a single disk drive. It works by remapping a single linear address range to multiple address ranges, and keeping track of the currently active range. Whenever a request maps to a range that is not active, a callback is made in order to switch to the new range.

This class is not included in *store_std_classes*, because it requires an application-specific callback.

`error_t store_mvol_create` (*struct store **`phys`, *error_t* (*`swap_vols`) (*struct store **`store`, *size_t* `new_vol`, *ssize_t* `old_vol`), *int* `flags`, *struct store ***`store`) [Function]

Return a new store in *store* that multiplexes multiple physical volumes from *phys* as one larger virtual volume. *swap_vols* is a function that will be called whenever reads or writes refer to a block which is not addressable on the currently active volume. *phys* is consumed.

7.3.4.12 `remap` store

`extern const struct store_class store_remap_class` [Variable]

This store translates I/O requests into different addresses on a different store.

`error_t store_remap_create` (*struct store **`source`, *const struct store_run **`runs`, *size_t* `num_runs`, *int* `flags`, *struct store ***`store`) [Function]

Return a new store in *store* that reflects the blocks in *runs* and *runs_len* from *source*; *source* is consumed, but *runs* is not. Unlike the `store_remap` function, this function always operates by creating a new store of type '`remap`' which has *source* as a child, and so may be less efficient than `store_remap` for some types of stores.

7.3.5 Store RPC Encoding

The store library also provides some functions which help transfer stores between tasks via RPC:

`struct store_enc` [Structure]

This structure is used to hold the various bits that make up the representation of a store for transmission via RPC. See `<hurd/hurd_types.h>` for an explanation of the encodings for the various storage types.

`void store_enc_init` (*struct store_enc **`enc`, *mach_port_t **`ports`, *mach_msg_type_number_t* `num_ports`, *int **`ints`, *mach_msg_type_number_t* `num_ints`, *off_t **`offsets`, *mach_msg_type_number_t* `num_offsets`, *char **`data`, *mach_msg_type_number_t* `data_len`) [Function]

Initialize *enc*. The given vector and sizes will be used for the encoding if they are big enough (otherwise new ones will be automatically allocated).

`void store_enc_dealloc` (*struct store_enc ****enc***) [Function]

Deallocate storage used by the fields in *enc* (but nothing is done with *enc* itself).

`void store_enc_return` (*struct store_enc ****enc***, *mach_port_t *****ports***, *mach_msg_type_number_t ****num_ports***, *int *****ints***, *mach_msg_type_number_t ****num_ints***, *off_t *****offsets***, *mach_msg_type_number_t ****num_offsets***, *char *****data***, *mach_msg_type_number_t ****data_len***) [Function]

Copy out the parameters from *enc* into the given variables suitably for returning from a `file_get_storage_info` RPC, and deallocate *enc*.

`error_t store_return` (*const struct store ****store***, *mach_port_t *****ports***, *mach_msg_type_number_t ****num_ports***, *int *****ints***, *mach_msg_type_number_t ****num_ints***, *off_t *****offsets***, *mach_msg_type_number_t ****num_offsets***, *char *****data***, *mach_msg_type_number_t ****data_len***) [Function]

Encode *store* into the given return variables, suitably for returning from a `file_get_storage_info` RPC.

`error_t store_encode` (*const struct store ****store***, *struct store_enc ****enc***) [Function]

Encode *store* into *enc*, which should have been prepared with `store_enc_init`, or return an error. The contents of *enc* may then be returned as the value of `file_get_storage_info`; if for some reason this can't be done, `store_enc_dealloc` may be used to deallocate the memory used by the unsent vectors.

`error_t store_decode` (*struct store_enc ****enc***, *const struct store_class *const ****classes***, *struct store *****store***) [Function]

Decode *enc*, either returning a new store in *store*, or an error. *classes* is the mapping from Hurd storage class ids to store classes; if it is zero, *store_std_classes* is used. If nothing else is to be done with *enc*, its contents may then be freed using `store_enc_dealloc`.

`error_t store_allocate_child_encodings` (*const struct store ****store***, *struct store_enc ****enc***) [Function]

Calls the `allocate_encoding` method in each child store of *store*, propagating any errors. If any child does not have such a method, `EOPNOTSUPP` is returned.

`error_t store_encode_children` (*const struct store ****store***, *struct store_enc ****enc***) [Function]

Calls the encode method in each child store of *store*, propagating any errors. If any child does not have such a method, `EOPNOTSUPP` is returned.

`error_t store_decode_children` (*struct store_enc ****enc***, *int* ***num_children***, *const struct store_class *const ****classes***, *struct store *****children***) [Function]

Decodes *num_children* from *enc*, storing the results into successive positions in *children*.

`error_t` **`store_with_decoded_runs`** (*struct store_enc* ***`enc`***, *size_t* ***`num_runs`***, *error_t* (*****`fun`***) (*const struct store_run* ***`runs`***, *size_t* ***`num_runs`***)) [Function]

Call *fun* with the vector *runs* of length *num_runs* extracted from *enc*.

`error_t` **`store_std_leaf_allocate_encoding`** (*const struct store* ****`store`***, *struct store_enc* ****`enc`***) [Function]

`error_t` **`store_std_leaf_encode`** (*const struct store* ****`store`***, *struct store_enc* ****`enc`***) [Function]

Standard encoding used for most data-providing (as opposed to filtering) store classes.

`typedef` **`error_t`** (**store_std_leaf_create_t*) (*mach_port_t* ***`port`***, *int* ***`flags`***, *size_t* ***`block_size`***, *const struct store_run* ****`runs`***, *size_t* ***`num_runs`***, *struct store* *****`store`***) [Typedef]

Creation function used by `store_std_leaf_decode`.

`error_t` **`store_std_leaf_decode`** (*struct store_enc* ****`enc`***, *store_std_leaf_create_t* ***`create`***, *struct store* *****`store`***) [Function]

Decodes the standard leaf encoding which is common to various builtin formats, and calls *create* to actually create the store.

8 Stored Filesystems

Stored filesystems allow users to save and load persistent data from any random-access storage media, such as hard disks, floppy diskettes, and CD-ROMs. Stored filesystems are required for bootstrapping standalone workstations, as well.

8.1 Repairing Filesystems

FIXME: finish

8.2 Linux Extended 2 FS

FIXME: finish

8.3 BSD Unix FS

FIXME: finish

8.4 ISO-9660 CD-ROM FS

FIXME: finish

8.5 Diskfs Library

The diskfs library is declared in `<hurd/diskfs.h>`, and does a lot of the work of implementing stored filesystems. `libdiskfs` requires the threads, ports, iohelp, fshelp, and store libraries. You should understand all these libraries before you attempt to use diskfs, and you should also be familiar with the pager library (see Section 4.2 [Pager Library], page 22).

For historical reasons, the library for implementing stored filesystems is called `libdiskfs` instead of `libstorefs`. Keep in mind, however, that diskfs is useful for filesystems which are implemented on any block-addressed storage device, since it uses the store library to do I/O.

Note that stored filesystems can be tricky to implement, since the diskfs callback interfaces are not trivial. It really is best if you examine the source code of a similar existing filesystem server, and follow its example rather than trying to write your own from scratch.

8.5.1 Diskfs Startup

This subsection gives an outline of the general steps involved in implementing a filesystem server, to help refresh your memory and to offer explanations rather than to serve as a tutorial.

The first thing a filesystem server should do is parse its command-line arguments (see Section 8.5.2 [Diskfs Arguments], page 56). Then, the standard output and error streams should be redirected to the console, so that error messages are not lost if this is the bootstrap filesystem:

`void diskfs_console_stdio` (*void*) [Function]

Redirect error messages to the console, so that they can be seen by users.

The following is a list of the relevant functions which would be called during the rest of the server initialization. Again, you should refer to the implementation of an already-working filesystem if you have any questions about how these functions should be used:

`error_t diskfs_init_diskfs` (*void*) [Function]

Call this function after arguments have been parsed to initialize the library. You must call this before calling any other diskfs functions, and after parsing diskfs options.

`void diskfs_spawn_first_thread` (*void*) [Function]

Call this after all format-specific initialization is done (except for setting `diskfs_root_node`); at this point the pagers should be ready to go.

`mach_port_t diskfs_startup_diskfs` (*mach_port_t* ***bootstrap***, *int* ***flags***) [Function]

Call this once the filesystem is fully initialized, to advertise the new filesystem control port to our parent filesystem. If *bootstrap* is set, diskfs will call `fsys_startup` on that port as appropriate and return the *realnode* from that call; otherwise we call `diskfs_start_bootstrap` and return `MACH_PORT_NULL`. *flags* specifies how to open *realnode* (from the `O_*` set).

You should not need to call the following function directly, since `diskfs_startup_diskfs` will do it for you, when appropriate:

`void diskfs_start_bootstrap` (*void*) [Function]

Start the Hurd bootstrap sequence as if we were the bootstrap filesystem (that is, `diskfs_boot_flags` is nonzero). All filesystem initialization must be complete before you call this function.

8.5.2 Diskfs Arguments

The following functions implement standard diskfs command-line and runtime argument parsing, using argp (see section "Argp" in *The GNU C Library Reference Manual*):

`error_t diskfs_set_options` (*char* *****argz***, *size_t* ***argz_len***) [Function]

Parse and execute the runtime options specified by *argz* and *argz_len*. `EINVAL` is returned if some option is unrecognized. The default definition of this routine will parse them using `diskfs_runtime_argp`.

`error_t diskfs_append_args` (*char* ******argz***, *unsigned* *****argz_len***) [Function]

Append to the malloced string `*argz` of length `*argz_len` a NUL-separated list of the arguments to this translator. The default definition of this routine simply calls `diskfs_append_std_options`.

`error_t diskfs_append_std_options` (*char* ******argz***, *unsigned* *****argz_len***) [Function]

Appends NUL-separated options describing the standard diskfs option state to *argz* and increments *argz_len* appropriately. Note that unlike `diskfs_get_options`, *argz* and *argz_len* must already have sane values.

`struct argp * diskfs_runtime_argp` [Variable]

If this is defined or set to an argp structure, it will be used by the default `diskfs_set_options` to handle runtime option parsing. The default definition is initialized to a pointer to `diskfs_std_runtime_argp`.

`const struct argp diskfs_std_runtime_argp` [Variable]

An argp for the standard diskfs runtime options. The default definition of `diskfs_runtime_argp` points to this, although the user can redefine that to chain this onto his own argp.

`const struct argp diskfs_startup_argp` [Variable]

An argp structure for the standard diskfs command line arguments. The user may call `argp_parse` on this to parse the command line, chain it onto the end of his own argp structure, or ignore it completely.

`const struct argp diskfs_store_startup_argp` [Variable]

An argp structure for the standard diskfs command line arguments plus a store specification. The address of a location in which to return the resulting `struct store_parsed` structure should be passed as the input argument to `argp_parse`; FIXME xref the declaration for STORE_ARGP.

8.5.3 Diskfs Globals

The following functions and variables control the overall behaviour of the library. Your callback functions may need to refer to these, but you should not need to modify or redefine them.

`mach_port_t diskfs_default_pager` [Variable]
`mach_port_t diskfs_exec_ctl` [Variable]
`mach_port_t diskfs_exec` [Variable]
`auth_t diskfs_auth_server_port` [Variable]

These are the respective send rights to the default pager, execserver control port, execserver itself, and authserver.

`mach_port_t diskfs_fsys_identity` [Variable]

The `io_identity` identity port for the filesystem.

`char ** diskfs_argv` [Variable]

The command line with which diskfs was started, set by the default argument parser. If you don't use it, set this yourself. This is only used for bootstrap file systems, to give the procserver.

`char * diskfs_boot_flags` [Variable]

When this is a bootstrap filesystem, the command line options passed from the kernel. If not a bootstrap filesystem, it is zero, so it can be used to distinguish between the two cases.

`struct rwlock diskfs_fsys_lock` [Variable]

Hold this lock while doing filesystem-level operations. Innocuous users can just hold a reader lock, but operations that might corrupt other threads should hold a writer lock.

`volatile struct mapped_time_value * diskfs_mtime` [Variable]

The current system time, as used by the diskfs routines. This is converted into a `struct timeval` by the `maptime_read` C library function (FIXME xref).

`int diskfs_synchronous` [Variable]

True if and only if we should do every operation synchronously. It is the format-specific code's responsibility to keep allocation information permanently in sync if this is set; the rest will be done by format-independent code.

`error_t diskfs_set_sync_interval` (*int* ***interval***) [Function]

Establish a thread to sync the filesystem every *interval* seconds, or never, if *interval* is zero. If an error occurs creating the thread, it is returned, otherwise zero. Subsequent calls will create a new thread and (eventually) get rid of the old one; the old thread won't do any more syncs, regardless.

`spin_lock_t diskfs_node_refcnt_lock` [Variable]

Pager reference count lock.

`int diskfs_readonly` [Variable]

Set to zero if the filesystem is currently writable.

`error_t diskfs_set_readonly` (*int* ***readonly***) [Function]

Change an active filesystem between read-only and writable modes, setting the global variable *diskfs_readonly* to reflect the current mode. If an error is returned, nothing will have changed. *diskfs_fsys_lock* should be held while calling this routine.

`int diskfs_check_readonly` (*void*) [Function]

Check if the filesystem is readonly before an operation that writes it. Return nonzero if readonly, otherwise zero.

`error_t diskfs_remount` (*void*) [Function]

Reread all in-core data structures from disk. This function can only be successful if *diskfs_readonly* is true. *diskfs_fsys_lock* should be held while calling this routine.

`error_t diskfs_shutdown` (*int* ***flags***) [Function]

Shutdown the filesystem; *flags* are as for `fsys_shutdown`.

8.5.4 Diskfs Node Management

Every file or directory is a diskfs *node*. The following functions help your diskfs callbacks manage nodes and their references:

`void diskfs_drop_node` (*struct node* ****np***) [Function]

Node *np* now has no more references; clean all state. The *diskfs_node_refcnt_lock* must be held, and will be released upon return. *np* must be locked.

`void diskfs_node_update` (*struct node* ****np***, *int* ***wait***) [Function]

Set disk fields from *np*`->dn_stat`; update ctime, atime, and mtime if necessary. If *wait* is true, then return only after the physical media has been completely updated.

`void diskfs_nref` (*struct node* ****np***) [Function]

Add a hard reference to node *np*. If there were no hard references previously, then the node cannot be locked (because you must hold a hard reference to hold the lock).

`void diskfs_nput` (*struct node *`np`*) [Function]

Unlock node *np* and release a hard reference; if this is the last hard reference and there are no links to the file then request light references to be dropped.

`void diskfs_nrele` (*struct node *`np`*) [Function]

Release a hard reference on *np*. If *np* is locked by anyone, then this cannot be the last hard reference (because you must hold a hard reference in order to hold the lock). If this is the last hard reference and there are no links, then request light references to be dropped.

`void diskfs_nref_light` (*struct node *`np`*) [Function]

Add a light reference to a node.

`void diskfs_nput_light` (*struct node *`np`*) [Function]

Unlock node *np* and release a light reference.

`void diskfs_nrele_light` (*struct node *`np`*) [Function]

Release a light reference on *np*. If *np* is locked by anyone, then this cannot be the last reference (because you must hold a hard reference in order to hold the lock).

`error_t diskfs_node_rdwr` (*struct node *`np`, char *`data`, off_t `off`, size_t `amt`, int `direction`, struct protid *`cred`, size_t *`amtread`*) [Function]

This is called by other filesystem routines to read or write files, and extends them automatically, if necessary. *np* is the node to be read or written, and must be locked. *data* will be written or filled. *off* identifies where in the file the I/O is to take place (negative values are not allowed). *amt* is the size of *data* and tells how much to copy. *dir* is zero for reading or nonzero for writing. *cred* is the user doing the access (only used to validate attempted file extension). For reads, `*amtread` is filled with the amount actually read.

`void diskfs_notice_dirchange` (*struct node *`dp`, enum dir_changed_type `type`, char *`name`*) [Function]

Send notifications to users who have requested them for directory *dp* with `dir_notice_changes`. The type of modification and affected name are *type* and *name* respectively. This should be called by `diskfs_direnter`, `diskfs_dirremove`, `diskfs_dirrewrite`, and anything else that changes the directory, after the change is fully completed.

`struct node * diskfs_make_node` (*struct disknode *`dn`*) [Function]

Create a new node structure with *ds* as its physical disknode. The new node will have one hard reference and no light references.

These next node manipulation functions are not generally useful, but may come in handy if you need to redefine any diskfs functions.

`error_t diskfs_create_node` (*struct node *`dir`, char *`name`, mode_t `mode`, struct node **`newnode`, struct protid *`cred`, struct dirstat *`ds`*) [Function]

Create a new node. Give it *mode*: if *mode* includes `IFDIR`, also initialize '`.`' and '`..`' in the new directory. Return the node in *npp*. *cred* identifies the user responsible for the call. If *name* is nonzero, then link the new node into *dir* with name *name*; *ds* is the result of a prior `diskfs_lookup` for creation (and *dir* has been held locked since). *dir* must always be provided as at least a hint for disk allocation strategies.

`void diskfs_set_node_atime` (*struct node *np*) [Function]

If disk is not readonly and the noatime option is not enabled, set `np->dn_set_atime`.

`void diskfs_set_node_times` (*struct node *np*) [Function]

If `np->dn_set_ctime` is set, then modify `np->dn_stat.st_ctime` appropriately; do the analogous operations for atime and mtime as well.

`struct node * diskfs_check_lookup_cache` (*struct node *dir, char *name*) [Function]

Scan the cache looking for *name* inside *dir*. If we don't know any entries at all, then return zero. If the entry is confirmed to not exist, then return -1. Otherwise, return *np* for the entry, with a newly-allocated reference.

`error_t diskfs_cached_lookup` (*int cache_id, struct node **npp*) [Function]

Return the node corresponding to *cache_id* in `*npp`.

`void diskfs_enter_lookup_cache` (*struct node *dir, struct node *np, char *name*) [Function]

Node *np* has just been found in *dir* with *name*. If *np* is null, that means that this name has been confirmed as absent in the directory.

`void diskfs_purge_lookup_cache` (*struct node *dp, struct node *np*) [Function]

Purge all references in the cache to *np* as a node inside directory *dp*.

8.5.5 Diskfs Callbacks

Like several other Hurd libraries, `libdiskfs` depends on you to implement application-specific callback functions. You *must* define the following functions and variables, but you should also look at Section 8.5.6 [Diskfs Options], page 65, as there are several defaults which should be modified to provide good filesystem support:

`struct dirstat` [Structure]

You must define this type, which will hold information between a call to `diskfs_lookup` and a call to one of `diskfs_direnter`, `diskfs_dirremove`, or `diskfs_dirrewrite`. It must contain enough information so that those calls work as described below.

`const size_t` *diskfs_dirstat_size* [Variable]

This must be the size in bytes of a `struct dirstat`.

`int diskfs_link_max` [Variable]

This is the maximum number of links to any one file, which must be a positive integer. The implementation of `dir_rename` does not know how to succeed if this is only one allowed link; on such formats you need to reimplement `dir_rename` yourself.

`int diskfs_maxsymlinks` [Variable]

This variable is a positive integer which is the maximum number of symbolic links which can be traversed within a single call to `dir_lookup`. If this is exceeded, `dir_lookup` will return `ELOOP`.

`struct node * diskfs_root_node` [Variable]

Set this to be the node of the root of the filesystem.

`char * diskfs_server_name` [Variable]

Set this to the name of the filesystem server.

`char * diskfs_server_version` [Variable]

Set this to be the server version string.

`char * diskfs_disk_name` [Variable]

This should be a string that somehow identifies the particular disk this filesystem is interpreting. It is generally only used to print messages or to distinguish instances of the same filesystem type from one another. If this filesystem accesses no external media, then define this to be zero.

`error_t diskfs_set_statfs` (*fsys_statfsbuf_t *statfsbuf*) [Function]

Set `*statfsbuf` with appropriate values to reflect the current state of the filesystem.

`error_t diskfs_lookup` (*struct node *dp, char *name, enum lookup_type type, struct node **np, struct dirstat *ds, struct protid *cred*) [Function]

`error_t diskfs_lookup_hard` (*struct node *dp, char *name, enum lookup_type type, struct node **np, struct dirstat *ds, struct protid *cred*) [Function]

You should not define `diskfs_lookup`, because it is simply a wrapper for `diskfs_lookup_hard`, and is already defined in `libdiskfs`.

Lookup in directory *dp* (which is locked) the name *name*. *type* will either be `LOOKUP`, `CREATE`, `RENAME`, or `REMOVE`. *cred* identifies the user making the call.

If the name is found, return zero, and (if *np* is nonzero) set `*np` to point to the node for it, which should be locked. If the name is not found, return `ENOENT`, and (if *np* is nonzero) set `*np` to zero. If *np* is zero, then the node found must not be locked, not even transitorily. Lookups for `REMOVE` and `RENAME` (which must often check permissions on the node being found) will always set *np*.

If *ds* is nonzero then the behaviour varies depending on the requested lookup *type*:

`LOOKUP` Set `*ds` to be ignored by `diskfs_drop_dirstat`

`CREATE` On success, set `*ds` to be ignored by `diskfs_drop_dirstat`.
On failure, set `*ds` for a future call to `diskfs_direnter`.

`RENAME` On success, set `*ds` for a future call to `diskfs_dirrewrite`.
On failure, set `*ds` for a future call to `diskfs_direnter`.

`REMOVE` On success, set `*ds` for a future call to `diskfs_dirremove`.
On failure, set `*ds` to be ignored by `diskfs_drop_dirstat`.

The caller of this function guarantees that if *ds* is nonzero, then either the appropriate call listed above or `diskfs_drop_dirstat` will be called with *ds* before the directory *dp* is unlocked, and guarantees that no lookup calls will be made on this directory between this lookup and the use (or destruction) of *DS.

If you use the library's versions of `diskfs_rename_dir`, `diskfs_clear_directory`, and `diskfs_init_dir`, then lookups for '`..`' might have the flag `SPEC_DOTDOT` ORed in. This has a special meaning depending on the requested lookup *type*:

`LOOKUP` *dp* should be unlocked and its reference dropped before returning.

`CREATE` Ignore this case, because `SPEC_DOTDOT` is guaranteed not to be given.

`RENAME`
`REMOVE` In both of these cases, the node being found (`*`*np*) is already held locked, so don't lock it or add a reference to it.

Return `ENOENT` if *name* isn't in the directory. Return `EAGAIN` if *name* refers to the '`..`' of this filesystem's root. Return `EIO` if appropriate.

`error_t diskfs_direnter` (*struct node *`dp`, *char *`name`, *struct node *`np`, *struct dirstat *`ds`, *struct protid *`cred`) [Function]
`error_t diskfs_direnter_hard` (*struct node *`dp`, *char *`name`, *struct node *`np`, *struct dirstat *`ds`, *struct protid *`cred`) [Function]

You should not define `diskfs_direnter`, because it is simply a wrapper for `diskfs_direnter_hard`, and is already defined in `libdiskfs`.

Add *np* to directory *dp* under the name *name*. This will only be called after an unsuccessful call to `diskfs_lookup` of type `CREATE` or `RENAME`; *dp* has been locked continuously since that call and *ds* is as that call set it, *np* is locked. *cred* identifies the user responsible for the call (to be used only to validate directory growth).

`error_t diskfs_dirrewrite` (*struct node *`dp`, *struct node *`oldnp`, *struct node *`np`, *char *`name`, *struct dirstat *`ds`) [Function]
`error_t diskfs_dirrewrite_hard` (*struct node *`dp`, *struct node *`np`, *struct dirstat *`ds`) [Function]

You should not define `diskfs_dirrewrite`, because it is simply a wrapper for `diskfs_dirrewrite_hard`, and is already defined in `libdiskfs`.

This will only be called after a successful call to `diskfs_lookup` of type `RENAME`; this call should change the name found in directory *dp* to point to node *np* instead of its previous referent. *dp* has been locked continuously since the call to `diskfs_lookup` and *ds* is as that call set it; *np* is locked.

`diskfs_dirrewrite` has some additional specifications: *name* is the name within *dp* which used to correspond to the previous referent, *oldnp*; it is this reference which is being rewritten. `diskfs_dirrewrite` also calls `diskfs_notice_dirchange` if `dp->dirmod_reqs` is nonzero.

`error_t diskfs_dirremove` (*struct node *`dp`, *struct node *`np`, *char *`name`, *struct dirstat *`ds`) [Function]
`error_t diskfs_dirremove_hard` (*struct node *`dp`, *struct dirstat *`ds`) [Function]

You should not define `diskfs_dirremove`, because it is simply a wrapper for `diskfs_dirremove_hard`, and is already defined in `libdiskfs`.

This will only be called after a successful call to `diskfs_lookup` of type `REMOVE`; this call should remove the name found from the directory *ds*. *dp* has been locked continuously since the call to `diskfs_lookup` and *ds* is as that call set it.

`diskfs_dirremove` has some additional specifications: this routine should call `diskfs_notice_dirchange` if *dp*`->dirmod_reqs` is nonzero. The entry being removed has name *name* and refers to *np*.

`error_t diskfs_drop_dirstat` (*struct node *`dp`*, *struct dirstat *`ds`*) [Function]

ds has been set by a previous call to `diskfs_lookup` on directory *dp*; this function is guaranteed to be called if `diskfs_direnter`, `diskfs_dirrewrite`, and `diskfs_dirremove` have not been called, and should free any state retained by a `struct dirstat`. *dp* has been locked continuously since the call to `diskfs_lookup`.

`void diskfs_null_dirstat` (*struct dirstat *`ds`*) [Function]

Initialize *ds* such that `diskfs_drop_dirstat` will ignore it.

`error_t diskfs_get_directs` (*struct node *`dp`*, *int `entry`*, *int `n`*, *char **`data`*, *u_int *`datacnt`*, *vm_size_t `bufsiz`*, *int *`amt`*) [Function]

Return *n* directory entries starting at *entry* from locked directory node *dp*. Fill `*data` with the entries; which currently points to `*datacnt` bytes. If it isn't big enough, `vm_allocate` into `*data`. Set `*datacnt` with the total size used. Fill *amt* with the number of entries copied. Regardless, never copy more than *bufsiz* bytes. If *bufsiz* is zero, then there is no limit on `*datacnt`; if *n* is -1, then there is no limit on *amt*.

`int diskfs_dirempty` (*struct node *`dp`*, *struct protid *`cred`*) [Function]

Return nonzero if locked directory *dp* is empty. If the user has not redefined `diskfs_clear_directory` and `diskfs_init_directory`, then 'empty' means 'only possesses entries labelled '`.`' and '`..`'. *cred* identifies the user making the call... if this user cannot search the directory, then this routine should fail.

`error_t diskfs_get_translator` (*struct node *`np`*, *char **`namep`*, *u_int *`namelen`*) [Function]

For locked node *np* (for which `diskfs_node_translated` is true) look up the name of its translator. Store the name into newly malloced storage and set `*namelen` to the total length.

`error_t diskfs_set_translator` (*struct node *`np`*, *char *`name`*, *u_int `namelen`*, *struct protid *`cred`*) [Function]

For locked node *np*, set the name of the translating program to be *name*, which is *namelen* bytes long. *cred* identifies the user responsible for the call.

`error_t diskfs_truncate` (*struct node *`np`*, *off_t `size`*) [Function]

Truncate locked node *np* to be *size* bytes long. If *np* is already less than or equal to *size* bytes long, do nothing. If this is a symlink (and `diskfs_shortcut_symlink` is set) then this should clear the symlink, even if `diskfs_create_symlink_hook` stores the link target elsewhere.

`error_t diskfs_grow` (*struct node *`np`*, *off_t `size`*, *struct protid *`cred`*) [Function]

Grow the disk allocated to locked node *np* to be at least *size* bytes, and set *np*`->allocsize` to the actual allocated size. If the allocated size is already *size* bytes, do nothing. *cred* identifies the user responsible for the call.

`error_t diskfs_node_reload` (*struct node *`node`*) [Function]

This function must reread all data specific to *node* from disk, without writing anything. It is always called with *diskfs_readonly* set to true.

`error_t diskfs_reload_global_state` (*void*) [Function]

This function must invalidate all cached global state, and reread it as necessary from disk, without writing anything. It is always called with *diskfs_readonly* set to true. `diskfs_node_reload` is subsequently called on all active nodes, so this call doesn't need to reread any node-specific data.

`error_t diskfs_node_iterate` (*error_t* (*`fun`) (*struct node *`np`*)) [Function]

For each active node *np*, call *fun*. The node is to be locked around the call to *fun*. If *fun* returns nonzero for any node, then stop immediately, and return that value.

`error_t diskfs_alloc_node` (*struct node *`dp`*, *mode_t* `mode`, *struct node **`np`*) [Function]

Allocate a new node to be of mode *mode* in locked directory *dp*, but don't actually set the mode or modify the directory, since that will be done by the caller. The user responsible for the request can be identified with *cred*. Set `*np` to be the newly allocated node.

`void diskfs_free_node` (*struct node *`np`*, *mode_t* `mode`) [Function]

Free node *np*; the on-disk copy has already been synchronized with `diskfs_node_update` (where `np->dn_stat.st_mode` was zero). *np*'s mode used to be *mode*.

`void diskfs_lost_hardrefs` (*struct node *`np`*) [Function]

Locked node *np* has some light references but has just lost its last hard reference.

`void diskfs_new_hardrefs` (*struct node *`np`*) [Function]

Locked node *np* has just acquired a hard reference where it had none previously. Therefore, it is okay again to have light references without real users.

`void diskfs_try_dropping_softrefs` (*struct node *`np`*) [Function]

Node *np* has some light references, but has just lost its last hard references. Take steps so that if any light references can be freed, they are. Both *diskfs_node_refcnt_lock* and *np* are locked. This function will be called after `diskfs_lost_hardrefs`.

`void diskfs_node_norefs` (*struct node *`np`*) [Function]

Node *np* has no more references; free local state, including `*np` if it shouldn't be retained. *diskfs_node_refcnt_lock* is held.

`error_t diskfs_set_hypermetadata` (*int* `wait`, *int* `clean`) [Function]

Write any non-paged metadata from format-specific buffers to disk, asynchronously unless *wait* is nonzero. If *clean* is nonzero, then after this is written the filesystem will be absolutely clean, and it must be possible for the non-paged metadata to indicate that fact.

`void diskfs_write_disknode` (*struct node *`np`*, *int* `wait`) [Function]

Write the information in `np->dn_stat` and any associated format-specific information to the disk. If *wait* is true, then return only after the physical media has been completely updated.

`void diskfs_file_update` (*struct node* `*np`, *int* `wait`) [Function]
: Write the contents and all associated metadata of file NP to disk. Generally, this will involve calling `diskfs_node_update` for much of the metadata. If *wait* is true, then return only after the physical media has been completely updated.

`mach_port_t diskfs_get_filemap` (*struct node* `*np`, *vm_prot_t* `prot`) [Function]
: Return a memory object port (send right) for the file contents of *np*. *prot* is the maximum allowable access. On errors, return `MACH_PORT_NULL` and set `errno`.

`struct pager * diskfs_get_filemap_pager_struct` (*struct node* `*np`) [Function]
: Return a `struct pager *` that refers to the pager returned by diskfs_get_filemap for locked node NP, suitable for use as an argument to `pager_memcpy`.

`vm_prot_t diskfs_max_user_pager_prot` (*void*) [Function]
: Return the bitwise OR of the maximum `prot` parameter (the second argument to `diskfs_get_filemap`) for all active user pagers.

`int diskfs_pager_users` (*void*) [Function]
: Return nonzero if there are pager ports exported that might be in use by users. Further pager creation should be blocked before this function returns zero.

`void diskfs_sync_everything` (*int* `wait`) [Function]
: Sync all the pagers and write any data belonging on disk except for the hypermetadata. If *wait* is true, then return only after the physical media has been completely updated.

`void diskfs_shutdown_pager` (*void*) [Function]
: Shut down all pagers. This is irreversible, and is done when the filesystem is exiting.

8.5.6 Diskfs Options

The functions and variables described in this subsection already have default definitions in `libdiskfs`, so you are not forced to define them; rather, they may be redefined on a case-by-case basis.

You should set the values of any option variables as soon as your program starts (before you make any calls to diskfs, such as argument parsing).

`int diskfs_hard_readonly` [Variable]
: You should set this variable to nonzero if the filesystem media can never be made writable.

`char * diskfs_extra_version` [Variable]
: Set this to be any additional version specification that should be printed for –version.

`int diskfs_shortcut_symlink` [Variable]
: This should be nonzero if and only if the filesystem format supports shortcutting symbolic link translation. The library guarantees that users will not be able to read or write the contents of the node directly, and the library will only do so if the symlink hook functions (`diskfs_create_symlink_hook` and `diskfs_read_symlink_hook`) return `EINVAL` or are not defined. The library knows that the `dn_stat.st_size` field is the length of the symlink, even if the hook functions are used.

`int diskfs_shortcut_chrdev` [Variable]
`int diskfs_shortcut_blkdev` [Variable]
`int diskfs_shortcut_fifo` [Variable]
`int diskfs_shortcut_ifsock` [Variable]

These variables should be nonzero if and only if the filesystem format supports shortcutting character device node, block device node, FIFO, or Unix-domain socket translation, respectively.

`int diskfs_default_sync_interval` [Variable]

`diskfs_set_sync_interval` is called with this value when the first diskfs thread is started up (in `diskfs_spawn_first_thread`). This variable has a default default value of 30, which causes disk buffers to be flushed at least every 30 seconds.

`error_t diskfs_validate_mode_change` (*struct node* *`*np`*, *mode_t* *`mode`*) [Function]
`error_t diskfs_validate_owner_change` (*struct node* *`*np`*, *uid_t* *`uid`*) [Function]
`error_t diskfs_validate_group_change` (*struct node* *`*np`*, *gid_t* *`gid`*) [Function]
`error_t diskfs_validate_author_change` (*struct node* *`*np`*, *uid_t* *`author`*) [Function]
`error_t diskfs_validate_flags_change` (*struct node* *`*np`*, *int* *`flags`*) [Function]
`error_t diskfs_validate_rdev_change` (*struct node* *`*np`*, *dev_t* *`rdev`*) [Function]

Return zero if for the node *np* can be changed as requested. That is, if *np*'s mode can be changed to *mode*, owner to *uid*, group to *gid*, author to *author*, flags to *flags*, or raw device number to *rdev*, respectively. Otherwise, return an error code.

It must always be possible to clear the mode or the flags; diskfs will not ask for permission before doing so.

`void diskfs_readonly_changed` (*int* *`readonly`*) [Function]

This is called when the disk has been changed from read-only to read-write mode or vice-versa. *readonly* is the new state (which is also reflected in *diskfs_readonly*). This function is also called during initial startup if the filesystem is to be writable.

`error_t (*diskfs_create_symlink_hook)` (*struct node* *`*np`*, *char* *`*target`*) [Variable]

If this function pointer is nonzero (and `diskfs_shortcut_symlink` is set) it is called to set a symlink. If it returns `EINVAL` or isn't set, then the normal method (writing the contents into the file data) is used. If it returns any other error, it is returned to the user.

`error_t (*diskfs_read_symlink_hook)` (*struct node* *`*np`*, *char* *`*target`*) [Variable]

If this function pointer is nonzero (and `diskfs_shortcut_symlink` is set) it is called to read the contents of a symlink. If it returns `EINVAL` or isn't set, then the normal method (reading from the file data) is used. If it returns any other error, it is returned to the user.

`error_t diskfs_rename_dir` (*struct node* **fdp*, *struct node* **fnp*, *char* **fromname*, *struct node* **tdp*, *char* **toname*, *struct protid* **fromcred*, *struct protid* **tocred*) [Function]

Rename directory node *fnp* (whose parent is *fdp*, and which has name *fromname* in that directory) to have name *toname* inside directory *tdp*. None of these nodes are locked, and none should be locked upon return. This routine is serialized, so it doesn't have to be reentrant. Directories will never be renamed except by this routine. *fromcred* is the user responsible for *fdp* and *fnp*. *tocred* is the user responsible for *tdp*. This routine assumes the usual convention where '`.`' and '`..`' are represented by ordinary links; if that is not true for your format, you have to redefine this function.

`error_t diskfs_clear_directory` (*struct node* **dp*, *struct node* **pdp*, *struct protid* **cred*) [Function]

Clear the '`.`' and '`..`' entries from directory *dp*. Its parent is *pdp*, and the user responsible for this is identified by *cred*. Both directories must be locked. This routine assumes the usual convention where '`.`' and '`..`' are represented by ordinary links; if that is not true for your format, you have to redefine this function.

`error_t diskfs_init_dir` (*struct node* **dp*, *struct node* **pdp*, *struct protid* **cred*) [Function]

Locked node *dp* is a new directory; add whatever links are necessary to give it structure; its parent is the (locked) node *pdp*. This routine may not call `diskfs_lookup` on *pdp*. The new directory must be clear within the meaning of `diskfs_dirempty`. This routine assumes the usual convention where '`.`' and '`..`' are represented by ordinary links; if that is not true for your format, you have to redefine this function. *cred* identifies the user making the call.

8.5.7 Diskfs Internals

The library also exports the following functions, but they are not generally useful unless you are redefining other functions the library provides.

`error_t diskfs_create_protid` (*struct peropen* **po*, *struct iouser* **user*, *struct protid* ***cred*) [Function]

Create and return a protid for an existing peropen *po* in *cred*, referring to user *user*. The node `po->np` must be locked.

`error_t diskfs_start_protid` (*struct peropen* **po*, *struct protid* ***cred*) [Function]

Build and return in *cred* a protid which has no user identification, for peropen *po*. The node `po->np` must be locked.

`void diskfs_finish_protid` (*struct protid* **cred*, *struct iouser* **user*) [Function]

Finish building protid *cred* started with `diskfs_start_protid`; the user to install is *user*.

`void diskfs_protid_rele` (*void* **arg*) [Function]

Called when a protid *cred* has no more references. Because references to protids are maintained by the port management library, this is installed in the clean routines list. The ports library will free the structure.

`struct peropen * diskfs_make_peropen` (*struct node *np*, *int flags*, *struct peropen *context*) [Function]

Create and return a new peropen structure on node *np* with open flags *flags*. The initial values for the `root_parent`, `shadow_root`, and `shadow_root_parent` fields are copied from *context* if it is nonzero, otherwise each of these values are set to zero.

`void diskfs_release_peropen` (*struct peropen *po*) [Function]

Decrement the reference count on *po*.

`error_t diskfs_execboot_fsys_startup` (*mach_port_t port*, *int flags*, *mach_port_t ctl*, *mach_port_t *real*, *mach_msg_type_name_t *realpoly*) [Function]

This function is called by `S_fsys_startup` for execserver bootstrap. The execserver is able to function without a real node, hence this fraud. Arguments are as for `fsys_startup` in `<hurd/fsys.defs>`.

`int diskfs_demuxer` (*mach_msg_header_t *inp*, *mach_msg_header_t *outp*) [Function]

Demultiplex incoming `libports` messages on diskfs ports.

The diskfs library also provides functions to demultiplex the fs, io, fsys, interrupt, and notify interfaces. All the server routines have the prefix `diskfs_S_`. For those routines, `in` arguments of type `file_t` or `io_t` appear as `struct protid *` to the stub.

9 Twisted Filesystems

In the Hurd, translators are capable of redirecting filesystem requests to other translators, which makes it possible to implement alternative views of the same underlying data. The translators described in this chapter do not provide direct access to any data; rather, they are organizational tools to help you simplify an existing physical filesystem layout.

Be prudent with these translators: you may accidentally injure people who want their filesystems to be rigidly tree-structured.[1]

FIXME: finish

9.1 symlink, firmlink

9.2 hostmux, usermux

9.3 shadowfs

[1] You are lost in a maze of twisty little filesystems, all alike. . . .

10 Distributed Filesystems

Distributed filesystems are designed to share files between separate machines via a network connection of some sort. Their design is significantly different than stored filesystems (see Chapter 8 [Stored Filesystems], page 55): they need to deal with the problems of network delays and failures, and may require complex authentication and replication protocols involving multiple file servers.

10.1 File Transfer Protocol

FIXME: finish

10.1.1 ftpcp, ftpdir

10.1.2 ftpfs

10.1.3 FTP Connection Library

FIXME: finish

10.2 Network File System

FIXME: finish

10.2.1 nfsd

10.2.2 nfs

11 Networking

FIXME: this subsystem is in flux

11.1 pfinet

11.2 pflocal

11.3 libpipe

11.4 Socket Interface

FIXME: net frobbing stuff may be added to socket.defs

12 Terminal Handling

FIXME: finish

12.1 term

12.2 term.defs

13 Running Programs

FIXME: finish

13.1 ps, w

13.2 libps

13.3 exec

13.4 proc

13.5 crash

14 Authentication

FIXME: finish

14.1 addauth, rmauth, setauth

14.2 su, sush, unsu

14.3 login, loginpr

14.4 auth

14.5 Auth Interface

FIXME: finish

14.5.1 Auth Protocol

FIXME: finish

Index

(

/

A

C

D

L

M

N

P

U

X

Z

Short Contents

Table of Contents