Porting the GNU Hurd to the L4 Microkernel

Marcus Brinkmann

August 2003

Contents

Chapter 1

Introduction

> *What is right in this particular case, like everything else, requires to be explained.*
>
> *Republic V* by Plato

1.1 Genesis

The GNU Hurd is a multi-user, time sharing, general purpose, network operating system. The Hurd's existence is motivated by perceived design flaws in Unix and other operating systems: either the system is overly restrictive and does not allow the user to perform interesting operations without the intervention of the administrator (e.g. mount file systems) or the security policy is overly lax and users can harm each other and the system. The Hurd emphasizes flexibility and security. The fundamental philosophy is:

> *The operating system should empower users while maintaining strict system security.*

Speed, although very important, is secondary to correctness and security. We have however reason to believe that we have identified important areas where the Hurd, due to its design, will not only be able to compete with traditional systems, but outperform them.

In order to achieve this goal, a multi-server architecture has been embraced. The initial prototype of the Hurd runs on a derivative of the Mach microkernel developed at Carnegie Mellon University in the 1980s and early 1990s. With this implementation, an authentication scheme similar to Kerberos was explored which separates the user identity from the process thereby allowing tasks to identify themselves not by an inherent attribute but using unforgeable identity

tokens. User space file systems and a user space virtual file system, VFS, allowed users to mount their own file systems including NFS and to create their own special file systems such as ftpfs without needing special permissions on the system and without harming other users. This was based on the observation that the only reason that users are not permitted to mount file systems in Unix is that it involves twiddling kernel data structures: with the VFS outside of the kernel, this was no longer an impedance.

During this implementation, much was learned including: Mach did not remove enough policy from the kernel and as a result, its mechanisms were still too heavy-weight. Mach moves what has now become POSIX from the Unix kernel into user space and only provides IPC, a scheduler, memory management and device drivers. This design leaves the resource allocation and management schemes in the kernel while the resource utilization was moved completely into user space (e.g. file systems). This made important information about resource utilization inaccessible to the allocation mechanisms and thus made smart implementations of resource managers extremely difficult and far less intelligent than their monolithic kernel counterparts. In keeping with the Hurd philosophy of empowering the user, it was observed that many applications could profit if they could control how the resources they are using are managed, e.g. which pages are evicted when there is memory pressure. This is not only an improvement over the Mach scheme, but also over a traditional Unix-like design: applications not only know how a resource is being used but also what its contents are. Unix knows how a frame of memory is allocated, e.g. to the block cache, but it does not know what its contents are nor the classes of data and the type of expected usage patterns of the data types contained therein. This scheme should permit an application to make far more intelligent decisions than are possible with the superficial knowledge that a monolithic kernel has.

The L4 microkernel makes implementing this philosophy possible: it aims to absolutely minimize the amount of policy in the microkernel while providing powerful *foundational* mechanisms for inter-process communication, memory movement (mapping and granting of pages via address space manipulation) and task and thread creation and manipulation.

Thus, while the L4 microkernel tries to minimize the policy that the kernel enforces on the software running on it, the Hurd tries to minimize the policy that the operating system enforces on its users. The Hurd also aims to provide a POSIX conformant, general purpose layer. This POSIX personality of the Hurd, however, is provided for convenience only and to make the Hurd useful: many applications target a subset of POSIX. Other personalities can be implemented and used in parallel. This default personality of the Hurd is not sandboxed: it provides convenient features that allow the user to extend the system so that all POSIX compatible programs can take advantage of them.

1.2 Work In Progress

This manual is less a manual than a series of notes about the effort to document the current strategy to port the Hurd to the L4 microkernel.

> Remarks about the history of a certain feature and implementation details are set in a smaller font and separated from the main text, just like this paragraph. Because this is work in progress, there are naturally a lot of such comments.

The port to L4 was set into action in the summer of 2002 when Neal H. Walfield went to the Universität of Karlsruhe. During that time, he worked with the L4 group and designed a basic IPC, Task and Thread API as well as doing extensive design work on the virtual memory manager. He was aided greatly by discussions with Marcus Brinkmann, Uwe Dannowski, Kevin Elphinstone, Andreas Haeberlen, Wolfgang Jährling, Joshua LeVasseur, Espen Skoglund, Volkmar Uhlig and Marcus Völp.

A public release of L4 was made in May of 2003. It was soon after this that Marcus Brinkmann began overhauling the proposed IPC system and identifying important flaws and scenarios that had been originally overlooked. He also revised the fork and exec strategy and began extensive work on the rest of the system.

Peter De Schrijver and Daniel Wagner started to design the device driver framework.

Niels Müller was the first one to realize that the exec server can be eliminated and gave helpful input on several aspects of the task server and IPC design.

During this process valuable input and discussion has come from many different corners including:

Chapter 2

Booting

A multiboot-compliant bootloader, for example GNU GRUB, loads the loader program `laden`, the kernel, σ_0, the rootserver and further modules. The loader is started, patches the kernel interface page, and starts the kernel. The kernel starts σ_0 and the rootserver. The rootserver has to deal with the other modules.

2.1 System bootstrap

The initial part of the boot procedure is system specific.

2.1.1 Booting the ia32

On the ia32, the BIOS will be one of the first things to run. Eventually, the BIOS will start the bootloader. The Hurd requires a multiboot-compliant bootloader, such as GNU GRUB. A typical configuration file entry in the `menu.list` file of GNU GRUB will look like this:

```
title = The GNU Hurd on L4
root = (hd0,0)
kernel = /boot/laden
module = /boot/ia32-kernel
module = /boot/sigma0
module = /boot/wortel
module = /boot/physmem
module = /boot/task
module = /boot/deva
module = /boot/deva-drivers
module = /boot/rootfs
```

GNU GRUB loads the binary image files into memory and jumps to the entry point of `laden`.

2.2 The loader `laden`

`laden` is a multiboot compliant kernel from the perspective of GNU GRUB. It expects at least three modules. The first module is the L4 kernel image, the second module is the σ_0 server image, and the third module is the rootserver image.

> Later, the L4 kernel will support the optional UTCB paging server σ_1, which has to be treated like the other initial servers by `laden`. A command line option to `laden` will allow the user to specify if the third module is the rootserver or σ_1. If σ_1 is used, the rootserver is the fourth module in the list.

`laden` copies (or moves) the three executable images to the right location in memory, according to their respective ELF headers. It also initializes the BSS section to zero.

> Laden has to deal with overlapping source and destination memory areas in an intelligent way. It currently will detect such situations, but is not always able to find a solution, even if one exists.
> If a memory area stretches out to the very last page addressible in 32 bit, the high address of the memory descriptor will overflow. This is in fact the behaviour of `kickstart`. `laden` currently truncates such an area by one page. This needs clarification in the L4 standard.

Then it searches for the kernel interface page (KIP) in the L4 kernel image and modifies it in the following way:

- The memory descriptors are filled in according to the memory layout of the system. On ia32, this information is – at least partially – provided by GNU GRUB.

  > GNU GRUB seems to omit information about the memory that is shared with the VGA card. `laden` creates a special entry for that region, overriding any previous memory descriptor.

- The start and end addresses and the entry point of the initial servers are filled in.

  > A future version of L4 should support adding information about the UTCB area of the initial rootserver as well. Until then, the rootserver has no clean way to create a new thread (a hack is used by the rootserver to calculate the UTCB addresses for other threads).

- The `boot_info` field is initialized.

  > The `boot_info` field is currently set to the GNU GRUB `multiboot_info` structure. This only works for the ia32 architecture of course. We might want to have a more architecture independent way to pass the information about further modules to the rootserver. We also might want to gather the information provided by GNU GRUB in a single page (if it is not).

2.3 The L4 kernel

The L4 kernel initializes itself and then creates the address spaces and threads for the initial servers σ_0 and the rootserver. It maps all physical memory idempotently into σ_0, and sets the pager of the rootserver thread to σ_0. Then it starts the initial servers.

2.4 The initial server σ_0

σ_0 acts as the pager for the rootserver, answering page fault messages by mapping the page at the fault address idempotently in the rootserver.

> σ_0 can also be used directly by sending messages to it, according to the σ_0 RPC protocol. This is used by the kernel to allocate reserved memory, but can also be used by the user to explicitely allocate more memory than single pages indirectly via page faults.

The thread ID of σ_0 is (`UserBase`, 1).

> We will write all thread IDs in the form (`thread nr`, `version`).

Any fpage will only be provided to one thread. σ_0 will return an error if another thread attempts to map or manipulate an fpage that has already been given to some other thread, even if both threads reside in the same address space.

2.5 The initial server σ_1

σ_1 is intended to provide a paging service for UTCB memory. This will allow orthogonal persistence to be implemented. It is not yet supported.

The thread ID of σ_1 is (`UserBase` + 1, 1).

2.6 The rootserver wortel

The rootserver that L4 started is the only task in the system which threads can perform privileged system calls. So the rootserver must provide wrappers for the system calls to other unprivileged system tasks.

> For this, a simple authentication scheme is required. The rootserver can keep a small, statically allocated table of threads which are granted access to the system call wrappers. The caller could provide the index in the table for fast O(1) lookup instead linear search. Threads with access could be allowed to add other threads or change existing table entries. The same scheme can be used in the device driver framework.
>
> The rootserver should have one thread per CPU, and run at a high priority.

Our rootserver is called wortel, and also bootstraps the operating system. Wortel thus acts as a simple manager OS and as a bootloader program.

> Ideally, there would be a real manager OS on top of L4 in which you can run different sand-boxed operating systems. Wortel implements only some rudimentary features such a system would provide: Access to the system memory and execution of privileged L4 system calls.
>
> If you had such a real manager OS, then this manager OS would start a bootloader to boot up a sand-boxed operating system. For simplicity, wortel currently implements such a bootloader for the Hurd system. Eventually, the code should be split to allow both components to develop independently.

The rootserver has the following initial state:

- Its thread ID is (`UserBase` + 2, 1).
- The priority is set to the 255, the maximum value.

  > The rootserver, or at least the system call wrapper, should run at a very high priority.

- The instruction pointer `%eip` is set to the entry point, all other registers are undefined (including the stack pointer).
- The pager is set to σ_0.
- The exception handler is set to `nilthread`.
- The scheduler is set to the rootserver thread itself.

So the first thing the rootserver has to do is to set up a simple stack.

Then the rootserver should evaluate the `boot_info` field in the KIP to find the information about the other modules. It should parse the information and create the desired initial tasks of the operating system. The Hurd uses a boot script syntax to allow to pass information about other initial tasks and the root tasks to each initial task in a generalized manner.

> The exact number and type of initial tasks necessary to boot the Hurd are not yet known. Chances are that this list includes the `task` server, the physical memory server, the device servers, and the boot filesystem. The boot filesystem might be a small simple filesystem, which also includes the device drivers needed to access the real root filesystem.

2.7 The physical memory server physmem

The physical memory server is the first component of the actual Hurd system that is started (wortel serves as a manager OS in the background, and its presence is of no relevance to Hurd programs other than the fundamental core servers described in this chapter). It provides memory management routines that allow tasks in the Hurd system to be self-paged.

The rootserver moves the physical memory server executable image to its ELF load address (and initializes the BSS section to zero), creates a new address space and several threads in this address space, starts the first thread and then maps all the fpages covering the executable image 1:1 into the address space at the first pagefault (the fpage on which the thread faulted is mapped last - this makes the thread fault repeatedly until the whole image is mapped).

> Wortel should follow the `exec()` protocol to startup the new task as closely as possible. However, there is little that wortel can provide to physmem in this terms.

So, the physical memory server runs on mapped memory in its own address space, but the virtual addresses of its executable image coincede with the physical addresses.

Then, in a private protocol between wortel and physmem, the following happens:

1. Physmem requests all system memory from wortel. Wortel maps the memory from σ_0 and maps it to physmem.

   > The memory is mapped, not granted, to allow wortel (of which we think as a manager OS here) to unmap and recover the memory in case of a (possibly forced) system shutdown.

2. For each module that has not been used yet, wortel requests a capability in physmem that can be used to map in pages from the range of memory that the module occupies. These capabilities should implement the same pager interface that mappable files implement.

   > The idea is that these capabilities can be used in the `exec()` protocol to start up the tasks for these modules. If a module is not a task, the capability can be used to access the module data by mapping it into the address space like a file. Physmem can even swap out pages that back these objects on memory pressure.
   >
   > So, the physical memory server is in fact a simple filesystem for these initial tasks, usable only for mapping operations.
   >
   > Wortel can then start up the other tasks in the module list using the normal `exec()` protocol.

The result is that all tasks except for the rootserver can be started and manage their memory through physmem like normal Hurd tasks.

Later on, wortel will provide physmem with further information retrieved from the task and deva servers.

2.8 The task server

The task server is the second Hurd server started by wortel. Its responsibility is to keep track of allocation of task and thread IDs in the system, and manage related resources (recording and restricting CPU usage).

FIXME More has to be said here.

2.9 The device access server deva

The device access server deva is the third Hurd server started by wortel. It implements access to a low-level device driver framework in a way that transparently fits into the overall Hurd system. This means that access to device drivers is managed via capabilities, and that physmem containers are used for data exchange between a user-level application and a low-level device driver.

It also provides system integration services to the underlying low-level device driver framework. In particular, it intermediates access to privileged resources and provides device drivers and related data from the systems filesystem.

FIXME More has to be said here.

2.10 The device access server archive

The device access server needs to load device drivers before a root filesystem service is available. In particular, it needs to be able to provide device drivers for the root filesystem to the device driver framework.

The device access server archive is an archive of device drivers that is loaded by the bootloader and contains drivers necessary to run the root filesystem.

2.11 The root filesystem

The root filesystem is the fourth and last Hurd server started by wortel. After the root filesystem starts up and has exchanged the necessary bootstrap information with deva, it starts up the rest of the operating system services from its filesystem.

The root filesystem is the first program to actually run in a proper environment, given that it can access device drivers, task and physmem services.

> From the time the root filesystem starts up, the bootstrap continues roughly as it is implemented in the Hurd running on GNU Mach.

Chapter 3

Inter-process communication (IPC)

The Hurd requires a capability system. Capabilities are used to prove your identity to other servers (authentication), and access server-side implemented objects like devices, files, directories, terminals, and other things. The server can use a capability for whatever it wants. Capabilities provide interfaces. Interfaces can be invoked by sending messages to the capability. In L4, this means that a message is sent to a thread in the server providing the capability, with the identifier for the capability in the message.

Capabilities are protected objects. Access to a capability needs to be granted by the server. Once you have a capability, you can copy it to other tasks (if the server permits it, which is usually the case). In the Hurd, access to capabilities is always granted to a whole task, not to individual threads.

> There is no reason for the server not to permit it, because the holder of the capability could also just act as a proxy for the intended receiver instead copying the capability to it. The operation might fail anyway, for example because of resource shortage, in particular if the server puts a quota on the number of capabilities a user can hold.

Capabilities provide two essential services to the Hurd. They are used to restrict access to a server function, and they are the standard interface the components in the Hurd use to communicate with each others. Thus, it is important that their implementation is fast and secure.

> There are several ways to implement such a capability system. A more traditional design would be a global, trusted capability server that provides capabilities to all its users. The L4 redirector could be used to reroute all client traffic automatically through this server. This approach has several disadvantages:
>
> - It adds a lot of overhead to every single RPC, because all traffic has to be routed through the capability server, which must then perform the authentication on the server's behalf.

- It would be difficult to copy a capability to another task. Either the cap server would have to provide interfaces for clients to do it, or it would be have to know the message format for every interface and do it automatically.
- It would be a single point of failure. If it had a bug and crashed, the whole system would be affected.
- Users could not avoid it, it would be enforced system code.
- It is inflexible. It would be hard to replace or extend at run-time.

Another approach is taken by CORBA with IORs. IORs contain long random numbers which allow the server to identify a user of an object. This approach is not feasible for the following reasons:

- Even good random numbers can be guessed. Long enough random numbers can reduce the likelihood to arbitrary small numbers, though (below the probability of a hardware failure).
- Good random numbers are in short supply, and is slow to generate. Good pseudo random is faster, but it is still difficult to generate. The random number generator would become a critical part of the operating system.
- The random number had to be transfered in every single message. Because it would have to be long, it would have a significant negative impact on IPC performance.

The Hurd implements the capability system locally in each task. A common default implementation will be shared by all programs. However, a malicious untrusted program can do nothing to disturb the communication of other tasks. A capability is identified in the client task by the server thread and a local identifier (which can be different from client to client). The server thread will receive messages for the capabilities. The first argument in the message is the capability identifier. Although every task can get different IDs for the same capability, a well-behaving server will give the same ID to a client which already has a capability and gets the same capability from another client. So clients can compare capability IDs from the server numerically to check if two capabilities are the same, but only if one of the two IDs is received while the client already had the other one.

Because access to a capability must be restricted, the server needs to be careful in only allowing registered and known users to access the capability. For this, the server must be sure that it can determine the sender of a message. In L4, this is easy on the surface: The kernel provides the receiving thread with the sender's thread ID, which also contains the task ID in the version field. However, the server must also know for sure if this task is the same task that it gave access to the capability. Comparing the task IDs numerically is not good enough, the server must also somehow have knowledge or influence on how task IDs are reused when tasks die and are created.

The same is true for the client, of course, which trusts the server and thus must be sure that it is not tricked into trusting on unreliable data from an imposter, or sends sensitive data to it.

> The `task` server wants to reuse thread numbers because that makes best use of kernel memory. Reusing task IDs, the version field of a thread ID, is not so important, but there are only 14 bits for the version field (and the lower six bits must not be all zero). So a thread ID is bound to be reused eventually.
>
> Using the version field in a thread ID as a generation number is not good enough, because it is so small. Even on 64-bit architectures, where it is 32 bit long, it can eventually overflow.

The best way to prevent that a task can be tricked into talking to an imposter is to have the `task` server notify the task if the communication partner dies. The `task` server must guarantee that the task ID is not reused until all tasks that got such a notification acknowledge that it is processed, and thus no danger of confusion exists anymore.

The `task` server provides references to task IDs in form of *task info capabilities*. If a task has a task info capability for another task, it prevents that this other task's task ID is reused even if that task dies, and it also makes sure that task death notifications are delivered in that case.

> Because only the `task` server can create and destroy tasks, and assign task IDs, there is no need to hold such task info capabilities for the `task` server, nor does the `task` server need to hold task info capabilities for its clients. This avoids the obvious bootstrap problem in providing capabilities in the `task` server. This will even work if the `task` server is not the real `task` server, but a proxy task server (see section 4.2 on page 32).

As task IDs are a global resource, care has to be taken that this approach does not allow for a DoS-attack by exhausting the task ID number space, see section 4 on page 29 for more details.

3.1 Capabilities

This subsection contains implementation details about capabilities.

A server will usually operate on objects, and not capabilities. In the case of a filesystem, this could be file objects, for example.

> In the Hurd, filesystem servers have to keep different objects for each time a file is looked up (or "opened"), because some state, for example authentication, open flags and record locks, are associated not with the file directly, but with this instance of opening the file. Such a state structure ("credential") will also contain a pointer and reference to the actual file node. For simplicity, we will assume that the capability is associated with a file node directly.

To provide access to the object to another task, the server creates a capability, and associates it with the object (by setting a hook variable in the capability). From this capability, the server can either create send references to itself, or to other tasks. If the server creates send references for itself, it can use the capability just as it can use capabilities implemented by other servers. This makes access to locally and remotely implemented capabilities identical. If you

write code to work on capabilities, it can be used for remote objects as well as for local objects.

If the server creates a send reference for another task (a client), a new capability ID will be created for this task. This ID will only be valid for this task, and should be returned to the client.

The client itself will create a capability object from this capability ID. The capability will also contain information about the server, for example the server thread which should be used for sending messages to the capability.

If the client wants to send a message, it will send it to the provided server thread, and use the capability ID it got from the server as the first argument in the RPC. The server receives the message, and now has to look up the capability ID in the list of capabilties for this task.

> The server knows the task ID from the version field of the sender's thread ID. It can look up the list of capabilities for this task in a hash table. The capability ID can be an index into an array, so the server only needs to perform a range check. This allows to verify quickly that the user is allowed to access the object.
> This is not enough if several systems run in parallel on the same host. Then the version ID for the threads in the other systems will not be under the control of the Hurd's `task` server, and can thus not be trusted. The server can still use the version field to find out the task ID, which will be correct *if the thread is part of the same subsystem*. It also has to verify that the thread belongs to this subsystem. Hopefully the subsystem will be encoded in the thread ID. Otherwise, the `task` server has to be consulted (and, assuming that thread numbers are not shared by the different systems, the result can be cached).

The server reads out the capability associated with the capability ID, and invokes the server stub according to the message ID field in the message.

After the message is processed, the server sends it reply to the sender thread with a zero timeout.

> Servers must never block on sending messages to clients. Even a small timeout can be used for DoS-attacks. The client can always make sure that it receives the reply by using a combined send and receive operation together with an infinite timeout.

The above scheme assumes that the server and the client already have task info caps for the respective other task. This is the normal case, because acquiring these task info caps is part of the protocol that is used when a capability is copied from one task to another.

3.1.1 Bootstrapping a client-server connection

If the client and the server do not know about each other yet, then they can bootstrap a connection without support from any other task except the `task` server. The purpose of the initial handshake is to give both participants a chance to acquire a task info cap for the other participants task ID, so they can be sure that from there on they will always talk to the same task as they talked to before.

Preconditions The client knows the thread ID of the server thread that receives and processes the bootstrap messages. Some other task might hold a task info capability to the server the client wants to connect to.

> If no such other tasks exists, the protocol will still work. However, the client might not get a connection to the server that run at the time the client started the protocol, but rather to the server that run at the time the client acquired the task info cap for the server's task ID (after step 1 below).
>
> This is similar to how sending signals works in Unix: Technically, at the time you write `kill 203`, and press enter, you do not know if the process with the PID 203 you thought of will receive the signal, or some other process that got the PID in the time between you getting the information about the PID and writing the `kill`-command.

FIXME: Here should be the pseudo code for the protocol. For now, you have to take it out of the long version.

1. The client acquires a task info capability for the server's task ID, either directly from the `task` server, or from another task in a capability copy. From that point on, the client can be sure to always talk to the same task when talking to the server.

   Of course, if the client already has a task info cap for the server it does not need to do anything in this step.

   > As explained above, if the client does not have any other task holding the task info cap already, it has no secure information about what this task is for which it got a task info cap.

2. The client sends a message to the server, requesting the initial handshake.

3. The server receives the message, and acquires a task info cap for the client task (directly from the `task` server).

   Of course, if the server already has a task info cap for the client it does not need to do anything in this step.

   > At this point, the server knows that future messages from this task will come from the same task as it got the task info cap for. However, it does not know that this is the same task that sent the initial handshake request in step 2 above. This shows that there is no sense in verifying the task ID or perform any other authentication before acquiring the task info cap.

4. The server replies to the initial handshake request with an empty reply message.

   > Because the reply now can go to a different task than the request came from, sending the reply might fail. It might also succeed and be accepted by the task that replaced the requestor. Or it might succeed normally. The important thing is that it does not matter to the server at all. It would have provided the same "service" to the "imposter" of the client, if he had bothered to do the request. As no authentication is done yet, there is no point for the server to bother.

This means however, that the server needs to be careful in not consuming too many resources for this service. However, this is easy to achieve. Only one task info cap per client task will ever be held in the server. The server can either keep it around until the task dies (and a task death notification is received), or it can clean it up after some timeout if the client does not follow up and do some real authentication.

5. The client receives the reply message to its initial handshake request.
6. The client sends a request to create its initial capability. How this request looks depends on the type of the server and the initial capabilities it provides. Here are some examples:
   - A filesystem might provide an unauthenticated root directory object in return of the underlying node capability, which is provided by the parent filesystem and proves to the filesystem that the user was allowed to look up the root node of this filesystem (see section 6.5.1 on page 59).

     In this example, the parent filesystem will either provide the task info cap for the child filesystem to the user, or it will hold the task info cap while the user is creating their own (which the user has to verify by repeating the lookup, though). Again, see section 6.5.1 on page 59.
     The unauthenticated root directory object will then have the be authenticated using the normal reauthentication mechanism (see section 6.1 on pagerefauth). This can also be combined in a single RPC.
   - Every process acts as a server that implements the signal capability for this process. Tasks who want to send a signal to another task can perform the above handshake, and then provide some type of authentication capability that indicates that they are allowed to send a signal. Different authentication capabilities can be accepted by the signalled task for different types of signals.

     The Hurd used to store the signal capability in the proc server, where authorized tasks could look it up. This is no longer possible because a server can not accept capabilities implemented by untrusted tasks, see below.
7. The server replies with whatever capability the client requested, provided that the client could provide the necessary authentication capabilities, if any.

   It is not required that the server performs any authentication at all, but it is recommended, and all Hurd servers will do so.
   In particular, the server should normally only allow access from tasks running in the same system, if running multiple systems on the same host is possible.

Result The client has a task info capability for the server and an authenticated capability. The server has a task info capability for the client and seen some sort of authentication for the capability it gave to the client.

If you think that the above protocol is complex, you have seen nothing yet! Read on.

3.1.2 Returning a capability from a server to a client

Before we go on to the more complex case of copying a capability from one client to another, let us point out that once a client has a capability from a server, it is easy for the server to return more capabilities it implements to the client.

The server just needs to create the capability, acquire a capability ID in the client's cap ID space, and return the information in the reply RPC.

FIXME: Here should be the pseudo code for the protocol. For now, you have to take it out of the long version.

> The main point of this section is to point out that only one task info capability is required to protect all capabilities provided to a single task. The protocols described here always assume that no task info caps are held by anyone (except those mentioned in the preconditions). In reality, sometimes the required task info caps will already be held.

3.1.3 Copying a capability from one client to another task

The most complex operation in managing capabilities is to copy or move a capability from the client to another task, which subsequently becomes a client of the server providing the capability. The difficulty here lies in the fact that the protocol should be fast, but also robust and secure. If any of the participants dies unexpectedly, or any of the untrusted participants is malicious, the others should not be harmed.

Preconditions The client C has a capability from server S (this implies that C has a task info cap for S and S has a task info cap for C). It wants to copy the capability to the destination task D. For this, it will have to make RPCs to D, so C has also a capability from D (this implies that C has a task info cap for D and D has a task info cap for C). Of course, the client C trusts its servers S and D. D might trust S or not, and thus accept or reject the capability that C wants to give to D. S does not trust either C or D.

The `task` server is also involved, because it provides the task info capabilities. Everyone trusts the `task` server they use. This does not need to be the same one for every participant.

FIXME: Here should be the pseudo code for the protocol. For now, you have to take it out of the long version.

1. The client invokes the `cap_ref_cont_create` RPC on the capability, providing the task ID of the intended receiver D of the capability.

2. The server receives the `cap_ref_cont_create` RPC from the client. It requests a task info cap for D from its trusted task server, under the constraint that C is still living.

A task can provide a constraint when creating a task info cap in the `task` server. The constraint is a task ID. The task server will only create the task info cap and return it if the task with the constraint task ID is not destroyed. This allows for a task requesting a task info capability to make sure that another task, which also holds this task info cap, is not destroyed. This is important, because if a task is destroyed, all the task info caps it held are released.

In this case, the server relies on the client to hold a task info cap for D until it established its own. See below for what can go wrong if the server would not provide a constraint and both, the client and the destination task would die unexpectedly.

Now that the server established its own task info cap for D, it creates a reference container for D, that has the following properties:

- The reference container has a single new reference for the capability.
- The reference container has an ID that is unique among all reference container IDs for the client C.
- The reference container is associated with the client C. If C dies, and the server processes the task death notification for it, the server will destroy the reference container and release the capability reference it has (if any). All resources associated with the reference container will be released. If this reference container was the only reason for S to hold the task info cap for D, the server will also release the task info cap for D.
- The reference container is also associated with the destination task D. If D dies, and the server processes the task death notification for it, the server will release the capability reference that is in the reference container (if any). It will not destroy the part of the container that is associated with C.

The server returns the reference container ID R to the client.

3. The client receives the reference container ID R.

   If several capabilities have to be copied in one message, the above steps need to be repeated for each capability. With appropriate interfaces, capabilities could be collected so that only one call per server has to be made. We are assuming here that only one capability is copied.

4. The client sends the server thread ID T and the reference container ID R to the destination task D.

5. The destination task D receives the server thread ID T and the reference container ID R from C.

   It now inspects the server thread ID T, and in particular the task ID component of it. D has to make the decision if it trusts this task to be a server for it, or if it does not trust this task.

If D trusts C, it might decide to always trust T, too, irregardless of what task contains T.

If D does not trust C, it might be more picky about the task that contains T. This is because D will have to become a client of T, so it will trust it. For example, it will block on messages it sends to T.

> If D is a server, it will usually only accept capabilities from its client that are provided by specific other servers it trusts. This can be the authentication server, for example (see section 6.1 on page 51).
> Usually, the type of capability that D wants to accept from C is then further restricted, and only one possible trusted server implements that type of capabilities. Thus, D can simply compare the task ID of T with the task ID of its trusted server (authentication server, ...) to make the decision if it wants to accept the capability or not.

If D does not trust T, it replies to C (probably with an error value indicating why the capability was not accepted). In that case, jump to step 8.

Otherwise, it requests a task info cap for S from its trusted task server, under the constraint that C is still living.

Then D sends a `cap_ref_cont_accept` RPC to the server S, providing the task ID of the client C and the reference container ID R.

> `cap_ref_cont_accept` is one of the few interfaces that is not sent to a (real) capability, of course. Nevertheless, it is part of the capability object interface, hence the name. You can think of it as a static member in the capability class, that does not require an instance of the class.

6. The server receives the `cap_ref_cont_accept` RPC from the destination task D. It verifies that a reference container exists with the ID R, that is associated with D and C.

   > The server will store the reference container in data structures associated with C, under an ID that is unique but local to C. So D needs to provide both information, the task ID and the reference container ID of C.

   If that is the case, it takes the reference from the reference container, and creates a capability ID for D from it. The capability ID for D is returned in the reply message.

   From that moment on, the reference container is deassociated from D. It is still associated with C, but it does not contain any reference for the capability.

   > It is not deassociated from C and removed completely, so that its ID R (or at least the part of it that is used for C) is not reused. C must explicitely destroy the reference container anyway because D might die unexpectedly or return an error that gives no indication if it accepted the reference or not.

7. The destination task D receives the capability ID and enters it into its capability system. It sends a reply message to C.

> If the only purpose of the RPC was to copy the capability, the reply message can be empty. Usually, capabilities will be transfered as part of a larger operation, though, and more work will be done by D before returning to C.

8. The client C receives the reply from D. Irregardless if it indicated failure or success, it will now send the `cap_ref_cont_destroy` message to the server S, providing the reference container R.

   > This message can be a simple message. It does not require a reply from the server.

9. The server receives the `cap_ref_cont_destroy` message and removes the reference container R. The reference container is deassociated from C and D. If this was the only reason that S held a task info cap for D, this task info cap is also released.

   > Because the reference container can not be deassociated from C by any other means than this interface, the client does not need to provide D. R can not be reused without the client C having it destroyed first. This is different from the `cap_ref_cont_accept` call made by D, see above.

Result For the client C, nothing has changed. The destination task D either did not accept the capability, and nothing has changed for it, and also not for the server S. Or D accepted the capability, and it now has a task info cap for S and a reference to the capability provided by S. In this case, the server S has a task info cap for D and provides a capability ID for this task.

The above protocol is for copying a capability from C to D. If the goal was to move the capability, then C can now release its reference to it.

> Originally we considered to move capabilities by default, and require the client to acquire an additional reference if it wanted to copy it instead. However, it turned out that for the implementation, copying is easier to handle. One reason is that the client usually will use local reference counting for the capabilities it holds, and with local reference counting, one server-side reference is shared by many local references. In that case, you would need to acquire a new server-side reference even if you want to move the capability. The other reason is cancellation. If an RPC is cancelled, and you want to back out of it, you need to restore the original situation. And that is easier if you do not change the original situation in the first place until the natural "point of no return".

The above protocol quite obviously achieves the result as described in the above concluding paragraph. However, many other, and often simpler, protocols would also do that. The other protocols we looked at are not secure or robust though, or require more operations. To date we think that the above is the shortest (in particular in number of IPC operations) protocol that is also secure and robust (and if it is not we think it can be fixed to be secure and robust with minimal changes). We have no proof for its correctness. Our confidence comes from the scrutiny we applied to it. If you find a problem with the above protocol, or if you can prove various aspects of it, we would like to hear about it.

To understand why the protocol is laid out as it is, and why it is a secure and robust protocol, one has to understand what could possibly go wrong and why it does not cause any problems for any participant if it follows its part of the protocol (independent on what the other participants do). In the following paragraphs, various scenarios are suggested where things do not go as expected in the above protocol. This is probably not a complete list, but it should come close to it. If you find any other problematic scenario, again, let us know.

> Although some comments like this appear in the protocol description above, many comments have been spared for the following analysis of potential problems. Read the analysis carefully, as it provides important information about how, and more importantly, why it works.

The server S dies What happens if the server S dies unexpectedly sometime throughout the protocol?

> At any time a task dies, the task info caps it held are released. Also, task death notifications are sent to any task that holds task info caps to the now dead task. The task death notifications will be processed asynchrnouly, so they might be processed immediately, or at any later time, even much later after the task died! So one important thing to keep in mind is that the release of task info caps a task held, and other tasks noticing the task death, are always some time apart.

Because the client C holds a task info cap for S no imposter can get the task ID of S. C and D will get errors when trying to send messages to S.

> You might now wonder what happens if C also dies, or if C is malicious and does not hold the task info cap. You can use this as an exercise, and try to find the answer on your own. The answers are below.

Eventually, C (and D if it already got the task info cap for S) will process the task death notification and clean up their state.

The client C dies The server S and the destination task D hold a task info cap for C, so no imposter can get its task ID. S and D will get errors when trying to send messages to C. Depending on when C dies, the capability might be copied successfully or not at all.

Eventually, S and D will process the task death notification and release all resources associated with C. If the reference was not yet copied, this will include the reference container associated with C, if any. If the reference was already copied, this will only include the empty reference container, if any.

> Of course, the participants need to use internal locking to protect the integrity of their internal data structures. The above protocol does not show where locks are required. In the few cases where some actions must be performed atomically, a wording is used that suggests that.

The destination task D dies The client C holds a task info cap for D over the whole operation, so no imposter can get its task ID. Depending on when D dies, it has either not yet accepted the capability, then C will clean up by destroying the reference container, or it has, and then S will clean up its state when it processes the task death notification for D.

The client C and the destination task D die This scenario is the reason why the server acquires its own task info cap for D so early, and why it must do that under the constraint that C still lives. If C and D die before the server created the reference container, then either no request was made, or creating the task info cap for D fails because of the constraint. If C and D die afterwards, then no imposter can get the task ID of D and try to get at the reference in the container, because the server has its own task info cap for D.

> This problem was identified very late in the development of this protocol. We just did not think of both clients dieing at the same time! In an earlier version of the protocol, the server would acquire its task info cap when D accepts its reference. This is too late: If C and D die just before that, an imposter with D's task ID can try to get the reference in the container before the server processes the task death notification for C and destroys it.

Eventually, the server will receive and process the task death notifications. If it processes the task death notification for C first, it will destroy the whole container immediately, including the reference, if any. If it processes the task death notification for D first, it will destroy the reference, and leave behind the empty container associated with C, until the other task death notification is processed. Either way no imposter can get at the capability.

Of course, if the capability was already copied at the time C and D die, the server will just do the normal cleanup.

The client C and the server S die This scenario does not cause any problems, because on the one hand, the destination task D holds a task info cap for C, and it acquires its own task info cap for S. Although it does this quite late in the protocol, it does so under the constraint that C still lives, which has a task info cap for S for the whole time (until it dies). It also gets the task info cap for S before sending any message to it. An imposter with the task ID of S, which it was possible to get because C died early, would not receive any message from D because D uses C as its constraint in acquireing the task info cap for S.

The destination task D and the server S die As C holds task info caps for S and D, there is nothing that can go wrong here. Eventually, the task death notifications are processed, but the task info caps are not released until the protocol is completed or aborted because of errors.

The client C, the destination task D and the server S die Before the last one of these dies, you are in one of the scenarios which already have been covered. After the last one dies, there is nothing to take care of anymore.

> In this case your problem is probably not the capability copy protocol, but the stability of your software! Go fix some bugs.

So far the scenarios where one or more of the participating tasks die unexpectedly. They could also die purposefully. Other things that tasks can try to do purposefully to break the protocol are presented in the following paragraphs.

> A task that tries to harm other tasks by not following a protocol and behaving as other tasks might expect it is malicious. Beside security concerns, this is also an issue of robustness, because malicious behaviour can also be triggered by bugs rather than bad intentions.
>
> It is difficult to protect against malicious behaviour by trusted components, like the server S, which is trusted by both C and D. If a trusted component is compromised or buggy, ill consequences for software that trusts it must be expected. Thus, no analysis is provided for scenarious involving a malicious or buggy server S.

The client C is malicious If the client C wants to break the protocol, it has numerous possibilities to do so. The first thing it can do is to provide a wrong destination task ID when creating the container. But in this case, the server will return an error to D when it tries to accept it, and this will give D a chance to notice the problem and clean up. This also would allow for some other task to receive the container, but the client can give the capability to any other task it wants to anyway, so this is not a problem.

> If a malicious behaviour results in an outcome that can also be achieved following the normal protocol with different parameters, then this not a problem at all.

The client could also try to create a reference container for D and then not tell D about it. However, a reference container should not consume a lot of resources in the server, and all such resources should be attributed to C. When C dies eventually, the server will clean up any such pending containers when the task death notification is processed.

The same argument holds when C leaves out the call to `cap_ref_cont_destroy`.

The client C could also provide wrong information to D. It could supply a wrong server thread ID T. It could supply a wrong reference container ID R. If D does not trust C and expects a capability implemented by some specific trusted server, it will verify the thread ID numerically and reject it if it does not match. The reference container ID will be verified by the server, and it will only be accepted if the reference container was created by the client task C. Thus, the only wrong reference container IDs that the client C could use to not provoke an error message from the server (which then lead D to abort the operation) would be a reference container that it created itself in the first place. However, C already is frree to send D any reference container it created.

> Again C can not achieve anything it could not achieve by just following the protocol as well. If C tries to use the same reference container with several RPCs in D, one of them would succeed and the others would fail, hurting only C.
>
> If D does trust C, then it can not protect against malicious behaviour by C.

To summarize the result so far: C can provide wrong data in the operations it does, but it can not achieve anything this way that it could not achieve by just following the protocol. In most cases the operation would just fail. If it leaves out some operations, trying to provoke resource leaks in the server, it will only hurt itself (as the reference container is strictly associated with C until the reference is accepted by D).

> For optimum performance, the server should be able to keep the information about the capabilities and reference containers a client holds on memory that is allocated on the clients behalf.
>
> It might also use some type of quota system.

Another attack that C can attempt is to deny a service that S and D are expecting of it. Beside not doing one or more of the RPCs, this is in particular holding the task info caps for the time span as described in the protocol. Of course, this can only be potentially dangerous in combination with a task death. If C does not hold the server task info capability, then an imposter of S could trick D into using the imposter as the server. However, this is only possible if D already trusts C. Otherwise it would only allow servers that it already trusts, and it would always hold task info caps to such trusted servers when making the decision that it trusts them. However, if D trusts C, it can not protect against C being malicious.

> If D does not trust C, it should only ever compare the task ID of the server thread against trusted servers it has a task info cap for. It must not rely on C doing that for D.
>
> However, if D does trust C, it can rely on C holding the server task info cap until it got its own. Thus, the task ID of C can be used as the constraint when acquiring the task info cap in the protocol.

If C does not hold the task info cap of D, and D dies before the server acquires its task info cap for D, it might get a task info cap for an imposter of D. But if the client wants to achieve that, it could just follow the protocol with the imposter as the destination task.

The destination task D is malicious The destination task has not as many possibilities as C to attack the protocol. This is because it is trusted by C. So the only participant that D can try to attack is the server S. But the server S does not rely on any action by D. D does not hold any task info caps for S. The only operation it does is an RPC to S accepting the capability, and if it omits that it will just not get the capability (the reference will be cleaned up by C or by the server when C dies).

The only thing that D could try is to provide false information in the `cap_ref_cont_accept` RPC. The information in that RPC is the task ID of the client C and the reference container ID R. The server will verify that the client C has previously created a reference container with the ID R that is destined for D. So D will only be able to accept references that it is granted access to. So it can not achieve anything that it could not achieve by following the protocol (possibly the protocol with another client). If D accepts capabilities from other transactions outside of the protocol, it can only cause other transactions in its own task to fail.

> If you can do something wrong and harm yourself that way, then this is called "shooting yourself in your foot".
> The destination task D is welcome to shoot itself in its foot.

The client C and the destination task D are malicious The final question we want to raise is what can happen if the client C and the destination task D are malicious. Can C and D cooperate and attacking S in a way that C or D alone could not?

In the above analysis, there is no place where we assume any specific behaviour of D to help S in preventing an attack on S. There is only one place where we make an assumption for C in the analysis of a malicious D. If D does not accept a reference container, we said that C would clean it up by calling `cap_ref_cont_destroy`. So we have to look at what would happen if C were not to do that.

Luckily, we covered this case already. It is identical to the case where C does not even tell D about the reference container and just do nothing. In this case, as said before, the server will eventually release the reference container when C dies. Before that, it only occupies resources in the server that are associated with C.

This analysis is sketchy in parts, but it covers a broad range of possible attacks. For example, all possible and relevant combinations of task deaths and malicious tasks are covered. Although by no means complete, it can give us some confidence about the rightness of the protocol. It also provides a good set of test cases that you can test your own protocols, and improvements to the above protocol against.

3.1.4 The trust rule

The protocol to copy a capability from one client to another task has a dramatic consequence on the design of the Hurd interfaces.

Because the receiver of the capability must make blocking calls to the server providing the capability, the receiver of the capability *must* trust the server providing the capability.

This means also: If the receiver of a capability does not trust the server providing the capability, it *must not* accept it.

The consequence is that normally, servers can not accept capabilities from clients, unless they are provided by a specific trusted server. This can be the `task` or `auth` server for example.

This rule is even true if the receiver does not actually want to use the capability for anything. Just accepting the capability requires trusting the server providing it already.

In the Hurd on Mach, ports (which are analogous to capabilities in this context) can be passed around freely. There is no security risk in accepting a port from any source, because the kernel implements them as protected objects. Using a port by sending blocking messages to it requires trust, but simply storing the port on the server side does not.

This is different in the Hurd on L4: A server must not accept capabilities unless it trusts the server providing them. Because capabilities are used for many different purposes (remote objects, authentication, identification), one has to be very careful in designing the interfaces. The Hurd interfaces on Mach use ports in a way that is not possible on L4. Such interfaces need to be redesigned.

Often, redesigning such an interface also fixes some other security problems that exists with in the Hurd on L4, in particular DoS attacks. A good part of this paper is about redesigning the Hurd to avoid storing untrusted capabilities on the server side.

Examples are:

- The new authentication protocol, which eliminates the need for a rendezvous port and is not only faster, but also does not require the server to block on the client anymore (see section 6.1 on page 51).
- The signal handling, which does not require the `proc` server to hold the signal port for every task anymore (see section 6.2.1 on page 54).
- The new exec protocol, which eliminates the need to pass all capabilities that need to be transfered to the new executable from the old program to the filesystem server, and then to the `exec` server (see section 6.2.3 on page 55).
- The new way to implement Unix Domain Sockets, which don't require a trusted system server, so that descriptor passing (which is really capability passing) can work (see section 6.3 on page 58.
- The way parent and child filesystem are linked to each other, in other words: how mounting a filesystem works (see section 6.5.1 on page 59).
- The replacement for the `file_reparent()` RPC (see section 6.5.2 on page 60).

3.2 Synchronous IPC

The Hurd only needs synchronous IPC. Asynchronous IPC is usually not required. An exception are notifications (see below).

There are possibly some places in the Hurd source code where asynchronous IPC is assumed. These must be replaced with different strategies. One example is the implementation of select() in the GNU C library.

> A naive implementation would use one thread per capability to select on. A better one would combine all capabilities implemented by the same server in one array and use one thread per server.
>
> A more complex scheme might let the server process select() calls asynchronously and report the result back via notifications.

In other cases the Hurd receives the reply asynchronously from sending the message. This works fine in Mach, because send-once rights are used as reply ports and Mach guarantees to deliver the reply message, ignoring the kernel queue limit. In L4, no messages are queued and such places need to be rewritten in a different way (for example using extra threads).

> What happens if a client does not go into the receive phase after a send, but instead does another send, and another one, quickly many sends, as fast as possible? A carelessly written server might create worker threads for each request. Instead, the server should probably reject to accept a request from a client thread that already has a pending request, so the number of worker threads is limited to the number of client threads.
>
> This also makes interrupting an RPC operation easier (the client thread ID can be used to identify the request to interrupt).

3.3 Notifications

Notifications to untrusted tasks happen frequently. One case is object death notifications, in particular task death notifications. Other cases might be select() or notifications of changes to the filesystem.

The console uses notifications to broadcast change events to the console content, but it also uses shared memory to broadcast the actual data, so not all notifications need to be received for functional operation. Still, at least one notification is queued by Mach, and this is sufficient for the console to wakeup whenever changes happened, even if the changes can not be processed immediately.

From the servers point of view, notifications are simply messages with a send and xfer timeout of 0 and without a receive phase.

For the client, however, there is only one way to ensure that it will receive the notification: It must have the receiving thread in the receive phase of an IPC. While this thread is processing the notification (even if it is only delegating it), it might be preempted and another (or the same) server might try to send a second notification.

> It is an open challenge how the client can ensure that it either receives the notification or at least knows that it missed it, while the server remains save from potential DoS attacks. The usual strategy, to give receivers of notifications a higher scheduling priority than the sender, is not usable in a system with untrusted receivers (like the Hurd). The best strategy determined so far is to

have the servers retry to send the notification several times with small delays inbetween. This can increase the chance that a client is able to receive the notification. However, there is still the question what a server can do if the client is not ready.

An alternative might be a global trusted notification server that runs at a higher scheduling priority and records which servers have notifications for which clients, and that can be used by clients to be notified of pending notifications. Then the clients can poll the notifications from the servers.

Chapter 4

Threads and Tasks

The `task` server will provide the ability to create tasks and threads, and to destroy them.

> In L4, only threads in the privileged address space (the rootserver) are allowed to manipulate threads and address spaces (using the ThreadControl and SpaceControl system calls). The `task` server will use the system call wrappers provided by the rootserver, see section 2.6 on page 7.

The `task` server provides three different capability types.

Task control capabilities If a new task is created, it is always associated with a task control capability. The task control capability can be used to create and destroy threads in the task, and destroy the task itself. So the task control capability gives the owner of a task control over it. Task control capabilities have the side effect that the task ID of this task is not reused, as long as the task control capability is not released. Thus, having a task control capability affects the global namespace of task IDs. If a task is destroyed, task death notifications are sent to holders of task control capabilities for that task.

> A task is also implicitely destroyed when the last task control capability reference is released.

Task info capabilities Any task can create task info capabilities for other tasks. Such task info capabilities are used mainly in the IPC system (see section 3 on page 11). Task info capabilities have the side effect that the task ID of this task is not reused, as long as the task info capability is not released. Thus, having a task info capability affects the global namespace of task IDs. If a task is destroyed, task death notifications are sent to holders of task info capabilities for that task.

> Because of that, holding task info capabilities must be restricted somehow. Several strategies can be taken:

- Task death notifications can be monitored. If there is no acknowdgement within a certain time period, the `task` server could be allowed to reuse the task ID anyway. This is not a good strategy because it can considerably weaken the security of the system (capabilities might be leaked to tasks which reuse such a task ID reclaimed by force).
- The proc server can show dead task IDs which are not released yet, in analogy to the zombie processes in Unix. It can also make available the list of tasks which prevent reusing the task ID, to allow users or the system administrator to clean up manually.
- Quotas can be used to punish users which do not acknowledge task death timely. For example, if the number of tasks the user is allowed to create is restricted, the task info caps that the user holds for dead tasks could be counted toward that limit.
- Any task could be restricted to as many task ID references as there are live tasks in the system, plus some slack. That would prevent the task from creating new task info caps if it does not release old ones from death tasks. The slack would be provided to not unnecessarily slow down a task that processes task death notifications asynchronously to making connections with new tasks.

In particular the last two approaches should proof to be effective in providing an incentive for tasks to release task info caps they do not need anymore.

Task manager capability A task is a relatively simple object, compared to a full blown POSIX process, for example. As the `task` server is enforced system code, the Hurd does not impose POSIX process semantics in the task server. Instead, POSIX process semantics are implemented in a different server, the proc server (see also section 6.2 on page 54). To allow the `proc` server to do its work, it needs to be able to get the task control capability for any task, and gather other statistics about them. Furthermore, there must be the possibility to install quota mechanisms and other monitoring systems. The `task` server provides a task manager capability, that allows the holder of that capability to control the behaviour of the `task` server and get access to the information and objects it provides.

For example, the task manager capability could be used to install a policy capability that is used by the `task` server to make upcalls to a policy server whenever a new task or thread is created. The policy server could then indicate if the creation of the task or thread is allowed by that user. For this to work, the `task` server itself does not need to know about the concept of a user, or the policies that the policy server implements.

Now that I am writing this, I realize that without any further support by the `task` server, the policy server would be restricted to the task and thread ID of the caller (or rather the task control capability used) to make its decision. A more capability oriented approach would then not be possible. This requires more thought.

The whole task manager interface is not written yet.

When creating a new task, the `task` server allocates a new task ID for it. The task ID will be used as the version field of the thread ID of all threads created in the task. This allows the recipient of a message to verify the sender's task ID efficiently and easily.

> The version field is 14 bit on 32-bit architectures, and 32 bit on 64 bit architectures. Because the lower six bits must not be all zero (to make global thread IDs different from local thread IDs), the number of available task IDs is $2^{14} - 2^8$ resp. $2^{32} - 2^26$.
>
> If several systems are running in parallel on the same host, they might share thread IDs by encoding the system ID in the upper bits of the thread number.

Task IDs will be reused only if there are no task control or info capabilities for that task ID held by any task in the system. To support bootstrapping an IPC connection (see section 3.1.1 on page 14), the `task` server will delay reusing a task ID as long as possible.

> This is similar to how PIDs are generated in Unix. Although it is attempted to keep PIDs small for ease of use, PIDs are not reused immediately. Instead, the PID is incremented up to a certain maximum number, and only then smaller PID values are reused again.
>
> As task IDs are not a user interface, there is no need to keep them small. The whole available range can be used to delay reusing a task ID as long as possible.

When creating a new task, the `task` server also has to create the initial thread. This thread will be inactive. Once the creation and activation of the initial thread has been requested by the user, it will be activated. When the user requests to destroy the last thread in a task, the `task` server makes that thread inactive again.

> In L4, an address space can only be implicitely created (resp. destroyed) with the first (resp. last) thread in that address space.

Some operations, like starting and stopping threads in a task, can not be supported by the task server, but have to be implemented locally in each task because of the minimality of L4. If external control over the threads in a task at this level is required, the debugger interface might be used (see section 7 on page 63).

4.1 Accounting

We want to allow the users of the system to use the `task` server directly, and ignore other task management facilities like the `proc` server. However, the system administrator still needs to be able to identify the user who created such anonymous tasks.

For this, a simple accounting mechanism is provided by the task server. An identifier can be set for a task by the task manager capability, which is inherited at task creation time from the parent task. This accounting ID can not be changed without the task manager capability.

The `proc` server sets the accounting ID to the process ID (PID) of the task whenever a task registers itself with the `proc` server. This means that all tasks which do not register themself with the `proc` server will be grouped together

with the first parent task that did. This allows to easily kill all unregistered tasks together with its registered parent.

The `task` server does not interpret or use the accounting ID in any way.

4.2 Proxy Task Server

The `task` server can be safely proxied, and the users of such a proxy task server can use it like the real `task` server, even though capabilities work a bit differently for the `task` server than for other servers.

The problem exists because the proxy task server would hold the real task info capabilities for the task info capabilities that it provides to the proxied task. So if the proxy task server dies, all such task info capabilities would be released, and the tasks using the proxy task server would become insecure and open to attacks by imposters.

However, this is not really a problem, because the proxy task server will also provide proxy objects for all task control capabilities. So it will be the only task which holds task control capabilities for the tasks that use it. When the proxy task server dies, all tasks that were created with it will be destroyed when these tak control capabilities are released. The proxy task server is a vital system component for the tasks that use it, just as the real `task` server is a vital system component for the whole system.

4.3 Scheduling

The task server is the natural place to implement a simple, initial scheduler for the Hurd. A first version can at least collect some information about the cpu time of a task and its threads. Later a proper scheduler has to be written that also has SMP support.

The scheduler should run at a higher priority than normal threads.

> This might require that the whole task server must run at a higher priority, which makes sense anyway.
>
> Not much thought has been given to the scheduler so far. This is work that still needs to be done.

There is no way to get at the "system time" in L4, it is assumed that no time is spent in the kernel (which is mostly true). So system time will always be reported as 0.00, or 0.01.

Chapter 5

Virtual Memory Management

The mind and memory are more sharply exercised in comprehending another man's things than our own.

Timber or *Discoveries* by Ben Jonson

5.1 Introduction

The goal of an operating system is simply, perhaps reductively, stated: manage the available resources. In other words, it is the operating system's job to dictate the policy for obtaining resources and to provide mechanisms to use them. Most resources which the operating system manages are sparse resources, for instance the CPUs, the memory and the various peripherals including graphics cards and hard drives. Any given process, therefore, needs to compete with the other processes in the system for some subset of the available resources at any given time. As can be imagined, the policy to access and the mechanisms to use these resources determines many important characteristics of the system.

A simple single user system may use a trivial first come first serve policy for allocating resources, a device abstraction layer and no protection domains. Although this design may be very light-weight and the thin access layer conducive to high speed, this design will only work on a system where all programs can be trusted: a single malicious or buggy program can potentially halt all others from making progress simply by refusing to yield the CPU or allocating and not releasing resources in a timely fashion.

The Hurd, like Unix, aims to provide strong protection domains thereby preventing processes from accidentally or maliciously harming the rest of the system.

Unix has shown that this can be done efficiently. But more than Unix, the Hurd desires to identify pieces of the system which Unix placed in the kernel but which need not be there as they could be done in user space and provide additional user flexibility. Through our experience and analysis, we are convinced that one area is much of the virtual memory system: tasks are often allocating as much memory without regard—because Unix provides them with no mechanism to do so—for the rest of the system. But it is not a cooperative model which we wish to embrace but a model which holds the users of the resource responsible for it and when asked to release some of its memory will or violate the social contract and face exile. Not only will this empower users but it will force them to make smarter decisions.

5.1.1 Learning from Unix

Unix was designed as a multiuser timesharing system with protection domains thereby permitting process separation, i.e. allowing different users to concurrently run processes in the system and gain access to resources in a controlled fashion such that any one process cannot hurt or excessively starve any other. Unix achieved this through a monolithic kernel design wherein both policy and mechanism are provided by the kernel. Due to the limited hardware available at the time and the state of Multics[1], Unix imposed a strong policy on how resources could be used: a program could access files, however, lower level mechanism such as the file system, the virtual file system, network protocol stacks and devices drivers all existed in the kernel proper. This approach made sense for the extremely limited hardware that Unix was targeted for in the 1970s. As hardware performance increased, however, a separation between mechanism and policy never took place and today Unix-like operating systems are in a very similar state to those available two decades ago; certainly, the implementations have been vastly improved and tuned, however, the fundamental design remains the same.

One of the most important of the policy/mechanism couplings in the kernel is the virtual memory subsystem: every component in the system needs memory for a variety of reasons and with different priorities. The system must attempt to meet a given allocation criteria. However, as the kernel does not and cannot know how how a task will use its memory except based on the use of page fault statistics is bound to make sub-ideal eviction decisions. It is in part through years of fine tuning that Unix is able to perform as well as it does for the general applications which fit its assumed statistical model.

[1] Multics was seen as a system which would never realize due to its overly ambitious feature set.

5.1.2 Learning from Mach

The faults of Unix became clear through the use of Mach. The designers of Mach observed that there was too much mechanism in the kernel and attempted to export the file systems, network stack and much of the system API into user space servers. They left a very powerful VMM in the kernel with the device drivers and a novel IPC system. Our experience shows that the VMM although very flexible, is unable to make smart paging decisions: because Unix was tied to so many subsystems, it had a fair knowledge of how a lot of the memory in the system was being used. It could therefore make good guesses about what memory could be evicted and not be needed in the near future. Mach, however, did not have this advantage and relied strictly on page fault statistics and access pattern detection for its page eviction policy.

Based on this observation, it is imperitive that the page eviction scheme have good knowledge about how pages are being used as it only requires a few bad decisions to destroy performance. Thus, a new design can either choose to return to the monolithic design and add even more knowledge to the kernel to increase performance or the page eviction scheme can be remove from the kernel completely and placed in user space and make all tasks self paged.

5.1.3 Following the Hurd Philosophy

As the Hurd aims, like Unix, to be a multiuser system for mutually untrusted users, security is an absolute necessity. But it is not the object of the system to limit users excessively: as long as operations can be done securely, they should be permitted. It is based on this philosophy that we have adopted a self paging design for the new Hurd VMM: who knows better how a task will use its memory than the task itself? This is clear from the problems that have been encountered with LRU, the basic page evition algorithm, by database developers, language designers implementing garbage collectors and soft realtime application developers such as multimedia developers: they all wrestle with the underlying operating system's page eviction scheme. By putting the responsibility to page on tasks we think that tasks will be forced to make smart decisions as they can only hurt themselves.

5.2 Self Paging

If memory was infinite and the only problem was worrying about one program accessing the memory of another, memory allocation would be trivial. This is not, however, the case: memory is visibly finite and a well designed system will exploit it all. As memory is a system resource, a system wide memory allocation policy must be established which maximizes memory usage according to a given set of criteria.

In a typical Unix-like VMM, allocating memory (e.g. using `sbrk` or `mmap`) does not allocate physical memory but `virtual memory`. In order to increase the amount of memory available to users, the kernel uses a `backing store`, typically a hard disk, to temporarily free physical memory thereby allowing other processes to make progress. The sum of these two is referred to as virtual memory. The use of backing store ensures data integrity when physical memory must be freed and application transparency is required. A variety of criteria are used to determine which frames are `paged out`, however, most often some form of a priority based least recently used, LRU, algorithm is applied. Upon `memory pressure`, the system steals pages from low priority processes which have not been used recently or drain pages from an internal cache.

This design has a major problem: the kernel has to evict the pages but only the applications know which pages they really need in the near term. The kernel could ask the applications for this data, however, it is unable to trust the applications as they could, for instance, not respond, and the kernel would have to forcefully evict pages anyway. As such, the kernel relies on page fault statistics to make projections about how the memory will be used, thus the LRU eviction scheme. An additional result of this scheme is that as applications never know if mapped memory is in core, they are unable to make guarantees about deadlines.

These problems are grounded in the way the Unix VMM allocates memory: it does not allocate physical memory but virtual memory. This is illustated by the following scenario: when a process starts and begins to use memory, the allocator will happily give it all of memory in the system as long as no other process wants it. What happens, however, when a second memory hungry process starts is that the kernel has no way to take back memory it allocated to the first process. At this point, it has two options: it can either return failure to the second process or it can steal memory from the first process and send it to backing store.

One way to solve these problems is to have the VMM allocate phsyical memory and make applications completely self-paged. Thus, the burden of paging lies the application themselves. When application request memory, they no longer request virutal memory but physical memory. Once the application has exhausted its available frames, it is its responsibility to multiplex the available frames. Thus, virtual memory is done in the application itself. It is important to note that a standard manager or managers should be supplied by the operating system. This is important for implementing something like a POSIX personality. This should not, however, be hard coded: certain application may greatly benefit by being able to control their own eviction schemes. At its most basic level, hints could be provided to the manager by introducing extentions on basic function calls. For instance, `malloc` could take an extra parameter indicating the class of data being allocated. These class would provide hints about the expected usage pattern and life time of the data.

5.3 Bootstrap

When the Hurd starts up, all physical memory is eventually transfered to the physical memory server by the root server. At this point, the physical memory server will control all of the physical pages in the system.

5.4 Memory Allocation Policy

5.4.1 Guaranteed Frames and Extra Frames

The physical memory server maintains a concept of `guaranteed frames` and `extra frames`. The former are virtual frames that a given task is guaranteed to map in a very short amount of time. Given this predicate, the total number of guaranteed frames can never exceed the total number of physical frames in the system. Extra frames are frames which are given to clients who have reached their guaranteed frame allocation limit. The physical memory server may request that a client relinquish a number of extant extra frames at any time. The client must return the frames to the physical memory (i.e. free them) in a short amount of time. The task should not assume that it has enough time to send frames to backing store. As such, extra frames should only contain remanufacturable data (i.e. cached data). Should a task fail to return the frames in a reasonable amount of time, it risks having all of its memory dropped—not swapped out or saved in any way—and reclaimed by the physical memory server. Note that the physical memory server does not know if a given frame is considered guaranteed or extra: it knows that a given task has G guaranteed frames and $G + E$ allocated frames, and E extra frames. The distinction between guaranteed and extra frames must be made by the task itself. One strategy is to remember which frames can be remanufactured (e.g. reread from disk or recalculated) and internally promote them to guaranteed frames when the frame becomes dirty being careful to never have less than E clean frames in the task. Given these semantics, guanteed frames should not be thought of as wired (e.g. `mlock`ed in the POSIX sense)—although they can have this property—but as frames which the task itself must multiplex. Thus the idea of self-paged tasks.

Readers familiar with VMS will see striking similarities with the self-paging and guaranteed frame paradigms. This is not without reason. Yet, differences remain: VMS does not have extra frames and the number of guaranteed frames is fixed at task creation time. Frames returned to VMS (in order to allocate a new frame) are placed in a dirty list (thus the actual multiplexing of frames is done in VMS, not in user space) thereby simulating a two level backing store: a fast memory backing store where frames are waylaid and swap, where they are sent to when sufficient memory pressure forces them out. It is in this way that a given task may access more than its quota of memory when there is low memory contention (e.g. if there are two tasks each with 100 frames and there

are 1000 frames in the system for tasks, the remaining 800 are not dormant). Our divergence from VMS is motivated by the location of file systems and device drivers in the Hurd: unlike in VMS, the file systems and device drivers are in user space. Thus, the caching that was being done by VMS cannot be done intelligently by the physical memory server.

5.4.2 An External Memory Policy Server

The number of guaranteed frames that a given task has access to is not determined by the physical memory server but by the `memory policy server`. This division means the physical memory server need only concern itself with allocation mechanisms; all policy decisions are delegated to the policy server provided by the underlying operating system. (An important implication is that although tailored for Hurd specific needs, the physical memory server is essentially separate from the Hurd and can be used by other operating systems running on the L4 microkernel.) It is the memory policy server's responsibility to determine who gets how much memory. This may be calculated as a function of the user or looking in a file on disk for e.g. quotas. As can be seen this type of data acquisition could add significant complexity to the physical memory server and require blocking states (e.g. waiting for a read operation on file i/o) and could create circular dependencies. The default memory policy server's mechanisms and policies will be discussed later.

The physical memory server and the memory policy server will contain a shared buffer of tupples indexed by task id containing the number of allocated frames, the number of guaranteed frame, and a boolean indicating whether or not this task is eligible for guaranteed frames. The guaranteed frame field and the extra frame predicate may only be written to by the memory policy server. The number of allocated frames may only be written to by the physical memory server. This scheme means that no locking in required. (On some architectures where a read of a given field cannot be performed in a single operation, the read may have to be done twice.) The memory policy server must not over commit the number of frames, i.e. the total number of guaranteed frames must never exceed the number of frames avilable for allocation.

Until the memory policy server makes the intial contact with the physical memory server, memory will be allocated on a first come first serve basis. The memory policy server shall use the following remote procedure call to contact the physical memory server:

error_t pm_get_control (out hurd_cap_t control)

This function will succeed the first time it is called and return a control capability. It will fail all subsequent times. By using a capability, the acquiring task may move or copy the capability to another task. This permits replacing the policy server on a live system. At this point, the physical memory server

will begin allocating memory according to the described protocol. Note that the inital buffer will be initialized with the current total allocations while the guaranteed frames will be set to zero. The memory policy server must request the shared policy buffer as soon as possible and adjust these values.

The shared policy buffer may be obtained from the physical memory server by the policy by calling:

error_t pm_get_policy_buffer (out l4_map_t buffer)

The returned buffer is mapped with read and write access into the policy memory server's address space. It may need to be resized due to the number of tasks in the system. When this is the case, the physical memory server shall unmap the buffer from the memory policy server's address space and copy the buffer internally as required. The memory policy server will fault on the memory region on its next access and it may rerequest the buffer. This call will succeed when the sender is the memory policy server, it will fail otherwise.

5.5 Containers

In a monolithic kernel, other than through pipes, little data is exchanged between tasks: all services are provided by the kernel, a trusted entity which is able to directly access tasks' address space. In a multiserver system, most data acquisitions come from user space servers. As such, powerful primatives for moving memory around is an absolute necessity: physical copying must be kept to an absolute minimum and there must be a way to use and preserve copy on write pages.

Containers are the basic abstraction used for allocating, addressing and sharing memory. Conceptually, containers contain a set of integers identifying `virtual frame`s in the physical memory server. A virtual frame references a physical frame but is not bound to a particular physical frame (thereby allowing the physical memory server to move the contents between physical frames for page blocking, assembly of DMA arena and memory defragmentation). Virtual frames are thus the sharing mechanism for physical frames. Although virtual frames cannot be copied, their contents may be logically copied such that a new virtual frame is created with the same underlying physical frame. Sharing may be either real, e.g. System V shared memory, or logical, e.g. copy on write.

When a virtual frame is allocated into a container, there may be no physical frame associated with it. The physical memory server guarantees that when the contents of the virtual frame is accessed a physical frame will be provided in a short amount of time (cf. guaranteed virtual frames above).

Each virtual frame in a container counts against the container's owner's total allocated frames. Only the owner of a container may allocate frames into a container.

Containers only hold virtual frames. When the contents of a frame are copied to backing store, no association between the data on the backing store and the the frame identifier in the container is maintained by the physical memory server.

When a task starts, it will allocate an initial container and several frames into it. Typically, the total amount of memory used by an application will exceed the total number of guaranteed frames. When the task reaches its maximum permitted allocation, it must reuse an available frame. Typically, the task will choose a victim page, unmap any pages that point to the associated frame, swap the frame out, mark the frame as swapped out and save the swap identifier in the mapping database. At this point, the task may reuse the frame. This example illustrates that imagining a virtual frame as bound to a page in a task's address space for its entire lifetime is incorrect. It should also now be clear that when the data is eventually brought back into memory from backing store, it may reside in a different virtual frame (as well as a different physical frame).

Containers are used for passing data between tasks. Typically there will be two tasks, a client and a server. L4 provides a mechanism to map pages from one address space to another. This mechanism could be used to e.g. map a file into a client task's address space. An analysis reveals several problems with this approach. If the server dies before the client, the mappings in the client's address space will suddenly disappear. Similarly, if the server is malicious, it may revoke the mappings at some inconvenient (i.e. unrecoverable) time causing the client to crash or unable to inform the user of the change. Also, if a server allocates resources on behalf of the the client it becomes impossible to do system wide resource accounting as many servers are not trusted by the system. All of these problems are solved by containers. When a client needs to read data from a server, it creates a container, adds the number of frames that the server will require for the operation to it and finally shares the container with the server. After sending a request to the server, the server copies the data into the provided container. It is important to understand that the server does not "fill" the container: the number of frames remains constant; the state of the bits changes. When the server returns to the client, the client revokes the share and is now able to map the frames into its address space by contacting the physical memory server. Should the server die, the client remains uneffected as the data lives in the physical memory server. The physical memory server is also trusted thus if a task is malicious, it can only be malicious during the initial copy of the data into the container, i.e. before the client starts using the data and thereby giving the client the opportunity to report an inconsistencies to the caller. Finally, as the resources are allocated by the client via system servers, global resource accounting is possible.

5.5.1 The Container Interface

Creating Containers A container may be created using:

error_t pm_container_create (out container_t container)

A container_t is, for all intents and purposes, a hurd_cap_t. If a container is shared with another task, the second task may allocate frames which count against the container's owner's total allocated pages. This must be used with care.

Sharing Containers To allow another task to access the contents of a container, the container must be shared. Clearly, it is not desirable to grant full access to the container to the remote task: trust between a client and a server must exist, however, that trust is typically limited in both directions (neither the client trusts the server fully nor does the server fully trust the client). Since clients provide server with the resources for the operation, servers need a guarantee that the client will not touch the resources while it is in a critical section. Horrific results can emerge if this happens during a DMA operation. Likewise, clients need to have the ability to cancel an exant request and reclaim shared resources if the server does not answer in a timely manner thereby also preventing the server from being able to steal resources. In both of these cases, the physical memory server acts as the trusted third party. The physical memory server allows a server to lock a container for a limited amount of time during which the client may not access or destroy the resource. At any other time, the client can cancel the server's access to the shared resource.

To facility this, a second class capability is provided to access containers. Using this capability, clients may not allocate or deallocate frames.

> error_t pm_container_share (in container_t container, in task_t remote, out container_t weak_ref)

weak_ref can be passed to the sharee using the normal capability passing protocol.

Allocating and Deallocating Memory Virtual frames may be allocated into a container using:

> error_t pm_container_allocate (in container_t container, in frame_t start, in out int count, in int flags)

start is the first frame identifier to use for the new memory. If *count* is greater than one then frames will be allocated in the subsequent $count - 1$ frame identifiers. The number of frames actually allocated is returned in *count*. If an identifier already references a virtual frame, `EEXIST` is returned. *flags* is a bitwise or of: `CONT_ALLOC_PARTIAL`, `CONT_ALLOC_SQUASH` and `CONT_ALLOC_EXTRA`. If `CONT_ALLOC_PARTIAL` is set and the number of frames which can be allocated before a memory allocation error occurs is greater than one but less than *count* then the maximum number of frames is allocated, count is set to that number and the error is returned. If `CONT_ALLOC_PARTIAL` is not set then partial

allocations will fail, count will be set to 0 and an error will be returned. If CONT_ALLOC_SQUASH is set and a frame identifier already references a frame, the virtual frame will be dropped and its contents lost. Using this flag is dangerous and be a sign of internal inconsistencies in the task! All virtual frames should be accounted for by the task and deallocated explicitly. If CONT_ALLOC_EXTRA is set then extra frames may be allocated otherwise the physical memory server will only allocate up to the guaranteed virtual frame limit. This flag should only be used by tasks able to handle the added complexity of the extra frame protocol. The contents of allocated frames is undefined.

Deallocating memory is done using:

> error_t pm_container_deallocate (in container_t container, in frame_t start, in out int count, in int flags)

The arguments have similar meaning as those in pm_container_allocate. CONT_DEALLOC_PARTIAL and CONT_DEALLOC_SQUASH are similar to CONT_ALLOC_PARTIAL and CONT_ALLOC_SQUASH respectively.

Mapping Memory The physical memory server guarantees that a mapping operation takes a short amount of time: no guarantee is made that this will happen immediately as the underlying physical frames may have to be allocated in which case the physical memory server may have to be reap physical pages from other tasks' extra frame allocations.

The physical memory server may unmap pages at any time. This allows the physical memory server to fucntionally lock the contents of the frame and move it to a new physical frame. As such, tasks must be prepared to reestablish a mapping with the physical memory server at anytime. The physical memory server is not a registry of mappings: it is a cache.

Read-only mappings may be returned when read/write mapping are requested: the physical memory server will never grant a read/write mapping if the frame is marked copy on write. In order to obtain a read/write mapping (and thus force the copy on write), the task must add the enforced write flag to the mapping request.

> error_t pm_container_map (in container_t container, in frame_t start, in int nr_frames, in int flags)

Flags may is a bitwise or of: CONT_MAP_READ, CONT_MAP_WRITE and CONT_MAP_FORCE_WRITE. CONT_MAP_FORCE_WRITE will only be respected if CONT_MAP_WRITE is also set.

Doing It All At Once When reading to or writing data from a server, the task will normally: allocate a new container, fill it with memory and share the container with the server. Since this is such a common operation, short cuts are provided to reduce the required number of rpcs:

error_t pm_container_create_with (out container_t container, in in int frame_count, out container_t weak_ref)

error_t pm_container_create_from (out container_t container, in container_t source, in frame_t start, in int count, out container_t weak_ref)

error_t pm_container_create_grather (out container_t container, in container_t source, in frame_t [] frames, out container_t weak_ref)

Copying Data Into or Out of Containers It is possible to copy data into containers by mapping the frames in question and using `memcpy`. If this technique is used there is no easy way to create logical copies (copy on write): an especially important technique for sharing executable and shared library text. A family of functions are available which logically copies the contents of one container to another:

error_t pm_container_copy (in container_t src, in frame_t src_start, in countainer_t dest, in frame_t dest_start, in int frame_count, out frame_t frame_error)

error_t pm_container_copy_scatter (in container_t src, in frame_t src_start, in countainer_t dest, in frame_t [] dest_frames, out frame_t frame_error)

error_t pm_container_copy_gather (in container_t src, in frame_t [] src_frames, in countainer_t dest, in frame_t dest_start, out frame_t frame_error)

error_t pm_container_copy_scatter_gather (in container_t src, in frame_t [] src_frames, in countainer_t dest, in frame_t [] dest_frames, out frame_t frame_error)

If a frame does not exist in the source, ENOENT. If a frame does not exist in the destination, ENOMEM is returned. In both cases, the frame identifier causing the error is returned in *frame_error*.

Locking Containers and Pinning Memory

Finding Deallocate Memory

Reusing frames release_data

5.5.2 Moving Data

Data will be moved around using containers. Describe how to read and write. Task -¿ FS -¿ Device drivers. Locking memory. Caching.

It is important that clients do the allocation for the memory which they use: not the servers doing allocations on behalf on clients: in the latter, there is no way to do resource tracking.

Discuss mmap: local function call. RPC is done when a page is faulted: do a read from the fs (into a container), then map the data from the container into the AS as required.

MAP_COPY sucks: fs must save all modified data. What happens when a 100MB file is completely rewritten (or 1GB, etc)? can we use upcalls? If we do, the fs still needs to hold the data in the intern. Can we copy the file on disk and use that as backing store (think how deleting an open file works).

Can a readonly private mapping once faulted be dropped or must we promote it to anonymous memory and send it to swap fearing that the underlying block might change between dropping it and rereading it (e.g. by another task modifying the file)?

5.6 Caching Store Accesses

It need not be explained how caching accesses to stores can radically improve the speed of the system. In a monolithic kernel this cache is added to by the readers, i.e. the device drivers, supplemented with metadata from the file systems in the form of expected access patterns based on the type of data and how the file was opened and managed by the virtual memory manager. In our design, this is impossible: each component—each device driver, each file system and the physical memory manager—all live in their own address spaces; additionally there will rarely be mutual trust: the physical memory server may not trust the file systems nor the "device drivers" (consider a network block device). A caching mechanism must be designed.

The purpose of caching is useful for multiple readers of a given block. Sometimes this is the same task, however, more often it is multiple tasks. Thus, having the caching scheme in each task is quite difficult as tasks do not trust one another and furthermore, tasks can die at any time thereby dropping their cache. The logical place to put the cache then is the common point of access, the file system.

An argument could be made that in reality, the common point of access is the device driver: there can be multiple accessors of the same store. The question must be asked: what happens when the device driver is made the cache point instead of the file system? Logically, a large tradeoff is made in terms of the ability to intelligently decide what frame to keep in the cache. The file system, for instance, has meta-data about how a given frame may be used based on how

a file is opened and may realize that some frames need not be placed in the cache because they will be used once and immediately discarded. This is true of the access patterns of multimedia applications. These types of hints may be gathered at file open time. The class of data is another way the file system is able to predict usage, for example, it understands the difference between meta-data—inodes and directories—and file data. A file system is also able to anticipate file-level access patterns whereas a device driver can only anticipate block-level access patterns, i.e. although file data is sometimes sequential, it is often scattered across a section of the disk due to fragmentation. The primary way a the device driver can really manage its cache is through historical data in the form of previous accesses (which is itself even more limited as the device driver is uninformed of cache hits in the file system cache). This type of data implies some form of LRU, least recently used, eviction scheme. It should now be clear that the file system can make smarter decisions about what which blocks to evict due to its ability to make predictions based on client hints and its greater understanding of the data in the store.

If we resign ourselves to keeping the cache only in the file system, then multiple users of a store will be penalized greatly: a block read by one client will always be reread if another client requests the same block: not only is the store accessed a second time, but twice as much memory will be used as there is no way to share the frame and use copy on write. Is this penalty worth the added intelligence in the file system? An argument can be made that using just one caching strategy is suboptimital when we could just have two: nothing stops both the file system and the device driver from caching thereby permitting the former to continue to maintain an intelligent cache and the device driver to have its simple LRU cache. This argument overlooks several important implications of having the two caches. First, complexity is being added to the device driver in the form of a list of frames it has read and given out. This increase in memory usage has a secondary effect: if the data structures become large (as it certainly will for large active stores), it will be impossible to keep the device driver in question in a small address space (an important optimization on architectures without tagged TLBs, table look aside buffers). Second, if both the file system and the device driver keep a cache, when the file system has a cache miss, the device driver then checks its cache before going to disk. The device driver will only ever have a cache hit if there are multiple readers: when there is a single user of a store, the file system's cache and the device driver's cache will be identical. This begs the question: how often will there be multiple users of a single store? The answer seems to be very rarely: assuming the common case that the store has some type of file system on it, there can only be multiple users if all users are readers (note that not even one can be a writer as this implies cache consistency issues across different users of the store). Since this is a very rare case, we argue based on the philosophy "do not optimize for rare cases" that the overhead is greater than the potential pay back from the optimization. Having multiple caches leads to a further problem: a frame is really not evicted from the system until it is purged from all caches. Thus if the file system cache is smart and chooses

the better frames to evict, the cooresponding physical frames will not really be freed until the device driver also drops its references to the frames. Thus, the effectiveness of the smarter caching algorithm is impeded by the device driver's caching scheme. Double caching must be avoided.

5.6.1 Caching in the File System

We have argued above that all block caching will be done at the file system layer. In this section, we detail how the caching will work.

The file system allocates extra frames as long as it can and adds all eligible frames to the cache by logically copying them into a local container (data which it reasons will be read once and then dropped may not be considered eligible). When the physical memory server wants frames back, it chooses a victim with extra frames and asks for a subset of them back. If a task has G guaranteed frames and $G + E$ frames allocated, the physical memory server can request up to E frames back from the task. We recall from the definition of the extra frames that extra frames must be given back quickly (i.e. there is no time to send them to swap).

Although a task chooses a frame to evict from its cache, it does not mean that the frame will be reused immediately, in fact, it is sometimes that case that the frame cannot be reused at all as another task has a reference to the frame (in the form of a logical copy). As such, it would be nice to be able to get frames back that might still be in the physical memory server. The following mechanism is thus provided: when a frame is returned to the physical memory server, the reference to the frame is turned into a soft reference. Only when the frame is actually reused by the physical memory server are soft references discarded. A task is able to convert a soft reference back to a hard reference by contacting the physical memory server and asking for the frame back. If this operation returns `ENOEXIST`, the frame has been reused and the frame must be remanufactured (e.g. by retrieving it from backing store). This operation may also fail and return `ENOMEM` if the task does not have enough guaranteed frames and there are no extra frames available.

> There is a problem here in the form of name space pollution: the task doing the caching has to remember the mapping of blocks to container identifiers in order to recover the soft reference but the task has no way to know when the physical memory server expires a given soft reference. Thus, while the physical memory server may drop a frame, the task will only ever know this when it tries to convert the soft reference to a hard reference and fails (i.e. gets a cache miss). For frames which this is never done, the memorized mapping will never be invalidated. This may not be a problem if a block offset to container id is used, however, if hashing is done or some other mapping of block offsets to container identifiers is used, this will pollute the cache container's name space.

5.6.2 Caching Interfaces

The physical memory server will do an up call to a victim task requesting a number of frames back. The physical memory server may do this at any time for any reason and it expects to receive the frames back from the task within a short amount of time (the victim task should not expect to be able to send the frames to backing store in that amount of time). The physical memory server will never request guaranteed frames. As such, this number will always be less than or equal to the number of allocated frames minus the number of guaranteed frames.

> void pm_return_frames (in int count);

The physical memory send this message to the task's memory control thread. The thread must always be ready to receive: the physical memory server will never wait (thus, the thread must be in the receiving state). If the thread is not ready, the physical memory server assumes that the task is misbehaving. The physical memory server does not wait for a reply, instead, the client must free the frames using `pm_release_frames` as described above.

5.7 The Memory Policy Server

At task creation time, the task must negotiate a medium-term contract for guaranteed frames and determine if it shall have access to extra frames. This may be renegotiated later. It must be renegotiated when the contract expires. The policy server will give the task enough time to send frames to swap before committing if the number of guaranteed frames is reduced.

5.8 Sending Data to Swap

When a task reaches its guaranteed frame allocation, it must begin to reuse its available virtual frames. If the data is frames is precious (i.e. not easliy constructed by e.g. a calculation or by rereading a file) then the task will want to save the contents for when it is needed in the future. This can be done by sending a frame to backing store.

> error_t pm_swap (in container_t c, in container_frame_t frame, in int count, out [] swap_ids)

The swap server resides in (or is proxied by) the phsyical memory server. This allows the logical copies of frames to be preserved across the swapped out period (i.e. logical copies are not lost when a frame is sent to swap). If this was not the case, then when a number of tasks all with a reference to a given physical

send the frame to swap, the swap server would allocate and write N times as opposed to once when all of the tasks eventually release any references to the frame.

Frame may not be sent to swap immediately. Instead, they are kept on an inactive list allowing thereby allowing a task to recover the contents of a frame before it is flushed to swap (that is to say, swap operations are not synchronous).

Since there may be multiple references to a virtual frame, it is recommended that `pm_container_orphan_data` be called before the frame is reused to prevent gratuitous copy on writes from begin performed. It also important to call this function if the frame was being used for shared memory.

Swap quotas (put the policy in the memory policy server).

5.9 Self Paging

As already explained, tasks are self-paged. The default implementation provided with the hurd has each thread in a task set its pager (i.e. its fault handler) to a common pager thread in the same address space. This thread maintains a mapping database which associates virtual addresses with either a frame of memory in a container or information on how to retrieve the data, e.g. from swap or a file server.

Normally, there is a single primary container for virtual frames that is created at start up. A task may choose to use more containers and generally will for short periods of time (for instance, for reading to and writing from servers). The pager must always be able to handle multiple containers. When using additional containers, frames need to be added to them to be shared with the server. The pager must provide a mechanism to allow the caller to steal guaranteed frames for this purpose and return them upon deallocation of the container.

5.9.1 The Pager

The pager itself may require a fair amount of memory for its database and all of the code and supporting libraries. This presents a problem: if the pager handles page faults, who will handle its faults? One of two solutions are possible: either all of the text and data must be wired into memory (thereby reducing the number of frames available for multiplexing application memory) or the pager is itself paged. The default self-pager implementation uses the latter option: the pager, now referred to as the primary pager, is backed by a final pager. The final pager only maps the pagers text and data thus it has a significantly smaller memory footprint. Care must be taken to be sure that the primary pager does not accidently allocate memory from common memory pools: in the very least it needs its own private `malloc` arena. As the primary pager will call, for instance, routines to manipulate capabilities, this text must be backed by the final pager.

Other code can also, however, makes calls to the capability library. This means that the primary pager must also have a copy of these mappings in its database.

The purpose of the final pager is to allow the data and some of the text of the primary pager to be swapped. As such, the final pager must be able to at least read data from file servers and retrieve data from backing store. This may imply significant overlap of the text for the primary and final pagers. In some case, however, it may be useful to have a second implementation of a function only for the final pager which is optimized for size or avoids making calls to certain libraries.

Managing Mappings

Mappings are normally made via calls to `mmap`. Unlike in Unix, this is not a system trap: instead it is almost always implemented locally. `mmap` must associate a region with either anonymous memory or with a file on disk. This is only a matter of creating a few entries in the mapping database so that faults will brings the data in lazily.

Rather than have the caller manipulate the mapping database directly, instead, a local ipc sent to the primary pager. If there is only ever a single thread which manipulates the mapping database, there will be locking requirements. If the pager thread is busy, then the local ipc call blocks in the kernel.

It is not always useful to fault memory in lazily: when a task has received data from a server, it will normally be in a container from where it must be consumed. The task will generally map the container into memory and then proceed to use or at least copy the data to some other location. Clearly, the faulting the pages in is a waste. As such, the pager should provide a mechanism which allows the caller to not only establish a mapping from a container but also to map the pages immediately in the address space.

5.9.2 Reusing Virtual Frames

Multiplexing frames: say the contents of a frame are sent to swap in order to reuse the frame for something else. The frame itself must be cleared, i.e. disassocitated with any logical copies. This is done using:

> error_t pm_release_data (in pm_container_t container, in pm_frame_t[] frames)

5.9.3 Taking Advantage of Self-Paging

extend malloc via e.g. the slab mechanism, extend fopen (how a file is used).

Chapter 6

The POSIX personality

The Hurd offers a POSIX API to the user by default. This is implemented in the GNU C library which uses the services provided by the Hurd servers. Several system servers support the C library.

6.1 Authentication

Capabilities are a good way to give access to protected objects and services. They are flexible, lightweight and generic. However, Unix traditionally uses access control lists (ACL) to restrict access to objects like files. Any task running with a certain user ID can access all files that are readable for the user with that user ID. Although all objects are implemented as capabilities in the Hurd, the Hurd also supports the use of user IDs for access control.

The system authentication server `auth` implements the Unix authentication scheme using capabilities. It provides auth capabilities, which are associated with a list of effective and available user and group IDs. The holder of such a capability can use it to authenticate itself to other servers, using the protocol below.

Of course, these other servers must use (and trust) the same `auth` server as the user. Otherwise, the authentication will fail. Once a capability is authenticated in the server, the server will know the user IDs of the client, and can use them to validate further operations.

The `auth` server provides two types of capabilities:

Auth capabilities An auth capability is associated with four vectors of IDs: The effective user and group IDs, which should be used by other servers to authenticate operations that require certain user or group IDs, and the available

user and group IDs. Available IDs should not be used for authentication purposes, but can be turned into effective IDs by the holder of an auth capability at any time.

New auth capabilities can be created from existing auth capabilities, but only if the requested IDs are a subsets from the union of the (effective and available) IDs in the provided auth capabilities. If an auth capability has an effective or available user ID 0, then arbitrary new auth objects can be created from that.

Passport capabilities A passport capability can be created from an auth capability and is only valid for the task that created it. It can be provided to a server in an authentication process (see below). For the client, the passport capability does not directly implement any useful operation. For the server, it can be used to verify the identity of a user and read out the effective user and group IDs.

The auth server should always create new passport objects for different tasks, even if the underlying auth object is the same, so that a task having the passport capability can not spy on other tasks unless they were given the passport capability by that task.

6.1.1 Authenticating a client to a server

A client can authenticate itself to a server with the following protocol:

Preconditions The client C has an auth capability implemented by the `auth` server A. It also has a capability implemented by the server S. It wants to reauthenticate this capability with the auth capability, so the server associates the new user and group IDs with it.

The server also has an auth capability implemented by its trusted `auth` server. For the reauthentication to succeed, the `auth` server of the client and the server must be identical. If this is the case, the participating tasks hold task info caps for all other participating tasks (because of the capabilities they hold).

1. The client C requests the passport capability for itself from the auth capability from A.

   > Normally, the client will request the passport capability only once and store it together with the auth capability.

2. The `auth` server receives the request and creates a new passport capability for this auth capability and this client. The passport capability is returned to the user.

3. The user receives the reply from the `auth` server.

   It then sends the reauthentication request to the server S, which is invoked on the capability the client wants to reauthenticate. It provides the passport capability as an argument.

4. The server S can accept the passport capability, if it verifies that it is really implemented by the `auth` server it trusts. If the client does not provide a passport capability to the trusted `auth` server, the authentication process is aborted with an error.

   Now the server can send a request to the `auth` server to validate the passport capability. The RPC is invoked on the passport capability.

5. The `auth` server receives the validation request on the passport capability and returns the task ID of the client C that this passport belongs to, and the effective user and group IDs for the auth cap to which this passport cap belongs.

   > The Hurd on Mach returned the available IDs as well. This feature is not used anywhere in the Hurd, and as the available IDs should not be used for authentication anyway, this does not seem to be useful. If it is needed, it can be added in an extended version of the validation RPC.

6. The server receives the task ID and the effective user and group IDs. The server now verifies that the task ID is the same as the task ID of the sender of the reauthentication request. Only then was the reauthentication request made by the owner of the auth cap. It can then return a new capability authenticated with the new user and group IDs.

   > The verification of the client's task ID is necessary. As the passport cap is copied to other tasks, it can not serve as a proof of identity alone. It is of course absolutely crucial that the server holds the task info cap for the client task C for the whole time of the protocol. But the same is actually true for any RPC, as the server needs to be sure that the reply message is sent to the sender thread (and not any imposter).

7. The client receives the reply with the new, reauthenticated capability. Usually this capability is associated in the server with the same abstract object, but different user credentials.

   > Of course a new capability must be created. Otherwise, all other users holding the same capability would be affected as well.

   The client can now deallocate the passport cap.

   > As said before, normally the passport cap is cached by the client for other reauthentications.

Result The client C has a new capability that is authenticated with the new effective user and group IDs. The server has obtained the effective user and group IDs from the `auth` server it trusts.

> The Hurd on Mach uses a different protocol, which is more complex and is vulnerable to DoS attacks. The above protocol can not readily be used on Mach, because the sender task of a message can not be easily identified.

6.2 Process Management

The `proc` server implements Unix process semantics in the Hurd system. It will also assign a PID to each task that was created with the `task` server, so that the owner of these tasks, and the system administrator, can at least send the `SIGKILL` signal to them.

The `proc` server uses the task manager capability from the `task` server to get hold of the information about all tasks and the task control caps.

> The `proc` server might also be the natural place to implement a first policy server for the `task` server.

6.2.1 Signals

Each process can register the thread ID of a signal thread with the `proc` server. The proc server will give the signal thread ID to any other task which asks for it.

> The thread ID can be guessed, so there is no point in protecting it.

The signal thread ID can then be used by a task to contact the task to which it wants to send a signal. The task must bootstrap its connection with the intended receiver of the signal, according to the protocol described in section 3.1.1 on page 14. As a result, it will receive the signal capability of the receiving task.

The sender of a signal must then provide some capability that proves that the sender is allowed to send the signal when a signal is posted to the signal capability. For example, the owner of the task control cap is usually allowed to send any signal to it. Other capabilities might only give permission to send some types of signals.

> The receiver of the signal decides itself which signals to accept from which other tasks. The default implementation in the C library provides POSIX semantics, plus some extensions.

Signal handling is thus completely implemented locally in each task. The `proc` server only serves as a name-server for the thread IDs of the signal threads.

> The `proc` server can not hold the signal capability itself, as it used to do in the implementation on Mach, as it does not trust the tasks implementing the capability. But this is not a problem, as the sender and receiver of a signal can negotiate and bootstrap the connection without any further support by the `proc` server.
>
> Also, the `proc` server can not even hold task info caps to support the sender of a signal in bootstrapping the connection. This means that there is a race between looking up the signal thread ID from the PID in the `proc` server and acquiring a task info cap for the task ID of the signal receiver in the sender. However, in Unix, there is always a race when sending a signal using `kill`. The task server helps the users a bit here by not reusing task IDs as long as possible.

Some signals are not implemented by sending a message to the task. `SIGKILL` for example destroys the tasks without contacting it at all. This feature is implemented in the `proc` server.

The signal capability is also used for other things, like the message interface (which allows you to manipulate the environment variables and `auth` capability of a running task, etc).

6.2.2 The `fork()` function

To be written.

6.2.3 The `exec` functions

The `exec` operation will be done locally in a task. Traditionally, `exec` overlays the same task with a new process image, because creating a new task and transferring the associated state is expensive. In L4, only the threads and virtual memory mappings are actually kernel state associated with a task, and exactly those have to be destroyed by `exec` anyway. There is a lot of Hurd specific state associated with a task (capabilities, for example), but it is difficult to preserve that. There are security concerns, because POSIX programs do not know about Hurd features like capabilities, so inheriting all capabilities across `exec` unconditionally seems dangerous.

> One could think that if a program is not Hurd-aware, then it will not make any use of capabilities except through the normal POSIX API, and thus there are no capabilities except those that the GNU C library uses itself, which `exec` can take care of. However, this is only true if code that is not Hurd-aware is never mixed with Hurd specific code, even libraries (unless the library intimately cooperates with the GNU C library). This would be a high barrier to enable Hurd features in otherwise portable programs and libraries.
>
> It is better to make all POSIX functions safe by default and allow for extensions to let the user specify which capabilities besides those used for file descriptors etc to be inherited by the new executable.
>
> For `posix_spawn()`, this is straight-forward. For `exec`, it is not. either specific capabilities could be markes as “do not close on `exec`”, or variants of the `exec` function could be provided which take further arguments.

There are also implementation obstacles hindering the reuse of the existing task. Only local threads can manipulate the virtual memory mappings, and there is a lot of local state that has to be kept somewhere between the time the old program becomes defunct and the new binary image is installed and used (not to speak of the actual program snippet that runs during the transition).

So the decision was made to always create a new task with `exec`, and copy the desired state from the current task to the new task. This is a clean solution, because a new task will always start out without any capabilities in servers, etc, and thus there is no need for the old task to try to destroy all unneeded capabilities and other local state before `exec`. Also, in case the `exec` fails, the old program can continue to run, even if the `exec` fails at a very late point (there is no "point of no return" until the new task is actually up and running).

For suid and sgid applications, the actual `exec` has to be done by the filesystem. However, the filesystem can not be bothered to also transfer all the user state into the new task. It can not even do that, because it can not accept capabilities implemented by untrusted servers from the user. Also, the filesystem does not want to rely on the new task to be cooperative, because it does not necessarily trust the code, if is is owned by an untrusted user.

1. The user creates a new task and a container with a single physical page, and makes the `exec` call to the file capability, providing the task control capability. Before that, it creates a task info capability from it for its own use.
2. The filesystem checks permission and then revokes all other users on the task control capability. This will revoke the users access to the task, and will fail if the user did not provide a pristine task object. (It is assumed that the filesystem should not create the task itself so the user can not use suid/sgid applications to escape from their quota restriction).
3. Then it revokes access to the provided physical page and writes a trusted startup code to it.
4. The filesystem will also prepare all capability transactions and write the required information (together with other useful information) in a stack on the physical page.
5. Then it creates a thread in the task, and starts it. At pagefault, it will provide the physical page.
6. The startup code on the physical page completes the capability transfer. It will also install a small pager that can install file mappings for this binary image. Then it jumps to the entry point.
7. The filesystem in the meanwhile has done all it can do to help the task startup. It will provide the content of the binary or script via paging or file reads, but that happens asynchronously, and as for any other task. So the filesystem returns to the client.

8. The client can then send its untrusted information to the new task. The new task got the client's thread ID from the filesystem (possibly provided by the client), and thus knows to which thread it should listen. The new task will not trust this information ultimatively (ie, the new task will use the authentication, root directory and other capabilities it got from the filesystem), but it will accept all capabilities and make proper use of them.

9. Then the new task will send a message to proc to take over the old PID and other process state. How this can be done best is still to be determined (likely the old task will provide a process control capability to the new task). At that moment, the old task is desrtoyed by the proc server.

This is a coarse and incomplete description, but it shows the general idea. The details will depend a lot on the actual implementation.

The startup information

The following information is passed to the new task by the parent (the filesystem in the suid case). Every item is a machine word.

1. `magic`

   The first four bytes are `E`, `X`, `E`, `C`.

2. `program header location`

3. `program header size`

   The location and size of the program header. The meaning of this field depends on the binary format.

4. `feature flags`

   This bit-field indicates which of the following information is present. If the information is not present, the corresponding machine words are undefined. This provides simple version control.

   > They could also be undefined.

5. `wortel thread ID`

6. `wortel control cap ID`

   The thread ID of the `wortel` rootserver, and the local ID of the `wortel` control cap. The `wortel` control cap allows the user to make privileged system calls. This field is only present if the user has this capability. Usually, this is only the case for some initial servers at bootstrap.

7. `physmem thread ID`

8. `physmem control cap ID`

   The thread ID physical memory server, and the local ID of the `physmem` control cap. This cap can be used to manage the physical memory of this task.

9. `physmem startup page container cap ID`

   The container cap ID for the startup code, containing this information, the initial pager, and other startup code. This container is mapped into the address space of the task outside of the actual program, and can be unmapped by the program after it has used this information and installed its own pager, by destroying this container, to reclaim the virtual address space and physical memory it occupies.

10. (More to come.)

6.3 Unix Domain Sockets

In the Hurd on Mach, there was a global pflocal server that provided unix domain sockets and pipes to all users. This will not work very well in the Hurd on L4, because for descriptor passing, read: capability passing, the unix domain socket server needs to accept capabilities in transit. User capabilities are often implemented by untrusted servers, though, and thus a global pflocal server running as root can not accept them.

However, unix domain sockets and pipes can not be implemented locally in the task. An external task is needed to hold buffered data capabilities in transit. in theory, a new task could be used for every pipe or unix domain socketpair. However, in practice, one server for each user would suffice and perform better.

This works, because access to Unix Domain Sockets is controlled via the filesystem, and access to pipes is controlled via file descriptors, usually by inheritance. For example, if a fifo is installed as a passive translator in the filesystem, the first user accessing it will create a pipe in his pflocal server. From then on, an active translator must be installed in the node that redirects any other users to the right pflocal server implementing this fifo. This is asymmetrical in that the first user to access a fifo will implement it, and thus pay the costs for it. But it does not seem to cause any particular problems in implementing the POSIX semantics.

The GNU C library can contact /servers/socket/pflocal to implement socketpair, or start a pflocal server for this task's exclusive use if that node does not exist.

All this are optimizations: It should work to have one pflocal process for each socketpair. However, performance should be better with a shared pflocal server, one per user.

6.4 Pipes

Pipes are implemented using `socketpair()`, that means as unnamed pair of Unix Domain Sockets. The `pflocal` server will support this by implementing pipe semantics on the socketpair if requested.

> It was considered to use shared memory for the pipe implementation. But we are not aware of a lock-free protocol using shared memory with multiple readers and multiple writers. It might be possible, but it is not obvious if that would be faster: Pipes are normally used with `read()` and `write()`, so the data has to be copied from and to the supplied buffer. This can be done efficiently in L4 even across address spaces using string items. In the implementation using sockets, the `pflocal` server handles concurrent read and write accesses with mutual exclusion.

6.5 Filesystems

6.5.1 Directory lookup across filesystems

The Hurd has the ability to let users mount filesystems and other servers providing a filesystem-like interface. Such filesystem servers are called translators. In the Hurd on GNU Mach, the parent filesystem would automatically start up such translators from passive translator settings in the inode. It would then block until the child filesystem sends a message to its bootstrap port (provided by the parent fs) with its root directory port. This root directory port can then be given to any client looking up the translated node.

There are several things wrong with this scheme, which becomes apparent in the Hurd on L4. The parent filesystem must be careful to not block on creating the child filesystem task. It must also be careful to not block on receiving any acknowledgement or startup message from it. Furthermore, it can not accept the root directory capability from the child filesystem and forward it to clients, as they are potentially not trusted.

The latter problem can be solved the following way: The filesystem knows about the server thread in the child filesystem. It also implements an authentication capability that represents the ability to access the child filesystem. This capability is also given to the child filesystem at startup (or when it attaches itself to the parent filesystem). On client dir_lookup, the parent filesystem can return the server_thread and the authentication capability to the client. The client can use that to initiate a connection with the child filesystem (by first building up a connection, then sending the authentication capability from the parent filesystem, and receiving a root directory capability in exchange).

> There is a race here. If the child filesystem dies and the parent filesystem processes the task death notification and releases the task info cap for the child before the user acquires its own task info cap for the child, then an imposter might be able to pretend to be the child filesystem for the client.
>
> This race can only be avoided by a more complex protocol:

Variant 1: The user has to acquire the task info cap for the child fs, and then it has to perform the lookup again. If then the thread ID is for the task it got the task ID for in advance, it can go on. If not, it has to retry. This is not so good because a directory lookup is usually an expensive operation. However, it has the advantage of only slowing down the rare case.

Variant 2: The client creates an empty reference container in the task server, which can then be used by the server to fill in a reference to the child's task ID. However, the client has to create and destroy such a container for every filesystem where it excepts it could be redirected to another (that means: for all filesystems for which it does not use `O_NOTRANS`). This is quite an overhead to the common case.

```
<marcus> I have another idea
<marcus> the client does not give a container
<marcus> server sees child fs, no container -> returns O_NOTRANS node
<marcus> then client sees error, uses O_NOTRANS node, "" and container
<marcus> problem solved
<marcus> this seems to be the optimum
<neal> hmm.
<neal> So lazily supply a container.
<marcus> yeah
<neal> Hoping you won't need one.
<marcus> and the server helps you by doing as much as it can usefully
<neal> And that is the normal case.
<neal> Yeah, that seems reasonable.
<marcus> the trick is that the server won't fail completely
<marcus> it will give you at least the underlying node
```

The actual creation of the child filesystem can be performed much like a suid `exec`, just without any client to follow up with further capabilities and startup info. The only problem that remains is how the parent filesystem can know which thread in the child filesystem implements the initial handshake protocol for the clients to use. The only safe way here seems to be that the parent filesystem requires the child to use the main thread for that, or that the parent filesystem creates a second thread in the child at startup (passing its thread ID in the startup data), requiring that this second thread is used. In either case the parent filesystem will know the thread ID in advance because it created the thread in the first place. This looks a bit ugly, and violates good taste, so we might try to look for alternative solutions.

6.5.2 Reparenting

The Hurd on Mach contains a curious RPC, `file_reparent`, which allows you to create a new capability for the same node, with the difference that the new node will have a supplied capability as its parent node. A directory lookup of `..` on this new capability would return the provided parent capability.

This function is used by the `chroot()` function, which sets the parent node to the null capability to prevent escape from a `chroot()` environment. It is also used by the `firmlink` translator, which is a cross over of a symbolic and a hard link: It works like a hard link, but can be used across filesystems.

A firmlink is a dangerous thing. Because the filesystem will give no indication if the parent node it returns is provided by itself or some other, possibly untrusted filesystem, the user might follow the parent node to untrusted filesystems without being aware of it.

In the Hurd port to L4, the filesystem can not accept untrusted parent capabilities on behalf of the user anymore. The `chroot()` function is not difficult to implement anyway, as no real capability is required. The server can just be instructed to create a node with no parent node, and it can do that without problems. Nevertheless, we also want a secure version of the `firmlink` translator. This is possible if the same strategy is used as in cross filesystem lookups. The client registers a server thread as the handler for the parent node, and the filesystem returns a capability that can be used for authentication purposes. Now, the client still needs to connect this to the new parent node. Normally, the filesystem providing the new parent node will also not trust the other filesystem, and thus can not accept the capability that should be used for authentication purposes. So instead creating a direct link from the one filesystem to the other, the firmlink translator must act as a middle man, and redirect all accesses to the parent node first to itself, and then to the filesystem providing the parent node. For this, it must request a capability from that filesystem that can be used for authentication purposes when bootstrapping a connection, that allows such a bootstrapping client to access the parent node directly.

This also fixes the security issues, because now any move away from the filesystem providing the reparented node will explicitely go first to the `firmlink` translator, and then to the filesystem providing the parent node. The user can thus make an informed decision if it trusts the `firmlink` translator and the filesystem providing the parent node.

> This is a good example where the redesign of the IPC system forces us to fix a security issue and provides a deeper insight into the trust issues and how to solve them.

Chapter 7

Debugging

L4 does not support debugging. So every task has to implement a debug interface and implement debugging locally. gdb needs to be changed to make use of this interface. How to perform the required authentication, and how the debug thread is advertised to gdb, and how the debug interface should look like, are all open questions.

Chapter 8

Device Drivers

This section written by Peter De Schrijver and Daniel Wagner.

8.1 Requirements

- Performance: Speed is important!
- Portability: Framework should work on different architectures.
  Also: Useable in a not hurdisch environment with only small changes.
- Flexibility
- Convenient interfaces
- Consistency
- Safety: driver failure should have as minimal system impact as possible.

8.2 Overview

The framework consists of:

- Bus drivers
- Device drivers
- Service servers (plugin managers, ω_0, deva)

8.2.1 Layer of the drivers

The device driver framework consists only of the lower level drivers and doesn't need to have a complicated scheme for access control. This is because it should be possible to share devices, e.g. for neighbour Hurd. The authentication is done by installing a virtual driver in each OS/neighour Hurd. The driver framework trusts these virtual drivers. So it's possible for a non Hurdish system to use the driver framework just by implementing these virtual drivers.

Only threads which have registered as trusted are allowed to access device drivers. The check is simply done by checking the senders ID against a table of known threads.

8.2.2 Address spaces

Drivers always reside in their own AS. The overhead for cross AS IPC is small enough to do so.

8.2.3 Zero copying and DMA

It is assumed that there are no differences between physical memory pages. For example each physical memory page can be used for DMA transfers. Of course, older hardware like ISA devices can so not be supported.

Still some support for ISA devices like serial ports and PS/2 for keyboard is needed.

With this assumption, the device driver framework can be given any physical memory page for DMA operation. This physical memory page must be pinned down.

If an application wants to send or receive data to/from a device driver it has to tell the virtual driver the page on which the operation has to be executed. Since the application doesn't know the virtual-real memory mapping, it has to ask the physical memory manager for the real memory address of the page in question. If the page is not directly mapped from the physical memory manager the application asks the mapper (another application which has mapped this memory region to the first application) to resolve the mapping. This can be done recursively. Normally, this resolving of a mapping can be sped up using a cache services, since a small number of pages are reused very often.

With the scheme, the drivers do not have to take special care of zero copying if there is only one virtual driver. When there is more than one virtual driver pages have to be copied for all other virtual drivers.

8.2.4 Physical versus logical device view

The device driver framework will only offer a physical device view. Ie. it will be a tree with devices as the leaves connected by various bus technologies. Any logical view and naming persistence will have to be build on top of this (translator).

8.2.5 Things for the future

- Interaction with the task server (e.g. listings driver threads with ps,etc.)
- Powermanagement

8.3 Bus Drivers

A bus driver is responsible to manage the bus and provide access to devices connected to it. In practice it means a bus driver has to perform the following tasks:

- Handle hotplug events

  Busses which do not support hotplugging, will treated as if there is 1 insertion event for every device connected to it when the bus driver is started. Drivers which don't support autoprobing of devices will probably have to read some configuration data from a file[1] or if the driver is needed for bootstrapping configuration can be given as argument on its stack. In some cases the bus doesn't generate insertion/removal events, but can still support some form of hotplug functionality if the user tells the driver when a change to the bus configuration has happened (eg. SCSI).

- Configure client device drivers

  The bus driver should start the appropriate client device driver translator when an insertion event is detected. It should also provide the client device driver with all necessary configuration info, so it can access the device it needs. This configuration data typically consists of the bus addresses of the device and possibly IRQ numbers or DMA channel ID's. The device driver is loaded by the associated plugin manager.

- Provide access to devices

  This means the bus driver should be able to perform a bus transaction on behalf of a client device driver. In some cases this involves sending a message and waiting for reply (eg. SCSI, USB, IEEE 1394, Fibre Channel,...). The driver should provide send/receive message primitives in this case. In

[1] It might be a good idea, if the device driver has no notion how the configuraiton is stored. It just asks the bus driver which should know how to get the configuration.

other cases devices on the bus can be accessed by memory accesses or by using special I/O instructions. In this case the driver should provide mapping and unmapping primitives so a client device driver can get access to the memory range or is allowed to access the I/O addresses. The client device driver should use a library, which is bus dependant, to access the device on the bus. This library hides the platform specific details of accessing the bus.

- Rescans

  Furthermore the bus driver must also support rescans for hardware. It might be that not all drivers are found during bootstrapping and hence later on drivers could be loaded. This is done by generating new attach notification, which are sent to the bus's plugin manager. The plugin manager then loads a new driver, if possible. A probe funtion is not needed since all supported hardware can be identified by vendor/device identification (unlike ISA hardware). For hardware busses which don't support such identification only static configuration is possible (configuration scripts etc.)

8.3.1 Root bus driver

The root bus is the entrypoint to look up devices.

8.3.2 Generic Bus Driver

Operations:

- notify (attach, detach)
- string enumerate

8.3.3 ISA Bus Driver

Inherits from:

- Generic Bus Driver

Operations:

- (none)

8.3.4 PCI Bus Driver

Inherits from:

- Generic Bus Driver

Operations:

- map_mmio: map a PCI BAR for MMIO
- map_io: map a PCI BAR for I/O
- map_mem: map a PCI BAR for memory
- read_mmio_8,16,32,64: read from a MMIO register
- write_mmio_8,16,32,64: write to a MMIO register
- read_io_8,16,32,64: read from an IO register
- write_io_8,16,32,64: write to an IO register
- read_config_8,16,32,?: read from a PCI config register
- write_config_8,16,32,?: write to a PCI config register
- alloc_dma_mem(for non zero copying): allocate main memory useable for DMA
- free_dma_mem (for non zero copying): free main memory useable for DMA
- prepare_dma_read: write back CPU cachelines for DMAable memory area
- sync_dma_write: discard CPU cachelines for DMAable memory area
- alloc_consistent_mem: allocate memory which is consistent between CPU and device
- free_consistent_mem: free memory which is consistent between CPU and device
- get_irq_mapping (A,B,C,D): get the IRQ matching the INT(A,B,C,D) line

8.4 Device Drivers

8.4.1 Classes

- character: This the standard tty as known in the Unix environment.
- block

- human input: Keyboard, mouse, ...
- packet switched network
- circuit switched network
- framebuffer
- streaming audio
- streaming video
- solid state storage: flash memory

8.4.2 Human input devices (HID) and the console

The HIDs and the console are critical for user interaction with the system. Furthmore, the console should be working as soons as possible to give feedback. Log messages which are send to the console before the hardware has been initialized should be buffered.

8.4.3 Generic Device Driver

Operations:

- init : prepare hardware for use
- start : start normal operation
- stop : stop normal operation
- deinit : shutdown hardware
- change_irq_peer : change peer thread to propagate irq message to.

8.4.4 ISA Devices

Inherits from:

- Generic Device Driver

Supported devices

- Keyboard (ps2)
- Serial port (mainly for debugging purposses)

8.4.5 PCI Devices

Inherits from:

- Generic Device Driver

Supported devices:

- block devices

8.5 Service Servers

8.5.1 Plugin Manager

Each bus driver has a handle/reference to which insert/remove events are send. The owner of the handle/refence must then take appropriate action like loading the drivers. These actors are called plugin managers.

The plugin manager is also the pager for the loaded driver.

> Obviously, the plugin manager needs some sort of exec format support. Maybe it's own ELF loader.

8.5.2 Deva

Deva stands for *Device Access Server*. This server implements basic services for the device driver framework like thread creation, thread deletion, etc. The device driver framework itself doesn't depend on any Hurd code. The interaction with the Hurd system will be abstracted by deva.

Which services must deva provide:

- task/thread manipulation (create, deletion)
- memory (de)allocation (virtual, physical)
- io ports
- driver (un)loading
- bootstrapping

8.5.3 ω_0

ω_0 is a system-central IRQ-logic server. It runs in the privileged AS space in order to be allowed rerouting IRQ IPC.

If an IRQ is shared between several devices, the drivers are daisy chained and have to notify their peers if an IRQ IPC has arrived.

For more details see http://os.inf.tu-dresden.de/~hohmuth/prj/omega0.ps.gz

Operations:

- attach_irq : attach an ISR thread to the IRQ
- detach_irq : detach an ISR thread from the IRQ

8.6 Resource Management

8.6.1 IRQ handling

IRQ based interrupt vectors

Some CPU architectures (eg 68k, IA32) can directly jump to an interrupt vector depending on the IRQ number. This is typically the case on CISC CPU's. In this case there is some priorization scheme. On IA32 for example, the lowest IRQ number has the highest priority. Sometimes the priorities are programmable. Most RISC CPU's have only a few interrupt vectors which are connected external IRQs. (typically 1 or 2). This means the IRQ handler should read a register in the interrupt controller to determine which IRQ handler has to be executed. Sometimes the hardware assists here by providing a register which indicates the highest priority interrupt according to some (programmable) scheme.

IRQ acknowlegdement

The IRQ acknowledgement is done in two steps. First inform the hardware about the successful IRQ acceptance. Then inform the ISRs about the IRQ event.

Edge versus level triggered IRQs

Edge triggered IRQs typically don't need explicit acknowledgment by the CPU at the device level. You can just acknowledge them at the interrupt controller level. Level triggered IRQs typically need to explicitly acknowledged by the CPU at the device level. The CPU has to read or write a register from the IRQ generating peripheral to make the IRQ go away. If this is not done, the IRQ handler will be reentered immediatly after it ended, effectively creating an endless loop. Another way of preventing this would be to mask the IRQ.

Multiple interrupt controllers

Some systems have multiple interrupt controllers in cascade. This is for example the case on a PC, where you have 2 8259 interrupt controllers. The second controller is connected to the IRQ 2 pin of the first controller. It is also common in non PC systems which still use some standard PC components such as a Super IO controller. In this case the 2 8259's are connected to 1 pin of the primary interrupt controller. Important for the software here is that you need to acknowledge IRQ's at each controller. So to acknowledge an IRQ from the second 8259 connected to the first 8259 connected to another interrupt controller, you have to give an ACK command to each of those controllers. Another import fact is that on the PC architecture the order of the ACKs is important.

Shared IRQs

Some systems have shared IRQs. In this case the IRQ handler has to look at all devices using the same IRQ...

IRQ priorities

All IRQs on L4 have priorities, so if an IRQ occurs any IRQ lower then the first IRQ will be blocked until the first IRQ has been acknowlegded. ISR priorities must much the hardware priority (danger of priority inversion). Furthermore the IRQ acknowledgment order is important.

The 8259 also supports a specific IRQ acknowledge iirc. But, this scheme does not work in most level triggered IRQ environments. In these environments you must acknowledge (or mask) the IRQ before leaving the IRQ handler, otherwise the CPU will immediately reenter the IRQ handler, effectively creating an endless loop. In this case L4 would have to mask the IRQ. The IRQ thread would have to unmask it after acknowledgement and processing.

IRQ handling by L4/x86

The L4 kernel does handle IRQ acknowlegdment.

8.6.2 Memory

If no physical memory pages are provided by the OS the device driver framework alloces pages from the physical memory manager. The device driver framework has at no point of time to handle any virtual to physical page mapping.

8.7 Bootstrapping

The device driver framework will be started by deva, which is started by wortel. All drivers and servers (e.g. the plugin manager) are stored in a archive which will be extracted by deva.

8.7.1 deva

For bootstrapping deva will only have a subset of drivers ready. As soon the filesystem runs deva can ask for drivers from the harddisk. If new drivers are available it has to inform the plugin manager to ask for unresolved drivers again.

Deva starts as first task a plugin server. The plugin server does then the rest of the bootstrapping process.

8.7.2 Plugin Manager

A Plugin manager handles driver loading for devices. It asks deva for drivers.

The first plugin server does also some bootstrapping. First, it starts the root bus driver.

8.8 Order of implementation

1. deva, plugin manager
2. root bus server
3. pci bus
4. isa bus
5. serial port (isa bus)
6. console

8.9 Scenarios

8.9.1 Insert Event

If a simple hardware device is found the ddf will load a driver for the new hardware device as follows (see Figure 8.1):

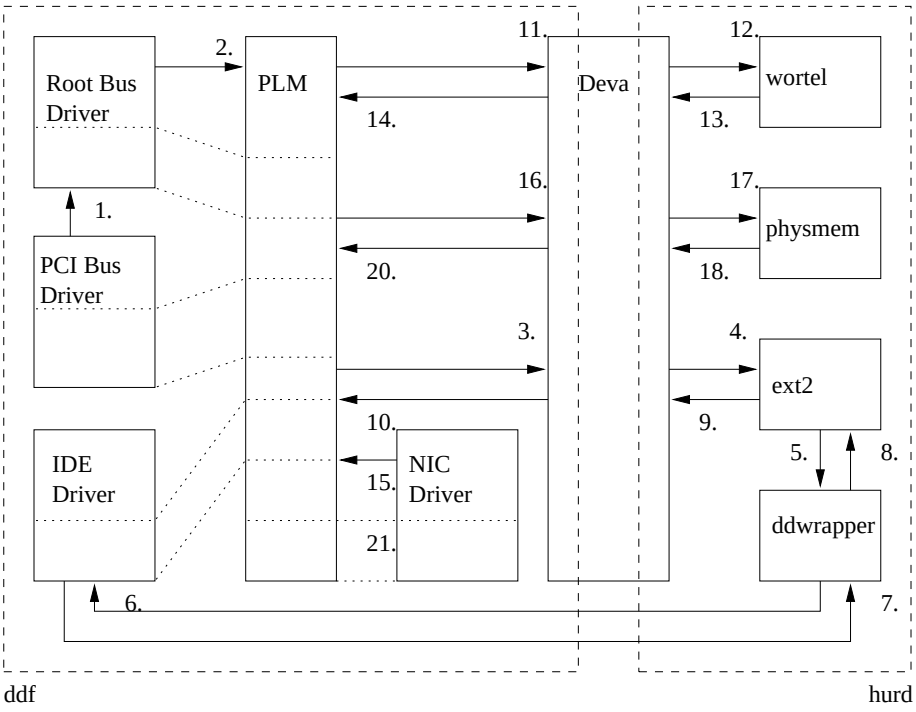

Figure 8.1: A new hardware device is detected (a network card) by the PCI root bus driver. The PCI root bus driver initiates the loading of the correct driver for the new hardware device.

1. The PCI Bus Driver detects a hardware device for which no driver has been loaded yet. It generates an insert event which it sends to one (all?) registered entity. The interface for the event handler has not been decided yet.
2. The Root Bus Driver receives the event signal. Note it is not necessary that the Root Bus Driver handles the insert signal for all drivers. It forwards the signal to the/a Plugin Manager (PLM).
3. The/a Plugin Manager (PLM) asks Deva to load the driver binary for the new device.
4. Deva forwards the loading request to the ext2 filesystem process. During bootstrapping Deva will handle the request by itself. Deva has an archive of drivers loaded by grub.
5. The ext2 process decides where it finds the device driver binary (block address)
6. The ddwrapper (device driver wrapper) forwards the read call from the ext2 process to the IDE Driver.
7. After checking if the caller is allowed start a read command, the IDE Driver reads the device driver from the disk.
8. The IDE Driver returns the data.
9. ddwrapper returns the data. XXX This might be wrong. IFRC, the data is returned in a container and only the handle of the container is transfered.
10. Ext2 returns the device driver (data).
11. Deva returns the device driver (data).
12. Ask Deva to create a new address space.
13. Deva asks wortel to create new address space.
14. wortel returns “a new address space”.
15. Deva returns “a new address space”.
16. PLM is registered as pagefault handler for the new driver address space. The bootstrap thread starts to run and generates a page fault.
17. PLM asks Deva for memory.
18. Deva asks physmem for memory.
19. physmem returns memory pages.
20. Deva returns memory pages.
21. PLM maps the device driver binary into the address space of the new driver.

Figure 8.2: For the new NIC driver a specialised plugin manager is loaded first.

8.9.2 Several Plugin Managers

For certain drivers it makes sense to have specialised plugin managers. The default plugin manger (dPLM) has to be asked to create a new plugin manager. It is loaded like a normal driver. The default plugin manager will also act as pager for the new plugin manager. When the new plugin manager is activated it registers itself to the Deva as new plugin manager. Deva will send all signals/messages from outside of the ddf to all registered plugin managers.