Building an SSI Cluster with GNU/Hurd

Brent Baccala

May 2019

Contents

1 Introduction

Operating system designers have tried for years to leverage the power of multiple computers and develop cluster operating systems. Notable attempts include VAX/VMS, MOSIX, Beowulf, Plan 9, ScaleMP, and Hadoop.

Some of these designs, like Beowulf and Hadoop, are built using libraries that allow programs which use those libraries to run distributed over multiple, otherwise independent, computers. Others, such as VAX/VMS, Plan 9, and ScaleMP, aspire to present the cluster as if it were a single machine – a Single System Image (SSI) – allowing unmodified programs to exploit cluster parallelism.

GNU/Hurd is an operating system based on the Mach kernel, which was designed for use in a cluster environment. Hurd is designed around a set of servers that communicate primarily by remote procedure calls mediated by the Mach kernel, making it a natural choice to build a POSIX-compatible SSI cluster. Development on Hurd has progressed slowly, but a Debian port has been available since 2013, and approximately three quarters of the Debian packages compile and run on Hurd.

Hurd needs several enhancements to operate as an SSI cluster. First, Mach messages must be transported over network links, a function provided by yet another Hurd server, called `netmsg`. Next, distributed shared memory must be implemented, without which we can't achieve full POSIX compatibility. Fortunately, Mach's memory management subsystem was designed to support distributed shared memory, and Hurd's interface to Mach's memory management subsystem is compartmentalized in a C library called `libpager`.

`libpager` is a crucial library. It provides the cache coherence logic necessary to implement inter-node shared memory, whether disk-based or not. It is used, for example, by the filesystem servers to implement `mmap()`. Hurd's original `libpager`, written in the mid-1990s, only supported a single client (a client being a Mach kernel), and was thus suitable for single node operation, but its API was designed to eventually support multiple clients. To support a multi-node Hurd cluster, a new `libpager` was written to support multiple clients (i.e, multiple Mach kernels). This allows both inter-node `mmap()` as well as inter-node POSIX shared memory.

Finally, Hurd's C library needs to be modified to allow cross-node forks, and Hurd's authentication mechanism needs to be extended as well.

The code described in this paper is available at `https://github.com/BrentBaccala/hurd`.

Hurd's `netmsg` server and the `libpager` library have been developed to the point where Hurd could be usable as a cluster operating system, though the current Debian distribution doesn't use this functionality. No cluster enhancements have yet been done on the C library or the authentication mechanism. At the time of this writing (2019), the major barriers to building a Hurd cluster are the 32-bit address space and the lack of SMP support.

2 Introduction to Mach Port-Based IPC

Mach inter-process communications is primarily done using variable-sized datagram messages that are transmitted reliably and in-order to queues from which tasks can receive them. In addition to opaque data (generally, but not necessarily interpreted as remote procedure calls), Mach messages can contain two more sophisticated structures. The first are port rights, which come in two basic flavors. Send rights give a task the ability to transmit messages to a queue, and receive rights give a task the ability to receive messages from a queue. A Mach message can transport send or receive rights for Mach ports.

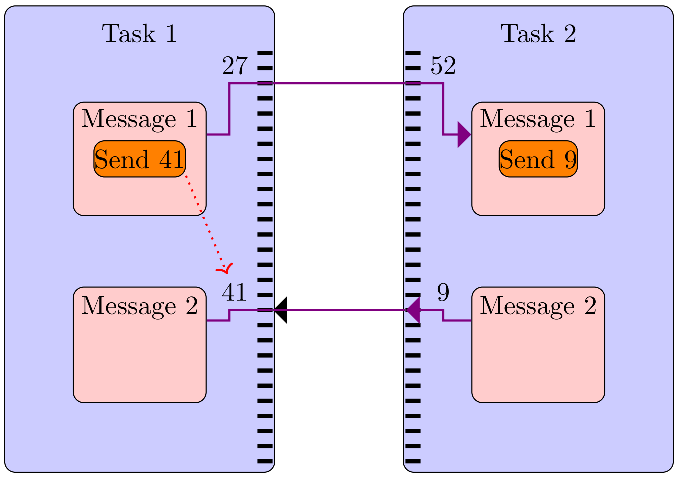

Figure 1: Transferring a send right

In Figure 1, Process 1 sends Message 1 to Process 2 containing a send right to the queue known to Process 1 as port 41, which could have been created by Process 1, or received from another task. The Mach kernel translates this into a send right to a previously unused port 9 in Process 2's port number space. Process 2 can now send a message to its port 9, which will be received by Process 1 on port 41.

This is basically the only way in which a process can obtain Mach ports since, much like POSIX file descriptors, port numbers are local to a process and interpreted by the kernel, so there is no way that a process can fabricate another process's port number, just as a POSIX process can not fabricate another process's file descriptor. A process is provided a few standard ports when it is initialized, one of which being a port with send rights that represents the root of the file system. By sending filename lookup requests to this port, the process can traverse the entire file system tree, each node of which is represented by a different port. The hurd file system is thus a hierarchy of ports, most of which recognize a standard set of RPCs performing file operations. The primary way for a program to advertise its services in Hurd is for it to listen for messages on a port that it attaches to a name in the file system.

All Hurd authentication is based on port rights. For example, a file system server may give out a send right which allows access to a particular file, simply by answering RPC messages transmitted to that port. The process receiving the send right is free to pass it to another process, which would then obtain the same rights to access the file. Note that since the first process would be free to proxy any requests to access the file, this isn't a security flaw. In fact, the process *can* proxy the requests by creating a new port of its own, handling out a send right to that port, and relaying message to the file system server, which would allow it to filter access to the file. This is roughly how file systems are implemented in Hurd. The file system server has read/write access to an entire disk partition, but gives out send rights that only allow access to individual files.

The second additional structure in a Mach message is out-of-band data. A Mach message can include pointer/length pairs to regions of memory, along with a flag indicating whether those regions should be deallocated when the message is sent. If a region is not deallocated, then it behaves much like a large block of opaque data copied in the Mach message. However, if a region is deallocated, then there is no need to actually copy the data in the region, as hardware memory management tables can be used to simply map it into the receiving task's memory space. This allows large blocks of memory to be efficiently transferred in Mach messages, and forms the basis for Mach's memory management system.

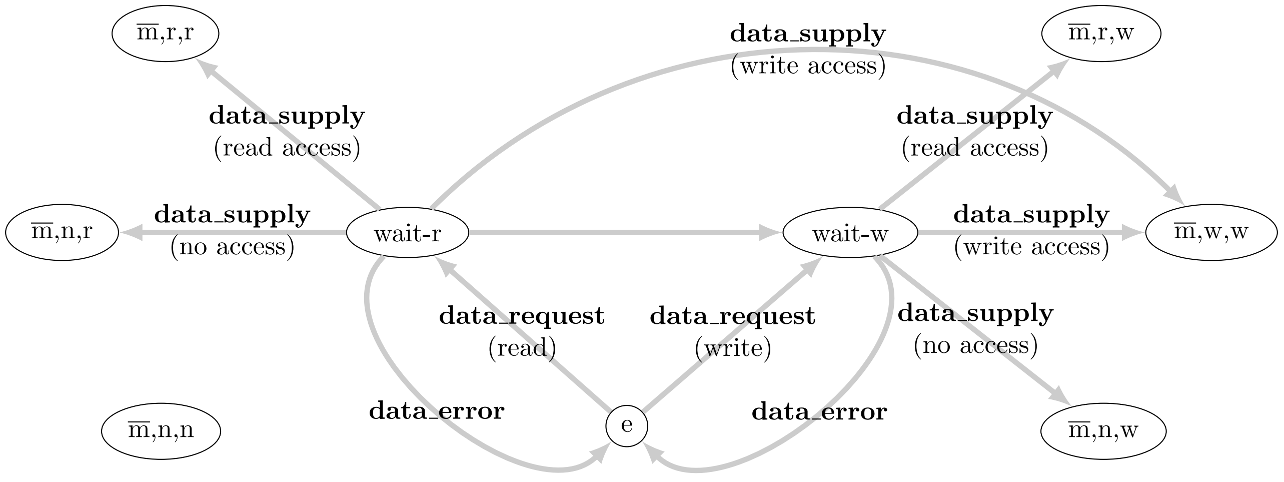

Figure 2: Page-in transitions. Adapted from Figure 6 in [2]

3 Introduction to Mach Memory Management

Mach's memory management was specifically designed to support multi-node clusters. Mach's model of memory management is that memory is managed by user-space processes, and system memory is a cache of that memory. Any user-space process that implements Mach's memory management protocol can be mapped into a process's address space. When the process attempts to access memory, a page fault is triggered which causes Mach to request the page from the user-space process using a well documented protocol. The faulting thread remains suspended until the page has been supplied to the kernel. The current kernel implementation requests pages one at a time, as the process attempts to access them, with no attempt to anticipate future page accesses.

Each virtual memory page in the Mach kernel can be in one of fifteen different states: empty, waiting for either read or write access, and twelve different states for pages in memory, depending on whether the page has been modified or not, the current access, and the desired access. Figure 2 shows the transitions involved in paging in; [2] uses five similar diagrams to completely illustrate all of the Mach memory manager's possible transitions.

Mach's ability to attach out-of-band memory to messages is thus a crucial component in the efficient implementation of its memory management. A page is typically supplied to kernel by attaching it to a `memory_object_data_supply` message. No copying needs be done, as the page is transferred to kernel control simply by adjusting bits in the hardware memory management tables. Likewise, transferring a modified page from the kernel to a user-space memory manager is done by attaching it to a Mach message from the kernel.

For example, consider how Hurd handles an `mmap()` call (Figure 3). A program uses directory lookup calls to obtain a file port, which is a send right to a port managed by a file server, and corresponds to a standard POSIX file descriptor. It then uses the `io_map` RPC to obtain another send right, also to a port managed by the file server, which implements the Mach memory management protocol. It passes that send right to the kernel, using a `vm_map` RPC, indicating that it wishes to map that memory object (i.e, the file), into its memory space. The kernel, after a `memory_object_init/ memory_object_ready` handshake with the file server, creates a new memory region in the program's address space, initially with no physical memory assigned to it, and returns that address to the program.

As soon as the program attempts to access this region of memory in any way (Figure 4), a page fault is triggered, which causes the kernel to send an `memory_object_data_request` message to the file server. The file server responds by reading the required page from disk (if necessary) and passing it to the kernel by attaching it to a `memory_object_data_supply` message. The kernel, now managing this region of memory, maps it to the correct page in the previously assigned region of memory, and allows the program to continue execution.

If the file server had instead replied with `memory_object_data_error`, the program would have received a seg fault instead. If another program maps the same file, the kernel will not request duplicate copies of the pages, but will arrange for the physical memory to be shared between the processes. Also, although the kernel now manages the memory, the file server can request it back at any time; the model is that the file server manages the memory, and the kernel is only caching it. If the kernel runs short on physical memory, it can either discard the page, if it is unmodified and if the file server didn't set the "precious" flag, or will return it to the file server.

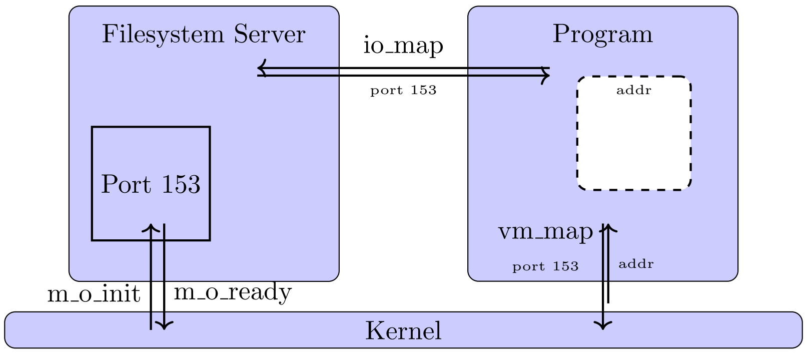

Figure 3: Initial mapping of a Mach memory object

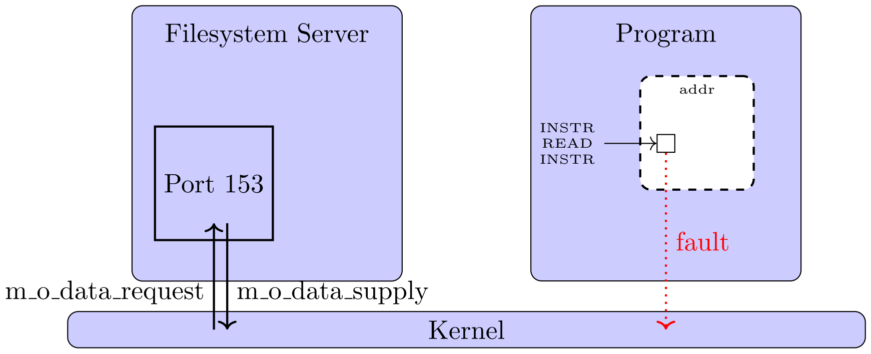

Figure 4: Initial access of a Mach memory object

Of course, it can be inefficient to keep a copy of a page in user space while its also in use by the kernel. Pages can be flagged "precious", indicating to the kernel that they must be returned to the user-space process and can not be simply discarded. This frees the user-space memory manager from needing to keep a copy of a page that has been supplied to the kernel. Otherwise, non-"precious" pages can be freely discarded by the kernel when they are no longer needed.

Each file corresponds to a separate memory object, and separate memory objects get separate control ports in the kernel. A future enhancement could be for the kernel to use a single control port for all of its memory objects, allowing `libpager` to reduce the number of these combination structures, if different files have the same usage pattern (a program and some of its shared libraries, for example).

The flexibility of allowing user-space processes to implement the memory management protocol does create unique failure and abuse scenarios. A program can implement the memory management protocol, map it into a region of memory, transfer that memory to another process, then trigger a fault by responding with memory_object_data_error to any access attempt. The other program, if not careful, could fail with a SIGSEGV just for trying to read the memory it received in a message! Thus, in Hurd, user-space programs, especially system servers, must exhibit the same kind of care when receiving memory regions in Mach messages as Linux kernel routines must consider when receiving memory from a user-space program.

4 Introduction to Hurd

The Free Software Foundation chose Mach in the early 1990s as the kernel for what was envisioned as its flagship operating system, Hurd. The name is an allusion to the collection of servers required to implement the core functionality usually found in an operating system kernel, but which are transferred to user space in the Mach design. Linux ultimately eclipsed Hurd with a faster development process and a simpler, more traditional design that packed functionality into a large, monolithic kernel. Hurd development has continued, however, and it now available as a Debian port, though without the bevy of features and device drivers that have been packed into Linux over the years.

For example, the file system, usually implemented as part of the kernel, is mostly decoupled in Hurd. In principle, a server called `pci-arbiter` is responsible for mapping PCI devices, such as a SCSI controller, into the memory space of user space programs that manage them. In reality, GNU Mach (the version of Mach used by Hurd) contains a significant number of device drivers ripped from an older version of the Linux kernel, so we do have some device drivers (including our disk controllers) in the kernel.

Aside from reading and writing raw disk blocks, however, the file system implementation is completely done in user-space servers, called *translators* in Hurd nomenclature. The most important file server is `ext2fs`, which interprets and manages an ext2 file system. The program interprets `dir_lookup` RPCs and typically responds to them by creating a port (in its own port space) that will represent a file handle, then replies to the `dir_lookup` message with a send right to the port. The file handle port handles messages like `io_stat`, `io_read`, and `io_write` to carry out basic I/O, as well as messages like `file_exec` (if the file is executable) and `file_chown` to change the file's ownership, and some more exotic messages like `io_async`, which requests that the fileserver send messages to the process when I/O is available (typically used for sockets rather than files), and `io_reauthenticate`, which tells the file server to drop this port's association with a user and perform an authentication exchange.

One of the advantages of this design is that mounting a loopback device is a non-privileged operation. All that is required is to run another copy of `ext2fs`, pass it the name of the file containing the loopback filesystem, and attach it to a name in the directory hierarchy. Voila! All future `dir_lookup` calls into that part of the filesystem will be relayed to the new `ext2fs`. Likewise, an `iso9660fs` translator is available that allows CD and DVD images to be loopback mounted without special permissions.

A great deal of Hurd's POSIX compatible API is implemented in the standard C library. It is here, for example, that `read()` is implemented using an `io_read` RPC exchange over a Mach port linked to a Hurd server. In addition to a program's standard threads of execution, each program also has a thread, managed by code in the C library, that listens on a message port. POSIX signals, for example, are delivered by sending message to this port. Also, while the Linux kernel can be queried to find a lot of information about a process, such as the command string used to invoke it, the user and group permissions it is running with, its environment, and its open file descriptors, among other things, Hurd does not track any of this information in the kernel. In Hurd, much of this information is obtained by querying the process directly, using its message port, though this has the disadvantage of depending on the correct operation of the program. In particular, it is fairly easy to experience a "hung" `ps` in Hurd, caused by a deadlocked program that is not responding to its message port.

A sampling of some major Hurd servers include the following:

`exec` is responsible for executing programs. It parses the ELF binary format and performs the necessary relocations and memory mappings to prepare a new Mach task and make it ready to run. If the program begins with the hash-bang sequence (`#!`), `exec` finds a suitable interpreter to run the program. Users can specify their own `exec` servers, so, for example, if a user wanted to use `qemu` to emulate a different architecture, no kernel changes would be required, only a custom `exec` server.

`auth` implements POSIX authentication based on users and groups. A Hurd process will be passed a send right to the `auth` server when it starts, and it is the possession of this send right that allows the process access to protected resources. The naive idea of simply passing this send right to a server would grant the server all of the process's access rights, so this is not done. Instead, the process creates a new port when it desires to authenticate to a server and passes send rights to that port to both the server and the `auth` server. The server then passes its copy of the send right to the `auth` server, which matches the two against each other and confirms that the desired access is permitted. Matching the send rights is trivial on a single node, since Mach guarantees that rights to the same port will always appear on the same port number.

`proc` tracks processes in a POSIX-compatible way. For example, the Mach kernel maintains no sense of a parent/child relationship between processes, or any concept of process ownership by a user or group. Killing a process (or sending any signal to a process) requires an authentication step to obtain the process's message port; this authentication is provided by the `proc` server. Also, some information about a process (like its command name and environment) needs to be provided without any authentication to allow a program like `ps` to operate; the `proc` server provides this functionality, as well.

`pfinet` is the Hurd's TCP/IP networking stack. Networking functions in the C library trigger RPC calls to this server.

5 `netmsg`

Due to its heavy reliance on message passing between user-space servers, Hurd is a natural choice to build an SSI cluster, if we can transport Mach messages over a network. Several programs have been written over the years to do this; all collectively go under the name of the "netmsg server", or just `netmsg`.

I wrote Hurd's current `netmsg` server in 2016. It operates over a single unsecured TCP/IP connection and has many limitations, but works well enough to implement distributed shared virtual memory. In server mode, it presents a single node, by default the root of the filesystem, to any clients that connect. In client mode, it acts as a translator, allowing a remote server's exported node to be mounted in the local filesystem. `netmsg` works by copying Mach messages almost verbatim across the network connection. Only the Mach port numbers are changed. Out-of-band memory is simply copied across the network connection, then deallocated.

```
root@qemu-hurd-1:~/github/netmsg# ./netmsg -s ~ &
[1] 761
root@qemu-hurd-1:~/github/netmsg#
```

```
root@qemu-hurd-2:~/netmsg/netmsg# settrans -ac node ./netmsg 192.168.55.101
root@qemu-hurd-2:~/netmsg/netmsg# ls node
github
hurd-0.9.git20190331
hurd_0.9.git20190331-5.debian.tar.bz2
hurd_0.9.git20190331-5.dsc
hurd_0.9.git20190331.orig-eth-filter.tar.bz2
hurd_0.9.git20190331.orig-libddekit.tar.bz2
hurd_0.9.git20190331.orig-libdde-linux26.tar.bz2
hurd_0.9.git20190331.orig-libmachdev.tar.bz2
hurd_0.9.git20190331.orig.tar.bz2
old
translator_primer
root@qemu-hurd-2:~/netmsg/netmsg#
```

Figure 5: A `netmsg` session

For example, Figure 5 shows a simple `netmsg` session between two Hurd virtual machines. On `qemu-hurd-1`, I started a `netmsg` server (`-s`) sharing `root`'s home directory, listening on `netmsg`'s default TCP port 2345. Then, on `qemu-hurd-2`, I attached an active (`-a`) translator to `node`, creating it (`-c`) if it didn't already exist, running `netmsg` as a client connecting to `qemu-hurd-1`'s IP address. Running `ls node` shows the remote directory.

Thus, we have a remote file system, but it's important to realize that this remote file system supports full Mach messaging. Reading a file, for example, is done by sending a Hurd `io_read` message that is relayed across the network, then received, processed, and answered by the remote Hurd server managing that file. This is in contrast to remote file systems like NFS that don't transport Mach messages, and only allow a subset of Hurd RPCs which can translated into NFS messages. In particular, since Mach virtual memory is implemented using the Mach messaging protocol, `netmsg` allows a remote file to be `mmap`'ed into local memory.

While usable, this version of `netmsg` has many flaws. Transporting all of the Mach messages over a single TCP/IP session provides reliable, in-order delivery as guaranteed by Mach, but at the cost of serializing all messages, and pausing all communication to wait for a TCP re-transmission, when only individual Mach ports need to be serialized. Also, using the system's TCP/IP stack requires context switches between `netmsg` and the TCP/IP server (a user-space program, like so much else in Hurd) for every Mach message being relayed. The current scheme doesn't do a good job of handling more than two nodes, because there's no mechanism to detect a cycle, which could result in a lot of unnecessary network chatter. Finally, it's become clear that the simplistic means of managing ports creates unavoidable race conditions.

If you load a stock[1] Hurd distribution and run `netmsg`[2] as shown above, you get a remote file system that lets you read and write files, but executing remote files doesn't work – the program just hangs. That's because Hurd, by default,[3] executes files not by reading them into memory, but by `mmap`'ing them into memory. This requires the *remote* file server's memory management code to service *two* clients – its local kernel, and the remote one that is now trying to `mmap` the file, and a stock Hurd file server can't do that. We need to upgrade `libpager`.

6 `libpager`

Hurd's implementation of the Mach memory management protocol is implemented in a library called `libpager`. The original `libpager`, written in the mid-1990s, only supported a single client (i.e, a single Mach kernel), and was thus suitable for single node operation. Its API, however, is flexible enough to allow a more sophisticated, multi-client `libpager` as a drop-in replacement. For dynamically linked servers, all that is required is to update the system shared library to

[1] "Stock" distributions are available at `https://cdimage.debian.org/cdimage/ports/latest/hurd-i386/`
[2] `netmsg` is available at `https://github.com/BrentBaccala/hurd`
[3] It's not too hard to change this behavior by changing the `exec` server.

obtain multi-node functionality. Note, however, that Hurd's root file system will almost always be implemented using a statically linked server, since it needs to start quite early in the boot process.

`libpager` acts as a translator between the Mach kernel and a server program that parses the on-disk structure of a file system. On one side, `libpager` implements the Mach memory management protocol interacting with the Mach kernel. On the other side, `libpager` presents an API to the server program, which is responsible for producing the pages. The APIs primary functions are as follows:

- `pager_read_page()`

  This routine is expected to allocate memory (unless it returns an error) containing the requested data and pass it to `libpager`, which is responsible for deallocating it (typically by passing it to the kernel). A common use scenario is that `pager_read_page()` reads a page from disk and returns it. `pager_read_page()` can be called from multiple threads to service multiple paging requests, and must implement its own thread safety. However, it is free to serialize itself with a global mutex, and this will not prevent `libpager` from servicing other requests.

- `pager_write_page()`

  This routine is expected to save a modified memory page just before `libpager` releases its last copy of the data. It does no memory allocation or deallocation. It receives allocated memory and leaves it allocated, with `libpager` responsible for dellocation, or passing it to the kernel, if a new request has arrived in the time required for `pager_write_page()` to complete.

- `pager_unlock_page()`

  This routine is called when a page that was previously retrieved for read-only access must now be promoted to read-write access.

- `pager_sync()` and `pager_sync_some()`

  This routine requests that copies of modified pages be obtained and submitted to `pager_write_page()`. Doesn't otherwise interfere with `libpager` operation.

- `pager_return()` and `pager_return_some()`

  This routine is similar to `pager_sync()`, but it requests that the kernel(s) discard their copies of the pages after returning them. Nothing prevents the kernels from immediately re-requesting copies of the pages, but those new requests will trigger calls to `pager_read_page()`.

- `pager_flush()` and `pager_flush_some()`

  This routine instructs the kernel to discard dirty pages without bothering to save them! Thus, the "sync", "return", and "flush" routines cover all possible combinations of two binary options: "discard pages" and "return pages":

	discard	return
`pager_sync()`		✓
`pager_return()`	✓	✓
`pager_flush()`	✓	

6.1 Configuration Options

Several `libpager` implementation choices exist, and perhaps should be configuration options, though they are currently hard-wired into the code.

- How long do we let clients hold a page if other clients are waiting for it?

  Currently we send a lock request immediately after supplying the page, which could create deadlock if the kernel returns pages without making any forward progress. If processes on two different nodes are both trying to write into a page, we might spend all of our time ping-ponging the page and forth without anything actually getting done.

  Check Mach kernel source to see what happens in this case. My guess is that if the process can be started immediately, it is, and this is only an issue if system load is high enough to prevent that.

- How to handle network partitions / failed nodes / malfunctioning kernels

  The CAP theorem states that a distributed system can't have all of three things: consistency, availability, partitioning. Right now, we sacrifice availability – if the network partitions, `libpager` can deadlock. If we introduce a timeout on lock requests, then we're sacrificing consistency.

  Currently we deadlock if a client doesn't respond; perhaps we should timeout while waiting for a client? How to pick the timeout value?

- Do we write a page back every time we handle it?

  Currently, we don't call `pager_write_page()` until a page is completely freed or sync'ed, but this can cause modified data to sit indefinitely in RAM when we'd rather save it to disk periodically.

  Note that if the server wishes to periodically sync data to disk, it can call `pager_sync()` on a timer thread. The question is whether we want to more proactively call `pager_write_page()` anytime we're handling a page that needs to be transferred between two clients.

- How do we obtain pages when we can read them from multiple sources?

  Currently, we read it again with `pager_read_page()`, but if the local kernel has a copy, it's probably faster to ask the kernel for a copy, since it's just a manner of flipping bits in a hardware page table to obtain a new "copy" of the data. Otherwise, we probably want to get the data from the local disk, but if the disk is very busy and we've got fast network connections, we might want to get that data from another node in the cluster.

  We don't have a good way of detecting which client is the local kernel, and the Mach memory management API doesn't provide a way to request a copy of an unmodified, non-"precious" page from the kernel. You can't just request that the kernel hand you back an unmodified page, except by flagging it "precious". That's not too bad if the kernel and the memory manager are both local on one node, but if you do this over the network, you're sending unnecessary copies of the memory pages around.

  How do we figure out which client is the local kernel? One possible solution is for `netmsg` to listen on `/servers/netmsg` for an RPC that accepts a send right and identifies which node currently hosts the corresponding receive right. [1] defined the `norma_port_location_hint` RPC that might be useful.

  If the kernel used the same control port for all of its pager operations (it currently doesn't), then we wouldn't have to make this query for every memory object; a single ext2fs translator, for example, could globally identify the location of all of the control ports, one for each node, and this information could be shared amongst all of that translator's pager objects.

  A corner case is what happens when the read fails with an error. Currently, we propagate that error to everything waiting on the page, even though we could request a copy of the page from one of the clients that still has it.

  Another possible future enhancement is to track how many calls to `pager_read_page()` are outstanding, and transition away from calling this routine if it appears to be overloaded.

- How to handle failed writes to the backing store?

  Do we keep the page around in memory to service future requests, and hope that maybe another write will succeed in the future, or discard the page? Perhaps keep it for a time? How to decide when to discard? Maybe only keep it if EAGAIN is returned?

  If we do discard the data, how do we handle future requests for that page? Currently, we return errors with no mechanism to recover.

  If we kept failing writes around indefinitely, that would provide a simple mechanism to implement a ramdisk – the read operation always returns an empty page and the write operation always fails.

- Are the pager_ functions thread-safe? Do we call them from another thread before they return?

  `pager_write_page()` is called in a protected manner – only from a single thread (but not always the same thread), and `libpager` waits until it returns before calling it again.

  Both `pager_read_page()` and `pager_unlock_page()` may be called reentrantly while another thread is waiting for another call to finish, but never on the same page.

6.2 The Pagemap

All Mach memory management operations operate on memory pages, so `libpager` needs to maintain a table that tracks the state of each page, called the pagemap. Hurd's original `libpager` had a relatively simple pagemap. The pagemap was an array of 16-bit shorts; each page had a single 16-bit entry, allocated as follows:

Two bits were allocated to each of two error codes. `ENOSPC` and `EDQUOT` were preserved; all other errors got collapsed into `EIO`. The remaining bits indicated if the page was in core, if we were waiting for it to be paged in or out to the backing store, and if the last attempt to write the data to the backing store had resulted in an error.

Clearly such a simple pagemap is inadequate for multi-client operation. For one thing, we could have a potentially unlimited number of clients with read-only copies of the page, as well as a potentially unlimited number of clients waiting on the page, so a fixed-size structure seems inadequate.

```
#define PM_WRITEWAIT  0x0200   /* queue wakeup once write is done */
#define PM_INIT       0x0100   /* data has been written */
#define PM_INCORE     0x0080   /* kernel might have a copy */
#define PM_PAGINGOUT  0x0040   /* being written to disk */
#define PM_PAGEINWAIT 0x0020   /* provide data back when write done */
#define PM_INVALID    0x0010   /* data on disk is irrevocably wrong */

#define PM_ERROR(byte) (((byte) & 0xc) >> 2)
#define PM_NEXTERROR(byte) ((byte) & 0x3)

enum page_errors
{
  PAGE_NOERR,
  PAGE_ENOSPC,
  PAGE_EIO,
  PAGE_EDQUOT,
};
```

Figure 6: Pagemap structure in *original* `libpager`

Nor is a separate structure for each page desirable if there are a large number of pages. We anticipate that many pages (say, all the pages in a single shared library) will exist in the same state, so there's one structure for each combination of its entries, with multiple pages pointing to it. Pagemaps can be quite large, so a pagemap of pointers suggests itself.

The new `libpager` is coded in C++, so there were several implementation options:

- A straight-up array of pointers

  Easy enough to understand, but then we have to code reference counting to know when those dynamic structures can be deallocated.
- An array of `shared_ptr`'s

  Simplifies the code, because it does all the reference counting for us, but at the cost of making the array four times bigger than it needs to be.
- A custom class with a clever copy constructor

  We just assign into the array and the copy constructor takes care of either finding an existing pagemap structure or creating a new one, at the cost of obscuring the fact that `pagemap[i] = new_client_list;` is far more complex than a simple assignment.

I used the clever copy constructor. The pagemap is an array of pointers to unique pagemap structures kept in a C++ `std::set` called `pagemap_set`. When working on a page, the main library code copies a pagemap entry into a temporary structure (`tmp_pagemap_entry`), modifies it, then assigns it back into the pagemap using an `operator=` that either finds a matching entry in `pagemap_set` and uses a pointer to it, or move-inserts `tmp_pagemap_entry` into `pagemap_set` and uses a pointer to it. No attempt is (currently) made to free unused entries in `pagemap_set`.

All the queues and lists are private, and we've got a bunch of methods to access them, allowing us to change the pagemap structure later if we want.

The pagemap is organized as an array with one entry for each virtual memory page in the memory object. In addition to the pagemap, there are several additional data structures maintained globally (i.e, for the entire memory object):

- The NEXTERROR list, a linked list of outstanding (error code, page, client) tuples
- A list of pages waiting to be written to backing store
- A list of outstanding lock requests
- A list of outstanding change requests

6.3 Access Contention

The primary data structures for mediating contention between clients are the ACCESSLIST and the WAITLIST in the pagemap. The ACCESSLIST can have multiple clients on it, if they all have read-only access. Write access can only be

```
struct pagemap_entry {
   /* ACCESSLIST
    * A list of clients that currently have access, along
    * with an indication if this is read-only or read-write access.
    */
   port_t clients_with_access[];
   bool write_access_granted;

   /* WAITLIST
    * A list of the clients waiting for access, in order of arrival,
    * and what kind of access they are waiting for (read or write)
    */
   port_t clients_waiting[];
   boolean clients_waiting_for_write[];

   /* PAGINGOUT
    * A flag that we set when we get a data return and start writing it
    * to backing store, and clear when the write is finished.
    */
   bool pagingout;

   /* ERROR
    * the last error code returned from a backing store operation
    * (hopefully KERN_SUCCESS)
    */
   kern_return_t error;

   /* INVALID
    * indicates that the backing store data is invalid because we got an
    * error return from a write attempt
    *
    * error returns on read or unlock operations do not set "invalid"
    */
   bool invalid;
};
```

Figure 7: The new pagemap structure

issued to one client at a time.

A client requesting access goes on a previously empty WAITLIST only when:

1. The client has requested any access and we have to start a page in. This happens when

   READ access requested - only READ clients on ACCESSLIST

   WRITE access requested - ACCESSLIST empty
2. The client has requested any access and we're waiting for the page to page out
3. The client has requested any access and another client (on ACCESSLIST) has WRITE access
4. The client has requested WRITE access and other clients (on ACCESSLIST) have any access

Putting a client on an empty WAITLIST triggers lock requests to all clients on the ACCESSLIST, with the exception of the requesting client itself, if it's already got read access and is requesting write access (an unlock request). Once the number of clients on the ACCESSLIST drops to zero (or one, in the unlock case), then the original request can be satisfied.

In the meantime, additional clients can be added to the WAITLIST, which basically triggers nothing. The rule is that if the WAITLIST is not empty, then we're already trying to revoke access to the clients on ACCESSLIST. Thus, the wheels have already been set in motion for the WAITLIST to be serviced, and we are free to add additional clients to the end of the WAITLIST. Remember, there's currently no guarantee that a client will have access for any period of time at all, so there's no time period when any clients will be on WAITLIST without an attempt being made to revoke existing access.

When a client with write access returns a page, we look to see if anything is on the WAITLIST. If so, the first client(s) on WAITLIST get the pages. If not, we flag that a write is in progress and start it by calling `pager_write_page()`. When it completes, we check the WAITLIST again to see if anything is now waiting. If so, we service the request. If not, we discard the data and notify the server (if it requested notifications) that the page has been evicted.

6.4 Error Handling

The three backing store callback functions can all return an error code.

Errors when reading from backing store are fairly straightforward – we got an error, so all we can do is propagate the error to the clients. Actually, if there are outstanding copies of the page we tried to read, we could obtain a copy from another client, but we don't do that currently. Errors when writing are a bit more complicated – we've still got a copy of the page, do we keep it around and hope that the write will complete successfully at some point in the future? We don't do that currently, either. Errors returned from an unlock operation should probably only affect the unlock request. Should they also affect future unlock operations on this page, or should failing unlocks be retried? Currently, a failing unlock is propagated to everything waiting on the page (both read and write requests); probably should only be sent to write requests.

Also, error returns on unlock requests are inconsistently recorded in the pagemap. The error return code in the pagemap is only really used if a write operation failed, in which case the page is discarded and future requests all return the error code. An exception is if a kernel kept a copy of the page whose write failed (i.e, there was a sync request that triggered the write), in which case future operations fail until something causes the kernel to send us the page again, when we retry the write.

Handling error on unlock requests is somewhat tricky. [1] states, in its **NOTES** on **memory_object_lock_request**:

> When a running thread requires an access that is currently prohibited, the kernel issues a **memory_object_data_unlock** call specifying the access required. The memory manager can then use **memory_object_lock_request** to relax its access restrictions on the data.
>
> To indicate that an unlock request is invalid (that is, requires permission that can never be granted), the memory manager must first flush the page. When the kernel requests the data again with the higher permission, the memory manager can indicate the error by responding with a call to **memory_object_data_error**.

Thus, when `pager_unlock_page()` returns an error, we can't simply relay that error code in a reply to the kernel. Instead, we must flush the page and wait for a **memory_object_data_request** from that client, at which point we return the error code from `pager_unlock_page()` rather than processing the data request normally. This is the function of the NEXTERROR list. In the original `libpager` design, it was encoded in the pagemap with two bits, since there was only one client, and all errors except ENOSPC and EDQUOT were collapsed into EIO. In the new design, NEXTERROR is maintained completely separate from the pagemap, largely due to the expectation that it will seldom be used.

For example, consider what happens if client A has read access to page, and is thus alone on ACCESSLIST. Client B now requests read access to the same page, so it goes on WAITLIST and we call `pager_read_page()`. Before this call completes, client A requests write access via an unlock request, so it goes on the WAITLIST. Finally, `pager_read_page()` returns an error. What do we do? We send an `memory_object_data_error` to client B, and a lock request to client A, with a queued NEXTERROR for client A.

6.5 Flushing Pages

Mach kernels are allowed to proactively flush pages unless the "precious" flag is specified, so we can't be certain that a client on a page's ACCESSLIST actually possesses that page. All we know for sure is that a client not on ACCESSLIST does not possess the page. This makes it somewhat difficult to compute ACCESSLIST after a `pager_flush()`.

Examining the Mach kernel code for `memory_object_lock_request`, we see that the kernel might block while looping over the pages (and unlocks the memory object while doing so), so for a multi-page lock request, we need to consider the possibility that part of the page range might be processed, then other kernel operations happen (including VM page operations) before the remaining part of the page range is processed and the lock request completes.

Consider the following scenario:

- five clients have read access to page 50 and are on its ACCESSLIST (WAITLIST is empty)

- `pager_flush()` is called on pages 0-100
- page 50 flushes on client A
- client A re-requests access to page 50, and it is supplied (ACCESSLIST and WAITLIST don't change)
- the flushes complete
- only client A now has access to page 50 and should be alone on its ACCESSLIST

This scenario shows that ACCESSLIST, WAITLIST and a FLUSHING flag aren't enough; we have to track which clients requested access during the flush to correctly compute ACCESSLIST at the end.

Flagging all of our pages "precious" (so the kernel always returns it to us) is tempting. It provides us with a mechanism for retrieving a local, in-memory page that we want to serve out over the network without having to read it again from disk. It also simplifies the flushing logic above, since we don't have to worry about whether the page got returned or not when we get the lock completed message – precious pages always get returned. Flagging a page "precious" isn't that big a deal for a local client, because "returning the page" just means flipping bits in the VM tables, but it's a lot heavier operation for a remote client, since the page will be transmitted across the network unnecessarily, so we don't do this.

6.6 Random Test Code

All kinds of race conditions can be envisioned for `libpager`. Just to name a few, multiple updated copies of a page could be received while waiting for a single call to `pager_write_page()` to complete; a client could send an unlock request that crosses paths with a lock request generated by `libpager`, and the client would then perceive the lock request as a response to its unlock request; a client already on the WAITLIST could flush and re-request access, going on the WAITLIST twice if we're not careful.

I wrote a test program that excercises `libpager` by making a pseudo-random string of pager requests. The idea was to run it repeatedly using various seed values, then if it detected a bug, it could be re-run with the same seed value in order to reproduce the bug. It never actually found any bugs.

There are five possible client operations:

1. request READ access
2. request WRITE access
3. request unlock READ → WRITE
4. flush page
5. service a message from the memory manager

If the client has no access to a page, then 1, 2, or 5 are possible. If the client has read access, then 3, 4, or 5 are allowed. With write access, 4 or 5 are permissible.

There are four possible translator operations:

1. discard request
2. return request
3. sync request
4. complete a pending pager_read, pager_write, or pager_unlock (which one? there could be several)

1, 2, and 3 can be synchronous or asynchronous.

The test program should (but currently doesn't) check to ensure that following conditions are met:

- write access to a page is exclusive to a single client

- pages get written back in-order (though skips are possible) (partially done)
- if a page is modified, nobody ever sees an earlier version of it
- all data requests are properly answered with a data supply or a data error (or data unavailable)
- all data unlock requests are properly answered with a data lock
- if a data lock requests a reply, make sure one is received

For example, a possible bug could be triggered by the following scenario: A single client has write access and is actively changing the page. `pager_write()` is relatively slow, and `pager_sync()` is called three times in rapid succession. The client returns data three times. The first data return triggers a `pager_write()`, and the next two data returns are queued waiting for the write to complete. After the first `pager_write()` completes, we need to ensure that either the next two writes are either processed in order, or the second write is dropped completely.

The test code would test for this scenario by generating the following sequence (A is the client; T is the translator):

A2 T4 (the read completes) A5 (the data_supply is processed) T3 T3 T3 (the syncs) A5 A5 A5 (the syncs are processed) T4 T4 T4 (the writes complete)

Another possible bug scenario runs like this: a single client holds two pages with write access. It returns one (perhaps because `pager_sync()` was called) which triggers a call to `pager_write_page()`. While `pager_write_page()` is running, both pages are returned in a single `memory_object_data_return` message. Since the first page is still writing, we sleep to wait for it to finish. Then the second page is requested again. We *should* serve the copy that's queued to write, but if we're not careful we'll call `pager_read_page()` on the second page.

The test code would test for this bug as follows: (one client, two pages A and B)

AB2 T4 (the read completes) TA3 A5 (the sync is processed) TAB2 A5 (the sync is processed) B2 (page request) T4 (the read completes)

It was an interesting idea, but didn't quite pan out as well as I had hoped. For one thing, the operations can't be generated with equal probability. Client operation 5 (service a message) has to occur fairly frequently, or a queue of pending messages will build up. Likewise, translator operation 4 (complete a pending operation), also has to occur fairly frequently. On the other hand, client operation 5 can't run if there are no pending messages, and translator operation 4 can't run if there are no pending requests, so there is some synchronization that has to occur. Finally, I'm not sure that `libpager` can be fully tested using anything other than actual Mach kernels.

7 Distributed Shared Memory

Two shared memory implementations are available on modern Linux systems – the older System V shared memory API and the newer POSIX API. POSIX shared memory has not yet been implemented for Hurd, but System V shared memory is implemented by mapping "files", corresponding to shared memory keys, in the `/dev/shm` portion of the filesystem tree. Linux uses a similar approach. Thus, a multi-client `libpager` enables the use of inter-node shared memory. What is required to make this happen is a shared `/dev/shm` – shared with full Mach IPC semantics, not just NFS – so `netmsg` is required.

Replacing the root filesystem on Hurd requires a reboot, so unless you've recompiled the system translators with the new `libpager`, it's better to mount a new filesystem on `/dev/shm`, using the new `libpager`. Ideally, this would done with Hurd's ramdisk translator, `tmpfs`, but `tmpfs` doesn't currently implement the `libpager` interface, so it's unusable for `/dev/shm`. For demonstration purposes, we can create an ext2 filesystem and loopback mount it on `/dev/shm`.

This isn't ideal, because our shared memory is saved to disk, but it serves to prove the functionality desired. We can now start a `netmsg` server and run a program that allocates a shared memory segment, writes a string of data into it, and waits for an acknowledgment that it has been read:

On another machine, we use `netmsg` to connect `/dev/shm` to the first machine. The two machines now share their shared memory space, and a client program is able to map the shared memory segment, read the string, and acknowledge it.

This implementation services all shared memory requests by directing them to a single node. Obviously, this is not ideal. The `unionfs`[4] translator allows directories from different nodes to be overlaid on top of each other, and could possibly

[4]`unionfs` is not part of the stock Hurd distribution. Its source is available from `https://www.gnu.org/software/hurd/hurd/translator/unionfs.html`

```
root@qemu-hurd-1:~# dd if=/dev/zero of=shm bs=1M count=10
10+0 records in
10+0 records out
10485760 bytes (10 MB, 10 MiB) copied, 0.1 s, 105 MB/s
root@qemu-hurd-1:~# mke2fs shm
mke2fs 1.44.6 (5-Mar-2019)
Creating filesystem with 2560 4k blocks and 2560 inodes

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done

root@qemu-hurd-1:~# settrans -ak /dev/shm ~/hurd-0.9.git20190331/build-deb/ext2fs/ext2fs.static shm
root@qemu-hurd-1:~# ls /dev/shm
lost+found
```

```
root@qemu-hurd-1:~# ~/github/netmsg/netmsg -s /dev/shm &
[1] 2533
root@qemu-hurd-1:~# ~/github/shm/Dave.Marshall/shm_server
```

be used to overlay the various `/dev/shm`'s on each other. Then, a new shared memory segment would be created on the original process's node, but would immediately become visible to all other nodes. A more sophisticated translator could be developed that would dynamically balance shared memory management among the nodes.

`mmap`, as currently implemented in Hurd's C library, only allocates memory on the local node. It does not seem difficult to create a new translator that would service memory allocation requests on remote nodes, in a manner similar to Linux's NUMA (Non-Uniform Memory Access) support. Linux's NUMA implementation uses a zonelist for memory allocation. Memory allocation requests are attempted locally, and if this fails, then we fall back into the zonelist, which gives us a list of other nodes to try for memory allocation.

8 Distributed Tasks

In the Hurd architecture, processes are localized to a node, but our implementation of distributed shared memory allows cross-node forks with full POSIX semantics. A straightforward modification to Hurd's C library would allow fork calls to spawn processes on other nodes, but this has not yet been implemented.

Sharing threads between nodes is much more difficult, and seems basically impossible to implement using Mach's current design. The main problem is Mach port handling. Any thread in a task can listen for messages on a Mach port, and messages are only delivered once, so balancing the need to copy messages to multiple nodes (since any thread could be the only one listening to a given port) while guaranteeing unique delivery is quite problematic.

This suggests a natural division: processes are local to each node. A multi-threaded program can fork multiple times, and each forked process can spawn multiple threads. The processes are distributed between nodes, while each process is local to a node and supports multiple threads. Since we have inter-node, inter-process shared memory, almost all of the standard thread synchronization techniques (semaphores, mutexes, condition variables) will work unmodified. Just as a typical multi-threaded process should try to avoid thread contention on cache lines (typically 64 bytes), a multi-task process should try to avoid task contention on memory pages (4 KB on the x86 architecture).

Since we have a working implementation of distributed shared virtual memory, implementing cross-node forks doesn't seem too difficult. There is a problem, however. Mach allows the allocation of memory pages unmanaged by any memory manager. Unmanaged pages are typically used for a process's routine data pages, but a cross-node fork would require them to become managed in order to police multi-node access.

This suggests the need to attach a memory manager to the unmanaged pages to allow them to be distributed across nodes, and this would require an enhancement to the Mach kernel itself. I think that a new `memory_object_attach` RPC would be used for this purpose.

Once implemented, cross-node forks could be performed with full POSIX semantics. It's interesting that only at this point do we seem to need a Mach kernel function that wasn't anticipated by Mach's designers in the late 1980s.

```
root@qemu-hurd-2:~# settrans -ak /dev/shm ./netmsg/netmsg/netmsg 192.168.55.101
root@qemu-hurd-2:~# ls /dev/shm
lost+found  sysvshm-162e
root@qemu-hurd-2:~# ~/shm_client
abcdefghijklmnopqrstuvwxyz
root@qemu-hurd-2:~#
```

The same mechanisms used for cross-node forks could also be used to permit process migration. After using cross-node memory maps to create a duplicate process on another node (without any threads running), the threads of the original process would be stopped, the Mach port rights transferred using `netmsg`, duplicate threads would be created and started.

Some investigation would be required to see if Mach can cleanly transfer thread state like this. For example, I'm not sure how Mach would save state for a thread currently blocked in a `mach_msg()` system call. I think that it saves the user-space state with the instruction pointer set to the system call instruction. Restarting from this state would rerun the system call, which would create duplicate messages if, for example, the system call was the `mach_msg_send()` variant.

9 Current Status and Future Work

Both `netmsg` and the new `libpager` have been successfully tested in a virtual machine running Hurd, but their operation is unreliable and additional work is required before Hurd can be used as a viable SSI cluster operating system.

- 64-bit addressing and SMP

  This is the most important work needed on Hurd. We've got all of the major pieces of a working cluster operating system, but nobody cares since it can only use a single processor on each node and individual processes are limited to a 4 GB address space.

- `netmsg` improvements

  - Use a reliable datagram delivery protocol instead of TCP.
  - Add encryption and authentication.

    This might not be necessary if the protocol can be guaranteed to operate only over a local network, and might have adverse performance impacts, but should be available if desired.
  - Improved port allocation protocol

    We need to avoid race conditions when deallocating ports, also also detect loops when more than two nodes are interoperating.
  - Direct access to network hardware.

    Memory-mapped PCI hardware can be accessed with a library, avoiding a context switch to the networking server. This would require a network card to be dedicated to `netmsg`, but with networking hardware such as Cisco's VIC 1280, that's not a problem, since a VIC 1280 can present itself as anywhere from one to sixteen virtual NICs, allowing an additional virtual NIC to be allocated solely for use by `netmsg`.

- Other translator improvements

  - `libpager` integration with `tmpfs`

    `tmpfs`, the Hurd ramdisk, currently throws `SIGABRT` if any `libpager` functions are called. Since `tmpfs` is the most obvious choice to mount on `/dev/shm`, fixing this is a high-priority, low-difficulty task.
  - distributed filesystem

    Once we have the ability to run processes using distributed shared memory and distributed scheduling, we should enhance our file system translators to support replicated copies of files. This suggests the need for a...
  - distributed `libpager`

    We could use a shim process on remote nodes, to allow requests for an existing page go to a node that holds a copy of that page, instead of to the node with the disk. Two ways I can imagine this. Either ext2fs forks off multiple processes on different nodes, or a mechanism is developed to detect when a port is remote, figure out which node it (currently) resides on, and then `libpager` can fork a process and push it to the other node, invisible to ext2fs proper.

- Authentication

  As I outlined in this proposal, Hurd's current authentication scheme is inadequate for multi-node operation, but a straightforward enhancement is possible.

  Currently, if a process wishes to authenticate with a server (say, to obtain access to file), it creates a port (used solely for authentication) and passes two send rights to that port, one to the file server and one to a trusted authentication server. The file server then passes its send right to the authentication server, which matches it against the send right passed by the client, verifies that the client in question has the desired permissions, and replies to the file server. This allows the client to validate its permissions without transferring those permissions to a (potentially malicious) file server.

  The current authentication scheme relies on Mach's guarantee that rights to the same queue will always appear on a single port number, no matter how they were received. This allows the authentication server to match the send right that it receives directly from the client with the send right that was created by the client, but passed via the file server. The current `netmsg` design doesn't honor this assumption when more than two nodes are in use. If the client, file server, and authentication server are all on different nodes, then the authentication server will receive send rights on two different ports and won't be able to match them. An improved `netmsg` design is thus essential.

- C library changes

  Mainly allowing cross-node forks.

- Required Mach kernel enhancement

  - allow memory manager to attach to unmanaged pages

    This is necessary to implement cross-node `fork`'s with full POSIX semantics.

- Optional Mach kernel enhancements

  - `memory_object_data_return` could specify what permissions the kernel still holds on the page

    Then we could easily downgrade from WRITE to READ access without waiting for a lock completed message, or (like the current code), never downgrading from WRITE to READ.
  - `memory_object_lock_completed` message could specify what kind of lock was completed

    This would avoid the need to match reply ports to lock messages unless (like the current code), we just wait for all outstanding locks to complete before continuing with any of them.
  - ability to request a copy of a unmodified, un-precious page from the kernel

    This would allow `libpager` to serve out the page to other nodes without reading it again from the disk.
  - ability to notify memory manager when a non-precious page has been evicted

    Over the network, flagging a page "precious" is awfully heavy weight just to get notification that a page has been evicted, since a "precious" page will be sent back to the memory manager. Knowing that a page has been evicted may eliminate the need for a future lock request.
  - Use a kernel name port common across all memory objects

    This would allow sharing pagemaps between memory objects with the same usage pattern (i.e, kernels A and B both map files C and D)
  - Add the ability for a kernel to respond to a lock request by sending the page directly to another kernel, rather than back to the memory manager.

    This would cut our network utilization roughly in half for pages dominated by contention between nodes.

References

[1] *Mach 3 Kernel Interfaces.*

[2] *Mach 3 Kernel Principles.*

A `libpager` Pseudocode

```
service_waitlist: (pass in a pagetable pointer, a data pointer, a read/write flag, a deallocate flage)
  XXX: data length must be page_size, as this logic is for a single page
  call with pager locked
  use m_o_data_supply to supply first set of WAITLIST clients
    all but last specify deallocate=false; last one specifies deallocation flag as passed in
  remove those clients from WAITLIST and move them to ACCESSLIST
  what if WAITLIST still has clients on it after sending messages?
     [ ] use a timeout before sending more lock requests?
     [X] send lock requests right away to everything on ACCESSLIST


send_error_to_WAITLIST (pass in an error and offset):
  XXX: data length must be page_size
    ( XXX this causes a data_error to be sent for each page, when we could consolidate them, )
    ( but hopefully data_error is a corner case.  For ENOSPC or EDQUOT, though, not really. )
  send m_o_data_error's to everything on WAITLIST and clear WAITLIST
    anything on WAITLIST that's also on ACCESSLIST should get a lock instead of data_error
      and be put on NEXTERROR list
  (such a client sent an unlock request and is waiting for a lock, not data_error)

  [ ] flush everything else on ACCESSLIST
  [X] do nothing to ACCESSLIST
  [ ] do something more clever with stuff on ACCESSLIST
     (we got a read error, but other clients have copies of the data)


finalize_unlock: (pass in a page number and an error code)
  if page not flagged ERROR:
     send m_o_lock_request to first client on WAITLIST (no reply), remove this client from WAITLIST,
     and upgrade its ACCESSLIST entry to reflect WRITE access
     if there are additional clients on WAITLIST:
        [ ] use a timeout before sending a lock request to this client
        [X] send a lock request right away to this client
            (XXX note that we just sent a lock request to answer the unlock; really want a timeout here)
  else (page flagged ERROR):
     send m_o_lock_request to first client on WAITLIST (internal flush variant)
     remove this client from WAITLIST
     add this client and error to NEXTERROR list
     (wait for reply and don't remove from ACCESSLIST until we get it)
     (other clients on WAITLIST will be processed when the lock is answered)


m_o_data_request: (kernel requesting a single page)

  if this client is on NEXTERROR list and WRITE ACCESS WAS REQUESTED, send a m_o_data_error message,
        move NEXTERROR to ERROR, remove the NEXTERROR list entry, and return
  if the page is PAGINGOUT:
    if WAITLIST is empty and ACCESSLIST is not, send lock request (internal flush variant) to ACCESSLIST client
       ASSERT: there should only be one WRITE client on ACCESSLIST - the client that returned the data we're paging out
    add requesting client to WAITLIST and return
  if this client is already on ACCESSLIST, it flushed and is trying to re-aquire access,
     so check to see if WAITLIST is empty
     if so, remove client from ACCESSLIST and proceed (kernel flushed without being asked)
     if not, we're flushing, so add client to WAITLIST, run internal_lock_completed on this page, and return
     XXX: check to see if client is already on WAITLIST?
  if WAITLIST is not empty, add requesting client to WAITLIST and return
     XXX: check to see if client is already on WAITLIST?
  if another client has WRITE access (WAITLIST is empty), add requesting client to WAITLIST,
     send lock request (internal flush variant) and return
  if WRITE access is requested and ACCESSLIST is not empty (other clients have access),
     add requesting client to WAITLIST (it's empty),
     send lock requests (internal flush variant) and return
     (we could give out the page with READ access and wait for a unlock request, but I think not)
  else
     (ACCESSLIST is empty or READ access is requested and only READ clients are on ACCESSLIST)
     if page is flagged INVALID, send m_o_data_error to this client and return
     add this client to WAITLIST (it's empty)
     unlock pager
     read the page with pager_read_page()
     relock pager
     if pager_read_page() returned an error, mark page with ERROR and call send_error_to_WAITLIST
     if no error, service_waitlist(deallocate=true) (read/write status supplied by filesystem), clear ERROR


m_o_lock_completed:
  m_o_lock_completed could have been sent in response to an internal lock request, or a fs flush, sync, return
  find client/offset/length in list of lock requests and decrement locks_pending
  if locks_pending is zero:
    if internal_lock_outstanding, and WAITLIST is not empty, then call internal_lock_completed
    remove client/offset/length from list
  if no more clients for this offset/length, wake up any waiting thread


internal_lock_completed: (internal flush variant)
  use alloca() to allocate an array of flags, one set (in a uint8) for each page in current message:
    PAGEIN, UNLOCK, ERROR
  for all pages covered by message:
    if this client had WRITE access, do nothing (we're waiting for an m_o_data_return)
    if this client isn't on ACCESSLIST, do nothing (m_o_data_return already processed the data)
    if WAITLIST is empty, do nothing (m_o_data_request already called internal_lock_completed)
    else:
      remove client from ACCESSLIST
      if ACCESSLIST is empty but WAITLIST is not:
        if INVALID is not set: set PAGEIN flag
        else (INVALID set): call send_error_to_WAITLIST
          XXX: assumes that ERROR is correctly set whenever INVALID is set

      if ACCESSLIST has a single client with READ access and it's also the first client on WAITLIST requesting WRITE ac
        set UNLOCK flag
      otherwise, if ACCESSLIST has multiple clients, or a single client that's not first on WAITLIST,
        then we're still waiting for them to answer their locks, so do nothing
  unlock pager
  use alloca() to allocate an array of pointers and write lock flags, one of each for each page in current message
  for each page flagged PAGEIN:
    read the page with pager_read_page(), saving resulting pointer and write lock flag
    if pager_read_page() returned an error, set flag ERROR
    (a client with no access requested WRITE access and had to wait for other clients to flush)
  for each page flagged UNLOCK:
    call pager_unlock_page(), and set ERROR based on return value
    (a client with READ access requested WRITE access and had to wait for other clients with READ access to flush)
  relock pager
  for each page flagged PAGEIN:
    if page not flagged ERROR:
       service_waitlist(deallocate=true) (read/write status supplied by filesystem), clear ERROR
       XXX: this causes multi-page operations to broken up into single page ops
    else (page flagged ERROR): call send_error_to_WAITLIST
  for each page flagged UNLOCK: call finalize_unlock


m_o_data_unlock: (kernel requesting write access when it's already got read access)
  lock pager
  alloca() an array of UNLOCK and ERROR flags, one pair for each page in message
  for all pages:
    ASSERT: this client already on ACCESSLIST with read access
    ASSERT: PAGINGOUT flag is not set (only should be set if a client had WRITE access)
    if WAITLIST contains at least one client waiting for WRITE access, do nothing and return
      (we're either already trying to flush everything on ACCESSLIST, or will, including this client)
      (if it saw the flush request before sending the unlock, it would never have sent the unlock)
      (so it hasn't processed the flush yet, and will interpret it as a response to the unlock)
    else if WAITLIST contains only clients waiting for READ access, add client to WAITLIST for WRITE and return
      (ACCESSLIST contains only READ clients and WAITLIST contains only READ clients)
      (so we're waiting for a page in to finish, as it will trigger a flush that will answer this unlock)
    otherwise, if there's other clients on ACCESSLIST, add client to WAITLIST for WRITE (it's empty)
      and send lock requests to other clients (internal flush variant) and return
    otherwise, we're the only client on ACCESSLIST and WAITLIST is empty:
      add client to WAITLIST for WRITE
      set UNLOCK flag
  unlock pager
  for all pages with UNLOCK flag set:
    call pager_unlock_page() and set ERROR based on return value
  relock pager
  for all pages with UNLOCK flag set: call finalize_unlock


lock_object: (several types: flush, sync, return, response to an unlock request)
  NEW MODE NEEDED: asynchronous, but reply requested
  wrapper around m_o_data_lock message
  if sync requested, add to lock_requests list and increment count
     after m_o_lock_request message sent, wait for locks_pending and pending_writes to drop to zero,
     then decrement threads_waiting and remove lock_requests if zero

  lock_object() called by pager_ calls to flush, sync, and return data from kernel,
    as well as m_o_data_unlock

  if a m_o_data_return comes in while waiting for a synchronous lock to complete,
    we wait for the write to finish before returning from the lock
    the data return could be triggered by the lock request (i.e, a sync or a return)
  if a lock request comes in while a write is happening, we don't do anything to synchronize them


m_o_data_return: (kernel returning pages)
  (this can not be used in liu of a lock completed message for revoking WRITE access,
     because it could be spontaneously generated by the kernel, and
     doesn't indicate if the kernel still maintains WRITE access)
  npages = length / __vm_page_size;
  pm_entries points to 'npages' entries in pagemap
  lock pager
  ASSERT: this client should be all the page's ACCESSLISTs

  if kernel_copy is false:

    remove this client from all of these page's ACCESSLISTs
    [ ASSERT: these ACCESSLISTs should now be empty ]
    [ ASSERT: this client is not on any of the WAITLISTs (it had WRITE access, so there's nothing for it to wait for) ]
    [ not necessarily - if the pages are precious, they might be coming back with only READ access ]
    alloca() an array of UNLOCK, NOTIFY, and ERROR flags, one set for each page in message
    for all pages:
      if ACCESSLIST is empty and WAITLIST is not empty:
        service_waitlist(deallocate=false)
        (a client with no access requested WRITE access and the last client we were waiting for flushed)
      if ACCESSLIST has a single client with READ access and it's also the first client on WAITLIST requesting WRITE ac
        set UNLOCK flag
        (a client with READ access requested WRITE access and had to wait for other clients with READ access to flush)
      if both ACCESSLIST and WAITLIST are empty and pages are not DIRTY:
        set NOTIFY flag
        ( DIRTY pages will get their notifications done after their pager_write_page() calls )
    if any pages required UNLOCK or NOTIFY:
      ( this can't happen if pages are dirty )
      ASSERT: ! pages_dirty
      XXX: should this code be moved down further?
      unlock pager
      for all pages with UNLOCK set:
        call pager_unlock_page() and set ERROR based on return value
      for all pages with NOTIFY set:
        call pager_notify_evict()
        clear INVALID, ERROR, and NEXTERROR for this page
      relock pager
      for all pages with UNLOCK set: call finalize_unlock

  if pages are dirty:

    ASSERT: if kernel_copy = true, then this client is the only thing on ACCESSLIST
    ASSERT: if kernel_copy = false, then ACCESSLIST is empty
      (this client was the only thing on ACCESSLIST, because it had WRITE access to create a dirty page)

    [ ] for all pages with empty WAITLISTs and ACCESSLISTs:
        (XXX pager_sync needs to trigger a write even for pages with active clients)
    [X] for all pages in message:
         set PAGINGOUT in pagemap

    [ ] if PAGINGOUT was already set for any page, add this message, including kernel_copy flag
        and the pointer to the last lock incremented, to WRITEWAIT list and return
        ( allow simultaneous pager_write_page() calls for different pages )
    [X] if WRITEWAIT list is not empty, add this message (and kernel_copy and pointer) to WRITEWAIT list and return
        if WRITEWAIT list is empty, add this message (and kernel_copy and pointer) to WRITEWAIT list
           and process everything on WRITEWAIT list
        ( allow no simultaneous pager_write_page() calls )
  else (pages are not dirty):
    munmap() data


service_first_WRITEWAIT_entry:
  use alloca() to allocated an array of flags, one set (in a uint8) for each page in current message:
    NOTIFY, ERROR, and PAGEOUT
    ( perhaps put this is a subroutine so that this array is deallocated after every message is processed )
  for all pages in current message:
    search WRITEWAIT list for a matching page
    if no match, set PAGEOUT flag
    ( this avoids writing a page if we've got more recent data waiting to write )
    ( this could be triggered by a rapid succession of pager_sync() calls on a busy page and a slow disk )
  unlock the pager
  call pager_write_page() on all pages flagged PAGEOUT
    set ERROR on any error return
  relock the pager
  for each page we just wrote:
    search WRITEWAIT list for matching pages
    ( can't use PAGEOUT flag because WRITEWAIT list might have changed while pager was unlocked )
    set INVALID equal to ERROR
    if something found on WRITEWAIT:
      ASSERT: kernel_copy is true (else how could there be a later write?)
      do nothing
    else if none found and WAITLIST is not empty:
      clear PAGINGOUT
      (this client had WRITE access because the page was dirty; so everything on WAITLIST is a data request)
      if kernel_copy is false: service_waitlist (deallocate = false)
      (if kernel_copy is true then we're still waiting for a flush to complete)
    else (none found and WAITLIST is empty):
      clear PAGINGOUT
      if kernel_copy is false, set NOTIFY flag and clear ERROR and NEXTERROR

  munmap() current message

  unlock the pager
  if client requested notify_on_evict, call pager_notify_evict() for any page flaged NOTIFY
    also, clear INVALID, ERROR, and NEXTERROR for these page
    ( the kernel didn't keep a copy and it didn't get shipped back out in an m_o_data_supply message )
  relock the pager (this can be cleverly wrapped into previous unlock/lock cycle)

  signal wakeup on anything waiting on this WRITEWAIT entry
  remove this WRITEWAIT entry from WRITEWAIT list


pager_sync:
  only needs to do anything if a client has WRITE access
  might need to do multiple locks if different clients have WRITE access to different pages


pager_return:
  scan pagelist for specified range and form the union of all clients on the ACCESSLISTs
  send multiple locks for multiple clients
  set (or increment) locks_pending by the number of lock requests sent


pager_flush:
  scan pagelist for specified range and form the union of all clients on the ACCESSLISTs
  send multiple locks for multiple clients
  set (or increment) locks_pending by the number of lock requests sent


m_o_terminate:
  mach_port_destroy() control and name ports passed in
    (we had send rights and should have just received receive rights in the m_o_terminate message)


pager_shutdown:
  set terminating flag
  send pager_return (flush, return all, sync) to all active clients
```

```
in the meantime:
  ignore all m_o_data_request and m_o_unlock requests
  m_o_lock_completed messages: don't do "internal" processing, but do notify threads
  process m_o_data_return messages, but don't generate any lock requests or data supply messages as a result
  [ ] handle new m_o_init_object messages by adding them to client list, but not sending m_o_ready
      that way, they'll get m_o_destroy in response
  [X] ignore m_o_init_object messages
[ "De-allocating the abstract memory object port also has this effect" - Mach Kernel Principles p. 43 ]
[ so... we don't have to do this section ]
[ send m_o_destroy (error = ENODEV) to all active clients
[   now handle new m_o_init_object messages by sending m_o_destroy in response and adding to client list
[ wait for m_o_terminate replies from all clients
call ports_destroy_right, which will blow away our receive right, and ultimately call the destructor,
  which will then call drop_client on any remaining clients
```