A Critique of the GNU Hurd Multi-server Operating System

Neal H. Walfield
neal@gnu.org

Marcus Brinkmann
marcus@gnu.org

Abstract

The GNU Hurd's design was motivated by a desire to rectify a number of observed shortcomings in Unix. Foremost among these is that many policies that limit users exist simply as remnants of the design of the system's mechanisms and their implementation. To increase extensibility and integration, the Hurd adopts an object-based architecture and defines interfaces, which, in particular those for the composition of and access to name spaces, are virtualizable.

This paper is first a presentation of the Hurd's design goals and a characterization of its architecture primarily as it represents a departure from Unix's. We then critique the architecture and assess it in terms of the user environment of today focusing on security. Then follows an evaluation of Mach, the microkernel on which the Hurd is built, emphasizing the design constraints which Mach imposes as well as a number of deficiencies its design presents for multi-server like systems. Finally, we reflect on the properties such a system appears to require.

1 Introduction

The goal of the GNU project is to create an operating system consisting entirely of free software. By the end of the 1980s, the most important missing component was the kernel. As Unix systems were the primary operating system used by both GNU software users and developers and as the components written to that date were designed for such systems, a high degree of API compatibility was deemed necessary. With the hope of speeding development, the decision was made to base the system on a free version of the Mach kernel from CMU. The designers decided to exploit the microkernel foundation to build a more integrated and extensible system, improving its usability.

In [5], Bushnell outlines the Hurd's architecture and states that its goals, in addition to legacy compatibility, are to permit:

- Efficient sharing of scarce resources
- Greater extensibility and integration
- Mutually suspicious collaboration
- Sharing without prior arrangement

The intent was to improve the usability of the system through the creation of a well integrated, component based system in which system services can be easily replaced and extended at a fine granularity yet which is sufficiently compatible with existing APIs to run most important software packages with little modification, in particular, those from the GNU project. These concerns motivated a multi-server structured system with a distributed, user extensible, naming framework.

2 The GNU Hurd's Architecture

The Hurd is a set of objects. An object is similar to a closure: it implements an interface and consists of a program and state. These objects extend the objects exposed by the underlying microkernel, Mach [39], to include standard system functionality and to dictate system policy. System services are made available exclusively through objects.

Hurd objects are realized in user-space processes. Such processes are referred to as servers. To improve fault isolation and reduce that on which a program depends for its correct operation, its *reliance set*, [23, Ch. 5], a server implements a minimal number of related objects. Typically, a server decomposes some larger object. A file system server, for instance, exposes a part of backing store as a hierarchy of files and directories. This is in contrast to a monolithic system where many components execute in the kernel's protection domain and component boundaries are only a formality.

Objects are referenced by *capabilities* [8]. Capabilities both designate an object and authorize access to it. Mach provides protected capabilities: unforgeable, task local, opaque references held in a capability slot in a process's address space. They can only be communicated using the message passing facility.

A capability does not directly reference an object. On Mach, it references a kernel message queue, a *port*. A client holds a capability that permits queuing of messages, a *send right*, and a server, a capability that permits dequeuing of messages, a *receive right*. A server internally associates the kernel object with the user object.

A process may sense and manipulate an object only by invoking a capability which references it. Invoking a capability causes a message to be made available to the process implementing the referenced object. The message may carry data and capabilities determined by the invoker. When the invoker wants a reply, it includes a *reply capability*, a send-once right to a kernel message queue to which it has a receive right. It then waits for a message. This is the *remote procedure call* (RPC) pattern.

Because using object, whether it is implemented by Mach or by a user-space process, is only possible using the message passing interface, any process may transparently implement, proxy or extend an object insofar as it can interpose itself between the object and the user. This is a basic requirement for virtualization [28] and reference monitors [1].

2.1 System Structure

The Hurd is defined by approximately a dozen canonical interfaces. The `fs` interface is used in the examination and manipulation of directory and file meta-data. This includes traversing object relationships using symbolic names. The `io` interface is used to read from data sources and to write to data sinks. File handle objects usually implement both of these interfaces. The `fsys` interface is used for whole file system related operations, e.g., those set on Unix using the `-o` option to `mount`, as well as to obtain an unauthenticated file handle to the root of the file system.

Additional interfaces include the `auth` interface for managing identities and for the support of *identity based access control* (IBAC), the `password` interface for obtaining identity objects against passwords, the `exec` interface for help in instantiating programs and the `process` interface for process management including process identifiers (PIDs), session and process group management and non-preemptive signal delivery.

A Hurd system consists of at least the Mach kernel, an `auth` server, a `proc` server, an `exec` server, a `password` server and a file system server. These servers provide a similar level of abstraction as the system call interface of a traditional monolithic kernel.

The C library directly interacts with these servers in its implementation of POSIX and other higher level interfaces. Most programs use these interfaces exclusively. The implementation also contains hooks and extensions for more convenient use of some Hurd-specific features.

A number of utility programs extend the traditional collection of Unix utilities giving the user direct access to Hurd functionality. The most important of these is the `settrans` program for starting new servers and linking them to a name space.

2.2 Naming and Name Spaces

Although capabilities allow processes to reference objects, a convention is required to permit users to designate the objects on which a program should operate. The Hurd's solution appears similar to Unix's virtual file system (VFS), however, differentiates itself in that its realization is distributed, not centralized. In particular, any process, without special privilege, can implement the conventions of the Hurd's VFS and create and publish a commonly understood naming hierarchy.

In this framework, object relations are named symbolically. The traversal of object relationships, *name resolution*, is realized using the `dir_lookup` interface. There is no implicit root: resolution is always done relative to an explicitly referenced object. Applications, however, resolve most names either relative to the capability stored in its *root directory capability slot* or relative to the capability stored in its *current working directory slot*, which are normally filled by the parent process with a copy of its own respective references at process creation.

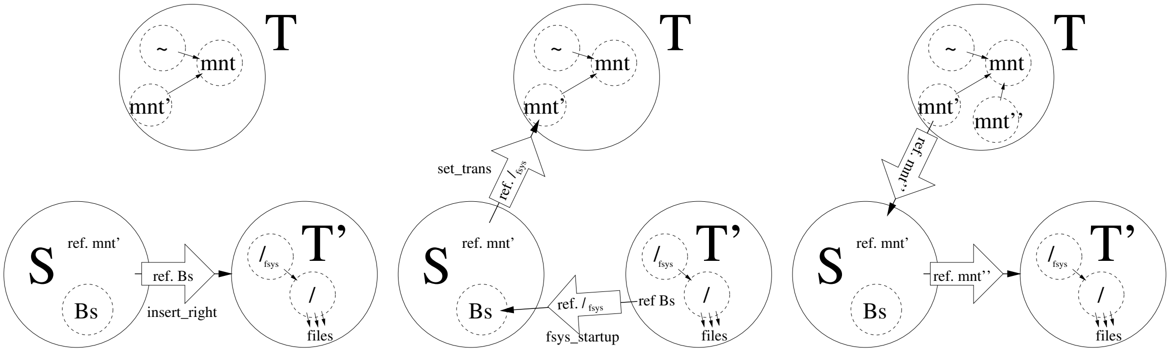
Often, the `dir_lookup` method does not actually return a capability referencing the resolved object: it creates a new object, a *handle*, which references some session state and the resolved object. These sessions are used primarily to fulfill some POSIX requirement. A file handle, for instance, includes a cursor, which records the session's current position in the file.

2.2.1 Extending a Name Space

When Alice wishes to access files on an FTP server from her Unix workstation, she likely uses an FTP program to copy the relevant files locally. Later, after having made some modifications, she again runs the FTP program to copy the modifications back to the server.

These steps are necessary as the programs Alice uses to manipulate the data cannot manipulate the objects on the FTP server: neither do the programs understand the object naming and access conventions of the FTP server nor is Alice able to instantiate her own file system that can make the objects available using the API they understand. In the latter case, the problem is that this typically requires uploading code to the kernel or using a fragile kernel service (e.g., a file system driver, most implementations of which assume correct input).

To work around this, the GNOME and KDE projects



(a) `settrans` has opened `~/mnt` and created the new translator instance, T'. `settrans` now inserts the bootstrap capability into the bootstrap capability slot of T'. `settrans` starts T' running.

(b) Upon finding a capability in its bootstrap capability slot, T' invokes `fsys_startup` passing a reference to an unauthenticated root object. S invokes `file_set_translator` propagating the capability to T.

(c) T installs the capability referencing T' unauthenticated root object and installs it in `mnt`, returns an unauthenticated reference to the new handle mnt" which S propagates to T".

Figure 1: The `settrans` program.

have built their own VFS implementations which are user extensible but only accessible using different calling conventions. As such, applications which do not use this API appear less integrated. The Linux kernel developers have acknowledged this problem and have recently extended their kernel to support unprivileged, user-space file systems.

On the Hurd, Alice could have used the ftpfs program to make the objects on the FTP server available in a portion of the VFS she controls. Alice does not require any special privilege to do this: ftpfs is a normal program which implements a protocol; she just requires the appropriate access to the node to which she wants to attach it. Such programs are referred to as *translators*, as they translate between a pair of naming and access conventions.

Hurd translators are linked to other translators by inserting a capability referencing the `fsys` object of the translator at the desired *mount point*. This is usually done using the `settrans` program and is conceptually similar to mounting a file system on Unix. In the scenario presented above, Alice could have run a command similar to the following:

```
$ settrans -a ~/mnt /hurd/ftpfs \
    username:password@site.org/~
```

`settrans` starts an instance of the ftpfs program and attaches it to the node `~/mnt`. The remaining arguments are passed to the translator which may freely interpret them, in this case, a URL.

The `settrans` program works by first obtaining a handle to the mount point. It then instantiates the program and, before setting it running, creates a port and inserts a send right to it in the process's *bootstrap capability slot* (Figure 1(a)). When the translator starts, it invokes the `fsys_startup` method on the bootstrap capability passing a capability referencing its `fsys` object as an argument. (This object is required by the parent translator in its `dir_lookup` implementation to redirect a caller to the translator.) `settrans` then invokes `file_set_translator` on the capability naming the mount point handle, passing the root capability as an argument (Figure 1(b)). The parent translator returns an unauthenticated handle to the mount point which `settrans`, in turn, forwards to the new translator (Figure 1(c)).

2.2.2 Name Resolution

A translator may use its `fsys` object simply as a *rendezvous* point. The `auth` and `password` servers do this: neither exposes objects in a way appropriate for a directory structure. Most servers, however, have a hierarchy of objects which they make accessible via the standard interfaces.

Object relationships are named symbolically and traversed using the `dir_lookup` method. The `dir_lookup` method accepts a *path*, a series of *path components*, symbolic names, separated by one or more / characters, and resolves the first path component. If the resolved object is also implemented by the server and path components remain, the server, without returning to the caller, may repeat the process using the resolved object as the new starting point and the remainder of the path. This is an optimization to reduce the number of RPCs when resolving paths with multiple components naming objects implemented by the same server.

This process continues until either an error occurs, the path is completely resolved or a named object is implemented by another server. These three scenarios correspond to the three types of replies: if an error occurs resolving a path component, e.g., a component does not name an object or an access error occurs, the server returns an error to the caller; if the path resolves to an object which the server implements, a capability which names a new handle to the resolved object authenticated using the invoked handle's identity is returned; and if, in resolving the supplied path, an object is encountered which names an object on another server, the server obtains a capability to a new unauthenticated root object handle on the server and returns this as well as the path which remains to be resolved to the client in the form of a so-called *retry message*. In the last case, after the client identifies itself to the new server, it invokes `dir_lookup` on authenticated handle and passes the rewritten path.

Some servers also directly support symbolic links (although, they can also be realized as normal translators). This does not require any special handling beyond recognizing them and rewriting the path appropriately. Resolution continues as normal.

To remain compatible with POSIX, a `dir_lookup` implementation is required to resolve the *dot-dot directory entry*, i.e., the directory entry which names the directory which contains the directory. (As a special case, POSIX indicates that dot-dot resolves to itself at the root.) If a process calls `dir_lookup` on a capability naming `/home/alice/mnt`, as in Figure 1, passing dot-dot as the path to resolve, the ftpfs instance would return a retry message which included a capability naming the object `/home/alice/mnt` on the parent translator and have rewritten the path to dot-dot.

The Unix `chroot` mechanism requires that a directory appear to a group of processes as a root. That is, the meaning of dot-dot must be overridden. The Hurd's mechanism, `file_reparent`, is slightly more general and requires less privileged. `file_reparent` creates a new handle for which dot-dot resolves to a provided capability. Handles derived from this one naming the same node preserve this property. (A void capability indicates that the directory should appear as a root.)

`file_reparent` also allows the realization of *firm links*, links which bind a portion of the VFS to another location, a sort of cross file system hard link. Linux has a similar mechanism, bind mounts.

2.2.3 Persistent Translators

On the Hurd, neither processes nor capabilities are persistent: files are the only persistent resource. To restore the operating environment, sufficiently privileged programs, usually at installation time, register a command to be run at system restart. When the system is shutdown, processes are informed of their imminent termination and given the opportunity to save any state. Then, on system restart, this is used to restore the system to the approximate state it was in when it was last shutdown.

This approach was inherited from Unix and, for a relatively static, centrally controlled system, is sufficient for configuration recovery. On the Hurd, users have much more control over their computing environment through the use of translators. To allow translators to be restarted transparently and consistent with the distributed architecture, a *passive translator setting* can be saved in the node on which the translator is set and, if no translator is running when the node is accessed, the translator will run the program specified in the passive translator setting. The program is started with the UID and GID of the node, which is often possible as it is normally the case that the parent either has the same identity as the translator or an identity which dominates that of the translator (e.g., root). When this is not the case, the translator is safely started without an identity.

The passive translator setting is saved using the `file_set_translator` method. Since the translator is started with the UID and GID of the node, it can typically only be set by the owner of the node. How and if the passive translator setting is saved is implementation defined. The ext2 file system implementation, for instance, allocates a file system block for the passive translator setting and saves the block address in the relevant inode.

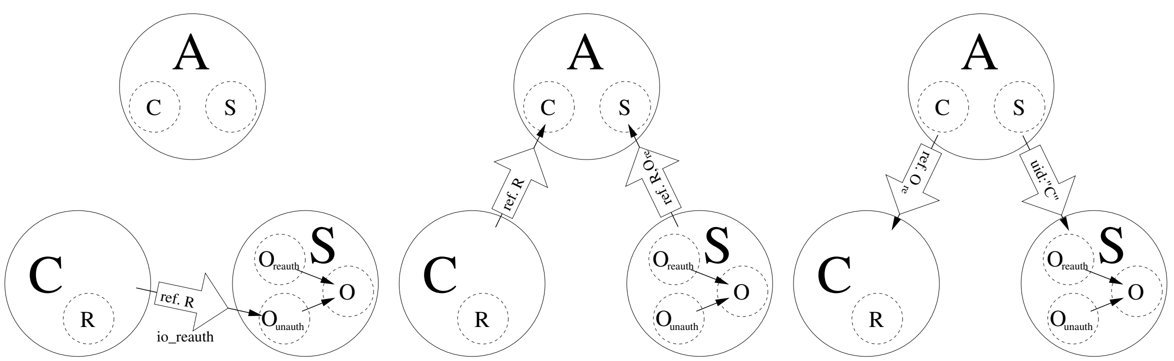
2.3 Protection and Security

Hurd servers control access to objects based on the identity of the subject. The policy is similar to Unix but the mechanism is quit different. On the Hurd, identities are first class objects (meaning that a single process may have more than one or none at all) and are managed by the `auth` server. The `auth` server also supports programs in the realization of IBAC by providing an authentication mechanism which allow programs to safely expose identities to others in a verifiable manner.

2.3.1 Identity Management

As identities are first class objects, a process may have access to any number of UIDs and GIDs or none at all. Moreover, because they are simply objects named by capabilities, a process is able, without any special privilege, to remove an identity by destroying it, so-called *discretionary authority reduction*. This technique allows applications to run with less excess authority thereby reducing the amount of damage a bug or an attacker can cause.

Applications which require access to a fixed number of resources known at start up and after which do not further require the authority an identity grants, can take advantage of this technique. For example, a network server



(a) The client invokes the `io_reauthenticate` method on an unauthenticated object handle passing a reference to a *rendezvous* object.

(b) Client and server invoke a capability referencing an id object, passing the *rendezvous* capability. The server also includes a capability referencing a new object handle.

(c) The auth server pairs the *rendezvous* references and returns the ids contained in the client's id object to the server which can now realize its access control policy; and a reference to the new object handle to the client.

Figure 2: The authentication mechanism.

which needs to bind to a TCP port below 1024, an operation normally reserved to the superuser, but which does not otherwise require the authority the superuser identity conveys, can run with no UIDs or GIDs after binding to the port. This pattern is not limited to those applications which require access to a resource to which only the superuser ID grants access: a document viewer, after opening a user specified file, could destroy the identity object to diminish the affects of a malicious macro.

It is possible to implement a similar scheme on Unix with the help of a UID and GID dedicated to the program instance. This solution has the disadvantage that it requires help from the system administrator who must reserve a UID and GID for each program instance that wants to use this technique. This is further complicated as only a process running as root is normally permitted to change its UID and GID. These factors taken together limit the use of this technique to critical system services. It is worth noting that recent work by Seaborn on Plash, a series of tools for practical least privilege for GNU/Linux packages this functionality [32]. Plash automatically allocates UIDs and GIDs, creates `chroots` and facilitates selected access delegation. This permits many programs to be run unmodified and without cumbersome configuration. Yet, to do this, Plash must provide safer implementations of dangerous functionality.

Servers which authenticate users such as FTP or ssh servers can also take advantage of this additional functionality: unlike the previous class of applications, these programs require the ability to change UIDs during the lifetime of the program. Because they interact with unauthenticated users while holding large amounts of authority, they are highly targeted. In particular, such programs are susceptible to buffer overflows and input validation errors during the login phase. On the Hurd, such a program instance can run with no identities. Then, after a user has provided a user name and password, it presents them to the password server in exchange for an identity object thereby increasing its authority. The Hurd's login program does this. Because the amount of havoc an attacker can wreck is proportional to the accessible authority, the effects of a breech are diminished proportionally.

On Unix, privilege separation [29] is used to isolate the parts of a program requiring root authority. This technique uses multiple processes which collaborate. One process implements the typically small number of required privileged operations exposing them via a simple interface and the other, the balance of the functionality. This eases verification of the privileged program and makes exploitation of bugs more difficult. This pattern also requires help from the system administrator for the allocation of an unused UID and GID for each program instance which uses this technique.

2.3.2 Authorization

IBAC is based on knowing the identity of the user. Thus, when a subject authenticates access to an object which is controlled by such a regime, it needs to disclose its identity to the object. On Unix, the identity manager and most servers are in the same trust domain. On the Hurd, this is not the case. This exposes a tension: the object must be confident of the authenticity of the presented identities but, because the client may not trust the server with its identity, the object should only be able to examine the identities, not be able to use them. Hohmuth et al. list a number of useful scenarios for the latter case [18]. The Hurd's `auth` provides a three-way handshake to support such mutually suspicious collaboration and sharing without prior arrangement.

When a client wishes to authenticate access to an object, such as when it crosses a translator boundary and only has an unauthenticated root handle, it uses `io_reauthenticate`. The client includes a so-called *rendezvous* capability referencing a new kernel message queue (Figure 2(a)). In response to this request, the server invokes the `auth_server_authenticate` method on a capability which references an identity object on a trusted `auth` server. It includes the *rendezvous* capability as well as a second capability which names an as of yet unauthenticated handle to the object. Without waiting for a reply, the client invokes the `auth_user_authenticate` method on a capability naming the identity object whose contents it wishes to disclose to the server. It also includes the *rendezvous* capability as an argument (Figure 2(b)). The `auth` server then pairs the *rendezvous* capabilities and completes the handshake by returning the identifiers in the identity object to the server and the capability naming the new object to the client (Figure 2(c)). The server then stores the identifiers in the object.

The capability to the new object is returned via the authentication server to avoid giving access to the authenticated object to an unprivileged third party. This prevents man in the middle attacks: if Bob has a connection to Alice via Mallet, i.e., Mallet is forwarding messages between the two, Alice and Bob can be sure that the established channel only traverses the union of their reliance sets. Otherwise, when Alice replies to Bob with the capability, Mallet could proxy the capability and observe all communication.

The authentication interface and protocol are designed such that it is possible to transparently interpose proxy `auth` servers between both the client and the common `auth` server as well as the server and the common `auth` server. This makes the implementation of, e.g., Debian's `fakeauth` relatively straightforward.

2.4 Abstractions

Although the Hurd provides a rich set of abstractions, many are easily circumvented for either flexibility or efficiency reasons. The `store` abstraction provides an example of the latter: a store abstracts seekable data stores such as files and block devices as well as combined stores such as those in a RAID configuration.

As there is a cost involved in providing this level of indirection, sufficiently privileged programs (i.e., programs which would be able to access the underlying store in its entirety in the case of a partitioned store) are able to bypass the store translator using the `file_get_storage_info` method. In the simplest case, the returned information can be passed to `libstore` thereby avoiding the additional context switches. This strategy has similarities to the Exokernel approach to abstraction elimination: remove mandatory abstractions and instead implement abstractions in user libraries [19].

2.5 Legacy Support

As it was expected that the bulk of applications would use the POSIX interface, it was important they not be treated as second-class citizens, e.g., via support through a poorly integrated subsystem. To this end, compatibility was realized through the use of a so-called *fat* C library where much of the POSIX API is implemented in terms of Hurd and Mach mechanisms.

This strategy provides several advantages: many legacy applications can be used with minimal modification; applications are rarely disadvantaged for having used the POSIX API; and few modifications are required to take advantage of Hurd features. For example, an FTP server on Unix normally requires the authority of the root user. This program can be modified using two isolated changes to take advantage of the Hurd's protection mechanisms. As described in Section 2.3.1, an FTP server needs to bind to a privileged TCP port and be able to change users. Instead, the server can drop its root identity after binding to the TCP port and use the password server to authenticate the user and obtain the respective identity object. Our experience suggests that such isolated changes are much more readily integrated by upstream authors.

3 A Critique

The design presented above has a number of shortcomings in reaching its own stated goals as well as the demands of a modern computing environment.

3.1 Security and Protection

Computers are used to store and process data. This data has value and, as such, should have appropriate mechanisms in place according to its owner's security policy to protect it from unauthorized access and disclosure and to ensure its availability. Although the Hurd provides some mechanisms for protecting data from other users, the Hurd does not provide mechanisms for the enforcement of a security policy for particular program instances: programs are assumed to represent the interests of the user and, as such, are run with the user's authority.

Although the US military was acutely aware of such threats over three decades ago [1], in the early 1990s when the Hurd was designed, the average computer user did not consider them important: malware was mostly non-existent. This sentiment is echoed by Bushnell at the end of his architectural overview of the Hurd: "[y]ou can't harm a process by giving it extra permission" [5]. Yet as programs are buggy [25, 26], sometimes malicious and often exploited [36], not providing some mechanism today is a serious shortcoming.

To mitigate these problems, users need to be able to provide a program instance access to only the objects it needs to realize the user's intent. That is, it should be possible to run programs consistent with the principle of least privilege (POLP) [31].

The discretionary authority reduction pattern described in Section 2.3.1 does not address this problem: although it useful in mitigating the effects of bugs and their exploitation, the use of this pattern is at the discretion of the program—not the user. As such, although it represents good programming practice, users have not gained any control: they still rely on the goodwill of programmers.

Capability practitioners contend that a well structured capability system can run programs under a POLP regime without modification and without being invasive to users. Polaris [34] and Plash [32] are two such systems built on top of Windows XP and GNU/Linux respectively which illustrate that this is possible. Their frameworks are based on three observations. First, most programs require access to a limited number of objects which can be statically enumerated. Second, authorization can often be inferred, e.g., when the user double clicks on a resource to launch the associated application [38]. Finally, additional access at run-time is only required by interactive programs and most often after a user interaction via an open or save dialog box. These can be replaced with a call to a trusted program, the *powerbox*, with access to all of the users resources which interfaces with the user on the programs behalf and delegates access to those objects the user authorizes. This can be done transparently by replacing the appropriate library routines.

If the Hurd were to abandon IBAC and implement such a framework, the structure of most Hurd objects would nevertheless remain problematic: most Hurd objects convey large amounts of authority which is not easily decomposed. This is often motivated by concern for POSIX compatibility. A directory, for instance, provides access not only to the sub-tree it dominates but to the entire name space due to dot-dot naming the physical parent. The behavior of dot-dot can be overridden using `file_reparent`, however, this requires explicit action violating the principle of safe by default [31].

3.2 Malicious File Systems

Most legacy applications assume that file systems are not malicious. This assumption is reasonable on a system where all file systems are part of a process's reliance set, as is the case on Unix. On the Hurd, where arbitrary programs are able to attach to and extend the virtual file system, this assumption leads to a security vulnerability. An ignorant backup program, for instance, may walk the VFS copying the objects it finds. A malicious file system can mount a denial of service attack by generating an infinitely deep directory structure populated with arbitrary amounts of pseudo-random data, using, relative to the backup program, little resource.

It can be argued that there are always scenarios requiring defensive programming and that this is simply one of which Hurd programs need to be aware. This would be correct but avoids the question of legacy support.

Our observation is that compatibility is not only respecting the interfaces but also the deep assumptions that programs have regarding the API. Thus, it is the responsibility of the compatibility layer to recognize these assumptions and to meet them.

3.3 A File or a Directory?

On Unix, a VFS node is strongly typed: it is either a file, a directory or some other well defined object. Yet, as the directory and file interfaces are mostly distinguishable, it is possible for an object to implement both. This appears useful as data can sometimes be seen as either a single file or a structured hierarchy of objects. It is convenient, for instance, to copy a backup archive by copying a single file, however, when searching for a file in the same backup archive, it is more convenient to view the data backup as a directory hierarchy and have the ability to search it using normal tools such as `find` and `grep`.

In this example, the view is selected by the use of disjunct sets of interfaces. Some programs, like `grep -r`, support multiple object types and rely on advice from the object in the form of the file type information for disambiguation. In this case, which view should be presented depends on the intent of the user. This motivates a mechanism by which a user or an agent acting on his behalf can acquire separate names for separate views on the same underlying object. Requiring explicit naming of views reduces ambiguity thereby simplifying code, removes a security risk and provides the user with greater expressiveness through the uniform interface. Adding a new naming mechanism would require that all programs be taught how to use it. Instead, the existing naming framework should be reused with the same effects and objects should implement a single type.

3.4 The Dot-Dot Directory Entry

The resolution of dot-dot to the physical parent was motivated by POSIX compatibility. Unfortunately, it requires server help. This is further complicated as processes may have different views of the VFS, e.g., processes running in a `chroot`. Additional support is thus required to override dot-dot so that `chroot`ed processes (and their children) do not see the physical parent of the root but a VFS root.

The `file_reparent` appears to solve this issue, however, introduces its own problems. If a translator is itself started in a `chroot`, say `/chroot`, and a process which has a different root directory, say `/`, attempts to resolve a path starting in the translator's name space but which ascends the hierarchy traversing the translator's root directory, it will get unexpected results.

Assume the translator is mounted on `/chroot/mnt` and the process, starting at `/chroot/mnt`, looks up `../../foo`. When the process invokes the `dir_lookup` method, the translator returns a retry message including a capability referencing the underlying node but whose logical root is set to `/chroot`. When the process retries the rest of the path with this handle, it will resolve to `/chroot` rather than `/` as the underlying file system compressed dot-dot at the *handle's* logical root. Had the process resolved dot-dot on its own, it would have arrived at the correct directory. What has happened is that the naming context has changed.

Pike argues for lexical name resolution, i.e., making applications responsible for the resolution of dot-dot, as POSIX semantics are actually rarely what users want [27]. Adopting such a policy on the Hurd, would not only improve the user experience but would also fix the above problem by entirely removing the need for server resolution of dot-dot and thus `file_reparent`. This also has the additional fortunate effect of significantly simplifying servers and proxies.

3.5 Passive Translators and Naming

Translators are started in two different scenarios: by a program, at the behest of a user; and by a file system, as it traverses the VFS and accesses a node which has a passive translator setting but no running translator. The latter scenario was motivated by the requirement for a mechanism which restores running translators after system restart.

When a program starts a translator, as when it starts any program, it first locates the executable object. This usually means performing a `dir_lookup` on the appropriate object with the provided path name. Having found the object, it then instantiates a new program instance including providing it with a naming context.

As passive translator settings do not include naming contexts—they are strings—the file system uses its own default naming context. Users tend to encounter this problem when they provide a relative path in the passive translator setting rather than an absolute path.

More of a concern, however, are the implications for enforcement of security policies: `chroot` is sometimes used as a protection mechanism as it restricts the name space of a set of processes, limiting reachability. Making the name space of the file system available to the encapsulated process renders this redundant.

Consider the case where the root of the translator's naming context is `/` and the root of a `chroot`ed program instance's naming context is `/chroot`. That is, the program which started the encapsulated program instance obtained a capability to `/chroot`, invoked `file_reparent` on it to override the object's dot-dot entry with the null capability and installed the resulting capability in the encapsulated program instance's root directory capability slot. The encapsulated program instance can escape by setting a passive translator on `/chroot/foo` (what it locally knows as `/foo`) and then `stat`ing the object:

```
$ settrans -cp /foo /hurd/firmlink /
$ ls -l /foo
```

When the translator examines the object, it sees that the node has no translator but does have a passive translator setting. It proceeds to start a translator by resolving the command name to an `fs` object relative to its own root and, in executing it, providing the program instance with a capability to its own root. If the translator is, as above, a firm link, a translator which makes some name space available at the translated node, then the encapsulated program has successfully escaped. Alternatively, the encapsulated program instance could debug the translator (it has the same UID) and simply copy the capability.

To avoid this, the program instance that sets the passive translator must also provide a naming context in which the passive translator is interpreted as well as a default naming context for the translator instance. That is, it must provide closures, not just strings [30]. Arguably, the file system has at least the former: it need only remember which handle the passive translator was set with. The problem is that the handles are not persistent and the main motivation behind passive translators is that since capabilities and processes are not persistent, a method is needed to restore translators.

This problem, known as *trusted recovery* [9], can be fixed by making the system persistent thereby circumventing the reconfiguration problem [21, 33, 37]. This may appear as overkill, however, persistence is a highly desired feature: desktop environments work hard to restore running applications to the state they were running in when the user logged out; and many users, in particular laptop users, choose to suspend to disk or memory rather than turn the computer off.

3.6 Server Allocations

On the Hurd, most objects are accessed via sessions. This is usually motivated by POSIX compatibility. File handles, for instance, are required to maintain the cursor position and record the logical dot-dot binding. For each session, the server must allocate some storage. In the case of objects that cause allocations, this is not a problem. However, with only sense access to an object, the client should not be able to allocate additional storage. Yet, this is the case and, as such, a malicious program, having only authorized to use the sense interface to an object, is able, in bounded space, to cause the server to consume an unbounded amount of memory: it simply enters an endless loop performing an `open` on the file. The server cannot tell which process is causing the allocation; it can, at best, implement a local per-user memory quota. This has the unfortunate side effect of potentially limiting legitimate uses of the server (what is the right quota?). It also makes a new denial of service attack possible: an encapsulated process can exhaust the user's resource quota. Again, identity based access control is inadequate.

To avoid this, sense interfaces should be designed such that they do not require server allocations or that the client provide the resources by passing a capability which names a resource pool of some sort, similar to EROS space banks [33] or resource containers [3], against which the server then allocates the resources.

When possible, allocations should be avoided. In the case of the cursor, this is possible. As multiple processes can access a single file descriptor (i.e., a single handle), this raises the question of how to coordinate access to the cursor. The majority of shared file descriptors name pipes. As pipes are unseekable, they do not require a cursor. In rest of the rare cases in which two processes share a file description to a seekable object, they must coordinate access to the cursor anyway. This already requires that they be mutually trusting. However, as this is quite complicated, it is normally avoided by immediately `dup`ing the file handle on receipt. Thus, it appears, a shared cursor is rarely required.

4 Evaluating Mach

Liedtke argues that the microkernel approach to system structure is often rejected based primarily on the perceived high cost of the message passing mechanism [22]. We observe additional shortcomings in Mach regarding resource scheduling and resource accounting that we contend also need to be addressed for the microkernel approach to have competitive performance and be able to support safe use of potentially malicious programs.

4.1 Resource Scheduling

Most systems provide tasks the illusion that they are running on a machine with infinite resources: tasks allocate virtual memory, memory that the kernel transparently moves between physical memory and backing store; likewise, threads need never explicitly yield the CPU as the kernel automatically preempts them [7]. This is convenient insofar as it relieves applications from having to respond to resource shortages, perform resource multiplexing and simplifies dynamic reallocation of resources. Assuming that competition for the physical resources remains relatively low, good resource utilization can be achieved without application support as evidenced by the many monolithic kernels which successfully employ such techniques. When this assumption is violated, when significant resource multiplexing occurs, system performance can significantly degrade if poor scheduling decisions are made [17]. It also tends to push real-time applications into a privileged scheduling class.

4.1.1 Efficiency

For centralized resource management, a monolithic kernel has two resource scheduling advantages over a multiserver: it can better predict resource usage patterns and more components can interact with the scheduler.

Due to their centralized nature, monolithic kernels have a higher level view of how users and processes use resources: they implement the high level abstractions such as UIDs, file systems and network protocols and directly interact with the users of these resources. These abstractions can provide important hints regarding expected resource usage. A monolithic kernel, for instance, can relatively straightforwardly implement file based read-ahead. On the Hurd, these abstractions are implemented by user-space servers, which Mach does not only not regard as special but of which Mach has no additional knowledge. As such, by itself, Mach is only able to implement disk based read-ahead. Such optimization techniques cannot be reliably implemented in the respective user-space servers as these processes do not have information regarding memory pressure and thus cannot correctly determine how aggressively to act.

Second, because these high-level abstractions are implemented by the monolithic kernel, these components are able to hook into the resource management framework in ways which violate formal component boundaries. Linux, for instance, in addition to employing a page replacement strategy based on memory access patterns, drains a number of caches including that maintained by the widely used slab allocator [4], the directory entry cache, the inode cache and the disk quota entry cache. Gorman reports that the last three have a "cascading effect [which] allows a lot more pages to be freed" [15, Sect. 10.4]. In a strict sense, these caches consist of anonymous memory.

These components have local knowledge which the resource manager by itself is unable to observe. Such local knowledge is not limited to kernel components: many applications could improve their resource use if they had more control over the scheduling policy. This includes database applications [35], scientific applications [6], multimedia and other adaptive applications [10], garbage collectors [17] and cache managers.

The last class of applications, compute data and are able to cache it for opportunistic reuse. This includes interactive applications such as web browsers, which lay out HTML documents, and image viewers, which decompress and scale image files. Keeping these computed objects in otherwise idle memory can result in reduced processing costs (both CPU time and power) when they are again required. Mach, like most operating systems, however, provides no mechanism to restrict such data to idle memory or a feedback mechanism. As such, the memory manager may page the data to backing store. This may be more expensive than simply rerendering the object on demand. A JPEG file, for instance, may be a few hundred kilobytes compressed or several megabytes uncompressed. Sending all of this data to backing store and then paging it in again when required is likely less efficient than simply discarding the data and reading the original image file and uncompressing it again on demand.

As applications are unable to intelligently manage their cache, they must act conservatively. GQView, a popular image viewer for GNOME, maintains, by default, a 10 MB cache of rendered images [12]. gThumb, another image viewer for GNOME, keeps a static cache of four images and preloads the image following and that previous to the requested image [2]. Neither application pro-actively frees its cache.

Yet even this can cause resource underutilization. Getty of the One Laptop per Child (OLPC) project, a group developing a resource poor laptop for children in developing nations based on GNU/Linux, has advised developers to act conservatively: to avoid caching data and rerender when necessary [14]. Similarly, Linksys recently revised their popular router to use vxworks instead of GNU/Linux and were able to halve the 16MB RAM and 4MB of flash thereby increasing profitability despite the engineering costs [24].

The need for such mechanisms has been articulated by the academic research community over a decade ago [20, 13]. Monolithic systems have resisted this problem by generally performing well enough for common workloads and relying on resource over-provisioning, special tuning and highly privileged and specialized extensions to handle the rest. We contend these problems are not isolated to multi-server systems but simply exasperated by them and that safe mechanisms need to be found to better exploit local information and knowledge.

We observe that there are two specific classes of scheduling scenarios over which more control is useful: distribution and multiplexing. In the first case, some controlling agent wants to distribute resources among a number of principals. For instance, the system administrator wants to divide resources among users using, e.g., proportional share; and a user, among her own tasks according to her priorities. The second class is that of applications which make direct use the resources. They want to influence the memory eviction policy and how threads are scheduled.

4.1.2 Real-Time

Applications with real-time properties are those which contain tasks the result of whose utility is influenced by the wall clock. The most obvious example of real-time applications found on general-purpose operating systems are multimedia applications. Interactive applications also have a real-time aspect. Although support for real-time applications is not an explicitly stated goal of the Hurd, given the increasing use of applications which have such properties, the Hurd's lack of support reduces the usability of the system.

The realization of real-time properties depends on the ability of programs to be able to make predictions of progress. This does not necessarily require hard resource guarantees: statistical guarantees for the these classes of applications are sufficient. What is required is that the amount of resources available to the application, their access properties be known and these be reasonable.

Virtual resources as specified by Mach and most commodity operating systems fail to satisfy this last property. The worst case access times of virtual resources is essentially infinite in particular, compared with their average case access times. Currently, applications have to hope that data will not be paged and that the CPU allocation will remain at least as large as in the recent past. This encourages conservative behavior.

An approach to enable applications to meet real-time properties is to provide a real-time application class. Applications in this class are allowed to make guaranteed physical resource reservations. As this is easily abused, the admission criteria are quite strong. We would like to improve the ability of unprivileged applications to fulfill their real-time properties.

This motivates the mechanisms to allow untrusted programs to request resource schedules, to make scheduling information available and to provide visible revocation. Applications with real-time properties also desire the ability to request resource schedules which include properties such as duration and jitter. This ties in with the desire for better control of resource scheduling and should be solvable in a uniform fashion.

Additionally, many real-time applications also have the property that they are adaptive by which is meant that they are able to trade result quality against utility: if there are insufficient resources available to finish the task within the appointed deadline but a lower quality result can be produced within the time constraint, it is of greater utility than a perfect but late result [10].

4.1.3 Safety

The problem posed in introducing mechanisms to support higher control of policy is that they can quickly become complicated compromising system safety [11]. In particular, the kernel must be careful to not create dependencies on the correct behavior of code which is not intended to form part of the reliance set, i.e., a buggy or malicious program should not be able to adversely affect other programs in the system. In a monolithic system, memory components which could take advantage of this are already part of the reliance set.

4.2 Resource Accounting

Consistent with the illusion that resources are infinite, Mach performs no resource accounting. This introduces a security hole as virtual resources are, in fact, limited: the degree of multiplexing of physical memory is limited by the amount of backing store reserved for that purpose, opening up the possibility of a denial of resource attack.

Because resources are not accounted, simply allocating large amounts of resources is sufficient to perform a denial of resource attack. It might seem that the affects of such an attack could be mitigated without changing the resource allocation API by enforcing reasonable per-process quotas. This is easily overcome by malicious entities, however, by spawning multiple processes. Extending the quotas to users will not work either: users are a Hurd abstraction not known to Mach.

The problem underpinning the above thought experiment is the assumption that we can successfully coax an implicitly named resource principal out from where there is none: the process which allocates the resource is often not the resource principal. When a process reads data from a file, it invokes the `io_read` method on a capability naming an `io` object. The server then allocates memory on behalf of the client, reads the data into the memory and returns it to the calling process. Thus, although the file system invoked the kernel to perform the resource allocation, the allocation should, in this case, be charged to the process, or rather, the principal on whose behalf the process is running. This problem is explored in the context of monolithic kernels by Banga et al. in [3].

To ward off denial of resource attacks, the idea of a resource principal or container needs to be introduced. Such an abstraction needs to be able to be passed around in a secure manner allowing servers to allocate resources on behalf of clients but which clients are able to recover in the case where the server misbehaves. Later versions of OSF Mach introduced so-called resource ledgers for accounting the use of wired memory and swap space, however, they rely on ambient authority, i.e., they are not directly named, complicating allocation. A better model may be KeyKOS's CPU meters and space banks [16] and as later evolved in EROS [33] which do not have these deficiencies.

5 Lessons and Thoughts

Our experience with the Hurd has led to a number of realizations that may help future designers of general purpose operating systems.

5.1 Security

The Hurd empowers the user by making a number of useful privileged operations unprivileged. By reifying identities, it also provides discretionary authority reduction mechanisms allowing programs to protect themselves from attackers. Neither of these address a major security problem of today: the inability to protect data from program instances.

To achieve this, programs should be run consistent with the principle of least privilege. Recent research on capability systems suggests that capability can provide a usable framework to realize such a system. In particular, to allow run-time delegation of authority, the powerbox, a trusted program, interacts with the user on the program's behalf.

5.2 Naming and Binding

Recovering the configuration of the system on restart improves usability. To facilitate this, the Hurd allows users to save translator settings in nodes. When the node is accessed and no translator is running, the file system can then transparently restart the translator. The problem that this raises is that the naming context cannot be serialized, leading to a number of security concerns as a malicious user is able to confuse file systems.

This is a general problem wherever symbolic names are severed from their naming context and often occurs at the storage boundary where there is no easy way to serialize a naming context. As symbolic names are primarily of interest to users and not to programs, we suggest that symbolic names be avoided and capabilities be used as designators. The problem this raises is that capabilities, like naming contexts, need to be saved. By making the system persistent, this problem is circumvented.

5.3 Resource Management

We have noted that although virtualized resources are convenient, they are also often problematic due to inefficient resource scheduling and their ineffectiveness when trying to realize real-time properties. This former problem is not unique to multi-server systems but particularly pressing as it appears the techniques used by monolithic kernels to compensate for lack of local knowledge, namely, introspection of high-level functionality to help predict resource usage patterns, cannot be used by a microkernel where such functionality is implemented in user-space. We contend that an interface is needed to allow unprivileged programs an increased ability to influence resource scheduling both regarding distribution and multiplexing.

5.4 Legacy Support

Legacy support is highly desirable for non-mainstream operating systems: application developers tend to target widely deployed systems, however, deployment penetration appears to be strongly correlated with the number of applications available. Moreover, if a system lacks support for just one or two applications, users will reject it.

The Hurd aimed to not only run legacy applications, but to tightly integrate them and provide them with many of the advantages of Hurd mechanisms. In this regard, the Hurd was successful. The Hurd, however, in its strict support of POSIX, unnecessarily complicated the system structure. We have observed that this was often motivated by questionable features such as server resolution of dot-dot and a server implemented cursor.

Finally, care must be taken to preserve non-functional API and ABI requirements such as trust assumptions. The most important example of the Hurd's failure in this regard is that of legacy programs being susceptible to attack by malicious file systems.

6 Conclusion

The Hurd started with the observation that a number of useful Unix mechanisms, in particular, those regarding the extension of the VFS, should be available to users. By adopting a multi-server system and a decentralized naming framework, the Hurd makes it possible for users to provide their own file systems implementations and integrate them into the VFS.

The Hurd's has two noteworthy security shortcomings: it does not provide mechanisms to protect resources from program instances; and symbolic names are often separated from their naming contexts.

An important goal of the Hurd was to support POSIX applications. Sometimes this was done too faithful compromising parts of the system structure. Other times, the Hurd failed to consider important aspects of legacy compatibility, namely, assumptions applications have regarding behavior.

To allow the efficient use of resources on microkernel based systems, it appears that applications must participate in resource scheduling. We observe that there are two main areas where applications can usefully extent greater control of resource scheduling: distribution and multiplexing.

7 Acknowledgements

Roland McGrath as well as Thomas Bushnell have been pivotal in helping the early development of our understanding of the Hurd. We are indebted to Jonathan Shapiro for informative discussions regarding security and system design. Michael Hohmuth, Thomas Bushnell, Jed Donnelley, Thomas Schwinge and Jonathan Shapiro provided helpful criticism of early drafts of this document.

References

[1] Anderson, J. P. Computer security technology planning study. Tech. rep., Electronic Systems Division, Oct. 1972.

[2] Bacchilega, P. gThumb v2.8.0. `http://gthumb.sf.net`, Nov. 2006.

[3] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *3rd USENIX Symposium on Operating Systems Design and Implementation* (Feb. 1999).

[4] Bonwick, J. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer* (1994), pp. 87–98.

[5] Bushnell, M. Towards a new strategy of OS design. *GNU's Bulletin 1*, 16 (Jan. 1994).

[6] Cox, M., and Ellsworth, D. Application-controlled demand paging for out-of-visualization. Tech. Rep. NAS-97-010, NASA Ames Research Center, July 1997.

[7] Denning, P. J. The working set model for program behavior. *Communications of the ACM 11*, 5 (May 1968), 323–333.

[8] Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3 (Mar. 1966), 143–155.

[9] Department of Defense. *Trusted Computer System Evaluation Criteria DOD 5200.28-STD*. Dec. 1985.

[10] Domjan, H., and Gross, T. R. Managing resource reservations and admission control for adaptive applications. In *30th International Conference on Parallel Processing* (Sept. 2001).

[11] Druschel, P., Pai, V. S., and Zwaenepoel, W. Extensible kernels are leading os research astray. *Proceedings of the 6th Workshop on Hot Topics in Operating Systems* (May 1997).

[12] Ellis, J. GQView v2.0.4. `http://gqview.sf.net`, Dec. 2006.

[13] Engler, D. R., Gupta, S. K., and Kaashoek, M. F. AVM: Application-level virtual memory. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)* (May 1995), pp. 72–77.

[14] Getty, J. $100 laptop / OLPC (One Laptop Per Child). `http://gettysfamily.org/wordpress/?p=11`, Nov. 2005.

[15] Gorman, M. *Understanding the Linux Virtual Memory Manager*. Bruce Perens' Open source series. Prentice Hall Professional Technical Reference, 2004.

[16] Hardy, N. The KeyKOS architecture. In *Operating Systems Review* (Oct. 1985), vol. 19, pp. 8–25.

[17] Hertz, M., Feng, Y., and Berger, E. D. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (June 2005).

[18] Hohmuth, M., Peter, M., Härtig, H., and Shapiro, J. S. Reducing TCB size by using untrusted components - small kernels versus virtual-machine monitors.

[19] Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceño, H. M., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. Application performance and flexibility on exokernel systems. *16th Symposium on Operating Systems Principles* (1997).

[20] Kiczales, G., Lamping, J., Maeda, C., Keppel, D., and McNamee, D. The need for customizable operating systems. In *Proceedings of the Fourth Workshop on Workstation Operating Systems* (Oct. 1993), pp. 165–169.

[21] Landau, C. R. The checkpoint mechanism in KeyKOS. In *Second International Workshop on Object Orientation in Operating Systems* (Sept. 1992).

[22] Liedtke, J. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating System Principles (SOSP-14)* (Asheville, NC, Dec. 1993).

[23] MILLER, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, May 2006.

[24] NEEDLEMAN, R. Technology marches backward. `http://reviews.cnet.com/4520-3000_7-6542073.html`, June 2006.

[25] OSTRAND, T., AND WEYUKER, E. The distribution of faults in a large industrial software system. In *ACM SIGSOFT International Symposium on Software Testing and Analysis* (2002), pp. 55–64.

[26] OSTRAND, T., WEYUKER, E., AND BELL, R. Where the bugs are. In *ACM SIGSOFT International Symposium on Software Testing and Analysis* (2004), pp. 86–96.

[27] PIKE, R. Lexical file names in Plan 9 or getting dot-dot right. In *2000 USENIX Annual Technical Conference* (June 2000), USENIX, Ed.

[28] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (July 1974), 412–421.

[29] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *12th USENIX Security Symposium* (Aug. 2003).

[30] SALTZER, J. H. Naming and binding of objects. In *Operating Systems, An Advanced Course* (London, UK, 1978), Springer-Verlag, pp. 99–208.

[31] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. In *Proceedings of the IEEE* (Sept. 1975), vol. 63, pp. 1278–1308.

[32] SEABORN, M. Plash: tools for practical least privilege. `http://plash.beasts.org`.

[33] SHAPIRO, J. S., AND ADAMS, J. Design evolution of the EROS single-level store. In *2002 USENIX Annual Technical Conference* (2002), pp. 59–72.

[34] STIEGLER, M., KARP, A. H., YEE, K.-P., AND MILLER, M. Polaris: Virus safe computing for Windows XP. *Communications of the ACM 49*, 9 (2006), 83–88.

[35] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM 24*, 7 (July 1981), 412–418.

[36] THOMAS, R., AND MARTIN, J. The underground economy: priceless. *;login: 31*, 6 (Dec. 2006).

[37] TULLMANN, P., LEPREAU, J., FORD, B., AND HIBLER, M. User-level checkpointing through exportable kernel state. *IEEE International Workshop on Object-Orientation in Operating Systems* (Oct. 1996).

[38] YEE, K.-P. User interaction design for secure systems. In *International Conference on Information and Communications Security* (2002).

[39] YOUNG, M., TEVANIAN, A., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D., AND BARON, R. The duality of memory and communication in the implementation of a multiprocessor operating system. In *11th Operating Systems Principles* (Nov. 1987), pp. 63–76.