Masaryk University
Faculty of Informatics

Bachelor thesis

Improved VFS design for HelenOS

Jiří Zárevúcky

Brno, 2013

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Declaration

Hereby I declare, that this thesis is my original work, which I have created by my own. All sources, references and literature used or excerpted during the course of this work are properly cited and listed in complete reference to the due source.

Acknowledgement

I would like to thank my supervisor Zdeněk Říha for his professional advice and assistance during my work. Additionally, I would also like to thank every present or past contributor to the HelenOS operating system, especially the core development team, for creating an excellent platform for testing new ideas, and a friendly environment for discussing them.

Abstract

The thesis analyzes the purpose and implementation of file systems in modern operating systems descendant from UNIX, with focus on security issues that emerge in modern computing, where the code executed in a shared environment is rarely created (or even known) by the user executing it. A new file system architecture is then proposed that focuses on inter-process isolation as the means of ensuring security. This architecture is demonstrated by creating a prototype implementation within the HelenOS operating system, which is also described.

Keywords

operating system, file system, IPC, security, files, user accounts, UNIX, sandboxing

Contents

Introduction **ix**

1 Preliminaries **1**
- 1.1 Operating systems 1
  - 1.1.1 General overview 1
  - 1.1.2 Microkernel vs. Monolithic 2
  - 1.1.3 File systems 3
  - 1.1.4 Existing systems 4
    - 1.1.4.1 Traditional UNIX / POSIX 5
    - 1.1.4.2 GNU Hurd 6
    - 1.1.4.3 Plan 9 from Bell Labs 7
    - 1.1.4.4 4.4BSD-Lite 8
- 1.2 Summary 9

2 The proposed design **13**
- 2.1 Requirements 13
- 2.2 Filesystems 16
- 2.3 Files 17
  - 2.3.1 STORAGE files 17
  - 2.3.2 STREAM files 17
  - 2.3.3 Directories 18
- 2.4 File handles 18
  - 2.4.1 Access permissions 19
  - 2.4.2 Methods 19
- 2.5 Bind operation 22

3 Implementation **25**
- 3.1 HelenOS 26
- 3.2 IPC primitives 26
- 3.3 Advanced IPC mechanisms 27

3.4 Original file system implementation 28
3.5 New file system implementation 29
3.6 Inbox 30
3.7 Server 31
3.7.1 Unions 33
3.7.2 Restricted nodes 33
3.7.3 Virtual pipes 34
3.7.4 Unfinished work 34
3.8 User shell changes 36
3.8.1 Applications and Profiles 36

4 Conclusion **39**

Bibliography **41**

A C functions **43**
A.1 Conventions 43
A.2 Header files 44
A.3 Function listing 44
A.3.1 `<vfs/file.h>` 44
A.3.2 `<vfs/path.h>` 46
A.3.3 `<vfs/dir.h>` 47
A.3.4 `<vfs/inbox.h>` 47

B The Go language **49**
B.1 Porting the runtime 50
B.2 Fitness for the purpose 51

Glossary **53**

Introduction

Long term data storage is one of the key capabilities of modern computer systems. It is a deeply integrated functionality of virtually every contemporary operating system. Mechanisms related to persistent storage are present on almost every level, starting with basic drivers in the very core of the system, and ending with consumer's blu-ray movies and family photographs. It is therefore no surprise that such mechanisms have many expectations and even more potential problems, being some of the most visible parts of the operating system.

The persistent storage has several different, conceptually very distinct uses. Most obviously, the system itself needs persistent memory for its own components, data and configuration. For the most part, ordinary users have no business accessing (even knowing existence of) such data, and they should not be able to. Obviously, an exception to that is any administration mode the system provides.

Another of these uses is a simple extension of main memory. Most operating systems in use today are composed of transient and unreliable processes. To achieve an illusion of persistence and robustness, such processes need to store pieces of information that are used by subsequent executions of the same program. This includes application settings, data caches, work history, etc. There are two common factors in these examples. One is that the user does not need or want to manipulate with such data outside of the program (or even be aware of it), although they should be able to directly access it. The second is that such data is highly application-specific and no other application should ever need to access it, except in response to explicit user request.

Lastly, there is the data users themselves manage and use. This can be anything from financial documents and cryptographic certificates, to a music library and high-definition movies. Although this kind of data is not a responsibility of any application, some applications are used to access, view or change it. However, even such applications only need to access data they immediately work with. For example, a malicious scripted document should not be able to access any other documents, even if user commonly uses the same application to view them.

In chapter 1, I will first skim over several basic notions important to understanding operating systems. Anyone familiar with the concepts should be able to skip the early sections. After the basic notions are established, I will introduce several existing operating systems. Knowing the specifics of their respective approaches to data management will allow me to draw several conclusions regarding the suitability of current operating system concepts to the scenarios I mentioned above.

After identifying and discussing flaws in contemporary operating systems, chapter 2 will follow up by formulating a set of requirements that should be met by the system. I will present arguments for those requirements and explain how they help correct the identified flaws.

In the rest of the chapter, I will utilize those requirements by presenting a variant of the traditional file system abstraction. I will argue that this slight variation is sufficient to resolve problems in earlier systems, with very little negative impact in terms of (un)familiarity. I will also propose a programming interface for this system, which meets all the formulated requirements, yet is still close (at least conceptually) to the traditional POSIX interfaces.

In chapter 3, I will describe the prototype implementation of my proposed design. I will introduce my platform of choice and discuss not only the internal details of the system, but also several of the practical problems I have run into while implementing the design, some of which forcing me to reconsider choices I will have made previously.

Finally, section 3.8 will briefly discuss how can user-level software utilize the redesigned programming interface to form an environment that is intuitive to work with, but also secure in face of rogue applications and programming errors. A layout is proposed for system-wide, user-specific and application-specific data, and a set of policies is described that helps utilize all aspects of the new design the best way possible.

Chapter 1

Preliminaries

1.1 Operating systems

1.1.1 General overview

For the proper function of any reasonably complex computing environment, an *operating system* (shortly referred to as an *OS*, or plural *OSs*) is an indispensable component. There is a multitude of definitions of what exactly an operating system is. Rather than trying to select and explain any single one of those definitions, what I will do in the following paragraphs is to describe not the meaning of words, but rather what the operating systems are intended for and what is their function.

At the most elementary level, OS is a software layer that facilitates — in a very broad sense — the communication of user with the underlying machine. Here *user* can mean not only the human operator of the computer, but also an application intended for the human to use, or a service that uses resources of the system to accomplish its own tasks. By not being overly strict about what the *user* truly is, one is able to deal with different levels of abstraction without affecting the idea of an OS itself.

To achieve and simplify such communication, OSs provide a varying degree of abstractions. It is easier to think of independent threads of computation than it is to think about scheduling work among processors. It is simpler to draw to a graphical display by issuing understandable commands rather than by writing magic numbers to a prespecified area of memory. It is easier to work with named files than with an ATA bus.

The abstractions are stacked on top of each other, and usually the OS is structured such that each abstraction is being provided by a mostly independent (at least conceptually) piece of code. There is the scheduler and process manager that deal with executing user code; memory manager that provides programs with the illusion of an isolated, private memory space; there is the networking subsystem, device drivers with their own abstractions of physical devices, and possibly many more.

All of those are outside of scope of this thesis, and e.g. [1] and [2] provide an accessible study material for both theoretical and practical aspects of OS development. Our interests lie in a very specific part of this whole machinery that abstracts away the details of long-term data storage, and provide us with a simple metaphor of *files*. Almost every operating system ever created provides such an abstraction and we will later see how some of them do so.

1.1.2 Microkernel vs. Monolithic

Since the basic services an OS provides have many different parts, a natural question is how to structure those services. Modern computer hardware provides means for an operating system to control and restrict the operations of user programs it supervises. Since the user code should usually not be able to directly communicate with hardware devices, and must only see the resources that belong to it, such code is usually run in what is called a *protected mode*. But at least some parts of the system must run with full privileges, as they need to interact with devices inaccessible to anything but the core of the OS. This fully privileged core of the system is usually a single non-terminating program called *the kernel*.

There are two basic directions an aspiring system architect can take. One is to pack as much of the functionality as possible into the kernel. This has the advantage that every part of the core system can directly access the hardware, leading to a better overall performance. Unfortunately, there are many disadvantages as well. Any bug in a fully privileged code can cause immense damage to the entire system, while anywhere else it could only affect the system in which the bug occurred.

Another problem comes from the fact that the lack of strict module boundaries encourages dependencies on the internal details of largely independent parts. Unless a strict policy is adopted, unrelated modules can quickly become interdependent in ways that could be very difficult to understand or even notice. This further increases the impact a programming mistake can have on the system as a whole.

This arrangement of a system is called *monolithic*, and is the most prominent among contemporary mainstream OSs. The most interesting particular example is probably the Linux kernel. While monolithic in nature, a lot of effort has also been put into splitting the kernel into a lot of separately compilable modules, and as such, it is sometimes not considered a pure monolithic kernel. However, with its millions of lines of code running with full privileges, it still struggles with many problems inherent in monolithic design.

The other path a system design can take is to separate as much of the functionality as possible into isolated userland[1] processes. This has the obvious advantage that any failure — be it a bug or an adversarial attack — is contained to the component in which it occurred by the same means that isolate all user programs from each other. As all the parts need to explicitly communicate via rigidly specified IPC[2] protocols, this also imposes a much stricter requirements on the specification of subsystem interactions, leading to a more understandable and maintainable structure. A significant drawback is that IPC communication comes at a cost, and some consider the performance impact too large to be feasible.

1.1.3 File systems

An abstraction and organization of permanent memory is one of the important responsibilities of every operating system. Most contemporary systems organize data in a structure known as a *file system*. The usual understanding is that a file is a block of data accessible under a name. To impose more structure than a flat naming, most modern file systems also provide special files known as *directories*, which have no contents, but contain references to other files, which are then accessible as being children of the directory. This way, a hierarchical tree-like structure can be used to manage many files. Regular files can have arbitrary length (subject to limitations of the physical storage), and can be created, destroyed, read, written, renamed, etc. They can also have metadata associated with them — for example access/modification times or user permissions.

[1]That is, not running entirely inside the kernel memory space.

[2]Inter-Process Communication.

In practice, there are many ways that can be taken to deal with various specific problems. For example, systems must deal with the fact that there can be multiple physical storage devices, some of which may be inaccessible or privileged, and other that may be replaceable media such as floppy disks, optical disks or plug'n'play flash drives. Another source of trouble is the fact that different operating systems have different expectations about what kind of metadata a file possesses. This is reflected in a myriad of incompatible on-disk formats that are in everyday use.

Many modern systems deal with the variety by providing a piece of software that abstracts away most of the differences and provides a single consistent interface. The implementations differ in the exact mechanics and features, but for the most part they tend to be similar enough to provide a familiar environment. Unfortunately, this often leads to being constrained by past designs. One of the goals of this thesis is therefore to examine several related systems and provide an overview of their functionality, identifying several areas which became of concern only recently and which are not sufficiently addressed in any of the examined systems.

Because *file system* can mean both the operating system interface and a particular on-disk format (or its implementation), a convention is used in this document to call the former *the file system* and the latter *a filesystem*.

1.1.4 Existing systems

The following sections try to introduce several operating systems which are in some ways interesting. The emphasis is put on their interface to files and the way they manage process namespaces. These brief introductions are by no means complete or even sufficient, but I made an effort to highlight the most relevant aspects. Further references are provided for readers who wish to get more familiar with any of the discussed systems.

All of the discussed systems are at least conceptually descendant from UNIX and thus fairly similar in regards to their basic file operations. There are several interesting systems based on capabilities, whose design in one way or another approaches or even solves problems encountered in UNIX-like systems (for example, see [3] or [4]). I have decided not to include them in this document because they are too foreign to provide a meaningful comparison, even though I borrow some concepts from capability based systems.

1.1.4.1 Traditional UNIX / POSIX

Traditional UNIX file system interface, along with its permission model, is still widespread among systems descendant from UNIX, including most Linux distributions, BSD and Darwin. Although incremental improvements have accumulated over the years to reduce the effects of some of the original problems, the basic principles remain largely the same.

The original implementation uses a single global hierarchical namespace for all processes. The namespace has a root directory named simply "/" and all files are descendants of the root. Because only one namespace is present, a mechanism is provided to attach contents of devices (for example, memory media such as floppy disks, or more recently flash drives and DVDs). This mechanism is called *mounting* and makes it possible to access an entire namespace as if its root was just a directory in the global namespace, complete with a proper parent.

In order to limit access to files belonging to different users, a permission model is used that assigns to every file an owner, a group, and a set of permissions for owner, group and everyone else. Those permissions consist of three bits: `READ`, `WRITE` and `EXECUTE`. Same three bits are set for directories, where `READ` allows listing the directory, `WRITE` allows modification (i.e. adding, removing and renaming entries) and `EXECUTE` allows the user to access files descendant from the directory and to use the directory as the current working directory.

Neil Brown has criticized several drawbacks of this model in [5]. As he puts it, the model is both too complex (users tend to fail in managing their files' permissions thanks to the scope of micromanagement, among other things) and too simple (simple things are indeed simple, but complex things are often impossible). Thanks to the limitations imposed on the permission description, the seeming flexibility only makes it difficult to understand the model and, in fact, often jeopardizes security.

The POSIX standard also specifies a programming interface for accessing files, used in virtually every UNIX-like operating system. The interface uses a concept of *file descriptor*, which is a capability-like object that represents an open file. It also has the current position in the file associated with it. Although later extensions make it possible to use descriptors directly as first-class handles, this is rather an afterthought, and the original API is designed to represent files open for reading/writing in a global namespace, where every file can be named by a unique string.

1.1.4.2 GNU Hurd

"The Hurd is the GNU project's replacement for UNIX, a popular operating system kernel." [6]

The GNU project has long aimed to provide a fully functional free software replacement for traditional UNIX. GNU Hurd is an effort to provide a fully functional core to the GNU system, with reliability and extensibility in mind. A decision was made to build the system on top of the Mach microkernel, and because of this, the Hurd is essentially a collection of protocols specifying how to put various parts together, all designed to work with services provided by Mach.

This has led to a file system quite unlike any other UNIX implementation. Multiple user-space processes can implement parts of the protocol, which led to Hurd's file system becoming distributed, with no central VFS server. File handles in Hurd are implemented using *capabilities*[3], which can be used to access files and passed between processes using the Mach's message passing interface.

Because there is no central server, every operation on files must be done relative to an existing handle. Each process holds a handle specifying the root directory, and because this handle is usually inherited (except possibly for `chroot()`ed processes), this leads to a single global namespace, mimicking the traditional UNIX behavior. File systems in Hurd are provided by *translators*. A translator is a user process that implements part of the protocol and provides a file hierarchy to other processes. In order for other processes to be able to access the translator's tree, the root node is attached to another node, using a call `file_set_translator()`. The result is very similar to traditional UNIX mounts, replacing a file in the hierarchy with another file, but the mechanism is different.

Hurd complies with the POSIX standard in resolution of the `..` directory entry (*dot-dot*[4]). Hurd achieves this by providing a `file_reparent()` mechanism, which allows to create a new node, which behaves like a copy of the original, except that its `..` entry resolves to a provided parent. This is also used to implement an equivalent of *bind mounts*, which can mount any file or directory (not just a filesystem root). This is sometimes called a *firm link*, and behaves similar to a hard link, the main difference being that it works across different filesystems.

[3]An unforgeable token that carries authority to work with an object.

[4]A special file name reserved to mean a parent directory.

Walfield and Brinkmann [7] provide a thorough explanation of the Hurd's mechanisms and policies. They also criticize some aspects of the system, e.g. the behavior of dot-dot, whose POSIX semantics are usually detrimental to the user's experience. They also mention the inability to protect user's resources from processes executed by them.

These and some other observed weaknesses can be attributed to the focus on reimplementing UNIX. Keeping in mind that the Hurd project started over 20 years ago, it is natural that some of the original goals may no longer be valid. Strict adherence to an outdated standard that may be unsuitable for most modern systems is one of such goals. However, it is my opinion that the technical solution is valuable in its own right, and can serve as a foundation for a great innovative system, if the directions of the project are reconsidered.

1.1.4.3 Plan 9 from Bell Labs

Plan 9 is unique among operating systems discussed in this document as it is a distributed system, designed to run across many physical machines. While a general overview of the system can be found e.g. in [8], it is the mechanism used to deal with filesystems that is the most interesting and important aspect of the system.

The requirement to reasonably present user's resources uniformly in a heterogeneous environment is elegantly solved by making it possible for any process to export a file hierarchy using a common protocol called 9P. Every process then possesses the ability of combining hierarchies imported from other processes into a private namespace.

This is achieved using a per-process *mount table*, which contains a mapping of mount points (identified as references to filesystem nodes, or *channels*) and their destination nodes (also channels). Multiple directories can be mounted at a single location in the namespace. Their contents is then concatenated to form a union directory. However, the union only applies to a single level of the hierarchy — only the first match is used when walking down the directory tree. This is different from e.g. 4.4BSD-Lite, where an entire hierarchy is overlaid.

This mechanism is further generalized by allowing to mount not just a root of a filesystem, but also an arbitrary file, to arbitrary location. This is called *binding* and is somewhat similar to Linux's *bind mounts*.

In essence, binding and mounting are the same mechanism, the only difference being in the origin of the node that is being bound.

The mechanism itself is exceedingly important aspect of Plan 9, as it is used quite heavily to provide each process with a predictable namespace. For example, there is no $PATH environment variable. Instead, all directories containing binaries usable by the current process are bound to the standard location `/bin`. Similar conventions are used for other resources, e.g. console as `/dev/cons`, graphical display interface, devices providing time, system information, debugging capabilities and many more. The process does not need to know how exactly is each conventional device implemented — it just opens a file with a known name.

This makes it possible to run applications across multiple machines just as simply as it is to run them locally. The relevant devices are just implemented using a remote connection. At the same time, some seemingly dissimilar services are implemented as file servers, a prime example being the graphics server. All communication with the server is done using a file oriented interface and the server essentially just acts as a multiplexer, using files it found in its own namespace to implement commands issued to files it has provided its clients with.

Yet another important aspect of Plan 9 is the way it deals with dot-dot path component. The system processes it lexically, meaning that e.g. the path `/a/b/../c` is always equivalent to `/a/c` (the component preceding dot-dot is lexically removed). This solves a number of problem with processing paths in user shell, which is thoroughly explained in [9].

1.1.4.4 4.4BSD-Lite

This is a UNIX derivative, and a predecessor of modern *BSD systems. As such, most things said about UNIX/POSIX also apply here. However, of significant interest are *union mounts*, which were introduced in this version of BSD.

In classical UNIX, mounting requires an empty directory to be chosen as the mount point. The filesystem then completely replaces the contents of the directory, acting as its subtree. Union mounts make it possible to stack multiple file hierarchies on top of each other to a single overlaid hierarchy. This differs from Plan 9, which only concatenates a single level of the hierarchy. Here, all levels of the directory hierarchy are combined.

This presents new challenges in order to make the combined hierarchy behave in an expected manner. Only the topmost layer of the stack is writable, all the other parts of the union being read-only. The benefits of this arrangement are immediately obvious — attaching a new layer on top of a file system makes it possible to preserve the current state while enabling further modifications (which are stored in a different filesystem). Alternatively, one could combine an inherently read-only hierarchy (say, a CD-ROM image) with a temporary memory-backed storage, allowing transparent modifications without needing to copy an entire hierarchy in advance, consuming both time and memory.

It may happen that a file that is only present in a read-only layer is opened for write. In such case, it is copied to the writable layer and further modified there (*copy-up*). It is also possible to make it look as if a file in a read-only layer has been removed. This is achieved using a *whiteout*, which requires special support from the top-level filesystem. Whiteout simply tells the system that any file by that name is removed, and further layers should not be searched. In case a new directory is created later with the same name, it is marked as *opaque*, meaning that it should not be united with any directories in further layers, effectively still treating them as if they did not exist.

An in-depth description of design and implementation of the union mounts, including information not strictly relevant to this work, can be found in [10].

1.2 Summary

There are many problems with the original UNIX file system design, some of which have been discussed in [5][11][12].

One of those problems is the fact UNIX, as well as many other commonly used operating systems, has a single global namespace. What this means is that, essentially, a pathname uniquely determines a file in the file system. A program can access a file just by virtue of knowing its path, provided it has authority to do so. However, authority in UNIX file system is determined per-file and per-user, in a very limited way already described. Storing access permissions per-file is an error-prone mechanism, which many newcomers to the system fail to understand or use reliably. An example can be seen in [5].

Determining authority per-user, on the other hand, exposes an entirely orthogonal problem. Since user's processes are executed with full authority of the user launching it, that means the user must be completely confident that the programs always act strictly on behalf of the user. This was a reasonable assumption back when UNIX was created. Users were mostly programmers who were creating the programs to be run, and as such, the problem of process "loyalty" was not yet as pressing as it is today. But even then, bugs would happen, and data would be lost. Data the program would never actually need for its proper function.

The inability to only provide processes with the resources they need (principle of least privilege) has become even more limiting with the explosive development of home computing, where users are usually completely oblivious to the technical details of the system; applications are imported in a binary form from an untrusted, even unknown, third party; and the users themselves have little or no education in regards to security, storing their most sensitive data (ranging from bank account information to collections of private photographs) in a profoundly insecure way.

Later versions of UNIX, as well as its later derivatives, provide the `chroot()` operation that can partially solve this problem. It allows to set a different root directory for a running process, making the process believe the new root has no parent. However, this mechanism is so awkward to use that it has been mostly ignored, except for the most security critical areas and few other special purposes.

Apart from these most obvious problems, the UNIX design is also limiting. One of the core principles of UNIX's philosophy was that most resources are accessed as files, reusing a single abstraction to solve possibly many problems. However, it fell short of providing applications with means to expose their own services as files. More specifically, it did not provide any way for endpoint filesystems to be implemented by user-space processes. All filesystems must be implemented in the kernel and it is difficult to even allow ordinary users to utilize those services, even if they have access to the storage. For example, if the user possesses a disk image containing a FAT filesystem, there is no easy way to mount this image into the namespace, without requiring administrative permissions.

This limitation has been quite successfully removed in Plan 9 and GNU Hurd. Plan 9, with its ubiquitous 9P protocol, makes heavy use of user-space processes providing services as exported file hierarchies. GNU Hurd, on the other hand, makes it possible for file owner to set a *translator*, effectively equaling an unprivileged mount mechanism.

However, both systems leave some things to be desired. GNU Hurd suffers from the goal of being POSIX compatible, effectively reducing every technical advantage to mere means of implementing outdated concepts. Plan 9, while not suffering from the desire to be compatible (indeed, even though it can be seen as a direct descendant of UNIX, most of the system has been redesigned and reimplemented from scratch), did not go all the way to isolate processes from each other.

Chapter 2

The proposed design

It seems that UNIX-like systems are becoming increasingly outdated and unprepared for the requirements of modern computing environments. There have been experimental system with completely redesigned file system abstraction from the ground up, but they tend to be very unfamiliar and it is difficult to adapt existing environments to them.

In this chapter, I will attempt to propose a file system design that is suitable for modern systems, providing security from malicious software authors even in face of non-professional users, but retaining enough familiarity so that it is

1. Easy to understand and use for users with prior UNIX experience.
2. Similar enough, so that legacy applications can work with it.

2.1 Requirements

As a step towards designing a better system, let us consider some requirements the implementation should abide by:

1. Each process has its own namespace, independent of each other.
    - This means every process has its own root and freedom to attach more file hierarchies to its namespace, without affecting other processes.

- A path in one process can refer to an entirely different file in another one, or no file at all. It must therefore be easily possible to pass files from one process to another, not just by a name.
- All operations on files (including directories) must be doable on file handles directly (where file handles must be able to refer to unopened files).

2. File handles must behave as capabilities, granting access by possession and being unforgeable.
    - Isolated namespaces are important for security, it would therefore be undesirable if a file which is not a part of the namespace, or received from another process, could be accessed.
3. Files themselves do not have access permissions.
    - Permissions on files are remarkably difficult to use correctly. Instead, access rights should be a property of the reference.
    - If a process is not supposed to be granted access to a file, the file must not be accessible in its namespace — the permissions for a user is then the collection of files their login shell is granted access to when the user logs in.
4. To guard against the *confused deputy problem*, any process should start with access to no files at all. Access to all files must be explicitly granted by the parent (keeping in mind that granting access to a directory implicitly grants access to all its children). Any application that requires access to file its parent cannot access, must do so through authorizing with a dedicated service that has said access.
5. Any process should be able to provide a file hierarchy to other processes.
    - This allows a filesystem to be implemented as an unprivileged user process, only requiring special permissions if a physical device needs to be accessed.

Some systems access distinct filesystems as completely independent hierarchies, while UNIX derivatives provide a single joint tree to every process. If secure file handles can be used for any action on files, this distinction is no longer relevant. Any file handle represents a namespace of its own, and selecting a single one of them to mean a "root" directory is just a matter of interpretation.

However, if there was no way to join multiple hierarchies into one, explicitly managing unnamed handles would be inconvenient in most cases. Especially since the parent-child relationships of files are usually restricted to a single memory device, i.e. a file from one device can not be a parent of a file from a different device. A mechanism is thus needed for joining different filesystem trees into a single structure (which may and may not be a tree).

In UNIX and most of its derivatives, mounting replaces an empty directory with the root of a foreign tree. In GNU Hurd, translators provide a more general mechanism of attaching a hierarchy provided by some service to a file provided by another service. Both of these approaches are unusable because their effect is always global. For every process, the attached filesystem replaces the original node.

Plan 9 took a different approach. Instead of mounts being global, every process has its own namespace and private mount table. In this arrangement, the mount does not influence the target file in any way. Internally, the mount table is a list of bindings between files. When the namespace is navigated, the procedure checks whether the current file matches any in the *from* column of the mount table, and if it does, replaces the reference with a corresponding *to* file. Note that this allows generic binds — the destination need not be a filesystem root.

With this mechanism, an issue arises of how to resolve conflicts between the mounted directory, and a possibly non-empty mount point. Plan 9 deals with this problem using a simplistic variant of union filesystem. When a child is looked up, the bound destinations are looked up in sequence, and the first match is returned. Thus it is possible to bind multiple files to the same location. This version of uniting directories is easy to implement with a simple mount table, but is limiting in that only a single level of hierarchy is overlaid.

These simplified unions are very useful for the task Plan 9 needs them for — single level directories, such as `/bin`, can be combined easily from multiple sources. This has many benefits, such as making the $PATH environment variable unnecessary. There are, however, several scenarios in which this is not enough. For example, it would be beneficial in some cases to provide an application with a set of writable data files, but with read-only defaults as a fallback. With Plan 9, the writable versions would need to be copied beforehand in entirety. This is the basic problem all full-blown unioning file systems were created to solve.

Another slight inconvenience of per-process mount tables is that directories cannot be simply passed to another process. If the directory is handed over, any mounts to its children remain local to the source process, so the receiving process cannot see them. Alternatively, it is not possible to have two versions of a single directory, one with mounts and one without, in a single process.

Yet another problem with simple mount tables is that the "deep" overlaying mentioned above is impossible to implement with them. A directory inside such an overlay does not belong to any of the underlying trees, but instead refers to all of them. One would need that every directory, in a sense, contains its own table of mount points.

2.2 Filesystems

Historically, the specific on-disk data formats have had a large impact on the design and usage of the entire virtual file system. To avoid this idiosyncrasy, I completely disregard the differences of available features of various existing formats.

Every filesystem consists of a single rooted file tree. Internal nodes in the tree are always directories. There is no implicit limitation on the number of entries of any given directory, and every filesystem with writable directories should support UTF-8 encoded file names with length of at least 255 bytes. The filesystem is supposed to be a rooted tree — neither hard links, nor symbolic links are supported. If a filesystem created with another operating system contains hard links, they are interpreted as separate files. If a filesystem contains relative symbolic links, they may be interpreted inside the filesystem, but absolute symbolic links are always treated as broken. It is left to the implementation to decide how to treat broken links — ideally, the file is treated as a read-only explanatory text.

The only metadata a filesystem needs to support is file size. It also needs a distinction between directories and regular files. Additionally, the system can meaningfully use and manage creation, modification and access times, if supported. Named pipes are supported, but they are not required to be implemented. Thus, the system should be perfectly capable of running securely even with the legacy FAT filesystem.

2.3 Files

As with most file systems, a file is the basic named unit of storage. Every file has *metadata*, basic information about the file itself. This includes, among other things, the file's type, access/modify times, size, etc.

Different file types differ in what operations are valid on the file and what they mean. There are three basic file types.

2.3.1 STORAGE files

Files of type `STORAGE` represent the most ordinary kind of file. They are basically the same as regular files and block devices in UNIX terminology. The difference is that here, the type does not specify what the file is, but rather how it is expected to behave.

The `STORAGE` file has a size, and represents an array of records. The records are usually bytes, but not necessarily. Making the interface explicitly byte-specific does not bring any discernible benefits, and it is imaginable that special-purpose filesystems can benefit from this genericity. `read()` and `write()` operations accept a position and a record written at a particular position should be read by a subsequent read at the same position. Thus, a `STORAGE` file represents a real memory area.

This is the only type of file that can be `resize()`d. If a file is resized to a larger size than it was originally, it behaves as if the newly acquired span is filled with zeros. If a `write()` is made at a position beyond the current size, the file is first resized accordingly (just like in POSIX).

2.3.2 STREAM files

Files of type `STREAM` are different from `STORAGE` in that they do not have a well-defined size or persistent records. This kind of file does not need to represent any real memory, and there is no prespecified relation between reads and writes. In UNIX terminology, this can be a pipe, a socket, or a character device (e.g. a console input/output).

The meaning of `read()` and `write()` is subject to a particular filesystem's intentions, and the position argument is meaningless for them.

2.3.3 Directories

Files of type `DIRECTORY` are quite ordinary directories. They do not hold any data, and instead contain named references to other files. Their size is equal to the number of entries they contain. The `write()` call is meaningless for them. Instead, `create()`, `rename()` and `unlink()` are used. The `read()` call works similar to the `STORAGE` type, except that in this case, entire entry names are the records. When more than one entry is returned in a single call, they are separated by binary zeros.

A reference to any descendant of a directory can be accessed by calling the `walk()` method.

2.4 File handles

File handles are a central concept to the design of the programming interface, since several of the requirements mean that a file need not have any name or path from the application's point of view. A handle is thus the only valid way of accessing a file. A handle is a capability — it conveys an authority to act upon a file, and it cannot be forged.

There are only three ways to obtain a handle to a file:

1. A handle referring to the root of a filesystem is given to the process that implements it.
2. A handle can be passed from one process to another (through IPC).
3. A handle can be received by calling methods on another handle.

A range of methods is provided for use with file handles. Some of the terminology has been inspired by the 9P protocol, which is used as the standard file access protocol on Plan 9 systems. The set of methods has changed significantly since the early versions of the implementation. For example, handles were originally designed to use reference counting for management, the `bind()` operation affected and entire namespace (which was explicit), etc. It has turned out that some interfaces are much easier to implement and use than others.

There is one special handle, here called the *nil handle*. The nil handle does not point to any file.

2.4.1 Access permissions

Since handles possess the authority to act upon a file, it is necessary to provide a set of access rights and a way to restrict them if needed. Currently, there are only three flags that determine the handle's authority: `READ`, `APPEND`, and `WRITE`.

Even without any permissions, anyone can query the metadata of any file they have access to. Additionally, if the file is a directory, they can `walk()` it to access children. However, it is not possible to `read()` or `write()` the file, or list/change directory entries.

`READ` is the authority to `read()` the file and access all its information. `WRITE` is the authority to `write()` to an arbitrary position in the file, and `create()`, `move()` and `unlink()` directory entries.

`APPEND` is a special authority. It allows user to `write()` into the file, but only at the end. It is not possible to change any contents already written, or `resize()` the file. For directories, `APPEND` allows creating new entries using `create()`, but not renaming them or removing with `unlink()`. Also, `APPEND`-only directory cannot be a target of the `move()` call, due to security concerns. Note that `APPEND` is a reduced variant of `WRITE`, so `WRITE` implies `APPEND`.

Although currently both regular files and directories use the same set of permissions, it is possible that the permission set will be split to allow separate permissions for directories and their children. So far, however, this has not been implemented.

It may be surprising that an "execution" permission is not included. However, in a decentralized multi-server system, such a permission is completely pointless. Because program loader is a child task like any other, it always and only needs read permission to execute a file. There is no reasonable way to distinguish between reads intended for executing a program and reads intended for anything else.

2.4.2 Methods

This section describes the set of basic operations proposed for file handles. Note that the description here is conceptual. The specific implementation is free to choose how to handle in/out parameters and error signaling.

All methods implicitly return an error/success code. On error, return values/handles are invalid. All methods fail when provided with nil handle as an argument, unless the parameter is marked by a question mark, in which case nil is a valid argument.

Several methods available in the current implementation are not described here. For example, methods that provide debugging features are not included. Similarly, methods for reading and writing metadata are unspecified here.

`Clone(handle?): out_handle`

> The `Clone()` method returns a new handle which points to the same file as the provided handle. If the provided handle is a nil handle, the method succeeds and result is again a nil handle.

`Put(handle?): nothing`

> The `Put()` method destroys a handle. It does not affect the file the handle points to. Its meaning is that it states the handle is no longer used. Using a handle after it has been put is an error. If the handle has been previously used to `Open()` the file for IO, `Put()` is equivalent to the standard `close()` function.

`Walk(handle, name): out_handle`

> `Walk()` returns a new handle pointing to a child named by `name`. If no such child exists, an error is indicated. `Walk()` does not require any permission.

`Bind(handle?, path, bind_handle, flags): out_handle`

> `Bind()` method binds a new descendant to the provided handle. For details, see section 2.5.

`Unbind(handle, path, unbind_handle?, flags): out_handle(?)`

> `Unbind()` method removes a descendant previously bound using the `Bind()` method. For details, see section 2.5.

`Select(handle, index): out_handle`

> `Select()` method is provided as a way to access components of a union. For details, see section 2.5.

`Restrict(handle, permissions): out_handle`

`Restrict()` creates a new handle with reduced authority. The new handle points to the same file, but has limited permissions. The handle's authority can only be reduced, never increased.

`Open(handle, mode): nothing`

`Open()` method marks the `handle` as open for I/O and prepares the underlying file (if appropriate). If the file cannot be read/written or the handle does not posses the required authority, the method fails. Once open, methods `Read()` and `Write()` can be used on the handle. To close it, `Put()` must be called. If a `Clone()` method is called on an open handle, the resulting cloned handle is not open. `mode` can be one of `READ`, `WRITE` and `APPEND`, or any bitwise inclusive sum of those. Calling `Open()` on an already open handle results in adding the specified privileges, if possible.

`Read(handle, position, array): bytes, entries, version`

`Read()` method reads contents of the file at `position` into `array`. Only as many entries as can be stored in `array` are read. On success, `bytes` is returned as the number of bytes written into `array`, `entries` is the number of completely read entries, and `version` is the numerical indicator of changes to the file. Every change to the file changes the version number, including directory changes.

```
Write(handle, position, array):
    bytes, entries, newsize, version
```

`Write()` method writes the contents of `array` to the file at `position`. It is possible that not the entire contents of `array` is written without error. An error is returned only when the write is completely failed. `bytes` is set to the number of bytes successfully used for writing, and `entries` is the number of written entries. `newsize` is the new size in entries of the file, and `version` is the version number the file has immediately after the write.

`Resize(handle, newsize): nothing`

The `Resize()` method changes the size of a file. The file must be open for write. If the new size is less than the current size, the file is truncated. If the new size is greater, the contents beyond the original end of file are implementation-defined.

`Create(handle, name, kind): out_handle`

> `Create()` creates a new file as a child of the file pointed to by `handle`. `kind` can be `DIRECTORY`, `STORAGE` or `STREAM`. The meaning of `STREAM` kind is specific to the backend filesystem, and may be unsupported. `STORAGE` kind usually creates an ordinary file. After successful call, `out_handle` points to the newly created file. Otherwise, nil handle is returned and an error is indicated.

`Unlink(phandle, name, uhandle): out_handle`

> `Unlink()` removes a child identified by `name` from the directory pointed to by `phandle`. Additionally, if `uhandle` is not a nil handle (which it can be), it must point to the file that is to be removed, otherwise the operation fails. After successful call, `out_handle` points to the file that has been removed, and `uhandle` is invalid (as if `Put()` has been called on it). On failure, all handles remain unchanged.

```
Move(source_handle, source_name,
     dest_handle, dest_name, expect_handle?): nothing
```

> `Move()` atomically removes a child of `desc_handle`, and at the same time attaches it to the `source_handle`. Both handles must be in the same filesystem. If the operation cannot be executed in full, neither directory is changed. This should hold even in face of transient failures (e.g. power outage), unless such guarantee is impossible to implement for the given filesystem.

2.5 Bind operation

The endpoint filesystem servers only provide simple file trees. In order to present user (or an application) with a single consistent namespace, an operation is needed that joins multiple independent directories into a single structure. For this purpose, the `bind()` operation has been created.

Compared to the standard `mount()` operation, `bind()` does not affect the constituent directories in any way, so the binding has a purely local effect. This is reflected in the fact that the operation actually creates a new virtual directory, and returns a handle to this directory. It is not possible for the `bind()` operation to affect other processes.

It is also a method for creating union directories. It is possible to bind multiple directories at the same location, or to bind a directory on top of an existing non-empty directory. In such a case, the resulting directory acts as a union of the components, as is explained bellow.

The `bind()` method has four parameters. First is the handle to the file that serves as the *base* for the resulting union. Second is the path at which a new file is bound (*bindpath*). Third is a handle to the file that is to be bound (*target*). The last is one of `TOP`, `BOTTOM` and `REPLACE`.

To understand the operation, it is important to understand that it affects not just the base directory, but the entire tree of which *base* is the root. What it does is that it attaches the target into this tree at bindpath.

There are two possibilities. There may or may not exist a file at this path already. If such a file does not exist, and no prefix of the bindpath is a non-directory file, then the target will act simply as a descendant of base at bindpath. Every previously unknown name along the path is considered to be a directory with part of the path as a single entry.

If the file already exists, and the flag is set to `REPLACE`, then the resulting tree behaves as if the file originally at bindpath has been replaced with the target. In any other case, a union is created, by virtue of multiple files occupying the same path (we will call this as being *stacked*). When a non-directory is stacked with a non-directory, and the flag is set to `TOP`, then the file from target is used. Otherwise, the file from base is used. When two directories are stacked, their entries are concatenated, but any changes are reflected only in the topmost directory in the stack. If a change would require changing more directories in a stack (for example, `unlink()`ing an entry that is present in more layers), the changing operation fails without modifying any file.

Finally, it is possible that a directory is stacked with a non-directory. In such a case, the type of this file is reported as `MIXED` and most operations simply fail unconditionally. In order to use any of the stacked files, the `Select()` method must be used.

`Select()` method expects a union as an argument, along with a numerical index. It then returns a handle to a single layer of the stack, as denoted by the numerical index. The topmost (writable) layer of the union has index `0`, with bottom-most layer having the largest index.

It is possible for the writable layer to not contain a file that is present in one of the lower layers. If such a file is written to, it is transparently copied to the topmost layer, including creation of any ancestor directories. This is called *copy-up*. There is one special case in which base is a nil handle. What happens in such a case is that the nil handle acts as an empty, unwritable directory. Everything else, however, works as usual.

An important part of this mechanism is that paths to the bound targets are considered to be directories for the purpose of returning directory entries and indicating `MIXED` nodes, but until the target file itself is reached, they are not considered to be part of the stack. Thus, if a `TOP` bind with a non-empty bindpath is made, the base is still the topmost layer until the bindpath is walked in its entirety.

As has already been stated, the operation returns a new handle. The original handles passed to the operation are unaffected, and their respective files have no knowledge of any `bind()` having been called. The resulting directory can later be used as an argument to the `unbind()` operation. The operation takes the same path that has been used for the original bind, and returns a new handle that behaves as if the original bind has been undone.

Chapter 3

Implementation

In order to verify and evaluate various decisions made in the design process, it was helpful to start implementing the ideas early on. As a results, many areas of the implementation have been rethought and changed (sometimes significantly). Several instances of such departures from the original intention will be documented further on.

It was necessary to select a suitable platform for this implementation. Thanks to many interesting aspects of the system, HelenOS[13] was selected. Even though this document reflects the latest iteration of the design, it is probable that it is not final and will further change, especially based on feedback from the community. However, I do not expect any significant changes in existing mechanisms to occur.

As a more debatable decision, I have chosen not to implement the server itself using the traditional C programming language. Instead, the Go language has been used. Although it required me to expend significant effort in order to use Go within HelenOS (which is out of scope of this thesis), the safety and expressivity of the language has made the actual prototype implementation easier, allowing me to disregard many problems specific to the C language. It also allowed me to evaluate the benefits and drawbacks associated with implementing critical system components in a higher-level memory-safe programming language.

3.1 HelenOS

HelenOS is a relatively young open-source system which began as a student project of Jakub Jermář on Faculty of Mathematics and Physics at Charles University in Prague. It has since evolved into fairly usable, yet still simple and incomplete system. It is a friendly platform for exploring new ideas, and prides itself on being the most portable and modular micro-kernel multi-server operating system currently being developed. Being developed mostly through student projects and theses, the purpose of the system is not to serve as an alternative to mainstream operating systems, but rather to be a good platform for implementing new ideas and experimenting with them, which has made it perfectly suitable as a platform for this very thesis.

The core of the system is formed by a microkernel called *Spartan*, which provides only the very basic services such as task management (task being the resident name for processes), thread scheduling and synchronization, virtual memory management, access to hardware, and finally, inter-task communication (commonly known as *IPC*). Most of the abstractions that are usually the kernel's responsibility are implemented by separate isolated tasks, some of which are provided with extra privileges by the kernel, in order to access hardware resources.

3.2 IPC primitives

Contrary to the past development in the area, which has come to favor synchronous communication, recent Spartan versions only support asynchronous IPC natively. The terminology of the system uses a phone call metaphor, with a phone on one end and an answerbox on the other end. In practice, the initiating task creates a phone connected to the destination task's answerbox. It can then send arbitrary number of calls. The initiating task can then continue with other work or wait for the reply. The calls are stored in the destination answerbox until the task retrieves them. The receiving task will process the call and send an answer, which is again stored in the initiating task's answerbox.

Normally, such a complicated protocol would create an undue burden on the programmer, requiring a complex body of callbacks in all places IPC is used. To eliminate this problem, there exist two mechanisms in the C library. The first is a concept of fibrils. A fibril is a unit of execution that is scheduled cooperatively in userspace, with no intervention or even knowledge from the kernel. Fibrils simply run in their thread's context and yield to other fibrils when they need to block for some reason. Thus, there is M:N relationship between fibrils and thread. There can be arbitrary number of fibrils with their own stacks and contexts, but usually only a single thread is created, and it is never useful to create much more threads than there are processors (interestingly, the Go programming language uses the exact same concept, except that it is called Goroutines instead of fibrils).

The other piece of software that provides an efficient abstraction of the IPC mechanisms is called the asynchronous framework. Because the kernel does not understand the concept of fibrils, there is no natural way to route IPC calls to them. The job of the async framework is to solve this problem by employing dedicated manager fibrils. Manager fibrils are the only ones that access the task's answerbox, and it is their responsibility to route the calls to their correct destination, be it a newly spawned handler or a fibril waiting for answers. It is the async framework that manages all the complex magic involved in asynchronous communication. It remembers which fibril waits for which message and takes care of registering relevant callbacks.

3.3 Advanced IPC mechanisms

Simple calls explained above would by themselves not be sufficient to efficiently handle the requirements of real inter-task communication. For this reason, there are several special kernel-known IPC methods which encode a task's intention to

- create a new connection
- copy a larger block of memory
  (the standard call only allows sending several words of data)
- sharing a segment of memory
- negotiate a change of data maintained by a third party

These methods are predefined and kernel is aware of their meaning. When the tasks successfully negotiate one of the above actions, kernel automatically executes it. Additionally, calls can be forwarded to another task, and as such it is possible to pass e.g. a data copy request across several tasks, with only the endpoints participating in the actual copy.

Of much interest to us is that the last mentioned action, negotiating a third-party change, can be used to implement passing opaque and unforgeable handles (maintained by a third-party server) between tasks. Indeed, that is the exact use case for which the mechanism has been designed. Note that the resulting mechanism can behave very much like the generic concept of capabilities, except that neither the kernel, nor any other task, need to enforce security. The server, as the only party that has the means to execute a method on any handle, has all the authority it needs to enforce secure access to resources it manages.

3.4 Original file system implementation

Similar to most of the system, HelenOS's file system support is rather spartan. It has a single central VFS server which acts as a switchboard for filesystem implementations exposing specific storage devices. Thus, most of the logic is encapsulated in a service specific to a filesystem type, and every such service announces its availability to the VFS server, which in turn makes the services available to all clients in the system.

The entire file subsystem is composed of three basic parts:

1. the VFS server itself

2. file system backends (*endpoints*) aided by the libfs library

3. client-side support implemented in the common C library

Being as it is now, HelenOS has no concept of users or user-level protection. As such, all processes in the system share a single global namespace and there is no support for file permissions of any kind — every process has unlimited access to any file. Additional filesystems are mounted into this global namespace in a manner mostly identical to that of traditional UNIX — filesystem root simply replaces an empty directory. It is also possible for a single instance of a filesystem driver to handle translation of many devices or for every device to have a dedicated driver. VFS server itself keeps tabs on all the drivers and manages them according to user's wishes.

3.5 New file system implementation

The separation into parts has been kept essentially the same. While it was suggested that it might be interesting to spawn a server per namespace, distributing the subsystem, I have not pursued this direction because of the inherent difficulties associated with completely decentralizing the core support. Interestingly, having VFS server explicitly handle namespaces has recently proven to be unnecessary and even restrictive.

The endpoint filesystems have been purposefully changed to the least extent possible. But because filesystems are no longer being registered globally with the VFS server (one of the requirement being that any task should be able to provide a file hierarchy without interfering with global resources) some changes were necessary. Most of the modifications, however, were limited to the supporting libfs library.

As filesystems are no longer registered with a global service, it was necessary to create a way to manage them on a per-instance basis. In order to do so, a communication channel needs to be established between the spawning task and the filesystem task. Standard IPC has shown to be very inflexible for this purpose. Without further extensions to the API, it is not possible for the filesystem task to be a server to another protocol besides the FS protocol it implements, and creating IPC connection in the other direction would require the parent to be a server, which would counterintuitively cause several other problems. Instead, implementing and using virtual pipes as the control connection was found to be the best solution, given the currently available interfaces.

However, for receiving handles to devices for mounting, and conversely, sending back handles to the root directory, pipes normally cannot be used. To work around this problem, two VFS methods have been added to allow sending file handles across pipes. Although this solution seems somewhat forced, it works well and so far no better solution has been invented. As to how the control pipe is handed to the spawned task, seeing how a filesystem driver has by itself no use for a file system, it would be efficient to simply hand over the pipe at the task's root "directory". However, the actual implementation uses a more generic *inbox* mechanism, which is explained in the next section.

Besides the server-side changes, the client-side support from the C library also required some changes. The original design simply built the VFS IPC interface to reflect standard POSIX file descriptors and methods. Since the redesigned VFS server does not follow POSIX conventions, there is no longer a one-to-one correspondence between available methods and standard POSIX functions. To address this, an entirely new set of functions has been introduced. Most of the functions directly reflect the methods provided by the server, and several provide higher-level functionality, but they are not intended to be used by end-user programs. Instead, these functions are used to implement standard interfaces such as the Standard IO Library, which is supposed to be used in most cases. A thorough breakdown of available functions in provided as appendix A.

3.6 Inbox

There are several parts of the library that do not reflect any interface but instead implement a mechanisms of their own. One such part is the *inbox*. Most traditional systems have three special predefined files that are present in every process. Those are the standard input, the standard output and the standard error output. On POSIX-compatible systems, these files even have fixed predefined file descriptor numbers of 0, 1, and 2, respectively. Additionally, POSIX allows the parent to specify a file to be opened for a particular (arbitrary) descriptor number.

Here, inbox instead contains a map with arbitrary strings as keys and file handles as values. The parent can set these named files in inbox while setting up a new task to run. A number of predefined names exist to accommodate for standard streams and other common files.

`__stdin`, `__stdout`, `__stderr`
: The standard streams.

`__root`
: The root directory.

`__elf`
: The program binary that is running (mainly for debugging).

Applications can use any names not prefixed by two underscores, which are reserved for use by system libraries. It is intended for shell to provide means to easily put files in inbox, allowing to pass programs their required data without giving them access to any part of user's namespace. Note that sometimes it is not desirable to bind the files into the task's own private namespace, which would otherwise also be suitable.

3.7 Server

That leaves us with the centerpiece of the entire subsystem — the server itself. The entire server is rather small, consisting of about 3600 lines of code, about 500 of which could be automatically generated (IPC boilerplate). The largest part is the support for union directories, accounting for almost 1000 lines. As has already been stated, the entire server is written in the Go programming language.

The code is separated in several files and an attempt was made to make the implementation as modular as possible, making it easy to understand its structure and extend it. The core part takes care of registering the server with the system and handling IPC messages.

`vfs.go` contains the basic initialization and part of the low-level IPC handlers, while `client_ipc.go` holds the bulk of method handlers. The handlers just decode parameters, call the appropriate methods and return answers. Most of the code in this file is highly repetitive and could be replaced with declarative description, for automatic code generation. However, no such tool has yet been designed for HelenOS's IPC.

The methods doing the actual work are present in the `client.go` file, and match the names of methods in `client_ipc.go`. The naming scheme is that the real methods are named the same way their IPC protocol counterparts are (except that they do not possess the `VFS_` prefix, and are written in CamelCase, as per Go language conventions). IPC handlers for each method are named the same, with the addition of a `Call` suffix.

Thus, for example, the IPC handler for the `VFS_READ` method would be named `ReadCall()`, and the method called by this handler with proper decoded arguments would be simply `Read()`.

Both are methods of the `ClientContext` type, which contains all information specific to a particular client. Note that the management of per-client data (in this case, pointer to the `ClientContext` object) is part of the native async framework and is therefore not a concern for the server implementation. Bindings for the async framework API are part of the Go language port to the HelenOS system.

Each client has its numerical ID (exclusively used for debugging), a channel for incoming file handles from other clients, and most importantly, a private table of live handles, which is protected by a mutex. The table is a simple array of references to *nodes*, which is the server's internal representation of files. Each reference also stores the mode in which that particular handle has been opened for IO, if it has been. The client-side file handles are simply indices to this private array, and in order to access any file, its node needs to be added to the client's table. Since all IPC methods only work with these numerical handles, it is not possible to access any other files (except by circumventing the server altogether).

The nodes themselves are implemented as instances of the `Node` interface (`node.go`). There can be (and are) multiple types implementing this interface, making it possible to have several backends directly in the server itself. This is used in the server to implement pseudo-files that cannot or should not be implemented by endpoint filesystem drivers.

The most important implementation is the `FSNode` type (`node_fs.go`). It represents a file provided by an endpoint file server (simply called *endpoint*). Each node holds a reference to its backing `FileSystem` object, which represents a connection to the endpoint and provides methods to communicate with it. It also holds the numerical index of the represented file. Methods of the `FSNode` type are implemented by communicating with the endpoint server. Each `FSNode` also contains a read-write lock, which protects all operations on the file, which allows the endpoint to have multi-threaded design without having to deal with synchronization in most cases. This is supported by the requirement (enforced by the VFS server) that every file is represented by at most one `FSNode`.

There is no global table of connected filesystems, since it is not necessary. By registering with the server, the client providing the filesystem receives a handle to the root node. Once all `FSNodes` belonging to this filesystem cease to exist, the callback connection is terminated. On the other hand, if the connection is terminated unexpectedly, for example by the endpoint crashing, the nodes will simply respond with error to any method call. However, there is no built-in way to detach a filesystem which is still in use and running — if such an action is required (e.g. to forcefully unmount a physical drive without risking data loss) this must be done through the control connection of the endpoint server in question, if such exists.

3.7.1 Unions

Another type of nodes, besides `FSNode`, is the `UnionNode` (`node_union.go`). As the largest part of the server, this node type implements stacked files. Because file handles in this VFS design correspond directly to unopened files and all operations are relative, it was necessary to create pseudo-files that reflect the entire structure of a (sub-)namespace. In this manner, `UnionNode` not only contains references to all files it immediately unites, but also any unresolved binds that lexically belong to the subtree defined by the node.

The important part is that all components of the union, including the parts not yet reached, must honor the semantic defined by the specification. This is achieved by designing the node as an ordered list of all its components, in order of binding. Every time a new file is bound, it is prepended to the list of components as a tuple of relative bind path and the target node. On every walk action, any nonempty bind paths are walked if matching (or removed). Empty bind paths denote nodes that are already part of the union. A special case is an opaque bind, which, when resolved, removes all the binds that follow it in the list. When the nodes need to be accounted for in order, the list is first traversed from top to bottom, collecting `TOP` binds, then it is traversed in reverse, collecting `BOTTOM` binds. This ensures that opaque binds work as intended.

This implementation is fairly simple, and it is easy to see that it works as expected. The only drawback is that it is fairly inefficient when many files are individually bound at the same location with the same long path. A more efficient representation (in both performance and memory use) is possible in such case, but simplicity and obvious correctness of the current implementation would be jeopardized by such optimizations. It is possible to avoid this inefficiency by first constructing a virtual directory immediately containing binds as children, and then binding the resulting file at the long path.

3.7.2 Restricted nodes

The server provides means to create a restricted version of a handle already possessed by the client. This handle only gives its holder limited authority. For example, it is possible to restrict a handle such that the result is only usable for appending data, and every other action fails.

Although this restriction is conceptually a property of the handle, it must not be possible to lift this restriction by using the node in a client-agnostic context. For example, if the handle is used as a bind target, every handle that results by walking the resulting union must be restricted the same way. For this reason, the restriction may not be represented simply in the client-local handle table. Instead, a special very simple node type exists for this purpose.

This type is called `RestrictNode` (`node_restrict.go`). It is very simple, and wraps any other node object. It overrides several methods (specifically, `CREATE`, `OPEN`, `UNLINK`, `RESIZE`, and `MODE`) to do an additional permission check against a modifier stored in the wrapper. The `RESTRICT` method simply wraps the node in this wrapper, and since there is no way to "unwrap" the node, it is not possible to circumvent this restriction when only holding such wrapped node, regardless of the context.

3.7.3 Virtual pipes

Finally, the server provides a special node type that implements virtual pipes. Any pipe created by the `MKPIPE` method is an instance of `PipeNode` (`node_pipe.go`). Also, `FSNodes` that represent named pipes internally use `PipeNode` for IO, instead of passing data through the endpoint server. However, this is only done when the endpoint explicitly requests the server to do so. Stream files other than ordinary pipes are still implemented by the endpoint itself.

3.7.4 Unfinished work

There are several problems that have not been satisfiably resolved, and several intended features that I have not been able to implement, yet.

One of the unresolved problems is directory unlinking. In traditional UNIX systems, directories are removed recursively, and only an empty directory can be unlinked. This works well as long as you expect every file to have a well-defined path inside a file system. However, with entire directories being accessible as opaque handles, instead of names, it is natural to expect that any any process already working with a directory should retain it even if it is no longer part of the filesystem's root namespace. If this expectation is to be uphold, a problem arises of how to deal with garbage collection, and how to implement this correctly in every filesystem.

Also related to unlinking, but in a different area, is removing read-only files in writable unions. In the prior description, this has been stated to be impossible, but in a writable stacked directory, it is often useful to allow this. In existing implementations of stacked filesystems, a concept called *whiteout* is used. In its most sophisticated variant, it requires the writable layer to support special attributes for directory entries. Some entries can then be marked as explicitly "removed", instructing the system to hide that entry in any further layer. Whether and how to support this in HelenOS is an open question.

Then there are several technical challenges. One of the potentially very useful, but yet unsupported features, is explicit file locking. So far, file is locked only implicitly on some operations. For some applications, it would be very convenient to be able to synchonize work on a file explicitly, across multiple calls.

Another important feature is an ability to map files into the virtual address space. Among other things, this can be used to efficiently and easily share the program binaries and libraries, reducing memory consumption. However, in order to implement this in a multi-server system such as HelenOS, a support from the kernel is necessary, and this support is not present at the moment.

Perhaps the most important issue is the way various other parts of the system work with IPC. Specifically, there are global services for registering IPC servers, which makes it difficult to enable per-user services without reworking these parts of the system first. Also, currently every task has access to every registered service, making it conceptually possible to bypass the VFS server in access to physical storage. Not only that, but also some other mechanisms can be used unhindered by any task, regardless of relationships (for example, `kill()` call). This must be addressed before any security can be considered.

Last but not least, while the VFS server to endpoint protocol has been left almost unchanged during this work, for the sake of limiting the effect of changes on specific filesystem implementations, some changes in this protocol will eventually need to be made to enable new features and remove no longer needed logic.

3.8 User shell changes

As with any change whose goal is to improve an important part of a system, care must be taken to make sure other component work with it correctly and utilize the full potential. In this case, the structure of the system itself must account for the new mechanisms, otherwise the implementation itself would become almost meaningless.

Without further experience with running the system, it is difficult to speak about specific changes to the environment, since every change will be tested and eventually improved based on that experience. However, I will try to present several concepts that can serve as a foundation for future work.

Most obviously, several commands for mounting and binding filesystems need to be implemented. Not so obviously, since the management of available filesystem drivers has been removed from the VFS server, a framework for this use needs to be designed and implemented.

3.8.1 Applications and Profiles

One of the original goals of this work was to provide a mechanism that could "mimic" traditional user accounts (at least in the area of data protection) without the inherently limiting concept of per-file access control. Another, closely related goal, was to allow a per-application, per-task restrictions on accessible files.

In most situations, the system should work as follows: The most privileged parts of the system see all the files accessible to it. Let us call this collection of files the *root namespace*. When a user logs in, the user's shell that acts on behalf of users direct input has access to the *user namespace*, which is a strict subset of the root namespace. Then, every application executed by said user has its own *application namespace*, which is composed of a strict subset of the user namespace.

Application namespace itself has two parts. One is the resources every instance of the application implicitly acquires. This can be configuration files, external resource files, everything the application needs to function. Additionally, a task as a specific instance of the application can be granted access to additional files. For example, a document for a document reader, an image, a movie, etc. Application should not have implicit access to all the files it can work with, but instead, the access should be granted by the shell in response to user's request.

Given the discussed capability-like nature of handles-as-designed, it is easy to restrict any task to a limited set of files. In the extreme case, task can have access to no files at all. Therefore the question is, how to provide the access to files that the task is supposed to access. The simplest answer to this question is text-based configuration. By enumerating all the resources a user/application/service needs in a text file, the login program and application execution are suddenly reduced to interpreting text files, making them extremely simple. For each agent, a text file would simply define the relationship between the parent namespace and the child namespace.

Of course, there are programs that do not usually need a namespace. Most command-line interface utilities only access files explicitly mentioned in the command line, and can be regarded as shell subroutines rather than full-blown applications. It would therefore be beneficial to distinguish the two types of programs, separately working with *commands* and *applications*. Commands do not have a namespace, instead they share the namespace of the parent shell. They do not have any data of their own, and they are accessible from within applications (for example, using the `system()` C-language call). On the other hand, applications have their own namespace, populated only with application-specific data and shared files, such as command binaries. Of course, commands can be complex programs and should not always be allowed access to the namespace, if it can be avoided. However, it should be very easy to enforce such restrictions using shell script wrappers.

By properly designing a directory structure, it is possible (and easy) to have a version of everything at every level of the hierarchy. For example, the root namespace can have a default configuration for every application available to all users, and a set of commands available to all users and application. Every user can have a configuration for every application (including possibly private applications), and a set of commands private to the user and his applications. Finally, every application can have its own private commands in addition to that provided by the system and the user. Thus, the system follows a kind of a recursive structure.

Chapter 4

Conclusion

There have been several goals for this thesis. One was to evaluate and improve upon file system abstraction implementations derived from the traditional UNIX file system. I have explained that there are serious flaws in UNIX-like systems, and that although some more recent systems expended significant effort to improve technical realization of the file system, none of them reflect the security requirements of now ubiquitous consumer-grade systems on the file system level.

I have proposed a way to address the perceived deficiencies by redesigning the programming interface for accessing files and introducing several new operations. The changes to the already existing operations reflect the move from traditional file descriptors towards the concept of file handles as nameless "pseudo-capabilities"[1]. Thanks to this change, files do not have any implicit name, and any naming is a local information. It is natural to set one such handle as the root directory, and doing so implicitly creates process-specific namespaces, allowing every process to have a distinct set of accessible files.

The `bind()` operation has been proposed, as a way to combine multiple filesystem hierarchies into a single tree, but without making any globally visible changes to the constituent trees. This operation is presented not only as a replacement for the unsuitable UNIX `mount()` operation, but also as a tool for constructing and customizing user and application namespaces. The combination of the proposed concepts then allows a natural implementation of the principle of the least privilege, simply by providing a text-based manifest of files available to the user/application.

[1]The underlying system does not need to support the general concept of capabilities.

Based on this proposed design, I then created a prototype implementation of the concept. Apart from demonstrating the practical usefulness of the proposal, the implementation also serves as an experimental attempt at using a high-level language for implementing a critical system component. Although it was not a focus of this thesis, the implementation has shown both benefits and drawbacks of using the Go programming language in the context of core system programming, and shows that although replacing C in this area comes at a cost, this cost may be small enough to be offset by the gains in terms of reliability and maintainability.

For the purposes of the stated goals of this work, I have achieved everything I intended to. However, the work is very far from being finished. While the basic protocol and the server are present, only rudimentary support has so far been added to the user shell itself. In order to fully utilize the implemented mechanisms, it will be necessary to implement proper tools and policies.

Because the proposed framework removes certain useful possibilities available in the traditional UNIX system, such as controlled file sharing among users, it is also likely that additional services will be created to provide the missing functionality. Although the need for additional services in this case may seem like a setback, it has an objective benefit in that it is possible to provide much more features without cluttering core subsystem with rarely used features.

Eventually, after HelenOS's flaws in inter-process isolation are fixed, the implemented services can be used to provide full-featured and secure user accounts, and also a secure environment to run every application in its own limited "sandbox", both using the same mechanism. This will of course mean a lot more experimenting with various ways to achieve this, but the basic tools have been introduced, so it will probably not take long until the first proofs-of-concept are released.

Bibliography

[1] TANENBAUM, Andrew S.; WOODHULL, Albert S. *Operating systems: design and implementation.* 3rd ed. Upper Saddle River, N.J.; Pearson Prentice Hall, c2006, xvii, 1054 s. ISBN 0131429388.

[2] TANENBAUM, Andrew S. *Modern operating systems.* Englewood Cliffs, NJ: Prentice-Hall, 1992.

[3] SHAPIRO, Jonathan S.; SMITH, Jonathan M.; FARBER, David J. *EROS: a fast capability system.* ACM, 1999. Available at `http://eros-os.org`

[4] *GENODE: Operating System Framework* [online]. 2013 [cit. 2013-05-11]. Available at `http://genode.org/`

[5] BROWN, Neil. *Ghosts of Unix past, part 3: Unfixable designs.* In: LWN.net [online]. 2010-11-16 [cit. 2013-05-11]. Available at `http://lwn.net/Articles/414618/`

[6] *What Is the GNU Hurd?* [online]. 2013 [cit. 2013-05-11]. Available at `http://gnu.org/software/hurd/hurd/what_is_the_gnu_hurd.html`

[7] WALFIELD, Neal H.; BRINKMANN, Marcus. *A critique of the GNU Hurd multi-server operating system.* ACM SIGOPS Operating Systems Review, 2007, 41.4: 30-39. Available at `http://gnu.org/software/hurd/hurd/critique.html`

[8] PIKE, Rob, et al. *Plan 9 From Bell Labs.* In: Proceedings of the summer 1990 UKUUG Conference. 1990. p. 1-9. Available at `http://plan9.bell-labs.com/sys/doc/`

[9] PIKE, Rob. *Lexical file names in Plan 9 or getting dot-dot right.* In: 2000 USENIX Annual Technical Conference. 2000. Available at `http://plan9.bell-labs.com/sys/doc/`

[10] PENDRY, Jan-Simon; SEQUENT, U. K.; MCKUSICK, Marshall Kirk. *Union mounts in 4.4 BSD-lite.* AUUGN, 1997, 1.

[11] BROWN, Neil. *Ghosts of Unix past, part 2: Conflated designs.* In: LWN.net [online]. 2010-11-04 [cit. 2013-05-11]. Available at `http://lwn.net/Articles/412131/`

[12] BROWN, Neil. *Ghosts of Unix past, part 4: High-maintenance designs.* In: LWN.net [online]. 2010-11-23 [cit. 2013-05-11]. Available at `http://lwn.net/Articles/416494/`

[13] HelenOS project. Documentation. Available at `http://helenos.org/documentation`

Appendix A

Provided C-language functions

In order to allow other programs access to services provided by the VFS server, a number of functions is provided in the core C library. Because the semantics of the operations do not match any standard API, most functions are modelled to match the IPC interface directly. Other file-access libraries, such as standard POSIX functions or the `<stdio.h>` library, are implemented by calling these functions.

Apart from programs that need to use operations specific to the system (`bind` shell command, for example), most applications should use the standard IO library to access files. The POSIX API was previously the basic interface to VFS server's functionality, but now that it is just another wrapper for a lower-level interface, it is intended that its use is replaced with the standard IO library and the functions will be eventually moved to the POSIX compatibility library (libposix).

A.1 Conventions

There are two groups of functions — one group models the IPC calls made to the server, the other group implements additional functionality (such as path lookup relative to the current working directory, which is represented entirely in libc).

Functions of the first group are prefixed with `file_`. The others are prefixed with `path_`. Functions of the first category follow a rigid form. Except for a few exceptions, they return an integer return value which is purely used for success/error code. Functions that return a new handle do so by means of the last parameter, which is passed as a variable address. The first parameter is always the handle on which the operation is done, if applicable.

A.2 Header files

There are several header files, each containing prototypes for a part of the API. They are all in the `vfs` directory.

`<vfs/abi.h>`	Contains all the constants and types used in the code. May be moved to a more conventional location.
`<vfs/file.h>`	Mostly functions that map to the IPC method calls.
`<vfs/path.h>`	Wrapping functions that use path parameters instead of handles.
`<vfs/dir.h>`	Functions for reading directory entries (similar to, but more convenient than `<dirent.h>`).
`<vfs/inbox.h>`	Implementation of the inbox mechanism, as explained in section 3.6.

A.3 Function listing

A.3.1 `<vfs/file.h>`

```
file_t file_clone(file_t file);
int    file_put(file_t file);

int file_walk(file_t base, const char *path, file_t *result);
int file_bind(file_t base, file_t file, const char *path,
        file_bind_flags_t flags, file_t *result);
int file_unbind(file_t base, file_t file, const char *path,
        file_unbind_flags_t flags, file_t *result);
int file_select(file_t base, int index, file_t *result);
int file_restrict(file_t base, file_mode_t mode,
        file_t *result);

int file_open(file_t file, file_mode_t mode);
int file_read(file_t file, uint64_t *entry, uint8_t *buffer,
        size_t *bytes, unsigned *ver);
int file_read_all(file_t file, uint64_t *entry,
        uint8_t *buffer, size_t *bytes);
int file_contents(file_t file, uint8_t **buffer,
        size_t *bytes);

int file_write(file_t file, uint64_t *entry,
        const uint8_t *buffer, size_t *bytes);
int file_write_all(file_t file, uint64_t *entry,
        const uint8_t *buffer, size_t *bytes);
int file_resize(file_t file, uint64_t entries);
int file_sync(file_t file);

int file_create(file_t parent, const char *name,
        file_type_t type, file_t *result);
int file_unlink(file_t parent, const char *name, file_t expect);
int file_move(file_t old_parent, const char *old_name,
        file_t new_parent, const char *new_name);

int file_mkpipe(file_t *result);

int file_stat(file_t file, file_info_t *info);
int file_attr_get(file_t file, file_attr_t attr,
        int64_t *result);
int file_attr_set(file_t file, file_attr_t attr, int64_t val);
int file_debug_string(file_t file, char *buffer,
        size_t buffer_size);

/* Create an IPC connection to a service (/dev filesystem). */
int file_session(file_t file, exch_mgmt_t mgmt,
        async_sess_t **session);

/* Registration for endpoints. */
void file_fs_set_handler(async_client_conn_t receiver);
int file_register(file_system_info_t *info, void *userdata,
        file_t *result);

int file_pass_handle(file_t file, async_exch_t *exch);
int file_receive_handle(file_t *result);
```

A.3.2 <vfs/path.h>

```
typedef enum path_lookup_flags {
        PATH_LOOKUP_CREATE_FILE   = 1,
        PATH_LOOKUP_CREATE_DIR    = 2,
        PATH_LOOKUP_MUST_CREATE   = 4,
        PATH_LOOKUP_CREATE_PARENT = 8,
} path_lookup_flags_t;

/* Creates a canonical absolute path
 * from base and relative path.
 */
char *path_absolutize(const char *base, const char *path,
        size_t *retlen);

/* Utility functions for working with paths. */
char *path_split_base(char *path);
char *path_fragment(char **path);

/* CWD management. */
const char *path_get_current(size_t *retlen);
int path_set_current(const char *path);

/* Task's local root directory. */
file_t path_root(void);
void path_root_set(file_t file);

/* Path-based lookup function for files. Relative to CWD.
 * Can create a new file, including directories.
 */
int path_lookup(const char *path,
        path_lookup_flags_t flags, file_t *result);

/* Path-based bind function.
 * Binds to the current root directory.
 */
int path_bind(const char *path, file_t file,
        file_bind_flags_t flags);
```

A.3.3 `<vfs/dir.h>`

```
typedef struct dir *dir_t;

int dir_open(file_t file, dir_t *d);
int dir_read(dir_t d, const char **entry);
int dir_rewind(dir_t d);
int dir_close(dir_t d);
```

A.3.4 `<vfs/inbox.h>`

```
enum {
        INBOX_MAX_ENTRIES = 256,
};

file_t file_inbox_set(const char *name, file_t file);
file_t file_inbox_get(const char *name);

int file_inbox_list(const char **names, int capacity);
```

Appendix B

The Go programming language and libraries

As has been stated in chapter 3, Go has been chosen as the implementation language. Given the prominence of C in the area of system programming, this may be somewhat surprising. However, I believe that in order to create reliable systems, C must be abandoned. C is dangerous. There are many more hard-to-discover problems in C programs than in any other widely used language. To name a few

- The results of integer operation are undefined when an overflow would normally occur. In name of performance, these pathological cases are left undetected.
- Manual memory management and pointer arithmetic makes it extremely easy to access invalid memory regions. Newcomers to the language spend months or years understanding the mechanisms. Invalid pointers can be easily left behind after an object has been destroyed.
- No boundary checks on arrays. Buffer overflows are among the most common sources of security weaknesses in C programs.
- No implicit memory initialization.

Apart from outright reliability problems, maintainability of C programs is also questionable. There is no support for modules or namespaces. No methods. Some parts of the syntax have long been known to be badly designed (for example, type definitions). Preprocessing directives are badly integrated into the language and make any conditional compilation difficult to understand.

To sum it up, C is a good fit for small pieces of very performance-sensitive code. In a subsystem bounded by IPC overhead and hard drive accesses, raw CPU performance is probably not an issue, and more care should be taken to use reliable tools. Go language provides garbage-collected environment with improved syntax, well-defined overflow conditions, methods, interfaces, native support for concurrency, arrays with bounds-checking and most importantly, it is impossible to produce memory faults in pure Go code, since there is no pointer arithmetic (which is unnecessary in ordinary programs).

All information and resources about the language are located online at `http://golang.org/`.

B.1 Porting the runtime

Since both implementations of the language are targeted at systems compatible with POSIX, it was necessary to first port the runtime library to HelenOS. There are currently two production-quality implementations — the `gc` compiler based on Plan 9 compiler toolchain, and a `gcc` frontend called `gccgo`.

Both compile the source to the native binary code, and each has a compiler-specific runtime library implementation. Because of the limited number of platforms supported by the `gc` implementation, I chose to try porting the `gccgo` runtime. This is also because the `gcc` toolchain is already integrated into the HelenOS build scripts. The largest problem present was the amount of platform-specific code in the runtime.

The first area that was necessary to replace was the goroutine implementation. Goroutines are a concept similar to threads, except that they are scheduled internally by the library in the context of much fewer real threads. This means Go uses a M:N threading model, and the scheduler implementation is quite complex and tightly related to the way Linux system calls are called. Interestingly, HelenOS already has an exact same concept of *fibrils*, which can be used as a near drop-in replacement for the original implementation, allowing Go programs to call libc functions (some of which internally depend on the fibril implementation) from Go code. The original implementation uses platform-specific hacks to mimic asynchronous execution for blocking system calls. Since HelenOS IPC is inherently asynchronous, it is unnecessary to make any such provisions.

Another platform-dependent part of the runtime is the memory heap implementation, which depends on the ability to allocate a huge span of virtual address space. When I started working on the port, HelenOS did not support allocating spans of address space without committing physical memory. Due to this, it was necessary to make the heap segmented, making it more complicated and slower. Recently, support for late-reserved memory areas was introduced to HelenOS, which means it was possible to revert the implementation to its original form.

Unfortunately, there are still several parts of the runtime that I have not had time to implement properly. For example, exception handling, or function calls through reflection. Since the missing parts correspond to functionality not necessary for most programs, it was still possible to implement VFS server with it, which was the main goal. This single-sightedness is a reason for ignoring several other problems until the thesis is complete, which means the port is not yet ready to be widely used.

Apart from the runtime library itself, the Go standard library contains a lot of mostly platform-agnostic packages. The packages using system calls ("os" and "net") were partly reimplemented to use correct calls, but otherwise, most of the libraries work completely unchanged.

B.2 Fitness for the purpose

Objectively, Go is a superior implementation *language* thanks to its safety guarantees and features not present in C, but there are several concerns that need to be accounted for. It is a young language, with only two different implementations, the main one only supporting three most common processor architectures. This is a big difference from C, which can be cross-compiled to virtually any platform.

Another problem is that, being as young as it is, the resulting binaries are often inefficient. When speaking about raw performance and low-level optimization, this problem does not affect the `gccgo` implementation so much, but it is still very inefficient in its management of memory objects. Go does not explicitly discern heap memory from stack memory, so allocation is subject to pointer escape analysis. Without advanced cross-function analysis, many cases in which a C program would use stack allocations are translated as heap allocations, increasing the pressure on garbage collector. An example of this is the server itself, as it allocates thousands of objects per second when in use, although most of the code paths do not need any garbage-collected memory. It is also very difficult to avoid this problem, even when the code is optimized with this specific problem in mind.

Regardless, I expect that further improvements to the compilers and runtimes will eventually remove the problems I encountered, while the language semantics are a good fit for reliable system components.

Glossary

application A program, or a collection of programs, whose purpose is providing a self-contained environment for a particular task. For example, WWW browser, file manager, e-mail client, vim.

bind An operation whose purpose is to make a file accessible under a new path.

chroot A UNIX operation that changes a perceived filesystem root for a particular process.

command A program whose purpose is extending the functionality of user interface, by executing a single action on behalf of the user. For example, UNIX commands grep, cat, ls.

endpoint (filesystem) See translator.

file An operating system's representation of a named fragment of persistent memory.

file system (In this document.) A collection of services provided by the operating system, whose primary purpose is presenting permanent memory devices as collections of files.

filesystem (In this document.) A data format describing the way files are represented on an addressable permanent memory device. Alternatively, a service that implements access to some specific format.

IPC Inter-Process Communication. A mechanism used for sending messages among otherwise isolated processes. Always implemented by the kernel.

kernel A non-terminating program that provides the core services within an operating system.

mount A traditional UNIX operation whose purpose is to interpret a storage device as a filesystem and attach it to the system's namespace.

namespace A collection of files accessible by name in a process.

node The system's representation of a file.

operating system A program, or a collection of programs, that provide abstract computing interface on top of a physical digital system. See section 1.1.

overlaying See union.

POSIX A standardization effort aiming at providing a solid common ground for all UNIX implementations.

protected mode A processor mode with a restricted set of runnable instructions, provided to prevent program from manipulating sensitive resources.

server (In the context of microkernel systems.) An independent program that provides part of the operating system's functionality. A microkernel operating system can be seen as a collection of the kernel and multiple separate servers.

translator A server that usually interprets a physical device and makes it accessible as a file hierarchy. It "translates" between different levels of abstraction.

union A way to present multiple separate file trees as a single combined tree, by virtue of overlaying the directory structure on top of each other.

UNIX An operating system originally developed in 1969 at Bell Labs, which introduced many now ubiquitous concepts and has been adapted into many different systems in use today.

VFS Virtual File-System. An abstraction layer that present many different physical filesystem formats under a single uniform programming interface. Part of an operating system.