Decentralizing UNIX Abstractions in the Exokernel Architecture

by

Héctor Manuel Briceño Pulido

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science

and

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1997

Author . . . .
Department of Electrical Engineering and Computer Science
February 7, 1997

Certified by . . . .
M. Frans Kaashoek
Associate Professor
Thesis Supervisor

Certified by . . . .
Gregory R. Ganger
Postdoctoral Associate
Thesis Supervisor

Certified by . . . .
Dawson R. Engler
Ph.D. Candidate
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Decentralizing UNIX Abstractions in the Exokernel Architecture
by
Héctor Manuel Briceño Pulido

Submitted to the Department of Electrical Engineering and Computer Science on
February 7, 1997 in partial fulfillment of the requirements for the degrees of
Bachelor of Science
and
Master of Engineering in Computer Science and Engineering

Abstract

Traditional operating systems (OSs) provide a fixed interface to hardware abstractions. This interface and its implementation hurt application performance and flexibility. What is needed is a flexible and high-performance interface to OS abstractions that can be customized to each application's needs.

To provide more flexibility and performance to applications, the exokernel architecture decentralizes OS abstractions and places them in libraries. This thesis investigates how to decentralize OS abstractions while maintaining proper semantics. It also describes the implementation of a prototype library operating system – ExOS 1.0 – that has enough functionality to run a wide variety of applications from editors to compilers. ExOS 1.0 serves as an excellent tool for OS research and as a step toward the full understanding of how to design, build, and use library operating systems.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Thesis Supervisor: Gregory R. Ganger
Title: Postdoctoral Associate

Thesis Supervisor: Dawson R. Engler
Title: Ph.D. Candidate

Acknowledgments

The work presented in this thesis is joint work with Dawson Engler and Frans Kaashoek. It goes without saying that the discussions with Tom Pinckney and Greg Ganger greatly increased the breadth and depth of this work.

I thank the members of the Parallel and Distributed Operating Systems group for withstanding me (Costa and Rusty, why are you smiling?) and providing a joyful research environment.

The painstaking task of reading the whole thesis was endured several times by Dawson Engler, Frans Kaashoek and Greg Ganger. Their invaluable feedback is greatly appreciated.

Dawson Engler is thanked specially for questioning many of the issues covered in this thesis, and poking my brain with his questions and suggestions. Without his feedback, this would have been a truly different thesis.

Tom Pinckney is thanked for answering my numerous questions and listening to all my comments (including the random ones).

My parents are thanked for all their encouragement throughout life and their example of hard work and excellence.

A special thanks goes to the Fundación Gran Mariscal de Ayacucho and the Venezuelan Government for making it possible to enrich my education at MIT.

This research was supported in part by ARPA contract N00014-94-1-0985, by a NSF National Young Investigator Award to Prof. Frans Kaashoek, and by an Intel equipment donation.

Contents

List of Tables

Chapter 1

Introduction

It has been widely recognized that traditional operating systems (OSs) should provide much more flexibility and performance to applications [2, 3, 4, 14]. The exokernel architecture is intended to solve the performance and flexibility problems associated with traditional OSs by giving applications protected and efficient control of hardware and software resources. Exokernels simply protect resources, allowing applications to manage them. Libraries implementing OS abstractions can be linked with applications to provide programmers the same interfaces as traditional OSs under this architecture. Two major questions of this thesis are "can library operating systems provide the same abstractions as traditional OSs?" and "if so, how?." This thesis explores mechanisms for providing common OS interfaces while simultaneously maintaining the flexibility and performance advantages of the exokernel architecture.

Traditional OSs enforce abstractions on hardware resources and the software structures used for resource management. They enforce these abstractions by using a well-defined system call interface and keeping all system state centralized in a privileged address space. Centralization simplifies the sharing of system state because all system state is available to all processes when a system call is being serviced. The system call guarantees that the state will not be corrupted or maliciously modified. Unfortunately these advantages come at the cost of flexibility and performance, since applications are forced to use specific abstractions with specific policies to access hardware resources. For example, disk blocks in UNIX systems are cached, and the cache uses a least-recently-used policy for cache block replacement. Applications that benefit from either eliminating caching or using a different policy cannot replace this policy under traditional UNIX systems.

1.1 Decentralization Advantages

The exokernel OS architecture gives applications more control over resources by decentralizing resource management from the OS into unprivileged libraries (called library operating systems). In contrast to conventional OSs, in which a central authority both protects and abstracts hardware resources, exokernels only protect resources, leaving their management to applications. For example, exokernel "processes" manage their own virtual memory. Processes handle page faults, allocate memory and map pages as necessary. This control allows for the implementation of mechanisms such as copy-on-write and memory mapped files all at user-level. Thus, any of these traditional abstractions can be optimized and customized on a per-process basis.

With the exokernel's powerful low-level primitives, traditional OS abstractions can be implemented in library operating systems. Applications can use a specific library OS depending on their access patterns, usage of abstractions, and desired policies. Additionally, if none of the available abstractions are suited for a particular application, unprivileged programmers can safely implement their own abstractions. In this way, libraries provide maximum flexibility and performance to applications.

1.2 Decentralization Challenges

To decentralize OS abstractions many problems must be addressed: state that was previously persistent across process invocations may no longer be; shared state can be corrupted by other applications if not properly protected; resources can be accessed concurrently, so there must be mechanisms to guarantee a minimum amount of concurrency control; finally, since all names manipulated by an exokernel are physical names such as physical page number or a physical disk block, there must be mechanisms to map these names to the logical names commonly used by applications.

The main question then becomes how to implement library operating systems on exokernels. To answer this question, this thesis will explore an implementation of the application programming interface (API) provided by OpenBSD, a BSD-like UNIX OS. This interface provides a widely used set of OS abstractions that, in addition to allowing us to explore the issues in decentralizing OS abstractions, will provide the exokernel prototype with a large application base to test and evaluate the exokernel and library OS architectures. The main challenge of this implementation is conforming to the API while at the same time maintaining the flexibility and performance of the exokernel OS architecture.

1.3 Solution and Contribution

This thesis addresses the challenges of decentralizing OS abstractions by first exploring what semantics they have and how their state is shared. With this information as a foundation, mechanisms for decentralizing OS abstractions are described. Two main approaches are identified: duplicating the mechanisms used under centralized schemes and using mechanisms inherent to decentralized schemes. The insights gained are applied to OpenBSD to create a library operating system called ExOS 1.0 that provides the same OS abstractions as OpenBSD. ExOS 1.0 provides enough functionality to run a variety of applications from compilers to editors.

1.4 Thesis Overview

The remainder of this thesis is organized as follows. Chapter 2 describes the issues and mechanisms used to decentralize OS abstractions. Chapter 3 describes the design, implementation and evaluation of ExOS 1.0 . Chapter 4 discusses planned future work with library operating systems, including a partial design of ExOS 2.0. Chapter 5 discusses work related to library operating systems and the decentralization of UNIX abstractions. Finally, Chapter 6 concludes.

Chapter 2

Decentralizing OS Abstractions

This chapter discusses the issues that arise when decentralizing OS abstractions. In order to provide better insight, centralized and decentralized approaches will be contrasted. Although this thesis does not cover all possible OS abstractions, this chapter gives a high-level overview of common OS abstractions in order to make concrete the issues. Semantics define an important part of OS abstractions, therefore the most important semantic issues will be enumerated. In order to contrast centralized and decentralized approaches, the features or abilities of centralized OSs will be described along with examples of how to implement the major abstractions. At this point, the issues and methods to decentralize OS abstractions can be better understood.

This chapter is divided into four sections. Section 2.1 describes the general OS abstractions. Section 2.2 describes the major semantics characteristics of such abstractions, including protection, atomicity, and concurrency. Section 2.3 presents some of the features and abilities of centralized OSs and how they relate to the semantics of abstractions. Section 2.4 concludes with the mechanisms and issues related to decentralizing OS abstractions.

2.1 OS Abstractions

To focus the discussion of decentralizing OS abstractions, it is useful to first describe some common OS abstractions. This section describes OS abstractions, including some invariants that are enforced in most systems. The abstractions described are divided into four categories: processes, virtual memory, file systems, and communication.

- **Processes**: A process is basically a program in execution. It consists of the executable program's data and stack, its program counter, stack pointer, and other registers, and all other information needed to run the program [17]. Processes can be suspended and resumed to allow multiple processes to timeshare a machine. There are usually a set of credentials associated with each process that define the resources it is allowed to access. Typically a process can create another process creating a family tree of processes.

- **Virtual Memory**: The abstraction of virtual memory simulates access to larger memory spaces and isolates programs from each other. This is usually done by moving memory pages back and forth between physical memory and a larger backing store (e.g. a disk), and using hardware support to remap every memory access appropriately. For example, when two different processes access memory location 5, that location will correspond to a different physical memory addresses for each process, which provides isolation of memory between the processes. The only way processes can share regions of their address space is via explicit sharing. Most OSs provide ways to explicitly setup regions of memory that are shared between processes. The credentials held by the processes can be used to validate the sharing of these memory regions.

- **File System**: File system abstractions permit data to be made persistent across reboots. Information is stored in entities called files. Files are usually hierarchically named to simplify and organize file access. These names are unique and persistent across reboots. Additional information is stored about each file, like access times and ownership, to allow accounting and to restrict access to the file. File system abstractions usually involve strong semantics and invariants, because of the many requirements needed to guarantee persistence. For example, all meta-data (information about files) must be carefully written to a stable storage to guard against inconsistency in case of a power failure.

- **Communication**: Communication abstractions allow processes to exchange information with other processes on the same machine and on other machines in the network. The communication abstractions may have varying semantics, such as guaranteed delivery, best-effort, record-based, etc. Most current OSs do not protect against spoofing of communication channels that go over a network, but they do protect communication channels on the same machine. For example, they generally prevent the interception of information sent between two programs by a third program on the same machine. Also, most OSs restrict access to incoming data from the network to only the processes holding that connection.

2.2 OS Abstractions Semantics

Now that some common OS abstractions have been laid out, their semantics can be understood. This section explores seven of the main semantic characteristics that OS abstractions deal with in varying degrees: protection, security, concurrency control, naming, coherence, atomicity, and persistence. These semantic issues have to be considered in order to properly implement these abstractions under any scheme, be it centralized or decentralized. In some cases, the actual semantics are more strict than necessary and relaxing them will not change the behavior or the correctness of programs.

Protection: Protection prevents unwanted changes to state. Protection can lead to fault-isolation, and to an abstraction barrier. Fault-isolation is the containment of faults within processes. For example, the process abstraction has fault-isolation, in that the failure of one process does not directly affect any unrelated process. The abstraction barrier protects the state of an abstraction from changes due to methods outside the abstraction. For example, the internal state of most, if not all, UNIX abstractions is protected from processes. Processes are not able see or directly change this state, they can only call the procedures exported by the abstractions.

Access Control: Access control defines the kind of actions and entities that are allowed to perform these actions. For example, UNIX associates a <uid,gid> pair with each process. When a process wants to write a file, the OS compares the <uid,gid> pair that is allowed to write with that of the process to determine if the operation is allowed.

Concurrency: Concurrency defines the behavior of abstractions when they are simultaneously accessed or acted upon by programs. There are various degrees of concurrency, from none to complete serializability. For example, if files had no concurrency, it would be acceptable for two simultaneous writes to the same region of the file to produce the effect of writing any mixture of the two regions written. In contrast, most UNIX OSs use an absolute time semantics, specifying that the later write is what would be seen by the next read of the same region of the file (if both readers and writers are on the same machine). This absolute time concurrency can be said to be serializable – its outcome can be recreated from some ordering of all file writes.

Concurrency semantics provide invariants about abstractions that make it easier to reason about their possible states.

Naming: Naming permits unrelated processes to access an object or to communicate with each other. By agreeing on the name, two processes simultaneously or at different times can access the same object.

Naming is used under UNIX for files, pipes, and sockets. For example, by agreeing on a name beforehand, one process can write to a file any mysterious events that it detects, and another process later in time can read the same file and display the results to a user. This behavior is possible because the name associated with the file does not change through time.

Coherence: Coherence defines when and how the abstraction state is well-defined. For example, lack of coherence could imply that a change to a file by one process will not be seen by another process unless the other process explicitly synchronizes the file. UNIX OSs typically provide strong file coherence. If one process writes to a file, the change can immediately be observed by other processes on the same system.

Atomicity: Atomicity describes the behavior of abstractions interruptions. Strong atomicity guarantees make it easier to reason about and to restore the state of an abstraction after an unexpected interruption (e.g. a power failure).

For example, most UNIX abstractions are atomic with respect to explicit interruptions such as signals. Additionally, UNIX abstractions strive to be atomic with respect to unexpected interruptions such as power failure (although they do not always succeed). For example, file delete is atomic with respect to unexpected interruptions – the file is either removed or not. On the other hand, long file writes are not atomic – if an unexpected interruption occurs only part of the write may be completed.

Persistence: Persistence refers to the lifetime of an object. Objects may have longer lifetimes than their creators. For example, system V shared memory segments persist until explicitly removed. Even if these memory segments are not mapped by any process, the data in them will be available to any process that subsequently maps them. This organization allows processes to share information in memory across invocations. Another more useful example is the exit status of processes. Even after a process has terminated, its exit status will be available until the parent process is ready to use it. In contrast, the data memory of a process is not persistent; it vanishes when the process exits.

2.3 Centralized OSs Services and Features

Centralized OSs have certain abilities and features that help in implementing OS abstractions and in guaranteeing their semantics. This section summarizes the relevant characteristics of centralized OSs and describes how they are used to implement the semantics discussed in the previous section. The relevant features include:

- **Controlled Entry Points**: Calls into centralized OSs can only occur at well-defined entry points (i.e., system calls), thus guaranteeing that all the guards and proper checks have been executed before changing the state of any abstraction state. Processes in general do not have this characteristic; procedures can be entered at any point.

- **Different Protection Domains**: By executing in a different protection domain, abstraction state can be protected against wild reads and writes. Preventing wild reads ensures that no extra information is revealed about the internal state. Preventing wild writes guarantees that the state is not modified by methods external to the abstraction. This organization provides fault-isolation, because faults outside the abstraction are not propagated into the abstraction except (possibly) through the exported interfaces, which can guard against inconsistent data.

  Different protection domains combined with controlled entry points allow one to implement protection by separating the abstraction state from its client and by controlling the methods that modify the abstraction state. Processes can not intentionally or unintentionally modify the state of an abstraction, except through the well-defined methods. If a process has a fault, it will not affect the abstraction, except for the fact that the abstraction has to detect the termination of the process and properly clean up its state.

- **Well-Formed Updates**: By having controlled entry points and different protection domains, centralized abstractions can enforce well-formed updates. The abstraction methods are the only ones that can modify the object, and they can only be called at specific starting points. Thus, if they are correct, all updates to the abstraction state will be well-formed.

- **State Unification**: All of an abstraction's state can be unified in a single location. Preventing multiple copies eases the task of maintaining coherence. For example, in most centralized OSs, only the file system caches file blocks, so that no two processes see a different view of the file (unless they explicitly request this).

  Abstractions can control what state processes see and when they see it. The state can be made inaccessible while it is incoherent. For example, file blocks are usually cached in an OS to avoid repeated access to disk. If one process writes to a cached file block, other processes that want to read the block will wait for the ongoing write to complete and then read the copy cached by the file system abstraction.

  Atomicity can be implemented in a similar way to coherence. State is not allowed to be seen if an atomic operation affecting that state is in progress. Additionally, UNIX systems provide the notion of signals, a way to interrupt processes. Certain system calls should appear atomic even if interrupted by signals. Abstraction implementations usually wait until all resources for a given operation are available, then they check for any signals. If any signals are pending, the operation is aborted and no state is changed. Otherwise, the operation is carried through until fully completed. For example, if a large file write request is made, the file abstraction first checks for the available disk space. If there is enough and it does not have any signals pending, it will start the large write operation. Any process trying to look at the file at that point will be blocked until the write completes. Additionally, if any signals are posted after the write has started, they are ignored, because undoing the writes is difficult on traditional UNIX file systems. This "guard-action approach" guarantees that the write will appear atomic to all processes. Even the midway point is atomic (unless the system crashes) because no other process will be allowed to see the file.

- **Single Point of Serialization**: With unification, the centralized OS is a single point for serialization. Using standard mechanisms such as locks and critical regions, a centralized OS enforces concurrency semantics of abstractions. Because centralized OSs live longer than most processes, they can release locks even if the relevant process terminates.

  Concurrency control can be easily implemented with a single point of serialization. Processes accessing an abstraction do so through one or more well-defined entry points. The abstraction can block requests if there are requests in progress. For example, if two file writes operations to the same region of a file are requested, the file system abstraction blocks the second until the first is finished, thus, the behavior of concurrent write accesses to files is well-defined.

- **Global Information**: With unification of all abstractions in a centralized location, abstractions now have access to information about all processes. This information includes access to information across time. This eases the job of naming and persistence.

  In addition, all name resolution takes place at one location: the centralized OS. Together with access to global information, the names can be quickly translated to the underlying object. For example, if a process wants to access a shared memory segment, it will query its virtual memory manager for the segment. This manager in turn can communicate with the shared memory abstraction to locate the segment. Once located, the virtual memory manager for the process can map the segment in the process's address space.

- **Expanded Authority**: Centralized OSs generally have an expanded authority that regular processes do not have. This authority allows protected and controlled access to powerful resources.

  Security is enabled by the fact of global information and strict checks for access. The centralized OS is the holder of all resources and process credentials. It can use these credentials to validate access to resources. The drawback is that improper or missing checks will allow unauthorized access to a resource. This is in contrast to a scenario where the resources and credentials are separated and checks are always done.

- **Single Point of Update**: The centralized OS provides a single point of modification to improve the system and add abstractions. Once this modification has been made, all processes benefit from the change.

Unfortunately, while centralization can simplify the implementation of OS abstractions, it limits flexibility and performance. Flexibility is limited by providing only one fixed set of abstractions to resources. Thus, any program that could benefit from a different abstraction for a resource are unable to do so. Performance is limited by inflexibility and strong invariants. For example, if two writes are requested to two different regions of the same file, the UNIX file system abstraction will not allow the two writes to take place at the same time in order to guarantee coherency and atomicity. If the processes doing the writes do not require these semantics (e.g. if it is a temporary file), they cannot take advantage of this fact to improve their performance.

This problem can be solved by decentralizing of OS abstractions. With appropriately decentralized control, abstractions can be better customized and specialized by applications according to their needs.

2.4 Decentralizing OS Abstractions

There are two ways to decentralize OS abstractions: *partially* duplicate the features and abilities of centralized schemes or use the inherent abilities of decentralized schemes to implement the OS abstractions and their semantics in libraries. In some cases, the nature of the semantics required by an abstraction may limit the choices for implementation to centralized schemes. For example, in order to provide absolute coherence, it may be necessary to locate the state in a centralized location to avoid any communication with other holders of the same state. To better explain how semantics and abstractions can be implemented in a decentralized manner, this section presents mechanisms that can be used in decentralized settings and discusses possible implementations of the semantics properties described in 2.2.

The relevant abilities of decentralized OSs are:

- **Separation**: In contrast to centralized operating systems' unification, abstraction state is maintained separately for each process. This organization allows each process to hold and protect its own abstraction state. As in microkernels, a practical consequence is that errors in the OS code no longer crash the system, only the application they are associated with.

- **Minimized Authority**: Minimized authority follows the principle of least-privilege. Each abstraction has control over only its own internal state. Any modification to other abstraction state must be done through explicit interfaces with that abstraction.

- **Decoupled-changes**: In contrast to centralized operating systems' single-point-update, decentralized OSs decouple changes. This means that changes in the implementation of an abstraction for one process do not directly affect other processes. This structure gives room for flexibility and performance improvements, since the implementation of an abstraction can be optimized and customized for its specific usage.

- **Local Information**: If implemented properly, decentralized abstractions work mostly with local information. This organization has the advantage of reliability and scalability. Abstractions rely less on and have less contention for a central repository of information.

Until the bulk of the OpenBSD implementation can be done with decentralized state, there are many cases that local information is not enough to implement decentralized OS abstractions. For such cases, state must be shared either across different instances of the abstraction (state unification) or across different abstractions (global information). There are three mechanisms that can be used to share this state: *shared memory*, *exoservers*, and *in-kernel mechanisms*. With shared memory, processes using an abstraction map the abstraction state into the process' address space and act upon it directly. Exoservers are servers that follow the exokernel precept of separating protection from management: they limit their functionality to that required to guarantee invariants and protection, and leave resource management to applications. Finally, in-kernel mechanisms can be used to provide protection that would be impossible with servers or to enhance the performance of commonly used services.

Shared memory is an efficient way for processes to communicate with each other and share state. Enforcing semantics on updates to state in shared memory is hard, because there are no guarantees about the code used to modify this shared state (this is not the case for a privileged server or an in-kernel interface). However, some important semantic properties can be achieved. Concurrency can be achieved by using critical regions or non-blocking synchronization mechanisms. Some degree of protection can be achieved by mapping the pages read-only. Additionally, partial copies of the state in shared memory can be stored in each process in case it needs to be reconstructed or to detect corruption. Fault-isolation can be achieved by verifying the validity of data before using it and mapping errors to possible errors under the same UNIX API if data is inconsistent.

Exoservers can provide minimal protection for resources, allowing applications to decide what semantics to use on their abstractions. In this way, flexibility is achieved because programs decide the semantics and ways to access the resources. Performance is achieved, because applications know best how to access the resources and are not fixed by one specific interface. For example, a trusted file server can be used to manipulate the name space for files, and hand-off permission for further access to that file to other processes. Processes can then coordinate with the server access to the file if they want to maintain properties such as atomicity and concurrency, or can directly access the file, if these semantics are not required by the process.

In-kernel support can be provided for commonly-used functionality to increase efficiency and to increase the trust on data held in the kernel or other processes. In-kernel support should only be used for commonly-used mechanisms and should be by-passable by processes. Otherwise, the mechanism could limit future unenvisioned usages of the resource. For example, in-kernel support can be added in order to allow safe, efficient sharing of cached disk blocks. In the XOK exokernel, a kernel-maintained buffer cache registry keeps track of mappings between <device number, block number> pairs and physical pages. This registry is mapped read-only by everyone. For example, a process that wants to read disk block number 38 can query the registry for its location. If the block is in memory the process can map the physical page it is on. Otherwise, it can fetch the block from the disk and optionally ask the kernel to include it in the cache registry. Note that a process is not forced to use the registry. It is a service that can be bypassed, but it is likely to be used by most processes.

With these abilities and mechanisms for sharing state, the important characteristics of centralized OSs can be duplicated, and the semantics of OS abstractions can be implemented:

- **Protection**: Protection can be achieved by storing abstraction state in different protection domains. Either a separate protection domain is used per abstraction as a whole, or processes can hold multiple protection domains each containing partial state of the abstraction. For example, a process could have four different protection domains for the state of the four general classes of OS abstractions. In this way, the states for each abstraction could be completely isolated from each other. Furthermore, the state of the network abstraction for one process could be isolated from that of another process. This organization provides more fault-isolation within the same abstraction.

- **Access Control**: Access Control can be provided via minimized authority. Each process or abstraction only has access to its own state. If it needs to modify or act upon another abstraction, it must do so through explicit interfaces and have credentials that allow it to do so. This approach would provide even better security than is provided in a centralized system, because the effect of a bug is restricted to the authority of the associated abstraction [11].

- **Concurrency Control**: Concurrency control can be achieved either using standard distributed algorithms or duplicating the single-point serialization of centralized OSs via mechanisms such as critical regions and non-blocking synchronization [7]. Locks should be avoided under decentralized schemes because it is hard to reason about the current state of the abstraction when a lock holder fails to release the lock (e.g. it crashes before releasing the lock).

- **Naming**: Name resolution can either take place in a centralized location, or queries can be passed around until the name fully resolved. A centralized name server can be used, where different processes and abstractions register the names they can resolve. Conversely, only the roots of the names may be registered, and it would be up to the servers to pass any unresolved part of a name to another process for resolution.

- **Coherency**: The easiest way to achieve coherency is by keeping only one copy of the data. Either all of the data is placed in a centralized location, or a registry can be used to allow the data to be in different locations. When an application needs to access new data, it can check the registry. The registry can point to the holder or location of the data or respond that it has no knowledge of it. In the first case, the application fetches the data from the location or holder. In the latter case, the application fetches the data directly from the source (e.g., a disk) and registers itself as the new holder of it.

- **Atomicity**: Atomicity can be implemented similarly to how it is implemented in a centralized scheme. Under centralized schemes atomic semantics are provided by hiding abstraction state until an atomic operation has completed. By their global information and expanded authority properties, if the atomic operation is aborted they can roll-back any abstraction state. The lack of these properties under decentralized schemes make it difficult to implement atomic semantics should multiple abstraction need to be updated.

- **Persistence**: Persistence of state beyond the lifetime of a process can be achieved either by using a centralized approach or some form of token passing. Under a centralized approach, a server can hold any persistent state, either through the lifetime of the state or at the point that no one else has access to it. For example, system V shared memory segments are persistent even when no process has them mapped. A server can be used to hold the segments that are not currently mapped. Processes will hold the pages of the shared memory region while they are in use. Alternatively, a token passing mechanism can be used. The state can be passed around when the process holding it dies. For example, the exit status of a child process is persistent after the child dies. This status can be passed to the parent at the time of the child process's death. Therefore the parent will have access to the status at any time after the child's death.

Chapter 3

ExOS 1.0

Considerable insight regarding decentralization of OS abstractions can be gained from designing and implementing a simple library operating system. This chapter describes ExOS 1.0, a library operating system that implements OpenBSD abstractions for the XOK exokernel. Section 3.1 describes the high-level design of ExOS 1.0. Section 3.2 describes the kernel on top of which ExOS 1.0 is implemented. Section 3.3 goes into details about how major abstractions are implemented; section 3.4 evaluates the performance of ExOS 1.0 compared to that of OpenBSD. Section 3.5 provides a short discussion of what we have learned so far.

3.1 Design

In order to keep the implementation focused, it is important enumerate the design goals of ExOS 1.0:

- Proof of concept: The central design goal was to demonstrate that a library OS could provide enough functionality to run many existing applications.
- Simplicity: In order to implement ExOS 1.0 quickly, no fancy algorithms are used. This reduces the time required to debug the code and makes it easier to change which is important because the initial implementation is changed frequently.
- Performance: performance should be similar to that of OpenBSD for common cases. Inferior performance should only be a result of an immature implementation rather than fundamental problems.
- Flexibility: It should be easy to customize the abstractions for particular applications.
- Sacrifice some semantics: There is a tension between flexibility and strong semantics. This initial implementation was designed to sacrifice some semantics such as protection in order to increase flexibility. It is arguable if the additional protection that OSs provide is important when considering the powers of the traditional “root” user, and the relative ease with which unauthorized programs obtain this credential on today’s UNIXes.

Providing a full implementation of OpenBSD, along with all the semantics, was not the goal of ExOS 1.0. It is an implementation that gives enough functionality to run a wide variety of applications, like vi, gcc, make, and common benchmarks such as Andrew [9] and Lmbench [12]. Although it was not a design goal, ExOS 1.0 has also supported the implementation of an emulator that can run some statically-linked OpenBSD binaries unchanged.

ExOS 1.0 is implemented as a user-level untrusted library that is linked with each application. It has procedure calls for most of the system calls of OpenBSD . The system does not use servers to implement any UNIX functionality. Shared memory is used as the main way that the UNIX abstractions for each process communicate with each other. A in-kernel mechanism is used to provide coherence of cached disk blocks. This provides a system that is very customizable and optimizable per application, without affecting other applications (except via faulty implementations).

3.2 Experimental Framework: XOK

ExOS 1.0 is implemented as the default library OS for the XOK exokernel. The fundamental idea of exokernels is to decouple resource management from protection. The abstractions and policies that once resided in the OS are moved to libraries. The OS only protects the resources. This allows applications to specialize and customize OS abstractions according to their needs.

XOK protects the various hardware resources, including environments (i.e., protection domains), time quantums, hardware page tables, network devices, the block registry, and the console. These resources are protected by the use of hierarchically-named capabilities composed of two parts: properties and names. The properties define attributes such as valid, modify/delete, allocate, write, and name length. Capabilities have variable-length names. The names are used to define a dominance relationship between capabilities. That is, the dominating capability can do anything that the dominated capability can. A capability is said to dominate another capability if its name is a prefix of the other.

There is a list of capabilities associated with each process and with each resource. Whenever a process wants access to a resource, it presents a capability. If that capability dominates any of the capabilities associated with the resource, the process is granted access.

The networking device is protected by a packet filter engine [13]. The XOK implementation of the packet filter engine, DPF [5], uses dynamic code generation to obtain high performance.

3.3 Implementation of Major Abstractions and Mechanisms

3.3.1 Bootstrapping, Fork and Exec

A recurring problem with any OS implementation is bootstrapping: how to initialize and setup the system when it starts. After the first process is started, other processes can be created using *fork* and *exec*. *Fork* makes a duplicate image of the running process, with a different process identifier. *Exec* is used to create a new process that is the result of overlaying the image of the running process with that of the new executable. The problem is that XOK has no concept of UNIX processes or of file systems. XOK only has concepts of environments (i.e., context where programs execute in) and time quantums.

To solve the problem of bootstrapping, the very first process image is linked into the kernel. A simple loader in the kernel creates an environment, copies the image of the first process into that environment, and allocates a time quantum for it. This first process detects that it is the first process and initializes all of the major ExOS 1.0 data structures that live in shared memory. For usability, the first process also acts as a login server, allowing users to use the system via a simple telnet-like program. When other processes start, they can detect that the data structures have been already initialized and simply map them into their address space at the agreed upon virtual address.

Implementing *fork* and *exec* for ExOS 1.0 can be tricky because abstractions may need to be updated in special ways. To solve this problem, ExOS 1.0 provides mechanisms for abstractions to be notified of *fork* and *exec* calls with a handler of the new environment. In this way, abstractions can modify their own state and modify the state of the new process at *fork* and *exec* times. *Fork* works by creating a new environment, initializing its memory (including shared memory segments), calling any registered abstraction functions, and granting the new environment a time quantum so it can run. When the parent process sets up the memory of the child, it maps its own memory on the child's environment and sets the mappings to be copy-on-write. When the child writes to the first time to an untouched page, a page will be allocated, the touched page will be copied to the new page, and the new page will be mapped into the location of the old page.

Exec is implemented similarly to *fork*. For simplicity, *exec* is implemented by creating a new environment, setting its memory from an executable image, calling any registered abstraction functions, granting the new environment a time quantum, and freeing the environment of the parent. The interesting part of the *exec* implementation is having the child inherit the parent's resources and keeping the same process identifier number. The first problem is solved by the abstraction handler functions, which are aware that the parent environment will be deallocated and therefore do not increase the reference count for the resources. The second problem is solved with a table that maps XOK environments to UNIX process identifiers. Remapping the new environment's process identifier to that of the parent has the desired effect.

3.3.2 Process Management

UNIX processes use process ids to send signals and identify all process relationships. Process relationships form when a process (parent) forks off another process (child). The problem is that XOK does not use process ids to identify environments. This problem is solved by using a table that maps environments ids to process ids. Additionally, there is a structure bound to each environment called the "Uenv" structure that holds important information. This structure is used to store the process id of the parent process. In this way, processes know who their parent is and can determine the environment ids of other processes from their process ids. This information is necessary when processes need to communicate with each other using any of the XOK primitives. For example, if they need to read or write another environment's "Uenv" structure.

The "ps" program is implemented by using this table. The program locates all of the environments running as UNIX processes and reads their Uenv structures. From these structures, it can learn each process's process id, parent process id, arguments with which the process was called, and running status among other information.

3.3.3 Signals

Signals are the equivalent of software interrupts. They can be sent between processes explicitly or generated internally by a process in response to exceptions (e.g., division by zero). When a signal is received, it is ignored, a default action is taken, or a registered handler is executed. After handlers are executed, program execution resumes at the point of interruption (such as after a hardware interrupt).

The problem is that XOK does not directly support signals. Instead, processes can explicitly communicate with each other via IPC or shared memory. For the implementation of signals, neither IPC nor shared memory was used. Instead, when a signal is sent to a process, the sender writes to the Uenv structure of the target process's environment. This approach has the advantage of being simple and allowing the other process to handle signals when is safe to do so (an IPC would interrupt the current execution flow of the target process). By default, a process only checks for any pending signals at the beginning of the time slice.

Currently, it is possible for signals to delivered at inappropriate times. For example, although locks are discouraged, they do exist in ExOS 1.0 . Therefore, if a signal is delivered while holding a lock for an important table, like the file table, the lock might never be released. This problem is temporary, since it can be fixed in two ways: Completely remove locks from the library in favor of critical regions or disable signals while locks are held.

3.3.4 File Descriptors

File descriptors are small integers used to access resources such as files, sockets, pipes, and terminals. They are returned as handles when resources are accessed for the first time (e.g., when calling *open*, *socket* or *pipe*). An important feature of file descriptors is that they can be shared across processes when they *fork* or *exec*. Among other important state, file descriptors hold the current byte offset were read or write calls to a file should start. At the completion of any these calls the byte offset is incremented according to the number of bytes read or written.

There are two major challenges with the implementation of decentralized file descriptors under XOK. The first is correctly sharing the offset and attributes associated with the file descriptor. The second is that, since each application has its own implementation of OS abstractions, they could use different procedures for accessing the underlying object. So, for example, different processes may have different communication needs and may therefore have different implementations of sockets. However, processes should still be able to inherit a socket from their parent process. Therefore, there must be some way to flexibly manipulate objects (like files and sockets) that are inherited from other processes.

The first problem is solved by using a file table shared among all processes. The second problem is solved by using an object-oriented interface to access the objects. The file table contains offsets, attributes and other information about underlying objects that are shared and mapped at the same virtual address by all processes. By mapping it at the same virtual address, pointers can be inherited via *fork* and *exec* with no changes. This solution solves the problem of byte offset sharing. With

FD type	Description
NFS_TYPE	file objects from remotely mounted volumes
CFFS_TYPE	local file system objects
UDP_SOCKET_TYPE	UDP internet family sockets
TCP_SOCKET_TYPE	TCP internet family sockets
PIPE_TYPE	for file pipes.
PTY_TYPE	Simple non-conforming pseudo-terminal
CONSOLE_TYPE	PC3 terminal emulation on console
DUMB_TYPE	line oriented access to console
NPTY_TYPE	POSIX conforming pseudo-terminal
NULL_TYPE	for /dev/null access

Table 3-1. Supported File Descriptor Types

each file table entry, there is also an integer type for the particular descriptor. Table 3-1 shows the file descriptor types currently supported. When processes start, each file descriptor abstraction type registers a set of functions to be called for each high level file descriptor function. Therefore, when a process inherits a file descriptor, it will know what kind of file descriptor and which functions to use to manipulate it.

3.3.5 Files

ExOS 1.0 supports both a local and a remote file system. The local file system is based on the Co-locating FFS (C-FFS) [6]. The remote file system is based on Sun's Network File System (NFS) [16].

The NFS implementation is simple. The first process mounts the root file system from a statically-determined server and places in shared memory a socket handler (see 3.3.6) to be used for all communication with the NFS server. Processes map each remote file operation to NFS requests. Each process starts and waits for each NFS request it initiates. Currently, no file block or attribute caching is done, but it would not be difficult to add it.

The advantage of using a simple NFS implementation is that all the burden (dealing with the file system semantics) is placed on the server. Additionally, processes can optimize the handling of NFS requests according to their needs, because the NFS client is part of the ExOS 1.0 library.

The C-FFS implementation is somewhat more complicated. Block caching is needed for performance reasons to reduce the number of accesses to the disk. Additionally, C-FFS is implemented as a library in a decentralized manner, so there is no single server to deal with concurrency and coherency.

Coherency is the major challenge of the C-FFS implementation. There are two types of coherency problems associated with UNIX file systems: block coherency and file information coherency. The first relates to other processes being able to see file writes immediately after they take place, and the second relates to other processes being able to accurately query the attributes of the file. These challenges are solved with a kernel-maintained block registry and using shared table of *inodes* (an inode is the structure containing accounting information as well as disk location information for a particular file or directory).

The block registry allows applications to implement the traditional UNIX buffer cache with coherency properties in a decentralized manner. Whenever a process needs to read a block, it checks with the block registry first. If the block has been read by another process or is in memory somewhere, the block registry returns the physical page where the block resides. The process then maps this physical page. If the block is not in the registry, the process fetches the block and records its physical page with the registry. In this way, all accesses to disk blocks are guaranteed to be coherent because only one memory copy of it is used at any time by UNIX processes. Although, should a process not need this property, it can bypass the block registry (which results in the same behavior as user-level caching).

The *inode* structure contains a reference count and a pointer to disk blocks containing the actual information (i.e., owner, type, length, etc.) of the file. A shared table of all the inodes currently in use is shared and mapped by all processes. The reference count is used to keep track of how many processes are accessing the underlying file. This reference count helps to solve the additional problem of dealing with the UNIX semantic that allows processes to access a file even after its name has been removed. If the process holds a file descriptor associated with the file when the file is removed, it can still access the file until the file descriptor is closed. Processes that want to query the attributes of a file use the inode table to locate the disk block containing the full attributes of the file. The use of the block registry guarantees to processes who use it properly that there is only one in-memory copy of the inode. Therefore the attribute information in that copy is guaranteed to be coherent.

In order to use multiple file systems, UNIX has the idea of mounting and unmounting. The mount action connects two file systems at a specific directory in the base file system. If a pathname includes this directory, it will be taken from the mounted file system. For example, if NFS is the root file system and the root of a C-FFS file system is mounted at "/mnt", then the file "/mnt/foo" identifies the file "/foo" in the C-FFS file system. This abstraction is implemented by using a shared "mount table" that maps directories from one file system to another. This table is used when names are resolved in order to identify which file system the files reside on.

3.3.6 Sockets

Sockets permit bidirectional communication between two or more processes, on the same machine or across a network. There are two important classes of networking sockets: unreliable (based on the user-datagram protocol - UDP) and reliable (based on the transmission-control protocol - TCP).

There are three interesting actions regarding socket communication: connection setup, packet reception, and packet sending. Each is discussed below.

The problems with connection setup are usually intrinsic to the protocol itself. UDP is very simple, requires little space for connection state, and involves no communication in setting up a connection. The file table associated with file descriptors has extra space for application-defined data. This space is used to hold UDP connection state: the source and target destination. At connection time, this information is setup and a DPF filter is inserted into the kernel to receive packets destined for the new socket. The TCP implementation is somewhat more complicated, because of protocol-specific differences. Among other things, connection state is held in a separate table of TCP connections because TCP's data structures are much larger than UDP's.

XOK does not understand anything about network protocols. It simply provides packet filtering and raw send interfaces. Packet sending consists simply of passing a buffer to the network driver, which sends the packet directly onto the network. Packet reception is more difficult because sockets and therefore network connections can be shared between processes via file descriptors. To permit the sharing of sockets among processes, network buffers are mapped in shared memory. This permits any process sharing a UDP or TCP socket to read data from the network buffers.

3.3.7 Pipes

Pipes allow unidirectional data transfer between processes. One end of the pipe is used for reading and the other end for writing. They are commonly used to inject the output of one program to the input of another. For performance reasons, in transit data is held in memory, usually in a fixed size buffer. The important feature about pipes, is that all read and write requests regardless of size will complete successfully unless an exceptional condition arises (i.e., there are no more writers or readers). The challenge with the implementation of pipes is satisfying this behavior when reading or writing data that is larger than the size of the intermediary buffer.

The solution is to do partial reads and writes and block until there is more data or space. Pipes are implemented using a table of queues, those queues contain head and tail pointers, buffers, and locks. When a large read request needs to be satisfied, the queue is locked while as much data as possible is read. If the request is not satisfied, the operation waits until more data is available. This procedure repeats until the read request is satisfied. The same is done for writes. By the use of partial reads and partial writes, any amount of data can be transfered in one "read" or "write" operation.

3.3.8 Pseudo-Terminals

Hardware terminals like the computer console allow human interaction with the system. A terminal OS abstraction provides an interface between hardware terminals and processes. A pseudo-terminal OS abstraction is like a terminal abstraction except that instead of interacting with a keyboard and screen, it interacts with another process that is simulating a terminal device. Pseudo-terminals are used for remote logins to emulate the capabilities of a terminal, such as interrupts (for example, pressing control-C) and the ability to delete characters. The main problem with pseudo-terminals is that they are very complicated.

ExOS 1.0 addresses this problem by borrowing the implementation used for OpenBSD and incorporating it directly by implementing the surrounding functionality that it expects. In this way, very little understanding about the abstraction is needed in order for it to work. There are two major services that had to be implemented: allocation of memory and blocking primitives to allow processes to sleep while there is no "user" input. Allocation of memory is avoided by statically allocating the memory for pseudo-terminals in shared memory. It is necessary to store the terminals in shared memory, because they are shared across processes most of the time. The necessary blocking primitives are implemented using blocking primitives supplied by XOK.

Test	OpenBSD	ExOS1
wc a 4.4 MB file	0.28 sec	0.28 sec
hello program	0.0025 sec	0.08 sec
copy a 57 MB directory using pax -rw	53.2 sec	51.28 sec

Table 3-2. Application Benchmarks Result. Units are in seconds (smaller is better).

This implementation allows us to run applications like vi over telnet connections. However, no effort has been made to make this pseudo-terminal software extensible.

3.4 Performance Evaluation

A wide variety of benchmarks have been run to test the functionality and performance of ExOS 1.0 . Performance is not as great as it could be due to the slow process creation time (this is explained in the next section).

All performance comparisons were made on a 200 MHz Pentium Pro computer with 64 MB of memory. The OpenBSD measurements were done on the same machine using OpenBSD 2.0 (current). All measurements were done using the local file system. The times for the first and third benchmark are the wall-clock times (via the *csh's* time command); the time for the second benchmark is calculated using the *time* system call before and after "exec"ing the "hello" program.

Measurements for three simple application benchmarks are given in table 3-2. The first benchmark consists of using the *wc* utility to count the number of words in a 4.4 MB file. The second benchmark consists on timing the execution of a program that prints "hello world". The third benchmark duplicates a 57 MB directory using "pax -rw". The results show that ExOS 1.0 I/O performance is comparable to that of OpenBSD . There are still some deficiencies with process creation time demonstrated by the results of the second benchmark.

ExOS 1.0 can also successfully execute the Andrew Benchmark [9] and most of the Lmbench benchmarks [12] (except those using mmap and RPC which are not currently supported). Again, the benchmark results are not great: ExOS 1.0 has not been optimized at all – simplicity was one of the main design goals.

3.5 Discussion

At this point we are satisfied with the functionality of ExOS 1.0 . It allows us to execute a wide variety of applications, from compilers to editors. It is still missing some important functionality, including job control, mmap, and demand loading. In terms of performance, there is considerable work to be done. Process creation is currently very slow for several reasons, including large executable sizes, lack of demand loading, and suboptimal implementation of exec. The executable size is large because an entire operating system is statically linked to each application. This problem can be solved with shared libraries. The lack of demand loading means that the whole executable has to be read and mapped at process creation time. The implementation of exec is suboptimal because it has to create an extra environment and abstractions have to properly pass objects that are inherit. By using the same environment, it will be simpler to inherit objects across exec. It should not be hard to fix these problems in the future.

The current system is insecure. The unix credentials for a process are held in the process's memory and XOK capabilities are seldomly used. Although programs that are unaware of these security flaws will behave as if the system was secure, fixing the security problem is a high priority.

The system provides some minimal protection to data structures. They are protected by being mapped at very high addresses. This has been enough to protect against most wild writes.

Concurrency control is partially provided. For remote files, access is serialized by the server. For local files, concurrency is discouraged because the synchronization and concurrency functionality has not been fully added. Sockets and pipes fully satisfy their concurrency semantics by the use of locks.

Coherency for the file system is fully provided. For remote file systems like NFS, it is not a problem because no remote file or attribute caching is currently performed. For the local file system, coherency is implemented via the in-kernel block registry.

We believe that, although the system has many deficiencies at this point, it provides an excellent framework for OS research and an important stepping stone toward a good implementations of library operating systems. It is very easy to customize the interfaces and the underlying implementations of the UNIX abstractions. Additionally, most policy decisions, like page allocations etc., take place in the ExOS 1.0 library, leaving a significant room for experimentation and exploration.

Chapter 4

Future Work: ExOS 2.0

The ExOS 1.0 library operating systems represents a significant step towards understanding how to build library OSs, but many directions remain to be explored. This chapter describes the ideas and motivations behind ExOS 2.0 . Section 4.1 describes the design goals of ExOS 2.0 . Section 4.2 outlines the current vision for the future implementation of the major UNIX abstractions.

4.1 Design Goals

ExOS 1.0 provides much of the functionality of OpenBSD and maybe sufficient for environments like research institutions where it is generally assumed that there are no malicious users. ExOS 2.0 will be designed to show that the concepts of library operating systems and decentralization scale even to more stringent semantics and functionality. For example, ExOS 2.0 should be usable in multiuser time-shared environments, where there are potentially malicious users and where many buggy programs are used. In order to build a robust system, increased levels of protection, security and functionality are needed. There are four fundamental goals for ExOS 2.0 :

- **More Protection and Fault-Isolation**: A single process should not be able to render the system useless unintentionally by simply writing to locations in its virtual address space.
- **Security**: The security policies of UNIX must be enforceable. For example, processes should not be able to masquerade as other users.
- **Performance**: Performance should be similar to, if not better than, that of modern OSs (e.g., OpenBSD) for unaltered UNIX applications. For specialized cases where knowledge of the applications is exploited, performance should be significantly better.
- **More Functionality**: ExOS 2.0 should have full functionality including mmap, job control and demand loading of executables. Additionally, the functionality that is currently available should more closely match the semantics defined by their underlying abstractions.

The next section describes how these goals can be achieved.

4.2 Design Description

Compared to ExOS 1.0 , ExOS 2.0 will make more use of user-level servers and in-kernel mechanisms to provide more protection and security for the abstractions. Furthermore, shared memory structures will not be mappable (or writable) by all processes, only to those that need to use the underlying data structures. In this section, the invariants and semantics that the ExOS 2.0 design will focus on are presented for each of the major abstractions.

4.2.1 Local Directories and Files

The most important invariants and semantics associated with local directories and files for ExOS 2.0 are:

- The link count of the file must match to the number of names for the file, both in memory and on disk.
- Access, modification, and create times of files and directories must be updated whenever they are accessed, modified or created.
- All "live" files should be accessible by a pathname, except those that have been removed but are still being referenced by some process in the system.
- For directories with particular permissions, mandatory locking must be observed. That is to access a region of a file, either there is not a lock for it, or the process is the holder of the lock.
- For directories with particular permissions, any user can create files on it, but they can only delete files if they satisfy certain conditions (i.e. the user is the owner of the file).
- Users, identified by a user id and a list of group ids, must only be able to access, create and remove files or directories for which they have explicit permissions.
- All file operations must be atomic to signals. All file create, unlink, link operations are atomic to unpredictable interruptions like power failures (in this case assuming the "fsck" utility is ran when the power is restored).

In order to guarantee all these invariants, all namespace manipulations will be handled by a user-level server. Read operations on directories and file attributes can be optimized by allowing processes to map some of the server's data structure read-only.

- All file system operations to files must be serializable.
- All programs have the same coherent view of the file (except for a special case of mmap)

By using the block registry and a synchronization protocol, these last two invariants can be satisfied. The block registry permits processes to have the same view of the files at all times. The synchronization protocol will serialize file system operations. Additional in-kernel mechanisms will be used to optimize read and write access to the disk. A proxy permission can be setup by the owner of the file (the user-level server) to allow other processes to directly read and write specific regions of a file. In this way, applications can be allowed to customize and specialize access to individual files, while at the same time maintaining the semantics associated with UNIX files.

The ability to mount file systems on top of each other can be achieved by using the user-level server. Processes can register with the user-level server where they want to mount file systems, and how the requests should be served. So, for example, the initial process can mount NFS file systems on top of the C-FFS file system. When processes access files from the NFS file system, they will handle the requests themselves.

4.2.2 Sockets

The most important (for ExOS 2.0) invariants and semantics associated with sockets are:

- The data in the incoming network buffers must have been received from a network interface.
- Network packets must be accessed in a FIFO manner. Once read, they cannot be read again. If two processes are reading from the same UNIX socket, only one copy of any data will be read.

Although sockets can be shared, the most common case is that either they are used by a single process or they are handed-off from one process to another (this happens when a process "forks" and then the parent process closes the socket). These cases can be implemented directly by a process, by keeping the socket privately, and marking it "to be handed-off" when a *fork* takes place. Then the next process to access the socket will take control from then on. For the uncommon case of the socket actually being shared, this functionality can be migrated to a user-level server that will coordinate access to network packets.

- No two sockets can be bound to the same address. For unconnected sockets, the address is the local port number. For connected sockets, the address is the port number if a connection has not been established. Otherwise, the address consists of the port number and the <port number,network address> of the other side of the connection. This implies that packets received from the network belong to at most one socket.

This problem is already solved by the packet filter used in the kernel. It will not allow a packet to be delivered to two filters. If the filters are not shared, the packets can be delivered safely.

- UDP and TCP Network ports below 1024 can only be bound by privileged processes.

This invariant can be enforced by having a server own all filters that accept packets for these port ranges. Whenever a process wants to use a port in this range it most obtain the filter from this server.

4.2.3 Pseudo-Terminals

The most important invariants and semantics associated with pseudo-terminals for ExOS 2.0 are:

- Pseudo-terminals must be accessed in a FIFO manner (except for some special cases like *unputc*).
- It must be possible to open the two ends of a pseudo-terminal (called master and slave) at different times.
- It must be possible to revoke access to a pseudo-terminal while it is in use. This functionality is needed to guarantee the safety of login sessions.

Pseudo-Terminals are not performance critical. In addition to these invariants, they have many features and options. For these reasons, pseudo-terminal functionality will be placed in a user-level server. If a privileged user needs to revoke access to a pseudo-terminal, it can simply notify the server. Additionally, centralizing pseudo-terminal functionality simplifies the name resolution needed by the second invariant.

4.2.4 Signals

The most important invariants and semantics associated with signals for ExOS 2.0 are:

- Signals from a given process must only be sent to processes in a defined group or to processes owned by the same user.
- Processes must not be allowed to ignore certain signals (e.g., KILL) that are specific to terminate the receiver process.
- Once a signal has been sent to a process they cannot be canceled.

These invariants can be implemented in a protected way by using IPC to post signals. Processes receiving a signal will mark the signal pending when they receive the IPC. They will deliver the signal, when it is safe to do so. Processes can validate the sender of the signal from the environment id of the IPC request. For those signals that cannot be ignored under UNIX (i.e. SIGKILL), they will be sent as other signals. If no action is taken in part by the receiving process, then a process termination can be forced by deallocating the environment of the misbehaving process.

- Signals must be able to interrupt some potentially long lasting "system calls" (they are procedure calls in ExOS 1.0). The action of the system call will be aborted. For example, when reading from an empty pipe, the reader will block until some data is written to the pipe or a signal aborts the read.

Care must be taken when writing "system calls" to check for pending signals if the process is to potentially block for a long time. If a signal is pending and the process is waiting for data, any action will be unrolled if needed and the signal will be delivered. This guarantees the atomic property of system calls with respect to signals.

Chapter 5

Related Work

There has been previous work in the area of decentralizing shared protected state. The related work can be divided into two areas: The idea of library operating systems, which is the context of this thesis, and mechanisms for decentralizing shared protected state.

Library operating systems were first proposed by Anderson [2] in a two-page position paper. In that paper, Anderson proposes "an application-specific structure where as much of the operating system as possible should be pushed into runtime library routines linked in with each application." Although the paper presents the ideas, it does not present any platform or mechanisms to carry them about.

Microkernels like Mach [1], Chorus [15], and QNX [8] move many of the structures of traditional kernels into user-level servers. In contrast to the user-level servers discussed throughout this thesis, their user-level servers are privileged, and therefore hard to replace and processes are forced to use them. A closer approximation of the mechanisms described in this thesis was proposed by Maeda [10]. In this paper, system calls like *r*ecvfrom and *s*endto were handled directly by the process itself and only in exceptional cases, like when sockets are being shared, would they be handled by a server. This paper did not describe how this approach could be used for other operating system abstractions and missed an opportunity to handle more cases directly by the process. Finally, the SPIN operating system [3] also addresses the problem of protected sharing. SPIN exploits type-safe languages and protected environments to guarantee that state is only modified by specific code.

Chapter 6

Conclusion

Empirically centralized resource management can significantly hurt application performance and flexibility [3, 4]. To solve these problems, this thesis shows how to decentralize resource management using unprivileged library operating systems.

Unfortunately, while decentralization is has many advantages, it is can be difficult to implement well. To aid implementors, this thesis has made three main contributions:

1. It enumerates the different semantics and invariants that abstractions can provide and identifies stylized methods of enforcing these invariants in a decentralized setting, thereby easing the task of system implementors.

2. It enumerates the abilities provided by a centralized scheme, allowing a library operating system implementor to explicitly choose which abilities to duplicate and which to replace with decentralized mechanisms.

3. It presents a library operating system (ExOS 1.0) that makes these principles and observations concrete. ExOS 1.0 is not a toy system: it implements the bulk of a common Unix API and can run many complex applications (e.g., gcc, perl, vi, etc.).

While further work remains, this thesis demonstrates both that library operating systems are feasible and that building them does not require heroic efforts.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, July 1986.

[2] T.E. Anderson. The case for application-specific operating systems. In *Third Workshop on Workstation Operating Systems*, pages 92–94, 1992.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[5] D.R. Engler and M.F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1996*, pages 53–59, Stanford, CA, USA, August 1996.

[6] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. *Proceedings of the Winter 1997 USENIX Conference*, pages 1–17, January 1997.

[7] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[8] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April 1992.

[9] Howard, J.H. Kazar, M.L. Menees, S.G. Nichols, D.A. Satyanarayanan M. Sidebotham R.N., West, and M.J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[10] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, 1993.

[11] David Mazières and Frans Kaashoek. Secure applications need flexible operating systems. submitted, February 1997.

[12] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. *Proceedings of the San Diego USENIX Conference*, pages 279–294, 1996.

[13] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[14] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[15] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[16] Inc. Sun Microsystems. NFS: Network file system protocol specification. RFC: 1094, March 1989.

[17] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.