Transparently Interposing User Code at the System Interface

Michael Blair Jones

September 1992

CMU-CS-92-170

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

Part I

The Problem

Chapter 1
Introduction

1.1. Terminology

Many contemporary operating systems provide an interface between user code and the operating system services based on special ‘‘system calls’’. One can view the system interface as simply a special form of structured communication channel on which messages are sent, allowing such operations as interposing programs that record or modify the communications that take place on this channel. In this dissertation, such a program that both uses and provides the system interface will be referred to as a ‘‘system interface interposition agent’’ or simply as an ‘‘agent’’ for short.

1.2. Thesis Statement

The thesis of this dissertation is that a toolkit can be constructed that substantially increases the ease of interposing user code between clients and instances of the system interface by allowing such code to be written in terms of the high-level objects provided by this interface, rather than in terms of the intercepted system calls themselves. Providing an object-oriented toolkit exposing the multiple layers of abstraction present in the system interface provides a useful set of tools and interfaces at each level. Different agents can thus exploit the toolkit objects best suited to their individual needs. Consequently, substantial amounts of toolkit code are able to be reused when constructing different agents. Furthermore, having such a toolkit enables new system interface implementations to be written, many of which would not otherwise have been attempted.

Just as interposition is successfully used today to extend operating system interfaces based on such communication-based facilities as pipes, sockets, and inter-process communication channels, interposition can also be successfully used to extend the system interface. In this way, the known benefits of interposition can also be extended to the domain of the system interface.

1.3. An Informal Perspective

The following figures should help clarify both the system interface and interposition. Figure 1-1 depicts uses of the system interface without interposition. In this view, the kernel[1] provides all instances of the operating system interface. Figure 1-2 depicts the ability to transparently interpose user code that both uses and implements the operating system interface between an unmodified application program and the operating system kernel. Figure 1-3 depicts uses of the system interface with interposition. Here, both the kernel and interposition agents provide instances of the operating system interface. Figure 1-4 depicts more uses of the system interface with interposition. In this view agents, like the kernel, can share state and provide multiple instances of the operating system interface.

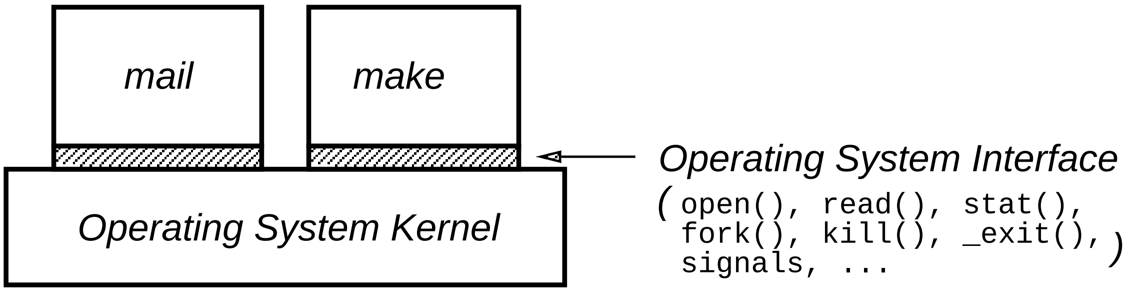

Figure 1-1: Kernel provides instances of system interface

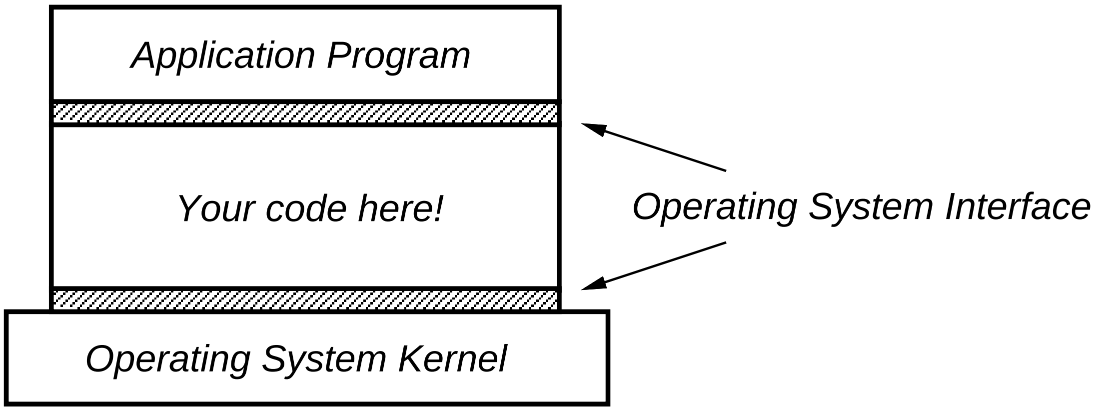

Figure 1-2: User code interposed at system interface

[1]The term ‘‘kernel’’ is used throughout this dissertation to refer to the default or lowest-level implementation of the operating system in question. While this implementation is often run in processor kernel space, this need not be the case, as in the Mach 3.0 Unix Server/Emulator [Golub et al. 90].

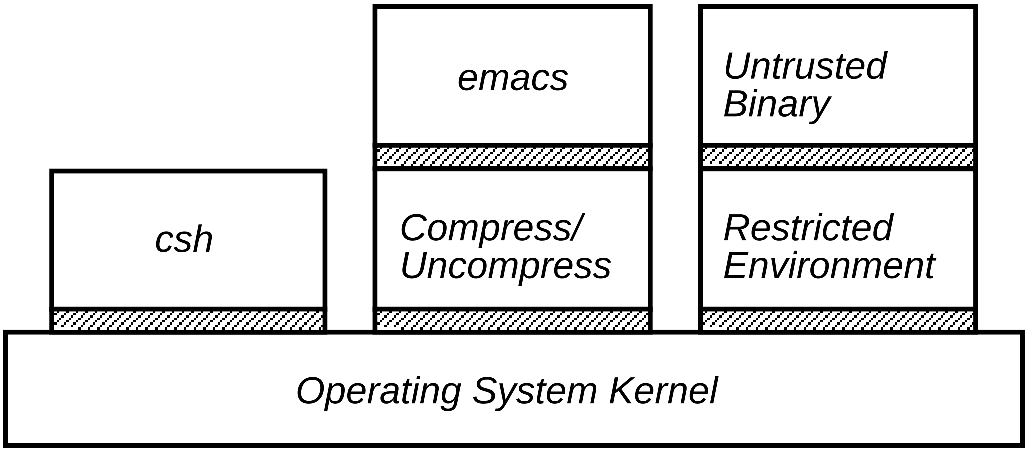

Figure 1-3: Kernel and agents provide instances of system interface

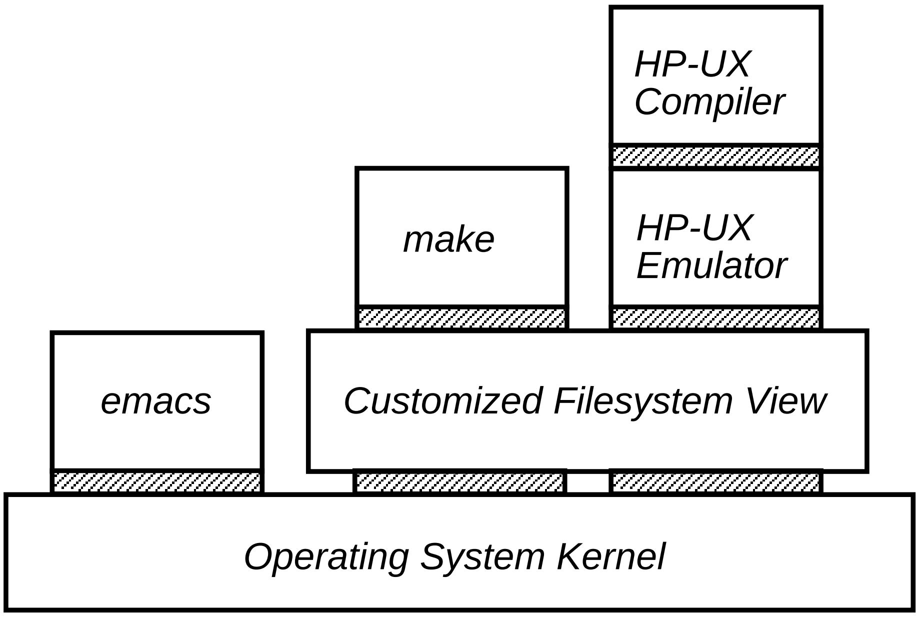

Figure 1-4: Agents can share state and provide multiple instances of system interface

1.4. Motivation

Today, agents are regularly written to be interposed on simple communication-based interfaces such as pipes and sockets. Similarly, the toolkit makes it possible to easily write agents to be interposed on the system interface.

Interposition can be used to provide programming facilities that would otherwise not be available. In particular, it can allow for a multiplicity of simultaneously coexisting implementations of the system call services, which in turn may utilize one another without requiring changes to existing client binaries and without modifying the underlying kernel to support each implementation.

Alternate system call implementations can be used to provide a number of services not typically available on system call-based operating systems. Some examples include:

- **System Call Tracing and Monitoring Facilities:** Debuggers and program trace facilities can be constructed that allow monitoring of a program's use of system services in a easily customizable manner.

- **Emulation of Other Operating Systems:** Alternate system call implementations can be used to concurrently run binaries from variant operating systems on the same platform. For instance, it could be used to run ULTRIX [Digital 89a, Digital 89b], HP-UX [Clegg et al. 86], or UNIX System V [AT&T 86] binaries in a Mach/BSD environment.

- **Protected Environments for Running Untrusted Binaries:** A wrapper environment can be constructed that allows untrusted, possibly malicious, binaries to be run within a restricted environment that monitors and emulates the actions they take, possibly without actually performing them, and limits the resources they can use in such a way that the untrusted binaries are unaware of the restrictions. A wide variety of monitoring and emulating schemes are possible from simple automatic resource restriction environments to heuristic evaluations of the target program's behavior, possibly including interactive decisions made by human beings during the protected execution. This is particularly timely in today's environments of increased software sharing with the potential for viruses and Trojan horses.

- **Transactional Software Environments:** Applications can be constructed that provide an environment in which changes to persistent state made by unmodified programs can be emulated and performed transactionally. For instance, a simple ''run transaction'' command could be constructed that runs arbitrary unmodified programs (e.g., `/bin/csh`) such that all persistent execution side effects (e.g., filesystem writes) are remembered and appear within the transactional environment to have been performed normally, but where in actuality the user is presented with a ''commit'' or ''abort'' choice at the end of such a session. Indeed, one such transactional program invocation could occur within another, transparently providing nested transactions.

- **Alternate or Enhanced Semantics:** Environments can be constructed that provide alternate or enhanced semantics for unmodified binaries. One such enhancement in which people have expressed interest is the ability to ''mount'' a search list of directories in the filesystem name space such that the union of their contents appears to reside in a single directory. This could be used in a software development environment to allow distinct source and object directories to appear as a single directory when running `make`.

1.5. Problems with Existing Systems

Increasing numbers of operating systems, e.g., Mach [Accetta et al. 86], SunOS version 4 [Sun 88a], and UNIX System V.4 [AT&T 89], are providing low-level mechanisms for intercepting system calls. Having these low-level mechanisms makes writing interposition agents possible. Nonetheless, they typically provide no higher-level tools or abstractions for effectively utilizing these mechanisms, making the use of such facilities unwieldy at best.

Part of the difficulty with writing system call interposition agents in the past has been that no one set of interfaces is appropriate across a range of such agents other than the lowest level system call interception services. Different agents interact with different subsets of the operating system interface in widely different ways to do different things. Building an agent often requires implementation of a substantial portion of the system interface. Yet, only the bare minimum interception facilities have been available, providing only the lowest common denominator that is minimally necessary. Consequently, each agent has typically been constructed completely from scratch. No leverage was gained from the work done on other agents.

1.6. Key Insight

The key insight that enabled me to gain leverage on the problem of writing system interface interposition agents for the 4.3BSD [Leffler et al. 90] interface is as follows: while the 4.3BSD system interface contains a large number of different system calls, it contains a relatively small number of abstractions *whose behavior is largely independent*. (These abstractions are such things as pathnames, descriptors, processes, process groups, files, directories, sockets, links, etc.) Furthermore, most calls manipulate only a few of these abstractions.[2]

Thus, it should be possible to construct a toolkit that presents these abstractions as objects in an object-oriented programming language. Such a toolkit would then be able to support the substantial commonalities present in different agents through code reuse, while also supporting the diversity of different kinds of agents through inheritance.

---

[2]For a detailed taxonomy of the use of these abstractions, see Appendix A.

1.7. Plan of Dissertation

This dissertation is structured as follows:

Part I presents an overview the problem addressed by this dissertation. Chapter 1 presents the thesis statement, some of the motivations for investigating it, and a key insight underlying the approach taken. Chapter 2 presents an overview of the approach taken in researching the topic. Chapter 3 presents an overview of other work related to the thesis topic.

Part II presents the design and implementation of the system interface interposition toolkit. Chapter 4 describes the high-level services implemented by the toolkit using several actual interposition agents to illustrate their use. Chapter 5 describes the low-level services implemented by the toolkit which are necessary for building interposition agents.

Part III presents the evaluation and results of this research. Chapter 6 presents an assessment of how well the design goals were met. Chapter 7 presents the results of a comparison to a best available equivalent implementation. Chapter 8 analyzes the Mach dependencies present in the low-level toolkit services and problems which were specific to the Mach and 4.3BSD interfaces. Chapter 9 analyzes the security implications of interposition on the system interface. Chapter 10 presents some lessons learned through this research which might benefit future interposition toolkit and agent writers. Chapter 11 discusses possible future work in this area. Chapter 12 summarizes my conclusions and outlines the contributions of this research.

Finally, Part IV contains the appendices and a list of other works referenced in this dissertation.

Chapter 2
Research Overview

2.1. Design and Structure of the Toolkit

I have designed and built a toolkit on top of the Mach 2.5 system call interception mechanism [Accetta et al. 86, Baron et al. 90, Golub et al. 90] that can be used to interpose user code on the 4.3BSD [Leffler et al. 90] system call interface. The toolkit currently runs on the Intel 386 [Intel 86]/486 [Intel 90] and the VAX [Digital 81b, Digital 81a].

This toolkit is structured in an object-oriented manner, allowing programs to be written in terms of several different layers of objects by utilizing inheritance. Abstractions exposed at different layers include such objects as pathnames, file descriptors, processes, signals, sockets, devices, etc., as well as the system calls themselves. The structure of the toolkit permits the agent to be written using whatever levels of abstraction are appropriate to the task it performs.

2.2. Design Goals

The four main goals that I attempted to achieve in building the toolkit were:

1. **Unmodified System:** Unmodified applications should be able to be run under agents. Similarly, the underlying kernel should not require changes to support each different agent (although the kernel may have to be modified once in order to provide support for system call interception, etc. so that agents can be written at all).

2. **Completeness:** Agents should be able to both use and provide the entire system interface. This includes not only the set of requests from applications to the system (i.e., the system calls) but also the set of upcalls that the system can make upon the applications (i.e., the signals).

3. **Appropriate Code Size:** The amount of new code necessary to implement an agent using the toolkit should only be proportional to the new functionality to be implemented by the agent — not to the size of the system interface. The toolkit should provide whatever boilerplate and tools are necessary to write agents at levels of abstraction that are appropriate for the agent functionality, rather than having to write each agent at the raw system call level.

4. **Performance:** The performance impact of running an application under an agent should be negligible.

2.3. Motivation for an Object-Oriented Toolkit

An underlying premise behind object-oriented programming is that when implementing similar functions it should be possible to extract the commonalities between them and share the implementations of the common functionality. The incremental work of building another similar function should then be proportional only to the differences. This approach pays off when substantial commonality exists between multiple logical functions. Some examples of such commonality are:

- For wide classes of interposition agents, much of the functionality needed to build each agent is also needed by other agents. Nearly all need translation from machine-specific system call numbers and argument formats to logical system call interfaces.
- Any agents manipulating `open()`ed objects need support for file (or socket) descriptors.
- Any agents manipulating pathnames need support for pathname component walking; many also need support for translating pathnames to open objects.
- Some facility for object reference counting or garbage collection is needed by most agents.
- Facilities for sharing objects among multiple client processes are needed by many agents, particularly those that allow changes made by one client process to be seen by another.

Not every agent needs to modify the behavior of the entire system interface. Indeed, to the extent that an agent does not modify the behavior of a particular portion of the system interface, it should be possible to pass uses of that portion through to the next level of system interface largely unmodified for execution. An object-oriented structure readily facilitates composing agents from just those toolkit components that will benefit them, while using inheritance to automatically access those functions that the agent has custom built.

Finally, an interposition toolkit should both be easier to use and provide more long term benefits than *ad hoc* approaches. One alternate approach to writing agents would be to always write one by copying another one and modifying it. This has several drawbacks. Such modifications tend to be of an undisciplined ‘‘whatever makes it work’’ nature; the toolkit approach helps maintain clean interfaces. Even for similar agents, any improvements made to one are not reflected back into the other; any improvements made to toolkit objects are shared by all clients. Likewise, as new agents are built with the toolkit, any new useful objects developed can be added to the toolkit suite, improving its usability over time; such accumulation of useful tools would be less likely with an *ad hoc* approach.

2.4. Using the Toolkit to Build Applications

As I built the toolkit, I also used it to implement several interposition agents. These agents provide:

- **System Call and Resource Usage Monitoring:** This demonstrates the ability to intercept the full system call interface.
- **User Configurable Filesystem Views:** This demonstrates the ability to transparently assign new interpretations to filesystem pathnames.
- **File Reference Tracing Tools** that are compatible with existing tools [Mummert & Satyanarayanan 92] originally implemented for use by the Coda [Satyanarayanan et al. 90, Kistler & Satyanarayanan 92] filesystem project: this provides a basis for comparing a best available equivalent implementation to a facility provided by an agent.

These and other agents are discussed in Section 4.2 and Appendix C.

Chapter 3
Related Work

This chapter presents an overview of past work providing the ability to interpose user code at the system interface or to otherwise extend the functionality available through the system interface. This topic does not appear to be well described in the literature; despite intensive research into past systems I have been unable to find a comprehensive treatment of the subject.

In particular, no general techniques for building or structuring system interface interposition agents appear to have been in use, and so none are described. Even though a number of systems provided mechanisms by which interposition agents could be built, the agents that were built appear to have shared little or no common ground. No widely applicable techniques appear to have been developed; no literature appears to have been published describing those *ad hoc* techniques that were used.

Thus, the following treatment is necessarily somewhat anecdotal in nature, with some past interposition agents and other system extensions described only by personal communications. Nonetheless, this chapter attempts to provide a representative, if not comprehensive, overview of the related work. It describes a number of past and present systems that provide mechanisms for extending the functionality of the system interface, with a particular emphasis on those where the potential extension mechanism was actually used to build useful extensions to the services provided via the system interface.

The related systems presented are grouped into rough categories by the particular low-level mechanism through which the system interface may be extended. Despite the apparent differences between the mechanisms provided by different systems, similar higher-level techniques can actually be used with a variety of low-level mechanisms. (The hiding of the differences between low-level system interface interposition mechanisms by the interposition toolkit is described in Section 4.2.2.) Thus, while the different techniques presented below have often been thought of as separate, a closer inspection will reveal that they are often overlapping, and in fact can be encompassed within in a common framework provided by a higher-level interposition toolkit.

3.1. Message-Based Systems

Message-based systems commonly allow applications to transparently intercept and modify communication occurring across an interface. This is typically accomplished either by establishing alternate bindings for the interface's communication channels at bind time or by transparently interposing an active agent on an existing communication channel. For instance, the Accent [Rashid & Robertson 81] and V [Cheriton 88] systems regularly used both of these approaches to good effect.

3.2. System Call-Based Systems

There are a large number of system call-based interfaces in existence. Some advantages often cited for this approach include the controlled crossing of operating system protection boundaries and the potential efficiency of implementation. Yet, just as it is useful to interpose active agents on message-based interfaces, the same benefits can be made available to system call-based interfaces as well. Indeed, in considering the relative merits of message-based and system call-based systems in the context of implementing process migration for the Sprite [Ousterhout et al. 88] operating system, Douglis and Ousterhout reached the conclusion that the two approaches are largely equivalent [Douglis & Ousterhout 91]. Likewise, this thesis demonstrates that interposition can equally well be applied to both message-based and system call-based systems, despite previous conventional wisdom to the contrary.

A large number of system call-based systems have allowed application code to implement system calls. Some examples are:

- OS6 [Stoy & Strachey 72a, Stoy & Strachey 72b] was a single-user system for the Modular One computer implemented in BCPL [Richards 69a, Richards 69b]. Separately compiled code segments communicated with one another by making indirect calls through the BCPL ‘‘global vector’’. Some slots in the global vector were reserved by convention for calls to system functions. However, programs could supply new values for these slots and serve as ‘‘the system’’ for subordinate programs. Furthermore, any routine could either be called via a normal procedure call, or could be ‘‘run’’, which involved protection against failure of the subordinate process.

  This capability of one program to serve as ‘‘the system’’ for other programs was used in OS6 to implement several enhancements to the base system, including a multi-terminal file editing system and a batch processing job queue. Both were run as normal applications on the base system [Stoy 92]. Other simpler enhancements to the base system implementation such as replacing the heap storage allocation routines with versions that also kept statistics were likewise implemented in this manner.

- The CAL TSS [Sturgis 74] system for the CDC 6400 was implemented as a set of layers with each layer being implemented by a program that ran on the virtual machine implemented by the previous layer. Each layer could provide ‘‘virtual instructions’’ and objects manipulated by those instructions, which it would interpret for subsequent layers. Instructions not provided by a given layer would be interpreted by the next layer.

This layered implementation technique was used pervasively in the construction of CAL TSS. It was used both for implementing new services and for implementing ‘‘ghost’’ versions of existing services for debugging purposes.

- The TENEX system [Bobrow et al. 72] (which was later released by DEC as TOPS-20 [Digital 78]) allowed superior processes to intercept system calls on behalf of inferior processes via the `tfork` JSYS [BBN 71, Thomas 75]. Intercepted system calls made by inferior processes would cause a software interrupt to occur in the intercepting superior processes.

  One major application of this facility was to implement both the RSEXEC [Thomas 73] and National Software Works [Forsdick et al. 78] Arpanet resource sharing environments that allowed participating TENEX systems to share files with one another via the Arpanet. Other applications included system call monitoring facilities, a restricted environment used during on-line student programming examinations that prevented accesses to other student’s files which could have potentially been used for cheating [Wohl 90], and even a monitor crash dump debugger that allowed programs to be run in an environment emulating that of the crashed monitor [Schilit 90]. The TENEX interception facilities had a relatively large number of applications written for them compared to other systems providing system call interception facilities, despite the fact that such applications were typically coded in assembler [Wohl 90].

- The MIT Incompatible Timesharing System (ITS) [Eastlake et al. 69] for the PDP-6 and PDP-10 provided a system call interception facility. This was used both to emulate TENEX/TOPS-20 and for trapping system calls in the debugger [Macrakis 90].

- The IBM Virtual Machine (VM) operating system for the System 370 [Parmelee et al. 72] allows for software emulation of a virtual 370. This has been used to run multiple virtual operating systems on a single physical processor.

  Unlike other systems that support interception of relatively high level calls to operating system services, VM provides the ability to implement an operating system on top of a software emulation of the bare machine. Thus each operating system run on top of VM must implement all system services (e.g., file systems, processes, etc.) from scratch.

- The IBM MVS operating system [IBM 87] allows privileged applications to perform ‘‘SVC screening’’ on behalf of other applications. Unlike VM, which provides a virtual copy of the bare machine, MVS provides a means of intercepting selected high-level system services invoked via the `SVC` instruction. This has been used to emulate pieces of somewhat different IBM operating system environments on top of MVS [Clayton 90].

- The Dartmouth Time Sharing System (DTSS) [Koch & Gelhar 86] for the GE 635 provided a facility called ‘‘`squeeze`’’ for intercepting system calls. Calls made from a given range of virtual addresses (in the range that was "squeezed") would cause software interrupts in another portion of the virtual address space. This was used in building a program debugger for DTSS, and for emulating an older GE/Honeywell batch operating system called GCOS [Colvin 90].

- UNIX System V.4 [AT&T 89] provides `/proc` operations for intercepting system calls. Execution of an intercepted system call causes the target process to stop, at which point it can manipulate the target process using other `/proc` operations. Thus far this facility has only been used in debuggers and to build a system call monitoring tool called "`truss`" [Gomes 90].

- SunOS version 4 [Sun 88a] provides a similar `ptrace()` operation. Like the System V `/proc` facility, it causes a program that makes an intercepted system call to stop, at which point the stopped program can be manipulated by another process using other `ptrace()` operations. This facility has been used in debuggers and to implement a system call monitoring program called "`trace`" [Gingell 90].

- Interposition is used by a number of Macintosh [Apple 88] applications to enhance or modify the behavior of the Macintosh operating system. Mac applications intercept system calls by replacing entries in the operating system's system call dispatch table with addresses for application-provided routines, usually saving the old values to be used internally and possibly restored later.

  Applications of interposition in the Macintosh range from the very simple to the fairly complex. As a simple example, some Mac applications built with MacApp [Apple 90] use interposition to temporarily override the behavior of cursor setting functions such that a "work in progress" cursor will be displayed. As a more complex example, FileSaver, one of the Norton Utilities for the Macintosh [Norton 90] intercepts all the file manipulation system calls in order to support an "UnErase" file operation. DiskDoubler [Salient 91] reimplements all filesystem operations to provide transparent filesystem data compression. Another application, Super Boomerang [Now 90], supplements the "Open..." file selection menu used by many applications to include file status information in addition to the usual file names. As a final example, the SAM [Symantec 91a] anti-virus software intercepts destructive file operations made by applications, informing the user of any suspicious behavior and asking for confirmation before allowing them to proceed.

- Interposition is widely used by MS-DOS [Microsoft 91] applications both to provide new implementations of existing MS-DOS facilities and to extend the system interface by providing new facilities through new system calls. As in the Macintosh, MS-DOS applications intercept system calls by replacing system trap dispatch table entries with the addresses of new trap handler routines, which themselves may make use of the old trap handlers.

  Applications of interposition in MS-DOS range from the very simple to the *extremely* complex. At the simple end, the Doskey [Microsoft 91] program reimplements keyboard reading calls to provide a command line editor and command history. At the other extreme, Microsoft Windows [Microsoft 87] interposes on the entire set of user interface calls, providing new implementations of them for the Windows environment, as well as providing a large set of new calls not otherwise available. Others provide more constrained services. For instance, Stacker [Stac 92] reimplements the filesystem operations to provide transparent filesystem data compression. The eXtended Memory Specification (XMS) implemented by the `HIMEM.SYS` driver specifies a new set of calls for accessing memory with addresses above 1024K [Duncan 90]. Similarly, the DCA/Intel Communicating Applications Specification [DCA/Intel 91] specifies a set of conventions for applications that provide or use a new set of calls supporting facsimile communication. Finally, both the Norton AntiVirus [Symantec 91b] and PC-cillin [Trend 90] anti-virus applications intercept destructive file operations made by applications, informing the user of any suspicious behavior and asking for confirmation before allowing the operations to proceed.

3.3. Relinking and Recompilation

A number of systems have been built that provide alternate system interface semantics for programs that have been relinked or recompiled. Some examples are:

- The Multics system [Organick 72] provided dynamic program linking facilities, which could be used to provide alternate versions of system services at run time.
- The Newcastle Connection [Brownbridge et al. 82] made multiple UNIX systems appear to be a single system to programs that had been statically linked with an alternate system call library.
- Three modern UNIX systems, OSF/1 [OSF 90], SunOS version 4 [Sun 88a], and UNIX System V.4 [AT&T 89] provide dynamic linking facilities, which may be used to tailor the system services used by a program at run time.
- Some thread systems are implemented as libraries that effectively provide new instances of the system interface for each thread. This may be done either to implement threads on top of a single-threaded system interface or to multiplex multiple library-level threads on top of a potentially different number of threads provided by the underlying system. For instance, versions of the Mach C Threads [Cooper & Draves 88] package utilize both schemes.

3.4. Interposition on Subset Interfaces

A number of other systems provide for partial system call interception or related facilities. The most common type allows for interception of filesystem operations only. Some examples are:

- The NFS filesystem interface [Walsh et al. 85, Sandberg et al. 85, Sun 86] has been used to construct active agents that intercept filesystem operations. The Sun NSE source control system [Sun 88b] is a good example of an agent utilizing the NFS filesystem mechanism to interpose agents providing new filesystem semantics.

- The ITOSS system [Rabin & Tygar 87] allowed executable processes called ‘‘sentinels’’ to dynamically extend the security model of a 4.2BSD system [Joy et al. 83]. A sentinel is invoked to make access control decisions whenever an access is attempted to a file which is guarded by the sentinel.

- The Watchdogs system [Bershad & Pinkerton 88] provided a mechanism that allowed application processes to implement user-defined filesystem semantics. It has been used to implement file compression and `biff` style file watching.

- The Taos operating system [McJones & Swart 87] for the Firefly [Thacker et al. 87] provides a mechanism for dynamically adding new filesystems implementations. System calls referencing files implemented by these filesystems are translated into RPC calls on corresponding user space file servers. This capability has been used to implement direct access to file versions stored in the Vesta configuration management system [Swart 92, Brown 92]. Several other systems provide similar services.

  Taos also provides a different facility that can be used to interpose on all pathname references. This allows programs to both record and/or modify all pathname references made by other programs [Swart 92].

3.5. Operating System Structuring Work

A number of other efforts have focused on different approaches to structuring operating system implementations than the monolithic kernel approach. These typically seek to provide a collection of logically distinct services implemented as a collection of servers. These interfaces taken together may then be viewed as the ‘‘system interface’’. The Accent [Rashid & Robertson 81] system took this approach.

Daniel Julin at CMU is currently investigating construction of generic operating system components that can be configured together to support a variety of different operating system interfaces and environments under Mach [Julin et al. 91, Guedes & Julin 91]. In many ways, his work is complementary to my own. His research investigates means of structuring and implementing flexible operating systems; my research provides a means of providing diverse implementations of operating system features to existing programs. Existing 4.3BSD programs could use agents constructed with my toolkit as an interface to operating system services provided as part of Julin’s work.

Finally, a related body of operating system structuring research has attempted to explore new designs for operating systems and operating system interfaces that are well suited for extensibility. Such systems as Choices [Campbell et al. 87] and Lipto [Druschel et al. 92, Druschel et al. 91] have explored using object-oriented interfaces and implementations to facilitate providing flexible and extensible operating system services. The interposition toolkit uses similar techniques to attempt to provide comparable flexibility and extensibility for an existing operating system interface as these systems do for new system interfaces.

3.6. Overview of Past Agents and Conclusions

A large number of systems have provided low-level facilities sufficient to interpose user code at the system interface. Both the number and types of interposition agents that have been built using these facilities have varied widely between the different systems. Those agents that have been built can be broken down into five somewhat overlapping categories:

1. Complete operating system emulations such as VM emulating OS/360 or TSO, TENEX emulating TOPS-10, and RSEXEC emulating an Arpanet-wide TENEX.
2. Debuggers and debugging facilities, such as those for CAL TSS, DTSS, ITS, and SunOS.
3. System call trace facilities, such as those for TENEX, UNIX System V.4, and SunOS version 4.
4. Adding new facilities to the operating system interface, as was done in OS6, CAL TSS, and MS-DOS.
5. Providing enhanced implementations of the existing operating system interface (often enhanced filesystem implementations), as was done in CAL TSS, TENEX, the Newcastle Connection, NFS, ITOSS, Watchdogs, Taos, and particularly in the Macintosh and MS-DOS.

Each of these interposition agents was constructed by hand; almost no code was reused. In particular, whatever boilerplate code was necessary in order to intercept, decode and interpret calls made to the raw system interface typically had to be constructed for each agent. Whatever levels of abstraction that were necessary in order to build each agent were typically constructed from scratch.

Nonetheless, despite these difficulties, a number of applications of interposition have been built, taking advantage of the apparent flexibility and configurability provided by utilizing a layered approach to system implementation. In particular, the fact that today people pay real money for interposition agents that provide enhanced implementations of operating system interfaces (see [Norton 90, Salient 91, Now 90, Symantec 91a, Microsoft 87, Stac 92, Symantec 91b, Trend 90] to name just a few) appears to validate the claim that interposition can be a useful and effective system building paradigm.

This dissertation presents a toolkit that attempts to simplify the construction of interposition agents. It does this by providing multiple structured views of the objects present in the system interface, allowing agents to be written at whatever levels of abstraction are appropriate to their particular functions, leveraging off of existing toolkit code whenever possible. While the toolkit itself is specific to a particular system interface, many of the techniques used should be applicable to other interfaces as well, as discussed in Section 10.4.

Part II

Design and Implementation

Chapter 4
Description of the Toolkit

4.1. Toolkit Overview

Different interposition agents need to affect different components of the system call interface in substantially different ways and at different levels of abstraction. For instance, a system call monitoring/profiling agent needs to manipulate the system calls themselves, whereas an agent providing alternate user filesystem views would prefer to manipulate pathnames and possibly file descriptors. I believe that the failure to provide such multi-layer interfaces by past system call interception mechanisms has made them less useful than they might otherwise have been.

The base layer of the toolkit handles intercepting the system calls themselves. Such operations as monitoring system call usage are done at this level. These facilities are discussed in Chapter 5.

The second layer of the toolkit is structured around the primary objects provided by the system call interface. In 4.3BSD, such objects include pathnames, file descriptors, pids, and process groups. Such operations as pathname transformations, filesystem usage monitoring, and process usage monitoring are done at this level.

A third set of toolkit layers focuses on secondary objects provided by the system call interface, which are normally accessed via primary objects. Such objects include files, directories, symbolic links, devices, pipes, and sockets. Operations that are specific to these secondary objects such as file encryption, directory transformations, etc. are done by these layers.

The toolkit is implemented in C++ [Stroustrup 87] with small amounts of C [Kernighan & Ritchie 78] and assembly language as necessary. This allows the benefits of object-oriented programming to be brought to bear. In particular, an object-oriented structure is useful given that multiple different implementations of similar objects are present in most interposition agents.

4.2. The Toolkit Layers

The following section illustrates the different layers of the toolkit through examples of their uses in specific agents. These layers are listed below in order of increasing levels of abstraction:

- Program Loader
- System Call Interception, Numeric, Symbolic
- Symbolic Trace
- Descriptor, Open Object, Pathname, Directory

These layers are presented through examples in the following section.

4.2.1. Program Loader

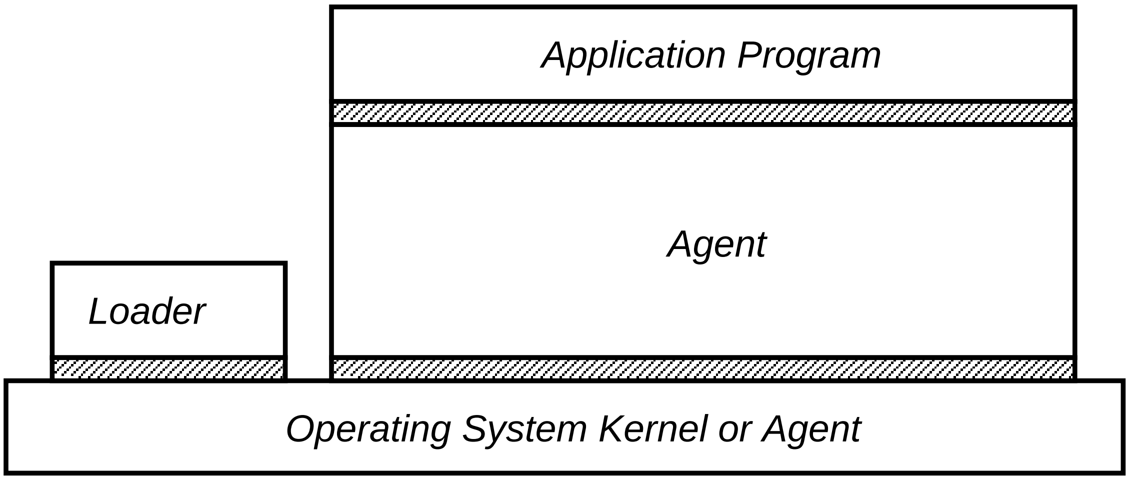

Figure 4-1: Loader used to load agent, application images

The first code that must be implemented in order to run an application under an interposition agent is a program loader. This is used for two things:

1. It is used by an initialization program to load the executable agent image into memory.
2. It is used by the agents to load executable application images into memory.

A loader must be written primarily because of an assumption present in the 4.3BSD interface that the operating system loads all program images into memory; the loader that is part of the `execve()` system call can not be used separately. Since agents need to able to load program images without doing a complete exec, a loader had to be provided by the toolkit. The loader is discussed further in Section 5.4. Figure 4-1 illustrates the use of the program loader by an initialization program that loads an agent image into memory, which in turn loads an application program image.

4.2.2. System Call Interception, Numeric, Symbolic Layers

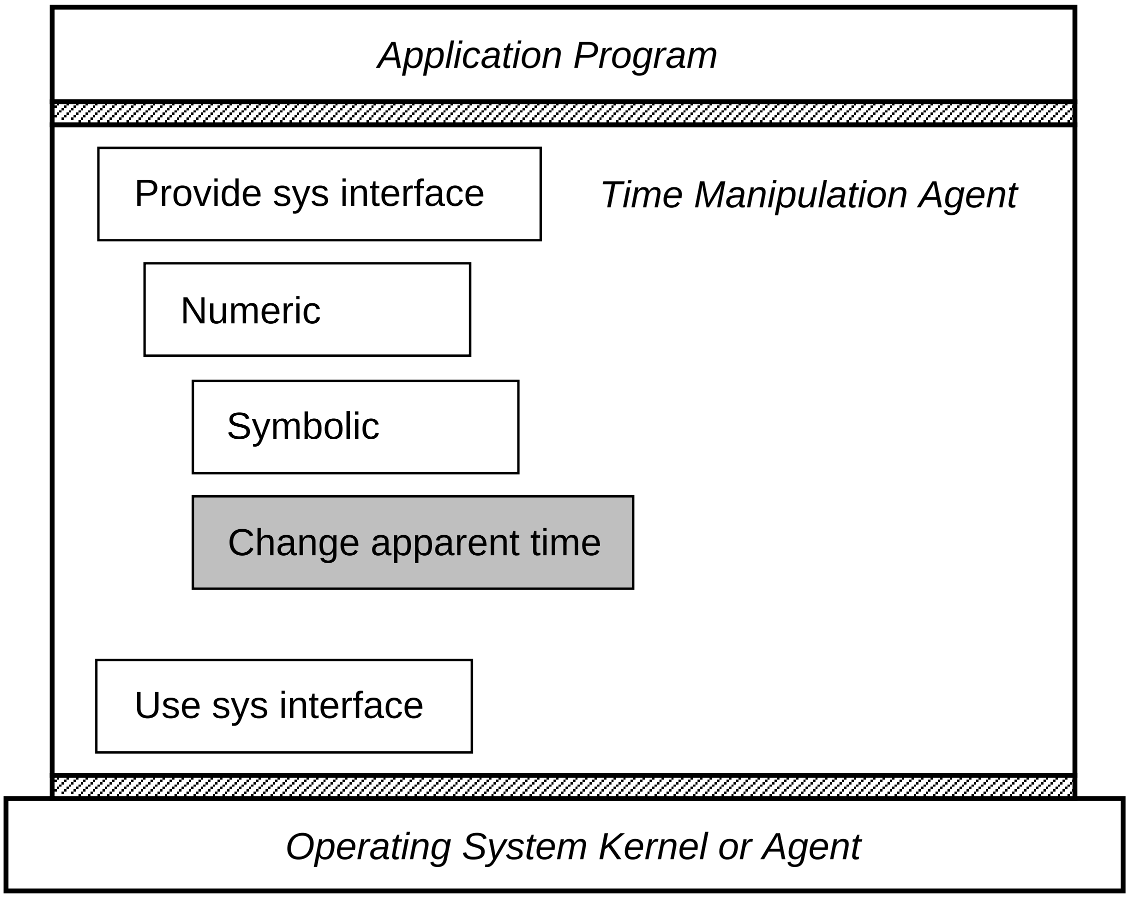

Figure 4-2: Apparent time of day manipulated using symbolic toolkit layer

Figure 4-2 presents the internal structure of a simple agent that changes the apparent time of day. In this and subsequent illustrations, unshaded boxes within the agent represent code provided by the toolkit and shaded boxes represent code that is specific to the particular agent. The structure of these example agents will be explored in order to illustrate the layers provided by the interposition toolkit.

The lowest-level layers provided by the toolkit are those that intercept calls made on the system interface and those that make calls on the next level of system interface. These layers hide the particular mechanisms used to intercept system calls and signals, those that are used to call down from an agent to the next level system interface, and those that are used to send a signal from an agent up to the application program. This layer also hides such details as whether the agent resides in the same address space as the application program or whether it resides in a separate address space. These facilities are discussed further in Sections 5.2, 5.3, and 5.5. These layers are indicated in Figure 4-2 by the boxes labeled ‘‘Provide sys interface’’ and ‘‘Use sys interface’’. For a presentation of the primary interfaces provided by the base toolkit layer, see Section B.1.

The next layer provided by the toolkit views the system interface as consisting of vectors of untyped numbers. At this layer, for instance, a `read()` system call request is represented as a vector of four numbers: a ‘‘3’’ for ‘‘`SYS_read`’’, plus three other numbers corresponding to the descriptor, the buffer address, and a count. This layer is indicated in Figure 4-2 by the box labeled ‘‘Numeric’’. While the system interface can be completely described by these sets of numbers, this is not a particularly useful level of abstraction at which to write interposition agents. For a presentation of the primary interfaces provided by the numeric system call toolkit layer, see Section B.2.

The next layer provided by the toolkit views the system interface as consisting of a set of typed virtual functions (or ‘‘methods’’ if you prefer the Smalltalk [Ingalls 80] terminology) with typed arguments on a system interface object. This layer is indicated in Figure 4-2 by the box labeled ‘‘Symbolic’’. It is the first that provides sufficient abstraction to make it easy to write simple interposition agents. For a presentation of the primary interfaces provided by the symbolic system call toolkit layer, see Section B.3.

Indeed, using the symbolic system call layer, it is possible to change the apparent time of day by implementing a derived version of the symbolic system call class that overrides the `gettimeofday()` function, replacing it with one that provides agent-specific behavior. The new `gettimeofday()` routine, plus initialization code, are all the agent-specific code then necessary to implement the simple time manipulation agent in Figure 4-2. For examples of actual code from this agent, see Section D.1.

4.2.3. Symbolic Trace Layer

Figure 4-3 presents an agent that traces the execution of client processes, printing each system call made and each signal received by a program. As before, unshaded boxes within the agent represent code provided by the toolkit and shaded boxes represent code that is specific to the particular agent. For examples of actual code from this agent, see Section D.2.

The tracing agent is implemented by a derived version of the ‘‘Symbolic’’ system call class that performs system call tracing. This derived class is indicated in Figure 4-3 by the box labeled ‘‘Trace’’. As well as being used to provide a system call tracing agent, this derived version of the symbolic system call class can be inserted into the class hierarchies of other more complex agents that are derived from the symbolic system call class. Thus, tracing versions of these agents can also be built through simple class

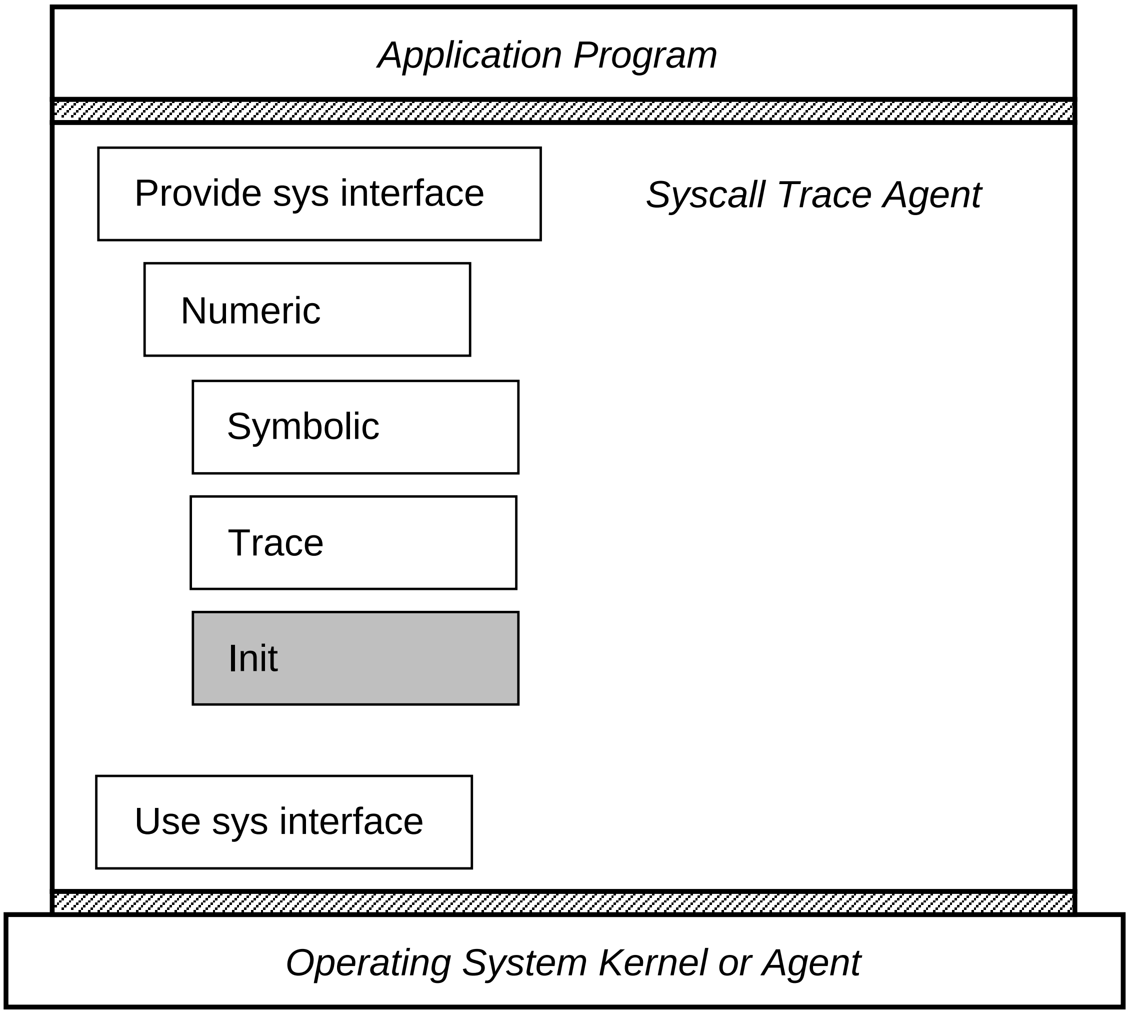

Figure 4-3: Tracing implemented as derived form of symbolic layer

composition. For instance, a version of the union directory agent (described in Section 4.2.4) that also traces all system calls was easily produced. This was done by building a new version of the union directory agent identical the original version except that its class hierarchy was modified by inserting the tracing form of the symbolic system call class directly above the symbolic system call class normally used by the agent. This capability has proven to be particularly useful when debugging new agents.

4.2.4. Descriptor, Open Object, Pathname, Directory Layers

Figure 4-4 presents an agent that implements an abstraction called a union directory [Korn & Krell 90, Hendricks 90]. Union directories provide the ability to view the contents of lists of actual directories as if their contents were merged into single ‘‘union’’ directories.

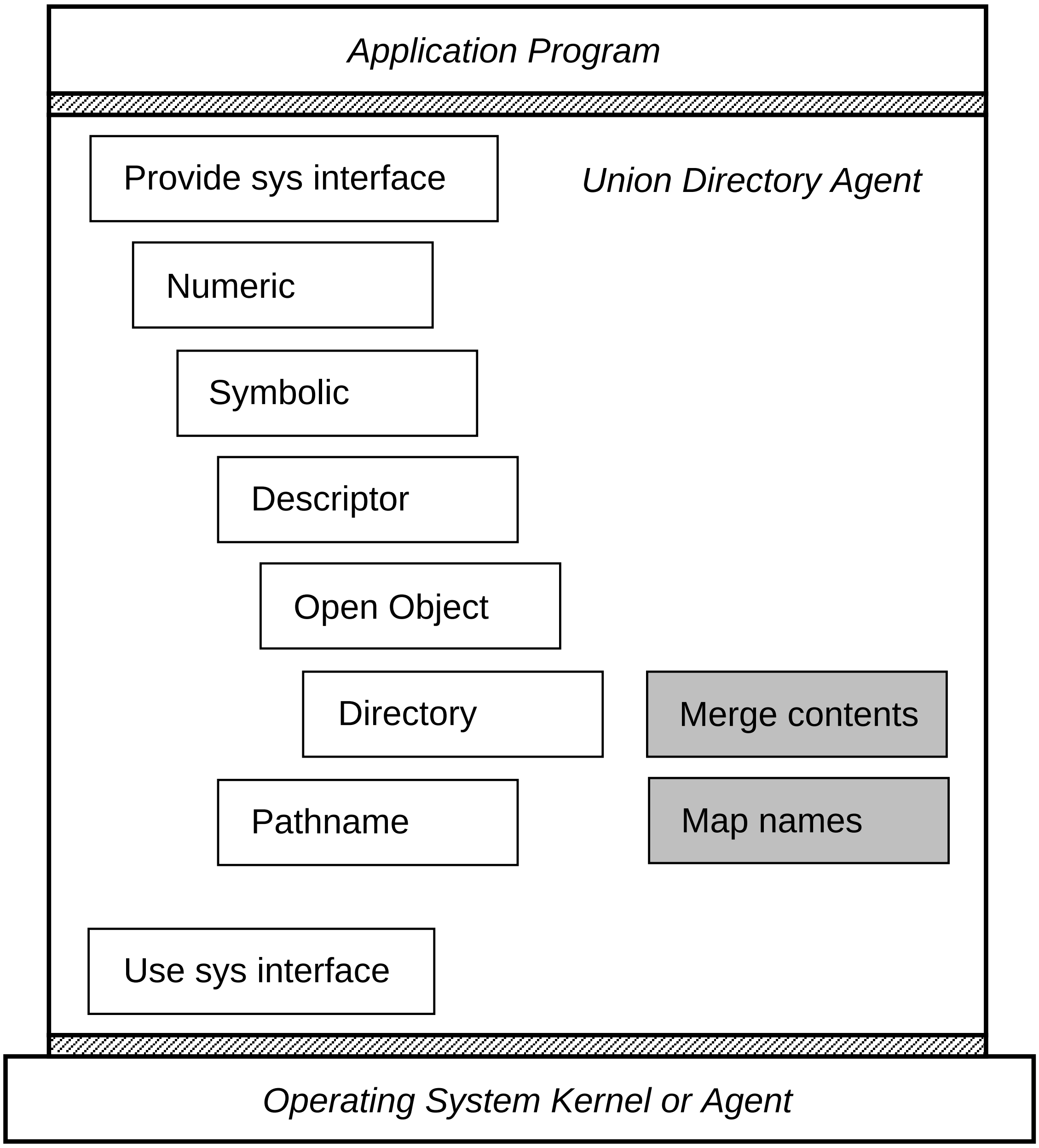

Figure 4-4: Union directories use derived forms of pathname, directory objects

The union directory agent illustrates uses of several toolkit layers that provide object-oriented views of particular abstractions present in the system interface. These layers make it possible to write derived classes that directly modify the behavior of these abstractions themselves, rather than having to write new versions of every system call that uses the abstraction in order to accomplish the same effect.

The two main system interface abstractions whose behavior is modified by the union directory agent are pathnames (indicated by the ‘‘Pathname’’ box in Figure 4-4) and file descriptors (indicated by the ‘‘Descriptor’’ box in Figure 4-4). The behavior of pathnames needs to be augmented since it must be possible to have names for a union directories. Similarly, the behavior of descriptors needs to be augmented since a union directory can be opened, and so it must be possible to have a descriptor for union directories. Neither of these capabilities are implemented by the base system; both are implemented by agent code that is specific to the union directory agent (indicated by the shaded boxes in Figure 4-4). For a presentation of the primary interfaces provided by the descriptor management and pathname management toolkit layers, see Sections B.4 and B.5, respectively.

Two other more specialized abstractions found in the toolkit are also used by the union directory agent. The first is reference-counted open objects (indicated by the ‘‘Open Object’’ box in Figure 4-4). This layer does the bookkeeping necessary to support multiple references to the same object, such as those that are produced by `dup()`. The second specialized toolkit abstraction used by the union directory agent is the directory object. This object provides an implementation of open directories whose contents can be listed. The primary interfaces for these toolkit classes are also presented in Sections B.4 and B.5, respectively.

Given the toolkit objects described above, writing the union directory agent is relatively straightforward. Most of the work is actually done by the underlying system and by the default toolkit class implementations. Only the differences between the desired and the default behavior need to be written. In particular, only two small derived classes needed to be implemented for the union directory agent.

The first of these classes maps client pathnames to either pathnames for normal objects provided by the underlying system, or to references to union directory objects implemented by the agent. This class is used by `open()` and other calls that use pathnames. It is indicated by the ‘‘Map names’’ box in Figure 4-4.

The second of these constructs the logical contents of a union directory from the actual contents of a list of underlying directories. This class is used by `getdirentries()` and related calls. It is indicated by the ‘‘Merge contents’’ box in Figure 4-4.

4.3. Toolkit Summary

The system interface interposition toolkit provides multiple views of the system interface with multiple levels of objects that correspond to the abstractions present in the system interface. This allows the behavior of these abstractions to be modified by implementing derived versions of these classes. Agents can be written using those toolkit components that are appropriate to the particular agent functions.

The low-level toolkit layers upon which the higher-level toolkit layers described in this chapter are built are described in Chapter 5. A complete list of toolkit modules and classes is presented in Appendix B.

Chapter 5
Boilerplate

This chapter describes the lowest layers of the toolkit. These layers perform such functions as agent invocation, system call interception, incoming signal handling, performing system calls on behalf of the agent, and delivering signals to applications running under agent code. Unlike the higher levels of the toolkit, these layers are sometimes highly operating system specific and also contain machine specific code. These low-level layers serve to hide these details, allowing agents to be written to more general and better behaved interfaces, leaving the details to the toolkit.

This chapter describes the low-level toolkit layers that are specific to the Mach 2.5 [Accetta et al. 86, Baron et al. 90] implementation of the 4.3BSD [Leffler et al. 90] system interface. Some aspects of these layers would differ when implemented for different operating systems that support different system interface interception mechanisms.

Different aspects of the toolkit boilerplate code will be described in subsequent sections.

5.1. Agent Invocation

Interposition agents built with the interposition toolkit are separately compiled programs that are statically linked to be loaded at a machine-specific location in high memory that is typically not used by applications. A special program called ‘‘`run`’’, which is part of the toolkit, is used to load an agent into a new address space, initialize its stack and arguments, and start it running.[3]

The arguments passed to an agent typically consist of the pathname of a program to run and the program’s arguments. If agent-specific arguments are also to be passed, they typically precede the program name, and are separated from the program name and arguments by a final agent-specific argument consisting of two dashes (‘‘--’’).

---

[3]For a discussion of the dependencies of `run` upon non-4.3BSD features, see Section 8.1.

After an agent has initialized itself, including parsing and acting upon any agent-specific arguments, it typically performs an emulated `execve()` call using the program name and arguments passed in by `run`. This runs the program using the system interface provided by the interposition agent.

Two examples of agent invocation are as follows:

```
run trace.agent csh
```

```
run timex.agent 'July 4, 1976 00:00' -- xclock -u 1
```

For examples of agents in use, see Appendix E.

5.2. System Call Interception

Interposition agents must be able to accept system calls made by the applications running on them. The interposition toolkit uses the Mach 2.5/3.0 system call interception facility called ''`task_set_emulation()`'' to redirect system calls made by applications to interposition agents. The `task_set_emulation()` call accepts an address space identifier[4], a system call number, and a handler routine address as parameters. It causes some state to be saved and control to be passed to the handler routine whenever the system call is made within the address space; child processes inherit a parent's system call emulation state.

The toolkit registers handlers for system calls implemented by interposition agents. These handlers save any necessary state, switch to an agent stack, and call the next level of toolkit services which collects the system call arguments and passes the saved state and system call information to code which interprets the system call. Once the system call has been executed and these services eventually return, the handler switches back to the application stack, restores any saved context information, and passes control back to the application.

5.3. System Call Invocation

An interposition must be able to use the system interface as well as provide it. Thus, it must be able to make system calls to the system interface implementation on which it is running, even if the call being made is also being intercepted by the agent.

[4]Mach address spaces are identified by ports and are known as ''tasks''.

The interposition toolkit uses a Mach 2.5 facility called "`htg_unix_syscall()`"[5] to make system calls on the kernel 4.3BSD implementation on behalf of agents. The `htg_unix_syscall()` call takes a system call number, a vector of system call arguments, and a vector for results as parameters and performs the system call, even if it is currently being intercepted with `task_set_emulation()`. It is like the indirect system call `syscall()`, except that it can be used for all system calls, including those with non-standard parameter passing mechanisms such as `wait()` and `sigcleanup()`.

Agents typically invoke services provided by the symbolic system call toolkit layer in order to make system calls down to the next layer. These are then passed down through the lower levels of the toolkit and are performed using `htg_unix_syscall()`.

Agents can also make system calls in the usual fashion using the system call stubs provided in the C library. Those which are not being intercepted by the agent go directly to the next level. Those which are being intercepted cause the agent to be recursively entered. The recursive entry is detected and the system call is performed using `htg_unix_syscall()`.

It was very convenient to be able to make normal system calls from within agents. This meant that agents could use normal library code whether or not the code happened to make any system calls. For instance, this allowed the normal standard output routines to be used from within the `trace` agent, instead of having to write new ones which avoided making any system calls.

5.4. Loading Applications

Programs are loaded and run under 4.3BSD using the `execve()` system call. This call clears the caller's address space, closes a subset of the descriptors, resets signal handlers, reads the program file, loads the executable image into the address space, loads the arguments onto the stack, sets the registers, and transfers control into the loaded image. No facility is provided which loads an executable image without first clearing the address space.

Since the agents reside in the same address spaces as the applications running on them[6], the `execve()` call must be reimplemented by every agent to allow the

---

[5]The acronym HTG, introduced by Doug Orr, stands for Honest To God.

[6]For a further discussion on co-resident agents, see Section 10.2.

application's portion of the address space to be reloaded, while maintaining the agent's portion. Also, since two other system calls (`ioctl()` and `fcntl()`) are used to supply implicit close-on-exec parameters to `execve()`, these calls must be reimplemented by all agents as well. Thus, the toolkit provides a user space implementations of `execve()`, `ioctl()`, and `fcntl()`, allowing application processes to be reloaded with new executable images, while preserving the agents sharing their address spaces. The same user space program loader is used by the `execve()` implementation as is used by the `run` program discussed in Section 5.1.

Several additional system calls are required in order to individually perform the many operations typically performed by `execve()`, such as clearing the process address space, reading a program image, closing descriptors, and resetting signal handlers. Unfortunately, these extra operations result in some performance loss relative to the kernel-based implementation[7].

5.5. Signal Handling and Delivery

Signals implicitly permeate the 4.3BSD interface. Many calls can implicitly cause signals to be synchronously generated. Many calls provide atomicity guarantees with respect to signals. Signals can interrupt blocked system calls. Returning from a signal handler called when a system call was interrupted can either restart the call or cause it to return an error code, depending upon a per-signal state flag. Another flag can cause signal handlers to be called on a different stack. In short, faithfully implementing any 4.3BSD system call in user space requires faithfully implementing the 4.3BSD signal mechanism in user space. The interposition toolkit provides a complete implementation of the 4.3BSD signal mechanism.

To do this, the toolkit intercepts the system calls which manipulate signal state, maintaining all application signal state within the toolkit. The toolkit also establishes handlers for all signals being handled by the application, allowing incoming signals to be coordinated with the rest of the signal implementation. For instance, this allows delivery of asynchronously arriving signals to be deferred while agent code is running.

Since the toolkit intercepts all signals handled by the application, it must also perform all signal delivery to the application. Thus, the toolkit provides an implementation of the code which pushes a signal frame and invokes a signal handler, as well as the system calls (`sigreturn()` and `sigcleanup()`) which pop a signal frame and return from a signal handler. The toolkit sets the signal action of any signal not handed by the

---

[7]For an analysis of low-level toolkit performance characteristics, see Sections 6.4.3 and 6.4.4.

application to match the action requested by the application, allowing the underlying system implementation to perform any resulting signal actions, such as suspending, resuming, or killing the process.

Many of the toolkit signal facilities are independently callable, allowing agents to tailor the behavior of the signal implementation, just as they are able to tailor the implementation of the system calls. For instance, this capability is used by the `trace` agent to print a trace message for each signal received.

5.6. Minimum Boilerplate Necessary

A moderate number of the 4.3BSD system calls must be reimplemented by all agents built on top of the Mach 2.5 system call interception and invocation facilities, as previously discussed in Sections 5.4 and 5.5. This is necessary in order to provide a faithful emulation of the 4.3BSD interface[8]. This section provides a comprehensive list and short discussion of these calls.

5.6.1. Process Management Calls

- `execve()`, `execv()`
- `fork()`, `vfork()`

The exec calls must be reimplemented in order to allow programs to be loaded without disturbing the agents which share their address spaces, as per Section 5.4. The fork calls must be reimplemented so that per-process agent state can be initialized for new processes.

5.6.2. Calls Supplying Hidden Exec Parameters

- `ioctl()`
- `fcntl()`

These miscellaneous control calls must be reimplemented because they supply implicit close-on-exec parameters to the exec calls via their `FIOCLEX` and `F_SETFD` subfunctions, respectively.

---

[8]For a further discussion on the degree to which the system interface can be faithfully emulated, see Section 10.1.

5.6.3. Address Space Management Calls

- `brk()`, `sbrk()`

The address space management calls are reimplemented because they are dependent upon state which is initialized by the exec calls. Specifically, the exec calls initialize the location of the address space ‘‘break’’ between allocated and unallocated memory. These calls fail to work properly when an exec call has not first been performed.

5.6.4. Signal Handling System Calls

- `sigvec()`
- `sigblock()`, `sigsetmask()`
- `sigpause()`
- `sigstack()`
- `sigreturn()`, `sigcleanup()`

The signal calls be reimplemented so as to make it appear that signal delivery only interrupts application code, as per section 5.5.

Part III

Evaluation

Chapter 6
General Results

This chapter presents an assessment of how well the toolkit design goals were met. As discussed in Section 2.2, these goals were:

1. Unmodified System
2. Completeness
3. Appropriate Code Size
4. Performance

Each of these goals will be discussed in turn.

6.1. Unmodified System

6.1.1. Unmodified Applications

Agents constructed using the system interface interposition toolkit can load and run unmodified 4.3BSD binaries. No recompilation or relinking is necessary. Thus, agents can be used for all program binaries — not just those for which sources or object files are available.

Applications do not have to be adapted to or modified for particular agents. Likewise, a general agent loader program (which is discussed in Section 5.1) is used to invoke arbitrary agents, which are compiled separately from the agent loader. Indeed, the presence of agents should be transparent to applications.[9]

[9] Of course, an application that is intent on determining if it is running under an agent probably can, if only by probing memory or performing precise performance measurements. This topic is discussed more fully in Section 9.2.

6.1.2. Unmodified Kernel

Agents constructed using the system interface interposition toolkit do not require any agent-specific kernel modifications. Instead, they use general system call handling facilities, which are provided by the kernel, in order to implement all agent-specific system call behavior.

The Mach 2.5 kernel used for this work contains a primitive that allows 4.3BSD system calls to be redirected for execution in user space. Another primitive permits calls to be made on the underlying 4.3BSD system call implementation even though those calls are being redirected. These facilities are discussed further in Sections 5.2 and 5.3.

6.2. Completeness

Agents constructed using the system interface interposition toolkit can both use and provide the entire 4.3BSD system interface. This includes not only the system calls, but also the signals. Thus, both the downward path (from applications to agents and from agents to the underlying system implementation) and the upward path (from the underlying implementation to agents and from agents to applications) are fully supported.

Completeness gives two desirable results:

1. All programs can potentially be run under agents. By contrast, if completeness did not hold, there would have been two classes of programs: those that used a restricted set of features that which could handle, and those that used features which agents could not handle. The interposition toolkit avoids these problems.

2. Agents can potentially modify all aspects of the system interface. Agents are not restricted to modifying only subsets of the system behavior. For instance, it would have been easy to envision similar systems in which agents could modify the behavior of system calls, but not incoming signals.[10]

---

[10] The treatment of signals is presented in Section 5.5.

6.3. Appropriate Code Size

Table 6-1 lists the source code sizes of three different agents, broken down into statements of toolkit code used, and statements of agent specific code.[11] These agents were chosen to provide a cross section of different interposition agents, ranging from the very simple to the fairly complex and using different portions of the interposition toolkit. Each of these agents is discussed in turn.

Sizes of Agents			
Agent Name	*Toolkit Statements*	*Agent Statements*	*Total Statements*
`timex`	2467	35	2502
`trace`	2467	1348	3815
`union`	3977	166	4143

Table 6-1: Sizes of agents, measured in semicolons

6.3.1. Size of the Timex Agent

The `timex` agent changes the apparent time of day, as discussed in Section 4.2.2. It is built upon the symbolic system call and lower levels of the toolkit. The toolkit code used for this agent contains 2467 statements. The code specific to this agent consists of only two routines: a new derived implementation of the `gettimeofday()` system call call and an initialization routine to accept the desired effective time of day from the command line. This code contains only 35 statements.

The new code necessary to construct the `timex` agent using the toolkit consists only of the implementation of the new functionality. Inheritance from toolkit objects is used to obtain implementations of all system interface behaviors that remain unchanged.

6.3.2. Size of the Trace Agent

The `trace` agent traces the execution of client processes, printing each system call made and signal received, as discussed in Section 4.2.3. Like the `timex` agent, it is built upon the symbolic system call and lower levels of the toolkit, which contain 2467 statements. However, the code specific to this agent is much larger, containing 1348 statements. The reason for this is simple: unlike the `timex` agent, the new work of the

[11] Note: The actual metric used was to count semicolons. For C and C++, this gives a better measure of the actual number of statements present in the code than counting lines in the source files.

`trace` agent is proportional to the size of the entire system interface. Derived versions of each of the 114 4.3BSD system calls (which are listed in Appendix A and summarized in Table A-1) plus the signal handler are needed to print each call name and arguments. The new code contains less than 12 statements per system call, 10 of which typically consist of:

- 1 routine declaration
- 1 variable declaration
- 2 calls to prepare for output (1 before and 1 after the system call)
- 2 calls to perform output (1 before and 1 after)
- 2 calls to flush output (1 before and 1 after)
- 1 return statement
- 1 call to make the system call itself

As with the `timex` agent, the new code necessary to construct the `trace` agent using the toolkit consists only of the implementation of the new functionality. Inheritance from toolkit objects is used to obtain implementations of all system interface behaviors that remain unchanged.

6.3.3. Size of the Union Agent

The `union` agent implements union directories, which provide the ability to view the contents of lists of actual directories as if their contents were merged into single ‘‘union’’ directories, as discussed in Section 4.2.4. It is built using toolkit objects for pathnames, directories, and descriptors, as well as the symbolic system call and lower levels of the toolkit. The toolkit code used for this agent contains 3977 statements. The code specific to this agent consists of three things: a derived form of the pathname object that maps operations using names of union directories to operations on the underlying objects, a derived form of the directory object that makes it possible to list the logical contents of a union directory via `getdirentries()` and related calls, and an initialization routine that accepts specifications of the desired union directories from the command line. Yet, this new code contains only 166 statements.

The new code necessary to construct the `union` agent using the toolkit consists only of the implementation of the new functionality. As with the other agents, inheritance from toolkit objects is used to obtain implementations of all system interface behaviors that remain unchanged.

6.3.4. Size Results

The above examples demonstrate several results pertaining the code size of agents written using the interposition toolkit. One result is that the size of the toolkit code dominates the size of agent code for simple agents. Using the toolkit, the amount of new code to perform useful modifications of the system interface semantics can be small.

Furthermore, the amount of agent specific code can be proportional to the new functionality being implemented by the agent, rather than proportional to the number of system calls affected. For instance, even though the union directory agent needs to change the behavior of all 30 calls that use pathnames, and all 48 calls that use descriptors, or 70 calls in all (eight of which use both)[12], it is written in terms of toolkit objects that encapsulate the *behavior of these abstractions*, rather than in terms of the system calls that use them. Thus, the agent specific code need only implement the new functionality since the toolkit provides sufficient underpinnings to make this possible.

Finally, there can be substantial code reuse between different agents. All the agents listed above were able to use the symbolic system call and lower levels of the toolkit, consisting of 2467 statements. Both the `union` and `dfs_trace`[13] agents, as well as others also described in Appendix C, are also able to use the descriptor, open object, and pathname levels of the toolkit, consisting of 3977 statements. Rather than modifying an implementation of the system interface in order to augment its behavior, the toolkit makes it possible to implement derived versions of the base toolkit objects, allowing the base toolkit objects that implement the system interface to be reused.

A more detailed look at toolkit and agent sizes is presented in Appendix F.

6.4. Performance

Sections 6.4.1 and 6.4.2 discuss the performance implications of running applications under interposition agents. Sections 6.4.3 and 6.4.4 discuss the performance of low-level operations used to implement interposition. For a further discussion of agent performance, see Section 7.5.

---

[12] See Appendix A and Table A-1 for details.

[13] The `dfs_trace` agent is described in Chapter 7.

6.4.1. Application Performance Data

This section presents the performance of running two applications under several different agents. The two applications chosen differ both in their system call usage and their structure: One makes moderate use of system calls and is structured as a single process; the other makes heavy use of system calls and is structured as a collection of related processes. Likewise, the agents chosen range from very simple to fairly complex. The results are discussed in Section 6.4.2.

6.4.1.1. Performance of Formatting This Document

Table 6-2 presents the elapsed time that it takes to format a preliminary draft of this document with Scribe [Reid & Walker 80] on a VAX 6250 [Digital 81b] both using no agent and when run under three different agents. In each case, the time presented is the average of nine successive runs done after an initial run from which the time was discarded.

This task requires 716 system calls. When run without any agents, it takes 131.5 seconds of elapsed time.

Format this document		
Agent Name	*Seconds*	*% Slowdown*
None	131.5	—
`timex`	132.0	0.5%
`trace`	135.0	2.5%
`union`	136.5	3.5%

Table 6-2: Time to format this document

When run under the simplest agent, `timex`, an additional half second of overhead is added, giving an effective additional cost of under one half percent of the base run time. When run under `trace`, an extra 3.5 seconds of overhead are introduced. Furthermore, when run under `union`, the most complex agent considered, there is only an additional 5.0 seconds, giving an effective agent cost of 3.5% of the base run time.

It comes as no surprise that `trace`, while conceptually simple, incurs perceptible overheads. Each system call made by the application to the `trace` agent results in at least an additional two `write()` system calls in order to write the trace output.[14]

[14] Trace output is not buffered across system calls so it will not be lost if the process is killed.

6.4.1.2. Performance of Compiling C Programs

Table 6-3 presents the elapsed time that it takes to compile eight small C programs using Make [Feldman 79] and the GNU C compiler [Stallman 90] on a 25MHz Intel 486 [Intel 90]. In each case, the time presented is the average of nine successive runs done after an initial run from which the time was discarded.

To do this, Make runs the GNU C compiler, which in turn runs the C preprocessor, the C code generator, the assembler, and the linker for each program. This task requires a total of 11877 system calls, including 64 `fork()`/`execve()` pairs. When run without any agents, it takes 16.0 seconds of elapsed time.

Make 8 programs		
Agent Name	*Seconds*	*% Slowdown*
None	16.0	—
`timex`	19.0	19%
`trace`	33.0	107%
`union`	29.0	82%

Table 6-3: Time to make 8 programs

When run under the simplest agent, `timex`, an additional three seconds of overhead are added, giving an effective additional cost of 19% of the base runtime. When run under `union`, which interposes on most of the system calls and which uses several additional layers of toolkit abstractions, the additional overhead beyond the no agent case is 13.0 seconds, giving an effective additional cost of 82% of the base runtime. When run under `trace`, an additional 17.0 seconds of run time are incurred, yielding a slowdown of 107%.

Again, it comes as no surprise that `union` introduces more overhead than `timex`. It interposes on the vast majority of the system calls, unlike `timex`, which interposes on only the bare minimum[15] plus `gettimeofday()`. Also, `union` uses several additional layers of implementation abstractions not used by `timex`.

As with the previous application, the larger slowdown for `trace` is unsurprising. Given the large number of system calls made by this application, the total performance becomes dominated by the output time required to write the trace log.

[15]For a discussion of the system calls that must be intercepted by each agent, see Section 5.6.

An analysis of low-level performance characteristics is presented in Sections 6.4.3 and 6.4.4. Performance of an agent that performs file reference tracing is discussed in Section 7.5.

6.4.2. Application Performance Results

The application performance data demonstrates that the performance impact of running an application under an agent is very agent and application specific. The performance impact of the example agents upon formatting this document was practically negligible, ranging from 0.5% for the `timex` agent to 2.5% for the `trace` agent. However, the performance impact of the example agents upon making the eight small C programs was significant, ranging 19% for `timex` to 107% for `trace`. Unsurprisingly, different programs place different demands upon the system interface, and different agents add different overheads.

The good news is that the additional overhead of using an agent can be small relative to the time spent by applications doing actual work. Even though no performance tuning has been done on the current toolkit implementation, the overheads already appear to be acceptable for certain classes of applications and agents.

Furthermore, the agent overheads are of a pay-per-use nature. Calls not intercepted by interposition agents go directly to the underlying system and result in no additional overhead[16].

Finally, even though some performance impact is clearly inevitable, presumedly the agent will have been used because it provides some benefit. For instance, agents may provide features not otherwise available, or they may provide a more cost-effective means of implementing a desired set of features than otherwise available. The performance ‘‘lost’’ by using an interposition agent can bring other types of gains.[17]

6.4.3. Micro Performance Data

This section presents the performance of several low-level operations used to implement interposition and of several commonly used system calls both without and with interposition. The results are discussed in Section 6.4.4.

---

[16]For a discussion of the system calls that must be intercepted by each agent, see Section 5.6.

[17]For a discussion on the tradeoffs of using interposition agents, see Section 12.3.

Table 6-4 presents the performance of several low-level operations used to implement interposition. All measurements were taken on a 25MHz Intel 486 running Mach 2.5 version X144. The code measured was compiled with gcc or g++ version 1.37 with debugging (`-g`) symbols present.

Performance of Low Level Operations	
Operation	*μsec*
C procedure call with 1 argument, result	1.22
C++ virtual procedure call with 1 argument, result	1.94
Intercept and return from system call	30
`htg_unix_syscall()` overhead	37

Table 6-4: Performance measurements of individual low-level operations

Table 6-5 presents the performance of several commonly used system calls both without interposition and when a simple interposition agent is used. The interposition agent, `time_symbolic`, intercepts each system call, decodes each call and arguments, and calls C++ virtual procedures corresponding to each system call. These procedures just take the default action for each system call; they make the same system call on the next level of the system (the instance of the system interface on which the agent is being run). This allows the minimum toolkit overhead for each intercepted system call to be easily measured. As before, these measurements were taken on a 25MHz Intel 486 running Mach 2.5 version X144; measured code was compiled with gcc or g++ version 1.37 with debugging (`-g`) symbols present.

Performance of System Calls			
Operation	*μsec with no agent*	*μsec with* `time_symbolic` *agent*	*μsec toolkit overhead*
`getpid()`	25	170	145
`gettimeofday()`	47	214	167
`fstat()`	54	220	166
`read()` 1K of data	370	579	209
`stat()` with 6 ufs name components	892	1101	209
`fork()`, `wait()`, and `_exit()`	10350	22350	12000
`execve()` with 6 ufs name components	9720	20000	10280

Table 6-5: Performance measurements of individual system calls

6.4.4. Micro Performance Results

Two times from Table 6-4 are particularly significant. First, it takes 30µsec. to intercept a system call, save the register state, call a system call dispatching routine, return from the dispatching routine, load a new register state, and return from the intercepted system call. This provides a lower bound on the total cost of any system call implemented by an interposition agent.

Second, using `htg_unix_syscall()`[18] to make a system call adds 37µsec. of overhead beyond the normal cost of the system call. This provides a lower bound on the additional cost for an agent to make a system call that otherwise would be intercepted by the agent.

Thus, any system call intercepted by an agent that then makes the same system call as part of the intercepted system call's implementation will take at least 67µsec. longer than the same system call would have if made with no agent present. Comparing the 67µsec. overhead to the normal costs of some commonly used system calls (found in Table 6-5) helps puts this cost in perspective.

The 67µsec. overhead is quite significant when compared to the execution times of simple calls such as `getpid()` or `gettimeofday()`, which take 25µsec. and 47µsec., respectively, without an agent. It becomes less so when compared to `read()` or `stat()`, which take 370µsec. and 892µsec., respectively, to execute in the cases measured without an agent. Hence, the impact will always be significant on small calls that do very little work; it can at least potentially be insignificant for calls that do real work.

In practice, of course, the overheads of actual interposition agents are higher than the 67µsec. theoretical minimum. The actual overheads for most system calls implemented using the symbolic system call toolkit level (see Section 4.2.2) range from about 140 to 210µsec., as per Table 6-5. Overheads for `fork()` and `execve()` are significantly greater, adding approximately 10 milliseconds to both, roughly doubling their costs. Some of the reasons for the higher `execve()` costs are discussed in Sections 5.4 and 5.6. While the current overheads certainly leave room for optimization (starting with compiling the agents with optimization on), they are already low enough to be unimportant for many applications and agents, as discussed in Section 6.4.2.

---

[18]For a discussion of `htg_unix_syscall()`, see Section 5.3.

Finally, it should be stressed that these performance numbers are highly dependent upon the specific interposition mechanism used. In particular, they are strongly shaped by agents residing in the address spaces of their clients. See Section 10.2 for a discussion on the possible implications of using other types of interposition mechanisms.

Chapter 7
Comparison to a Best Available Implementation

This chapter compares a best available implementation of a task that was implemented without benefit of the toolkit with an equivalent interposition agent constructed using the toolkit that performs the same task. In particular, this chapter compares a set of special purpose distributed file reference tracing tools known as DFSTrace [Mummert & Satyanarayanan 92], which were produced by Lily Mummert of the Coda [Satyanarayanan et al. 90, Kistler & Satyanarayanan 92] filesystem project, with an agent that was built using the system interface interposition toolkit to generate equivalent traces.

The DFSTrace file reference tracing tools were chosen as a basis of comparison for a number of reasons.

- They implement an extension to the behavior of the system interface. In particular, they add the ability to log use of portions of the system interface. Thus, this task should be able to be implemented using the system interface interposition toolkit.
- They are a real system in actual use. This provides a basis for comparison against a real system that was independently built, rather than just comparing toolkit agents against systems that were built only for purposes of comparison. Thus, a toolkit agent can be compared against a real system, rather than just a straw man implementation.
- They have a high quality of implementation. Their implementation has been optimized so as to minimize the impact that they make upon overall system behavior. Thus, a toolkit agent can be compared against a best available equivalent implementation.

7.1. The DFSTrace file reference tracing tools

This section presents a brief overview of the structure of the DFSTrace file reference tracing tools [Mummert & Satyanarayanan 92]. Figure 7-1 depicts the use of these tools.

The DFSTrace file reference tracing tools consist of several different components, which taken together provide the ability to produce traces of file usage on multiple hosts,

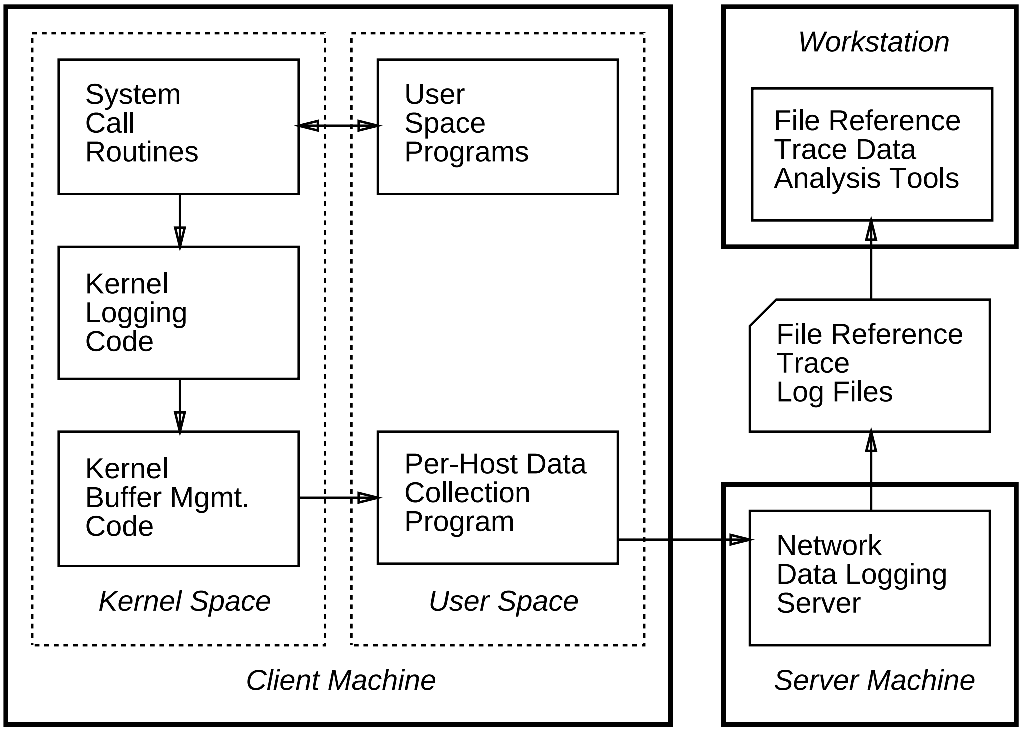

Figure 7-1: Structure of DFSTrace file reference tracing tools

and to later analyze these traces with a package of trace analysis tools. These components are described below.

7.1.1. Kernel Logging Code

The DFSTrace tools use a version of the Mach 2.5 kernel that has been modified to add logging code to each of the filesystem related system calls. Each such system call performed results in a variable-length structured log record being written to an internal kernel buffer. The arguments and results of each logged system call are included the log record.

Each log record is prefaced by a timestamp, process ID, and other identifying information. References to open files are logged as kernel open file table numbers, rather than as process specific file descriptor numbers. Also, system calls are counted on a kernel wide basis, rather than on a per-process basis.

7.1.2. Kernel Buffer Management Code

The DFSTrace tools utilize a kernel device driver that manages a buffer of log records. This driver buffers records accepted from the kernel logging code and allows blocks of these records to be read by user code via the `/dev/dfstrace` device. Operations for all processes in the system are recorded in the order performed.

7.1.3. Per-Host Data Collection Program

A data collection program is run on each host on which tracing is being performed to extract trace records from the kernel. This program reads blocks of trace records from the kernel trace record buffer via the `/dev/dfstrace` device using one LWP [Satyanarayanan 91] library lightweight process. It then transmits these records via the internet using an RPC2 [Satyanarayanan 91] interface to the network data logging server from a different lightweight process. Trace records are transmitted over the network instead of being written to local files so that traces collected on a potentially large set of machines can be recorded by a central data logging machine. This also minimizes the impact of tracing on filesystem usage.

7.1.4. Network Data Logging Server

The DFSTrace tools use a server to write file reference trace data to log files. The server can accept connections from multiple per-host data collection programs. Thus, a single logging server is typically used to log the trace data from a number of different machines.

7.1.5. Data Analysis Tools

A number of different programs exist for performing analysis of logged file reference trace data files. These tools perform such functions as extracting file usage information and access patterns, identifying periods of high activity, providing breakdowns of processes by user, and providing summaries of operations used.

7.2. The DFS_Trace Agent

This section presents an overview of the structure of the `dfs_trace` agent, which was built using the system interface interposition toolkit. The `dfs_trace` agent is intended to provide equivalent functionality to the DFSTrace file reference tracing tools so that a realistic comparison of an agent to a best available implementation can be performed. Figure 7-2 depicts the use of this agent.

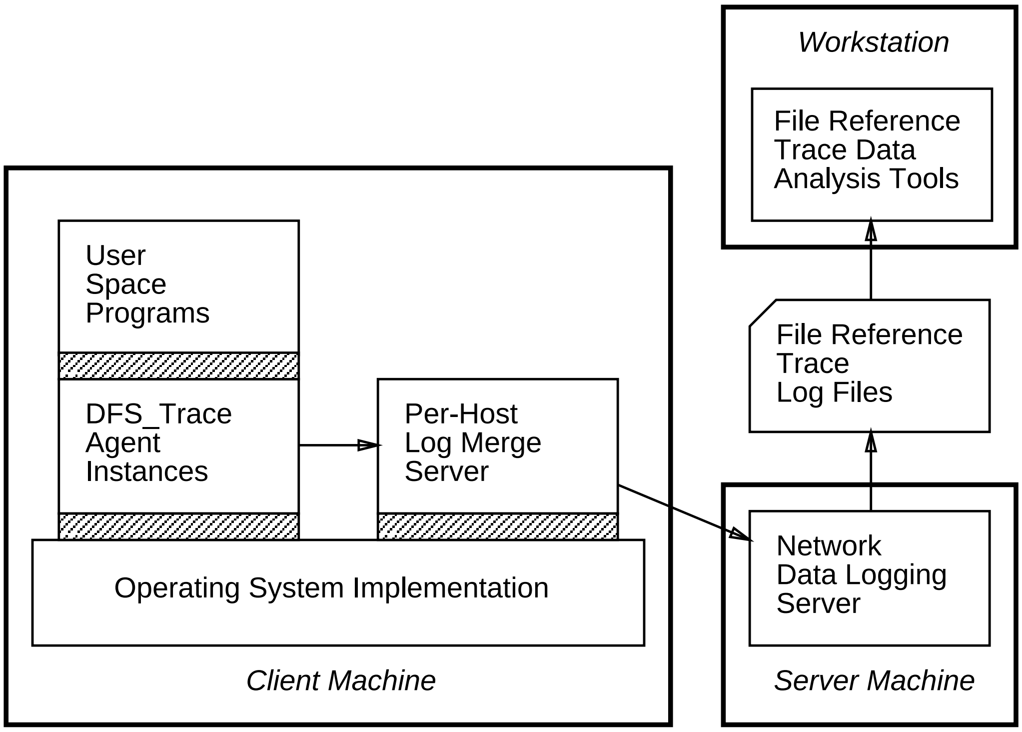

Figure 7-2: Structure of dfs_trace agent

The `dfs_trace` agent performs DFSTrace style file reference traces by replacing the kernel logging code, kernel buffer management code, and per-host data collection program of the DFSTrace file reference tracing tools with two components: an agent and a per-host log merge server. These components are described below. The DFSTrace network data logging server and data analysis tools continue to be used and require no modifications.

7.2.1. DFS_Trace Agent

The original DFSTrace implementation uses kernel code that logs the traced system calls made by all processes to an in-kernel buffer. The `dfs_trace` agent replaces this kernel code with agent code that logs the traced system calls made by processes run under the agent to a per-host log merge server. For each logged call, the agent constructs a log record with an equivalent structure to that which would have been constructed by the kernel logging code. It then sends the log record to the per-host log merge server using a MIG [Draves et al. 89] interface via Mach IPC [Young et al. 87, Draves 90].

7.2.2. Per-Host Log Merge Server

The log merge server collects log records from all `dfs_trace` agents monitoring processes on that host and merges them together into a single buffered stream of log records. It then transmits these records via an RPC2 interface to the DFSTrace network data logging server in the same way that the per-host data collection program did. Thus, the per-host log merge server replaces both the kernel buffer management code and the per-host data collection program.

As well as simply collecting log records, the log merge server synthesizes several values that the DFSTrace kernel logging code would have taken from kernel data structures. This is described in the following section.

7.3. Simulating Kernel Dependencies

There are number of ways in which the DFSTrace file reference tracing tools implicitly depend upon the tracing being performed inside the kernel. This section examines these dependencies, and describes how these situations are handled by the `dfs_trace` agent and log merge server.

7.3.1. Running Time Counter

The DFSTrace kernel logging code assumes that the time of day is represented as a free running counter that can be directly read in the manner described by Lamport [Lamport 88]. This can typically be done with two fetches, two stores, another two fetches, and a compare; infrequently an additional fetch and store are required. This method is used in a number of places by the kernel tracing code to obtain the time of day.

The `dfs_trace` agent uses the `gettimeofday()` call provided by the underlying system interface implementation in order to obtain the time of day. It must do this because no free running time counter is typically available in user space [Black 90], and in particular, none is available on the Intel 386 [Intel 86] or 486 [Intel 90], the machines on which the `dfs_trace` agent is run. Thus, the assumption of a running time counter can be removed, albeit at a much higher cost.

7.3.2. Virtual File System File IDs

The DFSTrace file reference tracing tools assume that a filesystem type and file system specific file identifier can be obtained for all referenced filesystem objects. This allows objects resident in the different types of filesystems supported by Mach 2.5 through the virtual filesystem interface (VFS) [Kleiman 86] to be differentiated by filesystem type. (Types supported are the Berkeley Fast Unix File System (UFS) [McKusick et al. 84], the Andrew File System (AFS) [Howard et al. 88], the Sun Network File System (NFS) [Sun 86], and the Coda File System [Satyanarayanan et al. 90].) Furthermore, for AFS and Coda files, the file identifier has a high probability of uniquely identifying files present in their respective distributed file systems across multiple machines. File identifiers are included in trace records both for objects explicitly referenced by traced system calls, and optionally for all components of all referenced pathnames.

The `dfs_trace` agent uses the `stat()` and `fstat()` calls provided by the underlying system[19] to obtain device and inode numbers for pathnames and open objects referenced by descriptors. These are used to construct trace file identifiers that describe all files as if they were UFS files. These identifiers will be the same as those built by the DFSTrace tools for UFS files, and will differ for all other filesystem types. Uniqueness of the file identifiers across machines is consequently lost, while uniqueness of file identifiers within a single machine is still preserved. While global uniqueness is potentially useful, in practice this property was never used [Mummert 92].

One property of file identifiers which was used by the original DFSTrace tools that the `dfs_trace` agent does not preserve is the ability to identify the filesystem type of each traced filesystem object. Some of the DFSTrace data analysis tools produced breakdowns of filesystem activity by filesystem type. All activity reported by the `dfs_trace` agent appears to have come from UFS filesystems. The reason for this is that the 4.3BSD interface hides filesystem specific characteristics beneath a generic filesystem interface. The generic filesystem interface makes it possible for programs to operate the same on many different types of filesystems; indeed, programs can not distinguish between filesystem types using normal 4.3BSD facilities. Thus, the `dfs_trace` agent attempts to make a reasonable approximation by assuming that all files are UFS files.

---

[19]Only calls made by applications are traced; calls made by the agent are not.

7.3.3. Virtual File System File Attributes

The DFSTrace kernel logging code assumes that file attributes such as type, size, owner, and reference count can be obtained directly from virtual filesystem [Kleiman 86] data structures for all referenced filesystem objects. File attribute information is included in trace records for most of the traced calls. For example, type, size and owner are logged for `open()` calls and size is logged for `close()` calls.

The `dfs_trace` agent uses the `stat()` and `fstat()` calls provided by the underlying system to obtain attribute information for pathnames and open objects. This is approach is necessary since file attribute information is not available from in-memory data structures in user space. Nonetheless, this approach allows the assumption that attributes can be obtained from in memory data structures to be removed, albeit at a much higher cost.

7.3.4. Quick Access to Miscellaneous Information

The DFSTrace kernel logging code assumes that other miscellaneous process information can be obtained directly from kernel data structures. For instance, it assumes that the process ID and current user ID can be obtained from memory. Such information is included in all trace records. The current process ID is one component of the record header included in all trace records; the current user ID is logged for the `fork()` call.

The `dfs_trace` agent uses calls provided by the underlying system interface implementation to access such miscellaneous information. For instance, it uses the `getpid()` and `getuid()` calls to obtain the current process and user IDs. The agent keeps a cached copy of the current process ID in memory which is initialized in each child at process creation time. Thus, while an initial performance penalty is paid to access the process ID, the cost is amortized over each access. While the agent does not currently do so, caching could also be used to improve access to other pieces of miscellaneous state in user space as well.

7.3.5. Per-Component Pathname Lookup Tracing

The DFSTrace kernel logging code instruments the internal kernel pathname operations in order to be able to provide per-component traces of all pathname lookup operations. This exposes such operations as descending into the directory hierarchy, following symbolic links, and crossing mount points, which are normally performed transparently.

The `dfs_trace` agent simulates pathname lookup operations on a per-component basis by explicitly traversing each pathname presented to the system interface. The `lstat()` call provided by the underlying system is used on each path component to retrieve file identifier information and symbolic link values, and to determine when mount points are being crossed. Thus, per-component pathname lookup traces can be simulated by the agent, although at a much higher cost than is possible in the kernel.

7.3.6. Interleaved Traces of Multiple Processes

The DFSTrace kernel logging code records traced calls in the precise order executed on a uniprocessor[20] since the logging is done as part of the actual execution. Subsequent components of the DFSTrace file reference tracing tools count on this ordering.

The `dfs_trace` agent and the per-host log merge server simulate this property by logging trace data recorded from each process by the `dfs_trace` agent in the single per-host log merge server. Each log record is immediately sent from the agent to the single log merge server, which buffers them in the order received. Since Mach IPC ports maintain a FIFO queueing discipline, this provides a reasonable approximation of the actual order of execution. Small ordering inconsistencies could still result from the agent being context switched between execution of a call and the logging of it. If it was deemed important to do so, the log merge server could sort all incoming log records by the enclosed timestamp.

7.3.7. Kernel Global File Table

The DFSTrace file reference tracing tools log all references to file descriptors as kernel global file table indices, rather than per-process descriptor numbers. This makes it easier to accumulate statistics on uses of actual objects, rather than on particular references to the objects, which may be aliased to different descriptor numbers within a single process, or may come from entirely different processes. Several operations such as `open()`, `read()`, `write()`, `seek()`, and `close()` update and/or log open object usage statistics, which are maintained on a kernel-wide basis.

The `dfs_trace` agent and the per-host log merge server simulate the kernel global file table and operations upon it so as to be able to provide equivalent traces to those produced by the DFSTrace tracing tools. They do so in the following manner.

---

[20]The kernel logging code does not contain sufficient locking code to execute correctly on a multiprocessor [Mummert 92].

The `dfs_trace` agent uses a specialization of the Open Object layer of the interposition toolkit to open object statistics on a per-process basis independent of particular process descriptor numbers. For instance, this means that toolkit code handles descriptor aliasing and renaming caused by such calls as `dup()`, etc. Specialized forms of such operations upon open objects such as `read()`, `write()`, `seek()`, and `close()` update per-process open object statistics.

Furthermore, when a new open object is created by such operations as `open()`, a per-host unique open object ID is created and associated with that open object. This unique ID consists of the creating process ID and a counter. This unique ID is inherited across `fork()` operations, enabling operations from different processes through inherited descriptors to be identified as operating on the same object. Each log record that would have contained a kernel global file table index is augmented with the corresponding unique open object ID when sent to the log merge server.

The per-host log merge server hashes each unique open object ID to a simulated kernel global file table index. It then deposits the resultant simulated file table index into the received log record constructed by the agent. Per-process statistics are merged to form per-host statistics. Thus, the `dfs_trace` agent and log merge server together are able to simulate the kernel global file table and the statistics that are kept in association with it by the DFSTrace file reference tracing tools.

In comparison to some of the other kernel features that are simulated by the `dfs_trace` agent and log merge server, the kernel global file table is actually relatively inexpensive to simulate. While a substantial amount of bookkeeping must be done, no extra system calls need to be performed.

7.3.8. Per-Machine System Call Counts

The DFSTrace file reference tracing tools maintain traced system call counts on a kernel wide basis. Running system call tallies are periodically logged, in addition to the usual log records associated with each individual logged system call.

The `dfs_trace` agent and log merge server simulate the kernel wide system call counts in a similar manner to the global file table simulation. Running counts are kept by the agent on a per-process basis. The log merge server merges the per-process counts to form per-host counts. As it is with the global file table, simulating the per-machine system call counts is relatively inexpensive since it only requires additional bookkeeping in memory and no additional system calls.

7.4. Software Engineering Comparisons

This section compares the two tracing implementations based on several software engineering criteria. Criteria used include code size, modularity, implementation time, and code difficulty.

7.4.1. Code Size

Table 7-1 presents several measures of the amount of code in those portions of the DFSTrace file reference tracing tools that were replaced by the `dfs_trace` agent and the companion log merge server. Kernel and user space code are presented on separate lines. As in Section 6.3, the actual metric used for statements was to count semicolons. A second set of counts is presented that also includes the number of C preprocessor directives (e.g., `#define`, `#include`, `#if`, `#endif`, etc.). Table 7-2 presents the same measures for the code that replaced it. Agent and server code are likewise presented on separate lines.

Original DFSTrace Tracing Implementation			
Component	*Statements*	*Statements + Preprocessor Directives*	*Files*
Kernel logging and buffer management	919	1321	30
Per-host data collection program	264	306	5
Totals	1183	1627	35

Table 7-1: Code size for original DFSTrace tracing implementation

Interposition Agent Tracing Implementation			
Component	*Statements*	*Statements + Preprocessor Directives*	*Files*
`dfs_trace` agent	1040	1209	18
Per-host log merge server	317	375	6
Totals	1357	1584	24

Table 7-2: Code size for interposition agent tracing implementation

Several observations can be made by comparing the two tables:

- **Code Size:** The two implementations contain roughly the same amount of code. This is unsurprising since they perform equivalent functions. The interposition agent implementation contains 15% more statements than the original. This is mostly due to the additional code needed to simulate kernel features used by the original implementation, as described in Section 7.3.

- **Conditionals:** When C preprocessor directives are included in the statement counts the two implementations still contain roughly the same amount of code, but in this case the interposition agent implementation actually contains 3% fewer statements than the original. This is mostly due to the large number of compile-time conditionals in the original implementation, which allow the kernel to be built with tracing disabled. No such conditionals are needed in the agent, since unlike the kernel tracing implementation, the agent implementation consumes no additional resources (other than disk space) and adds no overheads when not in use.

Table 7-3 presents several measures of the amount of code in those portions of the DFSTrace file reference tracing tools that are used with both the original and the interposition agent implementation of the tracing code. This is presented to provide a more complete view of the total size of the DFSTrace file reference tracing tools.

Code Used With Both Tracing Implementations			
Component	*Statements*	*Statements + Preprocessor Directives*	*Files*
Network data logging server	395	431	2
Miscellaneous logging related files	344	460	9
Data analysis support library	4531	4962	42
Data analysis programs	1610	1757	11
Totals	7144	7916	69

Table 7-3: Code size for tracing tools used with both implementations

Finally, Table 7-4 summarizes these results, separately presenting sizes for those portions of the original DFSTrace file reference tracing tools that are replaced by the `dfs_trace` interposition agent and log merge server. Grand totals for the entire original DFSTrace file reference tracing tools are then given.

Synopsis of Original DFSTrace Tracing Implementation			
Component	*Statements*	*Statements + Preprocessor Directives*	*Files*
Kernel logging and buffer management	919	1321	30
Per-host data collection program	264	306	5
Code used by both implementations	7144	7916	69
Grand Totals	8327	9543	104

Table 7-4: Code size synopsis of original tracing implementation

An important fact apparent from the summary is that a relatively small portion of the original DFSTrace file reference tracing implementation is replaced by the equivalent interposition agent. Depending upon which measure is used, only 14% (of statements) or 17% (of statements and preprocessor directives) of the total system is actually replaced. This fact is pertinent to the upcoming discussions of implementation time and code difficulty in Sections 7.4.3 and 7.4.4.

7.4.2. Modularity

Table 7-5 presents several properties of the sources for those portions of the DFSTrace file reference tracing tools that were replaced by the `dfs_trace` agent and the companion log merge server. Table 7-6 presents the same measures for the code that replaced it. Agent and server code are likewise presented on separate lines.

Original DFSTrace Tracing Implementation			
Component	*Machine Dependent Files*	*Files Modified*	*Total Files*
Kernel logging and buffer management	12	26	30
Per-host data collection program	0	0	5
Totals	12	26	35

Table 7-5: File properties for original DFSTrace tracing implementation

Interposition Agent Tracing Implementation			
Component	*Machine Dependent Files*	*Files Modified*	*Total Files*
`dfs_trace` agent	0	0	18
Per-host log merge server	0	0	6
Totals	0	0	24

Table 7-6: File properties for interposition agent tracing implementation

Several observations can be made by comparing the two tables:

- **Modularity:** The interposition-based file reference tracing implementation has substantially higher modularity than the original kernel-based implementation, as evidenced by using only 60% as many files. This is largely due to the hooks necessary to instrument the kernel for file reference tracing being scattered throughout 13 different files; in the `dfs_trace` agent logging calls are made in only 3 files.

- **Code Modification:** While the original implementation required changing 26 kernel files, the `dfs_trace` agent code consists only of new files. Rather than changing an existing implementation, it is implemented as derived versions of base objects supplied by the toolkit.
- **Machine Dependence:** The original implementation contains 12 machine dependent files. These actually break down as four files for each of three supported machine types. While the traces themselves are machine independent, the structure of the kernel requires machine dependent code to be implemented for each machine type. By contrast, the `dfs_trace` agent contains no machine dependent code. The underlying machine dependencies are hidden by the interposition toolkit.

7.4.3. Implementation Times

This section presents estimates of the times spent implementing the original DFSTrace file reference tracing tools and the `dfs_trace` interposition agent and accompanying log merge server. A discussion of the difficulty of implementing both sets of file reference tracing tools based on these times is presented in Section 7.4.4.

7.4.3.1. Implementation Times for Original DFSTrace Tools

Table 7-7 presents very rough estimates of the amount of time spent writing and debugging the original DFSTrace file reference tracing tools. These estimates were gathered during an interview [Mummert 92] with the author, Lily Mummert, and are based both on her working notes made during the implementation effort and her memory of it. The corresponding amounts of time spent on the `dfs_trace` interposition agent and accompanying log merge server are presented in Table 7-8 later in this section.

Very rough estimates of time to implement DFSTrace tools		
Task	*Estimated % Time Spent*	*Estimated Days Spent*
Kernel logging and buffer management	40%	34
Per-host data collection program	15%	13
Network data logging server	5%	4
Data analysis support library	20%	17
Data analysis programs	5%	4
Maintaining and using collected data	15%	13
Totals	100%	1 semester

Table 7-7: *Very* rough estimates of time spent implementing DFSTrace tools

While the table presents the time estimates as if the work were done continuously for a duration of a single semester on a full-time basis, of course in reality, it was performed over a longer period, interleaved with other activities. A summary of major events in the project follows.

The DFSTrace file reference tracing project began in the summer of 1989 with feasibility studies. A prototype of the kernel logging code was running by the end of that summer. Debugging and enhancements occurred during the summer of 1990. In early 1991, the kernel log data packing code was rewritten for increased portability and the per-host data collection program was changed to use the new network data logging server. In the spring of 1991, the data analysis support library and most of the data analysis tools were written; it was then that the system saw its first real use. Experience gained from the initial use led to several more enhancements: adding tracing for `read()` and `write()` operations and for pathname resolution operations on a per-element basis. During this time, bugs continued to manifest themselves and were only sometimes diagnosed; in particular, one bug that crashed the kernel every night was never identified and eventually just stopped manifesting itself. By the summer of 1991, the system was essentially complete. Throughout the rest of 1991, kernel work continued on an occasional basis, both to stay current with updated versions of the mainline kernel and to continue fixing bugs. The `execve()` tracing code was reorganized in late 1991. In early 1992, a support program was written to automatically dump trace data to tape. From early 1991 on, the tools have proven valuable both for providing traces to drive the DFSTrace file reference design and for other unanticipated uses, such as debugging problems with machine configurations.

Several other points from my interview with Mummert bear mentioning:

- **Steep Kernel Learning Curve:** She had to work through a steep learning curve before becoming productive in the kernel. While other aspects of the project progressed rapidly, nothing involving the kernel was as simple or progressed as quickly as she would have hoped.
- **Unproductive Debugging Environment:** No source level debugger was available for kernel code, making debugging the kernel code more tedious than code for the rest of the tracing tools, which were implemented in user space.
- **Unanticipated Software Maintenance:** An unexpectedly large amount of time was spent maintaining and updating kernel code after it was already largely working. Mummert estimates that even once the kernel code was ‘‘done’’, she still spent a half day every two weeks on kernel maintenance. This served largely to keep the kernel logging code up to date with new versions of the kernel and to fix bugs introduced through version skew. This caused considerable frustration as it became a time sink, with no effective return for the time spent.

- **Kernel Structure Unhelpful:** Certain portions of the kernel were not well structured for the task of implementing file reference traces. Instrumentation could not be done in a particularly structured manner; logging code was scattered throughout many different modules. Machine dependent changes were needed for logically machine independent functionality. Code paths through some operations were tortured and unobvious. As much was spent on `execve()` as all the other calls put together.

In summary, while the implementation of the majority of the DFSTrace file reference tracing components proceeded efficiently and soon yielded correctly working results, implementation of the kernel components consumed an disproportionate amount of time, with bugs continuing to manifest themselves long after the initial implementation was complete. Indeed, combining the results from Tables 7-4 and 7-7, we see that roughly 40% of the total effort was spent on only 11% of the code. The effective rate at which working kernel code was written was approximately 27 statements per day, as opposed to the rest of the code, where the effective rate was approximately 145 statements per day.

While these rates are based on very rough estimates of the implementation times for the DFSTrace tools, and are accurate to at most a single significant digit, one conclusion remains unmistakable: implementing kernel code consumed a disproportionate amount of time.

7.4.3.2. Implementation Times for Agent-Based Tracing Tools

Table 7-8 presents estimates of the amount of time spent writing and debugging the `dfs_trace` interposition agent and accompanying log merge server.

Combining results from Tables 7-2 and 7-8, this gives an effective rate at which working agent and server code was written of approximately 194 statements per day.

7.4.4. Code Difficulty

While it might be tempting to directly compare the amounts of time spent building the kernel-based and agent-based file reference tracing implementations and to draw sweeping conclusions about the relative productivity of the different implementation strategies from those comparisons, doing so would be grossly misleading (at best). Too many uncontrolled variables were present during both implementations for direct comparisons to be meaningful. Some of these were:

- **Different Programmers:** The kernel-based and agent-based tracing implementations were produced by different programmers. That large productivity differences exist between individual programmers is well established [Brooks 75].

Estimates of time to implement `dfs_trace` agent & server	
Task	*Days Spent*
Reworking kernel log data manipulation code to run in user space interposition agent	2
Implementing MIG interface between `dfs_trace` agent and log merge server and analyzing interactions between MIG code, RPC2 code, and the LWP lightweight process library used by the RPC2 code that sends data to the network data logging server	1
Reimplementing trace data buffer management code to operate in user space log merge server	1
Producing derived symbolic system call implementation that performs tracing of most system calls	1
Producing derived open object implementation and log merge server code that together simulate the kernel global file table numbers used in DFSTrace-style file reference trace records	1
Producing derived pathname implementation that performs per-element name resolution tracing	1
Totals	7

Table 7-8: Estimates of time spent building agent-based tracing implementation

- **Different Design Constraints and Specifications:** The original DFSTrace protocols and tools were being designed as they were being implemented. They were free to take advantage of useful facilities provided by their kernel-based implementation environment, while avoiding those that would be difficult or expensive [Mummert 92]. By contrast, the agent-based tracing tools were constrained to compatibly implement the existing DFSTrace log record protocols. While the agent lacked the freedom to tailor the protocol to the implementation environment, it had the advantage of implementing a precisely defined set of existing specifications.

- **Different Experiences and Expertise:** The programmers brought sets of skills and weaknesses to their respective tasks. For instance, Mummert was unfamiliar with Mach/BSD kernel programming, but was familiar with multi-threaded programming when she began the DFSTrace project [Mummert 92]; I was unfamiliar with the LWP [Satyanarayanan 91] lightweight process library, the RPC2 [Satyanarayanan 91] remote procedure call system, and the DFSTrace log record protocols, but was intimately familiar with writing interposition agent code.

- **Different Priorities:** Mummert built the DFSTrace tools on a part-time basis, while sometimes also working on other projects. I worked exclusively on the agent-based tracing tools while implementing them. While the implementation time was a primary interest to me, it was of secondary interest to her. Or, as it was elegantly put by Mary Thompson: ‘‘Lily didn’t know she was in a race!’’

Nonetheless, the data on building the kernel-based and agent-based file reference tracing implementations presented in Section 7.4.3, while essentially consisting of only two data points, currently provides the best available basis for evaluating the software engineering costs of constructing interposition agents using the interposition toolkit. Thus, at least a few preliminary observations seem to be in order.

The `dfs_trace` agent and accompanying log merge server took seven days to implement, as per Table 7-8. Together, these are functionally equivalent[21] to the DFSTrace kernel code and per-host data collection program, which took on the order of fifty days to implement, as per Table 7-7. If all other factors were equal (which they clearly are not), this would seem to indicate that interposition agents implementing some tasks can be more efficiently constructed than can kernel modifications that perform the same tasks.

While this result is not conclusively supported by the data available, two elements of the data seem to lend it some credence:

1. Similar code size. The amounts of code in the two implementations are quite similar. (The amounts were 1183 and 1357 statements for the kernel-based and agent-based implementations, respectively, as per Section 7.4.1.)
2. Similar non-kernel coding rate. The rates at which non-kernel code was produced in the two implementations are also similar. (The rates were 145 and 194 statements per day for the kernel-based and agent-based implementations, respectively, as per Section 7.4.3.)

These lend support to the possibility that the two results are at least somewhat comparable. Particularly, the second result seems to indicate that Mummert and I may be similarly productive when working outside the kernel.

Of course, I would like to be able to say that interposition agents can be constructed more efficiently than some equivalent implementations constructed by other means, but the scanty data currently existing neither proves nor refutes this claim. Substantially more study would be required before any general conclusions could be drawn. All that can currently be said with certainty is that the interposition agent-based implementation of the DFSTrace file reference tracing tools was produced far more quickly than was the original.

[21]with small exceptions, see Section 7.3.

7.5. Performance Comparisons

This section compares the performance of the two tracing implementations. Measurements taken using the Andrew filesystem benchmarks [Howard et al. 88] are used as a basis for comparison.

7.5.1. Overall Performance Comparisons

Table 7-9 shows the percentage slowdown in elapsed time for four different levels of tracing under both the original DFSTrace tracing implementation, and under the `dfs_trace` interposition agent implementation. The amount of trace data generated for each of these levels is also presented.

Tracing Overhead for AFS Benchmarks			
Tracing Level	*Original % slowdown*	*Agent % slowdown*	*Trace data (K-bytes)*
Default	3.0%	64%	525
Default, Read/Write	5.5%	69%	625
Default, Name Resolution	6.5%	133%	1480
All	7.0%	138%	1584

Table 7-9: Tracing overhead for the Andrew filesystem benchmarks

As Table 7-9 shows, while the original DFSTrace implementation causes applications to run only slightly slower, ranging from 3% to 7% for the AFS filesystem benchmarks, the equivalent agent-based implementation imposes a substantial performance penalty, ranging from 64% to 138%. The reasons for this performance loss are analyzed in the following sections.

7.5.2. Detailed Performance Analysis

This section analyzes some of the reasons that the agent-based file reference tracing implementation is significantly slower than the original kernel-based implementation. The analysis focuses on the performance of the default tracing level, which was chosen because it exhibits the proportionately worst slowdown.

Tables 7-10 presents some initial statistics about the application and agent measured. The AFS filesystem benchmarks perform 23093 system calls that are intercepted by the `dfs_trace` agent, 477 of which are successful `fork()`/`execve()` pairs. The application causes 12655 trace records to be sent, totalling 525 kilobytes of trace data.

AFS Benchmarks: System Calls and Resulting Log Records	
System calls intercepted	23093
`fork()`/`execve()` pairs	477
DFSTrace log records sent	12655
Total size of log records	525K

Table 7-10: System calls made by AFS benchmarks and resulting log records

Table 7-11 presents the elapsed time to run the AFS filesystem benchmarks with and without tracing. It also factors the system call and trace record counts from Table 7-10 into these times, yielding average slowdown figures of 2.90 milliseconds per intercepted system call and 5.29 milliseconds per log record. While they serve as an interesting starting point, these averages are not particularly informative. Thus, a number of more detailed measurements were made in an attempt to understand where the performance loss actually occurs.

`dfs_trace` Agent Overhead Summary for AFS Benchmarks				
Default Tracing Level	*Seconds elapsed*	*Agent % slowdown*	*µsec per syscall*	*µsec per log record*
Elapsed time with no tracing	105.0	—	4550	—
Elapsed time with agent tracing	172.0	—	7450	13590
Total slowdown	67.0	63.8%	2900	5290

Table 7-11: Agent tracing overhead summary, default tracing level

Table 7-12 presents a breakdown of where the additional time is being spent when the AFS filesystem benchmarks are run under the `dfs_trace` agent at the default tracing level. Both elapsed time and percent slowdown are presented for all costs. Where it makes sense, the average slowdown per intercepted system call and per log record are also presented. The remainder of this section discusses these overheads in greater detail.

When detailed measurements were taken, two pleasant surprises were the small amounts of time spent by the log merge server and in communicating with the log merge server using the MIG [Draves et al. 89] interface and Mach IPC [Young et al. 87, Draves 90]. The log merge server time, which actually includes the time spent by the network data logging server as well, accounted for only a 1.2% slowdown. Likewise, the entire Mach IPC and MIG costs accounted for only a 2.0% slowdown, despite the fact that every log record was sent from the agent to the log merge server in an individual IPC

`dfs_trace` Agent Overhead Analysis for AFS Benchmarks				
Default Tracing Level	*Seconds elapsed*	*Agent % slowdown*	*μsec per syscall*	*μsec per log record*
Log merge server	1.3	1.2%	—	102
RPC and Mach IPC costs	2.1	2.0%	—	165
Log message construction	28.3	27.0%	—	2240
`dfs_trace` object layer	3.0	2.9%	129	237
`open_object` & `pathname` toolkit layers	5.6	5.3%	242	—
`numeric` & `symbolic` toolkit layers	5.0	4.7%	216	—
Syscalls implemented by base toolkit layer	18.7	17.8%	809	—
Not accounted for	8.0	7.6%	346	632
Total slowdown	67.0	63.8%	2900	5290

Table 7-12: Agent tracing overhead analysis, default tracing level

message. While this path could clearly be optimized by batching log records, etc., the current 165μsec. overhead per log record is not a major contributor to the slowdown. Neither of these caused a large portion of the 63.8% total slowdown.

Almost half of the additional overhead can be attributed to a single `dfs_trace` agent routine, which constructs the tracing log records. It contributes a 27.0% slowdown to the 63.8% total. These costs are nearly all due to additional system calls that the agent must make in order to obtain information needed for the log records, which the original kernel-based implementation could obtain from in-memory data structures, as discussed in Section 7.3. For instance, each log record is prefaced by a timestamp, as per Section 7.1.1, which the agent must obtain via a `gettimeofday()` system call, as per Section 7.3.1. Likewise, each pathname referenced results in an additional `stat()` call, as per Section 7.3.2; descriptor references result in `fstat()` calls, as per Section 7.3.3. Other additional system calls are also required. While some optimizations of the agent-based implementation are possible, it can probably never match the very low costs of the kernel-based implementation for constructing DFSTrace log records, a core application requirement.

About a quarter of the additional overhead can be attributed to the low-level toolkit code which implements those system calls that must be implemented by each agent to provide a correct system interface implementation.[22] It contributes a 17.8% slowdown to the 63.8% total. Of the 18.7 seconds spent in this code, roughly 11 seconds, a 10%

---

[22]For a discussion of the system calls that must be intercepted by each agent, see Section 5.6.

slowdown, can be attributed to just `fork()`/`execve()` overheads, assuming that the actual `fork()`/`execve()` costs for this application and agent are similar to those presented in Table 6-5. While these implementations can still be optimized, once again the agent-based implementation can probably never completely match the performance of the kernel-based implementation for these system calls.

An additional 12.9% slowdown can be attributed to code implementing three layers of abstraction, two provided by the toolkit, and one specific to the `dfs_trace` agent. The primary costs within these layers are memory allocation and deallocation (which of course sometimes result in system calls) and C++ virtual procedure calls. Finally, the remaining 7.6% slowdown has not been accounted for.

7.5.3. Conclusions from Performance Comparisons

A large percentage of the performance loss of using the `dfs_trace` agent is a direct result of the agent having to make several additional system calls per trace log record. These system calls sometimes serve to gather information on objects maintained by the operating system that is needed for DFSTrace style log records. For instance, the agent must make system calls to construct file identifiers and retrieve file attributes for referenced files. The directly instrumented operating system implementation has a decided performance advantage for gathering such information, as the needed information is typically available via only a few memory references or procedure calls, instead of via a set of system calls. (Indeed, the easy availability of certain types of information within the kernel influenced the decision to include it the the DFSTrace log records in some cases [Mummert 92].) To the extent that such operating system maintained information is needed by interposition agents, as it is in this particular agent, clearly the faster the system call path, the better. Here, the system call overhead is a limiting factor, and so the best interposition-based implementation will always fare slightly worse than the best monolithic implementation in such cases.

That does not mean, however, that the gap need be as large as that between the original DFSTrace implementation and the current `dfs_trace` agent. Faster implementations certain key operations such as pathname lookup would go a long way towards reducing the gap.

One clear candidate for improvement made evident by the comparison is the interface for getting the time of day. While in the kernel, time can be obtained in a few instructions (see Section 7.3.1), outside the kernel the cost is at least an order of magnitude higher; the `gettimeofday()` call normally costs 25µsec. and an unoptimized agent manages to raise this to 170µsec., as per Table 6-5. As currently structured, the `dfs_trace` agent makes at least two such calls per log record. Having a free running time counter available in user space would eliminate most of this cost. Indeed, some have argued that free running time counters readable from user space should be ubiquitous [Black 90].

Another large percentage of the performance loss of using the `dfs_trace` agent comes from the current implementation of those system calls that must be implemented by each agent to provide a correct system interface implementation. As discussed in the previous section and Section 6.4.4, `execve()` is a prime culprit. In many ways, this type of loss is similar to the previous: the toolkit implementation of `execve()` requires several additional system calls to achieve the effect that the kernel accomplishes within one. While many optimizations to the current implementation of such calls are clearly possible, as in the previous case, the best monolithic implementation of these calls will always fare at least slightly better than the best interposition-based implementation.

More performance loss occurs due to the flow of control constantly passing up and down through several layers of abstraction, which are realized in the current implementation as separate procedures. While each additional layer adds only a slight overhead, in combination the overheads soon add up. However, unlike some of the other overheads, these are not intrinsic. They are amenable to reduction by several proven optimization techniques, from caching and inter-procedural analysis to dynamic code synthesis [Massalin 92].

The original DFSTrace tools and protocols were designed to provide portable distributed file reference traces with as little impact on overall system performance as possible. As Table 7-9 demonstrates, they clearly met this goal. Use of an instrumented operating system kernel to gather file references traces was a key to this success.

The `dfs_trace` agent attempted to duplicate the functionality and performance of the original DFSTrace tools using a user space interposition agent to gather file reference traces instead of a modified operating system implementation. While the functionality was mostly duplicated[23], the performance was in no way equivalent. Unlike the original kernel-based implementation, the `dfs_trace` agent incurred significant performance penalties gathering information critical to building DFSTrace log records due to having to repeatedly cross the system interface boundary. While use of the agent still left the resulting system quite usable, its use was not transparent with respect to perceived system performance.

---

[23]For the one major exception, see Section 7.3.2.

7.6. Conclusions from Comparisons

The exercise of attempting to build an interposition agent that was equivalent to the original DFSTrace file reference tracing tools both in function and performance and then comparing the result proved to be valuable in several respects. It provided a real basis of comparison, which clearly pointed out both strengths and weaknesses of interposition in general and of the specific interposition implementation used. It also served to highlight areas where both the system interface used and its implementation could be improved.

The two key points made evident by the comparison are:

- Agents can be easy to construct. It appears that constructing an interposition agent that provides an enhanced implementation of the system interface can be at least as easy and possibly easier than modifying an existing operating system implementation to perform the equivalent functions, as per Sections 7.4.3 and 7.4.4.
- Agents may not perform as well as monolithic implementations. Agents that need to access resources maintained by the underlying operating system implementation will be limited in their performance by the overhead involved in crossing the system interface boundary in order to access those resources. Hence, the best monolithic implementation of a given facility needing access to system resources will always perform better than the best interposition-based implementation of the same facility, as per Section 7.5.3.

Other points also made evident by the comparison are:

- Agents can be as small as the equivalent changes to a monolithic implementation. Interposition agents built using the interposition toolkit can contain no more new code than the amount of code changed or added to a monolithic system implementation to implement equivalent facilities, as per Section 7.4.1.
- Agents can be better structured than monolithic implementations. Interposition agents built using the interposition toolkit can be more logically structured and be more portable than a monolithic implementation of equivalent facilities, as per Section 7.4.2.
- Agents require no system modifications. Unlike monolithic implementations, where providing an enhanced implementation of a system often requires modifying the code implementing the system, interposition agents can provide enhanced implementations as an independent layer requiring no modifications to the underlying system, also as per Section 7.4.2.
- Agents can only use facilities exported across the system interface. Some information and facilities are unavailable to clients of the system interface that are available to system interface implementations. Thus, there are things that can be implemented within a given system interface implementation that can not be implemented outside of it. For example, some filesystem type information was unavailable to the `dfs_trace` agent that was available to the original kernel logging code, as per Section 7.3.2.

- Some facilities not currently exported across the 4.3BSD system interface probably should be. Doing so could provide both interposition agents and normal application programs useful facilities currently only available in the kernel. For instance, supplying a free running time counter in user space would significantly reduce the cost of obtaining the time of day, as per Section 7.3.1.

  Of course, the consequences of such additions should be carefully considered. For instance, interposing on a running time counter may be either impossible, or significantly more expensive than interposing on a system call. At best, agents that manipulate the apparent time of day, such as `timex`[24], would require additional mechanisms in order to do so. Caution is in order: features added for the convenience of one type of agent may actually make others more difficult or impossible to implement.

The comparison to a best available implementation presented in this chapter clearly points out that there are tradeoffs involved between using a monolithic implementation of a given facility versus an interposition-based implementation of the same facility and that neither approach clearly dominates the other in all respects. These tradeoffs are discussed further in Section 12.3.

---

[24]For a description of `timex`, see Section 4.2.2.

Chapter 8
Low Level Results

This chapter evaluates some of the boilerplate at the low levels of the interposition toolkit.

8.1. Mach Dependencies

The current interposition toolkit implementation supports building agents which interpose on the 4.3BSD [Leffler et al. 90] interface provided by Mach 2.5 [Accetta et al. 86, Baron et al. 90]. While most of the toolkit is built using 4.3BSD facilities, and while agents can be built using only 4.3BSD and toolkit facilities, a small portion of the toolkit depends upon Mach facilities not provided by 4.3BSD. This section describes the dependencies of the toolkit on Mach specific features.

- **Memory Management Facilities:** The loader used by the `run` program[25] and the toolkit `execve()` implementation[26] depends upon several Mach memory management facilities. Features used include the abilities to read, write, copy, map objects into, allocate, deallocate, and list the contents of the address spaces of other processes. Mach calls used are `vm_read()`, `vm_write()`, `vm_copy()`, `vm_map()`, `vm_allocate()`, `vm_deallocate()`, and `vm_region()`.

- **Process Management Facilities:** The loader also depends upon several Mach process management facilities. Features used include the abilities to suspend, resume, list the threads of, and abort any pending operations in, get the registers of, and set the registers of other processes and the threads within them. Mach calls used are `task_self()`, `task_by_unix_pid()`, `task_suspend()`, `task_resume()`, `thread_suspend()`, `thread_resume()`, `task_threads()`, `thread_abort()`, `thread_set_state()`, and `thread_get_state()`.

- **System Call Interception Facilities:** Interposition agents depend upon the Mach system call interception and invocation facilities described in Sections 5.2 and 5.3. Mach calls used are `task_set_emulation()` and `htg_unix_syscall()`.

---

[25]The `run` program is described in Section 5.1.

[26]The toolkit `execve()` implementation is discussed in Section 5.4.

While the `run` program currently loads agents into separate processes and so depends upon being able to manipulate the address spaces and process state of other processes, this implementation is largely a historical artifact, and could easily be changed to load agents into its own address space. While the loader is capable of loading arbitrary binaries into other address spaces, only the `run` program actually uses this capability. Thus, the dependence upon being able to manipulate the address spaces of other processes could actually be easily removed. At that point, the only features used beyond those provided by 4.3BSD would be the abilities to allocate and deallocate memory at arbitrary locations, list the allocated memory regions in the current process, and of course, intercepting system calls by agents and invoking intercepted system calls from agents.

8.2. Problems Encountered with the 4.3BSD and Mach Interfaces

This section presents several problems encountered with the 4.3BSD [Leffler et al. 90] and Mach [Accetta et al. 86, Baron et al. 90] interfaces when implementing the interposition toolkit and agents.

8.2.1. Set-User-ID Programs

4.3BSD allows programs to be marked set-user-ID [Ritchie 79, Grampp & Morris 84], causing future invocations of these programs to be run with the stored effective user IDs, instead of with the IDs of the processes invoking these programs. Programs can also be marked as set-group-ID, with analogous result upon the effective group ID. This can work because the kernel is trusted to correctly load the program and has sufficient access rights to change the effective IDs.

Agents, however, typically satisfy neither of these properties. They can not be trusted to load the image correctly and they may have insufficient access rights to change to the effective IDs. (They may also have insufficient access rights to even read the program binary, as per Section 8.2.2, since set-user-ID programs are often protected execute-only as well.) Consequently, set-user-id programs are typically only run by agents with the permissions of the processes invoking them, rather than the stored effective ID values.

8.2.2. Execute-Only Programs

4.3BSD allows programs to be protected execute-only. Even though a process has insufficient access rights to read such a program binary, the kernel will still load and run it for the process. Similarly to the set-user-ID case in Section 8.2.1, this can work because the kernel is trusted to correctly load the program and has sufficient access rights to read the program binary.

Likewise, agents satisfy neither of these properties. They can not be trusted to load the image correctly and they may have insufficient access rights to read the program binary. Consequently, execute-only programs often can not be run by agents.

Unfortunately, some commonly used programs are stored execute-only on some systems. Such programs include `ps`, `uptime`, and even `date`.

8.2.3. Asynchronous Signals

4.3BSD signals can arrive asynchronously with respect to the flow of control of the program. They can cause interruptions at any point unless disabled.

While the toolkit is prepared to handle signals during emulated system call execution and defer their delivery until the system call returns, certain critical sections must be protected against signal delivery. In particular, under some circumstances, signals need to be masked off during the sequence which returns from a system call to the application, but at the end of the return sequence the signals both need to be unmasked and control must be transferred back into application code as an atomic operation in order to both properly restore the application state and not lose signals. While this is straightforward to do using `sigreturn()`, having to do so adds the cost of an agent system call to the cost of every application system call handled by the agent in this manner.

This atomicity is necessary because the agent is running as part of the application process. If it were instead manipulating a separate suspended agent process, the compound jump-and-set-signal-mask operation would be unnecessary. Of course, the costs of such manipulation would likely be at least as high as those of the compound operation. For a further discussion on such tradeoffs, see Section 10.2.

8.2.4. Non-Uniform Parameter Passing

One annoyance of the 4.3BSD system interface is that system call parameters are not always passed in a uniform manner. While most calls accept a vector of arguments in a regular format, a few calls such as `wait()`/`wait3()`[27] and `sigcleanup()` use irregular parameter passing mechanisms. Such calls sometimes accept arguments in registers or processor status flag bits.

While the original Mach 2.5 `htg_unix_syscall()`[28] implementation supported making system calls that passed parameters in the regular fashion, it failed to handle the irregular calls. Early in this work, I had to reimplement `htg_unix_syscall()` to be able to pass parameters to the irregular system calls in order for agents to be able to use the complete 4.3BSD system interface.

8.2.5. Lack of Agent Stacking

The Mach 2.5 system call interception and invocation facilities `task_set_emulation()` and `htg_unix_syscall()`[29] do not support transparent agent stacking — running agents on top of other agents. Only one user space system call handler can be installed for each system call. Only the kernel system call implementation can be called down to with `htg_unix_syscall()`.

In order to implement agent stacking using the Mach 2.5 facilities, some form of central resource arbitration that is knowledgable of the stacked agents and the resources they require would be necessary. This arbitration code would perform such tasks as allocating portions of the address space to different agents, intercepting system calls occurring within the address space and dispatching them to the correct agent, establishing the bindings allowing one level of agent to call down to the system calls provided by the next, and allowing one level of agent to signal the next layer up. David Black has suggested the term ‘‘Traffic Cop’’ for such arbitration code.

---

[27] `wait()` and `wait3()` are actually the same system call.

[28] For a description of `htg_unix_syscall()`, see Section 5.3.

[29] For a description of `task_set_emulation()`, see Section 5.2.

Chapter 9
Security Implications

This chapter discusses the security implications of interposing user code at the system interface. The first issue discussed is the nature of the trust relationships that need to hold between applications, agents, and the underlying system implementation. Next, issues of whether agents are transparent or visible to applications running under them are discussed. Finally, paper designs for two agents providing enhanced security semantics are presented.

9.1. Trust Relationships

This section discusses the nature of the trust relationships that need to hold between applications, agents, and the underlying system implementation and the circumstances under which these relationships must hold.

9.1.1. Applications Trusting Agents

An interposition agent that provides the system interface to an application running under it has complete control over the application. It can read, write, allocate, and deallocate the application's address space; it can write the application's registers; it can suspend and resume the application; it potentially implements all of the system services used by the application. Agents can introduce covert channels [Lampson 73].

Thus, applications have no choice but to trust the instances of the system interface provided to them by interposition agents at the same level that they would trust instances of the system interface provided to them by the operating system kernel[30]. In this sense, any agents used must be a portion of Trusted Computing Base [DoD 85] if they are to be used as a component of a secure system.

---

[30]The term ''kernel'' here, as elsewhere in this dissertation, refers to the default or lowest-level operating system implementation; it is not meant to imply that this implementation necessarily runs in processor kernel space.

9.1.2. Kernel Trusting Agents

An interposition agent is fundamentally user application code which runs on an instance of the system interface just as other user code does. The kernel does not trust normal applications, since they may be unintentionally incorrect or intentionally malicious. For exactly the same reasons, it can not trust interposition agents.

9.1.3. Agents Trusting Applications

Agents built using the interposition toolkit reside in the same address spaces as the applications that use them. This is largely an artifact of the Mach system call interception mechanism used to support these agents, which redirects system calls to handlers in the same address space[31]. Agents written for MS-DOS and the Macintosh also share this property, as per Chapter 3. Since this gives applications the ability to modify the agents on which they are run, these agents have no choice but to trust applications.

Not all agents, however, must trust the applications that use them. Some system call interception mechanisms transfer control to agents running in separate protection domains, often by sending a message or software interrupt. Examples are the TENEX `tfork` JSYS, the System V `/proc` operations, and the SunOS `ptrace()` operations, as per Chapter 3. Since agents written using such mechanisms run in separate protection domains from the applications that use them, just as the kernel runs in a separate protection domain, these agents need not trust applications.

Different levels of trust are appropriate for different kinds of agents. Two distinct classes of agents have different security requirements in order to ensure correct operation.

One class of agents needs to restrict the possible behaviors of applications using them to a subset of those available to the agents. Such agents must be run in separate protection domains from the applications and must not trust their client applications to ensure correct operation. Otherwise the agents, and thus the restrictions, could be subverted by the applications.

For instance, Michael Fryd wrote an agent for Tops-20 [Digital 78] that implemented a restricted environment used during on-line student programming examinations at Carnegie-Mellon that prevented accesses to other student's files which could have

[31]For a discussion of the Mach system call interception mechanism, see Section 5.2.

potentially been used for cheating [Wohl 90]. This agent ran in a superior process that was protected from interference by client processes.

Even though such agents must be run in separate protection domains in order to ensure correctness, agents that attempt to restrict the possible behaviors of applications have been used even on systems where multiple protection domains are unavailable, such as MS-DOS [Microsoft 91] and the Macintosh [Apple 88]. For instance, the Norton AntiVirus [Symantec 91b] for DOS and SAM [Symantec 91a] for the Mac both attempt to intercept destructive operations made by potentially malicious applications. While such agents can not defend themselves from sufficiently clever applications, and so are not foolproof, in practice they do succeed in preventing undesirable behaviors of applications.

A distinct class of agents permit applications the full range of behaviors, while possibly providing additional features through the system interface. Since the full range of behaviors available to such agents are also made available to applications using them, the ability to access and possibly modify the agent under which a an application is run provides no new capabilities for the application. Thus, such agents may be run in the same protection domains as applications using them and may trust their client applications without weakening the correctness of the resulting system. Examples include Stacker [Stac 92] for MS-DOS and DiskDoubler [Salient 91] for the Mac.[32]

Of course, even for agents that can trust applications, arguments exist for running running agents in separate protection domains from their client applications. For instance, doing so protects agents from unintentional corruption by incorrect applications; program faults are better isolated when agents and client applications are run in separate protection domains. But counter-arguments exist as well. For instance, overall performance may be better if agents are co-resident with their client applications. For a further discussion on issues of co-resident versus protected agents, see Section 10.2.

---

[32] For a further discussion of these and other agents, see Chapter 3.

9.2. Agent Transparency

This section discusses whether agents are transparent or visible to applications running under them and some of the implications of these properties. Possible mechanisms by which applications might detect the presence of agents are presented. Possible policies concerning the desirability of agent transparency versus agent visibility are presented separately.

9.2.1. Mechanisms

There are a number of different mechanisms by which an application could attempt to determine if it was being run under an agent. Some of these are:

- Probing memory for agent code. Co-resident agents may be detected by applications running under them by probing memory for the presence of an agent, particularly if agents are loaded at well-known addresses. Even when agent addresses are not known *a priori*, this approach is clearly feasible on machines having only 32 bits of address space to search, requiring, for instance, only 2^{20} accesses to probe every page when a 4K page size is in use. Locating co-resident agents via exhaustive memory search becomes more difficult at current processor speeds with larger address spaces, such as the 64 bit address spaces provided by the MIPS R4000 [MIPS 91] and the DEC Alpha [Sites 92], where roughly 2^{50} probes would be required.

- Making precise performance measurements. Applications could attempt to determine if they are being run under an agent by precisely timing calls on the system interface, and comparing the results to known values for these calls with no agents present for specific processor models and operating system software versions. This approach counts on detecting the additional overheads caused by interposition agents. While this approach will likely lead to some false positives due to non-agent delays, such as paging and being context switched, a careful implementation with access to accurate timings will likely not produce many false negatives.

- Using an explicit call to determine if agents are present. Some system interfaces provide an explicit call for determining if agents are present. The TENEX interface, for example, provides such a call to applications [Thomas 75]. Others such as Mach 3.0 [Golub et al. 90], MS-DOS [Microsoft 91], and the Macintosh [Apple 88], which support co-resident interposition agents, permit applications to determine the addresses of routines that handle system calls. Of course, if such calls for querying an application's interposition state are able to be intercepted by interposition agents, the agents can always lie, and say that no agents are present. Nonetheless, such a call could be provided via a trusted communication path [DoD 85] to a trusted computing base (TCB), in a similar manner to the VMS [Digital 85] trusted input path [Blotcky et al. 86], which can be used to establish a secure communication path to the login program. Only in the presence of a trusted communication path to a TCB can a negative result from such a call be trusted.

- Using a call to query for extensions. Some interfaces define an extension mechanism, and provide a call that queries for the presence of specific extensions. For instance, the X Window System provides an extensions interface [Scheifler & Gettys 86] that allows X servers to portably implement extensions and X clients to portably query for their existence. While such extensions may be implemented as part of the base system implementation, they could also be implemented by agents layered on top of the base implementation. If an application somehow knows that the base system does not provide a given extension, then the presence of the extension can be used as an indication of the presence of an interposition agent.

There are a number of mechanisms by which applications can attempt to determine if they are being run under agents. Some utilize explicit interfaces to query for the presence of agents. With others, applications use implicit properties of their run-time environments to ascertain whether agents are present. In conclusion, no matter how closely an agent emulates the behavior of a native implementation of the system interface that it presents to applications, I believe that sufficiently clever applications will always be able to determine to within any chosen non-zero delta of accuracy that they are being run under interposition agents.

9.2.2. Policies

There are a number of different policy positions that could be taken on the issues of agent transparency versus agent visibility. Different policies are appropriate under different circumstances. Some issues governing these policy decisions are:

- ''Truth in execution environment''. One possible policy is that applications should be told or be able to determine if they are running under interposition agents. Just as people must be told in the United States if their telephone calls are being recorded, under some circumstances applications should probably be told that they are running under agents.

  Certainly in a secure system, applications can only be trusted to the extent that the facilities used by those applications are trusted. An application running on an agent can only be trusted if the agent is trusted. In a system supporting delegation [Gasser & McDermott 90, Lampson et al. 91], the services provided by a system implementation can be readily distinguished from those provided by that system implementation with an interposition agent. Trust can be evaluated based on the total set of participating components, since their authorizations are visible to the applications.

- Publicizing extensions. Agents will sometimes add extensions to the system interface. In order to be able to use these extensions, applications must be able to determine that they are present. In such cases, agents must make themselves visible to interested client applications.

- Functional transparency. Agents must correctly emulate the normal behavior of the system interface if applications run under them are to function correctly. More precisely, agents must maintain all the invariants depended upon by the applications run on them if these applications are to function correctly. This is the minimum sense in which agents must be transparent.

- Complete transparency. Agents which attempt to non-invasively monitor and possibly restrict the behavior of applications need to remain undetected by applications run under them; if applications detect the presence of such agents they could modify their behavior, invalidating the results of the monitoring. For instance, one such type of agent provides restricted environments for running untrusted binaries, a problem sometimes known as Secure Remote Execution [Tygar & Yee 91]. While complete transparency is clearly desirable for such agents, it is unlikely that agents can remain undetected by sufficiently resourceful applications, as discussed in the preceding discussion on mechanisms in Section 9.2.1.

9.3. Agents Providing Enhanced Security Semantics

This section presents a paper design for two agents that provide enhanced security semantics beyond those provided by the base system implementation.

Such agents should be run in separate protection domains from applications, just as the kernel is in a separate protection domain from applications, as per the trust discussion in Section 9.1.

9.3.1. Agent Providing a Restricted Execution Environment

This section presents a design for an agent that provides a restricted environment in which untrusted binaries can be safely executed. This problem is known in the security literature as Secure Remote Execution [Tygar & Yee 91].

An agent supporting secure execution of untrusted binaries may need to constrain the possible effects of these programs in the following areas:

- **Data Read:** The agent may need to constrain the data which untrusted programs can read. For instance, one possible policy is to limit untrusted programs to reading files which are readable by all users.
- **Data Written:** The agent probably needs to constrain the files which could be written. For instance, one possible policy is to limit untrusted programs to writing files which are writable by all users. A more sophisticated policy is to make it appear to untrusted programs as if they can write anything which a normal program would be permitted to, but to actually perform all writes to a hidden shadow area, while simulating having written them in place.
- **Resources Consumed:** The agent probably needs to to constrain the resources that can be consumed by untrusted programs. For instance, such resources as disk space, memory, and processes might need to be limited by the agent.

It is possible to construct an agent which guarantees that programs run under it can not corrupt data. This can be done by intercepting all operations which reference or can operate on filesystem objects and redirecting any which attempt to modify existing data to a hidden shadow area, which is written to instead. Data in the shadow area is always used in preference to data found in the normal file systems. The existence of the shadow area would be hidden from the programs. For instance, a unique shadow area could be created under the `/tmp` directory when the agent is invoked. Then any writes attempted to the file `/usr1/mbj/.cshrc` might actually be performed on a shadow copy such as `/tmp/untrusted_shadow.32260/usr1/mbj/.cshrc`. Similarly, any new files added to `/usr1/mbj/bin/` would actually be added to `/tmp/untrusted_shadow.32260/usr1/mbj/bin/`. Deleted or renamed filesystem objects would also be recorded, and the apparent pathname space presented to the programs would be updated accordingly.

This agent could be constructed using the pathname and descriptor layers provided by the interposition toolkit. Derived versions of the toolkit pathname objects would be used to map program pathnames into actual pathnames, with non-destructive accesses being passed through and destructive accesses being mapped to pathnames in the shadow area. A derived version of the toolkit directory object is needed to allow the apparent contents of directories to be listed when the result is actually a composite of any original directory, the contents of any shadow directory, and any recorded deletions or renames. Derived versions of the toolkit descriptor objects are needed so that descriptors can refer to agent objects such as simulated open directories.

If the agent needs to constrain the data which untrusted programs can read, an additional access control check can be performed in the agent. This check for prohibited read accesses would likely be in the same place as the check for attempted write accesses to existing data.

If the agent needs to constrain the resources used by untrusted programs, it can intercept all relevant resource allocation and deallocation calls and monitor the resources in use. The agent can then prevent programs from exceeding the desired resource constraints.

Finally, note that this agent would actually be very similar to the union directory agent described in Section 4.2.4. The `union` agent already presents the contents of a list of directories as if they were a single directory. The restricted environment agent would mostly add additional bookkeeping to what is already done by the `union` agent.

9.3.2. Agent Providing Extended Filesystem Protection Semantics

This section presents a design for an agent that provides enhanced filesystem protection semantics beyond those provided by the base system implementation. While most operating systems supply an existing mechanism for mediating access control that is adequate for the purposes intended, the particular mechanism chosen may not be appropriate in all circumstances or application domains. Indeed, the desirability of being able to evaluate potentially arbitrary access control predicates was established by ITOSS [Rabin & Tygar 87].

To accomplish this, the agent would intercept operations in which access control decisions are made and allow for the execution of arbitrary predicates at these points. These points are mostly those calls that use pathnames. As with the restricted environment agent described in Section 9.3.1, these predicates could be added in derived versions of the pathname objects provided by the interposition toolkit. The toolkit code would then cause them to be invoked for all pathname accesses.

If access revocation [Denning 82] is required, all operations upon objects for which access might be revoked must also be intercepted so as to be able to deny access under circumstances requiring revocation. This could be done by producing derived forms of the toolkit objects that represent reference counted open objects. References to open objects would be validated by this code, and would return errors if access had been revoked to the referenced open objects.

Under the UNIX security model [Grampp & Morris 84, Leffler et al. 90], the agent must be run with sufficient permissions to allow it whatever types of access that it might grant to clients. For instance, if general UNIX style set-user-ID [Ritchie 79, Grampp & Morris 84] and set-group-ID semantics must be emulated, this implies that the agent must be run as `root`. Of course, this means that the agent has effectively taken charge of all access control decisions for applications run under it, since `root` has full access to all system resources. Under a security architecture with delegation [Gasser & McDermott 90, Lampson et al. 91], the agent must be authenticated to clients, but need not be authorized to perform all operations permitted to clients.

9.4. Security Conclusions

Interposition agents change the security characteristics of computer systems. Agents can strengthen the security of the total system; they can also weaken it.

Programs run under interposition agents are no more secure than the agents themselves, just as they are no more secure than the kernel which provides the base operating system implementation. On the other hand, programs run under interposition agents can be constrained by the security models implemented by the agents; these models can be more secure than those provided by the base operating system implementation.

At least the same level of care should be exercised when deciding if an agent is trustworthy as is exercised when deciding if a normal application program is trustworthy. The security of its client applications depends on it.

Chapter 10
Lessons for Toolkit and Agent Writers

This chapter presents some of the lessons learned in building the interposition toolkit and interposition agents for the Mach/4.3BSD system interface that might be valuable to those writing future interposition toolkits or interposition agents.

10.1. Degree of Emulation

The system interface consists of both a set of interfaces and a set of invariants on the objects manipulated by these interfaces. Some of these invariants are part of the documented system interface specification; others are artifacts of the particular implementation. Applications rely on both.

In order to successfully run an application using an instance of the system interface provided by an interposition agent, the agent must preserve the subset of those invariants that are depended upon by the application. This leaves toolkit and agent writers with several possibilities:

- Preserve *all* invariants. If possible, this would allow all applications to be run under the agent. However, this is actually not possible, since it is impossible to determine all the invariants that applications might ever count on.
- Preserve documented invariants. This would allow all applications that obey the system interface specification to be run. Of course, most systems are inadequately specified for this approach to strictly practical. Many commonly used invariants are typically undocumented, often because they seemed ‘‘obvious’’.
- Preserve invariants needed by a set of target applications. Preserving only the subset of invariants needed by a chosen set of applications may be substantially easier than attempting to maintain the complete set of invariants. While, in a theoretical sense, it is still impossible to determine the set of invariants depended upon by an arbitrary finite set of applications, in practice, well-behaved applications tend to repeat their behavior on successive invocations, and so an experimental approach can be sufficient.

10.1.1. Examples of Invariants Preserved

This section presents examples of system interface invariants maintained by the Mach 2.5 implementation of the 4.3BSD system interface that are counted on by some applications and are preserved by all of the agents that I constructed. While there are certainly thousands of invariants maintained, the following two illustrate invariants that had to be explicitly preserved by toolkit code.

- **Least Free Descriptor Allocated:** When a new descriptor is allocated by a system call, 4.3BSD always allocates the lowest numbered free descriptor. While this property is not described in the `open()` manual page, it is nonetheless depended upon by some applications, such as Unix Emacs [Gosling 82][33]. The toolkit descriptor management layer preserves this invariant.[34]

- **Signals Interrupt Application Code:** When a signal is delivered under 4.3BSD, the program counter at the point of interruption, which is available to the signal handler, always points to application code that was interrupted. In early versions of the interposition toolkit, agent code could be interrupted by signals that were delivered to application signal handlers, with the interrupted program counter pointing to the interrupted agent code. This confused both some applications and the agents. Low-level toolkit code was subsequently added to make it appear to applications that signals only interrupt application code.

10.1.2. Examples of Invariants Not Preserved

This section presents examples of system interface invariants maintained by the Mach 2.5 implementation of the 4.3BSD system interface that are counted on by some applications and are not preserved by some of the agents that I constructed.

- **Zeroed Stack:** The contents of the stack are zeroed by the kernel-based `execve()` when a program is loaded. My agents leave some non-zero words on the stack beyond the stack pointer. Some programs that neglect to initialize stack variables will work on the kernel implementation, but fail under agents. This is a case of programs depending upon an invariant that is an artifact of a particular implementation which is not part of the system interface specification.

- **Sixty-four Descriptors:** The kernel is compiled to allow up to 64 open files per process. This limit is available both through the `getdtablesize()` system call and through the symbolic constant `NOFILE` in `<sys/param.h>`; the `getdtablesize()` call is meant to replace uses of

---

[33]Some applications, such as Unix Emacs, close descriptor zero, and then count on the the next `open()` call returning descriptor zero without checking the `open()` return value.

[34]The IEEE POSIX standards committee found that a sufficiently large body of existing practice depended upon this behavior to warrant mandating it in implementations claiming conformance to the base POSIX standard [IEEE 90].

the older `NOFILE` constant, allowing the maximum number of open files to be changed without requiring programs to be recompiled. Some agents, such as the `trace` agent, require descriptors for their operation and so reduce the number of descriptors available to applications. These agents return the reduced number from `getdtablesize()`; they return an invalid descriptor error for references to agent descriptors. Some programs, such as `csh`, use the value of `NOFILE`, rather than the number returned from `getdtablesize()`, and so may attempt to reference agent descriptors.[35] This is a case of programs depending upon an obsolete invariant.

- **Contents of Directory on Single Device:** The 4.3BSD directory format requires that all entries in a directory reside on a single device.[36] The union directory agent can violate this invariant by constructing a single logical directory from several actual directories which may reside on different devices. Many programs do not list the contents of directories, and of those that do, most never use this particular invariant; but not all programs will run successfully under such agents. This is a case of an invariant that is part of the documented system interface that is not depended on by a significant set of applications.

10.1.3. Conclusions on Degree of Emulation

For applications to run correctly on agents, the agents must preserve all the invariants depended upon by the applications. Minimally, this means that agent, and particularly toolkit writers, must be aware of these invariants and the tradeoffs involved in maintaining or not maintaining them. Maintaining only the invariants in the system interface specification is insufficient to ensure that applications continue working; maintaining some of the invariants that are in the specification is unnecessary for certain classes of applications. Just as those who port or reimplement operating systems must be aware of the invariants depended upon by applications, toolkit and agent writers must be aware of them as well.

---

[35]`csh` closes all unknown descriptors upon startup. It runs successfully under such agents because it ignores the error returned from `close()` on an invalid descriptor, and because it never subsequently attempts to use these descriptors.

[36]Directory entries contain an inode number but no device number.

10.2. The Choice of Co-Resident Agents

The agents that I built reside in the same address spaces as their client processes. This decision was made for several reasons:

- **Existing Mechanism:** The Mach system call interception mechanism redirects system calls to handler routines in the same address space. This made co-resident agents a natural choice for the Mach/4.3BSD environment.

- **Efficiency:** System calls can be redirected very quickly within the same address space, as no context switch needs to occur[37]. Likewise, client data can be accessed rapidly, without the use of any special cross-address space mechanisms.

This decision had several drawbacks. Some of these were:

- **Address Space Management:** When agents are co-resident with their clients, a portion of the address space must be dedicated to agent use, and so becomes unavailable to applications. This must be done in such a way that applications are unaffected. Also, if multiple agents are in use, care must be taken that the agents use mutually exclusive ranges of the address space, and do not interfere with one another's operation.

- **Management of Other Resources:** The address space is not the only resource shared when agents execute as part of the client process. Such resources as descriptors and the process signal state must be managed in such a way that the agent's use of them is transparent to applications.

- **Code to Hide Agent:** A substantial amount of code is necessary in order to hide the presence of agents from applications sharing the same address space and other process resources. The `execve()` call must be reimplemented by the agent to allow the application's portion of the address space to be reloaded, while maintaining the agent's portion. Much of the descriptor management interface must be reimplemented to allow `execve()`'s implicit descriptor management (close-on-exec) to function properly. The entire signal delivery mechanism must be reimplemented so as to make it appear that signal delivery only interrupts application code, hiding the fact that agent code could actually be interrupted as well. These issues are discussed further in Section 5.6.

- **Lack of Agent Stacking:** One of the largest drawbacks of co-resident agents is that agent stacking — running agents on top of other agents — can not be done transparently. Some form of central resource arbitration that is knowledgable of the stacked agents and the resources they require is necessary in order to perform such tasks as allocating portions of the address space to different agents, intercepting system calls occurring within the address space and dispatching them to the correct agent, establishing the bindings allowing one level of agent to call down to the system calls provided by the next, and allowing one level of agent to signal the next layer up. These issues are discussed further in Section 8.2.5.

---

[37]For the time to intercept system calls under Mach 2.5, see Section 6.4.3.

- **Agents Trusting Applications:** My agents have no choice but to trust applications, since they run in the same protection domain. This makes it impossible to write an agent that restricts the possible behaviors of applications without the same restrictions being placed on the agent itself. These issues are discussed further in Section 9.1.3.

- **Efficiency:** Despite the efficiency advantage of being able to intercept system calls and access application resources without context switching, there are some performance drawbacks to co-resident agents. These come mostly from the additional layers of code that are executed in order to hide the agent and the resources it uses from client applications. The clearest such example is that the cost of `execve()` is currently doubled by being reimplemented in user space, as per Section 6.4.3. Some of these costs could be avoided by running agents in separate address spaces, albeit at the cost of additional context switches. Having built co-resident agents but not separate agents, it would be premature to conclude that one type of agent is necessarily more efficient than the other for all applications.

Having built co-resident agents for the 4.3BSD interface using the Mach system call interception facilities, the shortcomings of this approach are now apparent. Not having built agents residing in separate address spaces, the drawbacks of this approach are less clear. Future toolkit and agent writers should be aware of these tradeoffs when deciding between running future agents in the same or separate address spaces than their client applications.

10.3. Toolkit Interface Design Criteria

This section presents some of the criteria and guidelines used when designing the interposition toolkit interfaces. The main criteria were:

- **Completeness:** Interfaces should provide a complete encapsulation of the abstractions manipulated by them. All operations that can be performed on the objects manipulated by a given interface should be able to be performed using that interface; clients of an interface should need no knowledge of its implementation; side interfaces or special ‘‘trap doors’’ should be avoided if at all possible.

- **Independence:** Interfaces should manipulate a single abstraction or closely related set of abstractions whenever possible. Furthermore, side effects of operations from one interface upon objects manipulated by other interfaces should be avoided whenever possible.

- **Increasing Levels of Useful Abstractions:** New interfaces should be introduced for each new level of abstraction at which agents might need to produce a derived implementation.

Of course, these criteria are often in conflict with one other and real systems need to strike a balance among them. For instance, in the 4.3BSD system interface, descriptors and pathnames are nearly, but not quite, independent: the `open()` and `creat()` calls produce descriptors from pathnames; the `getdirentries()` and `read()` calls produce sets of pathname elements from descriptors; the `bind()` and `connect()` calls associate pathnames with descriptors. Thus, complete pathname and descriptor interfaces are mutually dependent. Multiple inheritance is one technique that can help with such situations. Parameterized or generic interfaces is another.

If anything, I found that adherence to good design principles was even more important in building the interposition toolkit than it usually is. If the goals of flexibility and extensibility were to be realized, the interfaces needed to be as clean and well-defined as possible. Any shortcuts that I took in the design, or assumptions that I made about how an interface would be used, tended to come back to haunt me: the shortcuts reduced the flexibility of the toolkit objects; the assumptions often ended up not being valid. In my experience building the interposition toolkit and agents using it, I found that adherence to these good design principles helped produce clean interfaces that are both flexible and extensible.

10.4. Applicability to Other Interfaces

This dissertation presents a toolkit that attempts to simplify the construction of interposition agents. It does this by providing multiple structured views of the objects present in the system interface, allowing agents to be written at whatever levels of abstraction are appropriate to their particular functions, leveraging off of existing toolkit code whenever possible. While the toolkit itself is specific to a particular system interface, many of the techniques used should be applicable to other interfaces as well.

The success of this approach hinges on the ability to cast an existing interface into a set of well-structured object-specific interfaces, each of which is sufficient to fully encapsulate the behavior of the particular object. I believe that this should be possible for existing interfaces that satisfy two key requirements:

- the interface contains a reasonably small number of abstractions;
- the behavior of these abstractions is largely independent.

Interfaces satisfying these properties should be amenable to this approach.

The 4.3BSD interface [Leffler et al. 90] satisfies these properties. Other interfaces, such as the X Window System interface [Scheifler & Gettys 86], also do [Jones 92]. Not every interface does. As a glaring counterexample, consider the C library (libc) taken as a whole, which satisfies neither. The C library contains a huge number of abstractions that are often implemented in terms of one another in interdependent and non-obvious ways.

Chapter 11
Future Work

This chapter presents a brief look at some future research directions which could follow from this work.

11.1. Construct New Agents

One obvious possibility that this research opens up is the ability to construct new interposition agents. Such agents could include the restricted environment agent described in Section 9.3.1 and emulators for variant operating systems, such as HP-UX [Clegg et al. 86], on top of 4.3BSD [Leffler et al. 90].

11.2. Ports to Other Machines

The toolkit currently includes machine-dependent support for only the Intel 386/486 [Intel 86, Intel 90] and the VAX [Digital 81b, Digital 81a]. Machine dependent support for newer processor families such as the SPARC [Garner et al. 88], the MIPS [Kane 87], and the HP Precision [Lee 89] can easily be provided.

11.3. Support for Multiple Threads

Mach 2.5 makes it possible to write multi-threaded programs which use both 4.3BSD Facilities and Mach facilities. The interposition toolkit does not currently support such multi-threaded programs.

11.4. Support for Additional Abstractions

The abstractions currently fully supported by the toolkit are system calls, descriptors, pathnames, and directories. Classes for other 4.3BSD abstractions such as time, processes, protection, etc. could also be added. These classes would encapsulate all the uses of their corresponding abstractions within the system interface, allowing the behavior of these abstractions to be modified via inheritance.

Of course, not all system interface abstractions are as well-behaved as those currently supported. For instance, consider the time abstraction: while the `gettimeofday()` call presents time in a straightforward manner, many other calls use time implicitly. Such facilities as the modification and access timestamps on filesystem objects and interval timers are closely related to, but not identical to the time of day. While support for such abstractions may require more mechanism than is immediately apparent, it should nonetheless be possible to provide it in such a manner that agents can implement any desired policies.

As a side note, I believe that the use of multiple inheritance might help make it possible to flexibly combine the objects representing such largely independent abstractions. For instance, one independent class could have been written to encapsulate the uses of descriptors and another could have been written to encapsulate the uses of pathnames. A class that inherits from both of them could then have been used to capture the few places such as `open()` and `getdirentries()` where there are interactions between them. I expect that this strategy of combining largely independent classes, handling the areas of interdependence with multiple inheritance, would become more important as the number of not quite independent abstractions used within an agent grows. The toolkit currently does not currently use multiple inheritance.

11.5. Building Stacked Agents

Another possible future direction is to support stacked interposition agents. Several approaches are possible:

- The ‘‘traffic cop’’ approach, as described in Section 8.2.5.
- Providing Mach kernel support to treat application system calls as per-thread Mach exceptions [Black et al. 88].
- Using a mechanism in which one process can intercept another’s system calls, such as the SunOS version 4 [Sun 88a] `ptrace()` call.

11.6. Support for Agents in Separate Address Spaces

The current toolkit only supports agents which are co-resident with their clients, as discussed in Section 10.2. One avenue of future work is to construct a new set of toolkit boilerplate which supports agents which are in separate address spaces from their clients. Among other advantages, this would allow agents to be written which would be protected from their client applications, as per Sections 9.1.3 and 10.2. One obvious choice of platforms would be to produce toolkit boilerplate to use the SunOS version 4 [Sun 88a] `ptrace()` function for system call interception, as per Section 11.7.

11.7. Ports to Other UNIX-Like Operating Systems

Given that a number of contemporary operating systems present substantially similar interfaces to that of the Mach 2.5/BSD system on which the toolkit has been developed, a large portion of the toolkit should be portable and directly applicable to these systems as well. Some of these systems are described in Chapter 3, Related Work. For instance, the toolkit should easily port to SunOS, System V.4, and the Mach 3.0 Unix Server/Emulator [Golub et al. 90]. These ports would allow useful interposition agents to be easily portable among a range of contemporary operating systems.

A port to SunOS [Sun 88a] would seem to be an obvious choice for a number of reasons. A separate address space version could be built using `ptrace()` to intercept system calls. A co-resident version could be built using dynamic linking to perform interposition for applications which follow the convention that system call traps are obtained only from dynamically linked libraries, and are not embedded within the application itself. Also, such a toolkit port might prove useful to members of the large Sun user base.

11.8. Application of Ideas to Other Interfaces

As discussed in Section 10.4, the techniques used to construct the interposition toolkit for the 4.3BSD [Leffler et al. 90] interface should be applicable to other well-behaved interfaces as well. For instance, this approach should be applicable to the X Window System [Scheifler & Gettys 86] interface, as explored in [Jones 92].

Chapter 12
Conclusions

12.1. Summary of Results

This research has demonstrated that the system interface can be added to the set of extensible operating system interfaces that can be extended through interposition. Just as interposition is successfully used today with such communication-based facilities as pipes, sockets, and inter-process communication channels, this work has demonstrated that interposition can be successfully applied to the system interface. This work extends the known benefits of interposition to a new domain.

It achieves this result through the use of an interposition toolkit that substantially increases the ease of interposing user code between clients and instances of the system interface. It does so by allowing such code to be written in terms of the high-level objects provided by this interface, rather than in terms of the intercepted system calls themselves.

The following achievements demonstrate this result:

- an implementation of a system call interposition toolkit for the 4.3BSD interface has been built under Mach,
- the toolkit has been used to construct the agents previously described,
- major portions of the toolkit have been reused in multiple agents,
- the agents have gained leverage by utilizing additional functionality provided by the toolkit, substantially simplifying their construction, and
- the performance cost of using the toolkit can be small relative to the cost of the system call interception mechanism and the operations being emulated.

12.2. Thesis Contribution

This research has demonstrated both the feasibility and the appropriateness of extending the system interface via interposition. It has shown that while the 4.3BSD system interface is large, it actually contains a small number of abstractions whose behavior is largely independent. Furthermore, it has demonstrated that an interposition toolkit can exploit this property of the system interface. Interposition agents can both achieve acceptable performance and gain substantial implementation leverage through use of an interposition toolkit.

These results should be applicable beyond the initial scope of this research. The interposition toolkit should port to similar systems such as SunOS and UNIX System V. Agents written for the toolkit should also port. The lessons learned in building this interposition toolkit should be applicable to building similar toolkits for dissimilar systems. For instance, interposition toolkits could be constructed for such interfaces as the MS-DOS system interface, the Macintosh system interface, and the X Window System interface.

Today, agents are regularly written to be interposed on simple communication-based interfaces such as pipes and sockets. Similarly, the toolkit makes it possible to easily write agents to be interposed on the system interface. Indeed, it is anticipated that the existence of this toolkit will encourage the writing of such agents, many of which would not otherwise have been attempted.

12.3. Applicability and Tradeoffs

Interposition is one of many techniques available. As in other domains such as pipes, filters, IPC intermediaries, and network interposition agents, sometimes its use will yield a substantial benefit, while sometimes its use would be inappropriate. As with other layered techniques, peak achievable performance will usually not be achieved. Nonetheless, interposition provides a flexibility and ease of implementation that would not otherwise be available.

12.4. Vision and Potential

The potential opened up by interposition is enormous. Agents can be as easy to use as filters. They can be as easy to construct as normal application programs. The system interface can be dynamically customized. Interface changes can be selectively applied. Indeed, interposition provides a powerful addition to the suite of application and system building techniques.

Part IV

Appendices and References

Appendix A
The 4.3BSD System Interface

As part of the preliminary investigation into this topic, I performed a detailed study of the 4.3BSD [Leffler et al. 90] system interface, producing a taxonomy of the objects used by each of the system calls. This taxonomy is presented in Section A.1. Some statistics gathered from the taxonomy are presented in Table A-1.

4.3BSD System Interface Object Usage Statistics	
Categorization of System Calls	*# of Calls*
All system calls implemented	114
Calls using pathnames	30
General purpose	7
Directory specific	6
File specific	6
Symbolic link specific	2
Socket specific	2
Device specific	5
Calls using descriptors	48
General purpose	18
Directory specific	1
File specific	1
Socket specific	17
Calls implicitly using descriptors	6
Calls not using pathnames or descriptors	44
Calls using PIDs	16
Calls implicitly using PIDs	3
Calls using process groups	7
Calls using UIDs or GIDs	20
Calls implicitly using UIDs or GIDs	4
Calls using time	12
Calls using signals	13

Table A-1: 4.3BSD system interface object usage statistics

This taxonomy shows that while there are a moderately large number of calls, most of them only manipulate a few different kinds of objects. The toolkit capitalizes on this specificity, taking advantage of the fact that the behaviors of the abstractions present in the system interface, and thus the system calls using them, are largely independent. See Section 1.6 for a discussion the use on this property of the 4.3BSD system interface by the interposition toolkit.

A.1. 4.3BSD System Interface Taxonomy

The remainder of this appendix presents a taxonomy of the different objects used in each of the 4.3BSD system call interfaces. Those calls that are obsolete, unimplemented, or undefined are not presented. A key to abbreviations and terms immediately follows the list of system calls and abstractions used.

4.3BSD System Calls and Abstractions Used

System Call Name	*System Call #*	*# of Args*	*Status*	*Uses Path-names*	*Uses Descrip-tors*	*Re-stric-tions*	*Uses PIDs*	*Uses UIDs/ GIDs*	*Other Abstractions Used*
syscall	0	7	.	y	y	.	y	y	system calls
_exit	1	1	.	.	i	.	i	.	status
fork	2	0	.	.	i	.	y	.	
read	3	3	.	.	y	.	.	.	mem
write	4	3	.	.	y	.	.	.	mem
open	5	3	.	y	y	!s	.	.	mode
close	6	1	.	.	y	.	.	.	
creat	8	2	.	y	y	!s!d	.	.	mode
link	9	2	.	y	.	f	.	.	
unlink	10	1	.	y	.	!d	.	.	
execv	11	2	.	y	i	f	.	i	mem
chdir	12	1	.	y	.	d	.	.	
mknod	14	3	r	y	.	b|c	.	.	mode, dev
chmod	15	2	.	y	.	.	.	.	mode
chown	16	3	.	y	.	.	.	y	
brk	17	1	o	.	.	.	.	.	mem
lseek	19	3	.	.	y	!s	.	.	
getpid	20	0	.	.	.	.	y	.	
mount	21	3	r	y	.	b,d	.	.	blk
umount	22	1	r	y	.	b	.	.	blk

4.3BSD System Calls and Abstractions Used (continued)									
System Call Name	*System Call #*	*# of Args*	*Status*	*Uses Path-names*	*Uses Descrip-tors*	*Re-stric-tions*	*Uses PIDs*	*Uses UIDs/ GIDs*	*Other Abstractions Used*
getuid	24	0	.	.	.	.	.	y	
ptrace	26	4*	.	.	.	.	y	.	mem, status
access	33	2	.	y	.	.	.	.	mode
sync	36	0	.	.	.	.	.	.	filesystems
kill	37	2	.	.	.	.	y	.	pgrp, sig
stat	38	2	.	y	.	.	.	y	dev, fsys, time
lstat	40	2	.	y	.	.	.	y	dev, fsys, time
dup	41	2	.	.	y	.	.	.	
pipe	42	0	.	.	y	p	.	.	
profil	44	4	.	.	.	.	.	.	mem, time
getgid	47	0	.	.	.	.	.	y	
acct	51	1	r	y	.	f	.	i	time, tty, fsys
ioctl	54	3	.	.	y	many	y	y	dev, pgrp, etc
reboot	55	1	r	.	.	.	.	.	boot flags
symlink	57	2	.	y	.	l	.	.	
readlink	58	3	.	y	.	l	.	.	mem
execve	59	3	.	y	i	f	.	i	mem
umask	60	1	.	.	.	.	.	.	mode mask
chroot	61	1	r	y	.	d	.	.	
fstat	62	2	.	.	y	.	.	y	dev, fsys, time
getpagesize	64	1	.	.	.	.	.	.	page size
vfork	66	0	.	.	i	.	y	.	
vread	67	3	o	.	y	.	.	.	mem
vwrite	68	3	o	.	y	.	.	.	mem
sbrk	69	1	.	.	.	.	.	.	mem
mmap	71	6	.	.	y	.	.	.	mem
munmap	73	2	.	.	.	.	.	.	mem
vhangup	76	0	r	.	.	t	.	.	tty
getgroups	79	2	.	.	.	.	.	y	
setgroups	80	2	r	.	.	.	.	y	
getpgrp	81	1	.	.	.	.	y	.	pgrp
setpgrp	82	2	.	.	.	.	y	.	pgrp
setitimer	83	3	.	.	.	.	.	.	time, sig

4.3BSD System Calls and Abstractions Used (continued)									
System Call Name	*System Call #*	*# of Args*	*Status*	*Uses Path-names*	*Uses Descrip-tors*	*Re-stric-tions*	*Uses PIDs*	*Uses UIDs/ GIDs*	*Other Abstractions Used*
wait	84	3*	.	.	.	.	y	.	time, sig
swapon	85	1	r	y	.	b	.	.	blk
getitimer	86	2	.	.	.	.	.	.	time, sig
gethostname	87	2	.	.	.	.	.	.	host name
sethostname	88	2	r	.	.	.	.	.	host name
getdtablesize	89	0	.	.	i	.	.	.	
dup2	90	2	.	.	y	.	.	.	
fcntl	92	3	.	.	y	.	y	.	sig
select	93	5	.	.	y	.	.	.	mem
fsync	95	1	.	.	y	!s	.	.	filesystem
setpriority	96	3	.	.	.	.	y	y	pgrp, prio
socket	97	3	.	.	y	s	.	.	sockopt
connect	98	3	.	y	y	s	.	.	sockaddr
accept	99	3	.	.	y	s	.	.	sockaddr
getpriority	100	2	.	.	.	.	y	y	pgrp, prio
send	101	4	.	.	y	s	.	.	mem
recv	102	4	.	.	y	s	.	.	mem
sigreturn	103	1*	.	.	.	.	.	.	sigmask, regs
bind	104	3	.	y	y	s	.	.	sockaddr
setsockopt	105	5	.	.	y	s	.	.	sockopt
listen	106	2	.	.	y	s	.	.	socket backlog
sigvec	108	3*	.	.	.	.	.	.	sig, sigmask
sigblock	109	1	.	.	.	.	.	.	sigmask
sigsetmask	110	1	.	.	.	.	.	.	sigmask
sigpause	111	1	.	.	.	.	.	.	sigmask
sigstack	112	2	.	.	.	.	.	.	sigstack
recvmsg	113	3	.	.	y	s	.	.	mem
sendmsg	114	3	.	.	y	s	.	.	mem
gettimeofday	116	2	.	.	.	.	.	.	time
getrusage	117	2	.	.	.	.	.	.	rusage
getsockopt	118	5	.	.	y	s	.	.	sockopt
readv	120	3	.	.	y	.	.	.	mem
writev	121	3	.	.	y	.	.	.	mem

4.3BSD System Calls and Abstractions Used (continued)									
System Call Name	*System Call #*	*# of Args*	*Status*	*Uses Path-names*	*Uses Descrip-tors*	*Re-stric-tions*	*Uses PIDs*	*Uses UIDs/ GIDs*	*Other Abstractions Used*
settimeofday	122	2	r	.	.	.	.	.	time
fchown	123	3	.	.	y	!s	.	y	
fchmod	124	2	.	.	y	!s	.	.	mode
recvfrom	125	6	.	.	y	s	.	.	mem, sockaddr
setreuid	126	2	.	.	.	.	.	y	
setregid	127	2	.	.	.	.	.	y	
rename	128	2	.	y	.	.	.	.	
truncate	129	2	.	y	.	f	.	.	file length
ftruncate	130	2	.	.	y	f	.	.	file length
flock	131	2	.	.	y	!s	.	.	file lock
sendto	133	6	.	.	y	s	.	.	mem, sockaddr
shutdown	134	2	.	.	y	s	.	.	
socketpair	135	4	.	.	y	s	.	.	sockopts
mkdir	136	2	.	y	.	d	.	.	mode
rmdir	137	1	.	y	.	d	.	.	
utimes	138	2	.	y	.	!s	.	.	time
sigcleanup	139	0*	o	.	.	.	.	.	sigmask, regs
adjtime	140	2	r	.	.	.	.	.	time
getpeername	141	3	.	.	y	s	.	.	sockaddr
gethostid	142	0	.	.	.	.	.	.	hostid
sethostid	143	1	r	.	.	.	.	.	hostid
getrlimit	144	2	.	.	.	.	i	.	rlimits
setrlimit	145	2	.	.	.	.	i	.	rlimits
killpg	146	2	.	.	.	.	.	.	pgrp, sig
setquota	148	2	r	y	.	b,f	.	i	fsys, quota
quota	149	4	.	.	.	.	.	y	blk, quota, tty
getsockname	150	3	.	.	y	s	.	.	sockaddr
getdirentries	varies	4	.	y	y	d	.	.	mem

Key to Abbreviations and Terms

Abbreviations and terms used in the 4.3BSD system call taxonomy are:

Number of arguments:

*	Indicates arguments passed in irregular fashion, or argument count varies between machines

System call status values:

o	System call obsolete but implemented
R	Root privileges required to use system call

Uses pathnames, descriptors, PIDs, and UIDs/GIDs values:

y	Abstraction is explicitly used
i	Abstraction is implicitly used

Restrictions on objects referenced by descriptors, pathnames:

b	Block special device
c	Character special device
d	Directory
f	Regular file
l	Symbolic Link
p	Pipe (pipes are also sockets)
s	Socket
t	Terminal device
|	Separates multiple permissible object types
!	Prefixes impermissible object types
,	Separates different restrictions on multiple arguments

Other abstractions used:

blk	Block device number
dev	Device number
fsys	File system
gid	Group identifier
hostid	Host identifier
mem	Process memory
mode	Access mode, sometimes including object type bits
pid	Process identifier
pgrp	Process group identifier
prio	Scheduling priority level
quota	File system quota information
regs	Processor registers
rlimits	Process resource limits
rusage	Process resource usage information
sig	Signal number
sigmask	Set of signal numbers
sigstack	Signal stack information
sockaddr	Socket address
sockopt	Socket option
status	Process exit status
tty	Terminal device
uid	User identifier

Appendix B

Toolkit Layers and Classes

This appendix presents the actual classes implemented in the interposition toolkit.

B.1. Base Toolkit Layer

The base toolkit layer primarily consists of a set of C routines that implement the bare minimum of the 4.3BSD system calls that must be intercepted by each agent, as per Section 5.6. It also contains the rest of the boilerplate described in Chapter 5 such as the loader and the signal emulation code.

The primary interfaces provided by the base toolkit layer are as follows:

```
extern int emul_fork(int rv[2]);
extern int emul_execv(int *args, int rv[2], void *regs);
extern int emul_ioctl(int d, unsigned long request, void *argp, int rv[2]);
extern int emul_execve(int *args, int rv[2], void *regs);
extern int emul_obreak(void *addr, int rv[2]);
extern int emul_sbrk(int delta, int rv[2]);
extern int emul_fcntl(int d, int cmd, int arg, int rv[2]);
extern int emul_sigblock(int mask, int rv[2]);
extern int emul_sigsetmask(int mask, int rv[2]);
extern int emul_sigpause(int mask, int rv[2]);
extern int emul_sigstack(struct sigstack *ss, struct sigstack *oss, int rv[2]);

extern int emul_sigreturn(int *args, int rv[2], void *regs);
extern int emul_sigvec(int *args, int rv[2], void *regs);
extern int emul_osigcleanup(int *args, int rv[2], void *regs);

extern void emul_handle_signal(int sig, int code, struct sigcontext *context);

extern int htg_syscall(int number, int args[], int rv[2],
        struct emul_regs *regs);
```

B.2. Numeric System Call Layer

The numeric system call layer views the system interface as consisting of vectors of untyped numbers. The 4.3BSD system call numbers and the numbers of arguments for each system call are listed in Section A.1.

The primary interfaces provided by the numeric system call class are as follows:

```
class numeric_syscall {
public:
    virtual int syscall(int number, int args[], int rv[2], void *regs);
    virtual void init(char *agentargv[]);
    virtual void signal_handler(int sig, int code, struct sigcontext *context);
    void register_interest(int number);
    void register_interest_range(int low, int high);
};
```

B.3. Symbolic System Call Layer

The symbolic system call layer views the system interface as consisting of a set of typed virtual functions. It is the first layer that provides sufficient abstraction to make it easy to write simple interposition agents.

The primary interfaces provided by the symbolic system call class are as follows:

```
class symbolic_syscall {
public:
    virtual void init(char *agentargv[]);
    virtual void init_child();

    virtual int sys_exit(int status, int rv[2]);
    virtual int sys_fork(int rv[2]);
    virtual int sys_read(int fd, void *buf, int cnt, int rv[2]);
    virtual int sys_write(int fd, void *buf, int cnt, int rv[2]);
    virtual int sys_open(char *path, int flags, int mode, int rv[2]);
    virtual int sys_close(int fd, int rv[2]);
    virtual int sys_creat(char *path, int mode, int rv[2]);
    virtual int sys_link(char *path, char *newpath, int rv[2]);
    virtual int sys_unlink(char *path, int rv[2]);
    virtual int sys_execv(int *args, int rv[2], struct emul_regs *regs);
    virtual int sys_chdir(char *path, int rv[2]);
    virtual int sys_mknod(char *path, int mode, int dev, int rv[2]);
    virtual int sys_chmod(char *path, int mode, int rv[2]);
    virtual int sys_chown(char *path, int user, int group, int rv[2]);
    virtual int sys_obreak(void *addr, int rv[2]);
    virtual int sys_lseek(int fd, off_t offset, int whence, int rv[2]);
    virtual int sys_getpid(int rv[2]);
    virtual int sys_mount(char *special, char *name, int rwflag, int rv[2]);
    virtual int sys_umount(char *special, int rv[2]);
    virtual int sys_getuid(int rv[2]);
    virtual int sys_ptrace(int request, int pid, void *addr, int data,
                    void *addr2, int rv[2]);
    virtual int sys_access(char *path, int mode, int rv[2]);
    virtual int sys_sync(int rv[2]);
    virtual int sys_kill(int pid, int sig, int rv[2]);
    virtual int sys_stat(char *path, struct stat *statbuf, int rv[2]);
    virtual int sys_lstat(char *path, struct stat *statbuf, int rv[2]);
    virtual int sys_dup(int oldd, int rv[2]);
    virtual int sys_pipe(int rv[2]);
    virtual int sys_profil(void *buff, int bufsiz, int offset, int scale,
                    int rv[2]);
    virtual int sys_getgid(int rv[2]);
    virtual int sys_acct(char *file, int rv[2]);
    virtual int sys_ioctl(int d, unsigned long request, char *argp, int rv[2]);
    virtual int sys_reboot(int howto, int rv[2]);
    virtual int sys_symlink(char *contents, char *newpath, int rv[2]);
    virtual int sys_readlink(char *path, void *buf, int cnt, int rv[2]);
    virtual int sys_execve(int *args, int rv[2], struct emul_regs *regs);
```

```
virtual int sys_umask(int mask, int rv[2]);
virtual int sys_chroot(char *path, int rv[2]);
virtual int sys_fstat(int fd, struct stat *statbuf, int rv[2]);
virtual int sys_getpagesize(int rv[2]);
virtual int sys_vfork(int rv[2]);
virtual int sys_vread(int fd, void *buf, int cnt, int rv[2]);
virtual int sys_vwrite(int fd, void *buf, int cnt, int rv[2]);
virtual int sys_sbrk(int incr, int rv[2]);
virtual int sys_mmap(void *addr, int len, int prot, int share,
                    int fd, off_t offset, int rv[2]);
virtual int sys_munmap(void *addr, int len, int rv[2]);
virtual int sys_vhangup(int rv[2]);
virtual int sys_getgroups(int gidsetlen, int *gidset, int rv[2]);
virtual int sys_setgroups(int ngroups, int *gidset, int rv[2]);
virtual int sys_getpgrp(int pid, int rv[2]);
virtual int sys_setpgrp(int pid, int pgrp, int rv[2]);
virtual int sys_setitimer(int which, struct itimerval *value,
                    struct itimerval *ovalue, int rv[2]);
virtual int sys_wait(int *args, int rv[2], struct emul_regs *regs);
virtual int sys_swapon(char *special, int rv[2]);
virtual int sys_getitimer(int which, struct itimerval *value, int rv[2]);
virtual int sys_gethostname(char *name, int namelen, int rv[2]);
virtual int sys_sethostname(char *name, int namelen, int rv[2]);
virtual int sys_getdtablesize(int rv[2]);
virtual int sys_dup2(int oldd, int newd, int rv[2]);
virtual int sys_fcntl(int fd, int cmd, int arg, int rv[2]);
virtual int sys_select(int nfds, fd_set *readfds, fd_set *writefds,
                    fd_set *exceptfds, struct timeval *timeout, int rv[2]);
virtual int sys_fsync(int fd, int rv[2]);
virtual int sys_setpriority(int which, int who, int prio, int rv[2]);
virtual int sys_socket(int domain, int type, int protocol, int rv[2]);
virtual int sys_connect(int s, struct sockaddr *name, int namelen,
                    int rv[2]);
virtual int sys_accept(int s, struct sockaddr *name, int *namelen,
                    int rv[2]);
virtual int sys_getpriority(int which, int who, int rv[2]);
virtual int sys_send(int s, void *msg, int len, int flags, int rv[2]);
virtual int sys_recv(int s, void *buf, int len, int flags, int rv[2]);
virtual int sys_sigreturn(int *args, int rv[2], struct emul_regs *regs);
virtual int sys_bind(int s, struct sockaddr *name, int namelen, int rv[2]);
virtual int sys_setsockopt(int s, int level, int optname,
                    void *optval, int optlen, int rv[2]);
virtual int sys_listen(int s, int backlog, int rv[2]);
virtual int sys_sigvec(int *args, int rv[2], struct emul_regs *regs);
virtual int sys_sigblock(int mask, int rv[2]);
virtual int sys_sigsetmask(int mask, int rv[2]);
virtual int sys_sigpause(int mask, int rv[2]);
virtual int sys_sigstack(struct sigstack *ss, struct sigstack *oss,
                    int rv[2]);
virtual int sys_recvmsg(int s, struct msghdr *msg, int flags, int rv[2]);
virtual int sys_sendmsg(int s, struct msghdr *msg, int flags, int rv[2]);
virtual int sys_gettimeofday(struct timeval *tp, struct timezone *tzp,
                    int rv[2]);
virtual int sys_getrusage(int who, struct rusage *rusage, int rv[2]);
virtual int sys_getsockopt(int s, int level, int optname,
                    void *optval, int *optlen, int rv[2]);
virtual int sys_readv(int fd, struct iovec *iov, int iovcnt, int rv[2]);
virtual int sys_writev(int fd, struct iovec *iov, int iovcnt, int rv[2]);
virtual int sys_settimeofday(struct timeval *tp, struct timezone *tzp,
                    int rv[2]);
virtual int sys_fchown(int fd, int user, int group, int rv[2]);
virtual int sys_fchmod(int fd, int mode, int rv[2]);
virtual int sys_recvfrom(int s, void *buf, int len, int flags,
                    struct sockaddr *from, int *fromlen, int rv[2]);
virtual int sys_setreuid(int ruid, int euid, int rv[2]);
virtual int sys_setregid(int rgid, int egid, int rv[2]);
virtual int sys_rename(char *oldpath, char *newpath, int rv[2]);
```

```
    virtual int sys_truncate(char *path, off_t offset, int rv[2]);
    virtual int sys_ftruncate(int fd, off_t offset, int rv[2]);
    virtual int sys_flock(int fd, int operation, int rv[2]);
    virtual int sys_sendto(int s, void *msg, int len, int flags,
                        struct sockaddr *to, int tolen, int rv[2]);
    virtual int sys_shutdown(int s, int how, int rv[2]);
    virtual int sys_socketpair(int domain, int type, int protocol, int sv[2],
                        int rv[2]);
    virtual int sys_mkdir(char *path, int mode, int rv[2]);
    virtual int sys_rmdir(char *path, int rv[2]);
    virtual int sys_utimes(char *path, struct timeval tvp[2], int rv[2]);
    virtual int sys_osigcleanup(int *args, int rv[2], struct emul_regs *regs);
    virtual int sys_adjtime(struct timeval *delta, struct timeval *olddelta,
                        int rv[2]);
    virtual int sys_getpeername(int s, struct sockaddr *name, int *namelen,
                        int rv[2]);
    virtual int sys_gethostid(int rv[2]);
    virtual int sys_sethostid(long hostid, int rv[2]);
    virtual int sys_getrlimit(int resource, struct rlimit *rlimit, int rv[2]);
    virtual int sys_setrlimit(int resource, struct rlimit *rlimit, int rv[2]);
    virtual int sys_killpg(int pgrp, int sig, int rv[2]);
    virtual int sys_setquota(char *special, char *file, int rv[2]);
    virtual int sys_quota(int cmd, int uid, int arg, void *addr, int rv[2]);
    virtual int sys_getsockname(int s, struct sockaddr *name, int *namelen,
                        int rv[2]);
    virtual int sys_getdirentries(int fd, void *buf, int cnt, long *basep,
                        int rv[2]);

    virtual int unknown_syscall(int number, int *args, int rv[2],
                        struct emul_regs *regs);

    virtual void signal_handler(int sig, int code, struct sigcontext *context);
};
```

B.4. Descriptor Management Classes

Descriptor management is done through three classes, `descriptor_set`, `descriptor`, and `open_object`. The `descriptor_set` class provides operations that affect the set of descriptors, i.e., those that allocate or deallocate descriptors. The `descriptor` class provides operations on the objects referenced by descriptors. The `open_object` provides for operations on reference counted open objects through multiple descriptors.

The primary interfaces provided by the `descriptor_set` class are as follows:

```
class descriptor_set {
public:
    // Descriptor set manipulation routines

    virtual void init(char *agentargv[], class DESC_SYMBOLIC_BASE *desc_sym);
    virtual DESCRIPTOR_CLASS *lookup(int d);
protected:
    virtual int alloc(int *d, DESCRIPTOR_CLASS *desc);
    virtual int alloc_geq(int *d, int lower_bound, DESCRIPTOR_CLASS *desc);
    virtual int enter(int *result, int d, DESCRIPTOR_CLASS *desc);
    virtual void remove(int d);

public:
    // System calls with knowledge of descriptors
```

```
    virtual int exit(int status, int rv[2]);
    virtual int fork(int rv[2]);
    virtual int open(char *path, int flags, int mode, int rv[2]);
    virtual int close(int fd, int rv[2]);
    virtual int execv(int *args, int rv[2], struct emul_regs *regs);
    virtual int dup(int oldd, int rv[2]);
    virtual int pipe(int rv[2]);
    virtual int execve(int *args, int rv[2], struct emul_regs *regs);
    virtual int vfork(int rv[2]);
    virtual int getdtablesize(int rv[2]);
    virtual int dup2(int oldd, int newd, int rv[2]);
    virtual int fcntl(int fd, int cmd, int arg, int rv[2]);
    virtual int select(int nfds, fd_set *readfds, fd_set *writefds,
                        fd_set *exceptfds, struct timeval *timeout, int rv[2]);
    virtual int socket(int domain, int type, int protocol, int rv[2]);
    virtual int accept(int s, struct sockaddr *name, int *namelen, int rv[2]);
    virtual int recvmsg(int s, struct msghdr *msg, int flags, int rv[2]);
    virtual int sendmsg(int s, struct msghdr *msg, int flags, int rv[2]);
    virtual int socketpair(int domain, int type, int protocol, int sv[2],
                        int rv[2]);

protected:
    // Member functions used internally

    virtual void close_on_exec();
    virtual void close_on_exit();
};
```

The primary interfaces provided by the `descriptor` class are as follows:

```
class descriptor {
public:
    // Object manipulation system call analogues

    virtual int read(void *buf, int cnt, int rv[2]);
    virtual int write(void *buf, int cnt, int rv[2]);
    virtual int close(int rv[2]);
    virtual int lseek(off_t offset, int whence, int rv[2]);
    virtual int dup(int rv[2]);
    virtual int ioctl(unsigned long request, char *argp, int rv[2]);
    virtual int fstat(struct stat *statbuf, int rv[2]);
    virtual int mmap(void *addr, int len, int prot, int share,
                        off_t offset, int rv[2]);
    virtual int dup2(int newd, int rv[2]);
    virtual int fcntl_dupfd(int lower_bound, int rv[2]);
    virtual int fcntl(int cmd, int arg, int rv[2]);
    virtual int fsync(int rv[2]);
    virtual int connect(struct sockaddr *name, int namelen,
                        int rv[2]);
    virtual int accept(struct sockaddr *name, int *namelen, int rv[2]);
    virtual int send(void *msg, int len, int flags, int rv[2]);
    virtual int recv(void *buf, int len, int flags, int rv[2]);
    virtual int bind(struct sockaddr *name, int namelen, int rv[2]);
    virtual int setsockopt(int level, int optname,
                        void *optval, int optlen, int rv[2]);
    virtual int listen(int backlog, int rv[2]);
    virtual int recvmsg(struct msghdr *msg, int flags, int rv[2]);
    virtual int sendmsg(struct msghdr *msg, int flags, int rv[2]);
    virtual int getsockopt(int level, int optname,
                        void *optval, int *optlen, int rv[2]);
    virtual int readv(struct iovec *iov, int iovcnt, int rv[2]);
    virtual int writev(struct iovec *iov, int iovcnt, int rv[2]);
    virtual int fchown(int user, int group, int rv[2]);
    virtual int fchmod(int mode, int rv[2]);
    virtual int recvfrom(void *buf, int len, int flags,
                        struct sockaddr *from, int *fromlen, int rv[2]);
```

```
    virtual int ftruncate(off_t offset, int rv[2]);
    virtual int flock(int operation, int rv[2]);
    virtual int sendto(void *msg, int len, int flags,
                    struct sockaddr *to, int tolen, int rv[2]);
    virtual int shutdown(int how, int rv[2]);
    virtual int getpeername(struct sockaddr *name, int *namelen,
                    int rv[2]);
    virtual int getsockname(struct sockaddr *name, int *namelen,
                    int rv[2]);
    virtual int getdirentries(void *buf, int cnt, long *basep, int rv[2]);

    // Member functions used internally

    virtual void implicit_close();
};
```

The primary interfaces provided by the `open_object` class are as follows:

```
class open_object {
protected:
    int reference_count;

public:
    // Object manipulation system call analogues

    virtual int read(void *buf, int cnt, int rv[2]);
    virtual int write(void *buf, int cnt, int rv[2]);
    virtual int close(int rv[2]);
    virtual int lseek(off_t offset, int whence, int rv[2]);
    virtual int ioctl(unsigned long request, char *argp, int rv[2]);
    virtual int fstat(struct stat *statbuf, int rv[2]);
    virtual int vread(void *buf, int cnt, int rv[2]);
    virtual int vwrite(void *buf, int cnt, int rv[2]);
    virtual int mmap(void *addr, int len, int prot, int share,
                    off_t offset, int rv[2]);
    virtual int fcntl(int cmd, int arg, int rv[2]);
    virtual int fsync(int rv[2]);
    virtual int connect(struct sockaddr *name, int namelen,
                    int rv[2]);
    virtual int accept(struct sockaddr *name, int *namelen, int rv[2]);
    virtual int send(void *msg, int len, int flags, int rv[2]);
    virtual int recv(void *buf, int len, int flags, int rv[2]);
    virtual int bind(struct sockaddr *name, int namelen, int rv[2]);
    virtual int setsockopt(int level, int optname,
                    void *optval, int optlen, int rv[2]);
    virtual int listen(int backlog, int rv[2]);
    virtual int recvmsg(struct msghdr *msg, int flags, int rv[2]);
    virtual int sendmsg(struct msghdr *msg, int flags, int rv[2]);
    virtual int getsockopt(int level, int optname,
                    void *optval, int *optlen, int rv[2]);
    virtual int readv(struct iovec *iov, int iovcnt, int rv[2]);
    virtual int writev(struct iovec *iov, int iovcnt, int rv[2]);
    virtual int fchown(int user, int group, int rv[2]);
    virtual int fchmod(int mode, int rv[2]);
    virtual int recvfrom(void *buf, int len, int flags,
                    struct sockaddr *from, int *fromlen, int rv[2]);
    virtual int ftruncate(off_t offset, int rv[2]);
    virtual int flock(int operation, int rv[2]);
    virtual int sendto(void *msg, int len, int flags,
                    struct sockaddr *to, int tolen, int rv[2]);
    virtual int shutdown(int how, int rv[2]);
    virtual int getpeername(struct sockaddr *name, int *namelen,
                    int rv[2]);
    virtual int getsockname(struct sockaddr *name, int *namelen,
                    int rv[2]);
    virtual int getdirentries(void *buf, int cnt, long *basep, int rv[2]);
```

```
    // Member functions used internally

    virtual void reference();
    virtual void implicit_close();
};
```

B.5. Pathname Management Classes

Pathname management is done through three classes, `pathname_set`, `pathname`, and `directory`. The `pathname_set` class provides operations that affect the set of pathnames, i.e., those that create or remove pathnames. The `pathname` class provides operations on the objects referenced by the pathnames. The `directory` class provides operations on open directories which list the names in the directory.

The primary interfaces provided by the `pathname_set` class are as follows:

```
class pathname_set : public descriptor_set {
protected:
    virtual int getpn(char *path, int flags, pathname **pn);

public:
    virtual void init(char *agentargv[], class PATH_SYMBOLIC_BASE *path_sym);
    // System calls with knowledge of pathnames

    virtual int open(char *path, int flags, int mode, int rv[2]);
    virtual int link(char *path, char *newpath, int rv[2]);
    virtual int unlink(char *path, int rv[2]);
    virtual int execv(int *args, int rv[2], struct emul_regs *regs);
    virtual int chdir(char *path, int rv[2]);
    virtual int mknod(char *path, int mode, int dev, int rv[2]);
    virtual int chmod(char *path, int mode, int rv[2]);
    virtual int chown(char *path, int user, int group, int rv[2]);
    virtual int mount(char *special, char *name, int rwflag, int rv[2]);
    virtual int umount(char *special, int rv[2]);
    virtual int access(char *path, int mode, int rv[2]);
    virtual int stat(char *path, struct stat *statbuf, int rv[2]);
    virtual int lstat(char *path, struct stat *statbuf, int rv[2]);
    virtual int acct(char *file, int rv[2]);
    virtual int symlink(char *contents, char *newpath, int rv[2]);
    virtual int readlink(char *path, void *buf, int cnt, int rv[2]);
    virtual int execve(int *args, int rv[2], struct emul_regs *regs);
    virtual int chroot(char *path, int rv[2]);
    virtual int swapon(char *special, int rv[2]);
    virtual int connect(int s, struct sockaddr *name, int namelen,
                        int rv[2]);
    virtual int bind(int s, struct sockaddr *name, int namelen, int rv[2]);
    virtual int rename(char *oldpath, char *newpath, int rv[2]);
    virtual int truncate(char *path, off_t offset, int rv[2]);
    virtual int mkdir(char *path, int mode, int rv[2]);
    virtual int rmdir(char *path, int rv[2]);
    virtual int utimes(char *path, struct timeval tvp[2], int rv[2]);
    virtual int setquota(char *special, char *file, int rv[2]);
};
```

The primary interfaces provided by the `pathname` class are as follows:

```
class pathname {
public:
    virtual int open(int flags, int mode, int rv[2], OPEN_OBJECT_CLASS **oo);
    virtual int link(pathname *newpn, int rv[2]);
```

```
    virtual int unlink(int rv[2]);
    virtual int chdir(int rv[2]);
    virtual int mknod(int mode, int dev, int rv[2]);
    virtual int chmod(int mode, int rv[2]);
    virtual int chown(int user, int group, int rv[2]);
    virtual int mount(pathname *dirpn, int rwflag, int rv[2]);
    virtual int umount(int rv[2]);
    virtual int access(int mode, int rv[2]);
    virtual int stat(struct stat *statbuf, int rv[2]);
    virtual int lstat(struct stat *statbuf, int rv[2]);
    virtual int acct(int rv[2]);
    virtual int symlink(char *contents, int rv[2]);
    virtual int readlink(void *buf, int cnt, int rv[2]);
    virtual int exec(char **argv, char **envp, int rv[2],
                    struct emul_regs *regs);
    virtual int chroot(int rv[2]);
    virtual int swapon(int rv[2]);
    virtual int connect_unix_domain(int s, int rv[2]);
    virtual int bind_unix_domain(int s, int rv[2]);
    virtual int rename(pathname *newpn, int rv[2]);
    virtual int truncate(off_t offset, int rv[2]);
    virtual int mkdir(int mode, int rv[2]);
    virtual int rmdir(int rv[2]);
    virtual int utimes(struct timeval tvp[2], int rv[2]);
    virtual int setquota(pathname *filepn, int rv[2]);
};
```

The primary interfaces provided by the `directory` class are as follows:

```
class directory : public OPEN_OBJECT_CLASS {
public:
    virtual int next_direntry();
    struct direct *direntry;  // Set by next_direntry()

public:
    virtual int read(void *buf, int cnt, int rv[2]);
    virtual int lseek(off_t offset, int whence, int rv[2]);
    virtual int getdirentries(void *buf, int cnt, long *basep, int rv[2]);
};
```

Appendix C
Specific Agents Constructed

This appendix presents a brief description of the agents constructed during this research.

C.1. hello_world

Uses: No toolkit code.

The `hello_world` agent prints its arguments and exits. It is used to test the `run` program and its use of the loader.

C.2. simple

Uses: Base toolkit layer.

The `simple` agent interposes on only the bare minimum of the 4.3BSD system calls that must be intercepted by each agent, as discussed in Section 5.6. It is used to test the low-level toolkit boilerplate and to provide best case toolkit implementation timings.

C.3. time_numeric

Uses: Numeric system call layer.

The `time_numeric` agent is used to time the numeric system call toolkit layer.

C.4. time_symbolic

Uses: Symbolic system call layer.

The `time_symbolic` agent is used to time the symbolic system call toolkit layer.

C.5. timex

Uses: Symbolic system call layer.

The `timex` agent makes it appear that the current time is sometime in the past.

C.6. trace

Uses: Symbolic system call layer.

The `trace` agent prints system call names, arguments, and results and signals received as programs are run. It is implemented as derived version of the symbolic system call class called `trace_symbolic` which prints calls, etc. as they are executed.

C.7. desc_symbolic

Uses: Simple descriptor management layer.

The `desc_symbolic` agent exercises the simple toolkit descriptor management classes. Exported descriptor numbers correspond to those in underlying implementation.

Related Agents: `desc_trace`, `trace_desc`.

The `desc_trace` and `trace_desc` agents use the `trace_symbolic` class to trace the calls being made by and upon the `desc_symbolic` agent, respectively. The tracing versions were helpful in debugging the original.

C.8. open_object

Uses: Full descriptor management layer.

The `open_object` agent exercises the toolkit descriptor management classes that perform reference counting of underlying open objects. Exported descriptor numbers are managed independently of those in underlying implementation.

Related Agents: `trace_oo`, `trace_oo_trace`.

The `trace_oo` and `trace_oo_trace` agents use the `trace_symbolic` class to trace the calls being made upon, and both being made upon and by the `open_object` agent, respectively. As in the simple descriptor management classes, the tracing versions were helpful in debugging the original.

C.9. pathname

Uses: Pathname management, descriptor management layers.

The `pathname` agent exercises the toolkit pathname management functions. Exported pathnames correspond to those in underlying implementation.

Related Agents: `trace_path`.

The `trace_path` agent adds tracing to the `pathname` agent.

C.10. dashed

Uses: Pathname management, descriptor management layers.

The `dashed` agent supports the use of ‘‘-’’ in pathnames with the VMS meaning (as an alias for ‘‘..’’). This exercises the toolkit pathname management functions, providing a simple test of augmenting the underlying implementation’s pathname semantics.

Related Agents: `trace_dashed`.

The `trace_dashed` agent adds tracing to the `dashed` agent.

C.11. union

Uses: Pathname management, descriptor management layers.

The `union` agent allows the contents of several directories to appear as a single directory. This provides an example of an agent that rearranges the pathname space relative to the underlying implementation.

Related Agents: `trace_union`.

The `trace_union` agent adds tracing to the `union` agent.

C.12. dfs_trace

Uses: Pathname management, descriptor management layers.

The `dfs_trace` agent implements a file reference tracing capability compatible with that provided by the DFSTrace [Mummert & Satyanarayanan 92] file reference tracing tools. This provides a basis for comparing an agent with a best available equivalent implementation that was independently produced.

Related Agents: `trace_dfs_trace`.

The `trace_dfs_trace` agent adds tracing to the `dfs_trace` agent.

Appendix D

Example Agent Code

This appendix provides some examples of actual agent code.

D.1. Timex Agent

The following code samples are from the `timex` agent:

```
extern "C" {
int parsedate(char *datestr, struct tm *tm, int currtm, int past, int error);
long gtime (struct tm *tm);
}


void timex_symbolic_syscall::init(char *agentargv[])
{
    int rv[2];
    struct timeval tv;
    symbolic_syscall::init(agentargv);
    (void) symbolic_syscall::sys_gettimeofday(&tv, 0, rv);

    if (agentargv[1] != 0) {
        struct tm tms;
        long funkytime;
        if (parsedate(agentargv[1], &tms, 1, 1, 1) != 0) {
            fprintf(stderr, "timex:  Can’t parse time \"%s\"\n", agentargv[1]);
            exit(1);
        }
        funkytime = gtime(&tms);
        offset = funkytime - tv.tv_sec;
    } else {
        srandom(tv.tv_sec + tv.tv_usec);
        offset = - (random() & 0x1fffffff);     // ’70s or ’80s anyone?
    }

    printf("[ Actual time = %d, offset = %d ]\n", tv.tv_sec, offset);
}

int timex_symbolic_syscall::sys_gettimeofday(struct timeval *tp,
                        struct timezone *tzp, int rv[2])
{
    int ret;
#if     DEBUG
    printf("[ sys_gettimeofday(0x%x, 0x%x) ]\n", tp, tzp);
#endif  DEBUG
    ret = symbolic_syscall::sys_gettimeofday(tp, tzp, rv);
    if (ret >= 0 && tp) {
        tp->tv_sec += offset;
    }
    return ret;
```

```
}
```

D.2. Trace Agent

The following code samples are from the `trace` agent:

```
class TRACE_SYMBOLIC_CLASS : public TRACE_SYMBOLIC_BASE {
public:
    virtual void init(char *agentargv[]);
    virtual void init_child();

    virtual int sys_exit(int status, int rv[2]);
    virtual int sys_fork(int rv[2]);
    virtual int sys_read(int fd, void *buf, int cnt, int rv[2]);
    virtual int sys_write(int fd, void *buf, int cnt, int rv[2]);
    virtual int sys_open(char *path, int flags, int mode, int rv[2]);
    virtual int sys_close(int fd, int rv[2]);
    ...
private:
    int trace_fd;       // dup()ed copy of stdout's descriptor
    int dtablesize;     // Simulated descriptor table size
    FILE *f;            // Trace output stream
    int trace_pid;      // Cached copy of process pid

    void print_start();
    void print_ret(int ret, int rv[2]);
    void print_retx(int ret, int rv[2]);
    void print_reto(int ret, int rv[2]);
    void print_ret2(int ret, int rv[2]);
    void print_fdset(int nfds, fd_set *fds);
};

void TRACE_SYMBOLIC_CLASS::init_child()
{
    TRACE_SYMBOLIC_BASE::init_child();
    trace_pid = getpid();
}

int TRACE_SYMBOLIC_CLASS::sys_exit(int status, int rv[2])
{
    register int ret;
    print_start();
    fprintf(f, "_exit(%d) ... ]\n", status);
    fflush(f);
    ret = TRACE_SYMBOLIC_BASE::sys_exit(status, rv);
    print_start();
    fprintf(f, "... _exit(%d) ->", status);
    print_ret(ret, rv);
    return ret;
}

int TRACE_SYMBOLIC_CLASS::sys_fork(int rv[2])
{
    register int ret;
    print_start();
    fprintf(f, "fork() ... ]\n");
    fflush(f);
    ret = TRACE_SYMBOLIC_BASE::sys_fork(rv);
    print_start();
    fprintf(f, "... fork() ->");
    print_ret2(ret, rv);
    return ret;
}

int TRACE_SYMBOLIC_CLASS::sys_read(int fd, void *buf, int cnt, int rv[2])
```

```
{
    register int ret;
    print_start();
    fprintf(f, "read(%d, 0x%x, 0x%x) ... ]\n", fd, buf, cnt);
    fflush(f);
    ret = TRACE_SYMBOLIC_BASE::sys_read(fd, buf, cnt, rv);
    print_start();
    fprintf(f, "... read(%d, 0x%x, 0x%x) ->", fd, buf, cnt);
    print_retx(ret, rv);
    return ret;
}

int TRACE_SYMBOLIC_CLASS::sys_write(int fd, void *buf, int cnt, int rv[2])
{
    register int ret;
    print_start();
    fprintf(f, "write(%d, 0x%x, 0x%x) ... ]\n", fd, buf, cnt);
    fflush(f);
    ret = TRACE_SYMBOLIC_BASE::sys_write(fd, buf, cnt, rv);
    print_start();
    fprintf(f, "... write(%d, 0x%x, 0x%x) ->", fd, buf, cnt);
    print_retx(ret, rv);
    return ret;
}

int TRACE_SYMBOLIC_CLASS::sys_open(char *path, int flags, int mode, int rv[2])
{
    register int ret;
    print_start();
    fprintf(f, "open(\"%s\", 0%o, 0%o) ->", path, flags, mode);
    fflush(f);
    ret = TRACE_SYMBOLIC_BASE::sys_open(path, flags, mode, rv);
    print_ret(ret, rv);
    return ret;
}

int TRACE_SYMBOLIC_CLASS::sys_close(int fd, int rv[2])
{
    register int ret;
    print_start();
    fprintf(f, "close(%d) ->", fd);
    if (fd >= dtablesize) {
        fprintf(f, " (fd >= simulated dtablesize %d, ignored)", dtablesize);
        ret = EBADF;
    } else {
        fflush(f);
        ret = TRACE_SYMBOLIC_BASE::sys_close(fd, rv);
    }
    print_ret(ret, rv);
    return ret;
}

...

void TRACE_SYMBOLIC_CLASS::print_start()
{
    fprintf(f, "[ %d: ", trace_pid);
}

void TRACE_SYMBOLIC_CLASS::print_ret(int ret, int rv[2])
{
    if (ret == ESUCCESS) {
        fprintf(f, " %d ]\n", rv[0]);
    } else if (ret == E_JUSTRETURN) {
        fprintf(f, " JUSTRETURN ]\n");
    } else {
        char *errstr = strerror(ret);
```

```
        if (errstr)
            fprintf(f, " errno=%d (%s) ]\n", ret, errstr);
        else
            fprintf(f, " errno=%d ]\n", ret);
    }
}
```

Appendix E

Examples of Agent Usage

This appendix contains transcripts of several simple sessions in which interposition agents were used.

This session shows two uses of the `trace` agent:

```
% run trace.agent sync
[ 8364: execve("/bin/sync", 0xbffff87c, 0xbffff884) ... ]
[ 8364: ... execve(...) -> JUSTRETURN ]
[ 8364: sync() -> 0 ]
[ 8364: _exit(0) ... ]
%
% run trace.agent echo Watch echo run!
[ 8366: execve("/bin/echo", 0xbffff860, 0xbffff874) ... ]
[ 8366: ... execve(...) -> JUSTRETURN ]
[ 8366: fstat(1, 0xbffff3dc) -> 0 ]
[ 8366: getpagesize() -> 0x1000 ]
[ 8366: obreak(0x113a0) -> 0 ]
[ 8366: obreak(0x11ffc) -> 0 ]
[ 8366: obreak(0x14ffc) -> 0 ]
[ 8366: ioctl(1, TIOCGETP, 0xbffff3b4) -> 0 ]
[ 8366: write(1, 0x12000, 0x10) ... ]
Watch echo run!
[ 8366: ... write(1, 0x12000, 0x10) -> 0x10 ]
[ 8366: close(0) -> 0 ]
[ 8366: close(1) -> 0 ]
[ 8366: close(2) -> 0 ]
[ 8366: _exit(0) ... ]
```

This session shows two uses of the `timex` agent:

```
% date
Wed Aug 19 02:36:11 EDT 1992
%
% run timex.agent date
[ Actual time = 714206177, offset = -355386281 ]
Fri May 15 20:11:36 EDT 1981
%
% run timex.agent 'July 4, 1976 00:00' -- date
[ Actual time = 714206181, offset = -508901781 ]
Sun Jul  4 01:00:00 EDT 1976
```

This session shows a use of the `union` agent:

```
% ls /usr/mbj/bin
ac              gdb.ss          mtime           tabconvert      without
clnztime        guess_term      notice          task_resume     x11.commands
cmp_mtime       intersection    ok              task_suspend    x11.start
copy            kermit          package         thread_abort    xroach
countargs       killer_bee      port_alias      thread_resume   xshow
date            lastlog         reset_tty       thread_suspend  xt
date_daemon     loop            run             tidy
dev_lookup      lt              shar            tidy_week
dev_register    merge4          stat            unpackage
flip            merge5          sunclock        vm_regions
%
% echo $path
/usr1/mbj/bin /afs/cs.cmu.edu/user/mbj/bin /usr/local/bin /usr/ucb /bin
 /usr/bin /usr/X11/bin /usr/vice/bin /usr/local/sdm/bin /usr/local/rcs/bin
%
% run union.agent ‘echo $path‘ -- ls /usr/mbj/bin
Mail            editres         makedepend      rwho            utimes
X               egrep           makepath        rxgen           uudecode
Xblit           emacs           man             sccs            uuencode
Xvga            enroll          maze            sccstorcs       uwm
[               eqn             md              scout           vacation
ac              error           merge           script          vgrind
addbib          ex              merge4          sed             vi
aedplot         expand          merge5          sendbug         view
alarm           explain         mesg            service         vm_regions
apply           expr            mig             setmodes        vm_stat
appres          f               mkdep           sh              vpasswd
apropos         false           mkdir           shar            w
ar              fgcc            mkdirhier       showrgb         w.iv25
ar11            fgrep           mkfontdir       showsnf         wall
as              file            mkstr           size            washtool
assign          find            more            sleep           wc
at              finger          mset            snapshot        wh
atobm           flip            msgs            soelim          what
atoplot         fmt             mt              sort            whatabout
atq             fold            mtime           sortbib         whatis
atrm            fpr             muncher         spell           whenis
awk             from            mv              spellin         whereis
basename        fs              name_version    spellout        which
```

... 52 lines of text omitted ...

```
crypt           lex             rcsit           tp              xrefresh
csh             lint            rcsmerge        tr              xrn
ctags           listfiles       rcsstat         translate_et    xroach
cu              listres         rcstime         troff           xrotmap
date            ln              rdist           true            xscope
date_daemon     lock            refer           tset            xsend
dc              log             release         tsort           xset
dd              logger          reset           tty             xsetroot
deassign        login           reset_tty       twm             xshow
delay           look            resize          u               xshowcmap
deliv           lookbib         rev             udebug          xstdcmap
deroff          loop            rgen            ul              xstr
dev_lookup      lorder          rlog            uncompress      xt
dev_register    lpq             rlogin          unexpand        xterm
df              lpr             rm              unifdef         xwd
diction         lprm            rmail           uniq            xwininfo
diff            lptest          rmdir           units           xwud
diff3           ls              roffbib         unlog           yacc
du              lt              rpcgen          unloose         yes
dumbplot        m4              rsh             unpackage       zcat
e               machine         ru              up
echo            mail            run             uptime
ed              mailq           runat           uptime.iv25
edit            make            ruptime         users
```

Appendix F
Sizes of Toolkit Layers and Agents

This appendix presents a detailed look at the source and binary sizes of the toolkit and agents.

F.1. Statements of Code Per Toolkit Layer

Toolkit Statements by Layer			
Toolkit Layer	*Machine Independent*	*Machine Dependent (Intel 386)*	*Total Statements*
Loader	565	71	636
Base	785	195	980
Numeric	104	39	143
Symbolic	667	41	708
Trace	1305	43	1348
Descriptor	635	0	635
Open Object	311	0	311
Pathname	485	0	485
Directory	79	0	79

Table F-1: Toolkit statement counts listed by layer

F.2. Statements of Agent Specific Code

Agent Specific Statements			
Agent Name	*Machine Independent*	*Machine Dependent (Intel 386)*	*Total Statements*
`simple`	62	76	138
`time_numeric`	6	0	6
`time_symbolic`	7	0	7
`timex`	35	0	35
`trace`	1305	43	1348
`desc_symbolic`	8	0	8
`open_object`	0	0	0
`pathname`	8	0	8
`dashed`	52	0	52
`union`	166	0	166
`dfs_trace`	1040	0	1040

Table F-2: Agent specific statement counts

F.3. Sizes of Agent Binaries

Sizes of Agent Binaries				
Agent Name	*Code Bytes*	*Data Bytes*	*Zero (bss) Bytes*	*Total Agent Bytes*
`simple`	53216	5184	1028	59428
`time_numeric`	57312	5896	916	64124
`time_symbolic`	69600	6848	916	77364
`timex`	85984	12692	7468	106144
`trace`	98272	7064	936	106272
`desc_symbolic`	81888	8268	916	91072
`open_object`	81888	10900	916	93704
`pathname`	94176	14008	916	109100
`dashed`	94176	16280	916	111372
`union`	98272	17448	944	116664
`dfs_trace`	126944	18124	3148	148216

Table F-3: Sizes of Agent Binaries

References

[Accetta et al. 86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young.
Mach: A New Kernel Foundation for UNIX Development.
In *Proc. Summer 1986 USENIX Technical Conference and Exhibition*. June, 1986.

[Apple 88] *Macintosh System Software User's Guide Version 6.0*.
Apple Computer, Inc., 1988.

[Apple 90] *Macintosh MacApp 2.0 General Reference*.
Apple Computer, Inc., 1990.

[AT&T 86] *System V Interface Definition, Issue 2*.
AT&T, Customer Information Center, P.O. Box 19901, Indianapolis, IN 46219, 1986.

[AT&T 89] *Unix System V Release 4.0 Programmer's Reference Manual*.
AT&T, 1989.

[Baron et al. 90] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young.
Mach Kernel Interface Manual.
Carnegie Mellon University School of Computer Science, 1990.

[BBN 71] *TENEX JSYS Manual — A Manual of TENEX Monitor Calls*.
BBN Computer Science Division, BBN, Cambridge, Mass., 1971.

[Bershad & Pinkerton 88]
Brian N. Bershad and C. Brian Pinkerton.
Watchdogs: Extending the Unix Filesystem.
In *Winter Usenix Conference Proceedings*. Dallas, 1988.

[Black 90] David Black.
The Mach Timing Facility: An Implementation of Accurate Low-Overhead Usage Timing.
In *Usenix Mach Workshop Proceedings*. Burlington, October, 1990.

[Black et al. 88] David L. Black, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young.
The Mach Exception Handling Facility.
Technical Report CMU-CS-88-129, Carnegie Mellon University School of Computer Science, April, 1988.

[Blotcky et al. 86] Blotcky, S., Lynch, K., and Lipner, S.
SE/VMS: Implementing Mandatory Security in VAX/VMS.
In *Proceedings of the 9th National Computer Security Conference*, pages 47-54. National Bureau of Standards, Gaithersburg, MD, 1986.

[Bobrow et al. 72] Bobrow, D.G., Burchfiel, J.D., Murphy, D.L. and Tomlinson, R.S.
TENEX, a paged time sharing system for the PDP-10.
Communications of the ACM 15(3):135-143, March, 1972.

[Brooks 75] Frederick P. Brooks, Jr.
The Mythical Man-Month — Essays on Software Engineering.
Addison-Wesley, 1975.

[Brown 92] Mark R. Brown et al.
Bridges: Tools to extend the Vesta configuration management system.
1992.
In preparation.

[Brownbridge et al. 82]
D.R. Brownbridge, L.F. Marshall, B. Randell.
The Newcastle Connection, or UNIXes of the World Unite!
Software — Practice and Experience 12:1147-1162, 1982.

[Campbell et al. 87]
Roy H. Campbell, Vincent Russo, and Gary Johnston.
Choices: The Design of a Multiprocessor Operating System.
In *Proceedings of the USENIX C++ Workshop*, pages 109-123. Santa Fe, New Mexico, November, 1987.

[Cheriton 88] David R. Cheriton.
The V distributed system.
Communications of the ACM 31(3):314-333, March, 1988.

[Clayton 90] Leigh Clayton.
Interposition in MVS.
April, 1990.
Personal communication.

[Clegg et al. 86] Clegg, F. W., Ho, G. S.-F., Kusmar, S. R., and Sontag, J. R.
The HP-UX Operating System on HP Precision Architecture Computers.
Hewlett-Packard Journal 37(12):4-22, December, 1986.

[Colvin 90] Alex Colvin.
Interposition in DTSS.
April, 1990.
Personal communication.

[Cooper & Draves 88]
Eric C. Cooper, Richard. P. Draves.
C Threads.
Technical Report CMU-CS-88-154, Carnegie Mellon University Computer Science Department, June, 1988.

[DCA/Intel 91] *DCA/Intel Communicating Applications Specification Version 1.2A*. Digital Communications Associates, Inc. and Intel Corporation, Intel Part Number: 301812-005, 1991.

[Denning 82] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

[Digital 81a] *VAX Hardware Handbook*. Digital Equipment Corporation, 1980-81.

[Digital 78] *DECSYSTEM-20 Monitor Calls Reference Manual*. Digital Equipment Corporation, 1978.

[Digital 81b] *VAX Architecture Handbook*. Digital Equipment Corporation, 1981.

[Digital 85] *VMS System Software Handbook*. Digital Equipment Corporation, 1985.

[Digital 89a] *ULTRIX Reference Pages, Section 1 (Commands)*. Digital Equipment Corporation, 1989.

[Digital 89b] *ULTRIX Reference Pages, Section 2 (System Calls)*. Digital Equipment Corporation, 1989.

[DoD 85] *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense Computer Security Center, Fort Meade, MD, CSC-STD-001-83, 1985.

[Douglis & Ousterhout 91] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software — Practice and Experience* 21(8):757-785, August, 1991.

[Draves 90] Richard Draves. A Revised IPC Interface. In *Usenix Mach Workshop Proceedings*. Burlington, October, 1990.

[Draves et al. 89] Richard P. Draves, Michael B. Jones, Mary R. Thompson. *MIG — The MACH Interface Generator*. Carnegie-Mellon University Computer Science Department, 1989.

[Druschel et al. 91] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Service Composition in Lipto. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 108-111. Palo Alto, October, 1991.

[Druschel et al. 92] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson.
Beyond micro-kernel design: Decoupling modularity and protection in Lipto.
In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 114-117. Yokohama, Japan, June, 1992.

[Duncan 90] Ray Duncan, Charles Petzold, M. Steven Baker, Andrew Schulman, Stephen R. Davis, Ross P. Nelson, Robert Moote.
Extending DOS.
Addison-Wesley, 1990.

[Eastlake et al. 69] D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, S. Nelson.
ITS 1.5 Reference Manual.
Memorandum no. 161, M.I.T. Artificial Intelligence Laboratory, July, 1969.
(Revised form of ITS 1.4 Reference Manual, June 1968).

[Feldman 79] S. I. Feldman.
Make — a program for maintaining computer programs.
Software — Practice and Experience 9(4):255-265, 1979.

[Forsdick et al. 78] Harry C. Forsdick, Richard E. Schantz, Robert H. Thomas.
Operating Systems for Computer Networks.
Computer :48-57, January, 1978.

[Garner et al. 88] Robert B. Garner, Anant Agrawal, Fayé Briggs, Emil W. Brown, David Hough, Bill Joy, Steve Kleiman, Steven Muchnick, Masood Namjoo, Dave Patterson, Joan Pendleton, & Richard Tuck.
The Scalable Processor Architecture (SPARC).
In *Proceedings of the Thirty-Third IEEE Computer Society International Conference (Compcon Spring 88)*, pages 278-283. March, 1988.

[Gasser & McDermott 90]
Morrie Gasser and Ellen McDermott.
An Architecture for Practical Delegation in a Distributed System.
In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1990.

[Gingell 90] Rob Gingell.
Interposition in SunOS.
April, 1990.
Personal communication.

[Golub et al. 90] David Golub, Randall Dean, Alessandro Forin, Richard Rashid.
Unix as an Application Program.
In *Summer Usenix Conference Proceedings*. Anaheim, June, 1990.

[Gomes 90] Ron Gomes.
Interposition in System V R4.
March, 1990.
Personal communication.

[Gosling 82] James Gosling.
Unix Emacs.
Carnegie-Mellon University Computer Science Department, 1982.

[Grampp & Morris 84]
F. T. Grampp and R. H. Morris.
UNIX Operating System Security.
AT&T Bell Laboratories Technical Journal 63(8), October, 1984.

[Guedes & Julin 91]
Paulo Guedes and Daniel Julin.
Object-Oriented Interfaces in the Mach 3.0 Multi-Server System.
In *Proceedings of the Second International Workshop on Object-Orientation in Operating Systems*. Palo Alto, October, 1991.

[Hendricks 90] Hendricks, David.
A Filesystem For Software Development.
In *Summer Usenix Conference Proceedings*, pages 333-340. June, 1990.

[Howard et al. 88] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J.
Scale and Performance in a Distributed File System.
ACM Transactions on Computer Systems 6(1), February, 1988.

[IBM 87] *Access Method Services, OS/VS2*.
IBM Corporation, 1987.
GC26-3841.

[IEEE 90] *IEEE Standard Portable Operating System Interface for Computer Environments (POSIX)*.
The Institute of Electrical and Electronics Engineers, Inc., 1990.
IEEE Std 1003.1-1988.

[Ingalls 80] Daniel H. H. Ingalls.
The Smalltalk-76 Programming System Design and Implementation.
Technical Report, Xerox Palo Alto Research Center, Palo Alto, CA, 1980.

[Intel 86] *80386 Programmer’s Reference Manual*.
Intel Corporation, 1986.

[Intel 90] *i486 Microprocessor Programmer’s Reference Manual*.
Intel Corporation, 1990.

[Jones 92] Michael B. Jones.
Inheritance in Unlikely Places: Using Objects to Build Derived Implementations of Flat Interfaces.
In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*. Paris, September, 1992.
To appear.

[Joy et al. 83] William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick.
4.2BSD System Manual.
Computer Systems Research Group, Computer Science Division, University of California, Berkeley, 1983.

[Julin et al. 91] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, Paul Roy.
Generalized Emulation Services for Mach 3.0: Overview, Experiences and Current Status.
In *Usenix Mach Symposium Proceedings*. Monterey, November, 1991.

[Kane 87] Gerry Kane.
MIPS R2000 RISC Architecture.
Prentice Hall, 1987.

[Kernighan & Ritchie 78]
Brian W. Kernighan, Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[Kistler & Satyanarayanan 92]
Kistler, J.J. and Satyanarayanan, M.
Disconnected Operation in the Coda File System.
ACM Transactions on Computer Systems 10(1), February, 1992.

[Kleiman 86] Kleiman, S.R.
Vnodes: An Architecture for Multiple File System Types in Sun UNIX.
In *Summer Usenix Conference Proceedings*. Atlanta, 1986.

[Koch & Gelhar 86]
Philip Koch and David Gelhar.
DTSS System Programmer's Reference Manual.
Dartmouth College, Hanover, NH, 1986.
Kiewit Computation Center TM059.

[Korn & Krell 90] Korn, David G. and Krell, Eduardo.
A New Dimension for the Unix File System.
Software — Practice and Experience 20(S1):19-34, Jun, 1990.

[Lamport 88] Leslie Lamport.
Concurrent Reading and Writing of Clocks.
Research Report 27, Digital Equipment Corporation, Systems Research Center, April, 1988.

[Lampson 73] B. W. Lampson.
A Note on the Confinement Problem.
Communications of the ACM 16(10):613-615, October, 1973.

[Lampson et al. 91]
Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber.
Authentication in Distributed Systems: Theory and Practice.
In *Proceedings of the 13th ACM Symposium on Operating System Principles*. October, 1991.

[Lee 89] Ruby B. Lee.
Precision Architecture.
IEEE Computer 22(1):78-91, January, 1989.

[Leffler et al. 90] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman.
The Design and Implementation of the 4.3BSD UNIX Operating System.
Addison-Wesley, 1990.

[Macrakis 90] Stavros Macrakis.
Interposition in ITS.
April, 1990.
Personal communication.

[Massalin 92] Henry Massalin.
Synthesis: An Efficient Implementation of Fundamental Operating System Services.
PhD thesis, Columbia University, 1992.

[McJones & Swart 87]
Paul R. McJones, Garret F. Swart.
Evolving the UNIX System Interface to Support Multithreaded Programs.
Research Report 21, Digital Equipment Corporation, Systems Research Center, September, 1987.

[McKusick et al. 84]
McKusick, M.K., Joy, W.N., Leffler, S.J., Fabry, R.S.
A Fast File System for Unix.
ACM Transactions on Computer Systems 2(3), August, 1984.

[Microsoft 87] *Microsoft Windows User's Guide*.
Microsoft Corporation, 1987.

[Microsoft 91] *Microsoft MS-DOS Operating System version 5.0 User's Guide and Reference*.
Microsoft Corporation, 1991.

[MIPS 91] *MIPS R4000 Microprocessor User's Manual*.
First edition, MIPS Computer Systems, Inc., 1991.

[Mummert 92] Lily B. Mummert.
Development of the DFSTrace File Reference Tracing Tools.
June, 1992.
Interviewed by Michael B. Jones.

[Mummert & Satyanarayanan 92]
Lily B. Mummert and M. Satyanarayanan.
Efficient and Portable File Reference Tracing in a Distributed Workstation Environment.
June, 1992.
To be published as a Carnegie Mellon University School of Computer Science technical report.

[Norton 90] *The Norton Utilities for the Macintosh*.
Peter Norton Computing, Incorporated, 1990.

[Now 90] *Now Utilities: File & Application Management, System Management, and System Extensions*.
Now Software, Inc., 1990.

[Organick 72] Organick, E.I.
The Multics System: An Examination of its Structure.
MIT Press, 1972.

[OSF 90] Open Software Foundation.
Application Environment Specification (AES), Operating System Programming Interfaces Volume.
Prentice Hall, Inc., 1990.
ISBN 0-13-043522-8.

[Ousterhout et al. 88]
Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N., Welch, B.B.
The Sprite Network Operating System.
Computer 21(2), February, 1988.

[Parmelee et al. 72]
R. P. Parmelee, T. I. Peterson, C. C. Tillman and D. J. Hatfield.
Virtual Storage and Virtual Machine Concepts.
IBM Systems Journal 11(2):99-130, 1972.

[Rabin & Tygar 87] Michael O. Rabin and J.D. Tygar.
An Integrated Toolkit for Operating System Security.
Technical Report TR-05-87, Harvard University Center for Research in Computing Technology, Cambridge, MA, May, 1987.
Revised August 1988.

[Rashid & Robertson 81]
Rashid, R. F. and Robertson, G.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 64-75. December, 1981.

[Reid & Walker 80] Brian K. Reid and Janet H. Walker.
SCRIBE Introductory User's Manual.
Third edition, UNILOGIC, Ltd., 1980.

[Richards 69a] Martin Richards.
BCPL: A tool for compiler writing and system programming.
In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 557-566. May, 1969.

[Richards 69b] Martin Richards.
The BCPL reference manual.
Technical Memorandum 69/1, The University Mathematical Laboratory, Cambridge, England, January, 1969.

[Ritchie 79] Dennis M. Ritchie.
Protection of Data File Contents.
U.S. Patent 4135240.
January, 1979

[Salient 91] *DiskDoubler User's Manual*.
Salient Software, Inc., 1991.

[Sandberg et al. 85]
Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B.
Design and Implementation of the Sun Network Filesystem.
In *Summer Usenix Conference Proceedings*. Portland, 1985.

[Satyanarayanan 91]
M. Satyanarayanan, Richard Draves, James Kistler, Anders Klemets, Qi Lu, David Nichols, Larry Raper, Gowthami Rajendran, Jonathan Rosenberg, Ellen Siegel.
RPC2 User Guide and Reference Manual.
Carnegie Mellon University School of Computer Science, 1991.

[Satyanarayanan et al. 90]
Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.
Coda: A Highly Available File System for a Distributed Workstation Environment.
IEEE Transactions on Computers 39(4), April, 1990.

[Scheifler & Gettys 86]
R. W. Scheifler and J. Gettys.
The X Window System.
ACM Transactions on Graphics 5(2), 1986.

[Schilit 90] Bill Schilit.
Interposition in Tops-20.
April, 1990.
Personal communication.

[Sites 92] Richard L. Sites (editor).
Alpha Architecture Reference Manual.
Digital Press, 1992.

[Stac 92] *Stacker 2.1 User Guide*.
Stac Electronics, Inc., 1992.

[Stallman 90] Richard M. Stallman.
Using and Porting GNU CC, for version 1.37.
Free Software Foundatation, Inc., 1990.

[Stoy 92] Joe Stoy.
Interposition in OS6.
May, 1992.
Personal communication.

[Stoy & Strachey 72a]
J. E. Stoy and C. Strachey.
OS6 — An experimental operating system for a small computer.
Part 1: General principles and structure.
Computer Journal 15(2):117-124, May, 1972.

[Stoy & Strachey 72b]
J. E. Stoy and C. Strachey.
OS6 — An experimental operating system for a small computer.
Part 2: Input/output and filing system.
Computer Journal 15(3):195-203, August, 1972.

[Stroustrup 87] Bjarne Stroustrup.
The C++ Programming Language.
Addison-Wesley, 1987.

[Sturgis 74] Howard Ewing Sturgis.
A Postmortem for a Time Sharing System.
Xerox Research Report CSL-74-1, Xerox Palo Alto Research Center, January, 1974.

[Sun 86] *Networking on the SUN Workstation*.
Sun Microsystems, Inc., 1986.
800-1324-03.

[Sun 88a] *SunOS Reference Manual*.
Sun Microsystems, Inc., 1988.
Part No. 800-1751-10.

[Sun 88b] *Network Software Environment Reference Manual*.
Sun Microsystems, Inc., 1988.

[Swart 92] Garret Swart.
Interposition in Taos.
July, 1992.
Personal communication.

[Symantec 91a] *Symantec AntiVirus for Macintosh*.
Symantec Corporation, 1991.

[Symantec 91b] *The Norton AntiVirus*.
Symantec Corporation, 1991.

[Thacker et al. 87] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr.
Firefly: A Multiprocessor Workstation.
Research Report 23, Digital Equipment Corporation, Systems Research Center, December, 1987.

[Thomas 73] Robert H. Thomas.
A resource sharing executive for the ARPANET.
In *Proceedings of the AFIPS National Computer Conference*, pages 155-163. June, 1973.

[Thomas 75] Robert H. Thomas.
JSYS Traps — A Tenex Mechanism for Encapsulation of User Processes.
In *Proceedings of the AFIPS National Computer Conference*, pages 351-360. 1975.

[Trend 90] *PC-cillin Virus Immune System User's Manual*.
Trend Micro Devices, Incorporated, 1990.

[Tygar & Yee 91] J. D. Tygar, Bennet Yee.
Dyad: A System for Using Physically Secure Coprocessors.
Technical Report CMU-CS-91-140R, Carnegie Mellon University School of Computer Science, May, 1991.

[Walsh et al. 85] Walsh, D., Lyon, B., Sager, G., Chang, J.M., Goldberg, D., Kleiman, S., Lyon, T., Sandberg, R., Weiss, P.
Overview of the Sun Network Filesystem.
In *Winter Usenix Conference Proceedings*. Dallas, 1985.

[Wohl 90] Aaron Wohl.
Interposition in Tops-20.
May, 1990.
Personal communication.

[Young et al. 87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black and Robert Baron.
The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.
In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. Austin, November, 1987.

Keywords: Operating systems, system interface, interposition toolkit, interposition agents.

Abstract

Many contemporary operating systems utilize a system call interface between the operating system and its clients. Increasing numbers of systems are providing low-level mechanisms for intercepting and handling system calls in user code. Nonetheless, they typically provide no higher-level tools or abstractions for effectively utilizing these mechanisms. Using them has typically required reimplementation of a substantial portion of the system interface from scratch, making the use of such facilities unwieldy at best.

This dissertation presents a toolkit that substantially increases the ease of interposing user code between clients and instances of the system interface by allowing such code to be written in terms of the high-level objects provided by this interface, rather than in terms of the intercepted system calls themselves. This toolkit helps enable new interposition agents to be written, many of which would not otherwise have been attempted.

This toolkit has also been used to construct several agents including: system call tracing tools, file reference tracing tools, and customizable filesystem views. Examples of other agents that could be built include: protected environments for running untrusted binaries, logical devices implemented entirely in user space, transparent data compression and/or encryption agents, transactional software environments, and emulators for other operating system environments.

Trademarks

Acknowledgements

To Trish, my wife,
for her constant love,
and willing assistance.

I want to thank my parents, Robert and Lois Jones, and the rest of my family for their love, for their faith, and for showing me that learning can be great joy. I particularly want to thank my grandmother, Esther Morrison, for passing the value of hard work and education down to me through the generations.

Two friends deserve special thanks: Mark Stevenson, for half a lifetime of friendship, and Professor Elcesser, for her example of perseverance through the trials of the Big T.

Rick Rashid has been a true mentor. He has freely shared his excitement for Computer Science and all of life! He has always made it clear that he expected me to do great things, and to have fun doing them!

Eric Cooper deserves thanks for the times he's spent with me listening to ideas which I've had, both good and bad, and helping me to discern the difference.

Satya has tried to instill into me, and all of his students, the value of being rigorous. This is a far better dissertation, and has been completed as quickly as it has, because of his guidance.

Doug Tygar has willingly expressed his confidence in my work on several occasions. He has both helped refine my work, and my presentation of it.

Garret Swart has been both a valuable committee member and a friend. I have benefitted from his unique ability to critique a design and to think through its consequences on more than one occasion. His enthusiasm for ‘‘neat ideas’’ is infectious.

Brian Bershad deserves special thanks for serving as an enthusiastic and valuable advisor, in fact if not in name. I understand my own work far better than I otherwise would have due to his insightful perspectives.

I want to thank Lily Mummert for the time she spent with me patiently explaining the DFSTrace tools and for keeping the detailed notes of her work that proved so valuable to my understanding of the costs of implementing them.

I want to thank Dave Redell for pointing the way to a wealth of interesting related work. As a consequence of his interest, I spent many pleasant afternoons doing what he called ‘‘software archaeology’’.

This work could not have happened without the fertile environment of both ideas and software provided by the Mach Group. It’s been a stimulating group to have been a part of.

I want to thank the Facilities Group for providing the rich and extremely reliable computing environment which we all take for granted. Howard Wactlar and Mike Accetta deserve far more credit than I can possibly give them.

Finally, I want to recognize Sharon Burks for the love she shows the students in her care. Thanks, and God bless you!

Michael Blair Jones
Pittsburgh, Pennsylvania
September, 1992

Table of Contents

x

List of Figures

List of Tables