Exhibit 22

Interposition Agents: Transparently Interposing User Code at the System Interface

Michael B. Jones

Microsoft Research, Microsoft Corporation
One Microsoft Way, Building 9S/1047
Redmond, WA 98052
USA

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 paper 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.

SIGOPS '93/12/93/N.C., USA
© 1993 ACM 0-89791-632-8/93/0012...$1.50

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 paper, 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. Overview

This paper 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. 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. Examples

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.

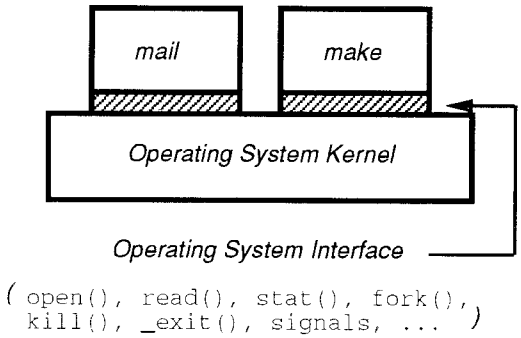

Figure 1-1: Kernel provides instances of system interface

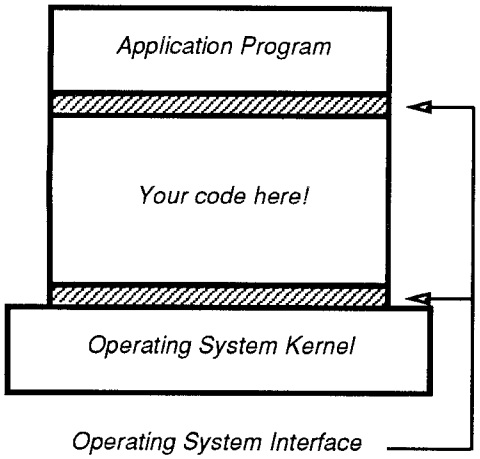

Figure 1-2: User code interposed at system interface

1.4. 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.

[1]The term "kernel" is used throughout this paper 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 [16].

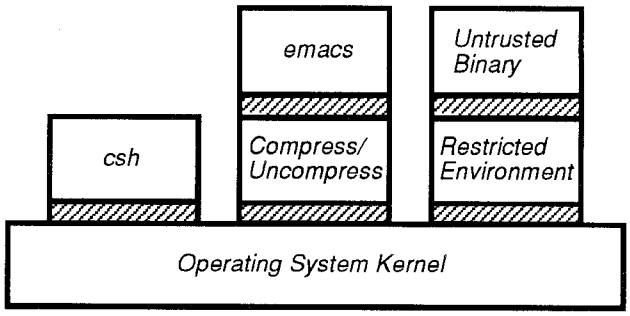

Figure 1-3: Kernel and agents provide instances of system interface

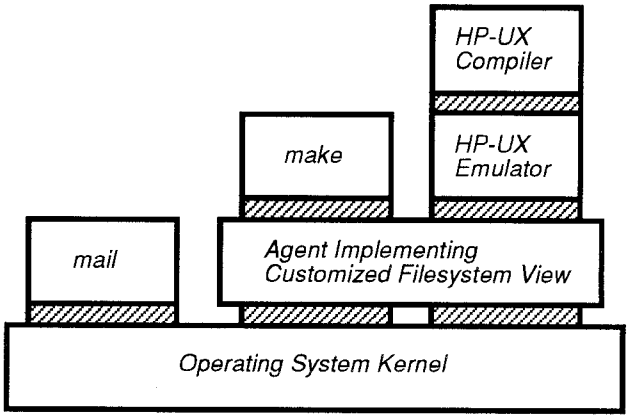

Figure 1-4: Agents can share state and provide multiple instances of system interface

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 [13], HP-UX [10], or UNIX System V [3] 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 are providing low-level mechanisms for intercepting system calls. Having these low-level mechanisms makes writing interposition agents possible. For instance, Mach [1, 16] provides the interception facilities used for this work, SunOS version 4 [44] provides new `ptrace()` operations used by the `trace` utility, and UNIX System V.4 [4] provides new `/proc` operations used by the `truss` utility. 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 [25] 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*. (In 4.3BSD, the primary system interface abstractions are pathnames, descriptors, processes, process groups, files, directories, symbolic links, pipes, sockets, signals, devices, users, groups, permissions, and time.) Furthermore, most calls manipulate only a few of these abstractions.

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. Research Overview

2.1. Design Goals

The four main goals of 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.

2.2. Design and Structure of the Toolkit

I have designed and built a toolkit on top of the Mach 2.5 system call interception mechanism [1, 5, 16] that can be used to interpose user code on the 4.3BSD [25] system call interface. The toolkit currently runs on the Intel

386/486 and the VAX. The toolkit is implemented in C++ with small amounts of C and assembly language as necessary. Multi-threaded hybrid 4.3BSD/Mach 2.5 programs are not currently supported.

As a consequence of using the Mach 2.5 system call interception mechanism, which redirects system calls to handler routines in the same address space, interposition agents reside in the same address spaces as their client processes. The lowest layers of the toolkit hides this Mach-specific choice, allowing agents to be constructed that could be located either in the same or different addresses spaces as their clients.

This toolkit is structured in an object-oriented manner, allowing agents to be written in terms of several different layers of objects by utilizing inheritance. Abstractions exposed at different toolkit layers currently include the filesystem name space, pathnames, directories, file descriptors and the associated descriptor name space, open objects referenced by descriptors, and signals, as well as the system calls themselves. (These abstractions are discussed further in Section 2.3.) Support for additional abstractions can be incrementally added as needed by writing new toolkit objects that represent the new abstractions and by using derived versions of the existing toolkit objects that reference the new abstractions through the new objects. Indeed, the current toolkit was constructed via exactly this kind of stepwise refinement, with useful toolkit objects being produced at each step. The structure of the toolkit permits agents to be written in terms of whatever system interface abstractions are appropriate to the tasks they perform. Just as derived objects are used to introduce new toolkit functionality, interposition agents change the behavior of particular system abstractions by using agent-specific derived versions of the toolkit objects representing those abstractions.

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 needs to manipulate higher-level objects such as pathnames and possibly file descriptors. The agent writer decides what layers of toolkit objects are appropriate to the particular task and includes only those toolkit objects. Default implementations of the included objects provide the normal behavior of the abstractions they represent. This allows derived agent-specific versions of toolkit objects to inherit this behavior, while adding new behavior in the implementations of the derived objects. 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.

2.3. Toolkit Layers

Figure 2-1 presents a diagram of the primary classes currently provided with the interposition toolkit. Indented classes are subclasses of the classes above. Arrows indicate the use of one class by another. Many of these classes are explained in more detail in this section.

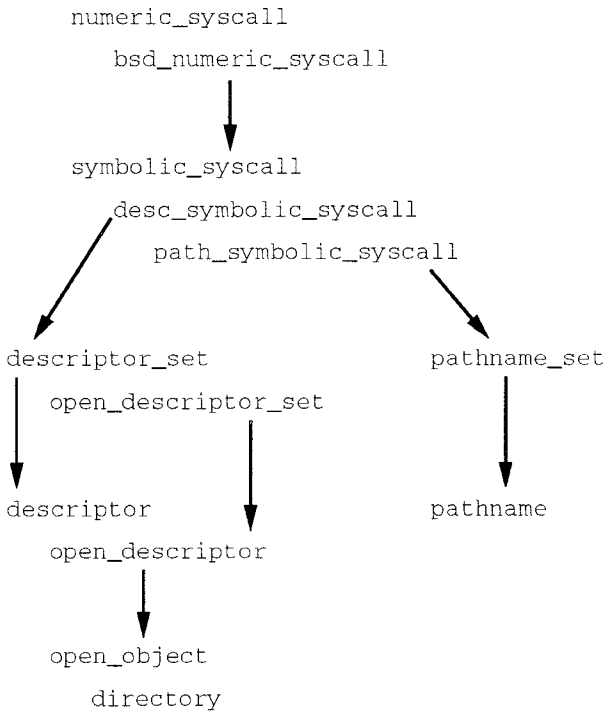

Figure 2-1: Primary interposition toolkit classes

The lowest layers of the toolkit 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 layers hide the 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. These layers also hide 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 layers are referred to as the *boilerplate* layers. These layers are not normally used directly by interposition agents.

The lowest (or zeroth) layer of the toolkit which is directly used by any interposition agents presents the system interface as a single entry point accepting vectors of untyped numeric arguments. It provides the ability to register for specific numeric system calls to be intercepted and for incoming signal handlers to be registered. This layer is referred to as the *numeric system call* layer.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.