Transparently Interposing User Code at the System Interface

Michael B. Jones

Carnegie Mellon University
Computer Science Department
5000 Forbes Avenue
Pittsburgh, PA 15217
USA
+1-412-268-3069
mbj@mach.cs.cmu.edu

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.

I am currently constructing an object-oriented toolkit which 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 which otherwise would not have been attempted.

I am also currently using this toolkit to construct several agents including: protected environments for running untrusted binaries, modified filesystem namespaces, logical devices implemented entirely in user space, transparent network data compression and/or encryption agents, and system call tracing tools. Examples of other interesting agents which could be built include: transactional software environments, and emulators for other operating system environments.

Having the flexibility to simultaneously support several variant sets of operating system services may be particularly useful in a workstation environment; a diversity of software which expects different operating system environments can be supported on a single operating system base. Even just the ability to provide multiple configurable views of the filesystem namespace provides a flexibility not found in traditional operating systems.

This research was supported by the Defense Advanced Research Projects Agency (DOD), and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. government.

1. Introduction

1.1. Background

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, allowing such operations as interposing programs which record or modify the communications taking place on this channel.

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 provides all instances of the operating system interface. Figure 1-2 depicts uses of the system interface with interposition. In this view both the kernel and interposition agents provide instances of the operating system interface. Figure 1-3 depicts more uses of the system interface with interposition. In this view, like the kernel, agents can share state and provide multiple instances of the operating system interface.

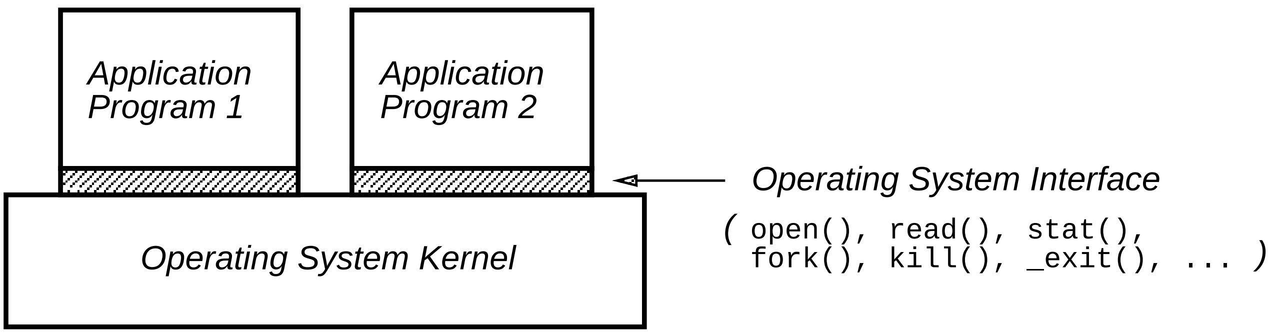

Figure 1-1: The kernel provides instances of the operating system interface.

1.2. Motivation for Interposition

Interposition can be used to provide programming facilities which would not otherwise 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 important 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 which allow monitoring of a program’s use of system services in a easily customizable manner.
- emulation of other operating system environments. Alternate system call implementations can be used to concurrently run binaries from variant operating systems (such as ULTRIX or System V binaries in a BSD environment) on the same platform.

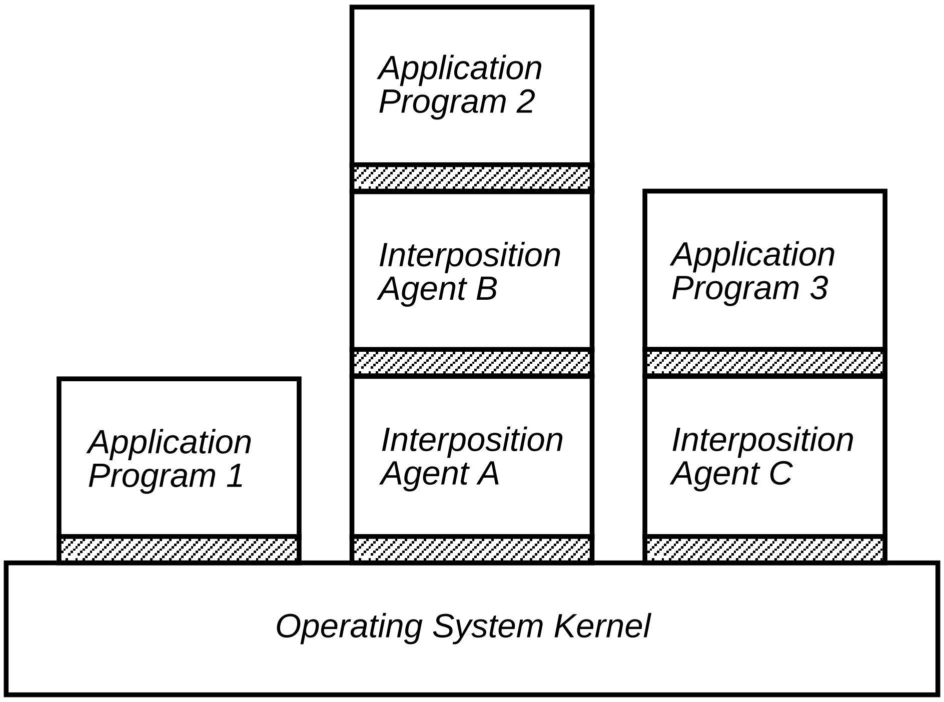

Figure 1-2: Both the kernel and interposition agents provide instances of the operating system interface.

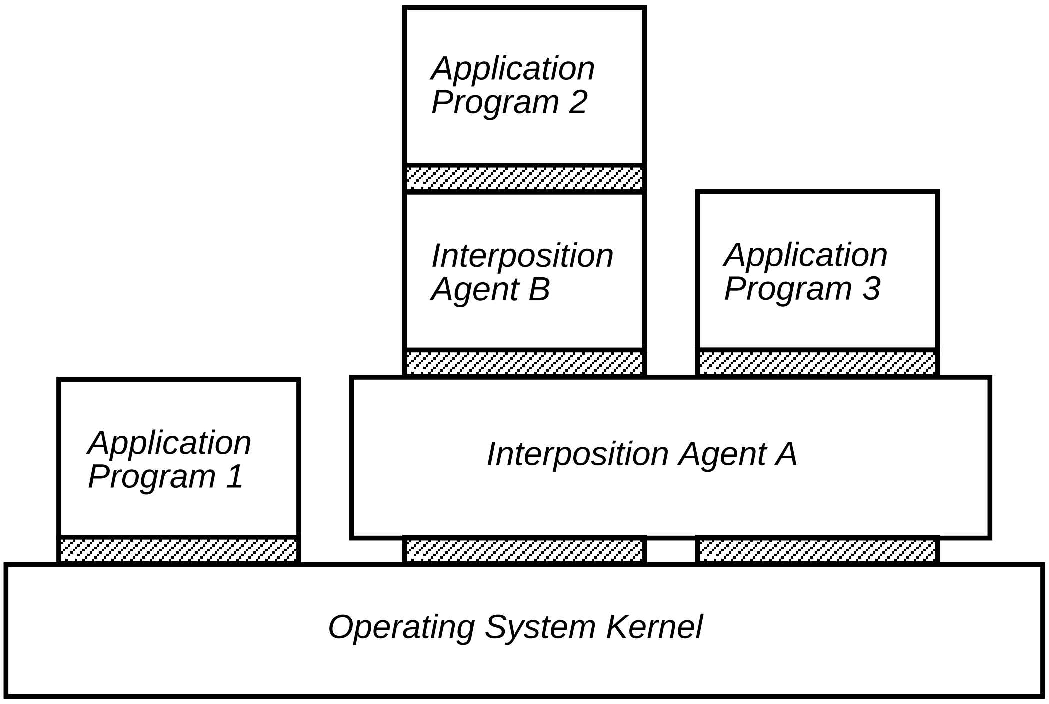

Figure 1-3: Like the kernel, agents can share state and provide multiple instances of the operating system interface.

- protected environments for running untrusted binaries. A wrapper environment can be constructed which allows untrusted, possibly malicious, binaries to be run in such a way that actions which they take may be monitored and emulated, possibly without actually being performed, and resources which they use can be limited in such a way that the untrusted binaries can't tell. 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 which provide an environment in which persistent state changes made by unmodified programs can be emulated and performed transactionally. For instance, a simple ''run transaction'' command could be constructed which 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 invocations could occur within another, transparently providing nested transactions.
- alternate or enhanced semantics. Environments can be constructed which 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 namespace 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`.

1.3. Problems with Existing Systems

Increasing numbers of operating systems, e.g., Mach, SunOS version 4, System V.4, are providing low-level mechanisms for intercepting system calls. 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. Thus, only the bare minimum interception facilities have been provided — the lowest common denominator which is generally useful. Consequently, each agent has typically been constructed completely from scratch. No leverage was gained from the work done on other agents. Not surprisingly, experience with past systems has shown that such facilities are rarely used.

2. Position Statement

An object-oriented toolkit can be constructed which 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 a 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 may then exploit the toolkit objects best suited to their individual needs. Consequently, substantial amounts of toolkit code are able to be reused when constructing different system call interception agents. Furthermore, having such a toolkit enables new system call implementations to be written which otherwise would not have been attempted.

The ability to simultaneously support multiple customized and special purpose implementations of the operating system services provides a flexibility in the operating system environment not generally available today. This flexibility may be particularly important in a workstation environment, where the user may, for instance, want the ability to run binary applications which expect somewhat different operating system environments (e.g., BSD 4.3, OSF/1, ULTRIX, System V, etc.) without having to have copies of each, and without having to reboot to switch worlds. Even just the ability to dynamically rearrange the appearance of the filesystem namespace will allow software to be used which has different configuration assumptions than a given workstation's default environment.

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 shared 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.

For wide classes of interesting emulation agents, a large portion 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 between multiple emulated processes is needed by many agents, particularly those which allow changes made by one emulated process to be seen by another.

Not every agent needs every kind of emulation support. Indeed, to the extent that an agent does not modify the behavior of a particular portion of the system interface, it is 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 which benefit them, while using inheritance to automatically access those functions which the agent has custom built.

Finally, an object-oriented 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. For one, 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.

4. Design and Structure of the Toolkit

I am currently designing and building a toolkit on top of the Mach system call emulation mechanism which can be used to interpose user code on the BSD 4.3 system call interface. 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 programmer to program at whatever levels of abstraction are appropriate for the agent being constructed.

The base layer of the toolkit handles intercepting the system calls themselves. Such operations as monitoring system call usage are done at this level.

The second layer of the toolkit is structured around the primary objects provided by the system call interface. In BSD 4.3, 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 used through primary objects. Such objects include files, directories, symbolic links, devices, pipes, and sockets. Operations which are specific to these secondary objects such as file encryption, directory transformations, etc. are done by these layers.

5. Status

The bulk of the toolkit, providing for numeric and symbolic system call interception, plus descriptor, open object, and pathname manipulation has been completed. Several useful agents have already been built. I am in the midst of writing my Ph.D. dissertation on this research.

Table of Contents

List of Figures