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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
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.

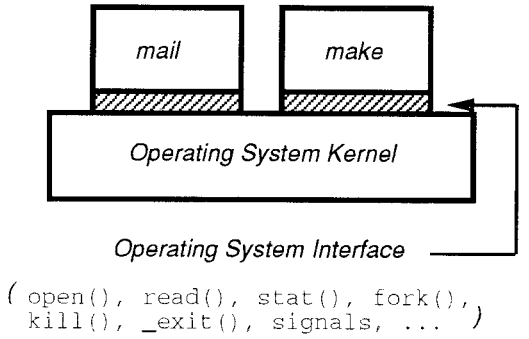

Figure 1-1: Kernel provides instances of system interface

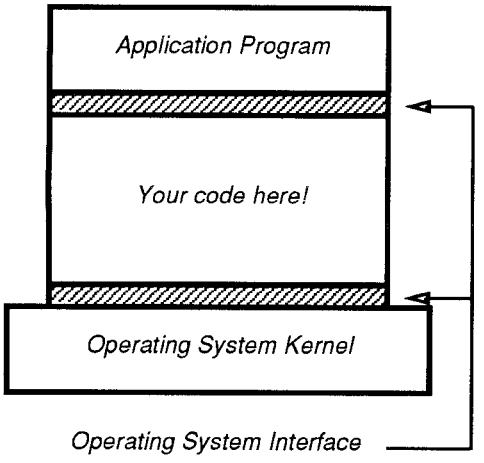

Figure 1-2: User code interposed at system interface

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].

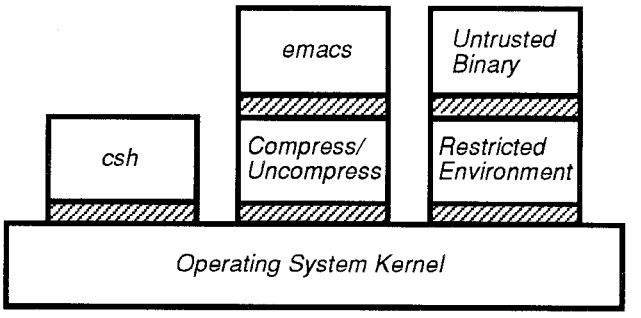

Figure 1-3: Kernel and agents provide instances of system interface

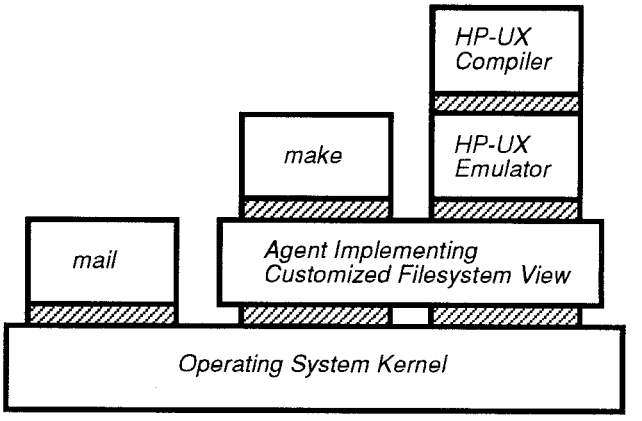

Figure 1-4: Agents can share state and provide multiple instances of system interface

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.

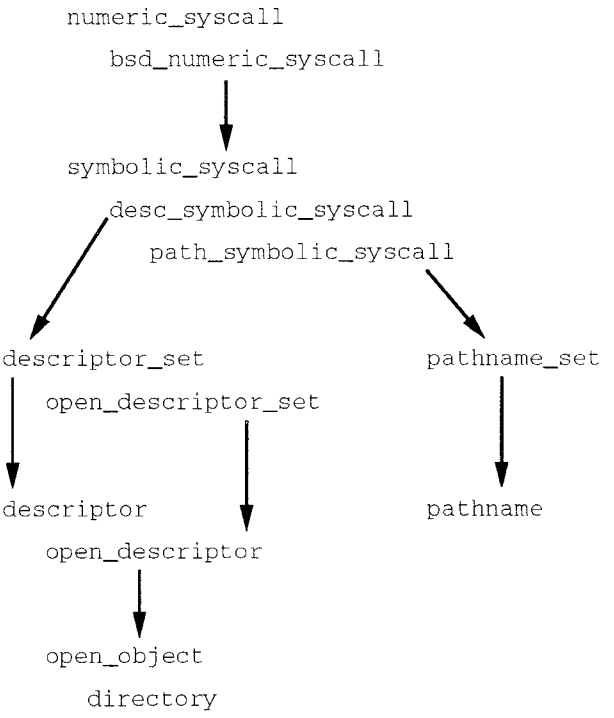

Figure 2-1: Primary interposition toolkit classes

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.

Example interfaces provided by the numeric system call layer 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);
};
```

For instance, using just the numeric system call layer, by using a derived version of the `numeric_syscall` class with an agent-specific `syscall()` method, an agent writer could trivially write an agent that printed the arguments of a chosen set of system calls as uninterpreted numeric values. As another example, one range of system call numbers could be remapped to calls on a different range at this level.

The first layer of the toolkit intended for direct use by most interposition agents presents the system interface as a set of system call methods on a system interface object. When this layer is used by an agent, application system calls are mapped into invocations on the system call methods of this object. (This mapping is itself done by a toolkit-supplied derived version of the `numeric_syscall` object.) This layer is referred to as the *symbolic system call* layer.

Example interfaces provided by the symbolic system call layer 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]);
  ... entries for all other 4.3BSD system calls ...

  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);
};
```

Agents can interpose on individual system calls by using a derived version of the `symbolic_syscall` object with agent-specific methods corresponding to the system calls to be intercepted. For instance, the `timex` agent, which is described in Section 3.3.1, changes the apparent time of day by using a derived `symbolic_syscall` object with a new `gettimeofday()` method. Likewise, the `trace` agent, described in Section 3.3.2, prints the arguments to each executed system call in a human-readable from individual system call methods in a derived `symbolic_syscall` object.

The second layer of the toolkit is structured around the primary abstractions provided by the system call interface. In 4.3BSD, these include pathnames, file descriptors, processes, and process groups. This layer presents the system interface as sets of methods on objects representing these abstractions. Toolkit objects currently provided at this level are the filesystem name space (`pathname_set`), resolved pathnames (`pathname`), the file descriptor name space (`descriptor_set`), active file descriptors (`descriptor`), and reference counted open objects (`open_object`). Such operations as filesystem name space transformations and filesystem usage monitoring are done at this level.

For example, agents can interpose on pathname operations by using derived versions of two classes: `pathname_set` and `pathname`. 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.

Example 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]);
  ... entries for other 4.3BSD system calls using pathnames ...
};
```

Example 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]);
  ... entries for other 4.3BSD system calls referencing objects via
    pathnames ...
};
```

The key to both of these interrelated classes is the `getpn()` operation, which looks up a pathname string and resolves it to a reference to a `pathname` object. The default implementation of all the `pathname_set` system call methods simply resolves their pathname strings to `pathname` objects using `getpn()` and then invokes the corresponding `pathname` method on the resulting object. The `pathname` method is responsible for actually performing the requested operation on the object referenced by the pathname.

With the `getpn()` operation to encapsulate pathname lookup, it is possible for agents to supply derived versions of the `pathname_set` object with a new `getpn()` implementation that modifies the treatment of all pathnames. For instance, this can be used to logically rearrange the pathname space, as was done by the `union` agent (described in Section 3.3.3). Likewise, it provides a central point for name reference data collection, as was done by the `dfs_trace` agent (described in Section 3.5.3).

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. These layers present the system interface as sets of methods on objects, with specialized operations for particular classes of objects. The only toolkit object currently provided at this level is the open directory `directory` object. Operations that are specific to these secondary objects such as directory content transformations are done at this level.

For example, agents can interpose on directory operations by using derived versions of the `directory` class. The directory class is itself a derived version of the `open_object` class (one of the second layer classes for file descriptor operations), since directory operations are a special case of operations that can be performed on file descriptors.

Example 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]);
};
```

Just as the `getpn()` method encapsulated pathname resolution, the `next_direntry()` method encapsulates the iteration of individual directory entries implicit in reading the contents of a directory. This allows the `union` agent (described in Section 3.3.3) to make it appear that the full contents of a set of directories is actually present in a single directory by providing a new `next_direntry()` function that iterates over the contents of each member directory. (And yes, that iteration itself is accomplished via the underlying `next_direntry` implementations.)

2.4. Using the Toolkit to Build Agents

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 [30] originally implemented for use by the Coda [38, 23] filesystem project: this provides a basis for comparing a best available equivalent implementation to a facility provided by an agent.

3. Results

3.1. Goal: Unmodified System

3.1.1. 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. Indeed, the presence of agents should be transparent to applications.[2]

3.1.2. 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 that are provided by the kernel in order to implement all agent-specific system call behavior. Also, a general agent loader program is used to invoke arbitrary agents, which are compiled separately from the agent loader.

The Mach 2.5 kernel used for this work contains a primitive, `task_set_emulation()`, that allows 4.3BSD system calls to be redirected for execution in user space. Another primitive, `htg_unix_syscall()`, permits calls to be made on the underlying 4.3BSD system call implementation even though those calls are being redirected.

3.2. Goal: 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

[2]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.

that used a restricted set of features that agents could handle, and those that used features that 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.

3.3. Goal: Appropriate Code Size

Table 3-1 lists the source code sizes of three different agents, broken down into statements of toolkit code used, and statements of agent specific code.[3] 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 3-1: Sizes of agents, measured in semicolons

3.3.1. Size of the Timex Agent

The `timex` agent changes the apparent time of day. It is built upon the symbolic system call and lower levels of the toolkit (see Section 2.3). 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 and an initialization routine to accept the desired effective time of day from the command line. This code contains only 35 statements.

The core of the `timex` agent is as follows:

```
class timex_symbolic_syscall : public
    symbolic_syscall {
public:
  virtual void init(char *agentargv[]);
  virtual int sys_gettimeofday(struct timeval
    *tp, struct timezone *tzp, int rv[2]);
private:
  int offset;    // Difference between real
    and funky time
};

int timex_symbolic_syscall::sys_gettimeofday(
    struct timeval *tp, struct timezone *tzp,
    int rv[2])
{
  int ret;
  ret =
    symbolic_syscall::sys_gettimeofday(tp,
    tzp, rv);
  if (ret >= 0 && tp) {
    tp->tv_sec += offset;
  }
  return ret;
}
```

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.

3.3.2. Size of the Trace Agent

The `trace` agent traces the execution of client processes, printing each system call made and signal received. 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 `trace` agent is proportional to the size of the entire system interface. Derived versions of each of the 114 4.3BSD system calls plus the signal handler are needed to print each call name and arguments, since each call has a different name and typically takes different parameters. Even so, the new code contains less than 12 statements per system call, 10 of which typically are of the form:

```
virtual int sys_read(int fd, void *buf, int
  cnt, int rv[2]);
```
(line from TRACE_SYMBOLIC_CLASS *class declaration)*

```
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;
}
```

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.

3.3.3. 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. It is built using toolkit objects for pathnames, directories, and descriptors, as well as the

[3]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.

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 toolkit pathname object that maps operations using names of union directories to operations on the underlying objects, a derived form of the toolkit 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.

3.3.4. 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), 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` agent and `dfs_trace` agent[4] are also able to use the descriptor, open object, and pathname levels of the toolkit, reusing a total 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.

[4]The `dfs_trace` agent implements file reference tracing tools that are compatible with existing tools [30] originally implemented for use by the Coda [38, 23] filesystem project. This agent is discussed further in Section 3.5.3.

3.4. Goal: Performance

3.4.1. 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 3.4.2.

3.4.1.1. Performance of Formatting a Document

Table 3-2 presents the elapsed time that it takes to format a preliminary draft of my dissertation with Scribe [36] on a VAX 6250 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 my dissertation		
Agent Name	*Seconds*	*% Slowdown*
None	131.5	—
`timex`	132.0	0.5%
`trace`	135.0	2.5%
`union`	136.5	3.5%

Table 3-2: Time to format my dissertation

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.[5]

3.4.1.2. Performance of Compiling C Programs

Table 3-3 presents the elapsed time that it takes to compile eight small C programs using Make [15] and the GNU C compiler [40] on a 25MHz Intel 486. In each case, the time presented is the average of nine successive runs done after an initial run from which the time was discarded.

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

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 3-3: 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 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 and the additional two `write()` operations performed per application system call for writing the trace log, the log output time constitutes a significant portion of the slowdown.

An analysis of low-level performance characteristics is presented in Sections 3.5.1.1 and 3.5.1.2.

3.4.2. 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 my dissertation 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.

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 is otherwise available. The performance "lost" by using an interposition agent can bring other types of gains.[6]

3.5. Other Results

3.5.1. Low-level Performance Measurements

3.5.1.1. 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.

Table 3-4 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 arg, result	1.22
C++ virtual procedure call with 1 arg, result	1.94
Intercept and return from system call	30
`htg_unix_syscall()` overhead	37

Table 3-4: Performance measurements of individual low-level operations

Table 3-5 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

[6]For a discussion on the tradeoffs of using interposition agents, see Section 5.3.

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. Measured pathnames are in a UFS [27] filesystem and contain 6 pathname components.

Performance of System Calls			
Operation	*µsec without agent*	*µsec with agent*	*µsec toolkit overhead*
`getpid()`	25	170	145
`gettimeofday()`	47	214	167
`fstat()`	54	220	166
`read()` 1K of data	370	579	209
`stat()`	892	1101	209
`fork(),wait(),_exit()`	10350	22350	12000
`execve()`	9720	20000	10280

Table 3-5: Performance measurements of individual system calls

3.5.1.2. Micro Performance Results

Two times from Table 3-4 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()`[7] 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 3-5) 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. 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 2.3) range from about 140 to 210µsec., as per Table 3-5. Overheads for `fork()` and `execve()` are significantly greater, adding approximately 10 milliseconds to both, roughly doubling their costs.

The `execve()` call is more expensive than most because it must be completely reimplemented by the toolkit from lower-level primitives, unlike most calls where the version provided by the underlying implementation can be used. The underlying implementation's `execve()` call can not be used because it clears its caller's address space. While the application must be reloaded, the agent needs to be preserved. Thus, the extra expense of `execve()` is due to having to individually perform such operations as clearing the caller's address space, closing a subset of the descriptors, resetting signal handlers, reading the program file, loading the executable image into the address space, loading the arguments onto the stack, setting the registers, and transferring control into the loaded image, all of which are normally done by a single `execve()` call. Likewise, `fork()` and `_exit()` are more expensive due to the number of additional bookkeeping operations required.

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 3.4.2.

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.

3.5.2. Portability

The interposition toolkit should port to similar systems such as SunOS and UNIX System V. Despite toolkit dependencies on such Mach facilities as the particular system call interception mechanism used, all such dependencies were carefully encapsulated within the lowest (boilerplate) layers of the toolkit. None of the toolkit layers above the boilerplate directly depends on Mach-specific services. Higher toolkit layers, while being intentionally 4.3BSD specific, contain no such dependencies. This 4.3BSD dependency imposes at most minor portability concerns to other UNIX-derived systems, given their common lineage and resulting substantial similarity. Thus, it should be possible to port the toolkit by replacing the Mach-dependent portions of the boilerplate layers with equivalent services provided by the target environment.

[7] The `htg_unix_syscall()` facility permits calls to be made on the underlying 4.3BSD system call implementation even though those calls are being intercepted.

Likewise, interposition agents written for the toolkit should also readily port. Even if there are differences between the system interfaces, the toolkit port should be able to shield the agents from these differences, unless, of course, the agents are directly using the facilities which differ.

One caveat, however, is probably in order. While a port of the toolkit could shield interposition agents from low-level system interface differences, it certainly can not shield them from system performance differences. If the toolkit is ported to systems that provide significantly slower system call interception mechanisms (as, for instance, mechanisms based on UNIX signals are likely to be), then some agents which previously exhibited acceptable slowdown might exhibit unacceptable slowdown when ported.

3.5.3. Comparison to a Best Available Implementation

As an element of this research, an interposition agent (`dfs_trace`) was constructed that implements file reference tracing tools that are compatible with the existing kernel-based DFSTrace [30] tracing tools originally implemented for use by the Coda [38, 23] filesystem project. This was done to provide a realistic basis for comparing a best available implementation of a task that was implemented without benefit of the toolkit with an equivalent interposition agent constructed using the toolkit.

While this comparison is not presented in detail here[8] several of the resulting conclusions are worth noting. 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.
- 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. For instance, the kernel-based DFSTrace tools in the default mode caused a 3.0% slowdown while executing the AFS filesystem performance benchmarks [19]. The agent-based implementation caused a 64% slowdown under the same workload.

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. For instance, the original DFSTrace kernel and user data collection code contains 1627 statements, compared to 1584 statements for the agent-based implementation.
- 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.
- 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. For instance, the original DFSTrace implementation required the modification of 26 kernel files in order to insert data collection code under conditional compilation switches; the agent-based implementation required no modifications to existing code since inheritance was used to add functionality. Also, the kernel-based implementation uses four machine-dependent files per machine type; the agent-based implementation is machine independent.

In summary, the interposition agent was more logically structured, was probably simpler to write and modify, and required no system modifications to implement or run. The kernel-based tracing tools were more efficient.

4. Related Work

This section presents a brief survey 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 section attempts to provide a representative, if not comprehensive, overview of the related work.

[8] See [21] for a detailed presentation of this comparison.

4.1. Overview of Past Agents

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 [33] emulating OS/360 or TSO, TENEX [7, 48] emulating TOPS-10, and RSEXEC [47] emulating an Arpanet-wide TENEX.
2. Debuggers and debugging facilities, such as those for CAL TSS [43], DTSS [24], ITS [14], and SunOS [44].
3. System call trace facilities, such as the "`SET TRAP JSYS`" facility under the TENEX/Tops-20 [12] EXEC, the `truss` program under UNIX System V.4 [4], and the `trace` program under SunOS version 4.
4. Adding new facilities to the operating system interface, as was done in OS6 [41, 42], CAL TSS, and the MS-DOS [29]/Windows [28] environment.
5. Providing enhanced implementations of the existing operating system interface (often enhanced filesystem implementations), as was done in CAL TSS, TENEX, the Newcastle Connection [8], NFS [50], ITOSS [34], Watchdogs [6], Taos [26], and particularly in the Macintosh [2] and MS-DOS.

Stackable interfaces are known to be useful in other domains as well. For instance, communication protocols are composed from stackable layers in the *x*-Kernel [20]. Streams are regularly stacked under UNIX System V [4]. Stackable layers are used for constructing flexible filesystems in Ficus [18, 17] and with an enhanced Vnode interface [11].

Finally, interposition is certainly commonly used on communication channels in message-based systems. For instance, interposition was regularly used in the Accent [35] and V [9] systems.

4.2. Analysis of Past Agents

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 [31, 37, 32, 45, 28, 39, 46, 49] to name just a few) appears to validate the claim that interposition can be a useful and effective system building paradigm.

5. Conclusions

5.1. 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 interprocess 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.

A more detailed presentation of these results can be found in [21].

5.2. 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, as explored in [22]. 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.

5.3. 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.

5.4. 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.

Acknowledgments

I'd like to extend special thanks to Rick Rashid, Eric Cooper, M. Satyanarayanan (Satya), Doug Tygar, Garret Swart, Brian Bershad, and Patricia Jones. Each of you have have offered helpful suggestions and criticisms which have helped shape and refine this research. I'd also like to thank Mark Weiser for his valuable comments on ways to improve this paper.

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.

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.

References

1. 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.

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

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

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

5. 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.

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

7. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L. and Tomlinson, R.S. "TENEX, a paged time sharing system for the PDP-10". *Comm. ACM 15*, 3 (March 1972), 135-143.

8. D.R. Brownbridge, L.F. Marshall, B. Randell. "The Newcastle Connection, or UNIXes of the World Unite!". *Software — Practice and Experience 12* (1982), 1147-1162.

9. David R. Cheriton. "The V distributed system". *Communications of the ACM 31*, 3 (March 1988), 314-333.

10. 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 (December 1986), 4-22.

11. David S. H. Rosenthal. Evolving the Vnode Interface. In *USENIX Conference Proceedings*, June, 1990, pp. 107-118.

12. *DECSYSTEM-20 Monitor Calls Reference Manual.* Digital Equipment Corporation, 1978.

13. *ULTRIX Reference Pages, Section 2 (System Calls).* Digital Equipment Corporation, 1989.

14. 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).

15. S. I. Feldman. "Make — a program for maintaining computer programs". *Software — Practice and Experience 9*, 4 (1979), 255-265.

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

17. Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Conference Proceedings*, June, 1990, pp. 63-71.

18. John S. Heidemann. *Stackable Layers: an Architecture for File System Development.* Master Th., University of California, Los Angeles, July 1991. Available as UCLA technical report CSD-910056.

19. 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).

20. N. C. Hutchinson and L. L. Peterson. Design of the *x*-Kernel. In *Proceedings of the SIGCOMM '88 Symposium*, Stanford, CA, Aug., 1988, pp. 65-75.

21. Michael B. Jones. *Transparently Interposing User Code at the System Interface.* Ph.D. Th., Carnegie Mellon University, September 1992. Available as Technical Report CMU-CS-92-170.

22. 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.

23. Kistler, J.J. and Satyanarayanan, M. "Disconnected Operation in the Coda File System". *ACM Transactions on Computer Systems 10*, 1 (February 1992).

24. Philip Koch and David Gelhar. *DTSS System Programmer's Reference Manual.* Dartmouth College, Hanover, NH, 1986. Kiewit Computation Center TM059.

25. 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.

26. 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.

27. 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).

28. *Microsoft Windows User's Guide.* Microsoft Corporation, 1987.

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

30. Lily B. Mummert and M. Satyanarayanan. *Efficient and Portable File Reference Tracing in a Distributed Workstation Environment.* To be published as a Carnegie Mellon University School of Computer Science technical report.

31. *The Norton Utilities for the Macintosh.* Peter Norton Computing, Incorporated, 1990.

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

33. R. P. Parmelee, T. I. Peterson, C. C. Tillman and D. J. Hatfield. "Virtual Storage and Virtual Machine Concepts". *IBM Systems Journal 11*, 2 (1972), 99-130.

34. Michael O. Rabin and J.D. Tygar. *An Integrated Toolkit for Operating System Security.* Tech. Rept. TR-05-87, Harvard University Center for Research in Computing Technology, Cambridge, MA, May, 1987. Revised August 1988.

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

36. Brian K. Reid and Janet H. Walker. *SCRIBE Introductory User's Manual.* Third edition, UNILOGIC, Ltd., 1980.

37. *DiskDoubler User's Manual.* Salient Software, Inc., 1991.

38. 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).

39. *Stacker 2.1 User Guide.* Stac Electronics, Inc., 1992.

40. Richard M. Stallman. *Using and Porting GNU CC, for version 1.37.* Free Software Foundatation, Inc., 1990.

41. 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 (May 1972), 117-124.

42. 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 (August 1972), 195-203.

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

44. *SunOS Reference Manual.* Sun Microsystems, Inc., 1988. Part No. 800-1751-10.

45. *Symantec AntiVirus for Macintosh.* Symantec Corporation, 1991.

46. *The Norton AntiVirus.* Symantec Corporation, 1991.

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

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

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

50. 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.