Efficient, Protected Extension of Commodity Operating Systems

by

Douglas Phillip Ghormley

B.S. (Carnegie Mellon University) 1991

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Thomas Anderson, Chair
Professor David Culler
Professor Andreas Kathol

Spring 1998

The dissertation of Douglas Phillip Ghormley is approved:

Chair Date

Date

Date

University of California at Berkeley

Spring 1998

Abstract

Efficient, Protected Extension of Commodity Operating Systems

by

Douglas Phillip Ghormley
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Thomas Anderson, Chair

A network of workstations, harnessed as a single, coherent cluster, offers significant advantages in terms of scalability, cost/performance, incremental growth, and fault tolerance. Effectively extending UNIX semantics from a single node to a cluster requires a level of virtualization for existing UNIX abstractions. Prior efforts to add a level of virtualization in the kernel suffer from many disadvantages, including high cost, implementation complexity, and lack of portability; efforts to virtualize at the user level suffer from restrictions on possible functionality and either lack protection or limited performance. This dissertation examines these problems through the real-world example of GLUnix, a user-level layer providing cluster functionality to a network of workstations.

To overcome the limitations encountered in both kernel-level and user-level approaches to virtualizing UNIX functionality, this dissertation explores how best to support low-cost extension of existing commodity UNIX operating systems. We demonstrate that efficient, protected extension of operating system functionality is possible with little or no change to existing commodity UNIX operating systems. We accomplish this by devising a general-purpose operating system extension method which leverages the concept of interposition to enable trusted, third-party extensions to modify operating system behavior. SLIC, our prototype of this extension technology, enables extensibility of existing commodity UNIX operating system interfaces with little or no modification to the operating system. SLIC enables novel system layers, such as GLUnix, to be largely constructed at the user level while at the same time overcoming the limitations associated with completely user-level operating system services. SLIC downloads extensions into the kernel that provide user-level access to the kernel interfaces necessary to extend UNIX semantics to the cluster. Additionally, SLIC has been used to implement a security patch, an encryption filesystem, and a restricted execution environment for untrusted applications.

Introducing general-purpose extensibility to operating systems raises the issue of composability. When multiple extensions, implemented by independent third-party developers, are composed on a single interface, problems can arise with the interaction of those extensions. This dissertation describes this problem, characterizes the sources of many of the conflicts, and presents a number of heuristics which can considerably alleviate the problem of extension conflicts.

Professor Thomas Anderson
Dissertation Committee Chair

To my wife, Sara,
who was never without a word of encouragement.
A greater gift has no man found in his wife
than I have in you.

And to my brother, Jay Mize,
who was a constant reminder of the hand and hope that guide me.
Your humility, teachable spirit, and faithfulness
have been an inspiration to me;
I will not forget you.

Contents

List of Figures

List of Tables

Acknowledgements

There are a number of people whose technical assistance was invaluable to this work. In particular I wish to thank Amin Vahdat for his help in designing, implementing, and testing GLUnix. I also wish to thank David Petrou, Josh Coates, Remzi Arpaci-Dusseau, Steve Rodrigues, and Keith Vetter for their assistance in developing, maintaining, evaluating, and tuning GLUnix. I wish also to thank David Petrou for his ideas and assistance in developing and testing SLIC as well as Steve Lumetta for his ideas on extension conflicts.

There are a number of people who, over the years, have made this work possible. A plethora of thanks also go to Steve Rodrigues, Amin Vahdat, David Petrou, and Thomas Anderson for their inestimable assistance in the GLUnix and SLIC work over the years. Steve is a veritable LaTeX genius and has provided critical assistance on occasions too numerous to mention. Special thanks go to Tom Anderson for his guidance and feedback in this work, and to David Culler for his help in pointing the way from research to dissertation.

My hearty appreciation also goes to Tom Anderson, David Culler, and Dave Patterson for their intellectual and pragmatic leadership of the NOW group—we'll make managers out of you yet.

I gratefully acknowledge the financial support of the Advanced Research Projects Agency (N00600-93-C-2481), the National Science Foundation (CDA 9401156), Sun Microsystems, California MICRO, Hewlett Packard, Intel, Microsoft, and Mitsubishi.

Chapter 1

Introduction

1.1 Motivation

Currently available techniques for modifying operating system functionality are inadequate to meet the crucial need for operating system developers to be able to quickly respond to new user requirements, recent operating system research, and security breaches. Current approaches generally require time-consuming modifications to large, unwieldy code bases or are restricted in terms of functionality and performance. A number of research projects have demonstrated novel operating system architectures to directly support third-party extensibility, but these architectures do not solve the problem of extensibility for existing commodity operating systems.

In practice, even when initial operating system development has been completed and the system has been released, on-going revisions are required to meet the changing demands of users, to incorporate improvements from operating systems research, and to fix security holes and other bugs; as an example, a list of system extensions on a randomly-selected graduate student's Macintosh shows a total of 145 system events being intercepted by 22 extensions ranging from system upgrades to commercial applications adding new functionality. The increasing popularity of multimedia, for example, requires the deployment of novel scheduling policies [NL97] in order to efficiently execute a mix of interactive and real-time applications. Realizing the benefits of new operating system features developed by the research community also require on-going modification of existing systems. For example, realizing the full potential of a network of workstations [ACPtNT95] (NOW) requires significant operating system modifications in the areas of single system image, fault tolerance, and security. Finally, the ease by which the internet can be exploited by unfriendly attackers intensifies the need for new operating system development structures and practices which dramatically reduce the multitude of security bugs which typically occur in today's complex operating systems [Tea96b].

Not only do these operating system modifications need to made on an on-going basis, but in many cases they must be made quickly in order to be effective. Delays in releasing system software translate directly into lost revenues for hardware and software vendors and, in an environment where hardware performance increases as rapidly as it does today, delays can also translate into lost performance. Delays in releasing new versions of an operating system can result in loss of revenue for hardware vendors (if the hardware requires special support provided by the delayed operating system) as well as software vendors. While new policies or mechanisms demonstrated by the research community may increase performance, the time involved in implementing and debugging the new features may introduce delays in development which negate the intended benefit of the feature. For example, delays encountered in modifying an operating system to support remote execution and migration features for a NOW can result in a cluster comprised of out-dated components, a situation which significantly reduces the cost-performance advantages of the NOW architecture; similar software delays contributed to the demise of the massively-parallel processor (MPP) industry. Furthermore, any additional complexity which is introduced into the system by the new feature will have a residual, negative effect, potentially increasing the time required for future modifications of the operating system. Finally, it is crucial that delays be minimized in patching security flaws which are routinely discovered and reported by organizations such as Carnegie Mellon's Computer Emergency Response Team (CERT) and the Department of Energy's Computer Incident Advisory Capability (CIAC). Preventing wide exploitation of these flaws requires immediate modification to the operating system. While many large, complex applications suffer from similar issues, the mission-critical nature of operating systems causes delays and bugs in the operating system to adversely affect all applications on the system.

Unfortunately, making changes to existing commodity operating systems is an increasingly difficult and time-consuming process, making delays in development unavoidable. Operating systems are large, complex systems developed over many years by large teams of programmers and contain hundreds of thousands, sometimes millions, of lines of code. In order to fully capitalize on the benefits of symmetric multiprocessors (SMPs), many commodity operating systems are now multi-threaded, increasing their complexity significantly and introducing subtle dependencies which make them difficult to modify. Development of new kernel functionality is also hindered by the relative lack of kernel debugging and profiling environments—operating system development environments are generally well behind the state-of-the-art as seen in user tools. Furthermore, installing multiple third-party operating system modifications today requires end users to obtain kernel source, apply patches implementing the new functionality, manually resolve any source code conflicts arising from independent patches, and recompile the operating system—a process that end users should never be required to do.

As one example, studies have clearly demonstrated the potential benefit of the NOW architecture [ACPtNT95, ADV+95, ADADC+97]. As a consequence of the difficulties involved in modifying current commodity operating systems, however, the full potential of NOWs remains unfulfilled. Extending operating system semantics (such as UNIX or Windows NT) from a single node to a cluster requires virtualizing a number of system abstractions and operations, including the process name space, group signals, standard input and output, and device access. The functionality necessary to accomplish this virtualization is not available at the user level. Consequently, fully functional cluster solutions are not widely available for commodity operating systems.

Additional consequences of the difficulty in modifying commodity operating systems are that features are added to operating systems very slowly and security bugs can be non-trivial to fix. There is a large catalog of new operating system functionality that has not been widely deployed, in part because of the difficulty of modifying existing systems: fast communication primitives [BALL90, vECGS92], upcalls [Cla85], distributed shared memory [LH89], user-level pagers [YTR+87], and novel schedulers [WW94, DAC96, NL97], in addition to the cluster-related features of load sharing [ZWZD92] and process migration [TLC85, DO91]. Functionality which benefits only a narrow class of applications may never be added to commodity operating systems because the cost of implementation is not justified. Even when new features are supported, pressures to minimize time-to-mark often result in systems being shipped with known bugs. It is not unusual for major releases of commodity operating systems to be riddled with inadvertent flaws introduced during development, typically requiring additional "bug fix" releases which may in turn introduce their own flaws. It is common wisdom that major new releases of an operating system (e.g., $x.0$) should be avoided until the predominant bugs are identified and fixed and the next version (e.g., $x.1$) is released. As a consequence of the difficulty in modifying and testing operating systems, it can take weeks to repair and distribute patches for even widely-publicized security flaws [Tea97] arising from coding errors; rectifying security problems due to design flaws can take much longer.

1.2 Goal

The goal of this work is to significantly simplify the process of evolving existing commodity UNIX operating systems by enabling rapid, low-cost extension to system software. While we will focus specifically on UNIX operating systems, the concepts employed and the results derived are generally applicable to other operating systems as well, such as Windows NT. The ideal system to achieve this goal would possess a number of characteristics: it would require few or no modifications to existing operating systems or applications; it would introduce little overhead; extensions would be protected from malicious applications and enforced upon uncooperative applications; multiple extensions from independent, third-party sources could be active simultaneously; and extension developers would be able to make use of state-of-the-art user-level development and debugging tools, including source-level symbolic debuggers (e.g., `gdb`, `purify`), profilers (e.g., `gprof`, `quantify`), integrated development and debugging environments, and multi-threaded verification tools (e.g., Eraser [SBN+97]). Although current operating systems generally support third party applications and device drivers, third party operating system features are not generally supported; enabling them would improve system responsiveness to changing user demands and operating system advances as well as increase system reliability and functionality through competition.

In particular, accomplishing this goal would enable the full benefit of cluster to finally be realized in commodity systems. By enabling third-party extension of existing commodity UNIX operating systems, cluster developers can implement extensions which provide user-level cluster layers with the functionality necessary to properly extend operating system semantics and functionality to a cluster. This would enable commodity UNIX operating systems to be augmented with cluster functionality. By implementing the majority of the cluster software at the user-level and using operating system extensions only to provide the necessary virtualization hooks, cluster software layers can retain the majority of the advantages of user-level construction, including the portability to other systems and the ability to trivially leverage advances in the underlying operating system and hardware.

Providing low-cost, third-party extensibility of the operating system would also enable independent software vendors (ISVs) to develop and deploy innovative operating system features. In particular, new operating system features developed by research projects could be transferred directly to end users without the need to convince or wait for operating system vendors to adopt the modifications. In contrast to operating systems where relatively few successful ISVs exist (Transarc being a notable exception), robust ISV markets exist in other areas, such as databases [Dyc97], Web software [Net94, GM95], and desktop publishing [Bro94, BV98, McC97, Fre97], in each case because extensibility has been designed into the system. Furthermore, many CERT and CIAC security advisories normally require the system administrator to wait for a patch from the operating system vendor; instead, the advisory could directly include a small extension to correct the flaw, reducing the window of vulnerability dramatically.

1.3 Traditional Alternatives

Unfortunately, currently available alternatives for implementing cluster services and other operating system features fall considerably short of this goal. The traditional methods of implementing new operating system features primarily fall into three categories: rewriting the operating system from scratch, modifying an existing operating system, or implementing at user level. In the paragraphs to follow, we will evaluate each of these traditional alternatives.

1.3.1 Rewriting the Operating System

Perhaps the most time-honored alternative for adding functionality to an operating system is to throw out the existing operating system and start over from scratch. This approach affords a number of advantages: there are no restrictions placed on possible functionality; new features are protected from and enforced on uncooperative applications by virtue of being located in the kernel; and new features can be tuned to minimize overhead. However, the costs involved with this alternative, while perhaps reasonable when operating systems were simpler and less complex, are prohibitive today for all but the most compelling and pervasive changes. Microsoft, for example, spent over $300 million and ten years in the development of Windows NT before it became a successful product [Zac94]. Rewriting an operating system from scratch incurs a substantial cost in reimplementing basic system features such as scheduling, virtual memory, networking code, and device drivers—features which are often not necessary to reimplement. Although some improvements are being made, kernel development and debugging environments are considerably behind the state of the art as seen in user-level tools, making rapid development difficult. Furthermore, operating system features developed in the process of reimplementing the operating system from scratch are not easily transferable to other operating systems. For example, the Sprite [OCD+88] and V [Che88] operating systems both implemented kernel support for a number of cluster features including process migration [DO91, TLC85], yet because these implementations were dependent on the internal aspects of the operating systems, little of the original implementations could be leveraged by others. Finally, this approach requires continual porting efforts to keep abreast of new hardware platforms. Because of the cost and time required to reimplement an operating system from scratch, in the past decade, only a handful of operating systems built this way have been successful, including Windows NT, Solaris, and Linux, yet none of these operating systems contain the full breadth of useful functionality demonstrated by research. Because of the high costs involved, this alternative falls considerably short of the goals of a low-cost, rapid-development architecture outlined above.

1.3.2 Modify the Operating System

Another option is to modify an existing operating system to include new features. While this approach can reduce development time significantly, it retains many of the disadvantages of reimplementing from scratch. The size and complexity of modern operating systems make development delays and bugs inevitable in practice. The approach of modifying the operating system prevents independent developers from easily merging their changes into a single, unified system, leveraging each other's work. Because of the complex, internal interactions of modern operating systems, substantial modifications can take significant time and effort to merge. Implementations are also difficult to transfer to other operating systems because of significant internal differences. This is true even for different versions of UNIX which started as from a single source base but diverged over the years due to independent customization [Web93]. This approach also suffers from the relative lack of advanced kernel development and debugging environments. However, modifications to an operating system kernel can often leverage the work done to port the native operating system to new hardware architectures, enabling the new feature to run with no additional modifications. Still, the costs involved with implementing new operating system functionality in this manner prevent it from being a low-cost, rapid-development solution, as demonstrated by the fact that major new releases of commodity operating system are usually years apart; even patches for publicized security problems still can take weeks to become available using this approach [Tea97].

1.3.3 Build at the User Level

In an attempt to overcome the tremendous costs of implementing new operating system functionality in the kernel, many developers have constructed new operating system services as user-level layers, logically situated between applications and the operating system. Examples of such systems include LSF [Zho92], Condor [BLL91], Libckpt [PBKL95], Ufo [AISS97], PPFS [HER+95], and U-Net [vEBBV95]. Implementing cluster support and other system services at the user level affords a number of advantages relative to kernel implementations: well-defined and well-documented interfaces (i.e., system calls and signals) simplify the implementation of new functionality; state-of-the-art development and debugging tools such as `gdb`, `purify`, and `quantify` are readily available; and finally, widely-supported standards such as POSIX facilitate porting to numerous platforms.

However, user-level approaches suffer from serious limitations. Building cluster services requires extending some UNIX interfaces which are not exposed to the user level, such as kernel access to application memory and group signal delivery. Other interfaces which are exposed to the user level must be extended in a way that is protected from malicious applications and enforced on uncooperative applications. To arbitrate shared cluster resources among competing applications, a cluster system must be protected from malicious or faulty applications and must be enforced on uncooperative applications. However, user-level approaches cannot currently provide protection and enforcement while maintaining low overhead. A number of user-level projects place new functionality in a library linked into the application's address space, including Condor, LSF, Libckpt, and Interposition Agents [Jon93]. Because new functionality runs in the application's address space, it is unprotected from malicious applications and consequently cannot enforce security guarantees or manage shared resources for competing applications. Placing new functionality in a separate process provides the necessary protection, but still requires application cooperation because the new functionality is not enforced on the application—applications must voluntarily access the new functionality by sending messages to the process implementing the new functionality. A malicious application can circumvent the new functionality and thwart resource sharing policies or security guarantees by accessing the original kernel functionality directly.

A number of hardware and software techniques can be used to enforce new functionality on an application and protect functionality loaded in the same address space as the application. The System V `/proc` pseudo-device [FG91] can be used to reliably intercept all system calls of an application, preventing the application from circumventing system call extensions. The `/proc` architecture, however, requires that the system call be intercepted by a separate user-level process, requiring costly address space context switches on each system call interception. Binary rewriting techniques such as Software Fault Isolation (SFI) [WLAG93] and EEL [LS95] can be used to protect new functionality from applications even when loaded in the same address space and can enforce functionality on applications by re-vectoring trap instructions to the new functionality. Unfortunately, SFI requires a number of compiler optimizations to achieve low overhead and therefore cannot be applied efficiently to existing binaries. Proof-carrying Code (PCC) [NL96] provides protection and enforcement as SFI and EEL but eliminates costly run-time overhead by annotating extensions with a proof that the code obeys some safety policy. Current proof sizes, however, are exponential in the size of the extension. Though generating these proofs is undecidable in the general case, they can be simplified by inserting run-time checks into the system, leading to a tradeoff between proof complexity and run-time overhead. Protected Shared Libraries (PSL) [BTC97] has the ability to protect libraries without the need for compiler optimizations, but requires per-thread address space control and does not enforce extensions on applications. The presence of these extension technologies demonstrates the crucial need to extend operating system functionality.

Furthermore, because user-level applications are untrusted, operating systems place many restrictions on user-level code. Consequently, only a very narrow class of system functionality can be implemented at the user level. For instance, in order to ensure fair resource sharing in the face of competing or malicious processes, operating systems do not trust applications to make global resource allocation and protection decisions or to arbitrate access to shared resources, effectively eliminating a large class of operating system functionality from being implementing in this manner. As an example, many operating systems have system calls which an application may use to provide hints to the virtual memory system about that application's data access patterns, but no commodity operating system today provides a way for user-level applications to define a completely new policy for allocation virtual memory pages among competing applications. Consequently, such virtual memory policies cannot be built at the user level using the functionality provided by today's commodity operating systems. User-level applications (and consequently, user-level operating system services) are limited to the functionality, performance, and policies provided by the underlying operating system.

1.4 Recent Alternatives

A number of academic research projects have sprung up in recent years to explore a fourth method for implementing operating system functionality that is less complex than modifying an operating system directly yet does not suffer from the numerous limitations of user-level approaches. These systems re-engineer kernel interfaces in order to support third-party extensibility; examples include Lipto [DPH92], Spin [BCE+94], VINO [SESS96], Exokernel [EKJO95], and Fluke [FHL+96]. While these systems require replacing or heavily modifying the operating system, once built they dramatically reduce the cost of implementing new operating system functionality by enabling third-party extensibility. Perhaps the most extreme example of this trend is the Exokernel [EKJO95] in which the bulk of operating system services are implemented as application-level libraries. While these projects have not yet fully addressed the issues of shared resource allocation and arbitration policies, they have demonstrated a qualitative improvement in extensibility. However, these systems have substantially different architectures from current commodity operating systems. Bringing such radically new operating system structures into main-stream commodity use would require wholesale replacement of current operating systems and would take, at best, many years. Consequently, for the foreseeable future, the existing commodity operating system base will remain difficult to modify by traditional methods.

A variant of this fourth approach is to make a single kernel interface extensible, rather than the entire system. The `vnode` interface [Kle86] is a prime example of this approach. Later efforts have explored modifications to the `vnode` interface to allow or composability of extensions [Ros90, KN93, HP95, MBKQ96]. However, because of significant internal differences across operating systems for such interfaces [Web93], extensions implemented on these interfaces are not easily portable to other systems. In ddition, applying this technique to make existing commodity operating systems generally extensible would require modifying and exposing all interfaces where additional functionality is desired, effectively re-engineering the majority of the operating system. Again, for the foreseeable future, beyond a few narrow examples, such interfaces are unlikely to become generally available in commodity operating systems.

A less dramatic solution can be found in the `/proc` pseudo-device [FG91] of many UNIX operating systems. `/proc` allows trusted processes to intercept system calls and signals destined for other processes and to perform a variety of operations; it does not require overhauling the operating system as was done for extensible operating systems. Thus, `/proc` can be used by third-party applications to create a modified operating system environment for one or more applications, effectively adding new functionality to the system without requiring access to or modification of kernel source code. However, current `/proc` systems are limited in functionality and performance. For example `/proc` can be used to intercept and modify the parameters of a system call, but cannot be used to securely modify the application data buffers referenced by the system call arguments; `/proc` also does not have access to group signal information. Additionally, `/proc` systems are designed so that extension code is located in a separate user-level process. While this separation protects sensitive extension code from potentially malicious or faulty applications, it limits performance since every invocation of the extension code requires an address space context switch. Performance is therefore only acceptable for extensions which are rarely invoked. The performance of `/proc` can be improved significantly by relocating the extension code into the same address space as the application (as was done by Interposition Agents) but this arrangement leaves the extension code unprotected from and unenforced upon applications, similar to the user-level library approach described above.

1.5 Our Solution

This dissertation enables third-party extensions of commodity UNIX operating systems in a way that combines the advantages of the user-level and kernel approaches while eliminating many of their disadvantages. Specifically, our approach combines the development tools, well-defined interfaces, and portability of user-level code with the enforcement, protection, low overhead, and broad functionality advantages of in-kernel solutions, while requiring little or no modification to existing operating system kernels and no modifications to existing application binaries.

We use the concept of interposition to insert trusted extension code directly into existing commodity UNIX operating system kernels, intercepting various operating system events and re-routing those events to trusted extension code located at the user level (for ease of development) or in the kernel (for performance). Each extension is transparently interposed on operating system interfaces—that is, the extension both uses and provides the kernel's event interface, enabling extensibility with little or no kernel modification and enabling the transparent stacking of multiple extensions. By intercepting and modifying system events such as system calls and signals, extensions can implement a wide variety of operating system extensions without having to rewrite or modify the base operating system. By re-routing intercepted events to extension code located at the user level, the system supports use of advanced user-level development and profiling tools. After development and testing are complete, performance-critical sections of an extension can be placed in the kernel to maximize performance.

This solution enables the development and deployment of cluster services which do not suffer from the traditional limitations of user-level implementations while avoiding the costs associated with adding such functionality in the kernel. Cluster operating system services can be implemented at the user level using small extensions to expose the kernel abstractions and interfaces necessary to virtualize UNIX semantics to a cluster. More generally, this solution simplifies adding multiple third-party extensions to existing commodity UNIX operating systems, thus closing the gap between research and production systems for a large class of extensions. By dramatically reducing the cost of implementing new operating system functionality, systems will also be able to respond more rapidly to hardware advances and security breaches.

While evaluated in the specific context of UNIX operating systems, this approach is also applicable to other operating systems such as Windows NT. Conceptually, operating systems provide abstractions of physical resources. In many cases, however, the abstractions provided by the system designers to applications are inadequate for some purposes. By transparently interposing on relevant kernel interfaces, extensions can be introduced into the system to augment system behavior in ways not easily done, and often not possible, from the user level.

1.6 Contributions

The hypothesis of this work is that it is possible to provide extension developers with the advantages of both kernel-level and user-level approaches simultaneously through interposition on existing kernel interfaces. Specifically, interposition on existing kernel interfaces enables rapid, low-cost development of a broad range of operating system functionality which is protected from malicious applications, is enforced on uncooperative applications, and can be accomplished with few or no modifications to existing kernels, is compatible with existing application binaries, and incurs low overhead. This hypothesis will be evaluated through an in-depth case study of GLUnix [GPR+98], a complex, user-level cluster system supporting an active user community. We will begin by describing the GLUnix system, illustrating the functionality and abstractions needed to effectively support a cluster environment. We will then identify four primary limitations in the functionality and performance of GLUnix arising from the inability of a user-level system to access the necessary kernel interfaces. We then present our prototype interposition system, Slic, a powerful, general-purpose extension mechanism. We then show how the extensibility provided by Slic enables GLUnix to gain access to the necessary kernel interfaces to overcome the four limitations identified earlier. Finally, we discuss a challenge introduced by the generality of Slic and experienced by all general-purpose extensibility mechanisms—extension composability conflicts. We define the problem and describe heuristics which can be utilized to alleviate the problem.

There are four main contributions of this work. First, this research provides a detailed characterization of a number of the limitations of user-level operating system services implemented on today's commodity UNIX operating systems; while this discussion uses GLUnix as a specific example, similar problems would be encountered in implementing other complex functionality, such as a cluster file system, and in non-UNIX operating systems which restrict application access to kernel state, such as Windows NT. Second, we demonstrate how small, in-kernel extensions can be used to overcome many of the limitations of user-level operating system services. Third, based on the experiences gained in interposing new functionality on existing kernel interfaces, we present a characterization of "interposable" interfaces, evaluating the characteristics of interfaces which affect the ability to interpose and describing how to design an interface to best support interposition; we briefly evaluate a number of existing operating system interfaces in this framework. Finally, we characterize the problem of extension stacking conflicts and outline a number of strategies for solving the problem.

1.7 Organization

In Chapter 2, we evaluate the construction of new operating system services at the user level through the example of GLUnix [GPR+98], a user-level operating system layer providing cluster services to a network of workstations [ACPtNT95], including remote execution, load balancing, support for parallel programs, job control, and integration with existing UNIX shells. In Chapter 3, we examine how GLUnix was unable to reach its goals due to user-level limitations. Chapter 4 introduces a general-purpose extensibility mechanism for existing commodity UNIX operating systems and describes the current prototype, SLIC [GPRA98]. Chapter 5 applies the general solution of SLIC to the particular problem of the user-level limitations encountered by GLUnix. Chapter 6 discusses the issues involved in stacking extensions while Chapter 7 concludes.

Chapter 2

GLUnix: Extending UNIX Semantics to a Cluster

To illustrate the need for adding new features to operating systems and to provide the necessary background for identifying the limitations of adding those features, this chapter presents the design and implementation of a real-world clustering system, GLUnix. GLUnix provides support for remote execution of parallel and sequential jobs on a network of workstations, but was unable to fully support transparent remote execution or process migration because of the limitations arising from its user-level construction. This chapter discusses the functionality needed by a cluster operating systems layer such as GLUnix to effectively harness cluster resources. In the next chapter, we will describe specific limitations encountered by GLUnix. Later chapters will demonstrate how kernel-level extensibility can be used to overcome these limitations.

2.1 Introduction

Historically, computing models have been separated into three distinct paradigms, each optimized for a particular class of applications: desktop workstations used primarily for interactive jobs, centralized compute servers such as supercomputers and mainframes used for compute-intensive or I/O-intensive batch jobs, and massively parallel processors (MPPs) used for batch parallel jobs. However, technology trends of the past two decades have steadily eroded the boundaries of these paradigms. The narrowing performance gap between commodity workstations and supercomputers as well as the availability of commodity high-speed local area networks [BCF+95, dP93, Ano96] has led to the belief that networks of workstations (NOWs) can support all three types of workloads [ACPtNT95].

By leveraging commodity high-performance workstations and networks, the primary challenge in building a NOW shifts to the system software needed to manage, control, and access the cluster. To effectively support the three traditional workloads, NOWs must support all the features available from workstations, MPPs, and supercomputers, including interactive parallel and sequential jobs, transparent remote execution of existing applications, and dynamic load balancing across the cluster. To keep pace with exponential improvements in hardware performance, the software should also be quickly portable to new platforms.

However, no existing cluster system supports all of these features. LSF [Zho92] and Utopia [ZWZD92] provide support for remote execution of interactive sequential jobs, but do not provide dynamic load balancing or fully support parallel jobs. The architecture used by both also does not provide the protection necessary to enforce cluster resource policies such as node reservations. PVM [Sun90] supports parallel jobs but lacks the gang scheduling necessary to run closely synchronized parallel jobs efficiently [DAC96]; PVM also does not provide integrated support for parallel and interactive sequential jobs, nor does it provide dynamic load balancing. Condor [BLL91] provides the migration necessary to dynamically balance cluster load, but does not transparently support existing binaries: jobs are restricted to a limited set of operating system features and must be re-linked with a special library; because of the functional restrictions, Condor is also not capable of supporting interactive jobs. Sprite [OCD+88] and V [Che88] provided dynamic load balancing through process migration [DO91] and I/O and job control for remote jobs, but did not support parallel jobs. However, rather than leveraging operating system advances by building on top of existing operating systems, these systems substantially modified the kernel to provide the virtualization necessary to support remote execution; because of the high cost involved in porting such systems to new workstation hardware, these systems do not support modern architectures and thus are unavailable to general users.

GLUnix (*G*lobal *L*ayer *Unix*) was designed to transparently provide a single, coherent cluster system on top of commodity UNIX systems in order to leverage hardware and operating system advances. Its design supports interactive and batch-style remote execution of both parallel and sequential jobs, maintains traditional UNIX semantics for I/O and job control, provides load balancing through intelligent job placement and dynamic process migration, and tolerates single-node faults. The system is built at the user level to minimize the effort required to leverage improvements in the underlying operating system and to port the system to new platforms. The next chapter will discuss the reasons why GLUnix was unable to completely fulfill this vision while remaining a portable, user-level solution.

While our discussion centers around GLUnix, the lessons learned are generally applicable to other systems attempting to provide secure, transparent access to remote resources at the user level. We begin this chapter by describing in detail our goals and the abstractions needed to meet those goals. This leads to an architectural overview of GLUnix as well as implementation status and usage statistics. We then provide an evaluation of the scalability, performance, and architectural design of GLUnix. Finally, we conclude with a discussion of related work.

2.2 Goals and Features

GLUnix was conceived in 1993 as the global operating system layer for the U.C. Berkeley NOW project. A primary goal of the NOW project was to construct a platform that could simultaneously execute interactive parallel and sequential jobs with negligible slowdown for either class of workload. Ideally, a NOW should guarantee one workstation's worth of performance to each user at all times, with the possibility of recruiting all available cluster resources (CPU, disk, memory, network) for parallel programs as needed. The feasibility of supporting integrated parallel and sequential workloads was established by an early simulation study of workloads measured from a 32-node CM-5 at Los Alamos National Laboratory and a 70-node workstation cluster at U.C. Berkeley [ADV$^+$95]. The study concluded that harvesting idle cycles on a 60-node workstation cluster could support both the parallel and sequential workloads studied with minimal slowdown.

Given our target NOW environment, we identified the following set of goals for our cluster software system:

- *Portability:* To keep up with the rapid pace of hardware evolution and to enable heterogeneous clusters, the system should be written to be quickly and easily portable to new operating systems or hardware platforms.
- *Timeliness:* An initial implementation of the software should be up and running user jobs as soon as possible. The complete system should be finished by the end of the NOW project.
- *Binary Compatibility:* The system should provide remote execution, load balancing, and coscheduling without requiring application modification or relinking. Existing applications should be able to benefit transparently from new features.
- *High Availability:* When a node crashes, the system should continue to operate. However, jobs with a footprint on a failed node can be terminated. The system may provide hooks for application fault-tolerance mechanisms.
- *Performance*: Sequential and interactive applications should run as if a single dedicated workstation were available to them. Similarly, parallel workloads should run as if a dedicated supercomputer were available.

We then identified the set of features that the system would need to provide in order to effectively utilize clusters for workstation, MPP, and supercomputing workloads:

- *Transparent Remote Execution:* Applications should be unaware that they are being executed on remote nodes. They should retain access to home node resources including shared memory segments, processes, devices, files, etc. Users should be able to interact with remote jobs as if they were running locally; support should be included for existing shells, job control, standard I/O piping and redirection, etc.
- *Single System Image:* There should be a single, cluster-wide view of cluster resources as if the cluster were a single machine. Users should be able to query job status or signal jobs from any node in the system.
- *Load Balancing:* The system should provide intelligent job placement policies to determine the best location to initially execute each job. The system should also dynamically migrate jobs either as cluster load becomes unbalanced or as nodes are dynamically added to or removed from the system.

- *Integrated Support for Parallel Jobs:* Users should be able to execute and interact with parallel jobs using standard UNIX semantics (e.g., job control and terminal I/O) when applicable. Efficient execution of a communicating parallel job requires that the individual processes be scheduled together [Ous82, FR92, ADV+95, DAC96]. In order to synchronize parallel computation, parallel jobs also need support for barrier operations.

2.3 GLUnix Abstractions

To provide remote execution of both parallel and sequential jobs, GLUnix must introduce some new abstractions as well as extend a number of existing UNIX abstractions to the cluster. Borrowing from MPP environments such as that of the CM-5, the new abstractions include *network programs* with globally unique *network program identifiers* (NPIDs) for cluster jobs and *virtual node numbers* (VNNs) to name the individual processes running within a parallel network program. The existing abstractions of signal delivery, standard I/O redirection, and scheduling must be extended to support parallel and remote jobs.

2.3.1 Network Programs

A *network program* is an executing parallel or sequential job that is controlled by GLUnix. Network programs can be located anywhere in the cluster and are identified using a 32-bit, cluster-unique *network program identifier* (NPID) which is assigned and tracked by GLUnix. Using a cluster-wide, location-independent identifier provides the level of virtualization necessary to enable transparent remote execution and dynamic process migration.

2.3.2 Parallel Programs and Virtual Node Numbers

GLUnix supports parallel programs that use the message passing, Single-Program-Multiple-Data (SPMD) programming model. In the SPMD model, a collection of N separate processes run the same application binary and use message passing to work cooperatively on a single computation. A parallel program is identified by a single NPID. The number of individual processes comprising the parallel program is referred to as the *parallel degree* of the program[1]; processes within a parallel program are distinguished by the use of *virtual node numbers* (VNNs), numbered $0 \ldots N-1$. Just as with NPIDs, VNNs are location-independent and enable transparent process placement and migration within the GLUnix cluster.

GLUnix also support the Multiple-Program-Multiple-Data (MPMD) model in the way that UNIX does via communicating sequential processes. GLUnix does not support distributed shared memory programming models at this time.

[1]GLUnix treats sequential programs as parallel programs with a parallel degree of 1.

2.3.3 Signal Delivery

In UNIX, signals are used to indicate exceptional conditions (page faults, I/O errors, etc.) or for process control (to suspend or terminate a running program). Signal delivery to individual processes and process groups must be extended to include remote jobs and distributed process groups. For signaling purposes, GLUnix treats a parallel program as a single unit by default, delivering the signal to each individual process in the parallel program. GLUnix also supports extended signaling semantics for parallel jobs, enabling a single individual process of a parallel program to be signaled.

2.3.4 I/O Redirection

GLUnix extends the UNIX abstractions of standard I/O byte streams (i.e. `stdin`, `stdout`, `stderr`) to remotely running sequential and parallel programs, including file redirection and pipes [RT74]. Input given to a GLUnix program, either from the console keyboard, a file, or a pipe, is replicated by default to all processes of a parallel program. Analogously, output from network programs is multiplexed onto a single output stream, whether directed to the user's console, a file, or a pipe. The `stdout` and `stderr` streams from remote processes are kept distinct as in traditional UNIX, allowing the user to pipe `stdout` to a file while `stderr` remains connected to the console. GLUnix further adds additional I/O modes to network programs; for example, each line of output from remote processes of a parallel program may be prepended with the VNN of the process which generated that line.

2.3.5 Parallel Program Support

Like MPP operating systems, GLUnix provides specialized scheduling support for parallel programs in the form of barriers and coscheduling [Ous82]. Since individual parallel program instructions are not executed in lock-step in the SPMD or MPMD models, barriers can be useful for synchronizing cooperating processes, enabling programmers to make assumptions about the state of computation and data in other processes. When an individual process of a parallel program enters the barrier, it is blocked until all other processes of the program have also entered the barrier. Once this occurs, all processes in the program are released from the barrier and resume execution.

Coscheduling, or gang scheduling, ensures that all processes of a single parallel job are scheduled simultaneously and provides coordinated time-slicing between different parallel jobs. In the absence of coscheduling, the individual operating systems in the NOW would independently schedule the processes of a parallel application; on those nodes in the system that have more than one job to run, there would be guarantee that the parallel program would be simultaneously chosen by all local schedulers. User-level cluster systems typically take this approach (e.g., PVM [Sun90]). However, a number of studies [CDD+91, FR92, ACK+95, DAC96] demonstrate that the lack of coscheduling leads to unacceptable execution times for frequently-communicating processes. This slowdown occurs when a process stalls while attempting to communicate or synchronize with a currently unscheduled process.

2.3.6 Requirement Summary

Supporting these new and extended abstractions places a number of requirements on GLUnix. First, it must provide a cluster-wide, coherent name space for NPIDs and VNNs. Second, signals and I/O must be transparently redirected to remote jobs and multiplexed to parallel jobs. And third, GLUnix must schedule parallel programs simultaneously across different nodes and provide support for barrier synchronization.

2.4 GLUnix Architecture

This section presents the GLUnix architecture, our solution to the requirements needed to support transparent remote execution, load balancing, and integrated support for parallel jobs. We begin by presenting the basic assumptions of the design and then discuss the design itself, illustrating how the system works.

2.4.1 Assumptions

In constructing GLUnix, we made two primary assumptions concerning the target environment: shared filesystems and homogeneous operating systems. Since the Berkeley NOW project included a separate group implementing a high performance global file system [ADN+95], GLUnix assumes that each node of the cluster has a single system image with respect to the file system, making file paths valid on all nodes. Further, GLUnix uses a shared file to store the network address of the centralized GLUnix master for bootstrapping. Our cluster currently uses an NFS-mounted file system for this purpose.

The second assumption was that operating systems and binary formats would be homogeneous across all nodes in the cluster. That is, a single GLUnix cluster can include SPARCStation 10's, 20's, and UltraSPARCs running Solaris 2.3–2.6, but it cannot include both SPARC and 80x86 machines. This assumption simplified the implementation of GLUnix by eliminating difficulties with byte ordering, data type sizes, and data structure layout for internal GLUnix communication services and ensures that application binaries can run on any machine in the cluster. Enabling multi-platform support for GLUnix would require porting to a network data transport mechanism such as XDR [Sun87] and developing a mechanism for supporting multi-platform binaries.

2.4.2 Architecture Alternatives

We faced three major design decisions in the development of GLUnix: kernel level vs. user level, centralized vs. decentralized, and multi-threaded vs. event-driven.

Kernel Level vs. User Level

The first and perhaps most important choice was to decide how much functionality to leverage from the underlying system. The options were to (i) build a new operating system, (ii) modify an existing operating system, or (iii) "build on top" of existing systems (now called "middleware"). Though it would likely provide the best performance, the first option of building a new operating system was immediately rejected because we felt that building a new OS was completely incompatible with our goals of portability and timeliness, far outweighing any potential performance improvements. Modifying an existing operating system enables the use of all in-kernel services, not just those available via the user application programming interface (API), and allows changing the implementation of those services to provide additional functionality or better performance. Existing applications can transparently take advantage of these modifications. The tradeoff is that making kernel modifications requires an understanding of an existing kernel—a large and complex system containing hundreds of thousands of lines of code. Kernel modifications are often not portable between successive releases of the same operating system [Web93], and not at all between different operating systems. In addition, operating system kernels typically lack state-of-the-art debugging and development tools, hindering development. Finally, understanding and modifying operating system kernels is a time-intensive task[2].

Building a middleware layer essentially reverses the advantages and disadvantages of modifying the operating system. Programming at the user level only requires an understanding of the kernel's API, which is considerably simpler than understanding kernel internals. Development is also assisted by numerous development and debugging tools which are frequently unavailable for kernel modifications. User-level layers can be more portable than kernel modifications, although to ensure portability, code must be restricted to a widely-available, standardized set of features (such as POSIX). However, middleware layers are limited by the functionality and performance of the API provided by the underlying operating system. In particular, we were concerned that the limitations of previous user-level systems [BLL91, Zho92, PBKL95] were an indication that the functionality required for transparent remote execution was simply not available at the user level. We concluded that the advantages of portability and timeliness made a user-level middleware layer the best option at the time. To alleviate the limitations of a user-level implementation while retaining compatibility with existing binaries, we planned to explore binary rewriting techniques [WLAG93, LS95] or develop novel kernel extension techniques.

Given the choice to implement a user-level solution, we then considered the question of whether all programs on a node would run under GLUnix or only GLUnix-aware programs. We chose a multi-phased approach: initially, programs must specifically request to run under GLUnix, either by user command or by application control; as we identified and solved transparency issues, we would gradually subsume more programs under GLUnix control until all programs on each node were run under GLUnix. This path enabled us to maintain the stability of the nodes during initial development and debugging (for example, we did not want operating system daemons running under GLUnix until the system was stable).

Centralized vs. Decentralized

Our next design decision concerned the high-level system architecture of GLUnix: a centralized or decentralized design. Centralized systems, with a single master node and a collection of client nodes, are relatively straightforward to design, build and debug. However, the master can be a single point of failure as well as a performance bottleneck. Fully

[2] In 1993, the only operating systems available for our hardware were commercial products; simply obtaining the source code is often a time-consuming task.

decentralized (peer-to-peer) systems do not depend on any single node and consequently are more resilient to failure and can scale more gracefully to large clusters. Decentralization, however, complicates state management and synchronization, making such systems more difficult to design and debug. The scalability of previous centralized systems [The94, Che88] led us to decide to begin with a simple, centralized design and to later decentralize system components as needed.

Multi-threading vs. Event-driven

Our final design decision concerned whether to use a multi-threaded or event-driven programming model for individual GLUnix components. Multi-threaded designs have two primary advantages over event-driven programming: increased concurrency with the potential for improved performance, and the relative simplicity of dealing with blocking operations [Bir89, Ous96]. Maintaining performance in a distributed system with event-driven programming requires that network message handlers use only non-blocking communication operations, treating reply messages as separate events. Event-driven architectures, however, are simpler to design and debug, since they avoid the deadlocks and race conditions of multi-threaded programming. Ultimately, the lack of a portable kernel thread package and thread-aware debuggers in 1993[3] convinced us to adopt an event-driven architecture for GLUnix.

2.4.3 Architecture Overview

GLUnix consists of three components: a per-cluster master, a per-node daemon, and a per-application library (see Figure 2.1). The GLUnix master is primarily responsible for providing a single system image for processes by storing and managing cluster state (e.g., the process name space for the cluster and load information for each node) and for making centralized resource allocation decisions (e.g., job placement and coscheduling). The GLUnix daemon running on each node collects information for the master, checking local load information every few seconds and notifying the master of changes. The daemon also serves as a remote proxy for the master in starting and signaling jobs and in detecting job termination. The GLUnix library implements the GLUnix API, providing a way for applications to request GLUnix services. Unmodified applications which have not been linked with the GLUnix library can still be run remotely via GLUnix and can still use standard UNIX facilities such as I/O and job control, but will not have access to advanced cluster features (such as parallel execution) provided by the GLUnix API.

Sequential and parallel jobs are submitted to GLUnix by a startup process which is linked to the GLUnix library. The startup process issues the execution request to the GLUnix master and then acts as a local proxy for the user's job, intercepting job control signals and forwarding them to the master and receiving I/O from the remote program and displaying it for the user.

Figure 2.2 depicts the logical internal structure of the master, the daemons, and the GLUnix library. Table 2.1 summarizes the primary function of each module. The `Comm` module is currently implemented using Berkeley Stream Sockets over TCP/IP, though it is

[3] For example, one of our target architectures, HP-UX, did not have kernel support for threads until 1996.

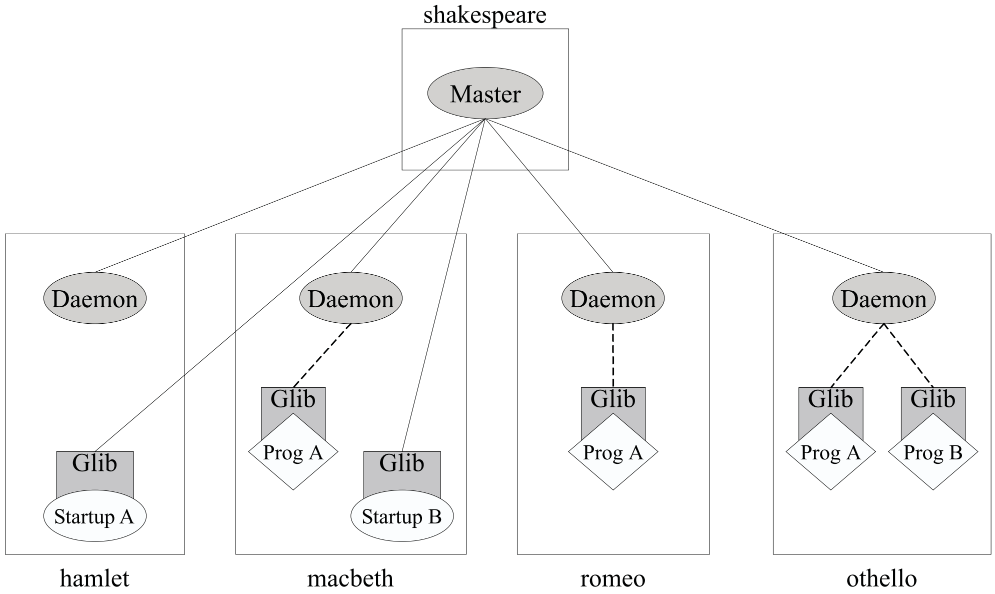


Figure 2.1: The basic structure of a GLUnix cluster. The solid lines indicate message connections between GLUnix components; the dotted lines indicate parent-child process relationships. In this example, the master is located on the machine named `shakespeare` and daemons are located on `hamlet`, `macbeth`, `romeo`, and `othello`. A user on `hamlet` has executed job A, a parallel job consisting of three processes located on `macbeth`, `romeo`, and `othello`. A second user has executed a sequential job B which GLUnix has placed on `othello`. The I/O connections from the startup to the remote processes are not represented.

designed to be easily replaceable by a module using faster communication primitives, such as Active Messages [vECGS92].

The main event loop is located in the `Comm` module. GLUnix events consist of both messages and signals. When a message arrives, the `Msg` module unpacks the message into a local data structure and then invokes the appropriate message handler. Outgoing messages are also packaged up by the `Msg` module and then transmitted by the `Comm` module. When a signal event arrives, the event loop invokes the appropriate handler for the signal. Each module registers handlers for the messages or signals they are interested in.

2.4.4 Transparent Remote Execution

GLUnix must take special care to make remote execution transparent to users and applications. When users run GLUnix programs from the command-line, the process `exec`'ed by the user's shell calls the `Glib_Spawn()` routine in the GLUnix library. `Glib_Spawn()` sends the GLUnix master a remote execution request containing the command-line arguments, the parallel degree of the program, and the user's current environment including groups, current working directory, umask, and environment variables. `Glib_Spawn()` then converts the calling process into a GLUnix startup process. When the master receives the execution request, it uses load information periodically sent from the daemons to select the N least loaded nodes in the system. Users can influence this selection process by supplying a description of the candidate nodes to select from (e.g., "`u0..u50,u73`") and can even disable the load sorting process to support controlled experimentation (e.g., "`=u0..u50,u73`"). Once the nodes have been selected, the `Rexec` module of the master registers the new network program with the `Pdb` module and then sends the execution request to each of the `Rexec` modules of the selected daemons. Upon receipt of the execution request, the `Rexec` module in the remote GLUnix daemon establishes I/O connections with the startup process for `stdin`, `stdout`, and `stderr`. The remote daemon's `Rexec` module then performs a `fork()`, recreates the environment packaged up by the startup process, and finally does an `exec()` of the program with the original command-line arguments. The remote daemon will report success or failure of the execution to the master, providing an error code in the latter case which is passed back to the startup process.

To extend UNIX signaling semantics to remote jobs, the startup process, on initialization, registers handlers for all catchable signals. When the startup catches a signal, sent to it either by the `tty` in response to a `Ctrl-C` or `Ctrl-Z` or via the `kill()` system call, the startup process sends a signal request to the master. The master looks up the target network program in the `Pdb` module and forwards the signal request to the daemons which are managing the processes of this network program. Those daemons then signal the appropriate local processes using the `kill()` system call.

GLUnix also extends standard I/O semantics of UNIX to remote jobs. UNIX provides applications with two distinct output streams, `stdout` and `stderr`. While `stderr` data is usually directed to a user's console, `stdout` can be easily redirected to a file or to another program using standard shell redirection facilities. To provide standard I/O to remote programs, GLUnix establishes three TCP/IP I/O connections between the startup process and each individual process in a network program. At the time of execution, the startup process creates an I/O listen socket, the address of which is forwarded to remote

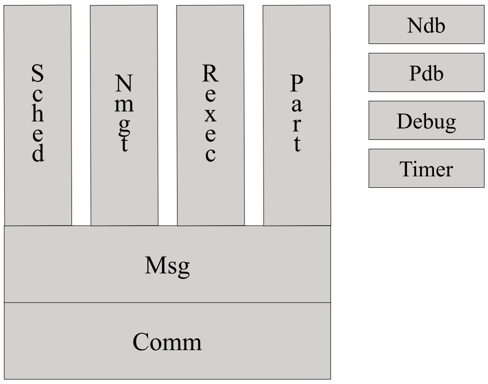


Figure 2.2: The logical internal structure of the GLUnix master, daemon, and library. Although the module structure is the same for all three, most modules have specialized code for each.

Comm	Abstracts away network communication; contains the main event loop of GLUnix
Msg	Performs message packing and unpacking, invokes handlers for incoming message
Ndb	Node database storing and managing state relating to each node in the system
Pdb	Process database which keeps track of the currently executing GLUnix processes
Sched	Implements coscheduling policy algorithm
Nmgt	Performs node management functions: collecting load data, detecting failed nodes
Rexec	Performs selection of least-loaded nodes and remote execution
Part	Manages GLUnix partitions and reservations
Debug	Defines multiple debug levels, supports dynamic selection of debug levels
Timer	Schedules periodic events

Table 2.1: Descriptions of the function of the GLUnix modules.

daemons in the execution request message sent to the master. Remote daemons receiving the request establish I/O connections after doing the `fork()` but before doing the `exec()` of the child process. Three I/O connections are established between the new child process and the startup process: one connection for `stdin`, `stdout` and `stderr`[4]. There must be separate streams for `stdout` and `stderr` in order to keep the stream data distinct and maintain UNIX I/O semantics. Many remote execution facilities such as `rsh` and PVM [Sun90] do not keep the two output streams separate and thus do not maintain proper redirection and piping semantics.

2.4.5 Barriers and Coscheduling

To effectively support parallel applications, GLUnix must provide support for coscheduling and barrier synchronization. GLUnix implements an approximation of coscheduling through a simple user-level strategy. To determine a time-slice order for all runnable parallel programs, the GLUnix master uses a standard matrix algorithm [Ous82] in which the columns of a 2-dimensional matrix represent nodes in the system and the rows represent time slices. Entries in the matrix represent individual processes of a parallel jobs; parallel jobs are placed in the first row with sufficient free space to run the job. Periodically, the master schedules a new row in the matrix, sending a message to each daemon identifying which NPID should be scheduled on that node. Each daemon maintains a NPID to local UNIX `pid` mapping in its process database. The daemon uses this mapping to send a UNIX `SIGSTOP` signal[5] to the currently running parallel process, and a `SIGCONT` signal to the newly runnable process. We chose this technique to avoid kernel modifications, maintaining an entirely user-level solution.

This solution is, however, only an approximation of coscheduling. By sending a `SIGSTOP` to those GLUnix processes which should not currently be running, GLUnix reduces the set of runnable processes which the local operating system will choose from. The current implementation of GLUnix, however, does not stop all other processes in the system, including system daemons or other non-GLUnix jobs. Thus, there may be interactive, non-GLUnix jobs running during what GLUnix considers to be the parallel program's time slice.

The GLUnix barrier is implemented using a standard binary-tree algorithm. A process of a parallel program invokes the `GLUnix_Barrier()` library call which sends a barrier request message to the local daemon. That daemon identifies the other daemons which are managing the processes for this parallel program. The daemons then use a standard binary tree reduction algorithm to determine when all processes of the parallel program have entered the barrier. When the daemon at the root of the barrier tree detects completion of the barrier, the release message is broadcast back down the tree, with each daemon sending a reply message to its local process, indicating that the barrier has completed. At this point the `GLUnix_Barrier()` library call will return and the process will continue executing. While barriers can also be implemented in parallel programs using message passing,

[4] While `stdin` can be multiplexed onto one of the output streams, a separate stream is currently used for convenience.

[5] UNIX processes cannot catch `SIGSTOP`. Thus, there are no worries about malicious programmers ignoring scheduling directives.

placing barriers in GLUnix provides a boot-strapping step necessary for some message passing layers and can simplify applications by eliminating the need to reimplement the barrier algorithm.

2.4.6 GLUnix Reservations

System developers as well as researchers often need exclusive access to a particular set of machines to take performance measurements or to test experimental kernels. GLUnix provides this functionality through a naming and reservation system. The system works by associating three lists with each machine: *aliases*, *users*, and *owners*. The alias list simply contains a list of valid names for each machine; the user list identifies the users who currently have permission to run applications on that machine; and the owner list specifies those individuals who have permission to reserve that machine. When running an application, a user can provide a *node specification*[6] to indicate the set of nodes which should be used to run the program. When receiving an execution request, the GLUnix master expands the node specification into a set of machines, then filters that set of machines by the user list associated with each machine, choosing the nodes to execute the job from the resulting set, thus ensuring that the application will run only on those node for which the user has permission. To make a reservation, the user supplies a specification of the nodes to reserve, the start and end times of the reservation, and a name for the reservation. When the time for the reservation arrives, GLUnix filters the reservation's node specification by the machines' owner lists, preventing users from reserving machines for which they do not have permission. GLUnix then assigns the reservation name as an alias to each machine selected. Users then indicate their desire to run on the reserved nodes by placing the reservation name in the node specification.

In addition to dramatically simplifying the process of reserving machines, the ability to associate a set of aliases with a group of machines and then to use that name to identify them has proved to be extremely useful. At many points in the project we had rapidly changing cluster configurations as different versions of networking software were installed and disks and memory rearranged. In addition, we have different architectures: SPARCStation 10's, 20's and UltraSPARCs, in both single processor and multiprocessor configurations. By assigning aliases to machines, users can refer to a set of machines by some resource characteristic rather than having to remember the current configuration. When the cluster configuration is modified, the aliases are simply updated, allowing subsequent uses of the alias to properly use the new configuration.

2.4.7 Failure Recovery

GLUnix is currently able to tolerate failures of the GLUnix daemons, startup processes, and, of course, user programs, or the machines where any of these are running. These failures are detected by the GLUnix master when its network connections fail. During normal operation, the GLUnix master maintains an open connection to each GLUnix daemon and startup process. When the master identifies a broken connection to a daemon,

[6] The node specification is actually a boolean equation consisting of aliases and set operations on those aliases. For example: "(myrinet2.3$\wedge$solaris2.6)$-$desktop".

`int Glib_Initialize();`	Enable GLUnix services for this program.
`int Glib_Spawn(int numNodes, char *progPath, char **argv);`	Start program `progPath` on `numNodes`.
`int Glib_CoSpawn(Npid npid, char *progPath, char **argv);`	Start up program `progPath` on the same nodes that network process `npid` is running on.
`int Glib_GetMyNpid();`	Get the NPID of this job.
`int Glib_GetMyVNN();`	Get the Virtual Node Number of this node within the program.
`int Glib_GetIpByVNN(VNN vnn);`	Get the IP address of the virtual node `vnn` in this process.
`int Glib_Signal(Npid npid, VNN vnn, int sig);`	Send a signal to the virtual node `vnn` within the network process `npid`.
`int Glib_Barrier();`	Wait for all other nodes in the network process to make this call, then proceed with execution.

Table 2.2: Selected functions of the GLUnix API.

it performs a lookup in its process database for any sequential GLUnix processes which were executing on that node, marks them as killed, and notifies the appropriate startup processes. If any parallel GLUnix programs had one of their processes on that node, GLUnix additionally sends a `SIGKILL` to the other processes of those programs. The master then removes the failed node from its node database and continues normal operation.

Similarly, when the master detects that a startup process has died in response to a node crash or a `SIGKILL`, the master sends a `SIGKILL` to the remote program associated with that startup and removes the process from its process database. Alternatively, the system need not kill the entire parallel program whenever a single process in it dies. Although not currently implemented, the master could notify the remaining processes of the failure, using a mechanism such as Scheduler Activations [ABLL92], and continue normal operation.

Currently the system does not provide any support for check-pointing or restarting applications, though applications are free to use check-pointing packages such as Libckpt [PBKL95] if necessary.

2.4.8 GLUnix Interfaces

Accessing GLUnix services is done through either the GLUnix API (for programmers) or through command-line utilities (for users of the cluster). Table 2.2 shows a portion of the GLUnix API. Initializing GLUnix services is done via `Glib_Initialize()`; this function creates sockets, locates the per-cluster master and per-node daemon processes, and prepares to run a program. `Glib_Spawn()` is the basic function to run a program under GLUnix. `Glib_CoSpawn()` is used to place an application on the same nodes as a currently

glurun	Run sequential or parallel applications on the cluster.
glukill	Send a signal to a GLUnix job.
glumake	Parallel version of GNU make facility.
glush	Modified `tcsh` to run applications remotely without requiring an explicit `glurun`.
glups	Query currently running jobs.
glustat	Query current status of all NOW machines.

Table 2.3: The GLUnix tools provided to users. Each of these utilities is linked with the GLUnix library to provide a service.

running application; it is used by the Mantis [LC96] parallel debugger. `Glib_GetMyNpid()` and `Glib_GetMyVNN()` return the NPID and VNN, respectively, of the requesting application. `Glib_Signal()` is used to send a signal to one or all of the VNNs of a program running under GLUnix. The `Glib_Barrier()` function is used to synchronize parallel applications.

To interact with GLUnix and to manipulate NPIDs and VNNs, we created a set of command-line tools analogous to traditional UNIX counterparts. The tools and their functionality are described in Table 2.3. The `glurun` utility allows unmodified applications to be run under GLUnix; `glurun` distributes jobs across one or more nodes using GLUnix to select the least loaded nodes in the cluster. The `glukill` utility enables users to signal their jobs from any node in the cluster. The `glumake` utility enables users to distribute compilations across the entire cluster, often speeding up compilation by an order of magnitude or more; `glumake` quickly became the most popular GLUnix tool. `glush` is a modified `tcsh` shell which uses `Glib_Spawn()` instead of `exec()` to run programs on the cluster. `glups` and `glustat` are used to query cluster status.

2.5 GLUnix Usage

In order to identify the limitations of a cluster system as seen in practice, it was important that GLUnix support actual users. GLUnix has been running and in daily use by the Berkeley Computer Science Division since September 1995. While initially running on 35 SPARCStation 10's and 20's, it has been running daily on 100 UltraSPARCs since roughly June of 1996. As a testament to its user-level construction, GLUnix runs on Solaris versions 2.3–2.6 and ports have been made to NetBSD and to Linux for the Intel $x86$ architecture family. As of January, 1998, GLUnix has had over 160 users during the course of its lifetime and has run 1.7 million jobs. GLUnix has been in daily use by both production users as well as system developers; Figure 2.3 shows the number of GLUnix jobs per week since October, 1995 while Figure 2.4 shows a histogram of the parallel degrees of jobs run under GLUnix. Monitoring statistics taken over the course of 100 days indicate that GLUnix was operational over 97% of that time.

User feedback concerning GLUnix has been positive. GLUnix has enabled or assisted systems research and program development in a number of areas. GLUnix has been used by the graduate Parallel Architecture and Advanced Architecture classes and by other research departments here at Berkeley. Implementations of the parallel programming lan-

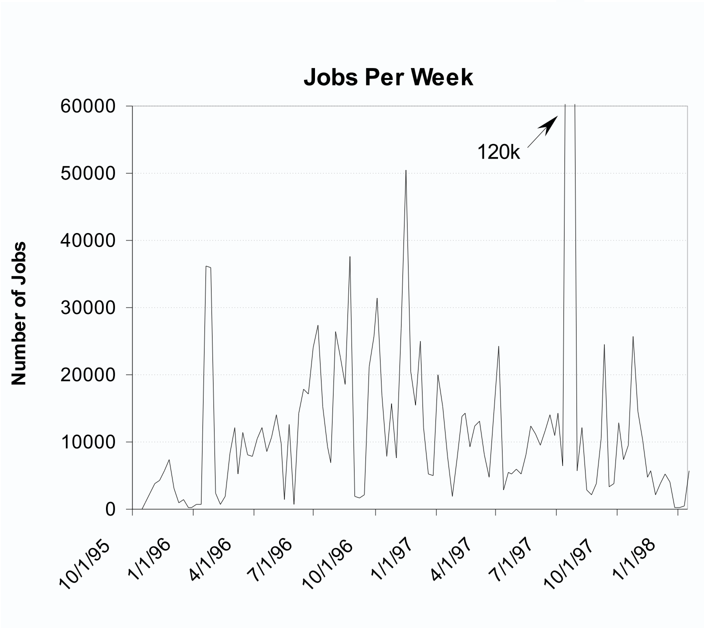


Figure 2.3: Jobs run per week from October, 1995 through January, 1998.

guage Split-C [CDG+93] and the Message Passing Interface (MPI) [The93] use the GLUnix library routines to run on the Berkeley NOW. A number of interesting parallel programs have been implemented using these facilities, including: NOW-Sort [ADADC+97], which set a record for the world's fastest sort time (8.4 GB in under one minute), p-murphi [DDHY92], a parallel version of the popular protocol verification tool, and a 32K by 32K LINPACK implementation that achieved 10.1 GFlops on 100 UltraSPARC-I processors, placing the Berkeley NOW 345th on the list of the world's 500 fastest supercomputers at the time[7].

The largest use of GLUnix and the Berkeley NOW cluster has been as a compute server for distilling thumbnail images off of the web[8]. The second most common use of the cluster has been as a parallel compilation server via the `glumake` utility. The cluster has been used to achieve a world record in disk-to-disk sorting [ADADC+97], for simulations of advanced log structured file system research [MRC+97], two-dimensional particle in cell plasma simulation [VLG95], three-dimensional fast Fourier transforms, and genetic inference algorithms.

GLUnix was also found to be extremely useful for testing and system administration. Initially, parallel stress testing of the Myrinet network and Active Messages was accomplished using scripts built around `rsh`. As GLUnix became stable and was put into daily use, the `rsh` scripts were quickly abandoned in favor of the increased performance and job control facilities of GLUnix. From that point, parallel testing of cluster components proceeded at a much faster pace. With respect to system administration, a large fraction

[7] See `http://netlib.cs.utk.edu/benchmark/top500.html`.

[8] See `http://http.cs.berkeley.edu/~eanders/pictures/index.html`

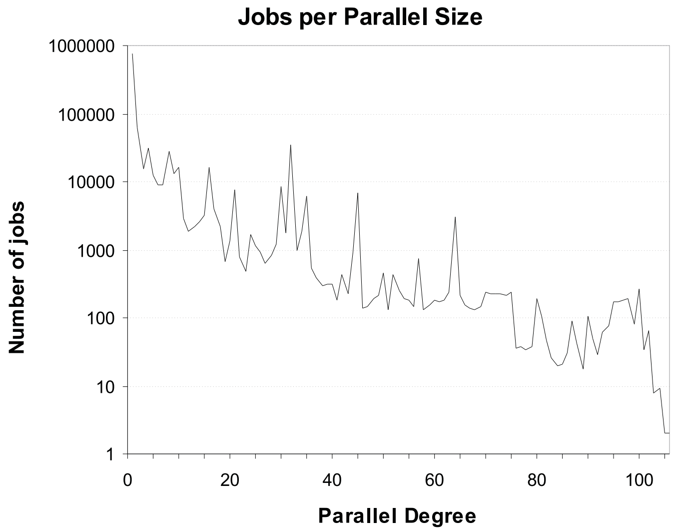


Figure 2.4: Histogram of the parallel degree of the jobs run under GLUnix.

of the parallel job requests under GLUnix have been for programs such as `ls`, `rm`, and `who`, which are not traditional parallel programs. System administrators and developers use GLUnix to run these programs simultaneously on all nodes in the cluster to carry out various administrative tasks.

The GLUnix batch facility (see the section on "Social Considerations, Batch Jobs") has been used for managing very large numbers of short and medium simulation runs, such as the thousands of simulations needed during the investigation of implicit coscheduling [DAC96].

2.6 Evaluation

To effectively harness cluster computing power and provide users with a usable cluster system, GLUnix must be able to execute remote jobs, and in particular parallel jobs, efficiently. This section evaluates the scalability and performance of GLUnix, describing some of the subtle performance tuning problems that were encountered. Then, based on our experiences in building and tuning the system, we evaluate the original architectural decisions made during the design of GLUnix.

2.6.1 Performance

This section evaluates the performance and scalability of GLUnix and relates some of our experiences in performance tuning. All of the data presented in this section are measured with GLUnix running on 167MHz Sun UltraSPARCs running Solaris 2.5.1. Executed binaries are stored on a standard Auspex NFS server which is shared by all machines. Although the cluster has a 1.28 Gb/s Myrinet network which is used by parallel programs running under GLUnix, the Active Messages networking software has only recently supported the client/server model needed by GLUnix, and we have not yet modified GLUnix to use it. Consequently, all measurements presented here are for GLUnix using BSD sockets over TCP/IP on a 10 Mb/sec switched Ethernet.

Figure 2.5 and Table 2.4 identify the various components of the remote execution latency of GLUnix for starting a sequential job. This breakdown was generated by inserting `gettimeofday()` calls at strategic points in the code and averaging the times over 50 runs. The `gettimeofday()` system call has a resolution of less than 1 μsec and an overhead of approximately 1 μsec, causing negligible perturbation of the system's performance. Segment A, "get master hostname", represents the time required to read the master's hostname from an NFS-mounted filesystem; while the master hostname should be cached by NFS, monitoring the network traffic reveals that two NFS transactions were occurring during this process. Segment C, "ping master", was not part of the original GLUnix design, but was added later in response to user feedback. When a program first initializes the GLUnix library by calling `Glib_Initialize()`, the library sends a ping message to the master to check if GLUnix is running. Without this feature a program using the GLUnix library would not be informed that GLUnix was down until some subsequent library call. The ping's cost includes the setup of a TCP connection between the startup process and the master as well as the exchange of GLUnix-internal connection information. Segment D, "`getcwd()`", represents the time required for the daemon to change to the NFS-mounted current working directory of the startup process; network traces revealed that this operation was generating two NFS RPCs to the file server. Segments E and G represent the time spent in the TCP/IP protocol stacks as well as physical link transmission time; the remote execution message transmitted was 2.2 kB. Segment I includes both the time to setup a master `tty` and to make three I/O connections from the remote child to the startup process (a separate connection is made for `stdin`, `stdout`, and `stderr`). Section J shows the cost of creating the slave `tty` and attaching the child process to it.

Figure 2.5 and Table 2.4 shows that the dominant factor in remote execution latency is the `tty` support which accounts for 45.8 msec, or 43.1%, of the total time. The second largest factor is the network file system which accounts for 27.2 msec, or 25.6%, of the total time. Neither of these factors occur in the master. Very little of a sequential program's latency is due to the master. The total time spent in the master is 2.4 msec, or only 2.2%, of the total time.

2.6.2 Scalability

One of the main concerns with a centralized master architecture is that the master will become a performance bottleneck. The upper line in Figure 2.6 shows the total elapsed wall-clock time for running a null program of different parallel degrees under GLUnix[9]. These times were measured using `tcsh`'s built-in `time` command and averaged over 20 consecutive runs. To eliminate file cache effects, each run of the same parallel degree was sent to the same set of nodes and the caches were warmed with an initial untimed run.

[9] Note that Figure 2.5 only presents a breakdown of the time required to start a remote job once the `Glib_Spawn()` library routine is invoked; in contrast, Figure 2.6 depicts the total end-to-end time required to load, start, and shut-down a remote job.

The size of the remote execution message sent to each daemon was 2.2 kB. GLUnix can run a sequential process in 0.24 sec and a 95-node parallel program in 1.3 seconds. For comparison, an `rsh` of a null application under similar conditions takes approximately 2.2 seconds to a single remote node, and over 90 seconds to 95 remote nodes.

The lower line in Figure 2.6 represents the time required just to establish the I/O connections from the parallel processes to the GLUnix startup process. These times were measured using a microbenchmark comprised of a single server with multiple clients, each client establishing three TCP/IP socket connections to the server. The slopes of each line are approximately the same, indicating that I/O connection establishment to a startup process is the primary scalability bottleneck for remote execution. Figure 2.5 identifies some of the costs which account for the gap between the two lines.

To illustrate more directly the scalability of the centralized master, independent of network stack overhead or link contention, Figure 2.7 presents the computation time spent in the master when running jobs of different degrees of parallelism (the conditions were similar to Figure 2.6). When running parallel applications, the time spent in the master increases by an average of 220 μsec/node. In contrast, Figure 2.8 graphs the time spent in the network stacks and on the physical network for the same test. The switched Ethernet begins to show serious contention after a certain threshold. Running `netstat` on the master node indicates a significant increase in collisions of outgoing Ethernet packets during remote execution. Remote execution messages in GLUnix are sent from the master to each daemon which will run a parallel job. Each of these daemons generate a reply message, indicating success or failure. When performing remote execution to many nodes, the replies from earlier daemons collide on the Ethernet with the remote execution requests sent to later daemons.

Figure 2.9 evaluates the skew when coscheduling parallel programs. Coscheduling in GLUnix does not generate replies from the daemons and hence will not experience the Ethernet collisions discussed above. The coscheduling skew graphed is the difference in time between when the first and last processes of a parallel jobs are sent a scheduling signal. The skew was measured by taking a timestamp in each daemon just before sending the `SIGSTOP` or `SIGCONT` signals to a parallel application. These measurements presume that the signal delivery and rescheduling times through each node's local operating system are statistically equivalent for different parallel degrees, a reasonable assumption given that each operating system performs the actions independently. The times measured include network stack overheads in the master and daemon as well as physical link transmission times. The coscheduling time slice used for these tests was one second.

The scalability data here reveal two points. First, the processing time required by a centralized master to handle remote execution is relatively low, at only 220 μsec/node. Additionally, one-to-many communication patterns also scale relatively well, such as coscheduling in Figure 2.9 at 203 μsec/node. Second, many-to-one communication patterns using TCP/IP over the switched Ethernet exhibit poor scalability, as demonstrated by Figures 2.6 and 2.8. Many of these network contention effects can be alleviated by modifying the GLUnix `Comm` module to use Active Messages [MC96] across the Myrinet [BCF$^+$95].

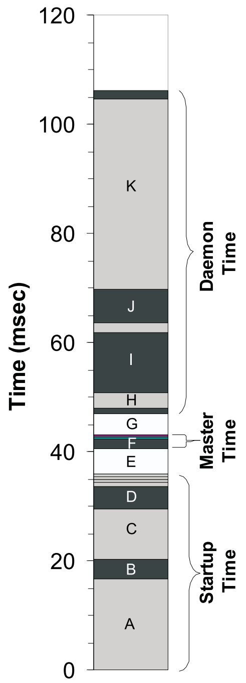


Figure 2.5: An analysis of the time required to start a null sequential program remotely under GLUnix. The labels correspond to the rows in Table 2.4. The total time depicted is 106.0 msec. Only the components which take 1.0 msec or more are labeled. The unlabeled segments amount to 3.8 msec, or 3.6%, of the time. The time spent in the master on non-network related activities is 0.4 msec. The final `exec()` call performed by the daemon is not depicted in this figure.

Graph Label	**Time**	**Description**
A	16.7 msec	get master hostname over NFS
B	3.6 msec	misc initialization
C	9.0 msec	ping the master
D	4.3 msec	`getcwd()`
E	4.7 msec	network latency
F	1.7 msec	master receives message
G	3.9 msec	network latency
H	3.0 msec	misc initialization
I	11.0 msec	setup `tty` master, connect to daemon
–	1.6 msec	`fork()`
J	6.2 msec	`chdir()`
K	34.8 msec	setup `tty` slave

Table 2.4: Time spent on various activities during remote execution. The graph labels correspond to the segments of Figure 2.5. Times are averaged over 50 runs.

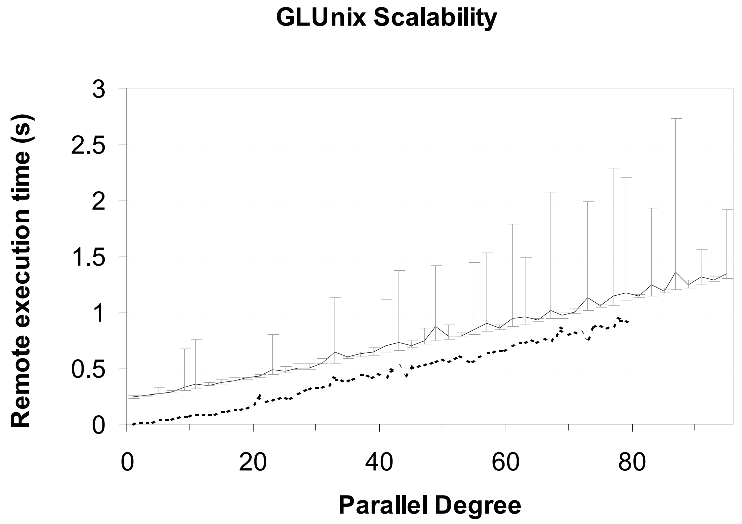


Figure 2.6: The upper line is the end-to-end time taken to run parallel programs of various sizes. Each data point is averaged over 20 runs. The error bars indicate the minimum and maximum time of those 20 runs, while the solid line graphs the average time. The lower line is the time taken by a microbenchmark to simply establish I/O connections to the startup process. The slope of both lines is 12 msec/node.

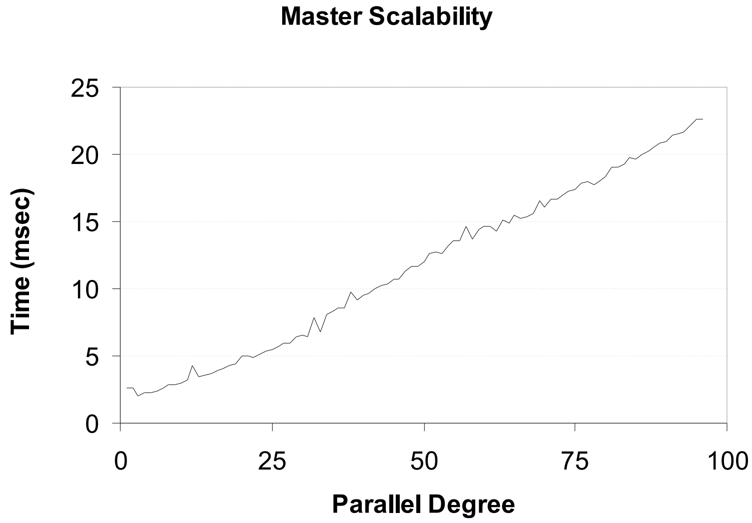


Figure 2.7: This graph measures the processing time spent in the master when handling execution requests for programs of various parallel degrees. This time does not include any time spent in the startup or daemon processes or in the network. The slope of the line is 220 μsec/node.

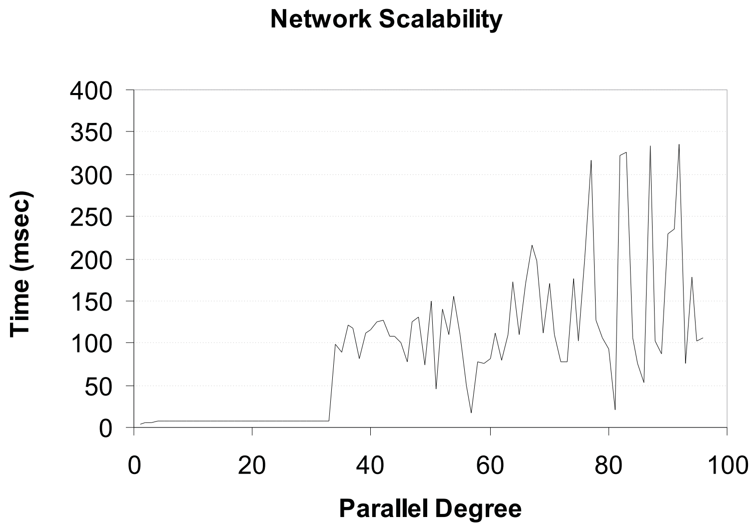


Figure 2.8: This graph shows the delay between when the GLUnix master begins sending the execution request messages and when the last request message arrives at a daemon. The time includes network stack delays on both sender and receiver as well as network transmission time. Measurements are taken on a 10 Mb/sec switched Ethernet. Each data point is averaged over 20 runs.

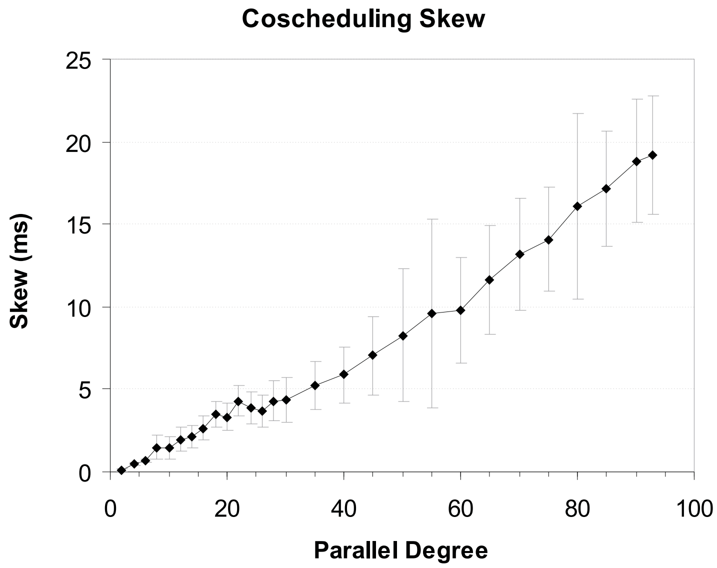


Figure 2.9: This graph measures the skew arising from imperfect coscheduling in GLUnix. The skew is defined as the difference between the earliest and latest times that two processes in a parallel program are started or stopped. Each data point represents the skew for a particular parallel degree averaged over 200 coscheduling events. The error bars indicate one standard deviation above and below the average. The slope of the line is 203 μsec/node.

2.6.3 Performance Tuning

In this section, we will describe some of our experiences with improving the performance of remote execution. These changes were driven by the needs of the parallel NOW-Sort application [ADADC+97], which set a record for the fastest disk-to-disk sort. To set the record, NOW-Sort had to sort as much data as possible in 60 seconds. Unfortunately, initial runs revealed that it took more than 60 seconds simply to start the program on 100 nodes.

Investigation revealed that running a parallel program with a large degree of parallelism was flooding the local NIS server during startup. When executing a program on a remote node, the GLUnix daemon must initialize the UNIX group membership for the remote program. Initially this was done using the `initgroups()` command, which accesses the cluster's NIS server to retrieve the requested group information. Running a parallel program on 100 nodes thus flooded the NIS server with 100 network connection requests at roughly the same time. It was found that the first few requests were processed quickly, but that others were quite slow, taking nearly one second each to complete. Further investigation revealed that the `getgroups()/setgroups()` system calls provide similar functionality without the NIS lookup. GLUnix was modified to use the new system calls.

After making the above change, we were quite surprised to discover that remote execution performance was actually worse. Additional measurements indicated that establishment of the I/O connections to the GLUnix startup process had become slower. Each process in the parallel program was making three I/O connections to the startup process. However, the connection queue length in Solaris 2.5.1 defaults to 32. Thus, the first few connections completed quickly, but when the queue was overrun, the connection requests had to timeout and retry. Optimized for the wide area, the TCP timeouts are coarse-grained [PD96], causing connection retries to proceed very slowly once the queue is overrun. To resolve this, we used a Solaris administration command to set the TCP driver's accept queue length to its maximum value of 1024. This improved performance dramatically. We further concluded that the NIS requests had originally staggered the execution of the GLUnix daemons, effectively serializing the I/O connections to the startup and reducing the probability of over-running the accept queue.

2.6.4 Architecture Evaluation

Implementing, debugging, tuning, maintaining GLUnix for over two years has given us valuable experience with a user-level, centralized, event-driven system. This section provides a brief evaluation of the major design decisions of this experience.

The decision to build GLUnix as a user-level layer was a crucial one. Even with state-of-the-art development and debugging tools, development and testing continued for nearly two years before the system was functioning acceptably. Given our time and resource constraints, had we attempted to implement such functionality in the kernel, it is unclear if the project would have ever reached sufficient stability to be able to support a real user community. The user-level construction has enabled GLUnix to easily track improvements in the underlying operating systems, Solaris—while development initially began on Solaris 2.3, GLUnix has run on versions 2.3–2.6 with approximately five lines of source code change.

Had we implemented GLUnix in the kernel, the changes would have likely been much more pervasive. GLUnix also runs on SPARC 10, 20, and UltraSparc workstations as well as the Sun Enterprise 5000; versions of GLUnix have also been ported to Linux and NetBSD in less than two weeks in each instance, an achievement which would not have been so easily accomplished had GLUnix been implemented in the kernel. However, the user-level construction of GLUnix also restricts its functionality in a number of areas described in the next chapter.

With respect to the decision to implement a centralized GLUnix master, measurements of the implementation indicate that the centralized master is not the performance bottleneck of the system. The master requires only 220 μsec of processing per node to start a parallel program. Thus the master would require less than a quarter of a second to process a 1000-node execution request. Current scalability bottlenecks of the system are BSD sockets over TCP/IP and the 10 Mb/sec switched Ethernet used by GLUnix. Performance improvements can be expected by porting GLUnix to use higher performance communication layers such as Active Messages [MC96] and faster networks such as the 1.28 Gb/s Myrinet [BCF$^+$95].

As development of GLUnix proceeded, maintaining all event handlers as non-blocking routines became increasingly frustrating and considerably reduced code readability, maintainability, and correctness. Distributed GLUnix components often interact in a remote procedure call (RPC) [BN84] like fashion, issuing requests and needing replies. To maintain system responsiveness, GLUnix must return to the main event loop after issuing a request to a remote node. This requires the programmer to manually encapsulate and store all local event state before returning to the event loop, only to manually restore that same state when the reply arrives. The multi-threaded programming model maps more naturally onto RPCs—threads may issue RPC requests and block awaiting replies; local state is automatically encapsulated on the thread's stack, requiring no additional effort from the programmer to save and restore it. Despite strong arguments in favor of event-driven software architectures [Ous96], given our experience and the increased availability of multi-threading support, today we would implement GLUnix as a multi-threaded program.

2.7 Related Work

While a number of systems provide portions of the functionality of GLUnix, to our knowledge, no single system supports all of our desired functionality for a NOW operating system: transparent remote execution that is compatible with existing application binaries, a single system image, load balancing, and integrated support for parallel and sequential jobs. In this section we discuss projects related to GLUnix, highlighting the differences between previous systems and GLUnix.

2.7.1 Kernel-level Solutions

While GLUnix takes a user-level approach to providing cluster resources, achieving portability but suffering from a number of user-level limitations, a number of previous efforts to provide cluster functionality have used in-kernel approaches to maximize the transparency and robustness of cluster functionality.

Hydra [WCC+74] was an operating system designed for a NUMA architecture and run on a cluster of PDP-11 mini-computers [BCM+70] in the C.mmp system [WB72]. Hydra was designed to provide a set of basic mechanisms from which an arbitrary set of operating system facilities and policies could be conveniently, flexibly, efficiently, and reliably constructed. Hydra acted as a "meta-environment", implementing a kernel or nucleus set of features that were universally applicable and absolutely reliable.

Locus [PWC+81] was a UNIX-compatible operating system for Vax/750s built around an highly available, distributed network-wide file system supporting automated replication in a heterogeneous environment. It supported cluster-wide naming for file and processes, providing transparent remote execution with a single system image. A derivative system, Transparent Network Computing (TNC), provides a subset of this functionality through modifications to existing UNIX kernels. Introducing a virtual process layer (called `vproc`) into the system enables the kernel to support location-independent control of processes. In contrast to GLUnix, Locus was a kernel-level solution and did not provide support for parallel applications.

Newcastle Connection [BMR82] provided a distributed file system constructed using interposition on a UNIX operating system running on a cluster of PDP11s. It required only minor changes to the underlying operating system to intercept file accesses on entry to the kernel, forwarding them to remote nodes to provide a transparent global file system. Other issues such as remote execution, migration, distributed process relationships, and support for parallel programs were not addressed.

VaxClusters [KLS86] provided a highly available single system image for the VMS operating system. It implemented a low level distributed lock manager to name and synchronize access to resources shared across the cluster, such as devices, files, print services, and user environments. VaxClusters provided login-time remote execution with load balancing.

Mach [ABB+86, Ras86] built upon the Hydra ideas of separation of policy and mechanism to demonstrate the "microkernel", which became the basis for a number of commercial operating systems including Windows NT. In a microkernel, the operating system is separated into two major components: the microkernel and one or more user-level operating system servers. The microkernel provides abstractions for low-level hardware resources such as the processor, memory, and devices and provides protected interprocess communication. Higher-level operating system features such as file systems and signals are provided in user-level operating system servers. Support is included for user-level pagers. However, by placing OS functionality in a user-level server, this architecture requires two address space context switches for an application to access the operating system. Studies demonstrating the performance limitations of this architecture [CB93] have led to efforts to place the user-level server into the microkernel's address space to improve performance [LHFL93].

NEST [AE87] modified a standard UNIX operating system to provide a fully distributed, load balancing, transparent remote execution system capable of tolerating node failures. To support location-independent remote execution, all file accesses were forwarded back to the home node for execution. The system also supported distributed process groups, but not migration. A NEST daemon running on each node in the cluster monitored the load on its node, communicating that information to other nodes through an advertisement/subscription interface.

Both V [Che88] and Sprite [OCD+88] were research operating systems written from scratch, providing in-kernel support for remote execution and dynamic process migration. V provides transparent remote execution by leveraging its network-transparent interprocess communication primitives and global process identifier space, while Sprite provided transparent remote execution by transparently routing all interactions with the remote process through the home node. Both systems were successful at providing remote execution and process migration. In-kernel process migration, however, came at an admittedly high implementation cost in the Sprite case [DO91]. Both systems provided location-independent file system access using a centralized, cache-coherent file system and diskless clients.

Plan 9 [PPTT91] supported user-initiated, login-time remote execution in a network of diskless terminals and compute servers. Many aspects of the user's environment were recreated on the remote node to provide a level of transparency. Remotely running applications were given access to home node devices including the mouse, screen, and keyboard. The windowing system used a form of interposition to intercept all accesses to the mouse and keyboard devices to demultiplex terminal I/O only to the active window.

The Hive [CRD+95] project modified Irix to improve the failure characteristics of distributed shared memory multiprocessors by enabling them to tolerate certain hardware failures (called *fault-containment*). Instead of having a single instance of the operating system manage the entire multiprocessor, the machine was divided into cells with a separate instance of the operating system presiding over each cell. Hive modified Irix to extend the UNIX process abstraction to span cell boundaries, enabling large parallel programs to use the entire machine. When running a parallel application, each cell's operating system ran a separate process for the application which encapsulated the threads executing in its cell; the address spaces of each process were kept consistent to maintain the shared memory abstraction. This same mechanism also supported migration of sequential processes between cells for dynamic load balancing.

Solaris MC [KBM+96] was a cluster operating system designed to provide a single system image for a cluster. It was implemented by modifying the Solaris kernel to extend UNIX abstractions to the cluster using a CORBA-compliant object-oriented model to provide cluster services; for example, standard file system accesses were converted by Solaris MC into distributed object invocations in the CORBA model, providing the illusion of a single, unified file system for the cluster. Solaris MC implemented a `vproc` layer similar to TNC to extend process management to the cluster. While Solaris MC did provide execution-time remote execution, in order to contain implementation complexity, it did not do dynamic process migration [Shi98] or provide support for parallel applications.

The OSF/1 AD TNC (Advanced Development, Transparent Network Computing) system [ZRB+93] was based on the Mach microkernel and the TNC derivative of Locus. Modifications were made to Mach and to the OSF/1 UNIX server to provide a single system image for a cluster, including a distributed file system, distributed sockets, and enhanced process management using a `vproc` layer. It provided minimal support for parallelism through the ability to fork a job on multiple nodes using a binary tree distribution. Transparent remote execution was supported using the `vproc` interface of TNC which provided a cluster-wide process identifier space and a location-transparent interface to process operations. The system also provided transparent remote execution and access to remote devices.

Because the above systems were implemented by replacing or heavily modifying the operating system kernel, they required significant porting effort to keep abreast of new hardware architectures. Consequently, despite the advanced cluster features they provided, all but the most recent of them are not widely available or in general use today. In contrast, GLUnix can be easily ported to new platforms and tracks advances in the underlying operating system with negligible effort.

2.7.2 User-level Solutions

A number of projects have attempted to provide cluster facilities at the user level. Condor [Lit87, BLL91, LS92] provides users with checkpointing and migration for batch jobs. Users must relink their applications with the Condor library which then interacts with daemons throughout the cluster to migrate the job to idle workstations. A common file system is not necessary, as the Condor library reroutes all file accesses (as well as other system calls) back to the home node. There are, however, a number of restrictions placed on applications using Condor. Only single process jobs are supported; parallel applications or applications which spawn children are not supported. Applications can also not make use of signals, timers, shared libraries, or interprocess communication (including terminal I/O). Finally, file operations must be idempotent—the application can either read or write from a file, but not both.

Libckpt [PBKL95] is an application library providing checkpointing. Similar to Condor, to use Libckpt, applications must be relinked with the library and are restricted from using certain functionality; Libckpt can cause a "correct but ill-behaved application to fail or produce incorrect output on recovery". Libckpt requires one line of source code change to applications which must also be linked with the checkpointing library, prohibiting transparent use with existing binaries. Libckpt supports a number of advanced checkpointing features; for example, incremental checkpointing is supported by mapping out memory to do copy-on-write checkpointing. However, when checkpoints are restarted, a number of elements of the system state for the process are not transparently maintained, including the process identifier, process relationships, signal dispositions, timers, shared file descriptors, or interprocess communication channels. To maximize portability, the only system state that is restored are open files. Libckpt has been ported to a variety of architectures and requires no kernel modifications.

PVM (Parallel Virtual Machine) [Sun90] provides support for collections of relatively independent programs communicating at a coarse granularity across wide-area, heterogeneous clusters. PVM was not designed to provide the same level of node coupling that GLUnix does or to provide a single system image—instances of PVM are manually created by each user, preventing different users from seeing or interacting with each other's jobs. PVM does not attempt to provide any transparency to remote execution—interaction with PVM is done through a special shell which does not recreate any of the user's execution environment on the remote node. PVM does not do global resource management, provide transparent job control, or support standard terminal I/O. Consequently, GLUnix provides a higher degree of transparent remote execution and a closer approximation of an SSI than does PVM. However, to improve fault tolerance, PVM is fully distributed, maintaining global knowledge in all daemons, whereas GLUnix has a centralized master and therefore a single point of failure.

LSF (Load Sharing Facility) [Zho92] provides command line integration with a load balancing and queuing facility. Like GLUnix, LSF recreates a substantial portion of the user's runtime environment for remote processes, including current working directory, file creation mask, and resource limits. LSF also is very similar to GLUnix in that it uses a startup process to redirect signals and I/O to remote jobs to improve transparency of remote execution. LSF itself does not provide support for parallel jobs, although versions of LSF have been integrated with PVM to provide support for coarse-grained parallelism. LSF allows users to define application resource requirements in terms of both static and dynamic machine characteristics (e.g., machine architecture and available memory), while GLUnix is only currently capable of supporting static machine characteristic requirements via the naming and reservation system. GLUnix, however, provides a level of SSI through a cluster-wide, globally-unique NPID space managed by the master, enabling job control and querying to be done from any node in the system and providing the virtualization necessary to implement process migration. In contrast, only LSF batch jobs can be manipulated from any node in the system.

The User File Organizer (Ufo) [AISS97] leveraged the `/proc` [FG91] file system of Solaris to provide a user's remote applications with access to files located at a home location without requiring users to have `root` permission on the remote node. Although unable to work for `setuid` programs (a limitation of `/proc` when not running as `root`), the system is able to provide transparent support for other binary applications. Ufo translates intercepted file accesses into FTP or HTTP requests which are sent to the home node. Performance measurements indicate that wide-area file transfer dwarfs the high cost of intercepting events and copying data through the `/proc` interface. Ufo does not address other issues of remote execution including load balancing, maintaining aspects of an application's environment other than files, single system image, or parallel jobs support.

Chapter 3

GLUnix Limitations

GLUnix, like many packages attempting to provide cluster services, is built at the user level to simplify implementation complexity, to leverage advances in the underlying hardware and operating system, and to maximize portability to other platforms. However, the decision to implement GLUnix at the user level had a major impact on the ability of GLUnix to efficiently and transparently execute programs remotely. This chapter presents a number of limitations arising from the user-level construction of GLUnix. While a number of limitations are presented here, later chapters will focus on the first four limitations as a representative sample: lost input, group signals, remote device access, and scalability. These limitations provide the motivation for the operating system extension technology introduced next in Chapter 4. Chapter 5 will demonstrate how that extension technology can enable GLUnix to overcome these four limitations. For the sake of completeness, we also describe a number of other limitations encountered. All of these limitations relate to the fact that some UNIX abstractions and operations cannot be properly intercepted at the user level.

3.1 Lost Input

Performing remote execution in a transparent fashion requires sending input from the user's terminal to the remotely executing job. Maintaining UNIX semantics requires that input be queued up at various points in the system and be delivered to a job only when that job performs a read operation on a file descriptor associated with the input stream. When a job reads only a portion of the available input and then exits, unread input should remain queued for the next job which reads from the input stream.

One common example of this behavior occurs when a user executes a job and, while it is still running, types future commands to the shell which will be executed when the current job terminates. A second example occurs when an input stream is piped to a shell script; each command consumes a portion of the input, leaving the remainder for future commands.

Unfortunately, remote execution of user jobs cannot efficiently and transparently maintain these semantics using standard user-level functionality. When a remote job attempts to read N bytes of input, exactly N bytes should be transfered from the home node to the remote node for consumption by the remote job. Determining the proper amount of input to send the remote node is problematic. Consequently, the GLUnix startup process forwards all available input to remote processes as it becomes available. A number of popular UNIX utilities routinely drop input because of this difficulty, including `rsh`, `ssh`, `telnet`, and `rlogin`. Because these tools do not know how many bytes of input, if any, the remote processes may be requesting, they send all available input to the remote node and suffer from the same problem. When the remote connection terminates, any input not consumed by the remote processes is lost, compromising the transparency of remote execution.

3.1.1 User-level Approaches

To transfer the precise amount of input from the home node to the remote node, the system must determine how many characters are being requested in the `read()` system call. Ascertaining this in a user-level system can be done either by modifying the C library to intercept the `read()` library call (which in turn issues a `read()` system call trap to the kernel) or by using a `/proc` style facility to intercept the `read()` system call and inspect the parameters. We will examine both of these alternatives in turn.

By replacing the standard C library with a modified one, GLUnix would be able to intercept invocations of the `read()` library call (which in turn invokes the `read()` system call) and determine the correct number of bytes requested by inspecting the arguments given to the procedure. The system could then send a message to the home node requesting the correct amount of input. For the problem of lost input, modifying the C library has a number of limitations in the realms of transparency, composability, distributability, and increased complexity.

In order to use a modified C library, statically linked applications would have to be relinked with the new library, violating the binary compatibility goals of GLUnix. For applications that are dynamically linked, the shared C library could be replaced with the modified one which would be automatically linked with each application at the time of execution. However, only invocations of the original `read()` library routine will be caught. The modified C library routines will be overshadowed by any other application or library functions which attempt to augment `read()` functionality by redefining the library function and issuing system calls via assembly traps. Direct invocations of the `read()` system call which do not go through the library will not be intercepted and proper behavior will not be emulated. Intercepting these assembly traps can be done using binary rewriting techniques such as Software Fault Isolation [WLAG93] and EEL [LS95], but these techniques cannot currently be applied to arbitrary application binaries with acceptable performance.

A second limitation is that the method of modifying the C library to add new functionality does not generally compose with functionality developed by other parties. For example, if the dynamic C library were modified to solve the problem of lost input, the application could not also take advantage of other, independent modifications to the C library without manually merging the two modifications into a single library. Since the C libraries of most commercial operating systems are proprietary source code, the code for modified commercial C libraries cannot be arbitrarily distributed to other developers, hindering their ability to leverage existing implementations. Modified commercial C libraries can generally only be distributed in binary form.

Third, modifying a C library to include this functionality involves non-trivial complexity relating to the `exec()` system call. For efficiency, the system should only forward read operations of `stdin` to the home node; other read operations should be performed locally. Consequently, since here is no way from the user level to reliably determine if an open file descriptor is bound to the original `stdin`, the modified C library must keep track of all open file descriptors which are associated with `stdin`. When an `exec()` happens, the address space of the original process is overwritten with the new process and all data structures stored in the process are lost. In order to maintain information about file descriptors across `exec()` calls, that information must first be written out to some other storage location, such as disk or a separate process, and then read in again after the `exec()` has completed.

In addition to these difficulties, modifying the C library also prevents certain performance optimizations from being safely implemented. Intercepting each `read()` system call and sending a message to a remote node requesting N bytes of input requires a round-trip delay across the network for each block of input read. A more efficient solution is to optimistically send all input to the remote node and then recover the unused input when the remote process terminates. While this would increase the performance of standard input considerably, performing this optimization in a modified C library suffers from problems of safety and efficiency. The unused input that is returned to the home node from the remote node can be considered a shared resource between potentially competing applications. A faulty or malicious program running remotely may, for example, *insert* dangerous commands into the input stream which will be returned to the home node and then unwittingly read and executed by the user's shell after the remote program terminates.

Performing this optimization in a modified C library is also complicated considerably by the `fork()` system call. As the remote process creates multiple children using `fork()`, there are now multiple copies of the modified C library executing concurrently. As input is optimistically forwarded to the remote node, these modified C libraries will be required to coordinate with each other in order to keep input in a sequential ordering. If the process requesting the data is different from the process caching the data, the input will have to be transferred across process boundaries. This violates the UNIX security model and enables a man-in-the-middle attack as a non-root process could pass arbitrary data to a root process or inspect data flowing from the user's terminal to a root process. The synchronizations and interprocess copies would also somewhat negate the intended benefits of the optimization.

Using facilities like the System V `/proc` pseudo-file system [FG91] would enable a system like GLUnix to reliably intercept `read()` system calls from an application and determine the number of bytes read. Current `/proc` systems, however, cannot selectively catch `read()` calls to `stdin` versus other streams, requiring *all* `read()` system calls to be caught. Furthermore, `/proc` systems require a separate user-level process to handle intercepted system calls—when a system call is intercepted, control is passed from the traced application to the server process and then back, requiring multiple address space context switches. Performing these two context switches on every `read()` system call would significantly limit application I/O performance.

A better solution to this problem would be to reliably and efficiently intercept `read()` system calls (not `read()` library calls), caching the data in a single, trusted location. Delivering the input to a single, trusted location would eliminate the need for synchronization to properly order the input and would prevent the man-in-the-middle attack described above. No existing user-level approach provides this set of features in a single solution.

3.2 Group Signals

In order to transparently support remote execution, UNIX signal semantics must also be maintained. In order to treat collections of related processes as a single logical group with respect to signals, UNIX provides an abstraction of *process groups*. Any number of processes may join a particular process group and group membership does not have to follow parent/child process relationships. Signals can be sent to a process group either on a number of occasions: for example, in response to a request from the user via the `kill()` system call or in response to terminal job control characters such as `Ctrl-C` or `Ctrl-Z`.

In GLUnix, to provide transparent remote execution, GLUnix must support the process group abstraction even when the processes comprising the group are physically distributed throughout the cluster. Any signals destined for the distributed process group should be forwarded to the appropriate nodes and delivered to the remote processes. GLUnix currently uses the startup process as a local proxy to catch all signals destined for a remote process, forwarding those signals to remote nodes for delivery. However, performing this operation correctly requires the startup to determine whether the signal was sent to a single process or to an entire process group. In Solaris, this information cannot be determined using user-level facilities. Consequently, the GLUnix daemons cannot know if the signal should be sent only to the remote process or to all the members of the process group on the remote node. Thus, GLUnix cannot transparently maintain signal semantics for remote execution.

There are a variety of examples involving group signals are sent. Group signals may be sent by a user at any time using the `kill` command. When a process group attached to a `tty` is running in the background and attempts to read input from the `tty`, the entire process group will be sent a `SIGTTIN`. When the user has executed a series of commands at a shell separated by pipes, the commands belong to a single process group; when the user types a Ctrl-C or Ctrl-Z, a `SIGINT` or `SIGTSTP` signal, respectively, is sent to all commands in the group.

3.2.1 User-level Approaches

In many versions of UNIX, there is not user-level interface available which can indicate whether a signal was destined for a single process or a process group. While group signals initiated by user-level programs are sent via system calls and can therefore be intercepted by either a modified C library or `/proc`, signals initiated by device drivers within the kernel (e.g., the terminal device driver) do not use the system call interface and hence cannot be properly handled. `/proc` can be used to intercept most signals delivered to a particular application, but it cannot intercept `SIGKILL` nor can it distinguish whether a given signal is sent to only a single process or to a process group. A modified C library would have these problem as well as being incapable of intercepting `SIGSTOP`.

It is possible to derive the group signal information using one or more secondary processes. When GLUnix creates a startup process, it can also create one or more additional startup processes which are placed in the same process group as the startup. Whenever the startup process receives a signal, it communicates with the secondary startup processes to determine if they also received the signal. If so, then the signal received may in fact have been a group signal. It is also possible, however, that the user independently signaled both processes; this may happen, for instance, if some users note the presence of the secondary process and, believing it to be a vestigial process from a previous run, signal it directly. This situation must be prevented in order to be certain that a given signal was indeed destined for a process group. `/proc` or a modified C library can be used to prevent other user applications in the system from signaling the secondary startup process.

This approach, however, doubles the process overhead of GLUnix. A more elegant solution would be to simply intercept the signal in the kernel when group signal information is readily available. Group information could then be passed to GLUnix via the startup process and group semantics would be maintained.

3.3 Remote Device Access

In a system in which remote execution is the norm, it will often happen that a remote process needs to access a device located on the user's home node. For example, a remote application may need to manipulate a `tty` or access a tape backup or CD-ROM device which is physically attached to the user's home node. When the remote process attempts to access such a device, however, it will issue system calls such as `read()`, `write()`, and `ioctl()` on the remote node, where the device is not located. Device accesses from the application must therefore be intercepted on the remote node and forwarded back to the home node for execution. While `read()` and `write()` system calls can be intercepted and sent back to the startup process for execution, the `ioctl()` system call introduces a complication. The `ioctl()` system call is a general-purpose system call designed to handle a wide variety of device control operations. It therefore has no pre-specified semantics. It takes a variable number of arguments, nearly any of which may be pointers to device-specific data structures located anywhere in the process's address space. Simply intercepting and forwarding the `ioctl()` system call back to the home node is insufficient since any user buffers referenced in the `ioctl()` are located on the remote node. Any data which the device must read should be copied to the startup process on the home node from the address space of the remote process. Similarly, any data written by the device should be copied to the address space of the remote process. In modern UNIX operating systems, such memory accesses of a process's address space by the kernel are handled by a special set of kernel copy routines (e.g., `copyin()` and `copyout()` in Solaris and `getuser()` and `putuser()` in Linux).

3.3.1 User-level Approaches

A modified C library would be able to intercepted the `read()`, `write()`, and `ioctl()` system calls, but would not know in general how to interpret the `ioctl()` parameters, and would therefore be unable to identify the appropriate application data buffers to transfer back to and from the home node. While it is certainly possible to hard code the semantics for any fixed set of well-known devices, it is impossible to know *a priori* the `ioctl()` semantics for every conceivable device.

The `/proc` file system is equally ill-suited to this problem. While the `/proc` file system could be used to intercept the system calls, it has the same problem of being incapable of interpreting the semantics of the `ioctl()` system call in any general way.

It is possible, however, to infer a certain amount of information about device accesses using user-level facilities. Using the `mprotect()` system call, and application can map out regions of its address space. To implement this solution, the `ioctl()` command and its arguments are not interpreted by GLUnix, but are simply transferred from the remote process to the startup process. The startup process unmaps its entire address space, issues the `ioctl()` system call captured on the remote node, and intercepts the `SIGSEGV` signals generated by the device when accesses to the application's unmapped address space fail. The startup process can then determine the address of the fault from the `siginfo` structure accompanying the signal delivery, fetch the relevant page from the remote node, map that page into its address space, and reissue the `ioctl()` command; this sequence would be repeated for every page touched by the device. This approach suffers from significant limitations. First, in order for the startup process to issue the `ioctl()` system call and catch the `SIGSEGV` signals, it cannot map out its entire address space; at a bare minimum, it will require a stack frame and code which can issue the `ioctl()` and hold the `SIGSEGV` signal handler. Consequently, device accesses which coincidentally reference these pages will not be caught, producing incorrect results. Second, manipulating address spaces using `mprotect()` introduces overhead. Third, although the `siginfo` structure does indicate the address of the fault, it does not indicate if the faulting instruction was attempting to read or write to the missing page. Consequently, the entire page must be copied from the remote process to the startup process and copied back after completion of the `ioctl()` (although diffs could be used to eliminate the need to copy unmodified pages back to the remote process). Finally, a separate `SIGSEGV` will be generated for each page accessed. While multiple pages can be prefetched from the remote node, the startup process cannot generally know if these pages will be needed by the device, introducing extra overhead in cases where they are not.

A far more elegant solution would be to issue the `ioctl()` system call in the startup process and intercept the device accesses of application memory in the kernel. If information regarding access type (read versus write) and length can be determined, this would allow the exact data needed to be copied over from the remote node to the startup in a single request.

3.4 Limitations on Scalability

In order to maintain UNIX I/O semantics during remote execution, GLUnix must properly transfer `stdin`, `stdout`, and `stderr` between the startup process and the remote children, as described in section 2.4.4. To provide this functionality to existing application binaries, GLUnix establishes TCP/IP pipes between the startup and remote processes, using the `dup()` system call to bind the pipes to the standard I/O file descriptors 0, 1, and 2. The application can then perform standard `read()` and `write()` system calls to the I/O file descriptors. Note that while `stdin` can be multiplexed onto one of the two output streams, in order to properly maintain `stdout` and `stderr` semantics, two distinguishable I/O streams are required. As described in section 2.6.2, these I/O streams limit scalability in two ways. First, as shown in Figure 2.6, accepting all these connections is the limiting factor in starting up parallel jobs in GLUnix. Second, since there are a maximum of 1024 file descriptors in many UNIX operating systems including Solaris, using separate TCP/IP connections for `stdout` and `stderr` limits the maximum size of a GLUnix job to 512 nodes. GLUnix scalability can therefore be doubled by multiplexing `stdout` and `stderr` onto a single I/O stream.

Active Messages provides a light-weight connection establishment abstraction called an endpoint [CMC97]. Porting GLUnix to use Active Messages would dramatically reduce the I/O connection establishment costs and would remove the limit on the number of connections. However, using Active Message endpoints for the I/O connections would require application modification, since endpoints cannot be accessed through native UNIX file descriptors. The FastSockets [RAC97] library provides a way of mapping standard UNIX sockets onto Active Messages, but requires applications to be relinked, which violates the GLUnix goal of binary compatibility.

3.4.1 User-level Approaches

By replacing the standard C library with a modified one, GLUnix would be able to intercept `write()` library calls to `stdout` and `stderr` and could multiplex the two streams onto a single output connection. The combined output streams would be demultiplexed by the GLUnix startup process. However, this approach suffers from all the problems associated with modifying the C library as presented in section 3.1.1: it will only work for those applications which are dynamically linked, do not issue system call traps or otherwise circumvent the `write()` library routine, and do not rely on other, independent modifications to the C library for other functionality. The complexity surrounding the `fork()` system call is somewhat different. Since no caching takes place in the modified library in this case, there is no man-in-the-middle attack introduced—output written by each process is handed directly to the operating system and cannot be covertly inspected by other processes sharing the same output channels. However, a denial-of-service attack does become possible. As `fork()` is used to create multiple children, each of these will inherit the standard I/O file descriptors of the parent process. Each of these children will then attempt to multiplex its output streams onto a common connection to the startup process. This connection is therefore a shared resource and, to avoid corrupting each other's data packets, the libraries will be required to synchronize access to the output stream. A faulty or malicious library which acquires and fails to release the lock protecting the output stream can prevent other processes from producing output. Alternatively, a faulty or malicious library could ignore the lock and corrupt the output stream, preventing the system from properly distinguishing between stdout and stderr. These types of attacks are not possible in standard UNIX and arise because the functionality provided in a library is unprotected from the application. Binary rewriting technologies such as SFI [WLAG93] or EEL [LS95] could be used to provide this protection, but cannot be efficiently applied to existing application binaries.

`/proc` can be used to intercept all `write()` system calls and to do the multiplexing safely, but also introduces significant overhead due to address space context switching as discussed in 3.1.1.

Scalability can also be improved by piping I/O through the GLUnix daemon instead of directly from remote children to the startup process. The daemons can then form a tree, funneling the output down to a single communication stream. This reduces the number of I/O connections needed at the startup to 1, removing the limit on the parallel degree of applications and improving performance during startup of parallel jobs. The daemons can then use Active Messages or other light-weight communication layers to transfer the I/O. There are, however, two limitations to this solution. First, it introduces numerous extra network hops for I/O, increasing latency considerably. Second, piping I/O through a daemon located on the same node as the application requires context switching from the application to the daemon and back, decreasing throughput. Connecting the application to a daemon running on a different node does not solve the problem in general, as there may be some other application on that node attempting to do I/O.

A more elegant solution to this problem would be to intercept I/O `read()` and `write()` system calls in the kernel, avoiding the synchronization complexity resulting from `fork()` as well as the performance problems arising due to context switches. Once the I/O system calls have been intercepted, data can be shipped using Active Messages or other light-weight protocol.

3.5 Coscheduling

GLUnix currently approximates coscheduling by using `SIGSTOP/SIGCONT` signal pairs. Unfortunately, this technique is not transparent to some programs. The delivery of the `SIGCONT` signal will generally interrupt any system call which the process may have been blocked in, returning an `EINTR` error code to that process. While this is the same behavior that a `Ctrl-Z/fg` combination would have from a shell, we have encountered a small number of programs and communication libraries which did not properly check for this condition and required modification to execute correctly when scheduled using this method.

3.5.1 User-level Approaches

One solution to this problem is to modify the C library to monitor all system calls issued through the library. When a system call fails with `EINTR`, the modified library could contact GLUnix to determine if a coscheduling signal was just sent to this process. If so, the library would transparently restart the system call. `/proc` could be used to provide the same functionality. However, to make this truly transparent, the library must only restart system calls that were interrupted by GLUnix coscheduling signals, but not by other signals. In general, however, it is not possible to correctly correlate the cause and effect of signals in this way; for example, if another signal arrived just before the process was descheduled by GLUnix, it is difficult to determine if the `EINTR` was caused by the coscheduling signal or by some other signal. However, since this is difficult to determine, it is unlikely that an application could depend on this behavior. The primary limitations of the `/proc` approach lie in the area of performance, while those of the modified C library lie in the areas of transparency for all existing application binaries.

A more robust and efficient solution to this problem would be to intercept the GLUnix coscheduling signals on delivery in the kernel. Instead of delivering the intercepted `SIGSTOP` signal to the application, signal delivery would be suppressed but the application would be put to sleep in the kernel on a semaphore. When the `SIGCONT` signal arrived, it too would be suppressed but semaphore would be released, enabling the application to continue running.

3.6 Migration

Migration is an extremely useful feature to provide in a network of workstations, enabling jobs to be optimally remapped to different nodes at run-time as cluster load fluctuates and to avoid scheduled down-times of nodes, for example, for maintenance. GLUnix, however, does not currently provide support for migration. Although Condor [BLL91], a widely-available user-level migration package, could be integrated into GLUnix, because of the limitations arising Condor's user-level construction, GLUnix would be unable to transparently provide migration for arbitrary application binaries. For example, because of its user-level construction, Condor cannot properly migrate applications which use open but deleted files or which use interprocess communication; this effectively precludes the vast majority of parallel jobs as well as any jobs making use of the GLUnix library.

UNIX provides the ability for an application to open a file, delete it, and then continue reading and writing that file's data; the file blocks are not reclaimed by the file system until after the deleted file is closed. A number of UNIX utilities use this capability for creating temporary files. However, when an process exits, either because it is being migrated to another node or it has died and will be restarted from a checkpoint, UNIX automatically closes any remaining open files in the process, causing open but deleted files to be reclaimed. When the migrating program is restarted on the remote node, the file that was deleted will no longer be accessible.

A second problem occurs with communication channels. When a process exits, UNIX closes any communication channels opened by the application. Closing these channels causes the remote end of the communication channel to receive an indication that the channel was closed. Transparent migration and checkpointing would require maintaining the open communication channel until the process is restarted as well as redirecting all future traffic to the new location. Any data for the communication channel buffered in the kernel would also have to be extracted from kernel buffers and reinserted into the stream when

```
file:   file.o
        cc -o file file.o

.o.c:
        cc -c $^
```

Table 3.1: Sample `Makefile` which exposes inconsistency of NFS when run under `glumake`.

the new process resumes execution at the new location. UNIX domain sockets provide an additional problem since open file descriptors can be passed from one application to another across these communication channels.

3.6.1 User-level Approaches

The simplest user-level approach to migration is to redirect all system calls, signals, and other events for the remote application through a startup process. By placing a level of indirection for the remote application in the startup process, the remote process can be moved to a new location with all state for currently open but deleted files and communication channels being handled by the startup process. This solution, however, has clear performance problems.

Once an open but deleted file is closed, the internal UNIX file system will permanently remove the file from the system. Neither `/proc` nor a modified C library is capable of intercepting the interfaces necessary to prevent this. However, both `/proc` and a modified C library could be used to move the file to a hidden location rather than deleting it. The file would then be deleted by the user-level system when the application finally closes the file. This allows the file to be reopened at the new location after migration. Since the file is still accessible via the file system, other applications could read or write data in the file, violating expected semantics, or delete the file, preventing a newly migrated process from reopening it. To prevent this, a modified C library or `/proc` would have to be used to monitor all applications in the system, preventing them from accessing or deleting the file. A modified C library would not be able to guarantee that other applications would not access or delete the file since a modified C library can only catch system calls invoked via library functions—assembly traps embedded in the application can circumvent the library, calling directly into the kernel. Binary rewriting techniques such as SFI or EEL could be used to reliably intercept all system call traps, but cannot currently be applied efficiently to existing application binaries. `/proc`, however, can reliably prevent all accesses or deletes of the file by intercepting the `open()` and `unlink()` system calls for all applications running in the system.

3.7 NFS

The weak cache consistency policy of NFS [WLS+85] for file data and attributes limits the utility of the `glumake` program which distributes compilations in parallel across the cluster. To improve performance, NFS clients cache recently accessed file attributes. Elements in this cache are flushed after a timeout period, eliminating most stale accesses in traditional distributed computing environments. The presence of these stale attributes, however, can break the illusion of transparent remote execution for programs that depend on them, such as `glumake`. When `glumake` spawns off a remote child to perform a compilation, the file attributes are properly updated on the node where compilation takes place, but the old attributes remain cached on other nodes. When checking file modification times to determine if a target needs to be rebuilt, `glumake` can incorrectly conclude that the target dependencies have not been updated because of these stale cached values. A typical manifestation of this problem is when a source file is recompiled into an object file, but the resulting executable is not built. Because `Makefile`s can be used for a wide variety of tasks, the `glumake` algorithm does not assume any knowledge about the nature of targets or commands executed. It does not know that recompiling a source file necessitates the recompilation of the associated executable; it only derives this information by noting that the intermediate object file on which the executable depends was updated. Table 3.1 shows a sample `Makefile` that exhibits this problem. After `file.c` is updated, lines 4 and 5 will cause `file.c` to be compiled into `file.o`. However, the time-stamp of `file.o` may not update right away on all nodes in the cluster, causing the dependency check for line 1 to access the old time-stamp rather than the new one. Consequently the executable may not be relinked.

3.7.1 User-level Approaches

While `glumake` could be modified to always relink a target when any of the items in its dependency tree have changed without checking intermediate timestamps, this approach is *ad hoc*—it does not solve the problem in general for all applications, nor is it transparent. This problem can also be resolved for a given `Makefile` by modifying it to add dependencies between the source files and the executable. However, requiring users to modify `Makefile`s to avoid consistency problems in the filesystem is neither transparent nor reasonable.

Another solution would be to reroute all file system updates through the GLUnix startup process so that there is a single consistent view of the timestamps. However, sending all file I/O during parallel compilation through a single node introduces extra network hops and data copies, decreasing performance and undermining much of the benefit of parallel compilation.

To resolve this problem without incurring the overhead of redirecting all file I/O through the home node, `/proc` or a modified C library could be used to implement a coherent attribute cache, bypassing the inconsistencies of NFS. When an application updates a file, that operation will be intercepted by `/proc` or a modified C library which will record the timestamp of the update. When another application on a different node queries the attributes of the file, instead of returning the possibly stale attributes from NFS, the query is intercepted and the proper attributes are returned. This enables any of a variety of distributed consistency algorithms to be used to keep attributes up to date.

However, managing distributed file system state cannot be safely and efficiently done in a library which runs unprotected from potentially malicious or faulty applications. Furthermore, applications which circumvent the C library routines or which are statically linked would still be exposed to the potential inconsistencies of NFS. Since querying file attributes is a common operation in a file system, the context switch involved in using `/proc` to intercept such attribute queries can lead to considerable overhead.

Solving this problem safely, efficiently, and transparently for all application binaries requires the ability to reliably and cheaply intercept all system calls for file accesses and attribute queries while being protected from malicious or faulty applications.

3.8 Heterogeneity

GLUnix currently only supports a homogeneous cluster, requiring all nodes to be in the same architecture family and run the same operating system. However, partially relaxing this constraint and allowing cluster nodes to run different operating systems would enable applications with specialized needs to run on the most suitable operating system. However, even if all nodes in the system are still of the same hardware architecture family, running different operating systems in the cluster can prevent applications compiled for one operating system from running on certain nodes of the cluster. Parallel applications will not be able to run a single binary across all nodes of the cluster, requiring users to maintain multiple binary versions of parallel programs.

A method which enables an application binary compiled for one operating system to transparently run on another operating system would be of great benefit in a cluster environment, as well as others.

3.8.1 User-level Approaches

Assuming that the hardware architecture families are the same and only the operating systems are different, a modified C library could be used to emulate the system call and signal interfaces of one operating system on top of another. When the application is loaded on the remote node, the C library which is dynamically linked with it will be provided by the remote operating system. On initialization, the library would communicate with the local GLUnix daemon to determine which operating system this application was compiled for. The C library would then intercept library calls made by the application, translating the argument formats of those calls to that of the native operating system. Similarly, the library would install handlers for all signals and on signal receipt would translate the signal information into the format expected by the application. There are, however, a number of limitations of using a modified C library for this solution. First, this approach will not transparently work for statically linked application binaries. Second, a modified C library will not catch system call traps made directly in the kernel by the application. Finally, if the application or some other library to which it is linked, whether statically or dynamically, overrides some of the modified C library routines in order to add some other functionality, the modified routines of the C library will not correctly emulate the behavior expected by the application.

`/proc` could also be used to intercept all system calls and signals of the application, emulating the operating system behavior expected by the application. This solution, however, would require multiple context switches for every system call and signal event, suffering a substantial performance penalty.

A simpler solution would be to reliably, efficiently, and transparently intercept all system calls and signals for an application.

3.9 Final Thoughts

Even if the problems mentioned above for a modified C library or the `/proc` file system are overlooked, there is another limitation of each which bears mentioning. If all the above limitations were solved using one of these two techniques, it would have to be done in a single solution—only one process at a time may attach to a given process using `/proc` and only one C library will be linked into an application at load time. Ideally, we should be able to apply solutions incrementally and composably, simplifying development and upgrades.

3.10 Problem Summary

Because of the problems listed in this chapter, GLUnix is unable to reliably, efficiently, and transparently provide remote execution and other cluster services from the user level. The next chapter introduces a general-purpose extensibility mechanism which can enable GLUnix to overcome these limitations. The following chapter will illustrate how that extensibility mechanism can enable GLUnix to overcome a representative set of the above limitations. Although SLIC solves or simplifies all of the above limitations by providing efficient access to the necessary interfaces, we will focus further discussion on four particular problems: lost input, group signaling, remote device access, and scalability limitations.

Chapter 4

SLIC: Extensibility for Commodity UNIX Operating Systems

Previous chapters have described the functionality needed to provide cluster services to a NOW and described the limitations encountered in attempting to implement that functionality at the user level. This chapter demonstrates that it is possible to provide a general-purpose extensible system capable of supporting a wide range of value-added functionality with little or no changes to existing commodity UNIX kernels. In particular, this mechanism enables user-level operating system layers to download small extensions into existing commodity UNIX kernels in order to overcome the limitations typically associated with user level approaches. This chapter presents the design, implementation, and evaluation of SLIC, a prototype of this extension mechanism which will be used in the next chapter to resolve a representative sample of the limitations of GLUnix identified in the last chapter.

4.1 Introduction

A number of systems have succeeded in reducing the cost of adding new functionality to operating systems by providing third-party extensibility. These systems include Hydra [WCC+74], SPIN [BSPP+95], VINO [SS94a], Exokernel [EKJO95], and Fluke [FHL+96]. However, these systems themselves are novel operating system architectures requiring substantial implementation effort in order to provide this extensibility. Consequently, although extensibility offers a great many advantages to system software developers, the high cost of providing it has prevented its wide-spread adoption. The result has been that existing commodity operating systems remain largely inextensible.

This chapter presents a low-cost approach to providing third-party extensibility for existing commodity UNIX operating systems. We take the approach of adding functionality with only minor modifications to the underlying operating system and no modification to application code or binaries. Our work differs from earlier efforts in that our solution—kernel-level interposition of trusted extensions on kernel interfaces—is simple to implement, is efficient, requires no specialized hardware support, protects extensions from malicious or faulty applications, enforces extensions on uncooperative applications, and supports extension stacking. We believe that no other system provides this powerful combination of features for extending existing commodity UNIX operating systems.

Prior attempts to provide operating system extensibility without significant modification suffered from significant limitations. Interposition Agents [Jon93] demonstrated an object-oriented toolkit for structuring extensions; it leveraged the Mach [ABB+86] system call redirection facility to transparently reroute application system calls to user-level extension code loaded in the application's address space. However, because extensions run unprotected in the application's address space, extensions require application cooperation and cannot enforce security guarantees or manage shared resources for competing applications. Software Fault Isolation (SFI) [WLAG93] uses binary rewriting techniques to insert protection checks on application `load` and `store` instructions, enabling extensions to be protected from faulty or malicious applications even when loaded in the same address space. Unfortunately, SFI requires a number of compiler optimizations to achieve low overhead and therefore cannot be applied efficiently to existing application binaries. Protected Shared Libraries [BTC97] uses per-thread segment registers not found on most architectures to efficiently protect libraries from applications without the need for compiler optimizations, but does not enforce extensions on applications.

To overcome the limitations of these systems, we have developed SLIC, a prototype system for efficiently inserting trusted extension code into existing operating systems with minor or no modifications to operating system source code. Conceptually, SLIC dynamically "hijacks" various kernel interfaces (such as the system call, signal, or virtual memory paging interfaces) and transparently reroutes events which cross that interface to extensions located either in the kernel (for performance) or at the user-level (for ease of development). Extensions both use and implement the intercepted kernel interface, enabling new functionality to be added to the system while the underlying kernel and existing application binaries remain oblivious to those extensions. SLIC dynamically interposes extensions on kernel interfaces by modifying jump tables or by binary patching kernel routines. The prototype currently runs on Solaris 2.5.1 and only supports interposition on the system call, signal, and kernel/application copy interfaces, although the concepts apply equally well to other interfaces in the kernel such as interrupts and virtual memory.

To demonstrate the utility of SLIC, we have used the prototype to implement a number of extensions which would have been significantly more difficult to accomplish by other means. One extension patches a security flaw publicized by CERT [Tea96b]. A second extension encrypts file, while a third provides a restricted process execution environment.

The rest of this chapter is organized as follows. Section 4.2 provides background on interposition and its characteristics. In section 4.3, we describe the design, implementation, and performance of SLIC, our prototype interposition system. Three sample extensions and their performance are presented in section 4.4.2. In section 4.4 we evaluate the prototype's overhead and present three sample extensions. Based on our experience with interposition as an extension tool, we also present the lessons we have learned about building system interfaces to support interposition effectively. Section 4.5 closes with a discussion of related work

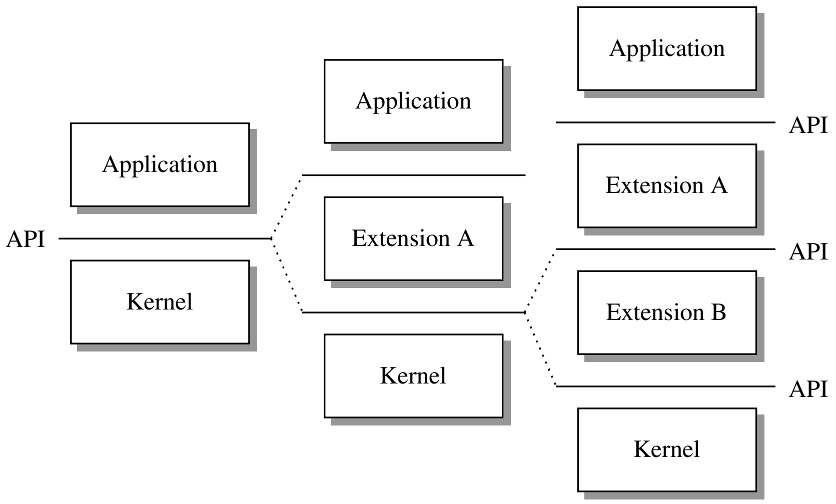


Figure 4.1: Interposing extensions on an interface. Solid lines indicate interfaces between components. Dotted lines illustrate the interposition of an extension on an interface. Events on the original interface are intercepted and routed through extension code. In this diagram, the original API on the left is maintained at each level on the right, making the interposed extension transparent to the applications, the kernel, and other extensions.

4.2 Interposition Background

Interposition is the process of capturing events crossing an interface boundary and forwarding those events to an interface *extension*. The extension performs some processing on the event and then either passes the event on to its original destination, modifies event arguments, calls other lower level routines, or forces the event to return. Figure 4.1 illustrates interposition on an interface by first one extension and then a second.

Interposition has a number of useful properties: it is *transparent*, *incremental*, and *composable*. Interposition is *transparent* because inserted extension code both uses and implements the original interface, enabling user applications and the kernel to remain oblivious to extension code.

Interposition is *incremental* since extensions need only capture the events that they are interested in. Extensions are not required to handle all events crossing the interposed interface boundary, enabling them to leverage the functionality of the existing interface; for example, an extension logging `fork()` system calls can use the underlying operating system's `write()` system call to store the log. Hence, extension writers only have to implement the desired extension functionality, not the functionality of the entire interface.

Because interposition maintains the original interface above and below the extension, it can be applied recursively, enabling multiple, independent extensions to *compose* their functionality. The right-hand side of Figure 4.1 shows the addition of a second extension to the extension stack. In the diagram, extensions A and B are oblivious of each other's presence, just as the application and kernel are oblivious to the presence of either extension.

The transparency, incrementality, and composability of interposition make it uniquely suitable for extending existing interfaces. Specifically, the transparency of interposition implies that interfaces not designed for extensibility can be extended. Incrementality simplifies extension development by ensuring that extensions need only provide the desired functionality without re-implementing substantial portions of the kernel. Composability means that multiple extensions provided by independent, third-party vendors can be applied to a single interface.

Using kernel-level interposition, extensions have a broad range of capabilities. Extensions can provide security guarantees (e.g., patching security flaws or providing access control lists), virtualize resources (providing cluster-wide process identifiers), modify data (transparently compressing or encrypting files), re-route events (sending events across the network for distributed systems extensions), or inspect events and data (tracing or logging).

However, interposition does have two important limitations. First, interposition requires a well-defined interface on which to capture events. Systems with poorly decomposed functionality may have few such interfaces. Second, new functionality can only be implemented in terms of existing functionality; for example, a cache-coherent file system can only be constructed through interposition if underlying layers expose a cache management mechanism in the file system interface [KN93].

Despite these limitations, the power and flexibility of interposition has led to its widespread use throughout modern computing systems. Forms of interposition can be found in virtual machines [IBM74], distributed file systems such as NFS [SGK+85], distributed shared memory systems such as TreadMarks [ACD+96], the 'pipe' construct of UNIX shells [RT74], World Wide Web proxy caches [BDH+94], MS-DOS terminate-and-stay-resident (TSR) utilities [SMK+90] and Macintosh toolbox extensions [Com92] (see Section 4.5 for a discussion of each of these).

4.3 SLIC Design and Implementation

To investigate the suitability of interposition for adding new functionality to existing operating systems, we have designed and implemented Slic, an interposition system for commodity UNIX operating systems. Slic leverages the transparency, incrementality, and composability characteristics of interposition to provide extensions with the following features:

Security: Extensions are protected from malicious or faulty applications and are enforced on uncooperative applications. This feature allows extensions to manage shared resources and/or enforce security guarantees. (Note that Slic assumes that extensions are trusted. Other research efforts have addressed issues involved with untrusted extensions [WLAG93, BSPP+95, HFG+96, NL96, PB96, SESS96].)

Ease of Development: During development and testing, extension writers are able to use state-of-the-art programming tools such as symbolic debuggers and performance analysis utilities.

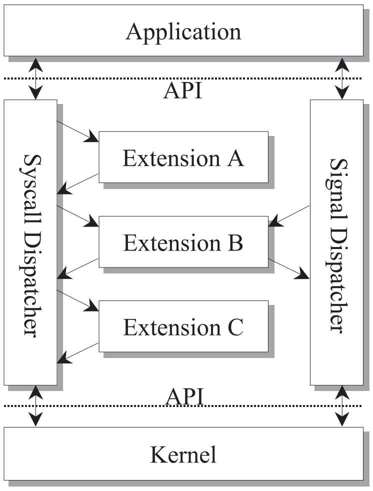


Figure 4.2: SLIC dispatchers and extensions. The dotted lines represent the interposed interface. Events crossing this interface are captured by a *dispatcher* and forwarded to one or more *extensions*.

Efficiency: Once development is complete, extensions can be invoked with minimal overhead. Per-extension overhead is a few times the cost of a procedure call. Processes that don't use an extension experience minor performance slowdown.

4.3.1 SLIC Architecture

SLIC is comprised of multiple *dispatchers* and *extensions* as well as various *support routines*. Dispatchers are responsible for intercepting system events on a particular interface and for routing those events to interested extensions. Extensions implement new operating system functionality by receive events from one or more dispatchers and inspecting or modifying those events. Support routines provide extensions with a simple, consistent interface to useful functionality such as memory allocation and synchronization primitives. Each dispatcher may provide additional support routines as appropriate for the interface; for example, our system call dispatcher provides routines to determine the children of a given process.

Dispatchers

Each SLIC dispatcher captures events on a single system interface. Dispatchers use two techniques to intercept interface events. For those interfaces which are invoked via jump tables, such as the system call, `vnode`, and virtual memory interfaces of Solaris, the dispatchers saves the original function address from the jump table and replaces it with the address of its own interception routine. For procedural interfaces which are called directly from various locations in the kernel, such as the signal interface, dispatchers must use binary patching to intercept events. The first few instructions of the relevant procedure are saved and replaced with instructions to jump to the dispatcher whenever the procedure is called. When the original routine needs to be invoked, the saved instructions are executed and control is returned to the original routine at the instruction following the binary patch. Using these techniques, SLIC dispatchers can capture interface invocations at the cost of a procedure call.

Once an event has been captured by a dispatcher, that event is passed to interested extensions for processing. Figure 4.2 depicts the relationship between dispatchers and extensions. The dispatcher maintains an ordered list of extensions through which intercepted events flow down and return values flow back up. Dispatchers do not make any policy decisions regarding the ordering of extensions—extensions are ordered by the system administrator.

Figure 4.3 shows a simplified portion of the system call dispatcher interface. Extensions express interest in certain events using a small number of predicates defined by the dispatcher. For example, our system call dispatcher filters system call events based on process identifier and system call type; an extension that only wishes to trace the `open()` system call from process 4191 can specify this easily by invoking `Slic_TraceProc` for process 4191 and calling `Slic_RegisterHandler()` to register a handler only for the `open()` system call. Our signal dispatcher provides similar functionality, enabling filtering on process identifier and signal type.

Upon receiving an event, an extension has a number of options available: the extension can pass the unmodified event down the stack by simply calling `Slic_IssueSyscall()`, the extension can modify the event parameters in `struct Slic_SyscallInfo` and then pass it along, or by setting fields in the `Slic_ReturnInfo` structure, the extension can force the event to return back up the extension stack with an arbitrary return value or error condition (e.g., when a system call would exploit a known security hole). When an event is forced to return, extensions further down the call chain (including the original kernel routines) never see the event. The return value flows back up the chain in reverse order, allowing interested extensions to inspect or modify the returned value. Additionally, while processing one event, extensions may arbitrarily initiate other events on the same interface using `Slic_IssueSyscall()` with a different `Slic_SyscallInfo` structure, capturing the return values as needed. For example, an extension logging `fork()` system calls can initiate a `write()` system call to store the log on disk. Additional events generated by extensions are passed by the dispatcher to the next extension in the chain. To later extensions in the chain, including the kernel, these extension-initiated events are indistinguishable from application-initiated events and are executed with the privileges (and limitations) of the user process.

Extensions

The SLIC architecture enables extensions to be structured in two ways, supporting a tradeoff between ease of extension development and performance. Extensions can be loaded at the user level or in the kernel, as shown in Figure 4.4, or as a combination of these types. The "Bounce" extension receives events from the dispatcher and hands them off to the user level extension; it also acts as an in-kernel proxy for the extension in order to access the dispatcher's utility functions from the user level. By providing extensions with the same interface whether at the user level or in the kernel, extensions can be easily developed and tested at the user level and then inserted into the kernel for performance. In

```
struct Slic_SyscallInfo {
       int syscallNum;
       int args[];
};

struct Slic_ReturnInfo {
       bool forceReturn;
       int  returnValue;
       int  errno;
};

Slic_RegisterExtension(int dispatcherId);
Slic_RegisterHandler(int syscallNum,
         void (handler)(Slic_SyscallInfo *sysInfo,
                        Slic_ReturnInfo *retInfo));
Slic_TraceProc(int pid, bool traceAllChildren);
Slic_UntraceProc(int pid, bool untraceAllChildren);

Slic_IssueSyscall(Slic_SyscallInfo *syscallInfo,
                  Slic_ReturnInfo *returnInfo);
```

Figure 4.3: A simplified version of the interface exported by the system call dispatcher to extensions.

all cases, the extensions are protected from and enforced upon uncooperative applications.

Extensions loaded at the user level are each encapsulated in a separate user-level process. This architecture enables extension development to proceed just like standard user applications, with access to user-level libraries (e.g., communication libraries) and state-of-the-art development tools (e.g., symbolic debuggers and performance analysis utilities). User-level extensions are protected from malicious or faulty applications by virtue of running

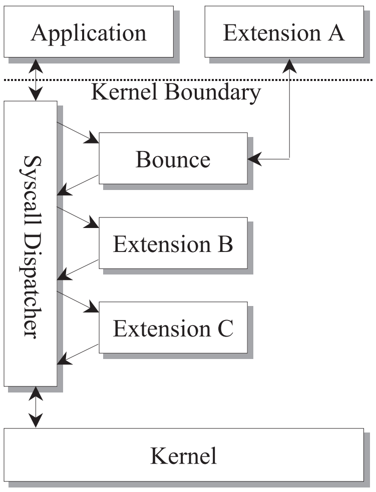


Figure 4.4: User-level extension environment. Extension A is being developed at the user level while extensions B and C are loaded in the kernel.

in a separate address space and are enforced on applications by the dispatcher which remains in the kernel. The disadvantage of this approach is that invoking the extension from the dispatcher requires costly context switches and kernel boundary crossings. This organization is similar to that employed by micro-kernels such as Mach [ABB+86] and `/proc` based systems such as Ufo [AISS97] but differs in that it supports extension stacking.

To maximize performance once development and testing are complete, extensions can be loaded directly into the kernel where they are invoked directly from the dispatcher with a procedure call. When events are frequent, this organization has considerably better performance than the user-level approach. In-kernel extensions are protected from malicious applications by virtue of being loaded in the protected kernel region of the address space; they are enforced on uncooperative applications by the dispatcher. There are two limitations to this approach: (i) the kernel is not protected from malicious or faulty extension code and (ii) there is no support for user-level development tools.

Using a simple upcall/downcall interface, SLIC extensions can also use both models simultaneously. Performance-critical sections of an extension can be located in the kernel, while functionality that is rarely used or which requires access to user-level libraries can be located in a user-level server [SKS90].

Appendix A provides a number of detailed examples illustrating the extension interface.

4.3.2 SLIC Implementation

The current implementation of SLIC provides dispatchers on the system call and signal interfaces of Solaris 2.5.1 running on SPARC 10, 20, and UltraSPARC workstations[1]. To minimize kernel modifications, all SLIC components—dispatchers, extensions, and support routines—are dynamically loaded into the kernel as loadable device drivers.

Solaris system calls are routed through the `sysent` table, which contains function pointers to the appropriate system call routines. The system call dispatcher intercepts system call events by replacing entries in this table with pointers to its own dispatch function. Solaris signal delivery proceeds through the `sigaddq()`, `sigaddqa()`, and `sigtoproc()` functions. The signal dispatcher intercepts signals by binary patching these functions at run-time. To enable binary patching, we changed one line of Solaris source code to make the kernel code writable by the SLIC dispatchers, although it should be possible to accomplish this without source code changes by manipulating the memory management hardware directly when SLIC is installed.

The current implementation catches events within the kernel, rather than at the machine level. System calls, for example, are caught at the `sysent` table rather than upon execution of the `trap` instruction. While the two approaches are conceptually similar, intercepting events within the kernel leverages a significant amount of machine-specific code which considerably simplifies the prototype.

[1] Although the SLIC prototype is implemented on Solaris, the principles underlying SLIC are generally applicable; we believe SLIC could easily be ported to other UNIX operating systems.

Shadow Structures

SLIC dispatchers and extensions need a way to record state that persists across event invocations. For instance, the system call dispatcher needs to keep track of which processes are marked for tracing. The appropriate place to store this information is in the process table or the thread structures, but in many operating systems, the size and organization of these structures are compiled into system utilities and other kernel modules.

To minimize modifications to the operating system, we implement *shadow structures* for processes and threads to store state for SLIC dispatchers and extensions. A naïve approach to implementing shadow structures would be to modify process creation and cleanup routines to include a call to manage the shadow structures. Instead, SLIC uses interposition to maintain shadow structures without kernel modifications. Shadow structures are created on demand, when a dispatcher or extension attempts to access a shadow structure. Initialization routines allocate space for the shadow structure and initialize it with a pointer to the underlying kernel's structures. To remove shadow process structures, SLIC interposes on the `wait()` and `waitid()` system calls to detect process death. In UNIX, all processes must be waited on, even if by the `init` process. By observing the return values to the `wait()` and `waitid()` system calls, SLIC determines which processes have exited and removes the corresponding shadow structure. Similarly, thread shadow structures can be deleted either on a `thr_exit()` system call or when the thread's process exits.

System Call Buffers

System calls transfer two forms of data: pass-by-value arguments and pass-by-address memory buffers. Inspecting or modifying the pass-by-value arguments in an extension is straightforward. Inspecting or modifying memory buffers located in an application's address space, however, requires more care.

There is a window of vulnerability between the time when an extension inspects or modifies a user-level data buffer and the time when the underlying operating system executes the system call. Other threads in the application can exploit this window of vulnerability to alter the buffer, effectively circumventing any security checks performed by an extension, thus violating the requirement that extensions be enforced on applications. For example, an application may be able to circumvent an access control list extension of the file system by changing the path name of the `open()` system call during this window of vulnerability.

To prevent this situation, before a memory buffer can be inspected or modified by an extension, that buffer must be protected from modification by the application. Copying these buffers to the kernel provides the necessary protection but introduces a different problem. During system call execution, UNIX kernels perform protection checks on memory buffer accesses, requiring that those buffers be located in the application's address space. These checks, performed in the kernel's `copyin()` and `copyout()` routines, are normally necessary to prevent malicious applications from accessing sensitive kernel data. Once a buffer has been copied into the kernel by an extension, the kernel's own security checks will fail when the underlying kernel routines attempt to copy the data, disallowing access to the buffer.

Our solution to this problem is to again apply interposition. SLIC maintains a per-thread list of valid, in-kernel extension buffers and interposes on the `copyin()` and `copyout()` routines in order to permit access to these buffers when appropriate. When an extension allocates an in-kernel buffer to a thread, SLIC adds the address and length of the allocated region to that thread's valid list. When the `copyin()` or `copyout()` routines are invoked, the interposed code checks the target address against the thread's valid list. If the address is a valid extension buffer, then instead of the original `copyin()/copyout()` routines, the kernel's `bcopy()` routine is invoked to copy the data to its final location. If the address is not a valid extension buffer, then control is passed to the original copy routine which performs the normal protection checks.

Note that this process introduces an extra copy for data buffers; buffers are copied into the kernel for use by extensions and are later copied again by the underlying kernel routines. Eliminating the extra copy is impractical since numerous kernel routines expect buffers to be copied to stack frames which have not yet been allocated when the extension is invoked.

4.4 Evaluation

To evaluate the effectiveness of SLIC at enabling efficient extensibility to existing operating systems, we first evaluate the overhead introduced by SLIC in performing the interposition and managing the extension stacks. Then we present three extensions implemented using SLIC: a patch for a security hole in Solaris, an encryption file system layer, and a restricted execution environment. Finally, we evaluate the suitability of existing kernel interfaces for interposition, defining how interfaces may be best designed to maximize the effectiveness of interposition.

4.4.1 Microbenchmarks

To measure the overhead imposed by SLIC on system calls and signals, we ran three microbenchmarks on a 167MHz UltraSPARC running Solaris 2.5.1. Unless otherwise noted, benchmark timings are averaged over 100,000 runs. The first microbenchmark performs a `getpid()` system call, which in Solaris is essentially a null system call. This microbenchmark measures the raw overhead of the system call dispatcher, but does not invoke the modified copy routines since there are no buffers involved. To quantify the overhead resulting from our interposition on the copy routines, the second microbenchmark performs a `sigprocmask()` system call which involves a memory copy of 16 bytes. The `kill()` microbenchmark measures the overhead of the signal dispatcher. It involves a single process sending itself a `SIGUSR1` signal. We use a single process to avoid context switches, thus maximizing the effect of our overhead. Because of the longer run time, results for this benchmark are averaged over 10,000 runs.

We tested our microbenchmarks with various configurations of SLIC: an unmodified system, a system with SLIC dispatchers but no extensions installed, and a system with the SLIC dispatchers and multiple extensions installed. The results of the `getpid()` benchmark with a varying number of extensions are presented in Figure 4.5. The incremental

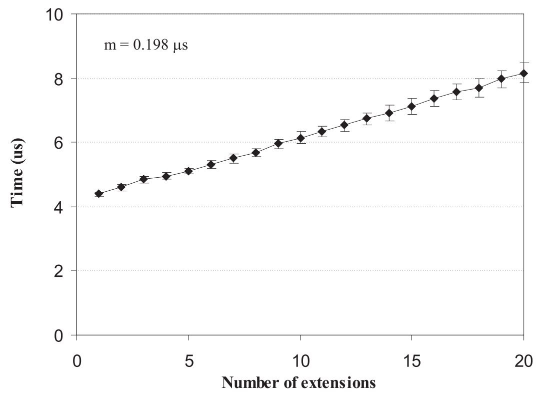


Figure 4.5: `getpid()` microbenchmark performance with a varying number of extensions. The time on an unmodified system is 2.82 μsec. Error bars indicate one standard deviation above and below the average.

	`getpid()`		`sigprocmask()`		`kill()`	
	Time (μs)	σ	Time (μs)	σ	Time (μs)	σ
Unmodified system	2.82	0.05	3.75	0.20	103.52	1.56
SLIC, no extensions	3.16	0.05	4.19	0.10	108.76	1.53
SLIC, 1 null extension	4.38	0.07	5.56	0.09	109.99	1.49
SLIC, user-level extension	61.83	0.81	60.20	0.88	187.67	2.00

Table 4.1: Microbenchmark performance of SLIC. For each benchmark, the table shows the average elapsed run time in microseconds and the standard deviation across the runs. The `getpid()` and `sigprocmask()` tests were run with the system call dispatcher loaded while the `kill()` test was run with the signal dispatcher loaded. User-level extension results are averaged over 10,000 runs.

cost of adding a new extension is statistically similar for all three benchmarks. Table 4.1 presents the one-time overhead cost of interposition for each of these benchmarks as well as the time to invoke a user-level extension.

The base overhead of the SLIC dispatchers is approximately 1.5 μsec for system calls and 5 μsec for signals. The incremental cost of loading additional extensions is approximately 0.2 μsec for both dispatchers. The extensions used for these measurements are null extensions which inspect events arguments but not event return values. Inspecting the return values in an extension consumes two additional stack frames and SPARC register windows—one is consumed by the extension while awaiting the return value and a second is consumed by the dispatcher which invoked the extension. Consequently, inspecting return values increases the per-extension cost from 0.2 μsec in Figure 4.5 to 1.60 μsec. The average cost of a standard procedure call which spills a register window on this platform is approximately 0.8 μsec.

There are two reasons why the overhead for the signal dispatcher is higher than that of the system call dispatcher. First, the signal interface is more complicated than the system call interface. Whereas system call events have a calling thread, a system call number, and system call arguments, the signal interface has a calling thread, a signal number, a target process, and an optional target thread. The shadow structure for each process and thread must be determined before extension processing can proceed; there are consequently more shadow structure lookups for the signal interface than the system call interface. The second difference is that intercepting signal events sometimes requires calling thread to pay two interposition costs. Specifically, in Solaris, the `sigaddq()` and `sigaddqa()` routines perform signal queueing side-effects which SLIC extensions may need to leverage; both of these routines call `sigtoproc()` in order to do further signal processing. However, some routines in the kernel invoke `sigtoproc()` directly, requiring SLIC to interpose on that as well to ensure that all signals are seen by the extensions. Those code paths in the kernel which invoke `sigaddq()` or `sigaddqa()` consequently experience the interposition overhead twice.

4.4.2 Extensions

To demonstrate the functionality and performance of SLIC, we have implemented prototypes of a variety of extensions: a security patch for a recent CERT advisory, an encryption file system, and a restricted execution environment. Without SLIC, responding to the CERT advisory would have required disabling `admintool` while awaiting a patch, and the encryption file system and restricted execution environment extensions would have required substantial kernel source code modification to achieve the same functionality and performance. Further, without SLIC, adding these features to an existing kernel would require *ad hoc* changes rather than providing a general solution that can be leveraged for future extensions.

CERT Advisory Extension

The Computer Emergency Response Team (CERT) regularly provides the Internet community with information regarding system security problems. Whenever possible, these advisories include information on how to resolve the reported problem. However, due to the lack of extensibility in existing systems, frequently this advice is to completely disable the insecure feature [Tea96a, Tea96b, Tea96c]. However, using SLIC, many of these advisories could be accompanied by small extensions which would resolve the problems without requiring changes to kernel source code. Though operating system vendors do respond to these advisories by supplying patches, those patches can take weeks to become available [Tea97].

To demonstrate patching a security hole in this manner, we have implemented an extension to patch a security hole discovered in the Solaris `admintool` [Tea96b] which allowed unprivileged users to truncate arbitrary files (see section A.2 for the source code to this extension). The `admintool` utility creates a local lock file to control access to shared files. By creating a symbolic link at the lock file location, malicious users could cause arbitrary files to be truncated when the lock file was created. Our 100-line extension monitors file operations, preventing symbolic links from being created at the lock location.

SLIC thus corrects the security problem while maintaining continued use of `admintool`.

Encryption File System

In a distributed file system, maintaining file privacy is a primary concern. In a networked environment with a central file server, traditional UNIX file protections can be easily circumvented by monitoring network traffic. To protect sensitive files, users may use encryption tools such as PGP [Gar94]. However, stand-alone encryption tools and libraries can be time consuming or cumbersome to use and are not easily integrated with existing applications. A more effective method of ensuring file security is to support file encryption directly in the file system, transparently encrypting file writes and decrypting file reads when communicating with the server. Rather than rewriting all file systems to support encryption, an easier approach is to implement a single encryption extension which interposes on all file traffic.

We have implemented a simple extension to demonstrate the feasibility of file system encryption using SLIC. This extension implements a trivial *exclusive-or* encryption algorithm similar to that used to test VINO [SESS96]. The prototype extension watches for `open()` and `creat()` system calls of files with a particular suffix and then records the process identifier and the file descriptor returned to the application. On subsequent `read()` or `write()` system calls to these file descriptors, the extension applies a byte-wise `xor` on the data. Key management and encryption algorithms are orthogonal to our demonstration that interposition provides a simple yet powerful means of transforming file system data.

Restricted Execution Environment

In UNIX, processes run by a user have access to all of the resources granted to that user. There are many cases, however, in which the user does not fully trust the program being run. For example, programs downloaded from untrusted sources may actually be Trojan horses designed to steal or destroy information [Win97, Clu97]. In addition, there are cases in which the user trusts the program, but not the data being processed, as in the case of web browser helper applications used by web browsers to display various data formats. Input data could potentially exploit bugs in helper applications such as `ghostview` to insert viruses into the system [Tea95].

The tracing facility of the standard Solaris `/proc` file system is one method that has been used to construct a restricted execution environment [GWTB96]. Potentially insecure system calls are captured and then selectively denied or altered. The `/proc` approach, however, suffers from two primary shortcomings. First, intercepting system calls using `/proc` is expensive, requiring two context switches over the base system call overhead. This is especially problematic for system-call intensive applications. Second, systems using `/proc` cannot properly handle the system call buffer problem described in section 4.3.2. While `/proc` does enable an extension to inspect system call buffer data, a multi-threaded application may maliciously modify the buffer between extension validation of the buffer and kernel use of the buffer, effectively subverting the security system. Using SLIC we have implemented a restricted execution environment extension that does not have these limitations.

	Procs	Total System Calls	System Calls Caught					
			CERT		Encrypt		REE	
MAB	471	40500	0	0%	2732	7%	9246	23%
TEX	3	2748	0	0%	1635	60%	703	25%
`gcc`	379	140656	0	0%	11031	8%	47888	34%

Table 4.2: Benchmark characterization. For each benchmark, this table presents the total number of processes created during a run, the total number of system calls issued by those processes, and the number and percent of those system calls which are caught by each extension. "MAB" is the Modified Andrew Benchmark and "REE" is the Restricted Execution Environment.

This extension is a modified version of the Janus system [GWTB96] and provides the user with a configurable security environment. For example, applications can be given a subset of read/write/execute access to any number of directory subtrees. The ability to `fork()` or to perform a variety of other system calls can be disabled. Any attempts by the application to perform a restricted operation results in the extension returning an `EPERM` error. The extension monitors only the subset of system calls necessary to maintain the security guarantees, thus minimizing overhead. When traced applications invoke restricted system calls, the extension checks the arguments to the call and determines if the call should be allowed or denied. The restricted environment used for benchmarking denies 45 system calls outright (e.g., `chown()`) and performs security checks, such as checking the path or file access permissions, for 21 additional system calls (e.g., `rmdir()`).

4.4.3 Application Performance

To evaluate the impact of these extensions on system performance, we ran the extensions under three benchmarks: the Modified Andrew Benchmark [Ous90], a TEX compilation of a 494-page (1.32 MB) document, and a `gcc` compilation of `emacs-19.34` without support for X Windows. The Modified Andrew Benchmark consists of multiple phases which create directory subtrees, copy files, search file attributes via `find`, search files for a text string via `grep`, and compile files. While this benchmark fits entirely in the file cache of modern systems and is therefore no longer useful for measuring file system performance, this benchmark is useful for exposing the overhead imposed by SLIC. The TEX and `gcc` benchmarks were chosen to be representative of document processing and compilation workloads. All measurements were run on a 167MHz UltraSPARC running Solaris 2.5.1 with all benchmark data files placed in a memory-mounted `/tmp` file system. Table 4.2 reports some relevant statistics for each benchmark.

Table 4.3 presents the results of running the benchmarks with each extension as well as with all extensions simultaneously. Because these benchmarks do not create any symbolic links, the CERT extension does not catch any system calls issued by the benchmarks, effectively acting as a null extension for these tests; consequently, the overhead depicted in Table 4.3 for the benchmarks running on CERT is exclusively due to the SLIC dispatchers and infrastructure. There are three anomalies in the table worthy of note: the MAB and `gcc` benchmarks appears to run faster with all three extensions loaded than with

	MAB			TEX			gcc		
	T	σ	S	T	σ	S	T	σ	S
Baseline	15.71	0.10		14.50	0.30		160.30	1.99	
SLIC, no extensions	15.92	0.16	1%	15.06	0.23	4%	159.13	0.34	-1%
CERT	16.12	0.30	3%	15.09	0.39	4%	160.11	0.55	0%
Encrypt	17.69	0.85	13%	15.87	0.53	9%	168.30	0.53	5%
REE	15.93	0.33	1%	15.15	0.73	4%	160.89	1.63	0%
CERT+Encrypt+REE	17.24	0.09	10%	15.90	0.37	10%	166.98	3.43	4%

Table 4.3: Benchmark performance on sample extensions. "Baseline" represents a machine without any SLIC dispatchers or extensions loaded. "SLIC, no extensions" represents dispatchers loaded, but no extensions; this row measures the effect of the SLIC interposition overhead. The rows labeled "CERT", "Encrypt" and "REE" present the benchmark elapsed times with a single extension loaded. The last line shows benchmark performance with all three extensions interposing simultaneously. For each benchmark, the table shows the average elapsed run time in seconds ("T"), the standard deviation across the runs, and the percent slowdown ("S").

just the Encrypt extension loaded and the gcc benchmark appears to run faster with SLIC loaded than on the baseline system. However, an inspection of the standard deviation in each of these cases reveals that the anomalies are well within one standard deviation of the mean and are likely to be experimental variance. Though SLIC imposes a certain amount of overhead on applications, the last line in Table 4.3 illustrates that much of the overhead experienced by the benchmarks is due to the SLIC dispatcher infrastructure, a cost which is only paid once; the per-extension overhead is small for these workloads.

4.4.4 Interface Evaluation

This section describes our experiences with implementing SLIC and presents a number of general principles for developing interfaces that are conducive to interposition. Although SLIC was designed for and will work with existing operating systems, there are a number of improvements that can be made to make these systems more interposition-friendly. We have drawn these lessons from our implementations of the system call and signal dispatchers for Solaris 2.5.1, as well as preliminary analyses of the process scheduler interface of FreeBSD 2.2.5R, and the system call interface of Linux 2.0.

The problems that we have encountered can be divided into four categories: the asymmetric trust mechanisms of the system call interface, the implicit event information in the interfaces we interposed on, an incomplete decomposition of functionality in the system, and miscellaneous implementation issues.

Asymmetric Trust Mechanisms

The solution adopted by the current prototype to correctly handle system call buffers (see section 4.3.2) introduces a second copy for those data buffers which must be securely examined by an extension. This second copy is not fundamental to interposition but rather arises out of idiosyncrasies of the Solaris system call routines. In Solaris, as in most UNIX systems, system call buffers must be copied into the kernel before they can be used. The copies are only necessary on interfaces such as the system call interface which have an asymmetry of trust—applications trust the kernel, but the kernel does not trust applications. Extensions which interpose on such interfaces are viewed by the kernel as being part of the untrusted user application. Interfaces which have a symmetry of trust do not need to copy data buffers before using them because it is assumed that no other threads in the trusted domain will maliciously modify the buffers to subvert the system.

Consequently, the ideal interface for interposition is one with a symmetry of trust. For the system call interface, this would mean interposing on events *after* the data buffer addresses had been validated and the buffers themselves had been copied into the kernel. In Solaris, this is impractical because the copy routine invocations are scattered throughout the system call handler routines. The ideal interposition-friendly interface would perform any required security checks and copies in separate routines before invoking system call handlers. This approach would eliminate the unnecessary second copies and would remove the need for SLIC to interpose on the kernel's copy routines.

Implicit Event Information

Many interfaces in today's operating systems rely on *implicit information*—information that is not passed directly as an argument to the event but rather is stored in global data structures. For example, when a system call is made, the process identifier of the application is not passed to the system call handler. Similarly, the return value of a system call is not returned explicitly according to normal calling conventions, but is stored in the kernel's process structure. The credentials determining a process's right to open a file or send a signal are also stored in the process structure. Accessing this implicit information requires an understanding of complicated kernel structures and kernel locking conventions.

To minimize the dependency of extensions on a particular version of the operating system, SLIC dispatchers provide extensions with simple utility functions to access a variety of implicit event information, hiding the details of kernel data structures and locking conventions. Unfortunately, providing these functions in the dispatchers increases the dependency of those dispatchers on the particular version of the operating system. An interposition-friendly interface would make implicit event information readily accessible when an event is raised, enabling SLIC to be more easily ported to new operating system versions.

Implicit event information also limits the functionality of extensions, as in the case of implicit credentials. If extensions cannot modify event credentials, then extensions would be restricted to the rights of the calling thread. For example, an extension could not write to files owned by `root` unless invoked from a `root` process. A naïve approach to circumvent this problem would be to enable extensions to modify a process's credentials stored in global data structures for the duration of the intercepted event. However, in a multi-threading environment, other threads running concurrently may access the modified credentials, producing unpredictable results. Including credentials as explicit parameters to the event enables extensions to modify the credentials as necessary.

The `ioctl()` system call poses a different problem for interposition. Originally designed as a way to manage arbitrary devices, `ioctl()` system calls have a variable number of parameters, nearly any of which may reference arbitrary memory buffers which may in turn contain pointers to other memory buffers. The meaning of the arguments and the structure of each buffer is defined by the particular device driver. Hence an extension cannot know in advance how to handle the arguments of any given `ioctl()` call. This is problematic for extensions such as system call logging or security extensions which may need to understand the arguments of an `ioctl()`. While it is possible to derive some information about device/application interactions by interposing on the `copyin()`/`copyout()` routines, general interpretation of `ioctl()` semantics is difficult. Events with semantics specified only by the execution of code are not conducive to interposition.

Separation of Policy and Mechanism

To reduce the dependencies on a particular hardware platform or operating system version, extensions should interpose on system policies while leveraging system mechanisms. A clean separation of policy and mechanism in the compiled kernel is therefore critical. However, this principle of decomposition is violated by today's operating systems. While experimenting with extending the scheduler interface in FreeBSD 2.2.5R, we found that a single scheduler routine, `cpu_switch()`, implements both the policy of selecting the next process to run as well as the actual context switch mechanism. Enabling interposition on the policy while leveraging the existing mechanism required separating the policy and mechanism into two routines, creating a procedural interface which could be interposed on.

A variation of this lack of separation occurs with the Linux routines for accessing system call data buffers, `getuser()` and `putuser()`. These routines are inlined at compile time, making interposition at run time extremely difficult. Interposing on the copy routines requires modifying the Linux source to disable inlining of those copy routines, again creating a procedural interface that can be interposed on, albeit at a small cost in performance.

Miscellaneous Issues

Our method of using binary patching to intercept procedural invocations is simple to implement, but requires that kernel code be writable. Unfortunately, Solaris is loaded such that the kernel code is read-only, preventing binary patching. By modifying a single line of Solaris source, we were able to make the kernel code writable.

Managing shadow structures for processes and threads complicates SLIC and increases kernel memory consumption. Adding a hook to the kernel's process and thread structures would eliminate these problems.

4.5 Related Work

There has been a significant amount of prior work supporting extensibility in the operating system kernel or at the user level. A number of these projects also leverage the concept of interposition to provide this functionality. This section discusses previous work in this area.

4.5.1 Virtual Machines

The use of interposition to extend operating system functionality has a long history. Virtual machine systems, also called *hypervisors*, such as VM/370 [IBM74] performed interposition at the hardware level in order to efficiently simulate one operating system on top of another. The simulated operating system would run natively on the hardware, trapping into the virtual machine only to emulate privileged operations such as device I/O.

Later work [BS95] used the hypervisor approach to coordinate a primary virtual machine with its backup to provide fault tolerance to a system without any changes to the underlying hardware, operating system, or applications. A prototype of this approach was demonstrated on HP PA-RISC machines and showed a factor of 2 in performance slowdown. Instructions which modify or query the machine environment are trapped by the hypervisor and forwarded to the other backups which ensure that the modifications occurs identically on all backups and that queries return the same value on all machines.

Fluke [FHL+96] is a novel operating system design based on microkernel and virtual machine concepts. Fluke enables virtual machines to nest within each other, with each virtual machine layer providing a portion of the new functionality. Portions of the virtual machine not extended at a particular layer are handled directly by underlying layers through an optimized path; thus common functionality such as process management, demand paging, fault tolerance, etc. can use a common, underlying implementation. While some layers impose almost no overhead, others impose as much as 35% overhead per virtual machine layer. Microkernel techniques are used to provide "horizontal" extensibility; that is, rather than requiring an extension to intercept events from all applications in the system, an extension stack can be associated with a set of processes, avoiding unnecessary invocations of the extension stack. Unlike previous virtual machines, extensions in Fluke do not virtualize a hardware interface, but rather a software interface similar to the events in Slic. Being based on Mach, Fluke extensions are encapsulated in separate user-level processes which, using the virtual machine concepts, can be nested inside of each other.

Disco [BDR97] leverages the concepts of virtual machine monitors to run multiple copies of a modified version of Silicon Graphic's Irix operating system on a scalable multiprocessor. Virtual machines share major data structures such as program code and file system buffer caches to reduce memory overheads associated with running multiple operating systems. It achieves most of the benefits of customizing the operating system for scalable multiprocessors, but with significantly smaller implementation effort. Disco uses a "few simple modifications" to Irix and supports a partial single system image. The hardware monitor intercepts virtual memory paging events going to disk and pages to remote memory instead to increase performance. Also, the machine monitor dynamically schedules virtual machines onto processors to balance the load in the system. The operating system was extended with a special virtual memory segment driver so it can share memory with other virtual machines. Software emulated privileged instructions and accesses to I/O devices. Disco does dynamic page migration and replication to approximate a uniform memory access architecture on a non-uniform memory access architecture. For efficiency, Diso doesn't intercept all device I/O, but rather replaced a number of Irix device drivers with special drivers that make a special monitor call (so they don't have to emulate all the I/O instructions for a programmed I/O stream), passing all the I/O information to the virtual machine in a single trap; this was done for the UART, SCSI, and Ethernet drivers. Sped up loads and stores to privileged registers by mapping them into a special memory page. The operating system passes hints to the monitor—request for acquiring and freeing memory pages, and indications of idleness.

4.5.2 Extensible Operating System Kernels

There has been a considerable amount of recent work [SS94a, BSPP+95, EKJO95] that has explored novel kernel designs for extensible operating systems. Of these systems, SPIN [BSPP+95] and VINO [SS94a] are the closest in concept to our work. Both offer extensibility through interposition on a number of kernel interfaces, but have explicitly crafted those interfaces for extensibility. SPIN and VINO also aggressively focus on ensuring kernel protection from untrusted extensions, SPIN by using a type-safe language [SSP+96, HFG+96], and VINO through software fault isolation [WLAG93] and in-kernel transactions [SESS96]. In contrast, Slic assumes trusted extensions and focuses on an evaluation of the technique of interposition and its suitability for legacy operating systems. While SPIN and Slic both support extension stacking, VINO only supports a single extension per event [Sma98], effectively preventing extension stacking.

Exokernels [EKJO95] explored the concept of separating protection from policy by constructing a minimal kernel which only protects hardware resources. Higher-level abstractions are placed into application-linked libraries. Arbitrary extensions can then be encapsulated as application libraries; since the majority of interfaces typically located in the kernel are located at user-level in Exokernels, extensions have easy access to the necessary interfaces. However, because most system functionality is unprotected from the application, it is unclear, how a single extension could arbitrate resources among competing applications in an Exokernel environment.

Watchdogs [BP88] was a modified UNIX system which enabled extensibility of a particular interface, namely the file system. Each file or directory in the file system could be associated with a *watchdog*, a user-level program that would be invoked on various file operations to either approve or deny the operation in question. The system maintained binary compatibility with existing applications.

Stackable file systems in Spring [KN93], Ficus [JPGH90], and BSD 4.4 [MBKQ96] extend the UNIX `vnode` interface [Kle86] to allow multiple file system layers to be arranged in a stack. Operations to a given file then flow down the stack of file systems, much like events in Slic flow down the stack of extensions. Ficus in particular demonstrated that this approach was sufficient to implement a remote file system while leverage the block layout policies and naming abstractions of underlying layers. Spring and later work based on Ficus [Hei95] both argued that coherent caching in a stackable file system requires augmenting the `vnode` interface with cache coherency operations.

The System V `/proc` process file system [FG91] modifies the System V UNIX kernel to provide user-level extensibility of the system call and signal interfaces. `/proc` enables trusted user-level processes to attach to user-level applications. As system calls from a monitored application enter the kernel, the `/proc` file system intercepts those system calls and transfers them to the monitoring application. The monitoring application has the ability to read or write any data in the application's address space and to modify the hardware registers for the application. Signals delivered to the application can also be intercepted using `/proc` and inspected before delivery. The major limitations of `/proc` are that multiple context switches are required to intercept and service an event, leading to high overhead, and only the system call and signal interfaces are supported. For example, the copy interface used by devices to read and write application data is not intercepted by `/proc`.

DOS on the IBM PC and the Macintosh operating system both had facilities which enabled extension of system features. While not designed for arbitrary, third-party extensibility, invocation of system functionality in both systems was made through hardware trap vectors which were unprotected from applications. By replacing trap vector entries with the address of a memory-resident extension routine, developers were able to transparently add new system functionality such as background printing and file system compression. Although these systems did not protect extensions from malicious applications, since they were single-user systems this was not a major limitation.

4.5.3 User-level Extension Systems

Interposition Agents [Jon93] demonstrated the usefulness of constructing extensions in terms of the high-level abstractions of an interface (such as path names), rather than the low-level events crossing that interface (such as `open()`) and provided an object-oriented toolkit to support those abstractions. Interposition Agents used the system call redirection facility of Mach which bounces system calls to extensions linked into an application's address space. Consequently, extensions are neither protected from nor enforced on applications and thus cannot implement security extensions or share data between distrustful applications. Additionally, the multiple protection boundary crossings limit the performance of the system. Slic enables high-performance interposition that is both enforced on and protected from applications, enabling a significantly larger class of extensions. In principle, the toolkit presented in [Jon93] could be ported to Slic, further simplifying the process of extension development.

COLA [KK92] enables interposition at the system call interface, but without any modification of the operating system kernel. It operates through interposition at the library level, linking extensions into the application's address space. Consequently COLA extensions are not protected from or enforced upon applications. Uses dynamic shared libraries to enable multiple extensions to intercept system calls at the C library level. Can't intercept system call traps made in an application, so it is not transparent for all applications.

Protected Shared Libraries (PSL) [BTC97] enables extensions to be securely loaded into an application's address space, so that user programs cannot access or modify extension code or data. PSL uses per-thread segment registers of the IBM RS/6000 to protect extension code from applications. Applications invoke extension code by trapping to the kernel which enables access to the library for that thread and then vectors the thread into the extension code. On return, the extension traps into the kernel which disables access to the library, again protecting it from the application, and finally returns to the user application. PSL, however, does not provide a mechanism for enforcing extensions on applications, which Slic does. PSL is primarily intended for adding new interfaces to a system, although it could be combined with the interposition mechanisms used in Slic to enable modification of existing interfaces. Finally, whereas the PSL protection technique relies on hardware support not available in most architectures, the principles in SLIC are generally applicable across a variety of operating system platforms.

The pipe construct of UNIX shells [RT74], System V streams [Vah96], OS/2 device monitors [Let88], NFS [SGK+85], web proxy caches [BDH+94], and even hardware architecture caches are all examples of interposition on a data stream rather than a generalized event stream. each of these systems transparently inserts some functionality on an interface. Pipes, streams, and OS/2 device monitors allow data to be transformed by filters from one format to another; examples include data compression or encryption. NFS, web proxy caches, and hardware architecture caches transparently add caching functionality to a data stream, enabling data that is remote to be accessed as if it were local. Hardware caches in multiprocessor systems also add cache coherency functionality to the stream, coordinating independent updates to data; NFS and web proxy caches do not currently provide such data coherency.

4.6 Summary

SLIC is a powerful, general-purpose extension mechanism for existing commodity operating systems, enabling a wide range of operating system functionality to be implemented by third-party developers. Leveraging the concept of interposition enables existing commodity operating systems to be extended with little or no modification to the operating system. By interposing trusted extensions on existing kernel interfaces, extensions are efficiently protected from malicious or faulty applications and enforced on uncooperative applications. To simplify the process of extension development and testing, events can be intercepted in the kernel and rerouted to extensions running as separate user-level processes, enabling developers to make use of state-of-the-art development and testing utilities. Once complete, extensions can be downloaded into the kernel where they can be efficiently invoked.

The primary challenges in construction the prototype of this extension technology, SLIC, were the management of extension state and enabling extensions to modify system call data buffers. Each of these challenges were solved by an additional use of the concept of interposition. In order to selectively determine which processes are traced by each extension and to store extension-specific state, SLIC maintains shadow structures for processes and threads, interposing on events to create the structures on demand and interposing on the `wait()` system call to deallocate them. To enable extensions to safely modify system call buffers, SLIC allows extensions to copy data buffers into the kernel, modify them, and then issue the system call; providing this ability required interposition on the copy routines used to access user data from the kernel.

Chapter 5

Removing GLUnix Limitations with SLIC

Now that we have demonstrated the needs of a cluster operating system layer, identified the limitations arising out of a user-level construction, and presented a general-purpose extensibility system, this chapter demonstrates the suitability of SLIC for overcoming the limitations of GLUnix. As described in chapter 3, due to its user-level design, GLUnix has a number of limitations concerning the transparency of remote execution. This chapter illustrates how the technology introduced by SLIC can be leveraged to resolve a representative sample of the limitations encountered in the user-level construction of GLUnix. This discussion will demonstrate that, although partial functionality can be achieved in some cases using the existing user-level approaches described in Chapter 3, these mechanisms cannot offer the full breadth of functionality, performance, and simplicity that is provided by SLIC. Throughout this discussion, the bulk of GLUnix remains at the user level; for each problem discussed, a small extension is loaded into the kernel using SLIC which interacts with the remainder of GLUnix at the user level to overcome a limitation. The limitations that will be addressed are lost input, group signals, remote device access, and scalability.

5.1 Lost Input

In order to solve the lost input problem described in Section 3.1, GLUnix must be able to determine the number of bytes requested each time a remote process reads from `stdin`. Consider the example of piping a file into a shell script such as that shown in Table 5.1. The `glush` program runs individual commands on remote nodes, similar to `glurun`.

```
1:  #!/usr/now/bin/glush
2:  date
3:  wc
```

Table 5.1: Sample shell script which exposes the problem of lost input.

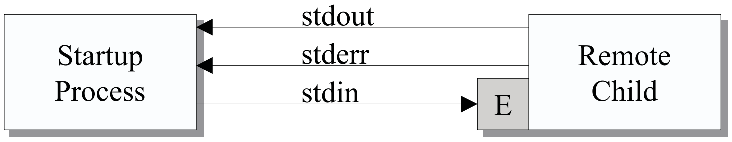


Figure 5.1: SLIC solution to lost input. The Startup Process executes on the user's home node while the Remote Child executes on the remote node. Horizontal lines indicate standard I/O streams. The box labeled "E" represents the extension loaded on the remote node.

The first command in the script, `date`, does not consume any input. However, since GLUnix does not know this, it sends all piped input to the remote `date` program; the input is lost when the `date` program exits. When the `wc` executes, there is no more input to be consumed and it returns a word count of an empty file.

Using SLIC, a small extension is loaded on the remote node, monitoring the `read()` system calls of the remote job. When the remote job attempts to read N bytes of input, the extension is invoked, determines the number of bytes requested from the system call parameters, and sends a message back along the input pipe indicating the number of bytes requested. The GLUnix startup process reads this information from the remote extension and then sends across precisely the amount of input requested. Figure 5.1 depicts this arrangement. This solution correctly solves the problem encountered by the shell script depicted in Table 5.1. The time required to pipe a 3.2 MByte file to a remote program prior to this fix was approximately 3.5 seconds using TCP/IP across a 10 Mbit/sec Ethernet, while the overhead and round-trip input request introduced by the extension increases the time to 4.5 seconds. Implementing the performance optimization mentioned above would eliminate the need for a round-trip message exchange on every read from the input stream, roughly returning the performance to what it was originally.

There are a number of advantages to this solution over the modified C library and `/proc` solutions. First, all `read()` system calls are caught regardless of whether the application is statically linked, dynamically linked, or issues system call trap instructions directly. Second, this extension is composable with other SLIC extensions as well as other modifications to the `read()` routine in the C library. Third, no special treatment is required for the `exec()` system call—the extension monitors the process based on it process identifier, which does not change across an `exec()` and the data structures managed by the extension are stored in the kernel and are unaffected by the transformation of the application's address space in response to the `exec()`. In addition, with respect to the optimization of optimistically sending input to the remote node, the input would be cached in the extension, protecting it from faulty or malicious applications and eliminating the need for cross-process synchronization or copying. Finally, SLIC extensions experience very little overhead as demonstrated by section 4.4.1, yielding a large performance improvement relative to `/proc` solutions.

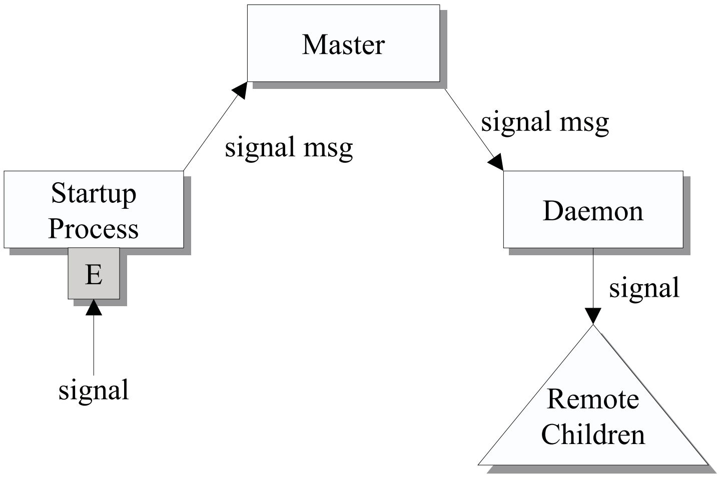


Figure 5.2: SLIC solution to group signals. The extension, "E", is loaded on the home node and monitors all signals sent to the startup process. Information on each signal, whether targeting a group or not, is handed off to the startup process.

5.2 Group Signals

In order to properly extend UNIX group signaling semantics to a cluster, GLUnix must be able to determine if a given signal was destined only for a single process or for an entire process group. Without the ability to do this, GLUnix cannot know if only a single remote process should be signaled, or whether the entire process group should be signaled. This problem occurs in practice when a remote job creates a child using the `fork()` system call; examples of jobs that may create children are shell scripts, `gmake`, and `find`. A signal delivered to the startup process may have been intended for both the remote process and it's child, or just the remote process. Consequently, GLUnix is frequently unable to properly carry out the user's intentions when signaling remote jobs.

Using SLIC, a small extension is loaded on the home node, intercepting all signals destined for the startup process (see Figure 5.2. When the startup process initializes, it marks itself for tracing by the extension, indicating that it desires group signal information. At the kernel interface, information is readily available that indicates whether the signal is destined for a process group or just a single process. When the extension intercepts a signal destined for the startup process, it records group information about the signal and allows signal delivery to proceed as normal. When the GLUnix startup process receives a signal, it calls into the extension to retrieve group signal information. The startup process then conveys this information through the GLUnix master to the remote daemons which issue the signal appropriately.

Performance measurement of the original GLUnix indicate that signal delivery from the startup process to a remote child takes 6 msec. Adding this group signal extension increases that signal delivery time to 7 msec.

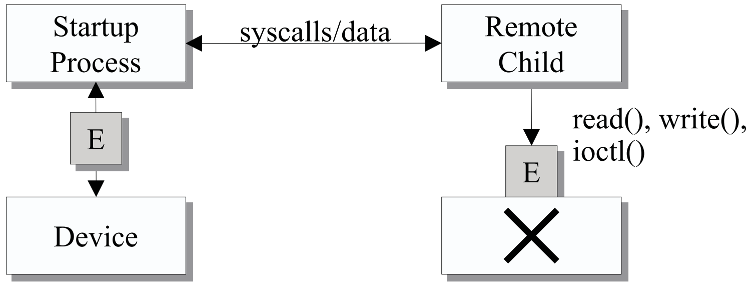


Figure 5.3: SLIC solution to remote device accesses. The extension on the right intercepts system calls accessing the device and forwards those system calls back to the home node. The startup process issues those system calls to the device while the extension on the left intercepts the `copyin()/copyout()` accesses of the application by the device. This information is used to copy the appropriate buffers to and from the remote node.

5.3 Remote Device Access

In order to provide remotely executing jobs with access to devices attached to the user's home node, GLUnix must be able to intercept not only the system calls issued to the device, but must also be able to intercept memory accesses from the device to the application's address space, redirecting those accesses to the address space of the remotely running process.

The key strategy which enables SLIC to solve this problem is to not even attempt to interpret the semantics of the `ioctl()` system call, but to indirectly identifying the locations and sizes of the relevant application buffers by intercepting the kernel copy routine accesses to the application. By effectively intercepting the device's accesses to the process's address space during the execution of the `ioctl()` system call, SLIC can precisely determine the locations and extents of the buffers needed by the device. Once this is known, the system can copy the relevant data to or from the remote process, effectively providing transparent remote access to the device.

This is accomplished with SLIC using two extensions, as depicted in Figure 5.3. The first extension is loaded on the remote node and intercepts any `read()`, `write()`, or `ioctl()` system calls issued to a device. The second extension resides on the home node, monitoring invocations of the `copyin()/copyout()` routines to the startup process. Intercepted system calls are sent from the remote node to the startup process on the home which reissues those system calls. Intercepted copy invocations are then handed to the startup process which allocates space for the accessed buffer. In the case of read accesses, the startup process then sends a request to the extension on the remote node which sends the requested portion of the remote process's address space back to the home node; the kernel copy operation on the home node is then permitted to continue. In the case of a write access, the operation proceeds immediately and the startup process is notified once the operation is complete. Data written by the device as well as the original target address are then sent by the startup process to the extension on the remote node which writes the data into the original target location. Finally, the original system call issued to the device

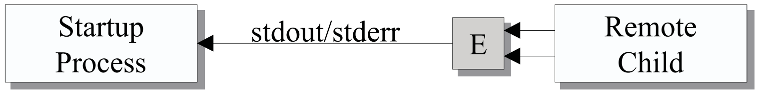


Figure 5.4: SLIC solution to increase the scalability of GLUnix. An extension is loaded on the remote node, monitoring standard I/O for the remote process. `read()` and `write()` operations are intercepted and multiplexed on to a single I/O stream which is demultiplexed by the startup process on the home node.

on the remote node is allowed to return. Note that `read()` and `write()` system calls are trivially handled by the same mechanism as `ioctl()` and do not require special treatment. Performance measurements indicate than an `ioctl()` call resulting in a read of 36 bytes takes an average of 0.1 msec on a single node and 90 msec across a shared 10 Mbit/sec Ethernet due to the multiple handshakes of the implementation.

5.4 Scalability

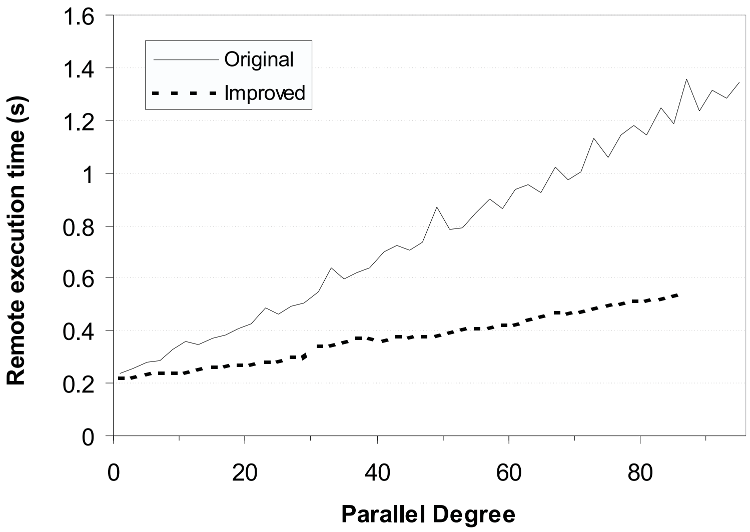


Figure 5.5: GLUnix scalability resulting from SLIC extension. The upper line shows the original scalability of GLUnix while the lower line shows the scalability of GLUnix using SLIC to multiplex I/O onto a single stream.

Section 2.6.2 demonstrates that the limitation in the scalability of GLUnix is the establishment of I/O connections between the remote program and the startup process. To transparently provide existing applications with I/O, GLUnix uses TCP/IP socket connections which are accessible using the standard file descriptor interface. In order to increase the scalability and performance of GLUnix, input and output from the remote child must be transparently intercepted and multiplexed onto a single stream.

The solution to this problem using SLIC, depicted in Figure 5.4, does not suffer from any of the above problems. A small extension is loaded on the remote node. The GLUnix daemon, during initialization of this remote process, creates a single I/o connection to the startup process and marks the new child process for tracing by this extension. The extension monitors system calls such as `open()`, `close()`, `dup()`, and `fcntl()` to keep track of those file descriptors which are associated with `stdout` and `stderr`. Whenever a `write()` system call is performed on one of those file descriptors, the extension writes the tuple <*type*, N, *data*> to the output connection, where *type* indicates `stdout` or `stderr`, N is the number of bytes of data written, and *data* is the output data written by the process.

Figure 5.5 shows the improvement in GLUnix scalability with this optimization as compared to the original scalability. The upper line shows the original GLUnix scalability using three separate I/O connections per child while the lower line shows the measured scalability of GLUnix using a single connection per child. The slope of the upper scalability line is 11.8 msec/node, while the slope of the lower line is 3.8 msec/node, showing a speedup of 3.1, which correlates very closely with the factor of 3 improvement that we would predict from reducing the number of connections from three per node to one.

Since this extension is located in the kernel, it performs the synchronization among multiple processes in a safe manner, preventing faulty or malicious user applications from denying output service to other processes. Because invoking the SLIC extension has low overhead and does not involve context switches, it has a negligible impact on I/O performance.

5.5 Summary

As described in chapter 3, the previous solutions to the above limitations of GLUnix are convoluted, *ad hoc*, inefficient, or not guaranteed to work for all applications. In contrast, by providing third-party extensions with protected, efficient access to the appropriate kernel interfaces, SLIC provides a general-purpose architecture for solving the above problems in a way that is simple, has low overhead, and works for all applications.

Chapter 6

Extension Composability

We have now demonstrated how trusted extensions loaded in the kernel using SLIC can be used to overcome the limitations of a user-level cluster system such as GLUnix without modifying the operating system to implement the necessary functionality. However, the ability to stack multiple, general-purpose, third-party extensions on a single interface raises the potential of extension conflicts. This chapter examines some of the historical problems with extension composability, classifies those problems, and describes a number of heuristics which can alleviate the problem considerably.

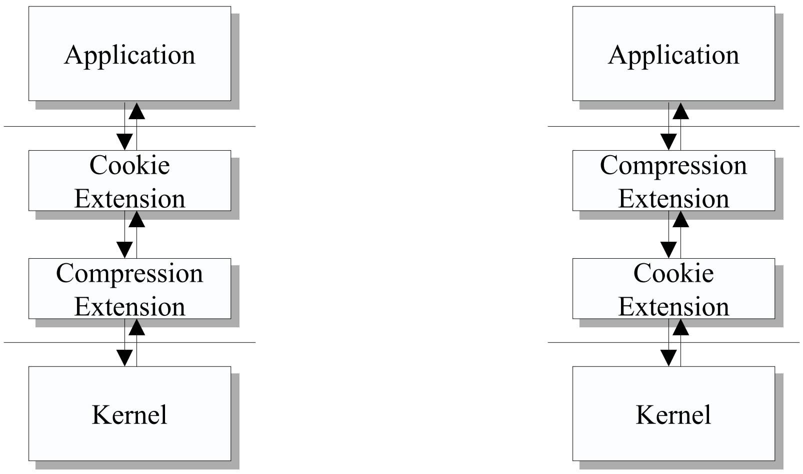
6.1 The Problem

Designers of monolithic operating systems must place careful thought into designing a set of functionality that is mutually consistent and operates together seamlessly. Systems which provide support for composing general-purpose, third-party extensions do not enjoy these properties. In fact, as independent extensions are stacked on a single interface, conflicts can abound, as classically demonstrated in the conflicts encountered on the IBM PC [SMK+90] and the Apple Macintosh [KR92].

Systems which provide extensibility but not stackability, such as the original `vnode` file system [Kle86] or VINO [SS94b, Sma98], do not encounter the conflicts discussed here—because individual extensions (e.g., file systems) exist "side-by-side" in the system, they do not affect each other, making them independent and oblivious of each other's functionality. Only systems that enable extensions to stack on top of one another to compose their functionality (such as newer `vnode` systems supporting stackable file systems [Ros90, KN93, HP95, MBKQ96]) encounter these conflicts.

6.1.1 Stackable File Systems

Prior work in stackable file system has described some of the problems of stacking for the specific case of file systems. Rosenthal [Ros90] recognized that caching in a file system stack could only be safely done on read-only file systems, although no explanation was given as to why this may be true. Later efforts in Spring [KN93] and by Heidemann [HP95, Hei95] recognized that providing the ability to cache data for read-write file systems is crucial to



(a) Proper Extension Ordering (b) Improper Extension Ordering

Figure 6.1: Example of ordering conflict. In (a), the Cookie Extension is able to properly inspect the cookie at the beginning of the file in its uncompressed form, while in (b), the Compression Extension will compress the data first, preventing the Cookie Extension from functioning properly.

performance but demonstrated that uncoordinated access to data cached at different layers in the file system stack can result in stale data or data loss. For example, a file system layer low in the stack may provide hard links for files, causing multiple file names to reference the same blocks of data. File systems located higher in the stack which are unaware of this aliasing and which cache blocks for the file can create data inconsistencies. Identifying the names of all hard links to a file in standard UNIX file systems requires scanning the entire directory structure. Heidemann and the developers of Spring both argue that the exiting `vnode` interface is inadequate to solve this problem; the `vnode` interface must be augmented by coherency operations in order to safely cache data for writable files in a stackable file system. In the description of stackable file systems for 4.4 BSD [MBKQ96], no mention is made of the possibility of conflicts.

In addition to the caching problem described above, there is another potential conflict in stackable file systems which does not appear to be mentioned in the literature. This conflict concerns the ordering of file systems and is illustrated in Figure 6.1. In this example, one extension compresses files while another performs some action based on a cookie stored at the beginning of the file. If the two extensions are ordered as in Figure 6.1a, they will both work properly. If, however, they are ordered as in Figure 6.1b, they will not work correctly since the cookie extension will be unable to properly interpret the compressed file cookies.

6.1.2 TSRs on the IBM PC

System calls on the IBM PC under DOS were accomplished by issuing hardware interrupts (`INT` assembly instructions). These interrupts would be redirected by a RAM-loaded interrupt vector into the BIOS or DOS. Because the interrupt vector table was loaded in RAM and because DOS did not provide memory protection, applications could patch the interrupt vector to redirect interrupts to application code. This was coupled with a system function called *Terminate and Stay Resident* (TSR) which enabled a program to return control to DOS while remaining allocated in memory. Consequently, utilities could extend system functions for arbitrary applications by patching system vectors and "backgrounding" themselves using the TSR function. The PRINT utility that shipped with DOS used this ability to patch the system timer interrupt (among others) in order to provide a background printing service. Third-party developers discovered that by patching the interrupt vector and using the TSR facility, they could insert a variety of functionality on top of the system facilities.

As TSRs (as the extensions came to be known) proliferated, so did conflicts. The book *Undocumented DOS* [SMK+90] cites "hot-key" (special key sequences used to pop up a background utility or otherwise signal an extension) conflicts, the corruption of graphical displays by pop-up windows, memory usage conflicts, and the need to use undocumented DOS features as contributing to the "TSR crisis". There are a number of different types of problems encountered by TSR implementors. TSRs must be careful to read the old value out of the interrupt vector before patching it; those that place their own routine address in the vector and hard-code the address of the original system routine encounter two critical problems: first, the TSR will not work properly with future versions of the operating system or firmware that have different addresses, and second, any TSRs which had previously been loaded on that interrupt vector would be instantly disabled since control would no longer be vectored to their routines. Other TSRs encountered a different problem related to hot-keys. Hot-keys became so popular that different extensions would attempt to intercept the same hot-key sequence. However, only the first interested TSR on the interrupt "chain" would see the hot-key.

Another class of TSRs (possibly to win the hot-key competition) insisted on being linked in at the head of the interrupt chain. These TSRs would intercept the system clock interrupt as well as other system functions. On every clock tick, the TSR would inspect the system vectors to ensure that it's own routine addresses were loaded in the vectors. If they weren't, the TSR would overwrite the vector addresses with its own routine addresses, effectively eliminating whatever interloper TSR had come along and "stolen" the vector. Moving a TSR to the head of the chain in this manner is tantamount to placing an element in a singly-linked list at the head without being able to traverse the list—elements in the list are effectively deleted. The linked list cannot be traversed because each TSR stores the address of the next routine in the chain at an arbitrary location; extensions cannot know where these "next" pointers are stored. This inability to traverse the TSR chain likewise means that TSRs cannot unlink themselves from the chain—once loaded they must remain loaded. Moving an extension to the head of the chain in this manner, however, can wreak havoc on the system. The interloper TSR that is now bypassed may have been a disk or memory compression utility, for example; by effectively removing the compression TSR from the interrupt chain, compressed data becomes unusable. Clearly, the system will become unstable as soon as two or more TSRs insist on being first in the chain.

Finally, because DOS was not a re-entrant system, in order for a TSR to safely make use of DOS features from within an interrupt handler, complicated synchronization at the boundary to DOS was required; if not done properly, race conditions could be introduced, leading to system instability. Of course, it is difficult to determine if a TSR had a race condition. Further, since the system idle loop was located inside of DOS, it was necessary for a TSR to use undocumented system functions in order to use DOS functionality (such as file I/O) safely [SMK+90].

In summary, conflicts among TSRs in DOS arise due to hot-key contention, improper chaining, improper synchronization when calling into DOS, as well as the ordering conflicts described above for stackable file systems. Resolving these conflicts on the IBM PC involved a trial-and-error approach of rebooting, loading a new combination of extensions, and testing.

6.1.3 The Macintosh Operating System

The Macintosh operating system was designed to use interposition to upgrade the system [Cob96]. Early Macintosh systems were shipped with the operating system stored in ROM. In order to install patches and upgrades without replacing the ROMs, all system services are accessed through jump tables stored in RAM. These trap vectors can be patched to vector system calls to RAM-loaded patches which will, if necessary, pass the call back to the original function. System functions on the Macintosh are invoked by issuing illegal instructions in the 68000 processor. The family of illegal instructions used had a hex value of "A" in the upper 4 bits which led to the name "A-Trap" or "A-Line". There were, however, only a limited number of illegal instructions and expanding the number of system functions required multiplexing a variety of functions on a single illegal instruction, leading to double and sometimes triple indirection. This had the unfortunate consequence that patching a single system function required intercepting the illegal instruction vector for a whole set of system functions. The interposed code was then responsible for determining if the particular function being invoked was in fact the one being patched. If not, the invocation was passed to the original demultiplexing routine. This organization was complicated considerably by the fact that there was no standard calling convention followed by all system functions: arguments were passed in registers, on the stack, or a combination of both; return values were placed in registers, on the stack, or in condition codes; register volatility conventions also varied across system calls. Consequently, patching system functions was considerably complex and error-prone and many extensions did not work together correctly.

While the patching ability was at first intended for use only by the Macintosh system developers, word of the functionality spread and other software vendors began to take advantage of it. Consequently, the number of patches increased significantly. A list of patches from a randomly-selected graduate student's Macintosh shows a total of 145 system events being intercepted by 22 patches ranging from system upgrades to commercial applications.

In addition to the conflicts encountered with extensions on the IBM PC, Macintosh extensions have an additional conflict to contend with. Some bugs in the Macintosh system routines are patched in a very peculiar fashion. Because some faulty routines are so large, instead of consuming considerable memory by replacing the faulty routine, these extensions intercept one or more of the smaller routines called by the faulty routine. The patch code then inspects the return value to determine if it was invoked from the faulty routine or from some other location; if invoked from the faulty function, the patch modifies registers or previous stack frames as necessary to compensate for the bug in the faulty routine. These patches are known as "come-from" or tail patches [KR92]. A problem occurs if another extension interposes on the tail patch—when the tail patch inspects the return address, instead of seeing the address at the faulty system routine, it will see the address of the other extension and will fail to correct the system bug. Since the set of system patches is dependent on the version of the operating system, extension writers cannot know *a priori* which system functions are safe to patch and which are not. The order in which extensions are loaded is critical. If the patch to fix the system bug is loaded first, then it will see the proper return address and correct the bug as intended. Only if some other extension is loaded first is there a problem.

These problems and the plethora of patches gave rise to a great number of patch conflicts on the Macintosh. In fact, there are on-line databases of known conflicts encompassing a broad range of software[1]. Third-party tools were also developed, such as the Conflict Catcher by Casady and Greene Software Products, which assist the user in identifying and resolving conflicts. These tools do not understand the nature or arbitrary patches or of conflicts. They simply work by consulting databases of known conflicts or by using a trial-and-error approach, disabling various combinations of patches until the conflict goes away. While these tools can automate the trial-and-error process for extensions which crash the machine during startup, many conflicts require users to manually trigger the conflict for each new combination of patches, a time-consuming task. One source of information on Macintosh conflicts states explicitly that fixing conflicts between system extensions involves experimentation and recommends disabling extensions until the culprit is identified [Poo97].

6.2 Sources of Conflict

There are two sources of conflict in extensible systems: syntactic, or mechanism conflicts, and semantic, or meaning conflicts. *Syntactic conflicts* arise when the mechanism of intercepting events or chaining extensions together in a stack are not done properly, as in the case of extensions dynamically moving themselves back to the head of the extension chain or not saving the old vector address before storing a new address for an extension function. Syntactic conflicts arise in systems which rely on extension developers to do the stacking manually—different developers make different, incompatible assumptions about extension stacking, causing extensions to interfere with each other; for example, different developers may make different assumptions about holding system locks that lead to system deadlock.

Semantic conflicts, however, are due to incompatible functionality which two extensions implement. For example, two extensions which both consume the same hot-key; both extensions cannot stake exclusive claim to the same key sequence. Another example occurs when two extensions attempt to save state in the same file, each corrupting the other's data. These are examples of a subclass of semantic conflicts called *inherent* conflicts. There is a second subclass of semantic conflict which we call *ordering* conflicts.

[1] See `http://www.mac-conflicts.com/` or `http://www.quillserv.com/www/c3/c3.html`.

These conflicts arise when two extensions operate as intended under one stack ordering, but operate improperly under another, as illustrated by the Compression and Cookie file system extensions in Figure 6.1. As a second example of an ordering conflict, consider a file encryption extension and a file compression extension. Compressing the original file data rather than the encrypted file data will likely produce a better compression ratio; in this case the extension will execute correctly, but less efficiently.

There are other sources of potential conflicts when third-party extensions are stacked on an interface. For example, an extension which catches both system call and signal events may deadlock in certain circumstances. If the extension receives and system call, acquires a lock, and then issues a system call which results in a signal being sent to the process, if the extension attempts to acquire the same lock again on receipt of the signal, it will deadlock. More complicated examples can occur when two extensions interact to cause the deadlock. For purposes of this discussion, we will only focus on the syntactic and semantic conflicts as defined above.

6.3 Syntactic Conflict Solutions

Of the two main classes of conflicts, syntactic conflicts are the simplest to resolve. Systems which do not rely on end developers to properly manage the interposition and chaining, but rather hide all such implementation details in the system will not suffer from syntactic conflicts. For example, if extension order can be dynamically changed, the system will manage the extension chain to guarantee that list consistency is maintained and no extensions are inadvertently disabled. In the case of SLIC, system locking conventions are handled by the SLIC dispatchers, relieving extension developers from both the burden of lock management as well as the possibility of system deadlock. Systems such as SPIN, VINO, and SLIC as well as the stackable file systems of Spring and 4.4 BSD do not suffer from syntactic conflicts.

6.4 Semantic Conflict Solutions

As of this writing, no commercial or research system provides a general solution, or even automated heuristics, to this problem. The stack ordering of file systems in the Spring [KN93] and 4.4 BSD [MBKQ96] file systems is left entirely to a system administrator. SPIN extension ordering is controlled by the module implementing a system event [BSPP+95], but there is insufficient information available to the module to resolve the semantic conflicts discussed here. The system developers of the Macintosh place *a priori* ordering constraints on their own system patches to reduce the likelihood of conflicts [Cob96]. While the Conflict Catcher for the Macintosh can automate the process of identifying extensions which crash the system on startup, it is unable to identify extension conflicts which do not crash the system or which occur only in certain circumstances; identifying these conflicts is done by trial-and-error. However, a system with N extensions has $N!$ possible extension combinations; identifying an operable ordering by trial and error can be impractical for more than a few extensions. Even identifying two extensions which conflict can take polynomial time.

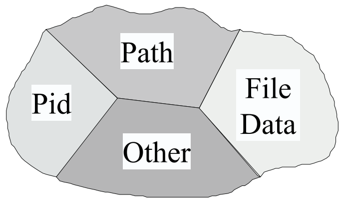


Figure 6.2: Hypothetical example of event domains. Event domains are defined such that extensions intercepting events in separate event domains cannot conflict; extensions intercepting events in the same event domain may conflict.

In fact, the general problem of determining if a set of extensions will experience a semantic conflict is undecidable. Consider one extension that writes to a certain file periodically and a second extension that writes incompatible data to the same file when a particular process terminates. If the process terminates, there will be a conflict; if it does not, there will be no conflict. It is not possible for the system to determine at load time if a particular extension will conflict with others.

Although extension stacking systems cannot, in general, determine when a conflict will occur, the heuristics described below can enable the system to know when conflicts *cannot* occur versus when they *might* occur. Conceptually, this process involves examining a known conflict situation and determining if the conflict can be resolved or not. The nature of the conflict must be understood so that the classification system can be revised to enable the system to either resolve the conflict or identify at load time the possibility of a conflict. At a minimum, the heuristics described below will significantly decrease the set of extension combinations that could be causing a particular system error, simplifying trial-and-error investigations.

6.4.1 Inherent Conflicts

Inherent conflicts arise when the functionality of two extensions cannot logically compose. Rather than attempting to resolve these conflicts automatically, we will enable the system to assist the system administrator in identifying and removing one or both of the conflicting extensions. The approach we advocate in attacking this problem is to decompose the possible extension functionality into separate domains which, by design, do not conflict. Each event which can be caught or initiated by extensions is then classified into one or more of the event domains. Finally, extensions are annotated by developers with the set of domains associated with the events caught or initiated by the extension. Extensions with disjoint domain sets cannot conflict; extensions with overlapping domain sets might conflict.

Consider the hypothetical example depicted in Figure 6.2 in which four event domains have been defined: “pid”, “path”, “file data”, and “other”. The `getpid()` system call falls into the domain “pid”, as do the `kill()` and `fork()` system calls. The “file path” domain includes system calls such as `open()`, `chroot()`, `readdir()`, and `exec()`. The “file data” domain includes system calls such as `read()` and `write()`. An extension that only interposes on "file data" system calls cannot conflict with an extension solely concerned with "pid" system calls. Each extension must be annotated with the event domains corresponding to all possible events that it intercepts or initiates. The mapping of events onto event domains must be clearly specified to facilitate accurate annotation.

Note that some system calls fall into multiple domains; for example, `open("/proc/4312")` uses a path which encodes a process identifier. This event therefore properly belongs to both the "pid" and "file path" domains. Unfortunately, with this simplistic classification system, this implies that all extensions which catch the `open()` system call would have to be annotated with both the "pid" and the "file path" domains, decreasing the effectiveness of the classification system. The following paragraph introduces a hierarchical scheme that can alleviate this difficulty and others.

The event domains can be made arbitrarily small. The "other" domain in Figure 6.2 can be refined into "VM", "file descriptor", etc. The "file path" domain can be refined by indicating the first path element of all files accessed by the extension; for example, an extension which only accesses files in "/tmp" would be in domain "path tmp" while an extension that monitors accesses to "/etc" would be in domain "path etc". An extension that may access files anywhere in the file system would have to fall into all of the refined "path" domains. There is a difficulty, however, in refining the event domains over time—extensions annotated and deployed with an older classification system would not have the refined annotations. A hierarchical domain system is a simple way to provide both backwards compatibility and increased expressiveness of the annotations. Returning to our hypothetical event domain example, in a hierarchical system, the original domain decomposition would be "/pid", "/path", "/file_data", and "/other". Extensions which monitor file accesses only in "/tmp" are then of type "/path/tmp", while extensions which access files anywhere are simply "/path". System calls which send signals now become "/pid/signal" while `fork()` and `vfork()` are "/pid/fork"; an extension which virtualizes process identifiers is simply of class "/pid". Older extensions marked with domain "/pid" will continue to work; their annotations will simply not be as informative as they could be. Older extensions annotated with "/other" are understood by the system to include any new categories added to the domain system, such as "/VM" and "/fd".

In this hierarchical system, the `open("/proc/4312")` event mentioned earlier would fall into the domains "/path/proc" as well as "/pid", although an extension that is only interested in `open()` system calls for files in `/tmp` could now simply be annotated as "/path/tmp", eliminating the need to be annotated with class "/pid" as well. Note that this requires that an extension only act on a subset of the `open()` system call events. There are a number of ways to do this. First, the system call dispatcher could be augmented with additional predicates (see Section 4.3.1) to reflect the refined hierarchical classification system, only delivering a subset of `open()` events to the extension. Second, the extension can be provided with a `Slic_Reject()` routine to indicate that an event is not of interest to the extension. Third, a separate routine akin to a *guard* in SPIN [BSPP+95] can be used to determine if the event is of interest to the extension.

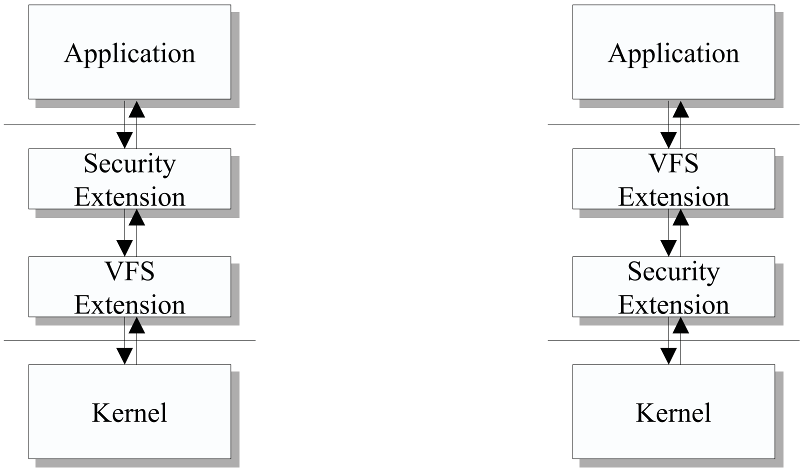
Conceptually, this process of refining the event domains begins with a single event domain, "/other", and proceeds by refining the event domains as needed. The trade-off with fine-grained event domains is that the more narrow the event domains become, the more of them there will be and the more complicated it will be to guarantee that they do not overlap and to annotate extensions.

An interposition system such as SLIC can assist the extension developer in annotating extensions correctly. At a small performance cost, the system can monitor the events captured and initiated by the extension, verifying that the domain annotations of the extension are accurate. While such a testing methodology does not prove the correctness of the annotations under all conceivable situations, it can be an invaluable verification aid in practice. This mechanism could also be coupled with code path verification tools to identify those code paths which had not been exercised by the developer's test.

6.4.2 Ordering Conflicts

The problem of ordering conflicts can be attacked using heuristics similar to the event domains described above. In this case, instead of annotating extensions with a description of what events the extension uses, extensions are annotated with a description of what they do with captured events. First let us consider the ordering conflict illustrated in Figure 6.1. Another example which exhibit the same behavior are the compression and encryption extensions described in section 6.2. A third example is an extension which sends files written to the "/printer" to the printer coupled with a Virtual File System extension; the Printer extension should be loaded above the VFS extension so that the "/printer" path name is interpreted at the virtual file level as seen by the application rather than at the physical file level. In each of these examples, there is an extension which desires to see the system call data as directly supplied by the application, without any transformations being applied by other extensions: the cookie extension must inspect the unmodified file data, the compression extension wishes to compress raw application data rather than encrypted file data, and the printer extension wishes to observe virtual file paths delivered straight from the application. To enable these extensions to receive "raw" application data, we will add a second dimension to the event domains developed above—each event domain associated with an extension will have two booleans attached to it: one to indicate if the events received must be received unmodified from the application and a second to indicate whether the events issued by the extension are also unmodified. The Compression extension would then have the two boolean values <TRUE, FALSE> to indicate that it wishes to compress unmodified file data, but that it does not maintain the data in its unmodified form. The Cookie extension, however, would be marked <TRUE, TRUE> since it does not modify the file data, but only inspects it. The system can then construct a chain of extensions by placing all <TRUE, TRUE> extensions at the top of the chain, nearest the application, followed by at most one <TRUE, FALSE>, and then as many <FALSE, TRUE> or <FALSE, FALSE> extensions as remain. Note that there can only be a single extension in the system which requires unadulterated input but which produces adulterated output. If two such extensions exist, the system can immediately mark the two as representing a possible conflict to the system administrator. These boolean annotations should be specified per-event domain so that the extensions which modify independent event domains do not have to be ordered with respect to each other; for example, two <TRUE, FALSE> extensions can peacefully coexist in the system as long as they access different event domains.

A different class of ordering problems occurs with extensions which enforce security



(a) Proper Extension Ordering (b) Improper Extension Ordering

Figure 6.3: Example of an ordering conflict which violates security guarantees enforced by an extension. In (a), the Virtual File System (VFS) extension issues system calls which are not inspected by the Security extension, possibly enabling security guarantees to be bypassed, while in (b), all system calls issued by the VFS extension pass through the Security extension, thus ensuring that the security guarantees will be maintained.

guarantees, as illustrated in Figure 6.3. In order to guarantee that security policies enforced by extensions are maintained in the presence of other extensions in the system, extensions which may modify events or issue new events must be stacked *above* all security extensions in the system. This stacking constraint ensures that all events pass through the security extensions before being executed by the underlying kernel. To encode this information, we introduce an *extension class* as a third dimension to extension annotations. Rather than indicating the events or type of data that an extension intercepts, as the previous two dimensions indicate, the extension class characterizes what the extension does with events. As with event domains, we will start with the class "other" and refine that class to reflect observations concerning certain extension conflicts. To handle security extensions, we introduce the extension class "security", indicating than an extension enforces some security guarantee on events. In order to avoid the security conflict in Figure 6.3a, we must adopt the constraint that all extensions in the "security" class must come at the end of the chain. This may still be insufficient in some cases. Security extensions themselves may also initiate new events, raising the possibility that a security extension lower in the chain may initiate an event which violates the security policy of an extension higher in the chain. To identify these potential conflicts, we refine the "security" class into "security, initiate" and "security, noinitiate". A security extension which only passes events through unmodified or aborts illegal events would be classified as "security, noinitiate", while others would be "security, initiate". Any number of "security, noinitiate" extensions may now be safely stacked at the end of the chain. At most one "security, initiate" extension in each event domain may precede the set of "security, noinitiate" extensions. If more than one "security, initiate" extension exists for a particular event domain, the system can flag them to the system administrator as having a potential semantic conflict.

Now consider an extension that only does tracing and/or logging of events, but does not modify events, initiate events, or deny events. Such an extension can safely be loaded at any level in the extension stack, even below security extensions, because it cannot violate security guarantees if it does not alter or initiate events in any fashion. We will designate such extensions as class "trace". Any number of "trace" extensions can exist for each event domain and can be loaded anywhere in the extension stack.

6.5 Summary

We have evaluated a number of examples of extension conflicts and developed some understanding of the nature of these conflicts. From that understanding, we have proposed a heuristic approach and used it to develop a basic design of a classification system which can be used to automate the process of identifying and resolving conflicts. Syntactic conflicts can be entirely eliminated by hiding details of extension chaining from extension developers. Although we do not attempt to resolve inherent semantic conflicts, the event domain annotations make these conflicts more manageable, assisting a system administrator in identifying sets of potentially conflicts extensions. Many ordering semantic conflicts can be handled automatically by the system using the extension classification system coupled with the event domain system.

Chapter 7

Conclusion

Traditional approaches for modifying operating system functionality are inadequate to meet the crucial need for operating system developers to be able to quickly respond to new user requirements, recent operating system research, and security breaches. Rewriting the operating system from scratch or modifying an existing operating system both require substantial implementation effort and result in systems which are often not widely available after a few years. The size and complexity of operating systems prevent either of these approaches from being capable of quickly responding to time-critical needs such as fixing widely-publicized security problems. Constructing new features at the user level enables the rapid, portable implementation of new features, but suffers from a number of limitations in the areas of protection, enforcement, and efficiency.

Prior efforts to address these problems by providing general-purpose extensibility fall into two categories, each with significant limitations. A number of projects have restructured or otherwise modified the operating system to provide extensibility in the kernel; while demonstrating the advantages of extensibility, the in-kernel approach of these projects coupled with the complexity of existing commodity UNIX operating systems effectively prevent these solution from being applied to existing operating systems in practice. The second category of extensibility projects provide extensibility using only user-level means such as dynamic libraries or binary patching; these approaches, however, suffer from problems of protection, efficiency, or lack of access to necessary kernel interfaces.

A new method is needed for extending operating system functionality which requires few or no modifications to existing commodity operating systems, yet enables trusted, third-party extensions to be protected from applications, enforced on applications, and have low overhead.

7.1 Research Contributions

To investigate these issues, we have a focused on the functionality needed to transparently support cluster features, using GLUnix as a case study. We have shown in a non-trivial, working system what functionality is needed to effectively harness cluster resources and what underlying abstractions are needed to support that functionality. In order to harness remote cluster resources transparently for existing applications, GLUnix must be able to intercept a number of interactions between applications and the kernel (such as terminal I/O, signals, and device accesses) and add a level of virtualization to those interactions, forwarding them to remote nodes as necessary. This system, GLUnix, has been very successful and has serviced over 160 users running more than two million jobs in the two and half years since it was deployed. The source code has also been downloaded by over 200 sites world-wide and is currently being ported to a number of different operating systems.

Using GLUnix as a case study, we identified a number of limitations that user-level systems encounter in providing cluster services. While constructing such cluster services at the user level provides numerous advantages, simplifying the implementation and maximizing portability, user-level solutions encounter a number of limitations. These limitations are not due to the implementation details of user-level systems such as GLUnix, but rather to the fact that UNIX does not provide sufficient functionality at the user level to virtualize its abstractions to the cluster. These limitations effectively prevent GLUnix and other user-level cluster systems from offering a truly transparent cluster view to the user. The limitations include lost input to remote applications, improper handling of group signals, inability to access remote devices efficiently, and limitations on scalability and coscheduling.

To overcome these limitations, we developed a powerful, general-purpose extension technology which enables kernel-level insertion of trusted extensions on existing kernel interfaces with little or no modifications. We presented and evaluated SLIC, a prototype of this extension technology, and have demonstrated the wide range of possible functionality through a number of example extensions. The key implementation challenges in developing SLIC were extension state management and modifying system call buffers, both of which were overcome using the concept of interposition.

We further demonstrated how GLUnix can use this technology to overcome the limitations of its user-level construction, focusing on the problems of lost input, group signals, remote device access, and scalability. By downloading relatively small extensions into the kernel using SLIC, the vast majority of GLUnix can remain at the user level while removing the limitations.

Finally, we investigated the causes of conflicts in prior extensible systems, the IBM PC and Macintosh, and have classified those conflicts. We proposed a heuristic to attack this problem and used it to develop a preliminary classification system that further enables SLIC and other extension systems to reduce the complexity of, and in many cases solve, the conflicts the arise when multiple extensions from independent, third party developers are loaded into a system. While the extension classification system proposed here is a useful starting point, to be useful in practice, further development of the classification system must be steered by real-world conflicts. Unfortunately, conflicts are often perceived as failures or negative results and so are not widely publicized, making analysis difficult.

There are number of consequences of this work. Prior to this writing, all general-purpose migration packages have been in-kernel solutions, depending heavily on operating system modifications [DO91]. Packages developed without kernel modifications are functionally restricted and require application recompilation [BLL91]. For the first time, SLIC opens the door of possibility for a migration package which is general-purpose, has low overhead, but does not require heavy modification to the operating system. Such a package would provide a qualitative improvement to current network of workstation implementations.

Prior to this work, the ordering of extensions and the identification of conflicts in extensible systems was left entirely to the system administrator or knowledgeable users. Using the heuristics and classification system we have proposed, these systems may now achieve a qualitatively higher level of automation for this task. By providing system support for identifying possible conflicts and resolving many ordering conflicts, the process of resolving conflicts can be substantially simplified and problems arising due to human error can be significantly reduced.

7.2 Future Work

There are a number of areas of future work which remain to be explored following this work. In addition to implementing the remaining GLUnix extensions, an interesting project would be to use SLIC concepts to interpose on and extend the virtual memory interface of UNIX. Virtual memory extensions would not only be able to support transparent remote execution for applications relying on Sys V shared memory segments, but would also enable a NOW to support parallel applications based on the distributed shared memory model. The low overhead of the SLIC interposition mechanism provides the possibility of significant performance improvements over user-level approaches to this problem such as Treadmarks [ACD+96] and less extensive kernel changes than systems such as Hive [CRD+95].

Given the large set of common functionality supported by the majority of UNIX operating systems today, it should be possible to construct a light-weight portability interface layer to enable SLIC extensions to run unmodified on multiple operating systems. Of course, some advanced features of each operating system may not be easily portable, but a portability layer could enable kernel-level SLIC extensions to enjoy the same level of portability that user-level applications do today. Such a portability layer would be advantageous in deploying SLIC and gaining a user community and would facilitate the free exchange of features across operating system communities.

More work remains to be done in the area of extension conflicts. Actual extension classification systems must be developed, tested with realistic extensions and applications, and refined. Doing this successfully requires a large set of extensions from which to work, leading to a kind of "chicken and egg" problem. Although the popularity of DOS and TSRs has dropped dramatically with the advent of Windows, the Macintosh operating system is still in general use and has a rich set of existing extensions with known conflicts which can be leveraged as a starting point. Although the classification system developed for the Macintosh would not transfer directly to other systems such as UNIX because of dramatic differences in the events, the Macintosh environment could be used to test the classification approach against known conflicts and to identify other useful heuristics.

One conflict not addressed in this discussion arises when extensions are permitted to add new events to an interface. The definition of interposition used in the SLIC prototype is one in which the interfaces above and below an extension are symmetric, preventing new events from being added to an interface. However, the ability to add new events can be useful; for example, a file system extension may add an "undelete" operation. The conflict arises when other extensions in the system, such as security extensions, were implemented based on a given event interface and do not know how to interpret the semantics of the new event. An "undelete" operation may permit a malicious application or user to circumvent the policies of a security extension. One approach to this problem is to use an object-oriented approach similar to the toolkit used in Interposition Agents [Jon93]. By translating events to object and method invocations, a security extension could monitor file system activity by extending the file path class. The "undelete" operation, although a new event, would still make use of file paths which are understood by the security extension. By translating events to abstractions, extensions are then written against a set of abstractions regardless of the events which manipulate them. Of course, in some sense, this just pushes the problem back to another level—what if an extension wishes to add a new abstraction to the system? For example, the abstraction of terminal sessions did not exist in early versions of UNIX, but were added later to improve the semantics of job control through a terminal. It is unclear how extensions could introduce new abstractions into the system without introducing a new class of semantic conflicts.

Finally, a deeper understanding of the power and limitations of interposition can be gained by restructuring the operating system around the concept of extension via interposition, although it is unlikely that such a system would enter mainstream use. A key element to the success of NOWs is their use of interchangeable commodity components. By using standardized, interchangeable components, NOWs can be customized to a particular task and can also be assembled very rapidly which increases the rate of technology advancement. By using commodity components, NOWs leverage many of the benefits of marketplace competition, such as a price/performance ratio that is superior to MPPs and supercomputers. Currently, the internal components of operating systems are not interchangeable because of significant implementation differences across systems. Consequently, individual operating system components cannot benefit from marketplace competition. Recently, some operating systems have begun modularizing certain system components such as the scheduler, allowing them to be replaced independent of the rest of the system. These systems, however, typically do not support module stacking, preventing new functionality from leveraging existing functionality. Many of the key benefits of the NOW architectural paradigm can be brought to bear on operating systems by restructuring the operating system around the concept of stackable, interchangeable components, placing the bulk of the operating system policies and functionality in extensions supporting standardized interfaces.

7.3 Closing Remarks

We have demonstrated that significant new operating system functionality can be efficiently implemented in existing commodity operating systems by third-party developers with little or no modification to kernel sources. Adopting this method for implementing future operating system features can revolutionize the field of operating systems. Future operating systems can provide basic, rudimentary services with advanced features supplied by independent software vendors. The resulting competition of operating system features will provide qualitative improvements in the operating system functionality available to end users. Users will be able to easily customize their operating system to a particular task by downloading the necessary extensions. Researchers will be able to directly transfer in-kernel operating system features to other research groups as well as to end users.

Bibliography

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 USENIX Summer Conference*, pages 93–112, June 1986.

[ABLL92] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Transactions on Computer Systems*, pages 53–79, February 1992.

[ACD+96] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[ACK+95] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve Steinberg, and Kathy Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[ACPtNT95] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[ADADC+97] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-Performance Sorting on Networks of Workstations. In *SIGMOD '97*, May 1997.

[ADN+95] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 109–126, December 1995.

[ADV+95] Remzi Arpaci, Andrea Dusseau, Amin Vahdat, Lok Liu, Thomas Anderson, and David Patterson. The Interaction of Parallel and Sequential Workload on a Network of Workstations. In *Proceedings of Performance/Sigmetrics*, May 1995.

[AE87] Rakesh Agrawal and Ahmed Ezzat. Location Independent Remote Execution in NEST. *IEEE Transactions on Software Engineering*, SE-13(8):905–911, August 1987.

[AISS97] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the Operating System at the User-Level: the Ufo Global File System. In USENIX, editor, *Proceedings of the 1997 USENIX Conference*, pages 77–90, January 1997.

[Ano96] Anonymous. Gigabit Ethernet. *Computer*, 29(2), February 1996.

[BALL90] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight Remote Procedure Calls. In *ACM Transactions on Computer Systems*, pages 37–54, February 1990.

[BCE+94] Brian Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin G. Sirer. SPIN—An Extensible Microkernel for Application-Specific Operating System Services. Technical report, University of Washington, 1994.

[BCF+95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[BCM+70] C. G. Bell, R. Cady, H. McFarland, B. DeLagi, O'Laughlin, J., R. Noonan, and W. Wulf. A new architecture for mini-computers: The DEC PDP-11. In *Proceedings of the AFIPS Spring Joint Computing Conference*, pages 657–675, Reston, VA, 1970. AFIPS Press.

[BDH+94] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. In *Proceedings of the Second International World Wide Web Conference*, pages 763–771, October 1994.

[BDR97] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, October 1997.

[Bir89] Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, January 1989.

[BLL91] Allan Bricker, Michael Litzkow, and Miron Livny. Condor Technical Summary. Technical Report 1069, University of Wisconsin–Madison, Computer Science Department, October 1991.

[BMR82] D. Brownbridge, L. Marshall, and B. Randell. The Newcastle Connection. *Software—Practice and Experience*, 12:1147–62, 1982.

[BN84] Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59, February 1984.

[BP88] Brian N. Bershad and C. Brian Pinkerton. Watchdogs—extending the UNIX file system. *Computing Systems*, 1(2):169–188, Spring 1988.

[Bro94] Kraig Brockschmidt. *Inside OLE 2*. Microsoft Press, Redmond, WA, 1994.

[BS95] T. Bressoud and F. Schneider. Hypervisor-based Fault Tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[BSPP+95] B. N. Bershad, S. Savage, E. G. Sirer P. Pardyak, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[BTC97] Arindam Banerji, John M. Tracey, and David L. Cohn. Protected Shared Libraries — a new approach to modularity and sharing. In *Proceedings of the 1997* USENIX *Technical Conference*, pages 59–76, January 1997.

[BV98] Diane Burns and Sharyn Venit. *The QuarkXPress 4.0 Handbook*. MIS:Press, New York, NY, 1998.

[CB93] J. Bradley Chen and Brian N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 120–133, December 1993.

[CDD+91] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. Technical Report 385, University of Rochester, Computer Science Department, February 1991. Revised May.

[CDG+93] David E. Culler, Andrea Dusseau, Seth C. Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.

[Che88] David R. Cheriton. The V Distributed System. In *Communications of the ACM*, pages 314–333, March 1988.

[Cla85] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[Clu97] Chaos Computer Club. CCC: Microsoft security alert. http://berlin.ccc.de/radioactivex.html, March 1997.

[CMC97] Brent Chun, Alan Mainwaring, and David Culler. Virtual Network Transport Protocols for Myrinet. In *Proceedings of the 5th Hot Interconnects Conference*, August 1997.

[Cob96] Jeff Cobb. Personal communication, November 1996.

[Com92] Apple Computers. *Inside Macintosh, Macintosh Toolbox Essentials*. Addison-Wesley, 1992.

[CRD+95] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 12–25, December 1995.

[DAC96] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the 1996 ACM SIGMETRICS Conference*, 1996.

[DDHY92] D.L. Dill, A. Drexler, A.J. Hu, and C.H. Yang. Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design: VLSI in Computers and Processors*, October 1992.

[DO91] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software—Practice and Experience*, 21(8):757–85, August 1991.

[dP93] Martin de Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN*. Ellis Horwood Publishers, second edition, 1993.

[DPH92] Peter Druschel, Larry L. Peterson, and Norman Hutchinson. Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 512–520, June 1992.

[Dyc97] Timothy Dyck. Informix slices its way to front. *PC Week*, 14(9):1, March 1997.

[EKJO95] D. R. Engler, M. F. Kaashoek, and Jr J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[FG91] Roger Faulkner and Ron Gomes. The Process File System and Process Model in UNIX System V. In *Proceedings of the 1991 USENIX Winter Conference*, pages 243–252, January 1991.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Steven Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[FR92] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–18, December 1992.

[Fre97] Nigel French. *PageMaker 6.5 Bible.* IDG Books Worldwide, 1997.

[Gar94] Simon Garfinkel. *PGP: Pretty Good Privacy.* O'Reilly and Associates, Sebastopol, CA, first edition, December 1994.

[GM95] James Gosling and Henry McGilton. The Java(tm) Language Environment: A White Paper. http://java.dimensionx.com/whitePaper/java-whitepaper-1.html, 1995.

[GPR+98] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. *Software—Practice and Experience*, 28(8), July 1998.

[GPRA98] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the 1998 USENIX Conference*, June 1998.

[GWTB96] Ian Goldberg, David Wagner, Randy Thomas, and Eric Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[Hei95] John S. Heidemann. Stackable Design of File Systems. Technical Report CSD-950032, University of California at Los Angeles, September 1995.

[HER+95] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of Supercomputing '95*, pages 385–394, Barcelona, July 1995.

[HFG+96] Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian N. Bershad. Language support for extensible operating systems. In *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[HP95] John Heidemann and Gerald Popek. Performance of Cache Coherence in Stackable Filing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 127–142, December 1995.

[IBM74] IBM. IBM Virtual Machine Facility /370 Planning Guide. Technical Report GC20-1801-0, IBM Corporation, 1974.

[Jon93] Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, December 1993.

[JPGH90] Thomas Page Jr., Gerald Popek, Richard Guy, and John Heidemann. The Ficus Distributed File System: Replication via Stackable Layers. Technical Report CSD-900009, Department of Computer Science at the University of California Los Angeles, April 1990.

[KBM+96] Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris MC: A multi computer OS. In *Proceedings of the 1996 USENIX Conference*, pages 191–203. USENIX, January 1996.

[KK92] Eduardo Krell and Balachander Krishnamurthy. COLA: Customized overlaying. In *Proceedings of the 1992 USENIX Winter Conference*, January 1992.

[Kle86] Steven R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 Summer* USENIX *Technical Conference*, pages 238–247, 1986.

[KLS86] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXclusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 4(2), 1986.

[KN93] Yousef Khalidi and Michael Nelson. Extensible File Systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 1–14, December 1993.

[KR92] Scott Knaster and Keith Rollin. *Macintosh Programming Secrets*. Addison-Wesley Publishing Co., second edition, 1992.

[LC96] Steven S. Lumetta and David E. Culler. The mantis parallel debugger. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 118–26, Philadelphia, Pennsylvania, May 1996.

[Let88] Gordon Letwin. *Inside OS/2*. Microsoft Press, 1988.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LHFL93] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeffrey Law. In-Kernel Servers on Mach 3.0: Implementation and Performance. In *Proceedings of the 3rd Unsenix Mach Symposium*, April 1993.

[Lit87] M. Litzkow. Remote Unix: Turning Idle Workstations into Cycle Servers. In *Proceedings of the 1987 USENIX Summer Conference*, pages 381–384, June 1987.

[LS92] M. Litzkow and M. Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proceedings of the 1992 USENIX Winter Conference*, pages 283–290, January 1992.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent Executable Editing. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, volume 30(6), pages 291–300, June 1995.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[MC96] Alan Mainwaring and David Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report CSD-96-918, University of California at Berkeley, October 1996.

[McC97] Deke McClelland. *Photoshop 4 Bible*. IDG Books Worldwide, 1997.

[MRC+97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the Performance of Log-structured File Systems with Adaptive Methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[Net94] Netscape Communications Corporation. *Netscape Navigator*, 1994. http://-www.netscape.com.

[NL96] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 229–243, October 1996.

[NL97] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 184–197, October 1997.

[OCD+88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[Ous82] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[Ous90] John Ousterhout. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proceedings of the 1990 Summer* USENIX *Conference*, pages 247–256, June 1990.

[Ous96] John Ousterhout. Why Threads Are A Bad Idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference, January 1996. See http://-www.sunlabs.com/˜ouster.

[PB96] Przemysław Pardyak and Brian N. Bershad. Dynamic binding in an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 201–212, October 1996.

[PBKL95] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the 1995 USENIX Summer Conference*, pages 213–223, January 1995.

[PD96] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufman, 1996.

[Poo97] Lon Poole. *Mac OS 8 Bible*. IDG Books Worldwide, 1997.

[PPTT91] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, a distributed system. In *Proceedings of the Spring 1991 EurOpen Conference*, pages 43–50, May 1991.

[PWC+81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 169–177, December 1981.

[RAC97] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local area communication with fast sockets. In *Proceedings of the 1997 USENIX Conference*, pages 257–274, January 1997.

[Ras86] Richard F. Rashid. From RIG to Accent to Mach: The Evolution of a Network Operating System. In *Proceedings of the Fall 1986 Joint Computer Conference*, pages 1128–1137, 1986.

[Ros90] David S. H. Rosenthal. Evolving the Vnode Interface. In *Proceedings of the 1990 USENIX Summer Conference*, pages 107–117, June 1990.

[RT74] Dennis M Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, 1974.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, October 1997.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christoper Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[SGK+85] R. Sandberg, D. Goldberg, Steven Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985* USENIX *Technical Conference*, pages 119–130. Sun Microsystems, June 1985.

[Shi98] Ken Shirriff. Personal communication, January 1998.

[SKS90] David C. Steere, James J. Kistler, and M. Satyanarayanan. Efficient User-Level File Cache Management on the Sun Vnode Interface. In *Proceedings of the 1990 USENIX Summer Conference*, pages 325–331, June 1990.

[Sma98] Chis Small. Personal communication, May 1998.

[SMK+90] Andrew Schulman, Raymond J. Michels, Jim Kyle, Time Paterson, David Maxey, and Ralf Brown. *Undocumented DOS*. Addison-Wesley, 1990.

[SS94a] Christopher Small and Margo Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard, October 1994.

[SS94b] Christopher Small and Margo Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard, October 1994.

[SSP+96] Emin Gün Sirer, Stefan Savage, Przemysław Pardyak, Greg DeFouw, Mary Ann Alapat, and Brian N. Bershad. Writing an operating system using Modula-3. In *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[Sun87] Sun Microsystems. XDR: External Data Representation Standard. Technical Report RFC-1014, Sun Microsystems, Inc., June 1987.

[Sun90] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[Tea95] Computer Emergency Response Team. Ghostscript Vulnerability. CERT Advisory CA-95.10, CERT, August 1995.

[Tea96a] Computer Emergency Response Team. Vulnerability in expreserve. CERT Advisory CA-96.19, CERT, August 1996.

[Tea96b] Computer Emergency Response Team. Vulnerability in Solaris admintool. CERT Advisory CA-96.16, CERT, August 1996.

[Tea96c] Computer Emergency Response Team. Vulnerability in WorkMan. CERT Advisory CA-96.23, CERT, October 1996.

[Tea97] Computer Emergency Response Team. Vulnerability in talkd. CERT Advisory CA-97.04, CERT, January 1997.

[The93] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883, November 1993.

[The94] Marvin Theimer. Personal communication, June 1994.

[TLC85] Marvin Theimer, K. Landtz, and David Cheriton. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[Vah96] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth SOSP*, pages 40–53, Copper Mountain, CO, December 1995.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.

[VLG95] J. P. Verboncoeur, A. B. Langdon, and N. T. Gladd. An object-oriented electromagnetic PIC code. In *Computer Physics Communications*, pages 199–211, 1995.

[WB72] W. Wulf and C. Bell. C.mmp — A multiminiprocessor. In *Proceedings of the AFIPS Fall Joint Computing Conference*, pages 765–777, Reston, VA, 1972. AFIPS Press.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–344, June 1974.

[Web93] Neil Webber. Operating System Support for Portable Filesystem Extensions. In *Proceedings of the 1993 USENIX Winter Conference*, pages 219–228, January 1993.

[Win97] Nick Wingfield. ActiveX used as hacking tool. `http://www.news.com/News/Item/0%2C4%2C7761%2C00.html`, February 1997.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[WLS+85] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun Network File System. In *Proceedings of the 1985 USENIX Winter Conference*, pages 117–124, January 1985.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, 1994.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In

Proceedings of the 11th ACM Symposium on Operating Systems Principles, pages 63–76, November 1987.

[Zac94] G. Pascal Zachary. *Showstopper! The Breakneck Race to Create Windows NT and the Next Generation at Microsoft.* Macmillan, Inc., 1994.

[Zho92] Songnian Zhou. LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems. In *Proceedings of the Workshop on Cluster Computing*, December 1992.

[ZRB+93] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, FaraMarz Rabii, and Durriya Netterwala. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the 1993 USENIX Winter Conference*, pages 449–468, January 1993.

[ZWZD92] Sognian Zhou, Jingwen Wang, Xiaohn Zheng, and Pierre Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems. Technical Report CSRI-257, University of Toronto, 1992.

Appendix A

Example SLIC Extensions

This Appendix provides some examples of SLIC extensions as a means of demonstrating the utility and power of the system.

A.1 Print Extension

This example demonstrates a trivial extension that merely prints the name of each system call issues by a given set of processes; when an `open()` system call is received, the extension additionally prints out the `pid` of the calling process.

Figure A.2 depicts the text of the sample extension, "Print". Lines 1–4 of this extension simply include the necessary SLIC headers. Lines 6 and 7 are needed by SLIC to

```
struct Slic_SyscallInfo {
       int syscallNum;
       int args[];
};

struct Slic_ReturnInfo {
       bool forceReturn;
       int  returnValue;
       int  errno;
};

Slic_SyscallRegisterExtension(int dispatcherId);
Slic_SyscallRegisterHandler(int syscallNum,
            void (handler)(Slic_SyscallInfo *sysInfo,
                           Slic_ReturnInfo *retInfo));
Slic_TraceProc(int pid, bool traceAllChildren);
Slic_UntraceProc(int pid, bool untraceAllChildren);

Slic_IssueSyscall(Slic_SyscallInfo *syscallInfo,
                  Slic_ReturnInfo *returnInfo);
```

Figure A.1: A simplified version of the interface exported by the system call dispatcher to extensions.

initialize global state and to transparently provide the same interface at the user level and in the kernel; the string on line 6 is simply a short, arbitrary description of the extension

Lines 9–25 define a system call handler, `Print_CatchSyscall`, which will be invoked when certain system calls are issues (line 34 tells SLIC which system calls should be associated with this handler). The `syscallInfo` argument contains a pointer to information on the system call issued. The `returnInfo` argument is used to communicate return value information from the handler to the SLIC dispatcher. (See Figure A.1 for the definitions of the `Slic_SyscallInfo` and `Slic_ReturnInfo` structures.) Line 14 prints the name of the system call intercepted, reading the system call number from the `syscallInfo` structure. Line 15 tests whether this system call is an `open()` system call. If so, then a `getpid()` system call will be issued by lines 16–22. The existing `syscallInfo` structure passed into this handler will also be passed to other handlers and finally used to issue the system call to the underlying kernel; consequently, a new `Slic_SyscallInfo` structure must be allocated (located on the stack, in this example) to hold the new system call information. Line 19 initializes the new system call information structure, `newSyscall`, and is critically needed by SLIC to be able to properly send the issued system call down the remainder of the extension stack. Line 20 sets the system call desired; since there are no arguments to the `getpid()` system call, the `args` fields do not need to be filled in. Line 21 issues the system call and line 22 prints the result. Note that in this simplified example, we do not check for failure conditions after issuing the system call at line 21; however, in general, extension do need to check for failure conditions on all system call issues, because some other extension lower in the stack could abort the system call with an arbitrary error code.

Lines 27–39 define the main initialization routine which SLIC will invoke when this extension is loaded into the system. Line 32 is needed to initialize this extension with the system call dispatcher. Extensions which make use of multiple dispatchers will need a line similar to this one for each dispatcher. Lines 33–25 register the system call handlers, in this case the same handler is registered for all system calls.

Line 36 now activates this extension using the `Slic_TraceProc()` call. If no processes in the system are marked for tracing by this extension, then the handlers will never be invoked. In this example, the extension marks all processes in the system for tracing by specifying the `init` process (pid 1) and all of it's children (the second argument specified whether all decendents of a process should be marked as well).

A.2 CERT Extension

This section presents a version of the CERT extension discussed in section 4.4.2. The code for the extension is listed in Figures A.3 and A.4.

The goal of this extension is to prevent a security violation in Solaris by preventing symbolic links from being created in the location `/tmp/.pwd.lock`. To do this, on initialization this extension determines the device and inode of the `/tmp` directory (lines 18–49, called from lines 61–64), storing that information in global variables (lines 13 and 14). When a `symlink()` or `rename()` system call is made, the extension checks the target location to determine if the application is attempting to place a symlink at the protected location. If so, the system call is denied with `EPERM` (lines 81–86), otherwise it is permitted.

One aspect of this extension that is worth of note is the initialization of the `tmp_dev` and `tmp_inode` global variables. This initialization is done by issuing a `stat` system call. According to the SLIC API, extensions can only safely issue new system calls while catching systems calls. Consequently, the `stat` system call cannot be issued from the `Slic_ExtensionMain()` routine; rather, the extension must wait for the first system call to be issued, and then perform initialization at that time (lines 61–64).

```
#include "syscall/syscall.h"
#include "slic/types.h"
#include "slic/extension.h"
#include "slic/proc.h"

Slic_GlobalState("Print system call extension");
Slic_SyscallGlobalState();

static void
Print_CatchSyscall(Slic_SyscallInfo *syscallInfo,
                   Slic_ReturnInfo *returnInfo,
                   void **extensionProcData, void **extensionThreadData)
{
    Slic_Printf("%s ", Slic_GetSyscallStr(syscallInfo->syscall));
    if (syscallInfo->syscall == SYS_OPEN) {
        struct Slic_SyscallInfo newSyscall;
        struct Slic_ReturnInfo  newReturnInfo;

        newSyscall.cookie = syscallInfo->cookie;
        syscallInfo->syscall = SYS_GETPID;
        Slic_SyscallIssue(&newSyscall, &newReturnInfo);
        Slic_Printf("pid is %d\n'', newReturnInfo->returnValue);
   }
   return;
}

int
Slic_ExtensionMain(Slic_ExtensionId id)
{
    int ctr;

    Slic_SyscallRegisterExtension();
    for (ctr = 0; ctr < NUM_SYSCALLS; ctr++) {
        Slic_SyscallRegisterHandler(ctr, Print_CatchSyscall);
    }
    Slic_TraceProc(1, TRUE);

    return ESUCCESS;
}
```

Figure A.2: A sample SLIC extension, “Print”.

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "syscall/syscall.h"
#include "slic/types.h"
#include "slic/extension.h"
#include "slic/copy.h"
#include "slic/util.h"

Slic_GlobalState("CERT system call extension");
Slic_SyscallGlobalState();

dev_t tmp_dev = (dev_t)-1;
ino_t tmp_inode = (ino_t)-1;

#define BAD_FILE    ".pwd.lock"

static void
Cert_Stat(Slic_SyscallInfo *info, char *path, dev_t *devNum, ino_t *inodeNum)
{
    Slic_SyscallInfo newSyscall;
    Slic_ReturnInfo  returnInfo;
    struct stat statBuf;
    caddr_t keyPath, keyBuf;

    keyPath = Slic_PermitCopyRange(info->cookie.thread, (caddr_t)path,
                                   strlen(path));
    keyBuf = Slic_PermitCopyRange(info->cookie.thread, (caddr_t)&statBuf,
                                   sizeof(statBuf));
    newSyscall.cookie = info->cookie;
    newSyscall.syscall = SYS_stat;
    newSyscall.args[0] = (int)keyPath;
    newSyscall.args[1] = (int)keyBuf;

    Slic_SyscallIssue(&newSyscall, &returnInfo);
    Slic_UnpermitCopyRange(info->cookie.thread, (caddr_t)path);
    Slic_UnpermitCopyRange(info->cookie.thread, (caddr_t)&statBuf);

    if (returnInfo.errno != ESUCCESS) {
        Slic_Printf("Cert_Stat: stat failed with errno %d\n",
                    returnInfo.errno);
        return;
    }
    *devNum = statBuf.st_dev;
    *inodeNum = statBuf.st_ino;
    Slic_Printf("Cert_Stat: path \"%s\" got device %lu, inode %lu\n",
                path, statBuf.st_dev, statBuf.st_ino);
    return;
}
```

Figure A.3: Part 1 of a sample SLIC extension, "CERT", used to fix a security hole identified by CERT advisory CA-96.16.

```
/* Symlink and Rename will both have the new filename as the second arg */
static void
Cert_Check(Slic_SyscallInfo *syscallInfo, Slic_ReturnInfo *returnInfo,
           void **extensionProcData, void **extensionThreadData)
{
    static  int initialized = FALSE;
    char    path[1024];
    int     len;
    dev_t   devNum = 0;
    ino_t   inodeNum = 0;

    if (!initialized) {
        Cert_Stat(syscallInfo, "/tmp", &tmp_dev, &tmp_inode);
        initialized = TRUE;
    }
    if (Slic_CopyInStr((caddr_t)(syscallInfo->args[1]),
                       (caddr_t)path, sizeof(path)-1, &len)) {
        returnInfo->forceReturn = TRUE;
        returnInfo->errno = EFAULT;
        return;
    }
    for ( ; len >= 0 && path[len] != '/'; len--)
        ;
    if (!strncmp(&(path[len+1]), BAD_FILE, strlen(BAD_FILE))) {
        if (len <= 0) {
            path[0] = '.';
            path[1] = 0;
        } else {
            path[len] = 0;
        }
        Cert_Stat(syscallInfo, path, &devNum, &inodeNum);
        if (devNum == tmp_dev && inodeNum == tmp_inode) {
            /* This is an invalid access; deny it */
            returnInfo->errno = EPERM;
            returnInfo->forceReturn = TRUE;
            return;
        }
    }
    return;
}

int
Slic_ExtensionMain(Slic_ExtensionId id)
{
    Slic_SyscallRegisterExtension();
    Slic_SyscallRegisterHandler(SYS_symlink, Cert_Check);
    Slic_SyscallRegisterHandler(SYS_rename, Cert_Check);
    return ESUCCESS;
}
```

Figure A.4: Part 2 of a sample Slic extension, "CERT", used to fix a security hole identified by CERT advisory CA-96.16.