Direct User Calls from the Kernel: Design and Implementation

by

Weihan Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

©Weihan Wang, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Traditional, general-purpose operating systems strictly separate user processes from the kernel. Processes can only communicate with the kernel through system calls. As a means to ensure system security, system calls inevitably involve performance overhead. *Direct User Callback from the Kernel*, or DUCK, is a framework that improves the performance of network-centric applications by executing a part of application code directly from inside the kernel. Because the code runs in kernel mode and can access kernel memory directly, DUCK is able to eliminate two important sources of system call overhead, namely mode switches and data copying.

One issue with DUCK is how to design an application programming interface (API) that is general, efficient, and easy to use. In this thesis, we present the design of the DUCK API, which includes functions for both direct user code execution and zero-copy buffer management. We have implemented DUCK prototypes on the Solaris/SPARC platform. An efficient way to implement direct user code invocation is through memory sharing between the kernel and user processes. However, because Solaris/SPARC separates the user and kernel address spaces, achieving memory sharing is difficult. In the thesis, we study the SPARC architecture and the Solaris virtual memory subsystem, and discuss three potential approaches to support memory sharing required by DUCK. We proceed to present micro-benchmark experiments demonstrating that our DUCK prototype implementation is capable of improving the peak throughput of a simple UDP forwarder by 28% to 44%.

Acknowledgements

I would like to thank the many people who directly and indirectly made this thesis what it is.

I would like to thank my supervisor Martin Karsten for providing invaluable focus to this research project. I could not have completed the work without his insightful advice and guidance. I am also very grateful to have worked under the guidance of my co-supervisor Tim Brecht, and thank him for his generous support to my research work, for his patience in listening to my ideas, and for his continuous encouragement.

My fellow colleague, Elad Lahav, also deserves my appreciation. His deep understanding of operating systems and valuable suggestions have been important to my work.

Finally, I would like to thank my friends and family for their patience, support, and love. They provide me with a window outside of work, making my life here at Waterloo especially pleasant and enjoyable.

Dedication

Dedicated to my parents and my wife.

For their love and for always encouraging me to follow my goals and ambitions and being there for me when I needed them the most.

Contents

1 Introduction **1**

2 Background and Related Work **4**

2.1 Motivation: Dedicated Servers 6

2.2 Related Work 9

2.2.1 Mode Switch Avoidance 9

2.2.2 Data Copying Avoidance 11

2.3 An Introduction to DUCK 13

2.4 Summary 14

3 DUCK API Design **16**

3.1 Direct User Code Invocation 16

3.2 Zero-copy Buffer Management 17

3.3 Examples Using the API 21

3.4 Emulation 23

4 DUCK Implementation on Solaris/SPARC **24**

4.1 Implementing Direct User Code Invocation 24

4.1.1 Address Space Switching 26

4.1.2 Combining Address Spaces 28

4.1.3 Address Space Duplication 32

4.1.4 Summary 33

4.2 Implementing Zero-copy Buffer Management 34

5 Performance Evaluation **37**
5.1 Evaluation Results 39

6 Conclusion and Future Work **44**

List of Tables

5.1 Peak throughput improvement under different packet sizes 43

List of Figures

3.1 Network protocol stack and buffer management 19

4.1 Address Space: Intel (left) vs. SPARC (right) 25
4.2 Solaris data structures for address space management 30
4.3 Solaris network protocol stack and buffer management 35

5.1 Evaluation Setup 39
5.2 Throughput for 10 bytes per packet 41
5.3 Throughput for 100 bytes per packet 41
5.4 Throughput for 500 bytes per packet 42
5.5 Throughput for 1000 bytes per packet 42

Chapter 1

Introduction

General-purpose operating systems perform tasks such as concurrent process execution, resource control, and user management through a common kernel. User processes are strictly separated from the kernel and can only communicate with the kernel using well-defined system call interfaces[1]. A well-known fact is that system calls inevitably trade off performance for system safety. In this thesis, we examine two sources of system call overhead, namely *mode switching overhead* and *data copying overhead*, and argue that for a class of applications where users only run a single dominant service on one computer system, safety can be traded for better performance. In the framework that we propose, a portion of the application code resides in user space but is executed directly from inside the kernel. When requesting kernel services, the code running in kernel mode can use mere function invocations instead of expensive system calls, and can therefore eliminate mode switches as well as provide opportunities to bypass data copying. We call this framework Direct User Callback from the Kernel, or DUCK.

An important aspect of DUCK is the way in which it interacts with the application, specifically, the design of DUCK's application programming interface (API). Generally speaking, a good API design features generality, imposes as few constraints to the applications as possible, and supports as many platforms as pos-

[1] While some operating systems such as Linux provide other interfaces such as the `/proc` file system, only system calls are common to all operating systems.

sible. It is efficient, allowing implementations with small performance overhead. Moreover, a good API design is easy to comprehend even without the manual. In our work, we strive to design an API according to these criteria. The resulting interface consists of two parts: one to avoid mode switches by executing user code from inside the kernel, and the other to avoid data copying by enabling zero-copy buffer management. The API design is a key component of this thesis.

A second challenge is how to implement DUCK with minimum alteration to existing operating systems. As will be discussed in detail in Chapter 2, although several schemes have been proposed with goals similar to our work, many of them require massive modifications or even reconstruction of the operating system. Because commercial, off-the-shelf operating systems are widely adopted today, a more attractive solution might be a less intrusive one that requires minimal modifications to both the kernel and the application, and that can be easily added to a stock operating system. This is an important goal as well as a contribution of our work.

We have implemented DUCK prototypes on two mainstream operating system and processor combinations, namely Linux over a 32-bit Intel Architecture (i.e. IA-32) and Solaris over a 64-bit SPARC. (Throughout this thesis, SPARC refers to the SPARC specification version 9 [6].) The IA-32 implementation is straightforward. Because operating systems on IA-32 including Linux usually have address spaces shared by both the kernel and the user process, the kernel can access and execute user memory in the same way as it accesses and executes kernel memory. On Solaris/SPARC however, the kernel and process address spaces are separate, making DUCK non-trivial to implement. The focus of this thesis is on the Solaris virtual memory subsystem and how DUCK is implemented on Solaris/SPARC.

This thesis makes the following contributions:

1. The design of the DUCK API, which is general, efficient, and easy to use.

2. A study of the SPARC architecture and Solaris virtual memory subsystem, and an analysis of candidate approaches to implementing DUCK on Solaris/SPARC.

3. A prototype implementation of DUCK on Solaris/SPARC which only requires several lines of changes to the existing Solaris kernel code.

4. Micro-benchmark results to validate the design and performance of the Solaris/SPARC implementation.

The outline of the remainder of the thesis is as follows. We present background information and related work in Chapter 2. In Chapter 3, we discuss the design of the DUCK API, followed by some examples using the API and a discussion of emulating DUCK in user space. DUCK's implementation on Solaris/SPARC is presented in Chapter 4, where we discuss three different techniques for direct user code invocation, and discuss the zero-copy implementation on Solaris. We present micro-benchmark experiments and results in Chapter 5. The thesis is concluded and future work is described in Chapter 6.

Chapter 2

Background and Related Work

Contemporary operating systems strictly separate the operating system kernel and user processes running on it. Because the kernel contains critical procedures and data structures for process and resource management, it has to be protected against arbitrary access by unprivileged users. In addition, multiprogramming operating systems prevent one process from accessing memory belonging to others in order to enforce program isolation [24].

A *virtual address space*, or *address space*, is a range of virtual addresses associated with each process. All the virtual memory of a process is contained in its address space. A special address space called the *kernel address space* is associated with the operating system kernel, containing all the virtual memory belonging to the kernel. The *memory management unit* (MMU) is a hardware module that can be found in many modern processors. It provides mappings from virtual to physical addresses as well as sophisticated mechanisms for address space management. The data structure of the mappings is defined by the MMU, but the mappings themselves are usually manipulated by the operating system and stored in main memory. Complex data structures are used for compact and efficient representations of the mappings. For example, the Intel IA-32 adopts a multi-level hierarchical data structure called a *page table* [9]. After the operating system sets up these data structures properly and tells the MMU where they are located in main memory, the MMU then automatically performs virtual-to-physical address

translations using the contents of these data structures. The operating system may modify the data structures dynamically to reflect changes in the system state. The operating system can also enforce access control policies with the MMU's help. For example, the MMU translates all memory references using the context of the current address space. The operating system may limit a process's memory accesses to its own address space by setting this address space as the MMU's current one. In this way memory in other address spaces cannot be accessed by the current process.

Virtual-to-physical address translations can be done using different techniques. In the commonly adopted *paging* system, a virtual address space is logically divided into fixed-length ranges called *virtual pages* or simply *pages*, which is the minimum unit by which address mappings can be defined. A virtual page is mapped to a *physical frame* (or simply *frame*), or to nothing, to indicate the page is inaccessible. These mappings are maintained by the operating system kernel and referenced by the MMU when performing address translations. Certain protection policies can be specified when defining each mapping. For instance, the operating system may mark a page as accessible only when the processor is in kernel mode. Therefore, accessing it from user mode triggers the MMU to generate a page fault, causing the operating system to identify and terminate the faulting process. In this way, the operating system can apply fine-grained access control within an address space.

Because user processes are strictly separated from the kernel, they have to use a set of well-defined *system calls* in order to interact with the kernel and to request operating system services. System calls are designed so that no process can affect the operating system in faulty or malicious ways. All requests and information passed through system calls are scrutinized before further processing by the kernel, and any illegal access will be rejected. While system calls guarantee system integrity, doing so inevitably introduces performance overhead, including:

- **Mode switching overhead**. One system call involves two mode switches, one from user mode to kernel mode when entering the kernel at the start of the system call, and the other when the system call returns from kernel to user mode. This incurs non-negligible hardware overhead. For example,

Intel processors provide the `sysenter` instruction to trap into kernel mode. It contains 28 micro-operations such as loading segment registers, updating state registers, changing the privilege level of the processor, among others. In addition, the operating system needs to maintain software state as well: general registers and kernel data structures reflecting the current context need to be saved when entering the kernel, and restored when returning to user mode. This is done by the processor copying them into and out of main memory.

- **Data copying overhead**. Many system calls trigger the asynchronous execution of tasks. For example, to transmit data through a network interface, a process calls the `send` system call along with a pointer to a data buffer prepared in the application's address space. When the system call returns, the data might be queued in the system but not actually transmitted. The kernel continues working after the system call to drain all the data to be sent. Because the process might alter its user-space buffer after the system call, the kernel has to make its own copy of the data before returning to user mode, and must use this in-kernel copy afterwards. Memory copying consumes a significant number of CPU cycles, and performance overhead is roughly proportional to the size of data being copied.

Our goal is to reduce both mode switching and data copying overhead for one class of applications which we call *dedicated servers*. We discuss these applications in Section 2.1, followed by a summary of related work in Section 2.2 and an introduction of our proposed framework in Section 2.3.

2.1 Motivation: Dedicated Servers

Because of a steady decrease in hardware prices and the rapid development of virtualization technologies, more organizations choose to use a single computer system, or a single virtual machine, to run one primary network service in user space, such as a Web server, an application server, a streaming server, a proxy

server, or an accounting and monitoring server. In such a configuration, which we call *dedicated servers*, the fate of the overall system is tightly coupled with a few user-level processes, where the failure or malicious behaviour of the primary service is functionally equivalent to a failure of the overall system. If the primary service becomes unavailable, the whole system loses its value even though the underlying operating system may still be intact. Although off-the-shelf, kernel-based operating systems are often used, the strict separation of processes and the kernel in these operating systems is less relevant, and we argue that safety and protection may be reduced to provide better performance. In particular, user processes directly exchanging information with the kernel without system calls may help reduce or eliminate mode switching and data copying overhead, especially for I/O centric services which intensively involve system calls.

One potential approach to eliminate system calls is to move the entire application into the kernel. However, there are several issues preventing this approach from being widely adopted:

- **Safety and error recovery**. Although safety is no longer our primary concern, we still want to isolate the application and the kernel as much as possible for a minimum compromise on safety and resilience to faults. As the 80-20 rule states, most performance overhead is caused by a small fraction of application code [20]. Only this fraction but not the entire application needs to be moved into the kernel; errors by the code remaining in user space can be identified and recovered by the operating system, instead of potentially crashing the kernel.

- **Programming limitations**. There are constraints for programs running in kernel mode, such as limited stack and heap sizes, disabled floating point arithmetic [16], and reduced capability of dynamic linking, making entirely in-kernel programs less flexible than user-space applications.

- **Development costs**. Developing kernel programs is notoriously difficult. Limited features and programming support make developing and debugging

in-kernel applications nontrivial even for experienced developers. Also, because the full set of user-level libraries are not available for the kernel, when writing the entire application in the kernel, developers have to build most required libraries from scratch, which is usually not practical.

- **Maintenance costs**. System calls are not available for kernel programs. In-kernel applications have to use private kernel functions to request operating system services. Unlike system call and library interfaces, kernel functions are not standardized, and they change rapidly as the operating system evolves. This introduces significant overhead for maintaining in-kernel applications across different operating systems or for maintaining different versions of one operating system.

Another approach to reduce system calls is to write most application code as usual but move critical parts of the code into a kernel module. Although this limits the amount of the code running in the kernel when compared with the previous approach, this approach splits an application into physically segregated parts. It is also difficult to implement arbitrary communications between the kernel module and the user-level part as there is no existing way of doing this on most operating systems. Ad hoc methods have to be devised for such communications.

Instead, in the design that we propose, the entire application is developed as usual and executed as a normal user process. At run-time, a portion of the application code is directly executed by a kernel thread from inside the kernel. The code running in the kernel can access operating system services without system calls. This scheme addresses the above issues from two perspectives. First, because developers can freely choose a portion of the application to be executed in the kernel without physically splitting the program, this approach provides much flexibility. Second, because the code is compiled as a part of the user program, developers can use the full range of libraries and development tools available in user space, working naturally in a programming environment that they are familiar with throughout development and maintenance cycles. Besides system call avoidance, our proposed framework also provides zero-copy facilities for the code running in the kernel to

perform network I/O. This is based on the fact that in-kernel code has the privilege to access kernel buffers directly.

2.2 Related Work

We focus on reducing the overhead caused by mode switches and data copying. Although both of these issues have been extensively studied, almost all previous projects only focus on one of them. As a major advantage of our work, DUCK eliminates both mode switches and data copying using a single framework. In the following two subsections, we classify related projects into two categories according to the primary issues they address, and discuss them separately.

2.2.1 Mode Switch Avoidance

A straightforward approach to limiting the number of mode switches is to develop the whole application inside the kernel, as adopted by the TUX web server [3] and the kHTTPd Linux HTTP Accelerator [26]. However, as discussed in Section 2.1, kernel development is difficult and prohibitive for application developers. Several projects have been proposed to allow code to be written as user-level programs but to be executed in the kernel.

Kernel Mode Linux [17] is one such project. It allows the entire user-space application to execute in kernel mode while the kernel is protected using type theory. Applications are written in a typed assembly language (TAL) and the kernel performs type-checking before executing the program. Using a type theory based on the TAL, the kernel can ensure that the typed-checked programs can never perform an illegal access to its memory. When the program executes in the kernel, system calls become plain function calls and can be invoked with little overhead. A drawback of this solution is inflexibility to execute only a part of an application in the kernel. In addition, the entire application must be written in the TAL, and the run-time overhead of such a language might offset the gains of avoiding system calls.

In contrast, Cosy [28] allows applications to execute manually designated portions of user code in the kernel. By using a custom C compiler, Cosy compiles the specified code segments into an intermediate representation called *compounds*. At runtime, compounds are safely executed in the kernel by an interpreter. In addition to performance overhead of the interpreter, Cosy's compounds are written in a highly restricted programming environment with limited features and language support. For example, only C statements and library functions that are supported by Cosy can be used in compounds. Our solution, by relaxing safety assumptions, allows programs to be compiled into native machine code and therefore does not limit the language features available to the applications.

Singularity [11] is a new operating system that uses software-based type checking to enforce process isolation. It significantly reduces performance costs by avoiding hardware protection and putting all processes, including the kernel, into one or only a few address spaces. Unfortunately, while stock operating systems are widely used in dedicated servers, Singularity is a completely new operating system written in a special language, Sing#. This makes it prohibitively difficult to port existing applications to the operating systems. Nevertheless, Singularity's static verification mechanism is of interest to DUCK, especially the use of safe high-level languages and the signing of approved applications. Such a scheme offers better safety than does DUCK, without introducing performance overhead after the application is loaded.

Single address space operating systems [15] are a class of operating systems that have only one global virtual address space; protection is provided through *protection domains* that control which pages a process can reference. While these operating systems are designed mainly to support 64-bit address spaces, their architectures usually provide additional benefits such as reduced mode switch overhead (because of no address space switches) and lightweight data sharing between user processes and the kernel. With the help of memory management units, we should be able to adopt protection domains as well as other techniques in those operating systems to the DUCK framework to enforce access control policies within one address space to the user code running inside the kernel.

Several projects have been proposed to reduce the number of kernel/user crossings without major changes to the operating system architecture. For example, as an improvement to the traditional UNIX event mechanisms, Scalable Event Notification Delivery [18] stores recent events into a user-space ring buffer providing users with events without the cost of system calls. Our approach is a step towards generalizing such mechanisms, with the hope that users can devise similar, efficient techniques with the DUCK framework.

2.2.2 Data Copying Avoidance

Several projects have been proposed to reduce data copying between user and kernel space. IO-Lite [19] is a unified zero-copy buffering and caching framework. It encapsulates non-contiguous data buffers in *buffer aggregates* and passes their pointers among different operating system components. All buffer operations are performed using these aggregates. Protection and security are maintained through access control and read-only sharing. As complete and ingenious a solution as it is, IO-Lite requires rewriting a substantial part of the kernel and the device drivers, as well as the applications, making porting to existing operating systems prohibitively difficult. The framework proposed [25] is also based on buffer exchange, which shares similar ideas and suffers from the same problem as IO-Lite.

Zero-copy TCP in Solaris [14] maps kernel buffers to user space allowing user processes to access them directly. Copy-on-write is used at the transmit side so that the copy semantics of the traditional socket interface are preserved. One major shortcoming of this approach is the difficulty of dealing with unused areas within a page, making the implementation of this approach nontrivial, and rendering it dependent on particular hardware platforms. Also, frequent virtual memory operations can be expensive when compared to conventional data copying.

Data-stream splicing [21] is a mechanism to reduce data copying for a class of servers such as Web proxies that forward data. Applications invoke the `ioctl` system call to instruct the system to "splice" two network sockets. After splicing, data received from one socket is automatically sent to the other and no data copying is involved during the process. However, because no application logic can

be inserted between the two spliced sockets, the application still needs to copy the data to the user address space to read or to modify the data. In contrast, our approach allows in-place data access from within the kernel and thus supports a much larger spectrum of applications. For example, packet monitoring services such as intrusion detection systems may examine certain fields of each packet before forwarding it to the next host and packet transcoding services such as network address translation (NAT) need to modify the content of forwarded packets. The ability to support in-place access to transmitted data is crucial for these types of applications.

The U-Net communication architecture [27] allows user processes to communicate with network interfaces directly without involving the kernel in the communication path. The architecture virtualizes the network interface in such a way that every process has the illusion of owning the physical interface. The role of U-Net includes multiplexing the actual interface and enforcing protection as well as resource allocation policies. A process has control over message buffers and over sending and receiving queues corresponding to the virtual interface it owns. Depending on the operating system structure and hardware mechanisms, the U-Net implementation may differ drastically. Among three operation modes, *direct-access U-Net* implements zero-copy data transfer. Application data structures are transmitted to or from the network interface without any data copying. By redesigning the network processing architecture from scratch, U-Net takes a different, bottom-up approach from our work to achieve zero-copy.

In practise, a zero-copy facility for file-to-socket data transfer has long existed in UNIX systems. The `sendfile` system call is used to copy data from one file to a socket. This copying is done within the kernel and no user-space buffers are required. Data is fed directly from file caches to the network interface. Because the applications have no method for accessing kernel data buffers, there is no opportunity for them to examine or modify the data being sent. The Linux kernel recently added a `splice` system call (in version 2.6.17) with the hope of offering a more general zero-copy framework for data transfer between any types of files [1]. However, the current implementation in version 2.6.17 only supports transferring from

a regular file to a socket. This system call accepts two file descriptors and a length as parameters. Once invoked, it reads *length* bytes from the first descriptor and writes them to the other without copying the data to user space. Like `sendfile`, `splice` does not allow the application to access the data being transmitted.

Finally, we note that the exokernel [13] and kernel extensions like VINO [22] and SPIN [12] are related to our work. These projects are mainly focused on augmenting the kernel and exposing lower-level sub-systems to interested applications. Our work, however, concentrates on improving application performance, although it also "augments" the kernel by "sinking" application logic into it.

2.3 An Introduction to DUCK

In this section, we introduce Direct User Callback from the Kernek (DUCK) to provide more background for this thesis. As described in Chapter 1, the main contributions of the thesis are the DUCK API design and the DUCK implementation on Solaris/SPARC.

The DUCK framework consists of three parts: 1) a kernel module that installs a new system call into the operating system that permits user code to execute from inside the kernel, 2) a user-space library to help invoke the system call and to perform housekeeping tasks, and 3) the application itself. To use DUCK, the developer first identifies a critical execution path, or a function, in the application code which frequently invokes system calls and can benefit most from reduced mode switching and data copying overhead. Next, instead of calling this function directly the application invokes the system call provided by the kernel module with the pointer to the function as a parameter. After entering the kernel the system call invokes the function even though the function body resides in user space. This requires memory sharing between the kernel and the user process, which may be implemented differently on different platforms. A drawback of this approach is that code running in the kernel may be subject to platform-specific restrictions, such as limited stack sizes and disabled floating point arithmetic. However, because the execution path critical to application performance is expected to be short, these

restrictions should not create significant problems for most server applications.

When the specified user function is executed in the kernel, all system calls it invokes are automatically replaced with direct calls to appropriate kernel functions. Because system call entries in user space are functions provided by a shared library, we can achieve system call replacement through dynamic linking. For example, when a normal user program written in C calls `gettimeofday`, a function named `gettimeofday` in the C standard library will be invoked, which is responsible for performing the actual system call. To replace `gettimeofday`, we define a function in the DUCK library that invokes the kernel function corresponding to the `gettimeofday` system call if the processor is in kernel mode, otherwise the function behaves the same as the user-space `gettimeofday` library function. The DUCK library function is also named `gettimeofday`, so that the dynamic linker can automatically "override" the original library function with the new function that we provide.

In addition to the new system call the DUCK kernel module also provides in-kernel services for zero-copy buffer management that can be used by user code when executing in kernel mode. Because the user code has kernel privileges, it can access kernel services and memory. Data copies are avoided in DUCK by directly accessing kernel memory. For example, when the application performs network I/O the user code running in the kernel can operate on buffers allocated from the kernel instead of requiring its own buffers in the user address space, thereby eliminating the data copying incurred by system calls. Because this requires different interfaces from traditional system calls, the user code may need modifications to avoid data copying. In the next chapter, we introduce the design of the DUCK application programming interfaces, including the interface for direct user code invocation and zero-copy buffer management.

2.4 Summary

In this chapter, we first identify two types of overhead involved in system call invocation, namely mode switching and data copying overhead. DUCK aims to reduce

both overheads for a class of applications we call dedicated servers. Because the fate of a dedicated server is tightly coupled with the primary application running on it, the security concerns of traditional operating systems can be traded for better performance of the primary application. To achieve this, we execute a part of the application code from within the kernel, so that system calls become function invocations and the application can access in-kernel data directly without copying the data to or from the user address space. Previous and related projects have been presented. Although other work has similar goals to DUCK, DUCK reduces both overheads within a single flexible framework. More importantly, DUCK is the only project that can execute arbitrary parts of application code in the kernel without altering the overall operating system architecture. This is important for an increasing number of organizations that deploy off-the-shelf operating systems as their strategy to reduce operational costs.

Chapter 3

DUCK API Design

In this chapter we present our design of the DUCK API through which applications interact with the underlying DUCK system. As discussed in Section 2.3, DUCK has two closely related goals: to reduce mode switches by executing user code from inside the kernel, and to avoid copying by allowing the user code to access and manipulate kernel buffers directly. Therefore, we divide the API into two logical parts to reflect these goals, and discuss them in the first two sections of this chapter. In Section 3.3 we provide a few examples showing how applications can use these two parts together; we discuss DUCK emulation in Section 3.4.

3.1 Direct User Code Invocation

When the DUCK system is initialized it dynamically installs a new system call. The application can use this system call to interact with the kernel. We define the system call as follows:

```
typedef int (* user_callback)(void * opaque);
int  duck_syscall(user_callback cb, void * opaque);
```

The application invokes the system call with two parameters: the pointer to a function and an opaque value. We call this function a *user callback*. It contains the code to be executed in the kernel and can be compiled and linked in normal

ways with other parts of the application. The in-kernel function that implements the system call invokes the user callback from the kernel, passing the opaque value to the callback as the sole parameter, and returns to user space after the callback returns. This implementation is as simple as:

```
int duck_impl(user_callback cb, void * opaque)
{
    return cb(opaque);
}
```

From the user's perspective `duck_syscall` is a thin wrapper around the callback function doing nothing but switching into the kernel before invoking the callback and switching back afterwards. The application can use the opaque value to exchange information with the callback function.

Finally, the system calls `duck_init` and `duck_fini` are provided to setup and cleanup DUCK. The `duck_init` function may return an error code if initialization fails:

```
int   duck_init();
void  duck_fini();
```

Once `duck_init` successfully returns, the cleanup procedure must be performed even if the program terminates abnormally. This can be done by forcing the kernel's function that cleans up a terminated process to call `duck_fini` for that process. This is not difficult to implement.

3.2 Zero-copy Buffer Management

To perform actual work, the user callbacks need to interact with kernel services. Sending and receiving network packets are two important services on which most network-centric applications depend. The conventional method for using these services is to invoke `recv` or `send` system calls or equivalents with user-provided buffers. These system calls are responsible for copying data from the buffer into

a kernel buffer and vice versa. User callbacks, however, execute in kernel mode and therefore can access kernel buffers directly. To receive packets, a callback can obtain the kernel buffer and directly examine the data in the buffer; to send packets, the callback allocates a new kernel buffer, fills the buffer with data to be sent, and passes it to the kernel for further processing. In this way performance overhead caused by data copying can be completely avoided.

Designing an API for direct kernel buffer access is challenging because different operating systems take different approaches to implementing kernel buffers. However, the API must provide a single interface for portability and ease of use. Next, we describe an abstract model for in-kernel network processing and buffer management which can be found in many mainstream operating systems, and then introduce our API design which is based on this model.

Figure 3.1 (left) illustrates the typical model for in-kernel network protocol processing, where the protocol stack is composed of several logical layers (the three layers in the figure are illustrative only). The user process interacts with the topmost layer through `recv`/`send` system calls, which in turn interacts with lower layers on the process's behalf. Network packets are encapsulated in *message buffers* (or *buffers* for short), and transmitted between layers through *queues* (arrows in the figure). On the receiving side, messages are constructed in the bottom layer, filled with the received data, pushed upward, processed by intermediate layers, and finally destroyed by the top layer after the data is consumed by `recv`. On the sending side, messages traverse in the reverse order: they are constructed at the top by `send` and destroyed at the bottom after the data is transmitted. Figure 3.1 (right) shows the abstract data structure for a message buffer. It has a *buffer head*, which contains nothing but a linked list of one or more *buffer blocks*. A buffer block corresponds to a contiguous *data area* where the actual data resides. In addition to the range of data areas buffer blocks also record the range of the actual data within the area (dashed arrows in the figure) to accommodate unused buffer spaces before and after the actual data. Buffer blocks provide flexibility and efficiency for buffer memory management. For example, protocol layers may dynamically insert new data areas to a buffer as a prefix or suffix of the data without copying existing

areas.

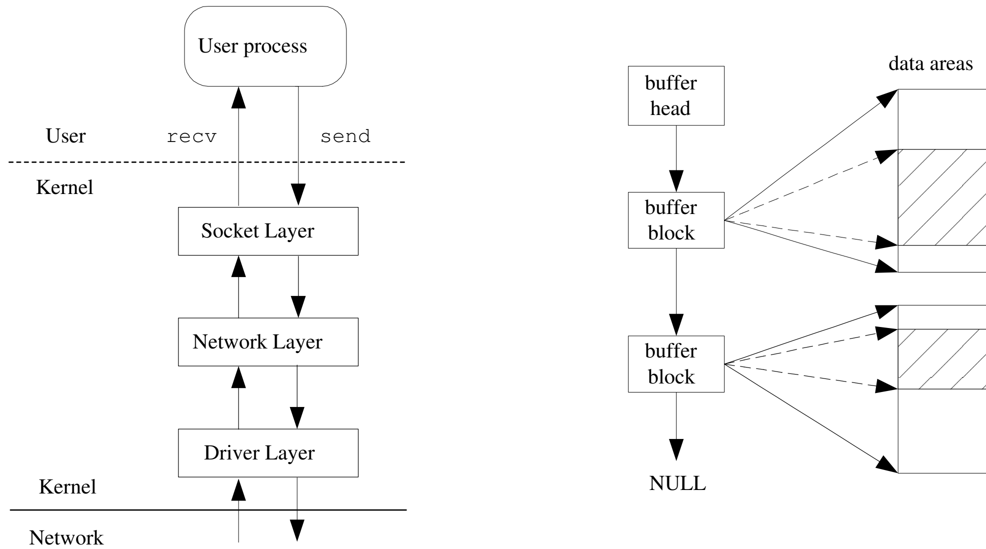

Figure 3.1: Network protocol stack and buffer management

Our API design is based on the above model. We provide five functions to access a buffer once the user callback obtains the pointer to the buffer head. Although the buffer resides in the kernel, and only code running in the kernel shall call these functions, it is not difficult to augment these functions so that when the application invokes them from user mode, a new buffer is constructed and buffer data is copied into the user address space. While this scheme introduces data copying, it provides transparency in using these functions, enabling the application to invoke them regardless of whether it is executing in kernel or user mode.

These five functions include `duck_blk_first`, which returns the pointer to the first buffer block for a given buffer head (buffer heads can be obtained by calling functions that will be introduced later). Given a buffer block, `duck_blk_next` returns the next buffer block. Functions `duck_blk_get_data` and `duck_blk_get_len` retrieve the starting point and the length of the actual data in a block, respectively. In addition, `duck_blk_adjust` allows the application to adjust the range of the actual data in a block. The function prototypes below summarize these five

functions. In the code we define the type of the pointer to a buffer head as `duck_sb` (meaning "DUCK socket buffer") and the pointer to buffer blocks as `duck_blk`. They are defined as `void *` because operating systems may implement buffers in different ways; the user shall treat these pointers as opaque values and not operate on them directly.

```
typedef void * duck_sb;
typedef void * duck_blk;

duck_blk  duck_blk_first(duck_sb sb);
duck_blk  duck_blk_next(duck_blk blk);
void *    duck_blk_get_data(duck_blk blk);
size_t    duck_blk_get_len(duck_blk blk);
void      duck_blk_adjust(duck_blk blk, void * begin, void * end);
```

Next, we need to specify functions for user callbacks to exchange message buffers with the kernel. The traditional `recv` system call implementation involves two steps when receiving network data; it first requests one or more message buffers from the underlying protocol stack, and then copies the data to a user-provided buffer and destroys the message buffers afterward. We provide a function called `duck_recv` in the API, which requests a kernel buffer filled with received data and returns a pointer to the buffer. The function `duck_recv` requires the file descriptor of a socket as a parameter. It requests exactly one message buffer from the protocol stack corresponding to the file descriptor before returning to the caller. Unlike `recv`, `duck_recv` users cannot specify the data length they expect, as the amount of data a buffer holds is predetermined by the kernel. The `duck_sb_get_len` function can be used to retrieve the data length of a given buffer.

The `duck_sb_alloc` function allocates a new buffer from the kernel. Its sole parameter is the preferred maximum data length of the buffer. The actual length might differ because of differences among implementations. For the same reason, the returned buffer may have one or more buffer blocks. There are two functions

that dispose of a buffer: `duck_sb_free` destroys a buffer, and all memory associated with it, and `duck_send` transmits a buffer through a socket. The buffer is freed automatically after the sending operation completes. The function prototypes below summarize the functions described above:

```
int       duck_recv(int fd, duck_sb * sb);
int       duck_send(int fd, duck_sb sb);
duck_sb   duck_sb_alloc(size_t preferred_len);
void      duck_sb_free(duck_sb sb);
size_t    duck_sb_get_len(duck_sb sb);
```

3.3 Examples Using the API

In this section, we give two simplified examples to illustrate how the API can be used in real applications. The first example sends an increasing integer to another host once per second; the second forwards any packets it receives from a sender to a third host.

The pseudo code of the first example is listed below. Details such as error handling are omitted. The function `main` initializes the file descriptor of the destination socket and passes it to the user callback through the opaque argument. The `callback` function will be executed in the kernel as user callback. It allocates a message buffer, initializes the buffer with an integer value, and then sends the buffer out in an infinite loop. The `sleep` call is made to insert one-second pause between iterations.

```
int callback(void * opaque) {
    int fd = *(int *)opaque;
    int cnt = 0;
    while (1) {
        duck_sb sb = duck_sb_alloc(sizeof(cnt));
        void * data = duck_blk_get_data(duck_blk_first(sb));
        *(int *)data = cnt++;
        duck_send(fd, sb);
```

```
        sleep(1);
    }
    return 0;
}

main() {
    int fd;
    // initialize fd as the file descriptor of a socket.
    // fd = ...
    duck_syscall(callback, &fd);
}
```

The second example is also quite simple. Instead of one file descriptor, however, `main` passes two file descriptors to the callback, one for receiving and one for sending data, this is done using a structure called `arg_t`.

```
struct arg_t {
    int fdin;
    int fdout;
};

int callback(void * opaque) {
    arg_t * args = (arg_t *)opaque;
    while (1) {
        duck_sb sb;
        duck_recv(args->fdin, &sb);
        duck_send(args->fdout, sb);
    }
    return 0;
}

main() {
    struct arg_t args;
    // initialize input and output file descriptors
```

```
    // args.fdin  = ...
    // args.fdout = ...
    duck_syscall(callback, &args);
}
```

3.4 Emulation

The platform-independent property of the API enables the implementation of a DUCK emulator, although an actual implementation is left for future work. We briefly describe how it would operate. Users may use the emulator to test their applications in user space without endangering the kernel because of faulty or malicious behaviour. The emulator can be replaced with a real implementation after the application is fully tested or verified. With proper design and implementation of the emulator, the replacement can be done dynamically, requiring no modification or recompilation of the application code.

In the emulator, the call to `duck_syscall` is reduced to a plain function call to the callback instead of trapping into the kernel and executing the callback from there. To emulate buffer management, buffers are constructed in the user address space and traditional system calls are used to copy data between the user and kernel address spaces. From the user's perspective callbacks still have access to DUCK "kernel" functions that are explicitly specified by the API, and despite some performance degradation because of copying, these functions behave exactly the same as a real implementation. Another minor drawback of the emulator is that there is no access to other kernel functions that could be used to further improve the application's performance.

Furthermore, to verify that the application is able to handle various message buffers, the emulator's `duc_recv` and `duck_alloc` may randomize the number of buffer blocks as well as their data lengths contained by one message buffer. A correctly implemented application should ensure that it can handle these variations.

Chapter 4

DUCK Implementation on Solaris/SPARC

In this chapter, we discuss issues related to DUCK's implementation in Solaris on the SPARC (Solaris/SPARC). We first describe our approach for invoking user callbacks from the kernel in Section 4.1, and then discuss the implementation of zero-copy buffer management in Section 4.2.

4.1 Implementing Direct User Code Invocation

An efficient way to invoke user code from the kernel is through *memory sharing* between the kernel and user address space. The kernel may execute a user callback using conventional function calls if user memory can be accessed and executed in kernel mode. Similarly, the callback may reference global variables in the user address space if it has access to user memory while it runs in the kernel.

For a number of operating systems running on processors like the IA-32 family, memory sharing is inherent. As illustrated in Figure 4.1 (left), each address space in these operating systems typically includes both virtual memory belonging to the process and belonging to the kernel. While user threads running in this address space can only access the portion of the address space designated as user space, kernel threads have the full privilege to read, write, and execute memory in either

user or kernel address space. As a result, direct user code invocation can be implemented easily on these platforms.

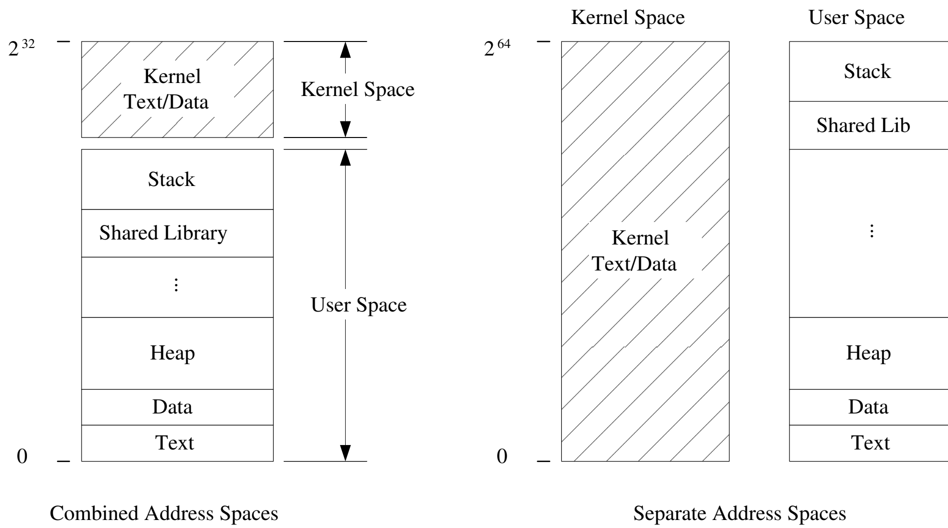

Figure 4.1: Address Space: Intel (left) vs. SPARC (right)

However, Solaris on SPARC separates user and kernel address spaces to allow both of them to span the whole address range. The operating system switches from the user to the kernel address space every time a user process traps into the kernel and switches back on returning. A thread only sees one address space at any time. This makes implementing memory sharing non-trivial. In the following subsections we discuss the Solaris virtual memory system in greater detail and present several approaches to implementing memory sharing. These include:

- Address space switching. Switch the current address space to the user address space before invoking the callback from inside the kernel, and switch back afterwards.

- Combining address spaces. Combine user and kernel address spaces into one as many operating systems do on IA-32.

- Address space duplication. Copy address mapping information from the user address space into the kernel.

Although not all these approaches work, they are helpful in understanding how the Solaris kernel works and why we chose the approach used in our prototype.

4.1.1 Address Space Switching

For instructions to reference address spaces other than the current one, the SPARC processor associates an *address space identifier* (ASI), a numerical value, with every data load/store instruction [6]. There are two important ASIs: the primary ASI (0x80) and the secondary ASI (0x81). When a load/store instruction does not specify an ASI explicitly, the primary ASI is used by default. For example, the instruction

```
LOAD 1234, reg
```

loads the data from virtual address 1234 to register *reg* using the primary ASI, whereas

```
LOAD 0x81:1234, reg
```

does the same thing but uses the secondary ASI as specified in the instruction. Note that control flow instructions such as CALL and JUMP do *not* support explicit ASIs. They are hard-coded to use the primary ASI.

ASIs point to address spaces, and each load/store instruction operates in the address space identified by the associated ASI. It is the kernel's responsibility to maintain the mapping from ASIs to address spaces, and the mechanism to manage this mapping depends on the specific processor. For example, Sun's UltraSPARC III [8] processor has two special registers named *context registers*, used to store the identifiers of address spaces mapped to the primary and secondary ASIs, respectively. Threads running in user mode do not have permission to modify these

registers.

On Solaris, when the processor executes in user mode, both primary and secondary ASIs point to the current process's user address space. When the processor traps into the kernel, the kernel changes the primary ASI to point to the kernel address space, so that subsequent load/store instructions refer to kernel memory by default. These instructions can refer to user memory at any time by using the secondary ASI explicitly. This is necessary because the kernel needs to copy data between user and kernel address spaces as a part of many system calls.

In short, primary and secondary ASIs enable access to both address spaces at the same time while in the kernel. DUCK could simply use this technique to access user memory while executing a callback from within the kernel. However, because control flow instructions that are required by function invocations do not support explicit ASIs, this technique cannot be used to execute callback functions from within the kernel. As a result, we next explore alternative solutions to *execute* the callback.

One idea is to set the primary ASI to point to the user address space before invoking the callback and switch it back afterward. This is difficult to achieve because, first of all, other methods are still required to implement memory sharing between the two address spaces. Second, this technique involves three steps: 1) switch to the user address space, 2) invoke the callback, and 3) switch to the original address space. However, as soon as the first step is executed, the processor flushes its instruction prefetch queue and immediately fetches the next instruction from the new address space. As a result, the second step can never take place and the processor fetches incorrect instructions. There are two possible workarounds. First, one could duplicate all the instructions in step two and three into the user address space and place them into the same virtual address where they are located in the kernel. Thus, the processor will be able to fetch proper instructions from the new address space. However, this may require dynamic code relocation if the user-space address has already been occupied.

The second workaround is to mimic the behaviour of system calls as they return to user mode. System calls on the SPARC use the DONE instruction to restore the

primary ASI and to continue execution from a specified user-space address in one atomic operation. Similarly, DUCK could use DONE to switch the address space and jump to the user callback in one instruction. The problem is that DONE also brings the processor into user mode, which is contrary to our goal of direct user code execution from inside the kernel to avoid mode switches.

4.1.2 Combining Address Spaces

Since address space switching is not feasible, we have explored other approaches. One idea is to mimic combined address spaces as in many operating systems on Intel platforms, so that a kernel thread can "see" user-space memory of the current process. To achieve this, we can combine the address space of the DUCK process ("DUCK process" refers to the process using DUCK) into the kernel, and use this combined address space as the new kernel address space. When the DUCK process is running, either in user mode or kernel mode, we let the primary and secondary ASIs point to this combined address space. It is notable that because page mapping entries are associated with protection bits, and kernel pages are marked using these bits as unaccessible from user mode, the combination will not expose kernel memory to unprivileged user threads.

How to merge the two address spaces? Fortunately, with the 64-bit address space on the SPARC, the kernel and most applications will leave a significant proportion of the address space unused. This offers ample space to import one address space into the other. The layout of one address space may need adjustments to avoid overlapping with the other. This is not a problem, because on most operating systems including Solaris, the layout of user address spaces may be adjusted at compile-time or at run-time if the program is compiled using position-independent code. As mainstream compile-time and run-time linkers support layout customization by using command line arguments or configuration files, the adjustment can be done easily. The next step after layout adjustment is to combine the user address space into the kernel. Before we explain how this is accomplished, we provide a brief overview of the Solaris virtual memory subsystem, with many technical details omitted. Please refer to [6] and [10] for detailed lower-level descriptions.

Solaris Virtual Memory Management

Figure 4.2 illustrates the kernel data structures used by Solaris for address space management. The right half of the figure shows the kernel data structures used for managing a user address space. The variable `curproc` is a per-processor pointer in the kernel, pointing to the process control block (`struct proc_t`) of the current process. Each process control block points to an object of type `struct as` a data structure containing all the information about an address space. Address spaces are logically divided into memory regions, such as heaps, stacks, and data areas. They are called *segment* in Solaris and each of them is managed by a *segment object* in the kernel (`struct seg`). The `as` object maintains an AVL tree containing all segment objects belonging to it. Each segment object also contains a list of all virtual pages in the corresponding region. The left side of the figure shows the data structures for kernel address space management which is almost identical to the right side. The only difference is that the `as` object is referenced by a global pointer called `kas`, instead of a process control block.

Each `as` object is also associated with an object of type `struct hat` (Hardware Address Translation). It stores virtual-to-physical address translation information for the address space. On Intel platforms, the `hat` includes a page table. On Solaris/SPARC systems a data structure similar to page tables is used called HPT, or *hashed page table*, which as the name implies uses a hash table to store mapping entries. Both the page table and HPT are used by the MMU to translate virtual into physical addresses. Solaris provides a set of kernel functions to allow other parts of the kernel to manipulate translation information in the same way across different hardware platforms. For example, the function `hat_memload` maps a virtual page to a frame by adding a new mapping entry to a given `hat` object. Similarly, `hat_unload` removes the mapping entry for a given page. These functions have the same signature but have different implementations across different platforms.

Most processors including Intel and SPARC use a hardware translation lookaside buffer (TLB) to cache mapping entries. The SPARC uses an additional per address space software cache called a translation storage buffer (TSB). While the

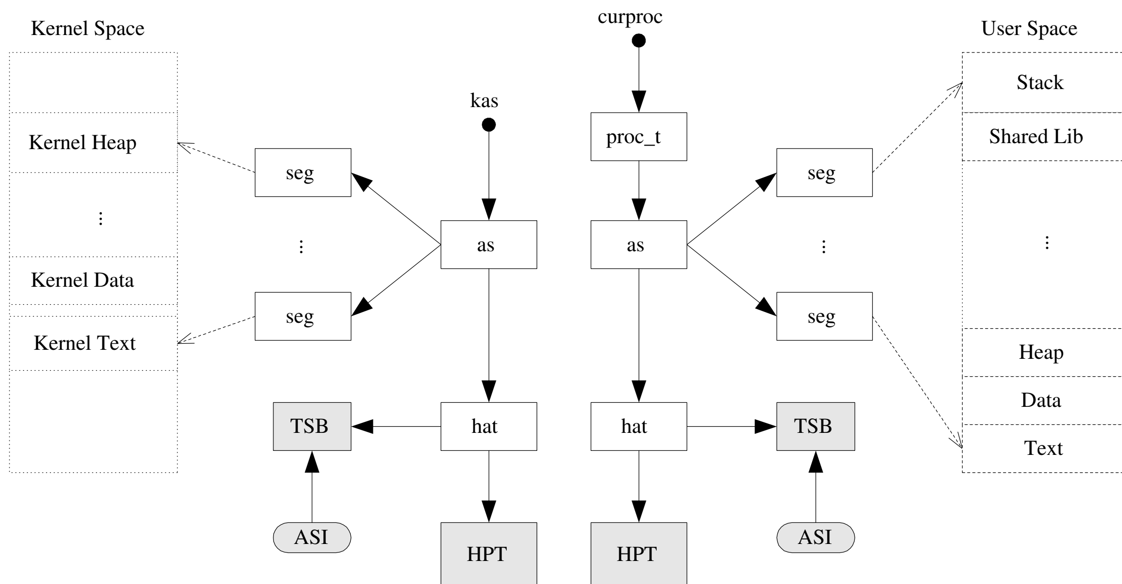

Figure 4.2: Solaris data structures for address space management
(gray boxes are SPARC specific data structures)

TLB is specialized hardware in the MMU, the TSB is an in-memory table maintained by the operating system. As mentioned earlier, the primary and secondary ASIs on the SPARC point to address spaces. In the implementation, they actually point to the identifiers of the TSBs corresponding to these address spaces (see Figure 4.2). To perform address translation the MMU searches for the mapping entry in the TLB and invokes the page fault handler if the entry is not found. Because the structure of the TSB is dictated by the hardware, when a page fault happens the hardware can compute the in-memory address of the TSB entry based on the current ASI value and the faulting address and provide this address as a hint to the page fault handler. The handler can then quickly determine whether the desired entry exists at the given location by examining entry flags at that location. If it does exist the handler manually injects the entry into the TLB (called *software TLB replacement*) and instructs the processor to retry the faulting instruction. Otherwise, the page fault handler resorts to the address space's HPT for further lookup, which involves more overhead than the TSB lookup.

Solaris Page Fault Handling

Next, we present a typical page fault handling scenario to illustrate how the kernel interacts with these data structures. Suppose the current thread executes in user mode and references a virtual address in the current process's heap and there is no physical memory mapped to that address, which triggers a page fault. This may happen when the virtual page is accessed for the first time or the physical frame has been paged out. Because the entry for the referenced address does not exist in the TLB, the TSB, or the HPT, the page fault handler needs to build a new entry. It first locates the `as` object of the current process through `curproc` and searches the object for the segment object in charge of the faulting address. Because segment objects have different ways of managing virtual memory in their own ranges, the fault handler passes control to a function registered with the segment object to perform actual the work.

The function registered with segment objects in charge of the user heap resolves page faults in two steps. First, it calls the physical memory management subsystem to allocate a new frame or reloads the frame from the swap space, depending on the situation. Next, it maps the faulting virtual page to the new frame by calling `hat_memload`. This fills the HPT, as well as the TSB and the TLB, with a new translation entry, so that the processor will find the appropriate mapping when retrying.

Combining Address Spaces

Now we describe how to merge a user address space into the kernel's address space after layout adjustment is done by the linker. Since any program wishing to use DUCK must call `duck_init` (Section 3.1), this system call merges its address space into the kernel's. The `duck_init` system call iterates through all the existing pages of the current process by examining its `as` and segment objects and duplicates the page translation information into the kernel's address space by calling `hat_memload` on the kernel's `hat` object. The `duck_init` system call then modifies the current process's `as` object, forcing it to point to the kernel's `hat` object instead of its own. These steps are expected to alter the operating system's behaviour in the following

ways:

- Because the kernel's HPT has included translation information for the DUCK process, the operating system can now translate addresses properly when a thread executes code from user-space memory while inside the kernel.
- Both the kernel and DUCK process now share the same `hat` so address space updates from either side can be reflected to the other. In particular, when the DUCK process maps a new page in the user address space, the kernel can see and use the page immediately without manual duplication.

Unfortunately, the current implementation for combining address spaces crashes the kernel whenever stack growth in the DUCK process crosses page boundaries. We suspect this is related to the way the SPARC *register window spill exception* [6] is handled in Solaris. Future work is required before this approach can be implemented reliably.

4.1.3 Address Space Duplication

In this section we describe the approach used in our prototype implementation for direct user code invocation, which we call *address space duplication.* This approach also adjusts the memory layout of the DUCK process as well as duplicates address translation information into the kernel's `hat` during initialization. However, we no longer share the kernel's `hat` with the DUCK process. Instead, we insert a few lines of code to the function body of `hat_memload` and `hat_unload`, so that every time the DUCK process updates its `hat`, the same change will be applied to the kernel's `hat`. We show the pseudo code of the new implementation of `hat_memload` below:

```
void hat_memload(hat, va, pa) {
    if (duck_proc && hat == duck_proc->as->hat) {
        hat_memload(kas->hat, va, pa);
    }
    /* hat_memload's original code goes here */
```

```
}
```

Its first parameter points to the `hat` object the function operates on; parameters *va* and *pa* are the virtual and physical addresses to be mapped (the actual implementation has more parameters which are omitted here). The `duck_proc` variable is global and points to a process control block. Its value is initialized to zero when the kernel boots, but `duck_init` sets it to the DUCK process's control block, and `duck_fini` resets it to zero[1].

The function `hat_unload` is modified similarly. Both `hat_memload` and `hat_unload` monitor activities on the DUCK process's `hat` and repeat them on the kernel's `hat` after the process starts. Monitoring stops when the process terminates. In this way the process's address space can be precisely mirrored into the kernel address space.

There is one remaining issue. Normally, if a page has not been mapped to a physical frame when it is referenced, the page fault handler is responsible for loading the frame and setting up the mapping (Section 4.1.2). The handler assumes that only threads in user mode reference user-space pages and relies on this assumption to resolve page faults. Therefore, referencing these pages by a kernel thread prevents the handler from working properly. A solution is to pre-load all pages of the DUCK process and disable paging for these pages by calling the `mlockall` system call during initialization. Because all the pages are locked into physical memory, referencing them later will not trigger the page fault handler. Although such locking may interfere with the kernel's paging algorithms, doing this should not impose significant side-effects on overall system performance as the process should be the dominant application on a dedicated server.

4.1.4 Summary

In this section, we discuss why direct user code invocation is not as easily achieved on Solaris/SPARC as on other platforms. Address space switching, combining ad-

[1]For simplicity, in this prototype implementation, any calls to `duck_init` fail if *duck_proc* is non-zero. This means only one DUCK process is allowed to run at any time. However, we do not see any obstacles that prevent running multiple DUCK processes.

dress spaces, and address space duplication, are then discussed as three possible approaches. Our prototype uses address space duplication which monitors activities on the process's `hat` and replays them on the kernel's `hat`. As the only approach that provides a complete solution among all the methods we explored, address space duplication is efficient as well. First of all, it does not involve per-callback-invocation overhead. Second, although extra work is required when updating the process's `hat`, it should not introduce significant overhead since updates are expected to be infrequent. Finally, `hat` updates are relatively inexpensive when compared with operations associated with updates such as paging and physical memory management.

4.2 Implementing Zero-copy Buffer Management

In this section, we discuss the implementation of zero-copy buffer management on Solaris. Figure 4.3 depicts the components of a Solaris protocol stack and data structures used for buffer management, from which we can see a close resemblance with Figure 3.1. Solaris organizes protocol stacks into *streams*. A *stream head* is always put at the top of a protocol stack and is responsible for constructing message buffers to be sent and for consuming buffers being received, as well as for interacting with the user processes. A device driver usually sits at the bottom end of a protocol stack, controlling the network interface card associated with the stack.

On Solaris each message buffer is represented by one or more objects of type `mblk`. The `cont` field of the object points to the next `mblk` in the same message buffer or to NULL if it is the last one in the buffer. Each `mblk` contains a pointer named `datap` pointing to a `dblk` object, which in turn contains the base address and the length of a data area. In addition, two fields in the `mblk`, `rptr` and `wptr`, record the beginning and the end of the payload in the data area held by its `dblk` object (not shown in the figure). This organization provides significant flexibility. For instance, because several `mblk`s can refer to the same `dblk`, and each `dblk` has an additional field recording the number of `mblk`s referring to it, data areas can be

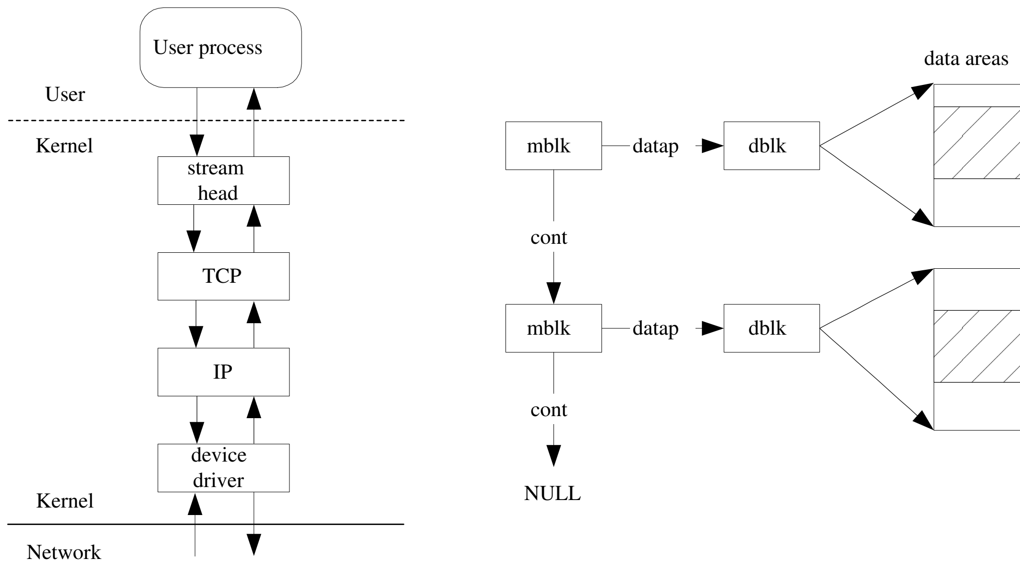

Figure 4.3: Solaris network protocol stack and buffer management

shared and reused without copying data.

Implementing the DUCK buffer management interface with these data structures is straightforward. We specify `mblk` as the implementation type of *both* DUCK buffer heads *and* buffer blocks and implement most DUCK functions with a single line of code. They are listed below and should be self explanatory (the code checking for invalid pointers is omitted):

```
mblk * duck_blk_first(mblk * sb)      { return sb; }
mblk * duck_blk_next(mblk * blk)      { return blk->cont; }
void * duck_blk_get_data(mblk * blk)  { return blk->rptr; }
size_t duck_blk_get_len(mblk * blk)
{ return blk->wptr - blk->rptr; }

void   duck_blk_adjust(mblk * blk, void * begin, void * end)
{ blk->rptr = begin; blk->wptr = end; }
```

To calculate the total data length of a message buffer, `duck_sb_get_len` loops

over all `mblk`s and returns the sum of the data length of all `mblk`s. The functions `duck_sb_alloc` and `duck_sb_free` are easy to implement, too, because the kernel already provides functions for buffer allocation and destruction.

The implementation of `duck_recv` and `duck_send` is almost identical to `recv` and `send`, except that the code that copies data between user and kernel address spaces needs to be modified to avoid copying data. To modify `recv`, we find the code that copies the `mblk`'s payload data into the user address space and destroys the `mblk`. Instead of doing this, `duck_recv` passes the buffers and delivers their pointers to the function that calls `duck_recv`. Similarly, instead of constructing new message buffers and copying data into these buffers, `duck_send` directly consumes the message buffer provided by the caller function. Meanwhile, all other logic for both `send` and `recv` such as flow control and control message handling is unmodified. On Solaris data copying is only performed by the stream head and therefore no layers below it need to be modified. In the actual implementation we duplicate the code of the stream head to `duck_recv` and `duck_send` and make changes there to reduce modifications to existing kernel code.

Chapter 5

Performance Evaluation

Micro-benchmarks are used to compare the performance of DUCK against traditional methods of performing network I/O. We use a simple UDP packet forwarder in the experiments. It is a user-level program written in C++ and compiled by GCC. This program is similar to the second example discussed in Section 3.3. It receives UDP packets from one network interface and immediately forwards them to another. When using traditional system calls, two system calls and therefore four mode switches and two data copies are required for each packet being forwarded. Because the main loop does nothing but calls `send` and `recv`, this simple forwarder helps us to understand the maximum benefit DUCK can provide for network-centric applications.

The forwarder runs on a Sun Blade 1500 workstation [5] with one single-core 1 GHz UltraSPARC IIIi processor, 2 GB of RAM, and an Intel PRO/1000 Ethernet NIC with two 1 Gigabit interfaces. We use a snapshot of the OpenSolaris [4] source code with timestamp "20060918," and modify the kernel functions `hat_memload` and `hat_memunload` as described in Section 4.1.3 to enable address space duplication. We then compile the snapshot using default configuration flags, install the compiled kernel to the machine, and use it throughout the experiments. All DUCK functionality is encapsulated in a Solaris kernel module [7]. Once loaded, the module inserts a new system call and links several new kernel functions into the running system. A user library is also provided to facilitate the user's inter-

action with the kernel module for DUCK initialization, user callback invocation, and kernel function calling from the callback.

A client machine is used to run a sender program which transmits packets to the forwarder at a specified rate, as well as to run a receiver program that receives and discards packets from the forwarder. Figure 5.1 illustrates this configuration. This machine has four Intel Xeon 2.8 GHz processors and 4 GB of RAM. It is chosen to be powerful enough so that it will not be overloaded before the SPARC machine. There are also two 1 Gigabit Ethernet interfaces on this machine. They are connected to the SPARC machine through a 1 Gigabit switch. The IP addresses of the four interfaces are deliberately assigned so that one interface on each machine belongs to one network, and the other belongs to another. The client machine sends packets to the forwarder through the first network, and the server in turn reflects them back to the receiver through the second network. The switch isolates two networks through VLANs. Because it can provide 1 Gigabit of throughput for each network, we can ensure that the network does not throttle system performance. This is important because DUCK may show little performance improvement if components other than the processor are the bottleneck (the maximum sending rate in our experiments is 80,000 1000-byte packets per second as shown in Figure 5.5, which corresponds to $80,000 \times (1,000 + 66) \times 8 = 682.24$ Mbps bandwidth on the links, where 66 is the total per-packet overhead including UDP headers, IP headers, and Ethernet overhead [23]).

Both small and large UDP message sizes are used to study the performance under different circumstances. The maximum message size used is 1000 bytes per packet. For each message size we vary the output rate of the sender and measure the input rate of the receiver. Because there is no packet loss between the forwarder and the receiver, we can regard the forwarder's output rate as equal to the receiver's input rate.

There are several considerations in the design of the experiments. First, we use UDP instead of TCP as UDP is *not* congestion controlled. One purpose of the experiments is to analyze the server's overloading behavior, but TCP automatically reduces sending rates on packet losses, preventing the client from offering rates

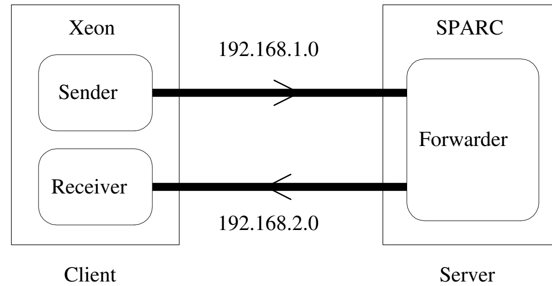

Figure 5.1: Evaluation Setup

Boxes represent machines; rounded boxes are programs running on the machine; solid bars with arrows indicate network connections and the directions of data flow.

beyond the server's capability. We choose UDP also because we hope to study the performance gain by reducing mode switching and data copying overhead, and UDP involves less other overhead than TCP. Additionally, we perform all measurements on the client machine to avoid affecting the server and the results.

5.1 Evaluation Results

We compare performance among the following four schemes:

- User-RS. The forwarder invokes `recv` and `send` system calls in user mode, as traditional applications do with this scheme. There are four mode switches and two data copies required for each forwarded message.

- Kernel-RS. The forwarder uses a DUCK user callback to forward packets from inside the kernel. It calls the in-kernel implementations of `recv` and `send` system calls to receive and send data. Therefore, two data copies but no mode switches are required per packet.

- Zero-copy. The DUCK user callback forwards packets using `duck_recv` and `duck_send`.

- IP-forwarding. In this scheme packets are forwarded using the Solaris kernel's built-in packet forwarder [2]. Because it works at the IP layer and runs inside the kernel, it should be the fastest among all the four schemes. We include this scheme because it help us ensure the network is *not* a bottleneck for the other three schemes as long as their throughput does not exceed that of IP-forwarding. It can also be used as an upper bound on performance.

We plot the performance results of these schemes in Figures 5.2 through 5.5. The figures show the results for packet sizes 10, 100, 500, and 1000. For each data point we repeat a 20-second experiment three times. Because we observe little fluctuation in the results, it is not necessary to execute longer experiments or more iterations. The average output rates are reported with standard deviations shown as error bars.

From these figures we can see consistent performance improvements from both mode switching and data copying avoidance, although their absolute throughput in packets per second decreases as packet sizes increase. When the packet size is 10 bytes, the peak throughput of kernel-RS is 15.5% higher than that of user-space forwarding; zero-copy provides an extra 12.6% improvement as compared to user-RS, leading to a total gain in peak throughput of 28.1%. Table 5.1 summarizes the individual as well as total improvement under different packet sizes. Also listed in the table is the ratio of zero-copy improvement over kernel-RS improvement. We notice a rise in this ratio as the packet size increases. This is expected, because data copying introduces more overhead when the packet size is larger, which leads to better results for copying avoidance. Meanwhile, the overhead of mode switches remains constant regardless of packet sizes, causing the absolute gain using kernel-RS to be steady.

After the peak all curves decline due to overloading. We can see from all the figures that both kernel-RS and zero-copy decline steadily with a rate close to user-RS. This indicates that kernel-RS and zero-copy behave well under overload.

In summary, the experimental results show the performance improvements that mode switching and data copying avoidance can offer. The total gain in peak throughput ranges from 28.1% to 43.8% across the examined packet sizes.

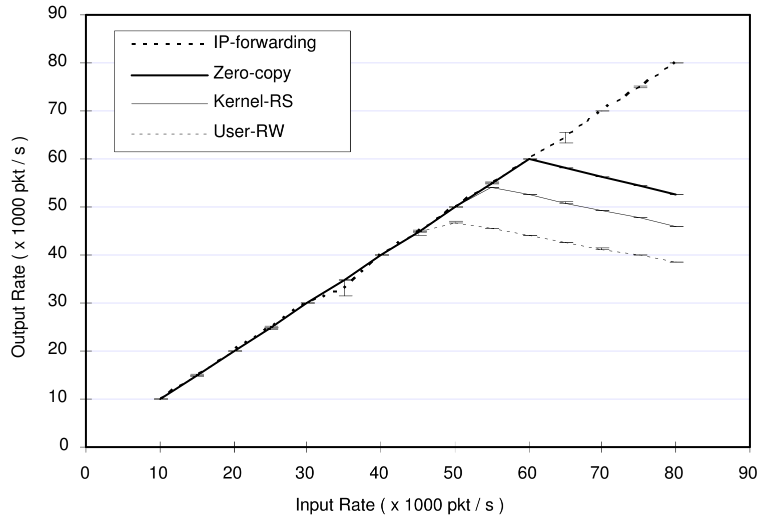

Figure 5.2: Throughput for 10 bytes per packet

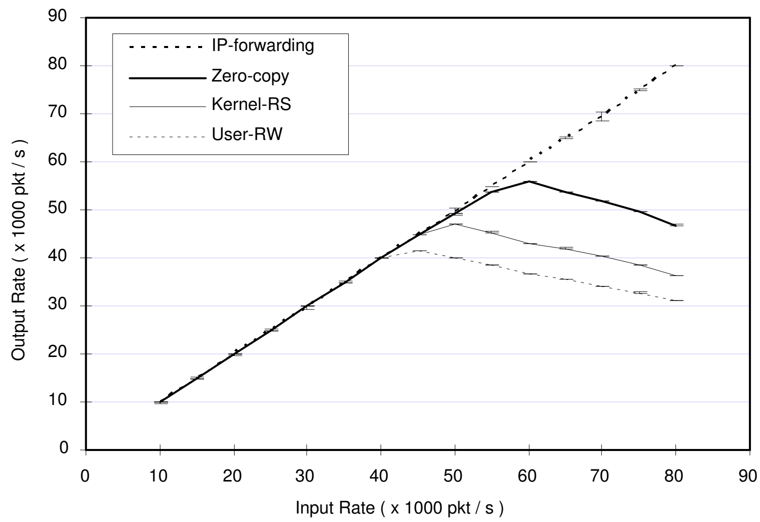

Figure 5.3: Throughput for 100 bytes per packet

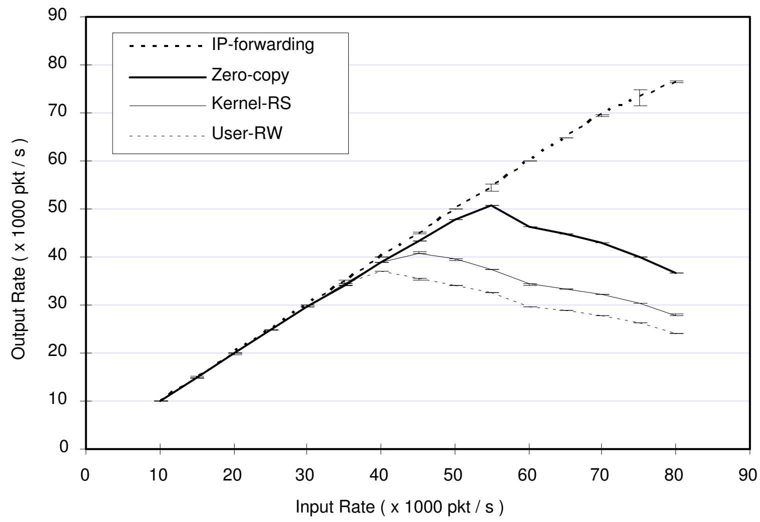

Figure 5.4: Throughput for 500 bytes per packet

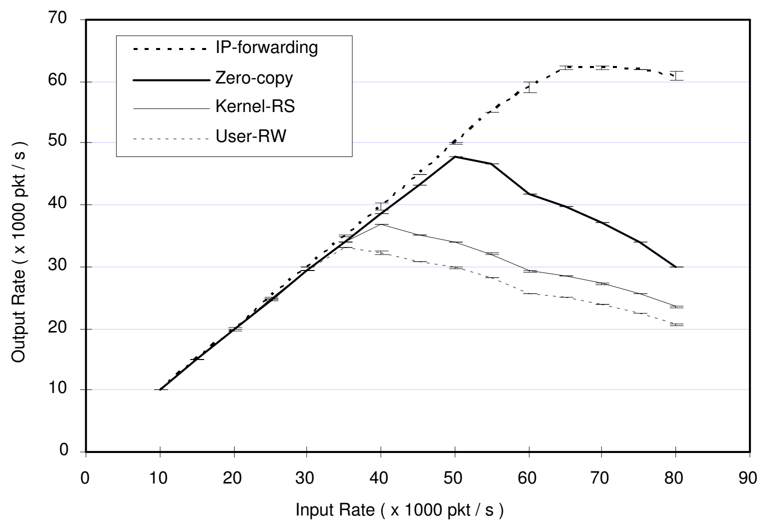

Figure 5.5: Throughput for 1000 bytes per packet

packet size	kernel-RS	zero-copy	total	$\frac{\text{zero-copy}}{\text{kernel-RS}}$
10	15.5%	12.6%	28.1%	0.8
100	13.3%	21.5%	34.8%	1.6
500	10.6%	26.8%	37.3%	2.5
1000	11.5%	32.3%	43.8%	2.8

Table 5.1: Peak throughput improvement under different packet sizes

Chapter 6

Conclusion and Future Work

Security concerns for conventional kernel-based operating systems are less relevant for dedicated servers where only one dominant service runs on a computer system. We argue that for such configurations security can be relaxed to obtain better performance. We are particularly interested in reducing system call overheads by executing a part of the primary application from inside the kernel. The code can then use plain function calls to request operating system services when running in the kernel, thereby reducing or even eliminating mode switching and data copying overheads.

One challenge with the proposed DUCK framework is how to design an application programming interface that is general, efficient, and easy to use. In this thesis, we present the design of the DUCK API, which includes functions for both direct user code execution and zero-copy buffer management. An abstract model adapted by many operating systems for network buffer management is given, based on which the buffer management interface is designed.

We have implemented DUCK prototypes on Solaris/SPARC. An efficient way to implement direct user code invocation is through memory sharing between the kernel and user processes. However, because Solaris/SPARC uses separate address spaces, achieving memory sharing is difficult. In this thesis, we study the SPARC architecture and the Solaris virtual memory subsystem, and discuss three potential techniques to enable memory sharing: address space switching, combining, and

duplication. Although address space duplication is the only complete solution we have explored so far, it is efficient as well. To share memory between the user and kernel address spaces, it first adjusts the memory layout of the DUCK process, removing all conflicting areas of the two address spaces. The process's address translation information is then duplicated into the kernel. When the process is running, any change to its `hat` data structure is replayed into the kernel's `hat`. By using this method we implement the DUCK prototype by changing only a few lines of code in the stock Solaris kernel. We also discuss zero-copy buffer management for Solaris, which is straightforward to implement.

Finally, experiments are conducted using the DUCK prototype on Solaris/ SPARC. A simple UDP forwarder is developed for the purpose of micro-benchmarks. The results show 28.1% to 43.8% peak throughput improvement when compared to conventional system call invocations.

Some possible areas for future work are:

- In the current approach, we modify `hat_memload` and `hat_unload` so that they can trace updates on the DUCK process's `hat` to ensure the kernel maps to the same pages (Section 4.1.3). However, when the callback running within the kernel is about to allocate a new virtual page in the user address space, the new `hat_memload` implementation might operate on the kernel's `hat` and not duplicate the update to the process's `hat`. This is because in this situation the `hat` object passed to `hat_memload` might be the kernel's but not the process's. Solutions to this problem are difficult and left as future work. Our current prototype works properly as long as the callback does not request new virtual pages in the user address space.

- We hope to compare the performance of DUCK with other approaches described in Chapter 2.

- In addition to micro-benchmarks presented in Chapter 5, macro-benchmarks are necessary to evaluate the performance benefits of DUCK under realistic settings.

- Along with the prototype implementation on Solaris/SPARC, we have also investigated Solaris/IA-32, FreeBSD/IA-32, Linux/IA-32, Linux/IA-64 (i.e. 64-bit Intel Architecture), and Linux/PowerPC, by examining their abilities to support DUCK. We believe DUCK can be implemented on these platforms easily. However, more operating systems and processors need to be investigated to verify the general applicability of DUCK.

- We are interested in integrating DUCK with complementary techniques. For example, Singularity's static verification and application signing [11] might be useful in reinforcing security without significantly hampering the system's run-time performance.

- Aspects other than performance should be qualitatively analyzed. We hope to deploy the DUCK framework into real applications and receive feedback from developers, to confirm DUCK's usability, effectiveness, and flexibility under different circumstances.

Bibliography

[1] E-mail archive: Explaining Linux `splice()` and `tee()`,
http://kerneltrap.org/node/6505.

[2] IP forwarding in Solaris 10 is enabled or disabled by the `routeadm` command. Sun Microsystems, Solaris Tunable Parameters Reference Manual,
http://docs.sun.com/app/docs/doc/817-0404.

[3] Red Hat, Red Hat Content Accelerator 2.2, 2002,
http://www.redhat.com/docs/manuals/tux/TUX-2.2-Manual.

[4] Sun Microsystems, OpenSolaris, http://opensolaris.org/.

[5] Sun Microsystems, Sun Blade 1500 Specification,
http://sunsolve.sun.com/handbook_pub/Systems/SunBlade1500/spec.html.

[6] *The SPARC Architecture Manual, Version 9.* SPARC International, 1994.

[7] *Writing Device Drivers.* Sun Microsystems, 2002.

[8] *UltraSPARC© III Cu User's Manual.* Sun Microsystems, 2004.

[9] *Intel© 64 and IA-32 Architectures Software Developer's Manual, Vol. 1, Ch. 3.* Intel, 2006.

[10] *Solaris(TM) Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd Edition).* Sun Microsystems, 2006.

[11] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing Process Isolation. In *2006 Workshop on Memory System Performance and Correctness*, pages 1–10. ACM Press, 2006.

[12] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *15th Symposium on Operating Systems Principles*, pages 267–284, 1995.

[13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *15th Symposium on Operating Systems Principles*, pages 251–266, 1995.

[14] Hsiao Keng and J. Chu. Zero-copy TCP in Solaris. *Proceedings of the USENIX Annual Technical Conference*, 1996.

[15] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 1992.

[16] R. M. Love. *Linux Kernel Development*. Sams Publishing, 2003.

[17] Toshiyuki Maeda and Akinori Yonezawa. Kernel Mode Linux: Toward an Operating System Protected by a Type Theory. In *8th Asian Computing Science Conference*, pages 3–17, 2003.

[18] Michal Ostrowski. A Mechanism for Scalable Event Notification and Delivery in Linux. Master's thesis, David R. Cheriton School of Computer Science, University of Waterloo, 2000.

[19] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[20] V. Pareto. Cours d'économie politique. *Reprinted as a volume of Oeuvres Complètes (Droz, Geneva, 1896–1965).*

[21] Marcel-Catalin Rosu and Daniela Rosu. Kernel Support for Faster Web Proxies. *Proceedings of the USENIX Annual Technical Conference*, 2003.

[22] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with Disaster: Surviving Misbehaving Kernel Extensions. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, 1996.

[23] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley Professional, 1993.

[24] Andrew S. Tanenbaum. *Modern Operating Systems 2nd Ed.* Prentice-Hall, 2001.

[25] Moti N. Thadani and Yousef A. Khalidi. Techinical Report: An Efficient Zero-Copy I/O Framework for UNIX. *Sun Microsystems Lab, SMLI TR-95-39*, 1995.

[26] Arjan van de Ven. kHTTPd: Linux HTTP Accelerator, http://www.fenrus.demon.nl/.

[27] 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 ACM Symposium on Operating Systems Principles.* ACM Press, 1995.

[28] Erez Zadok, Sean Callanan, Abhishek Rai, Gopalan Sivathanu, and Avishay Traeger. Efficient and Safe Execution of User-Level Code in the Kernel. In *19th IEEE International Parallel and Distributed Processing Symposium*, page 221. IEEE, 2005.