Chapter 8
The Exokernel Operating System and Active Networks

Timothy R. Leschke
University of Maryland, Baltimore County, USA

ABSTRACT

There are two forces that are demanding a change in the traditional design of operating systems. One force requires a more flexible operating system that can accommodate the evolving requirements of new hardware and new user applications. The other force requires an operating system that is fast enough to keep pace with faster hardware and faster communication speeds. If a radical change in operating system design is not implemented soon, the traditional operating system will become the performance bottle-neck for computers in the very near future. The Exokernel Operating System, developed at the Massachusetts Institute of Technology, is an operating system that meets the needs of increased speed and increased flexibility. The Exokernel is extensible, which means that it is easily modified. The Exokernel can be easily modified to meet the requirements of the latest hardware or user applications. Ease in modification also means the Exokernel's performance can be optimized to meet the speed requirements of faster hardware and faster communication. In this chapter, the author explores some details of the Exokernel Operating System. He also explores Active Networking, which is a technology that exploits the extensibility of the Exokernel. His investigation reveals the strengths of the Exokernel as well as some of its design concerns. He concludes his discussion by embracing the Exokernel Operating System and by encouraging more research into this approach to operating system design.

INTRODUCTION

The traditional operating system (OS) is seen as providing both management and protection of resources. As a manager, the OS controls how resources such as I/O devices, file-storage space, memory space, and CPU time get allocated. As a protector, the traditional OS controls how processes use these resources to avoid errors. Because the operating system's role is so important, it is the one

DOI: 10.4018/978-1-60566-850-5.ch008

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

program that is always running on a computer. The heart of the operating system is called the *kernel*.

Within a traditional computer system, the complexity of the hardware is masked behind the abstractions provided by the operating system. Although this abstraction prevents user programs from interacting with the hardware directly, user programs benefit by having a single interface that they can interact with. It is easier for a user program to interact with one operating system interface rather than developing software that must know how to interact with the different hardware components that could be present within a computer system at any one time. Because the operating system is the main interface between user programs and the raw hardware, the traditional operating system must be involved in every software-hardware interaction.

Although the traditional operating system is desirable because it provides a single interface for other software to interact with, it is ironic that this efficient interface should be the root cause of the modern computer system's performance bottleneck. By presenting a single interface to user applications, the traditional operating system is the middle-man between all user processes and the computer hardware. By being the middle-man, the operating system tries to be all-things to all-processes. This "all-things to all-processes" approach is precisely why the design of the traditional operating system is flawed. So long as the operating system is designed to meet the minimum of a broad spectrum of operational requirements, the optimization of any one process is very unlikely. Therefore, while every application might run on a traditional operating system, few applications run well (reaches it maximum performance level) on a traditional operating system.

A team of researchers at the Massachusetts Institute of Technology (MIT) has challenged the traditional operating system design with their experimental operating system - the Exokernel Operating System. Their new approach is to separate management of resources from the protection of those resources. The Exokernel Operating System provides only protection and multiplexing of resources while allowing user processes themselves to provide the management and optimization of that resource. As Engler, Kaashoek, and O'Toole say "Applications know better than operating systems what the goal of their resource management decisions should be and therefore, they should be given as much control as possible over those decisions" (Engler, Kaashoek, & O'Toole, 1995). By separating management from protection, the abstraction provided by the traditional operating system has been eliminated. Likewise, the door to process optimization has been opened and great advances in operating system speed and flexibility have become possible.

As we investigate the Exokernel Operating System, we will discuss how it is possible to separate management from protection while still multiplexing resources within a secure environment. We will discuss select Exokernel functions such as downloading code into the kernel, reading and writing to disk memory, exception and interrupt handling, interprocess communication, tracking resource ownership, protecting and revoking resource usage, and resource management. Networking with an Exokernel will be discussed as we look at packet sending, packet receiving, the naming and routing of packets, and network error reporting. Lastly, as an example of other technologies that benefit from the Exokernel, we will briefly explore the emerging technology of Active Networking and see how the Exokernel is the ideal operating system upon which to build this new technology.

In response to the Exokernel, we will investigate why the Exokernel has not been widely accepted as the main-stream approach to Operating System design. We will investigate the potential issues with providing customer support to an extensible operating system like the Exokernel. We will argue against removing *all* management from the kernel. We will discuss

code optimization and who is best suited for this task. We will concede that some processes - like multithreaded applications - perform worse in an Ekokernel environment. We will finally question if extensibility is even the solution to the growing problem of operating systems becoming the bottleneck of computer system performance. We will conclude that despite some of the issues that make the Exokernel commercially unacceptable at this time, the Exokernel's enhancements outweigh its shortcomings and therefore we encourage the reader to embrace its approach.

PROBLEM DESCRIPTION

The need for a new operating system design has been motivated by two forces - the need for speed and the need for flexibility. These two forces are explained next.

The Need for Flexibility

Trying to define the term "operating system" is an ongoing debate. Some say the operating system is simply that software program that sits between the hardware and the other user programs. It is that program that provides a simple interface that allows processes to interact with the hardware. Others say the operating system is the manager and protector of computer resources. It manages how every computer resource gets used and also protects the resources against improper use.

Some have argued that an operating system is defined by the manufacturer. As was stated previously, an operating system is "whatever comes in the box when it is purchased" (Leschke, 2004). This means that if a user-manual is included in the package, then the user-manual is a formal part of the operating system. Furthermore, whatever software is bundled-with or integrated-with the operating system is also a part of the operating system. For example, if a text editor or an entire suite of office application software is integrated into the operating system, then these items must also be considered to be part of the definition of an operating system.

Regardless of which definition of an operating system you agree with, we must all agree that the traditional operating system is that program that abstracts the hardware and offers the user a single interface with which to interact with. It is this single interface approach that has the traditional operating system literally caught in the middle between two different forces. One force is the need for a more flexible operating system. As 64 bit processors replace 32 bit processors, the need for an operating system that can interact with this new hardware has grown. Larger memory devices - such as hard drives that are now measured in terrabytes - as well as fiber-optic networks, high-bandwidth networks, and new data storage devices, have also demanded a more flexible operating system.

In addition to the flexibility demands being made by hardware, software applications are also demanding a more flexible operating system to meet their needs. For example, realistic gaming programs are requiring access to hardware in non-traditional ways. Portable computing and communication devices are requiring the ability to up-load and down-load data more easily. Database systems want to be able to access memory in their own ways, and real-time systems demand a specific performance level that can be best achieved through a flexible operating system. All of these have combined into one force that is demanding the traditional operating system become more flexible to change.

The traditional operating system could be pulled-apart by these forces if it continues to remain rigid in its design. Therefore, the traditional operating system needs a more flexible design.

The Need for Speed

Gene Amdahl has provided us with "Amdahl's Law" - one of the fundamental laws of computer

architecture. His law states that the increased speed that is gained by using an improved mode of execution is restricted by how much the new mode is actually used. For example, if an execution mode that is used 10% of the time is modified to be 100% faster, the entire efficiency of the system will only increase by about 5%. On the other hand, if a mode of execution that is used 90% of the time is modified to be 50% faster than it was before, the entire system will experience a 45% increase in efficiency. This means that a small improvement in a mode of execution that is used frequently will have a much larger impact on performance than a large improvement made in a mode of execution that is seldom used. If one wants to have the greatest impact on the efficiency of a system, one should try to improve those processes that account for the greatest share of the execution time. This means designers of computer systems must pay attention to changes in technology, identify those technologies that have had the greatest speed-up, and then make sure the old technologies that the new technology has to interact with do not impede the speed-up gained by the new technology.

As an example, the Central Processing Unit (CPU) is one of those technologies that enjoyed some great improvement in the recent past. Although the speed and capacity of the newest CPUs - as well as other key hardware components - have been increasing, if the rest of the computer system cannot keep pace with this increase, then the full benefit of the increase will not be realized. Just like the CPU, another part of the computer system that might not be keeping pace with the ever increasing speeds of hardware is the Operating System. As Engler, Kaashoek, and O'Toole explain, "Traditional operating systems limit the performance, flexibility, and functionality of applications by fixing the interface and implementation of operating system abstractions such as interprocess communication and virtual memory" (Engler, Kaashoek, & O'Toole, 1995). Furthermore, as another example, John Ousterhout states "Operating systems derived from UNIX use caches to speed up reads, but they require synchronous disk I/O of operations that modify files. If this coupling isn't eliminated, a large class of file-intensive programs will receive little or no benefit form faster hardware" (Ousterhout, 1989). The new era of operating system design demands that operating systems keep pace with faster hardware or risk being the cause of computer system speeds being stagnant.

The Approach

As Lee Carver and others state, an operating system is a necessary evil (Carver, Chen, & Reyes, 1998). Therefore, computers will have an operating system of one sort or another. The growing requirements that operating systems become faster and more flexible have encouraged many researchers to consider operating systems with radical designs. One of the new designs is an extensible operating system.

An extensible operating system is simply an operating system that is flexible to change. The needs of the underlying hardware can be better met by an operating system that can be easily modified. The needs of the user applications can also be better met by an operating system that can be easily modified. Speed is achieved by an extensible operating system because the system can be easily changed and optimized. The speed and flexibility issue are both addressed by an extensible operating system. By providing the hope of increased speed and a more flexible implementation, the approach offered by an extensible operating system, at least momentarily, seems to be one way to prevent the increased speed of computer systems from becoming stagnant while also addressing the rapidly changing needs of user applications.

A group of researchers at the Massachusetts Institute of Technology have implemented their version of the extensible operating system in what they have called the Exokernel Operating System.

The main approach of the Exokernel is to attempt a very clear separation between management and protection of resources. Management is left to the user processes - because user processes themselves know how to better utilize the resources under their control. Protection of the resources is provided by the Exokernel, but in a very minimal amount so as not to interfere with any attempts to optimize the user processes. The end result is an operating system that is easily modified to meet the changing needs of user processes while also allowing real optimizations to occur - which result in major speed-ups in process execution times. We will be taking a closer look at the separation of management from protection as we investigate the Exokernel Operating System.

THE EXOKERNEL SOLUTION

Previously we stated the Exokernel only provides protection and proper sharing of resources. According to Dawson Engler, the process of protecting resources consists of three major tasks; 1) tracking ownership of resources, 2) ensuring protection by guarding all resource usage or binding points, and 3) revoking access to resources (Engler, 1998). Lesser tasks of the Exokernel include; protecting a processes ability to execute privileged instructions, protecting the processing of the central processing unit, and protecting physical memory - which includes writes to "special" memory locations that are used by devices, and protection of network devices. We stress that while the Exokernel is providing protection of these activities, it is not getting involved in the micro-management of these activities. The micro-management of these activities is provided by user-processes that are located in user-space rather than kernel space. In other words, the Exokernel will grant a user-process access to a resource, and it might revoke that access if necessary, however, it does not regulate how the resource is used. This means that a user-process could use a resource improperly if it wanted to, but it also means the user-process has the freedom to optimize how the resource is utilized. This added freedom means software engineers need to develop computer programs that police themselves to ensure that the shared resources provided by an Exokernel are used properly.

Dawson Engler provides a better example of the separation of management from protection as he explains how the Exokernel protects physical memory. The accessing of physical memory through read and write requests are *privileged* instructions for a traditional operating system. The traditional kernel stands guard over the memory and verifies every read/write request to ensure each request has the proper access rights. Because the traditional operating system stands between the user-processes and the physical hardware, when a user-process wants to send a message to the hardware, it passes the message to the traditional operating system and the operating system then passes the message to the hardware on behalf of the user process. When a user-process passes a message to the operating system in this manner, it is called a system-call. One of the down-sides of the traditional system-call is that user-processes cannot directly execute privileged instructions. Because the traditional operating system is the constant middle-man that gets involved in every system-call, the overall efficiency within the entire computer system is greatly reduced.

In response to the issue of reduced global efficiency, Dawson Engler explains that the Exokernel's solution is "to make traditionally privileged code unprivileged by limiting the duties of the kernel to just these required for protection" (Engler, 1998). This means that the Exokernel allows user-processes to have much more direct access to memory. The Exokernel still gets involved a little, but only enough to ensure the memory access is "safe". Once safety is guaranteed, a user-process is allowed to directly access the hardware itself.

In the next pages, we continue to illuminate some of the unique aspects of the Exokernel Oper-

ating System. The aspects that we have chosen to look at are by no means a comprehensive list, but they are intended to leave the reader with a good understanding of the Exokernel's approach.

Tracking Ownership of Resources

The allocation of a resource is actually accomplished by what the research group calls the Library Operating System (LibOS). This LibOS is outside of the kernel; therefore the kernel is only minimally involved. The kernel gets involved just enough to record the ownership information associated with a resource. For example, when physical memory gets allocated, the kernel keeps track of which process the resource has been allocated to and which processes have 'read' and 'write' permissions (Engler, 1998). As a way to retain its minimal involvement, the Exokernel records resource allocations in what the research group calls an *open bookkeeping policy*. Through this open bookkeeping policy, as Engler explains, resource allocation records are made available to all user processes in read-only mode. This allows the user processes to look-up for themselves if the resource that they want is actually available. This means the kernel does not need to be interrupted by a process that keeps requesting a resource that is currently unavailable.

Ensuring Protection by Guarding all Resource Usage or Binding Points

It is very important for a process to retain use of a resource until it is done using it. For example, a process should be able to securely use a block of memory until the process decides to de-allocate it. The Exokernel uses what are called "secure-bindings" when binding a resource to a process.

A secure-binding separates the authorization to use a resource from the actual use of that resource. Authorization to use a resource is granted or denied when the resource is first requested. Once the process has the authority to use a resource, it retains this authority until it gives it up (Engler, Kaashoek, & O'Toole, 1995). The Exokernel is only minimally involved in this process as it only provides the authorization to use the resource and it does not get involved in the ongoing management of the use of that resource.

Furthermore, the Exokernel can provide secure-bindings without any special knowledge of what it is binding. The semantics of binding a resource to application software can get very complex. However, the Exokernel does not get involved in the details of the binding. It only gets involved to the extent that it can provide the security associated with that binding. As Engler, Kaashoek, and O'Toole say, "a secure binding allows the kernel to protect resources without understanding them" (Engler, Kaashoek, & O'Toole, 1995).

Revoking Access to Resources

Although a secure-binding, in theory, allows a process to use a resource until it is done with it - in reality, there still must be a way for the operating system to force a revocation of the resource binding under certain conditions. Unlike the Exokernel, when a traditional operating system brakes a resource binding it does so by what is known as *invisible revocation*. With an invisible revocation, the resource binding is simply broken and the process has no knowledge of the circumstances that prompted the revocation. A disadvantage of using invisible revocation is that operating systems "cannot guide de-allocation and have no knowledge that resources are scarce" (Engler, Kaashoek, & O'Toole, 1995).

When an Exokernel breaks a secure binding, it uses a technique that the researchers have named *visible revocation.* With visible revocation, communication occurs between the kernel and the process. Because of this communication, the process is informed of the need to have the resource binding broken. By being warned of the resource revocation before the event, the process

can prepare for it by saving any data that it needs and bring itself to a stable state. For example, a process may be asked to give up a page of memory and it may not matter which page of memory it de-allocates. Because the process is kept informed of the resource de-allocation, the process may be allowed to simply change a few pointers to reflect the change, or it may be allowed to choose which page of memory it gives up. It may also choose to write that page of memory to disk to free-up the memory requested by the kernel. In either case, the process cooperates with the kernel, and by doing so, the revocation of the resource is less intrusive for the process.

Just like in a traditional operating system, a process that is not cooperating with the kernel's request to de-allocate a resource must, on occasion, be forced to comply with the kernel's request. When a secure-binding has to be broken by force, it "simply breaks all existing secure bindings to the resource and informs the Library OS" (Engler, Kaashoek, & O'Toole, 1995).

Management by User Level Library

Previously we stated, the Exokernel Operating System's kernel is responsible for providing protection of resources while the management of those resources is left up to another entity. This other entity is known as the "user level library operating system", or "LibOS" for short. This LibOS lies outside the kernel where it is available to user processes.

The LibOS can be thought of as being very similar to a traditional operating system in that the LibOS is the middle-man between the user processes and the actual hardware. Like the traditional operating system, the LibOS provides the abstraction that user processes interact with when they want to communicate with the hardware. However, unlike a traditional operating system, a LibOS can be customized to fit the needs of the software applications. This customization leads directly to optimization, which in turn leads to a much more efficient operating system. Furthermore, since a LibOS is written with a specific user process in mind, a LibOS does not have to be all-things to all-processes as we stated was true of a traditional operating system. A LibOS can be simple and more specialized, primarily because "library operating systems need not multiplex a resource among competing applications with widely different demands" (Engler, Kaashoek, & O'Toole, 1995). So far as the LibOSs use standardized interfaces, these LibOSs allow for applications to be easily ported to different computing hardware. Because the LibOSs are so specialized, one may wonder if this leads to a lot of extra code in user space. One may also wonder if some of this code is also redundant. The Exokernel addresses this concern by what the researchers call "shared libraries".

Shared Libraries

Not all of the user level libraries have to be specialized code that is written for a specific process. Different processes can often reuse the same code that is written for another process. Therefore, the Exokernel allows processes to share code in what it calls a shared library. By sharing code, the amount of disk-space and memory-usage can be significantly reduced. The disadvantage of sharing code with a LibOS is what Douglas Wyatt calls a "bootstrapping problem" (Wyatt, 1997). The problem is that the code that is needed to load the LibOS from disk into memory is actually found in the LibOS itself (which cannot be accessed until the LibOS is actually loaded into memory). The solution to the bootstrapping issue will be explained when we discuss the shared library server (section 3.7).

Another of the key issues that Douglas Wyatt has identified with a shared library system is what he calls "symbol resolution" (Wyatt, 1997). The issue arises from the fact that when a program is run for the first time, it needs to know particular memory addresses in order to run correctly. One way to address this problem is to load a shared

library into the same virtual address space so that the particular memory addresses will be known prior to program execution. Although this approach works, it is not the best approach to this issue.

The symbol resolution issue might be better solved by what Wyatt calls an "indirection table" (Wyatt, 1997). Rather than force a shared library to always be loaded into the same virtual memory space, the solution requires a table be used to record the required memory addresses. This table is then provided to each shared library which uses the data within the table to calculate the relative offset of the memory address that it is looking for. Using an indirection table allows the shared libraries enough flexibility to load themselves into any address space.

Implementing a Shared Library

Before a program loads a shared library, it checks the indirection table to see if the library is already loaded somewhere else. If it is loaded somewhere else, it simply updates its page table to include the location of the existing library. However, if a program checks the indirection table and does not find a reference to the library that it needs, it loads the library and updates the indirection table to reflect the change.

Using an indirection table solves the symbol resolution issue described by Wyatt, but it does come at a price. The price is the extra time that is now needed to check the indirection table for libraries that are already loaded. When one considers the benefits of using an indirection table, one can see that using an indirection table is an expense that pays for itself. One of the benefits of an indirection table is that it requires the system to use less memory, which translates into having fewer page faults. Fewer memory page faults mean programs can run faster. Shared libraries can be updated and improved, which makes the system more flexible to change. Furthermore, when a shared library is changed, it can be compiled independently of the other programs that interact with it. This mechanism can be also implemented in a multi page size environment (Itshak & Wiseman, 2008).

The Shared Library Server

The bootstrapping problem mentioned previously is an issue that arises from trying to load a library operating system into memory when the code needed to do this is actually found within the library operating system itself. The researchers have addressed this issue with what they have called the "shared library server" or SLS.

The shared library server is started as soon as the Exokernel is booted. The shared library server is responsible for communicating with applications that want to communicate with a shared library. This communication includes the ability to 1) open, read, and write files, 2) map files from disk, 3) open and read directories, and 4) perform basic input and output operations. This basic functionality is just enough to help a shared library overcome the bootstrapping problem and load itself into memory.

Interprocess Communication (IPC)

The passing of messages between processes, or what is known as interprocess communication (IPC), is used so frequently within an operating system that it is a potential performance bottleneck it if is not accomplished efficiently. The Exokernel accomplishes interprocess communication by what Benjie Chen calls "protected control transfer" (Chen, 2000). The Exokernel implements IPC by using secure registers to pass data. Passing data by using secure registers allows the communication to be immediate, which means the data gets passed between the processes without any need for the kernel to get involved. This implementation allows the Exokernel to provide protection - in the form of a secure register. The Exokernel keeps itself out of the management details while

still retaining the role of the protector of resources. Because this form of interprocess communication is immediate, the Exokernel enjoys a processing speed-up.

Exceptions and Interrupts

An exception or interrupt requires a traditional operating system to save register data to a more secure location in order to protect and preserve its current state of operation. The kernel also has to respond to the exception, which requires the exception to be decoded by the kernel and then specific code needs to be executed to handle the issue raised by the exception. Once the exception has been dealt with, the kernel has to restore the registers to their pre-exception state and start running the original program from a point where the program counter was just prior to the exception.

An Exokernel, on the other hand, handles exceptions and interrupts by getting less involved. For example, exceptions and interrupts that arise from hardware are handled directly by the applications themselves. The Exokernel only gets involved enough to save important register information to what Engler calls an agreed upon, user accessible, "save area" (Engler, 1998). The Exokernel saves the register data, loads the exception, then starts to execute at the memory address of the code that has been written specifically to handle the exception. This special code is understood to be located in what we have been calling the user level library.

The kernel's job is done as soon as the Library OS takes over the handling of the exception. As soon as the exception is handled, the original register data is written back to the registers from where they were stored in user-accessible memory. The normal program execution continues from where it left off just prior to the exception without any further assistance from the kernel. Therefore, the kernel gets involved just enough to provide protection of the current state of the registers, whereas the actual management of the exception is handled in user space, by the Library OS.

Disk I/O

Disk I/O – reading and writing to memory locations – is accomplished asynchronously in order to minimize the involvement of the kernel. It is the Exokernel's "exodisk" that handles all read and write requests. When an application needs access to memory, the exodisk simply passes the request off to the disk driver. After the read or write request is made, the calling application immediately regains control. Since the request is asynchronous, the calling application has the option of waiting for the memory request to complete, or it can continue without waiting for a completion response from the exodisk.

When the memory read or write request completes, the Exokernel is notified of this event by the requesting application. However, very little more is required of the Exokernel. The Exokernel retains its minimal involvement by helping pass the disk I/O request to the exodisk and allowing all of the details of the disk I/O request to be handled by user-level code found in the Library OS.

Downloading Code into the Kernel

Since one of the goals of the Exokernel is to be optimally efficient, one of the ways the Exokernel attempts this is by downloading code into the kernel. Downloading code into the kernel is not unique to the Exokernel. It is a technique that other operating systems have used as a way to minimize the cost of a context switch.

One of the main advantages of downloading code into the kernel is that it eliminates the need for code to make what are called "kernel crossings", which can require an expensive context switch (Engler, Kaashoek, & O'Toole, 1995). Context switches are undesirable because they can severely impede an application's execution speed. By eliminating kernel crossings, context

switches are reduced, and applications experience faster execution speeds.

A second benefit of downloading code into the kernel is that "the execution of downloaded code can be readily bounded" (Engler, Kaashoek, & O'Toole, 1995). They mean downloaded code can be executed at times when there are just a few microseconds of processing time available. This processing time-slice is too small to allow for a full context switch, so a traditional approach that requires a context switch would normally not be able to take advantage of such a small processing time-slice. Engler has stated that being able to process code during these small time-slices makes the Exokernel more powerful (Engler, 1998). The Exokernel is more powerful because it can optimize its processing of code by taking advantage of these small time-slices and increase the throughput of the applications. The freedom to optimize, as we stated previously, is a key part of being able to increase the processing speeds of operating systems.

Packet Sending and Receiving

Networking with an Exokernel is accomplished by what Ganger and others have called "application-level" networking (Ganger, Engler, Kaashoek, Briceño, Hunt, & Pinckney, 2002). The Exokernel's application-level networking allows an application to interact *almost* directly with the networking interface. Because the Exokernel provides much less of an abstraction for the application, the application-level code can provide more of its own management. This means the Exokernel is in a much better position to optimize its own operations, which can lead to an over-all higher performance level for the entire computer. The research documents are a little vague about the details of how networking is actually achieved, but it is clear that a first-in-first-out (FIFO) send queue is used. The documentation is also clear that the Exokernel also has a way of receiving packets and delivering them to the proper receiving application. Both of these processes are explained next.

The Exokernel sends a packet on a network through a system call referenced by "send_packet". When this function gets called, the packet gets added to a first-in-first-out queue and the kernel's involvement in the transmission ends at this point. The rest of the transmission gets handled by a network interface card or a device driver. The Exokernel gets involved minimally, but only to the extent that is needed to provide for a secure networking environment. The networking transmission management details are provided by the device driver code, which is located in user-space.

Packet receiving by the Exokernel is handled a little differently, but without much more involvement by the Exokernel. According to Ganger and others, the Exokernel receives packets by using two major processes; *packet demultiplexing* and *packet buffering*. Packet demultiplexing involves deciding which application a particular packet should be associated with. The information that the Exokernel uses to accomplish this is actually found within each packet which is located at a particular memory offset value. The process of actually delivering a packet to a particular application is called packet buffering by Ganger and others. Similar to the pre-arranged "save-area" mentioned in section 3.9 (Exceptions and Interrupts), the Exokernel copies the packet to a pre-registered memory area. Once the packet is successfully copied to the appropriate memory area, the Exokernel's involvement is complete. From this point, any further management or handling of the packet is accomplished by user-level code.

Naming and Routing of Packets

When a packet's high-level identifier gets translated into a low-level identifier, this is called "naming". Before a packet can be properly routed through a network, it must first be identified by a

name. Therefore, naming of packets is an important component to routing packets. Naming is also how the Address Resolution Protocol (ARP) is able to assign a unique identifier to each computer within a network. Without a unique name for each computer on a network, it would be impossible to properly address packets. The Exokernel supports the naming and routing of packets by what Ganger and others call the "sharing model" (Ganger, Engler, Kaashoek, Briceño, Hunt, & Pinckney, 2002). The Exokernel implements the sharing model by publishing all of the Address Resolution Protocol information in a *translation table*. Because this translation table is made available to user code in a read-only format, an application can look-up the information that it requires without asking the kernel for assistance. When a process does not find the information that it needs in the translation table, it can then ask the network for the information. This is called the sharing model because all of the applications share the translation table.

Network Error Reporting

It is important to notify the sender of a packet when the addressee of a packet cannot be located. According to Granger and others, the Exokernel's "stray packet" daemon takes care of TCP segment errors and network packets that cannot be delivered to the correct location (Ganger, Engler, Kaashoek, Briceño, Hunt, & Pinckney, 2002). Although the exact details of how the daemon handles these issues is unclear, it is interesting to note that the kernel is very much involved in this service. The researchers justify the kernel's heavy involvement by noting that this service is most closely related to protection rather than management. Therefore, the Exokernel can still be thought of as providing mostly protection of resources and allowing the actual management to be provided by code found in user-space.

PERFORMANCE RESULTS

Perhaps the best way to evaluate the Exokernel is by considering its performance results. In the following paragraphs, we will look at five areas that show enhanced operating system performance due to the Exokernel's approach. These results suggest the Exokernel's extensible approach provides enough of an enhancement as to make this approach a desirable alternative to the traditional operating system design.

Common Applications Benefit from an Exokernel

When comparing the results of benchmark tests that were performed by Xok/ExOS (a version of the Exokernel) with FreeBSD and OpenBSD (two other operating systems), we see that Xok/ExOS was able to complete 11 tests in just 41 seconds - which is about 19 seconds faster than the operating systems it was compared with. On three benchmark tests, Xok/ExOS did behave slightly slower than the competition, but these results were expected because of how the benchmark test was weighted. The slightly slower results were also of such a small degree as to not be really significant. As an overall score, the researchers state the Xok/ExOS is about 32% more efficient than the other operating systems that it was tested against.

Exokernel's Flexibility is not Costly

Benchmark tests were also used to see if the Exokernel's flexibility added too much overhead and made its execution less efficient. In the test, Xok/ExOS was compared to OpenBSD/C-FFS. The Xok/ExOS completed the test in 41 seconds versus the 51 seconds of the competition. Thus, Xok/ExOS was about 20% faster than the other operating systems. It is results like this that prompted Engler to state that "an Exokernel's flexibility can be provided for free" (Engler, 1998). Part of the reason why the Exokernel is a little more ef-

ficient is because the Exokernel is leaner - largely because protection mechanisms that usually get duplicated in a traditional operating system are not present.

Aggressive Applications are Significantly Times Faster

One of the goals of the extensible approach of the Exokernel is that its performance can be optimized. In order to test this theory, researchers attempted to make optimizations to applications running on an Exokernel system. As an experiment, XCP and CP were tested against each other. Although XCP and CP are both file copy programs, XCP is a file copy program that is optimized to take advantage of the flexibility of the Exokernel Operating System. The test results show that the XCP file copy program can complete its tasks about three times faster than that of CP. Other experiments were also conducted to test the speed of a Cheetah web server. The Cheetah web server, when running on top of the Exokernel OS (Xok), was found to be four times faster, for small documents. These results support the claim that the Exokernel OS does allow the user to optimize application code to achieve significant speed-ups in processing speeds.

Local Control can Lead to Enhanced Global Performance

The Exokernel researchers wondered if only specific processes can be optimized, or if the global performance of an operating system can be optimized. The researchers tested the Exokernel as it ran multiple applications concurrently and compared the results with non-extensible operating systems. The results show the Exokernel is, at least, as efficient as the non-extensible operating systems. Furthermore, after an Exokernel Operating System is optimized – what the researchers have called "local optimizations" – the "global performance" of the Exokernel is also enhance. Therefore, local optimizations do in fact support the global optimization of the Exokernel.

Exokernel's File Storage Scheme Enhances Run-Time

The researchers conducted experiments to test what Robert Grimm calls the Exokernel's "fine grained interleaving of disk storage" (Grimm, 1996). In the experiments, two applications were compared as they each accessed 1,000 10-KByte files. The Ekokernel's "fine grained interleaving" seems to account for a 45% faster file access time than that of an operating system that does not use this "fine grained interleaving" approach. The Exokernel's flexibility also seems to be responsible for allowing the Exokernel to conduct "file insert" operations about 6 times faster. These test results seem to support the conclusion that the Exokernel's file storage scheme does enhance the over-all run-time of the operating system.

ACTIVE NETWORKING

One of the technologies that has benefited from the Exokernel is Active Networking. The concept of an active network evolved from research being conducted at the Defense Advanced Research Projects Agency. This group is known for developing the "DARPA Internet", which is the foundation for our modern day Internet.

In a traditional network like the Internet, data is passively transported from a start point to an end point. Along its journey, the data passes through nodes that route the data packets based on header information while ignoring the actual data found in the packet contents. In the words of David L. Tennenhouse and others, the DARPA research community identified the following problems with networks;

1. "The difficulty of integrating new technologies and standards into the shared network

infrastructure."

2. "Poor performance due to redundant operations at several protocol layers."
3. "Difficulty accommodating new services in the existing architectural model."

(Tennenhouse, & Wetherall, 1996).

Tennenhouse and others state, in contrast to a passive network, an active network contains nodes that "can perform computations on, and modify, the packet contents." Furthermore, "this processing can be customized on a per user or per application basis" (Tennenhouse, & Wetherall, 1996). David Wetherall states well the benefit of active networks when he states that active networks "enable a range of new applications that leverage computation within the network; and it would accelerate the pace of innovation by decoupling services from the underlying infrastructure" (Wetherall, 1999).

A good example of an active network is provided by Parveen Patel. Patel states that active packets may encrypt themselves before entering an un-trusted portion of a network. The code to conduct the actual encryption could be carried by the active packets themselves, or the code could be resident on the node and simply be executed by the packets when they arrive. In either case, the data packets are *active* within the network, encrypting and decrypting themselves as necessary when passing through un-trusted sections of a network (Patel, 2002).

Hrishikesh Dandekar and others at NAI Labs (Network Associates, Inc. of Los Angeles, California) provide the link in our discussion that joins Active Networks and the Exokernel Operating System. Their research is named AMP. They state AMP is "a secure platform upon which the mobile code [of an active network] can be safely executed" (Dandekar, Purtell, & Schwab, 2002). The interesting part is that "AMP is layered on top of the MIT ExoPC (Exokernel) operating System's Xok kernel" (Dandekar, Purtell, & Schwab, 2002).

The Exokernel is a good foundation upon which to build AMP because the Exokernel offers AMP security, flexibility, and extensibility. Because the Exokernel's security mechanism "dovetails" nicely with the needs of AMP, Dandekar and others have stated AMPs "development time is reduced, modularity is enhanced, and security requirements can be addressed in a straightforward manner" (Dandekar, Purtell, & Schwab, 2002).

Flexibility with the Exokernel is reflected in its lack of abstractions. As Dandekar and others state, "an exokernel provides a minimal set of abstractions above the raw hardware. Only those mechanisms required in order to control access to physical resources and kernel abstractions are provided" (Dandekar, Purtell, & Schwab, 2002). As hardware gets abstracted by the operating system, the use of that hardware becomes less flexible. AMP's "NodeOS" provides a set of interfaces through which the code within an active network can request services of the underlying operating system. Abstracted by these interfaces are services such as networking channels, thread pools, memory pools, and domains. As Dandekar and others explain, "these abstractions provide the active application of platform-independent means for accessing a common set of resources which will be available across all of the heterogeneous network" (Dandekar, Purtell, & Schwab, 2002). It is precisely because the underlying Exokernel provides a minimal set of abstractions that the AMP NodeOS can utilize this functionality so easily.

Lastly, the Exokernel was designed to be extensible. The library operating system of the Exokernel matches up nicely with the libraries found in AMP (libAMP). An application like AMP can only be as extensible as the operating system that supports it. Thus, the superior extensibility of the Exokernel makes possible the superior flexibility of AMP.

In conclusion of our look at active networking, we see that the three "problems with networks" that were identified by DARPA are addressed by

active networks. Active networks provide a means to 1) easily integrate new technologies into the network infrastructure, 2) optimize performance, and 3) easily accommodate new services. Because the Exokernel makes our example of active networking possible, the Exokernel must share in the credit of giving rise to a solution to the problems with networks as identified by DARPA.

ANALYSIS AND DISCUSSION

In the following paragraphs, we present some of the criticisms of the Exokernel Operating System that have been offered by some of her detractors. We address the criticisms and offer some of our own. We offer further comments and reactions as a way to stimulate more discussion about the Exokernel Operating System.

Customer-Support

We begin our commentary with a quote from Jeff Mogul of Compaq Western Research Laboratory. Mogul says, "Extensibility has its problems. For example, it makes the customer-support issues a lot more complicated, because you no longer know which OS each of your customers is running" (Milojicic, 1999).

What Jeff Mogul seems to be pointing out is that each extensible operating system can be modified to the point of being unique. If each extensible operating system is user-modified and user-configured, then the challenge becomes how to efficiently provide customer support for a group of users if each user is essentially using a different operating system. For example, if both the *file management system* and the *communication manager* are uniquely modified, then trying to solve the issues that arise from their interaction could be very difficult.

Although providing customer support for an extensible operating system might present new challenges, it does not mean the extensible approach has to be eliminated. Previously (section 3.4), we explained that it is not necessary to customize an entire operating system. In fact, many users may not even need to customize any of the operating system. They may simply rely on the services provided by the user level library. So, as long as users are using the standard code found in the user level library, they are all using the same version of the operating system and therefore the customer service issue becomes a non-issue. It is only those users that decide to modify and optimize the libraries that pose the problem to customer service.

Presumably, users that are savvy enough to optimize their own code are probably savvy enough to trouble-shoot the issues that may arise from working with such a flexible operating system. In reality, it might not even be individual users that are optimizing their own code but rather software manufacturers that customize a user-level library so their product works faster on the extensible operating system. If this were the case, then perhaps there is no additional customer support issue for users of the standard user level libraries, and perhaps customer support issues that arise from optimized user-level libraries should be handled by the creators of the optimized code. So, if the customer support issue is even an issue at all, it becomes and issue for the optimizers of the user-level libraries and not an issue for the engineers of the extensible operating system. Thus, perhaps an extensible operating system should be brought to market and second-party software manufacturers should accept the responsibility of customer support if they choose to modify the standard user libraries.

On the other hand, as was stated previously "extensibility could actually help the customer support issue" (Leschke, 2004). In so far as extensible operating systems are easier to fix, then it should be easier to eliminate bugs and offer the community a more solid, error-free operating system. Since user level libraries can be replaced independently of the kernel, providing updated and

corrected libraries to the users of an extensible operating system should be straight forward. So, if extensibility means there might be fewer bugs or issues in the kernel, and also if extensibility means updated user level libraries might be easily added to an existing system, then perhaps an extensible operating system might lead to less of a customer support issue.

Furthermore, one might even argue that customer support will be easier to provide with an extensible operating system. Consider a customer support issue that arises from the use of a particular user level library. In so far as the issue is contained within that one user level library, then the customer support provider only needs to be an expert in that one library. This means customer support employees can be specialized. In so far as it is easier to train someone to be an expert in a limited number of user level libraries rather than an expert in the entire operating system, it seems that training customer support personnel will also be much easier with an extensible operating system.

Eliminating Management

One may argue that it is not necessary to eliminate *all* management from the operating system. Perhaps there are some operating system management functions that cannot be further optimized, and therefore, they should continue to be provided by the kernel. Or, perhaps the amount of optimization that is possible is so small as to not be worth the effort to move them outside the kernel. Perhaps the better approach is to allow the kernel to manage those processes that cannot be further optimized and to move into user-space that code that can be further optimized. A very logical question is - how will we know when code cannot be further optimized? Unfortunately, this is a question that cannot be answered without some further experience. On one hand, it seems that all processes will probably agree upon some common approaches to management, whereas on the other hand, if one wants maximum flexibility then one must move all management out of the kernel and make the code available in user-space.

The point we are trying to argue is that maybe eliminating *all* management from the kernel is too strong of a position. Maybe some management should remain in the kernel while other management code should be moved to user-space where it can be modified and optimized. Further research into this issue may reveal that maximum optimization can be achieved even if the operating system kernel retains some of the management responsibility.

As an illustration of our point, we cite Riechmann and Kleinöder as they state "As multithreaded applications become common, scheduling inside applications play a very important role for efficiency and fairness" (Riechmann, & Kleinöder, 1996). They further state the Exokernel's design leads to inefficiency because the Exokernel's thread scheduling algorithm requires an additional thread switch during execution. Their solution is to separate the management policy from the management implementation. Their research demonstrates that if one places the thread switching mechanism inside the kernel while allowing user-level code to handle the scheduling algorithm, some efficiency is gained over the Exokernel's approach. The idea of two level scheduling was also used by (Reuven & Wiseman, 2006) even though they suggest to implement both of the scheduling level within the kernel, but their implementation will be activated only if thrashing occurs (Wiseman, 2009), (Jiang, 2009).

Riechmann and Kleinöder argue our point for us. Our point is that perhaps a clean division between implementing protection within the kernel and implementing management within user code is too extreme. The research conducted by Riechmann and Kleinöder suggests that a pure Exokernel approach might not be the best answer. The Exokernel's approach needs to be embraced, but just not tightly. In conclusion, perhaps the best design for an operating system is one in which

there is a separation of mechanism from policy. Although the Exokernel has a policy of only allowing code in user-space to handle management of processes, perhaps on occasion the actual mechanism of management has to allow for some code to be executed within the kernel whenever it is more efficient to do so.

Optimizing Usage

When we talk about optimizing a computer, we are really talking about optimizing computer hardware rather than computer code. Admittedly, we do optimize computer code, but only as a way to optimize computer hardware. Thus, optimizing computer hardware is always the real goal. As such, who is the most qualified to optimize computer hardware? Software engineers? One might argue that hardware engineers – those that have an intimate understanding of the hardware components – are best suited for optimizing computer hardware. If the extensible approach becomes main-stream, we might see the line that separates computer engineering from software engineering becoming less distinct. Perhaps the engineers of the future will play two roles – one of hardware engineer and one of software engineer. The engineers of the future will surely need a strong understanding of computer code - since that is how we communicate with hardware - but they will also need an expert understanding of computer hardware, since that is what is actually being optimized. So, the extensible approach to operating system design might lead to a paradigm shift in how computers are optimized - but the final result will be a society of fully optimized computers.

The paradigm shift that will be caused by the extensible operating system will help put the *science* back into the Computer Science of the Information Technology Industry. Computer developers will be forced to make good Computer Science decisions from the ground up. Optimization will become the central focus of the computer industry. Optimization will eliminate the design approach seen in the current monolithic operating systems and give birth to a new breed of operating system that can keep pace with advancing computer technology. The entire computer science industry will experience a tremendous speed-up once the extensible operating system design becomes fully embraced.

Is Extensibility the Answer?

Druschel and others have argued against the Exokernel by saying "it is unclear to what extent the performance gains are due to extensibility, rather than merely resulting from optimizations that could equally be applied to an operating system that is not extensible" (Druschel, Pai, & Zwaenepoel, 1997). Through their research, Druschel and others have shown that traditional monolithic operating systems can be optimized just like the Exokernel. They claim the key to the speed-up is the optimization, not extensibility.

The Druschel research group tempers their argument against the Exokernel by saying "the real value in extensible kernels lies in their ability to stimulate research by allowing rapid experimentation using general extensions" (Druschel, Pai, & Zwaenepoel, 1997). They seem to be saying that extensible operating systems provide a means to quickly engineer prototypes of operating systems. This fast prototyping has caused a speed-up in the research, which had lead to a quicker way to discover techniques for optimizing operating systems. Although the Druschel group would say extensibility is not *the* answer, they do support the extensible approach because it is a tool that can be used to speed-up research and help bring a solution to market faster.

The argument provided by the Druschel group is well founded. However, perhaps they are overstating their position. Although we will agree that any code can be optimized - even monolithic operating systems - we still hold strong to the point that it is extensibility that really makes optimiza-

tion possible. In order to optimize an operating system, the system must first be flexible enough to be optimized. An operating system that is extensible is by its very nature open to changes and therefore easy to optimize. Although traditional operating systems can be optimized, they lack the flexibility required to make the changes easy. Because extensible operating systems are easy to change, they are perhaps the best design to work with when trying to optimize an operating system. Therefore, we still maintain that it is extensibility that is the foundation of being able to optimize operating systems.

CONCLUSION

As we bring our discussion to a close, we recall the two forces that are stretching the capabilities of the modern monolithic operating system. On one side there is the need for the operating system to be more flexible to accommodate new technologies. On the other side is the need for the operating system to become faster so it can keep pace with faster hardware and faster communication speeds. Our discussion showed how an extensible operating system like the Exokernel might fulfill both needs. Extensibility allows an operating system to be flexible enough to meet the changing demands of new technologies, while also making optimization easier, which translates into faster operating systems that can keep pace with faster computing environments.

In conclusion, there is a need for a faster and more flexible operating system, and the extensible approach of the Exokernel seems to meet this need. The speed and flexibility offered by the Exokernel will help operating systems avoid being the performance bottleneck in computer systems for years to come. Although extensible operating system technology is still in its infancy, the initial findings are encouraging to researchers. If contemporary operating systems are to keep pace with the forces that are being placed upon them, then modern operating system designers need to embrace the extensible approach found in the Exokernel.

REFERENCES

Chen, B. (2000). *Multiprocessing with the Exokernel Operating System*. Unpublished.

Dandekar, H., Purtell, A., & Schwab, S. (2002). AMP: Experiences with Building and Exokernel-based Platform for Active Networking. In *Proceedings: DARPA Active Networks Conference and Exposition*, (pp. 77-91).

Druschel, P., Pai, V., & Zwaenepoel, W. (1997). Extensible Kernels and Leading the OS Research Astray. In *Operating Systems*, (pp. 38-42).

Engler, D. R., Kaashoek, M. F., & O'Toole, J. (1995). Exokernel: an Operating System Architecture for Application-level Resource Management. In *15th ACM Symposium on Operating Systems Principles* (pp. 251-266).

Ganger, G., Engler, D., Kaashoek, M. F., Briceño, H., Hunt, R., & Pinckney, T. (2002). Fast and Flexible Application-level Networking on Exokernel Systems. *ACM Transactions on Computer Science*, *20*(1), 49–83. doi:10.1145/505452.505455

Grimm, R. (1996). *Exodisk: Maximizing Application Control Over Storage Management*. Unpublished.

Itshak, M., & Wiseman, Y. (2008). AMSQM: Adaptive Multiple SuperPage Queue Management. In *Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2008)*, Las Vegas, Nevada, (pp. 52-57).

Leschke, T. R. (2004). Achieving Speed and Flexibility by Separating Management From Protection: Embracing the Exokernel Operating System. *Operating Systems Review, 38*(4), 5–19. doi:10.1145/1031154.1031155

Milojicic, D. (1999). Operating Systems - Now and in the Future. *IEEE Concurrency, 7*(1), 12–21. doi:10.1109/MCC.1999.749132

Ousterhout, J. (1989). *Why Aren't Operating Systems Getting Faster as Fast as Hardware*. Unpublished. Carver, L., Chen, B., & Reyes, B. (1998). *Practice and Technique in Extensible Operating Systems*. Manuscript submitted for publication. Engler, D. R. (1998). *The Exokernel Operating System Architecture*. Unpublished.

Patel, P. (2002). An Introduction to Active Network Node Operating Systems. *Crossroads, 9*(2), 21–26. doi:10.1145/904067.904072

Reuven, M., & Wiseman, Y. (2006). Medium-Term Scheduler as a Solution for the Thrashing Effect. *The Computer Journal, 49*(3), 297–309. doi:10.1093/comjnl/bxl001

Riechmann, T., & Kleinöder, J. (1996). *User-Level Scheduling with Kernel Threads*. Unpublished.

Tennenhouse, D. L., Smith, J. M., Sincoskie, W. D., Wetherall, D. J., & Minden, G. J. (1997). A Survey of Active Network Research. *IEEE Communications Magazine, 35*(1), 80–86. doi:10.1109/35.568214

Tennenhouse, D. L., & Wetherall, D. J. (1996). Towards an Active Network Architecture. *Computer Communications Review, 26* (2).

Wetherall, D. (1999). Active Network Vision and Reality: Lessons From a Capsule-based System. *Operating Systems Review, 34*(5), 64–79. doi:10.1145/319344.319156

Wyatt, D. (1997). *Shared Libraries in an Exokernel Operating System*. Unpublished.