Achieving Speed and Flexibility by Separating Management from Protection: Embracing the Exokernel Operating System*

Tim Leschke
Department of Computer Science
Illinois Institute of Technology
10 W 31st Street
Chicago, Illinois 60616
lesctim@iit.edu

ABSTRACT

The modern operating system is currently caught in a tug-of-war between two forces. At one end, there is a force that is demanding that the operating system become more flexible to handle the needs of evolving hardware and evolving user applications. At the other end, there is a force that is demanding that the operating system become more efficient to meet the needs of faster hardware. If the modern operating system does not keep pace with these two forces, it could cause the progress in computer design to become stagnant.

One possible solution to this problem is the Exokernel Operating System – an extensible (or easily modifiable) operating system developed at the Massachusetts Institute of Technology. Extensibility allows the operating system to be flexible to change and also open to optimization.

Extensibility within an operating system has resulted in several new issues. For example, extensibility seems to make customer-support harder to provide. Furthermore, some multithreaded applications perform worse in an extensible environment. Lastly, some have argued that it is optimization, not extensibility, that should be credited for the enhanced operating system speeds.

In this paper, we discuss the Exokernel Operating System with some detail. We explore some of the issues that have kept the Exokernel design from being the main-stream approach. We propose solutions to these issues and we conclude by trying to motivate the reader to embrace the Exokernel approach.

1. INTRODUCTION

A traditional operating system (OS) is seen as both a resource allocator and a control program. As an allocator, an OS manages the resources and allocates them to programs that need to use them to accomplish their own tasks. Some of the resources that need to be allocated include CPU time, memory space, file-storage space, and I/O devices. As a control program, a modern OS controls the execution of user programs to help prevent errors and also to help the machine avoid improper use. A traditional OS is often seen as the one program that is always running at all times on the computer. This program is sometimes called the kernel.

In a traditional operating system, the hardware resources are hidden behind abstractions that are provided by the operating system. These abstractions are intended to help provide the user with an environment in which it is convenient for the user to execute programs as well as to make the programs run efficiently. In this way, a

* Funded by a Research Fellowship provided by the Illinois Institute of Technology.

traditional operating system gets involved in the execution of every user process.

An OS is generally seen as being a program that is embedded between the hardware and the user processes themselves. It is this implementation that requires the traditional OS to be all things to all processes. This traditional view is flawed because it leads to the development of operating systems that lower the efficiency of the system. This is rather ironic since, as we mentioned previously, one of the goals of an OS is to help the system run efficiently.

The Exokernel Operating System that was developed by a team at the Massachusetts Institute of Technology (MIT) has challenged this traditional view of an OS and it has proven that there is a more efficient way to implement a kernel. With the Exokernel Operating System, the kernel is only responsible for protecting and multiplexing the resources (hardware), and it relies on the applications to manage themselves. 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" [5]. This idea leads the team to create an operating system that is striped of the normal abstractions. The Exokernel is only responsible for providing protection of the hardware and it leaves the job of managing the user processes to an application level program.

In our investigation, we will investigate how the Exokernel Operating System, and exokernel systems in general, make separation of management from protection possible. In our investigation, we will look at the Exokernel Operating System in general by briefly explaining how select operating system functions are accomplished within an exokernel environment. In our investigation of this area, we will discuss topics that include the tracking of resource ownership, protecting and revoking resource usage, management of resources, interprocess communication, exceptions and interrupts, read and write operations to disk memory, and downloading code into the kernel. We will also investigate networking with an Exokernel by looking at packet sending and receiving, naming and routing of packets, and network error reporting.

As a reaction to the Exokernel, we will explore some of the shortcomings of the extensible OS approach and look at the holes in the Exokernel's feasibility for being a commercial operating system. We will discuss the customer support issue for an extensible system in a commercial setting. We will question the wisdom of eliminating *all* management from the kernel. We will question the optimization of code and who should be in charge of this process. We will show how some multithreaded applications do not benefit from the Exokenel approach. Lastly, we will question if extensibility is even the solution to the need for a faster and more flexible operating system. We will conclude that the exokernel approach can be attributed with many impressive speed-ups and we want to encourage further research in this area.

The next section, section 2, describes the major motivation for attempting the extensible approach. Section 3 provides an overview of seven key areas within the Exokernel's design. Section 4 is a review of the performance results as provided by the original researchers at MIT. Section 5 provides an analysis and discussion of the issues that surround the Exokernel. We conclude with some remarks as provided in section 6.

2. PROBLEM DESCRIPTION

The motivation for an extensible operating system comes from two sources – the need for flexibility in an operating system, and the need for a faster operating system. We will investigate these two motivations separately.

2.1 THE NEED FOR FLEXIBILITY

An operating system is often seen as that "thing" that sits between the hardware and the software. Defining what an operating system actually is

can become a point of contention. Some would say an operating system (OS) is that program that *manages* and *protects* the hardware within a computer system. An OS provides a standard interface that allows user applications a simple way to communicate with the hardware. It makes resources available and also protects how those resources are used.

Another view of an operating system says it is whatever comes in the box when it is purchased. In other words, an operating system is defined by the manufacturer. If, for example, a manufacture includes a text editor as part of its operating system, then this functionality becomes part of the definition of what an operating system is. Some have even gone so far as to argue that the user manual that accompanies the software must also be considered part of the operating system.

No matter how one defines an operating system, we must at least agree that in a very basic sense, an operating system is the software program that provides the interface to the computer hardware. This interface has its challenges. This interface needs to be able to interact with quickly changing hardware. Hardware is becoming faster, and consequently, more demanding of the operating system. For example, faster processors, larger processors (64 bit vs. 32 bit), larger and different memory devices (CD/DVD), faster communication links (busses), and higher network bandwidth are just some of the hardware improvements that have increased the demands placed on the operating system.

On the application side of the operating system, demand has also grown. Realistic gaming programs that demand different types of hardware access, portable palm pilots that require special features for downloading or uploading data, database systems that have particular memory management needs, garbage collection for object oriented programming environments, and real-time applications that require a guarantee on performance levels all put a strain on the current understanding of an operating system.

The traditional operating system has found itself in the middle of a tug-of-war between the ever changing needs of the hardware and the increasing demands of the user applications. If the modern operating system remains rigid in its design, it could be ripped-to-shreds by the opposing forces. The modern OS needs to become more flexible to accommodate change, as well as more modifiable and customizable to handle the specific demands that are asked of it.

2.2 THE NEED FOR SPEED

One of the fundamental laws of computer architecture is attributed to Gene Amdahl. "Amdahl's Law," as it is known, tells us that the *speed-up* to be gained by using an improved mode of execution is limited by the amount of time that the improved mode is actually used. By example, this means that a 100% speed-up in an improved mode that is only used 10% of the time, will only cause about a 5% improvement (or speed-up) in the total execution of the system. What this means for the modern computer designer is that he must stay aware of trends in changing technology so he can identify and improve those areas that will likely impede the impact of the improved technology.

One of the areas of computer design that has enjoyed some great improvement in the recent past is the area of Central Processing Unit (CPU) speed. Although CPU speed, as well as other hardware speeds, may be increasing, it's full benefit will not be realized if the rest of the computer system cannot keep pace with its advances. One such area that may impede the benefits associated with faster hardware is the area of 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" [5]. Furthermore, as an example, "Operating systems derived from UNIX use caches to speed up reads, but they require synchronous disk I/O for operations that modify

files. If this coupling isn't eliminated, a large class of file-intensive programs will receive little or no benefit from faster hardware" [9] If the speed of computer systems is going to avoid becoming stagnant, we must enter a new age of operating system design. This new age will require that operating systems be more efficient to match the improvements made with faster hardware.

2.3 THE APPROACH

Some would say, an operating system is a necessary evil [1]. In so far as it is necessary, it cannot be eliminated form the equation. The modern demands that an operating system become faster and more flexible have forced researchers to consider radical changes in how operating systems are designed. One such radical design is called an extensible operating system.

To be extensible means to be flexible to change. An operating system that is extensible can be easily modified to accommodate the changing needs of the underlying hardware. An extensible system is also one that can be easily modified to accommodate the growing needs of user applications. Furthermore, this extensibility also allows the operating system to be optimized to provide faster service in an environment that is always demanding more speed. Extensibility appears to be the magic solution to the growing speed and flexibility issue. By promising both speed and flexibility, the extensible operating system seems to be the approach that will, at least temporarily, prevent computer speeds from becoming stagnant while still meeting the changing needs of the user applications.

One such extensible approach is provided by the Exokernel Operating System at the Massachusetts Institute of Technology. The Exokernel approaches the extensibility issue by separating management from protection. The Exokernel provides only multiplexing of resources while leaving management of resources to user level processes. The initial idea for this approach is that user processes themselves know better how to manage the use of those resources. The result of this approach is an operating system that is both flexible and efficient. This approach is flexible because user level code is easily and quickly modifiable. It is efficient because management processes can now be optimized to provide the quickest response. It is this operating system, the Exokernel, that we will be taking a closer look at.

3. THE EXOKERNEL SOLUTION

As we said previously, an exokernel is only responsible for protecting and multiplexing the resources. This resource protection consists of three major tasks according to Dawson Engler; 1) tracking ownership of resources, 2) ensuring protection by guarding all resource usage or binding points, and 3) revoking access to resources [4]. Some of the lesser tasks that an exokernel is responsible for include protecting physical memory, the CPU, network devices, writes to "special" memory locations that control devices, and the ability of a process to execute privileged instructions. What is important to understand here is that the exokernel only gets involved in these activities to the extent that it is providing protection. The exokernel does not get involved in the *details* of these activities. By this we mean that, for example, an exokernel gets involved with granting access to a resource and revoking access to a resource, but it does not manage how that resource should be used. Because of this increased freedom, a user process is allowed to use a resource improperly. Therefore, more responsibility falls on the shoulders of the programmers to ensure that their programs use the shared resources properly.

A more solid example of how an exokernel gets involved in protecting resources and not involved in management is seen in the example provide by Engler where he explains the protection of physical memory. In a traditional OS, reading and writing to memory are privileged instructions. Thus, every read/write request to disk memory has to be verified for access rights by the kernel. At this point, there is a kind of message relaying that the kernel gets

involved in. For example, a process that wants to access a memory location for writing will ask the kernel through what is sometimes called a system call. The kernel gets this system call and it then contacts the hardware on the processes behalf. In this way, a process is not allowed to execute privileged instructions directly. By forcing every read/write to go though the kernel, global efficiency is greatly reduced.

According to Engler, the Exokernel's answer to this situation is "to make traditionally privileged code unprivileged by limiting the duties of the kernel to just these required for protection" [4]. What this means for the user is that now a user process can access memory directly. A kernel will still verify that the memory access is safe, but any message relaying will be directly between the process and the hardware itself.

In the following pages, we investigate seven key parts of the Exokernel Operating System. This is not intended to be a comprehensive list, but rather a selection that will provide a good general understanding of the exokernel approach.

3.1 RESOURCE PROTECTION

TRACKING OWNERSHIP OF RESOURCES

When a resource gets allocated, it actually gets allocated by what is called the Library Operating System (LibOS) and not by the exokernel itself. When a resource gets allocated, the exokernel records who the owner is and any other information associated with that ownership. For example, if the resource was a physical memory page, the exokernel would record the process that owns it and the permissions like 'read' and 'write' that are associated with that allocation [4]. The exokernel records this information according to what can be called an open book-keeping policy. As Engler says, the book-keeping page table is made available as read-only to all the processes. Since the available resources are now visible to all of the processes, a process can look-up which resources should be available before it even submits a request for that resource. If the resource is not available, the request is denied.

ENSURING PROTECTION BY GUARDING ALL RESOURCE USAGE OR BINDING POINTS

The ability of a process to use a resource and not have that resource taken away by accident is very important. For example, if a process is granted a block of memory, it should be allowed to safely use that memory block without fear of having it taken away by another process. Furthermore, the owner should have the ability to de-allocate the resource when it wants. An exokernel allows a process to bind to a resource through what is called a "secure binding" [5].

A secure binding is simply the separation of authorization from the actual use of a resource. When a resource gets allocated, authorization is only allowed at the time of the binding. The process retains the authorization until it relinquishes it [5]. Since the exokernel only gets involved in the authorization of a binding at bind time, the exokernel does not have to get involved in the ongoing management of that secure binding.

Furthermore, an exokernel does not have to understand what it is binding with a secure binding. Application software can involve many complex semantics when it is binding to a resource. An exokernel only gets involved in the secure binding and it does not care about the details or semantics of that binding. As Engler, Kaashoek, and O'Toole say, "a secure binding allows the kernel to protect resources without understanding them." [5].

REVOKING ACCESS TO RESOURCES

Once a process has obtained a secure binding to a resource, there must be a way for the OS to revoke this binding. In a traditional OS, the binding is broken through what is known as *invisible revocation*. An invisible revocation is one in which the application is not informed of the de-allocation. A disadvantage of this

approach is, as Engler, Kaashoek, and O'Toole say, "library operating systems cannot guide de-allocation and have no knowledge that resources are scarce" [5].

An exokernel, on the other hand, revokes a secure binding for most resources by what has been called a *visible revocation*. A visible revocation allows communication between the kernel and the process. By communicating with the process, the kernel can inform it of its need to break the secure binding. The process can then prepare for this by saving any data that it needs to. For example, if a process was going to lose a certain page of physical memory, it could update its pointers to reflect this change. Furthermore, sometimes a kernel just needs a page of memory and it does not care which one. By communicating with the process, the process is allowed to choose which of its memory pages to give up. The process can then write this page to disk and free the memory page for the kernel. This is how revoking a resource can be accomplished if the process cooperates with the kernel.

Sometimes processes do not cooperate with the kernel, and when this happens, the kernel has to use more force. When the exokernel has to break a secure binding by force, it "simply breaks all existing secure bindings to the resource and informs the Library OS" [5].

3.2 MANAGEMENT BY USER LEVEL CODE

EXTENSIBLE MANAGEMENT LIBRARIES

Previously we said that the exokernel is only responsible for protection and it leaves the management duties up to another entity. We would be remiss in our discussion if we did not explain how management actually works. For an exokernel operating system, management of resources and processes is provided by an outside entity that is known as a "user level library operating system," or LibOS for short.

The LibOS provides the typical abstraction that lies between an application and the hardware. Since each application has its own LibOS, its LibOS can be custom-made to suit the needs of the application. This customized use of resources is one of the things that contributes to the efficiency of the exokernel system. Furthermore, a LibOS can be simple and more specialized, as Engler, Kaashoek, and O'Toole say, "because library operating systems need not multiplex a resource among competing applications with widely different demands" [5]. LibOS's allow for portability of applications if the LibOS's use standardized interfaces. Furthermore, having a specialized library operating system for each application would seem to add a lot of redundant user code to user space. The Exokernel OS overcomes this problem by what has been called "shared libraries."

SHARED LIBRARIES

The shared library of the LibOS allows multiple applications to share common collections of code. This sharing helps reduce the amount of redundant disk and memory usage. With a typical application, a small section of code is run at startup to load the necessary libraries into memory. This process can lead to what Douglas Wyatt calls a "bootstrapping problem" [11], because loading the LibOS from disk into memory is dependent upon the code that is in the LibOS itself. We will investigate this further, after we finish our shared library investigation.

One of the key issues that a shared library system must address is what Wyatt calls "symbol resolution" [11]. As Wyatt explains, there is a need for each program to have a reference to a particular memory location executed correctly at run-time. The solution to this problem is to always have a shared library loaded into the same location within the virtual address space.

A better solution to the "symbol resolution" issue is what Wyatt calls an "indirection table" [11]. With an indirection table, the actual address of a symbol is stored in one central table.

Each shared library is provided with a copy of this table, and this copy is what allows a shared library to look-up the relative offset of the symbol and jump to that location. Furthermore, since these offset locations are known when the shared libraries are compiled, these addresses can be incorporated and hard-coded into the shared library itself. Careful use of an indirection table in this manner allows a shared library to be loaded into any available address apace.

IMPLEMENTING A SHARED LIBRARY

When a program loads a shared library for the first time, it first checks to see if the library is already in the indirection table. If the library is not in the indirection table, it loads the library and it updates the indirection table to record where the new library is. If a second program tries to load the same library, it first notices that the indirection table already has an address for that particular library. Rather than loading a second copy of the same library, the program simply updates its page table to point to the existing library.

There is a cost associated with using a shared library. The cost is having to look up addresses in the indirection table. However, by sharing libraries, memory consumption is reduced which leads to less frequent page faults. Having less frequent page faults can allow applications to run faster. Another benefit of using a shared library is that the library can be updated and improved. As long as no interfaces are changed or new functions added, the new library can be compiled and used to replace the old one without changing or recompiling the applications that rely on it.

THE SHARED LIBRARY SERVER

Earlier we mentioned a "bootstrapping problem" that was the result of trying to load a LibOS into memory when the code needed to help us do this is actually in the LibOS itself. This problem is solved by what is called the "shared library server" or SLS for short.

When an exokernel is booted, the shared library server is also initiated. Once initiated, the SLS communicates with applications that want to load a shared library. As part of the process of helping load a shared library, an SLS provides a means to be able to open files, read files, write files, map files from disk, open directories, read directories, and perform basic console input and output. The basic workings of the shared library server provide just enough functionality to allow a shared library to overcome the bootstrapping problem and load itself into memory.

3.3 INTERPROCESS COMMUNICATION (IPC)

Benjie Chen says that one is able to implement fast IPC through what he calls "protected control transfer" [2]. This communication method allows secure registers to be used to pass data. Furthermore, this communication is immediate; once it is initiated, the message is passed immediately between processes and the registers without any need for kernel assistance. In this way, the exokernel only provides the secure registers and does not actively manage the communication. This speed-up in communication is just one of the many advantages of using an extensible operating system like Exokernel.

3.4 EXCEPTIONS AND INTERRUPTS

When a traditional operating system experiences an exception or an interrupt, it has to protect the current state of the system by saving some register data to a more secure memory location. The OS then has to decode the exception or interrupt and execute the appropriate instructions to handle the problem. Once the error has been dealt with, the OS then starts running at a program counter location that is saved prior to handling the error.

With the exokernel, the kernel gets less involved in the error handling. For example, all hardware exceptions are handled by the applications themselves. An Exokernel's limited role involves saving some registers in what Engler

calls an agreed upon "save area" [4]. This "save area" is a user-accessible memory location. Once the registers are saved, the exokernel loads the exception and jumps to an address specified by an application. This new address is application code written specifically to handle the particular exception. Presumably, this application level code is found in what we are calling the "User Level Library." Once again we see a situation where the kernel absolves itself of any management responsibility and leaves this up to the Library OS to handle.

Once the Library OS has taken over the handling of the exception, the Exokernel's job is done. When the Library OS is done handling the exception, since the register states were saved in user-accessible memory, the LibOS can simply access this information and continue with its normal execution without any further assistance from the exokernel. In this way, the exokernel does not get involved in the details of error handling. As we have seen, the LibOS handles the details of error handling, and the exokernel simply helps provide protection for the current state of the registers.

3.5 DISK I/O

Disk I/O that provides read and write to memory operations is achieved though an asynchronous method. When a calling application makes a read or write request, the Exokernel's "exodisk" system hands this request off to the disk driver itself. Control is immediately returned to the calling application. The Exokernel does not wait for the request to complete, instead it gives control to the application, and the application can choose to wait for a read or write request to complete or not. In this way, the Exokernel is again absolving itself of any management responsibilities.

The Exokernel briefly gets involved in disk I/O one more time when the read or write process completes. When a process completes, the Exokernel needs to notify the requesting application of this event. However, other than helping an application start executing a good read or write operation and then helping with some limited message passing, the Exokernel OS continues its roll as only a protection provider and it leaves management up to application/LibOS themselves.

3.6 DOWNLOADING CODE INTO THE KERNEL

One of the ways an exokernel system improves performance is through a technique of downloading code into the kernel itself. This technique is not unique to an exokernel system. Downloading code into the kernel is a technique that a variety of other operating systems have also taken advantage of.

There are two main advantages to downloading code into the kernel according to Engler Kaashoek and O'Toole. First, downloading code into the kernel eliminates the need to make "kernel crossings," which require context switches [5]. Kernel crossings slow down the execution speed of applications. By eliminating kernel crossings, applications benefit from having an enhanced speed.

The second main benefit of downloading code into the kernel is, as they say again, "the execution of downloaded code can be readily bounded" [5]. What they mean by this is that downloaded code can be executed at times when doing a context switch would prove to be too costly, like when there is only a few microseconds of processing time available for use. Engler says this second benefit is the most valuable of the two because it makes the code more powerful [4]. Downloading code is more powerful because it can be granted more freedom of ability than normal application code. Freedom, after all, is the main point of using an exokernel system in the first place.

3.7 NETWORKING

PACKET SENDING AND RECEIVING

The Exokernel makes use of what Ganger and others call "application-level" networking [6].

With this approach, user applications are allowed to interact *almost* directly with a network interface. This allows applications to manage their communication patterns better which can lead to higher performance. In true Exokernel style, the operating system only gets involved in this event in so far as it can provide protection of resources, and it leaves the actual management up to the user processes. The details of how the kernel gets involved in networking is a little vague from the research documents, but it is clear that the kernel provides a FIFO send queue. It is also clear that the kernel has a method for receiving packets from the wire and copies them to the proper receiving application. Both of these activities are explained next.

When it comes to sending a packet on a network, the Exokernel provides a system call referenced by "send_packet." This function adds the outbound packet to a first-in-first-out queue which is serviced by a network interface card or a device driver that handles the transmission from that point. That's it. That all the Exokernel does for sending packets. The Exokernel gets involved minimally, to the extent that is has to get involved, but not anymore than that. This minimal involvement in sending packets allows the Exokernel to provide a secure way to send packets, without managing the details of the network transmission.

When it comes to receiving packets, the Exokernel gets a little more involved, but not much more. When receiving a packet, there are two major parts which Ganger and others call *packet demultiplexing* and *packet buffering*. Packet demultiplexing is the process of deciding which application a particular packet should be associated with. This is accomplished by checking various data offset values in the packet. Packet buffering is the process of delivering packets to the appropriate application. The Exokernel does this by copying the packet into a pre-registered memory region. At this point, the role of the Exokernel is complete. Any further packet handling is done by user-level code.

NAMING AND ROUTING OF PACKETS

Naming is the process by which a high-level identifier is translated to a low-level identifier. Routing depends on naming to properly map a correct route through a network. Naming is of central importance to networking because it is how the Address Resolution Protocol (ARP) uniquely identifies a computer within a network. Uniquely identifying a computer is essential to properly addressing a packet. An exokernel supports this by what the Ganger group refers to as the "sharing model" [6]. With the sharing model, all of the ARP information is provided in a *translation table*. If an application needs some ARP information, it can access this read-only table and look for the information that it needs. If the information is not in the ARP table, the process can request the information from the network. In this way, all application processes share this translation table. Once again, in classic Exokernel style, we see that the operating system is providing the bare minimum of this service. It has exposed the ARP information to the processes while allowing the processes themselves to manage its use.

NETWORK ERROR REPORTING

The Exokernel uses what is called a "stray packet" daemon to handle unclaimed network packets. Occasionally, a recipient of a packet cannot be located. In such cases, it is important to convey this information to the sender of the packet. According to the Ganger group, this problem and other TCP segment errors are handled by this stray packet daemon [6]. The exact details of how the stray packet daemon handles this is never made clear by the Ganger group. What is clear is that this service is provided by the operating system because it is a protection-related service (not a management service).

4. PERFORMANCE RESULTS

When looking at the performance of the Exokernel system, we see that there are good results that support the argument for

implementing this new extensible method [4, 7]. As we look at some of the results, we will see how the experimental results show enhancements in five major areas of operating system performance.

4.1 BENCHMARK APPLICATIONS RUN SIGNIFICANTLY FASTER

Xok/ExOS (an example of an exokernel operating system) was able to complete 11 benchmark tests in 41 seconds [4]. This result is 19 seconds faster than the other two operating systems that it was compared to (FreeBSD and OpenBSD). There are three benchmark tests where Xok/ExOS was just slightly slower than the other operating systems, but this difference was very small and it was also expected because of how the benchmark test was weighted [4]. Overall, we see that Xok/ExOS is about 32% more efficient than the operating system that it was tested against.

4.2 EXOKERNEL'S FLEXIBILITY IS NOT COSTLY

Another benchmark test has been completed to test if the flexibility that exokernel offers adds any extra cost to application execution [4]. In the test, Xok/ExOS completed the tests in a total running time of 41 seconds. This value should be compared to the results of OpenBSD/C-FFS which completed all of its tests in 51 seconds, or about 20% slower than exokernel. From this result, we see that "an exokernel's flexibility can be provided for free" [4]. Furthermore, this efficiency is gained because some protection mechanisms that are duplicated in a traditional operating system are only implemented once by the leaner exokernel system.

4.3 AGGRESSIVE APPLICATIONS ARE SIGNIFICANTLY TIMES FASTER

When investigating this area, researchers were primarily interested in knowing if meaningful optimizations could be performed with applications running on an exokernel system. To investigate this, researchers compared two file copy programs, XCP and CP. XCP is a file copy program that is optimized to take advantage of the exokernel system. Results of tests show that XCP runs three times faster than CP [4].

In another experiment, web servers were tested that take advantage of the exokernel system. When a Cheetah web server was implemented on top of an exokernel system (Xok), there was a performance improvement of four times (for small documents) [4].

Both of these significant speed-ups demonstrate that an aggressively optimized application can in fact run at a significantly faster rate.

4.4 LOCAL CONTROL CAN LEAD TO ENHANCED GLOBAL PERFORMANCE

There is still much work that can be done in this area, but some test results suggest that an exokernel system can provide improvements in global performance. For example, tests were performed to evaluate how an exokernel system performed while running multiple applications concurrently [4]. The results show that an exokernel is able to perform comparably to widely used non-extensible systems. Furthermore, if an exokernel system is allowed to make local optimizations, global performance is allowed to increase. Therefore, local control can actually lead to enhanced global performance.

4.5 EXOKERNEL'S FILE STORAGE SCHEME ENHANCES RUN-TIME

Robert Grimm points out that experiments were done to test the effects of the Exokernel's "fine-grained interleaving of disk storage" [7]. In the experiments, two applications that accessed 1,000 10 KByte files each were compared. The results show that the Exokernel's implementation is able to access the files up to 45% faster than a system that does not use "fine-grained interleaving." Furthermore, a file *insert* operation can be performed 6 times faster as a result of the Exokernel's flexibility. Thus, the

Exokernel's file storage scheme improves the global efficiency of the entire system.

5. ANALYSIS AND DISCUSSION

As a way to provide a meaningful analysis of the Exokernel operating system, we discuss the criticisms that have been made by some of the Exokernel's detractors. We also offer our own reactions, comments, and criticisms as a way to foster more discussion of the Exokernel approach.

5.1 CUSTOMER-SUPPORT

As a launching pad for our commentary, let us start with the insight of 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" [8]. Presumably this is an issue because, with an extensible OS, each OS can be modified to a point that it is unique. If every user has a truly unique, self-modified and self-configured, operating system, then it does seem to be a challenge to provide support for this new product. For example, if an extensible OS has a uniquely modified *file management system* interacting with a uniquely modified *communication manager*, then it could be difficult to trouble-shoot issues that rise from their interaction.

However, just because customer support will be challenging for an extensible OS, it does not mean customer support has to be eliminated. As we saw in section 3.2 (*Management by User Level Code*), the entire operating system does not necessarily have to be customized. User applications may require a standardized library. In so far as users stick to using the standardized libraries, customer support becomes more manageable. Extensibility suggests there should be a choice among the standard libraries. Having choices might complicate the customer support issue, but this complication should not prohibit one from taking seriously the question of using an extensible operating system.

On the other hand, extensibility could actually help the customer support issue. If we assume that some customer support issues arise from bugs or errors in the original code of the operating system, the fact that this operating system is extensible should then make the system easier to fix. Being that the system is extensible, it positions itself to be easily fixed or modified. Of course, if there is an error in the kernel itself, perhaps a corrected version will need to be released. In any case, because extensibility can lead to faster corrections of the code, extensibility may actually lead to a lower customer support demand rather than an increase in customer support demand.

Extensibility could also make customer service easier to provide. For example, if a customer support issue arises from a particular LibOS, then the customer needs to only talk to a customer service representative that handles that particular LibOS. Because customer service providers can specialize in a sub-set of standardized LibOSs, they are no longer required to know and understand the entire operating system. This could greatly reduce the amount of time it takes to train new customer service representatives.

5.2 ELIMINATING MANAGEMENT

One may wonder if we need to eliminate *all* management from the operating system. It seems that there could be some existing management techniques that do not need to be optimized (because they are already fully optimized or perhaps there is not enough to gain by trying to optimize them). For example, as we saw with the Library Operating System, an open bookkeeping approach is used to display a resources availability through a page table. This helps the applications avoid requesting the information from the kernel. One must agree that avoiding a call to the kernel is good, but at what cost? If the kernel is not going to provide this service, then every application now has to

have a procedure to read the page table. Does this mean that every application will be duplicating this procedure and also duplicating the code too? This seems wasteful. One might suspect that the Exokernel group would reply that duplicated code can be avoided by using a *shared* library.

Perhaps we should consider providing management for those items that cannot be further optimized, and allow applications to manage those areas that can be optimized. The logically next question is, "how will we know when something cannot be further optimized?" Unfortunately, this cannot be known *a priori*. On one hand, it seems that all applications should be able to agree upon some common management strategies, while on the other hand, for maximum flexibility, we need to leave all of the options open and provide no fixed management strategies.

The main point is, perhaps the Exokernel is taking too strong of a position by eliminating *all* management. Maybe there is a happy medium between providing full management and providing no management. Further research may reveal that the Exokernel does not need to eliminate all management to achieve maximum optimization.

5.3 OPTIMIZING USAGE

Who knows better how to optimize hardware usage? Some would argue, computer engineers – those that design and build the actual hardware components - probably are the most qualified to optimize a particular piece of hardware. This suggests that hardware engineers should also be the ones to write the OS library code that optimizes a particular piece of hardware. This approach will tend to remove the line that currently seems to separate software engineering from hardware engineering. This means engineers will have to play two roles – one of hardware developer and another role of software developer. This will require a paradigm shift in the way computers are developed, but we will gain a society of better optimized computer resources.

This paradigm shift will enhance the Information Technology Industry. It will put the *science* back into the Computer Science that drives the industry. It will force developers to consider good computer science solutions. It will make optimization one of the key driving forces. Modern monolithic operating systems do not make optimization as important as this new extensible approach will make it. By making optimization a focal point of Information Technology, the entire industry will experience a tremendous speed-up.

5.4 MULTITHREADED APPLICATIONS

According to Riechmann and Kleinöder, "As multithreaded applications become common, scheduling inside applications plays a very important role for efficiency and fairness." [10]. They go on to explain that the Exokernel approach leads to an inefficient solution. The Exokernel requires an additional thread switch during the scheduling algorithm. Their conclusion is that a purely user-level thread switching and scheduling algorithm leads to an inefficiency. Their solution is to separate mechanism from policy. In their research, they placed the switching mechanism in the kernel while allowing the scheduling algorithm to be located at user level. This approach has resulted in some modest improvements over the Exokernel's approach.

Riechmann and Kleinöder support one of our points. While the traditional monolithic operating system provides management and protection of resources, the Exokernel strives to provide only protection. Perhaps the most efficient solution is somewhere between these two approaches. Perhaps modern computer engineers should explore the approach of the Exokernel without embracing it so tightly. The Exokernel has provided some impressive improvements in efficiency, and these improvements need to be explored. However, the radical approach of a *pure* exokernel might

not be the answer, but it could be a good place to start the investigation.

5.5 IS EXTENSIBILITY THE ANSWER?

An argument against the Exokernel is "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. [3]. Druschel and others have demonstrated through their research that the same gains that have been accomplished by optimizing an extensible operating system are also possible by optimizing current monolithic operating systems. The key is *optimization*, not extensibility.

Extensibility does have its place however. The Druschel group says that "the real value of extensible kernels lies in their ability to stimulate research by allowing rapid experimentation using general extensions" [3]. In other words, extensible systems allow for fast prototyping of experimental operating systems. More experimentation needs to be done to optimize current approaches to operating systems, and one way to improve the speed at which this experimentation takes place is to utilize the fast prototyping that is made possible by extensible systems. Thus, extensibility is not *the* answer (according to the Druschel group), but it is a tool that can be used in the development of a better solution.

The Drushel group makes a very interesting point. It does seem that it is optimization rather than extensibility that gives rise to the speed-up. However, one might argue that the Drushel group over states their position. One may argue it is extensibility that makes optimization possible. Therefore, extensibility cannot be over looked. What we need is an operating system that can be optimized for a multitude of user applications. Extensibility makes this possible. It is precisely because the operating system is easily modifiable that it can also be optimized to please all user applications. In conclusion, the Drushel group is correct in that it is optimization that causes the speed-up, but it is extensibility that makes the optimization possible. Therefore, extensibility is the key component for enhancing operating system speeds.

6. CONCLUSION

As we bring our investigation to a close, we recall that we began our investigation by looking at the two forces that are pulling at the modern monolithic operating system. There is a tremendous demand for an operating system to be more flexible to meet the changing needs of ever changing hardware and ever changing user applications. There is also a new demand that operating systems become faster – a problem that is solved through optimization. The magic approach that seems to satisfy these growing needs is *extensibility*.

As we looked at extensible operating systems, we focused on the implementation details of the Exokernel Operating System that was developed at the Massachusetts Institute of Technology. As we studied the Exokernel approach, we saw that this system provided protection of resources while leaving the management of those resources up to user-level applications. By separating management from protection, we saw how the Exokernel was able to become easily optimized. This optimization has lead to several examples of how the Exokernel is able to demonstrate significant speed-ups when compared to standard benchmarks. This solid performance improvement should encourage more research in the area of extensibility.

Lastly, we discussed some of the downsides associated with creating an extensible operating system. We mentioned the customer service issue, the need to eliminate *all* management, a new approach to optimizing hardware and software, and whether or not extensibility is even the solution to this problem. We admit that there are some issues with the Exokernel project. There are some problems that arise from such a radical new approach to operating system design. However, we conclude that these initial issues are minimal at best, and they should not

discourage further exploration into the extensibility question.

In conclusion, extensibility does seem to be the solution that will solve the flexibility issue and the speed issue, all at one time. This technology is still in its infancy, but the initial findings suggest that further studies should be encourage. Thus, if the modern operating system is to avoid being stagnant, operating system designers need to embrace the extensibility approach.

7. REFERENCES

[1] "Practice and Technique in Extensible Operating Systems" by Lee Carver, Ying-Hung Chen, and Theodore Reyes. University of California – San Diego, 1998.

[2] "Multiprocessing with the Exokernel Operating System" by Benjie Chen. Massachusetts Institute of Technology, February 2000.

[3] "Extensible Kernels are Leading the OS Research Astray" by Peter Druschel, Vivek Pai, and Willy Zwaenepoel. Rice University, 1997.

[4] "The Exokernel Operating System Architecture" by Dawson R. Engler. Ph.D. thesis, Massachusetts Institute of Technology, October 1998.

[5] "Exokernel: an Operating System Architecture for Application-level Resource Management." by Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. In the Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95), Copper Mountain Resort, Colorado, December 1995, pages 251-266.

[6] "Fast and Flexible Application-Level Networking on Exokernel Systems" by Gregory Ganger, Dawson Engler, M. Frans Kaashoek, Héctor Briceño, Russell Hunt, and Thomas Pinckney. In ACM Transactions on Computer Science Volume 20, Issue 1, February 2002.

[7] "Exodisk: Maximizing application control over storage management" by Robert Grimm. Massachusetts Institute of Technology, May 1996.

[8] "Operating Systems – Now and in the Future" by Dejan Milojicic. IEEE Concurrency, January-March 1999.

[9] "Why Aren't Operating Systems Getting Faster As Fast As Hardware?" by John Ousterhout. Digital Equipment Corporation, October 1989.

[10] "User-Level Scheduling with Kernel Threads" by Thomas Riechmann and Jürgen Kleinöder. University of Erlangen-Nürnburg, June 1996,

[11] "Shared Libraries in an Exokernel Operating System" by Douglas Wyatt. MS Thesis, Massachusetts Institute of Technology; August 1997.