KLOS: A High Performance Kernel-Less Operating System

Amit Vasudevan
Computer Science & Engg.
University of Texas at Arlington
vasudeva@cse.uta.edu

Ramesh Yerraballi
Computer Science & Engg.
University of Texas at Arlington
ramesh@cse.uta.edu

Ashish Chawla
Computer Science & Engg.
University of Texas at Arlington
chawla@cse.uta.edu

ABSTRACT

Operating Systems provide services that are accessed by processes via mechanisms that involve a ring transition to transfer control to the kernel where the required function is performed. This has one significant drawback that every service call involves an overhead of a context switch where processor state is saved and a protection domain transfer is performed. However, as we discovered, it is possible, on architectures that support segmentation, to achieve a significant performance gain in accessing the services provided by the operating system by not performing a ring transition. Further, such gains can be achieved without compromising on the separation of the privileged components from the non-privileged. Our service call mechanism results in 28 clock cycles in the best case and 50 clock cycles in the average case which is an order of magnitude faster than current widely implemented methods of service or system calls

1. INTRODUCTION

Most, if not all production level operating systems have a dual mode of operation – (1) the privileged level where the kernel resides and, (2) the unprivileged level where application and system processes execute. A ring transition mechanism is used to move from one level to the other. The idea behind this separation has always been protection and stability. However, on architectures that support segmentation, it is possible to achieve a significant performance gain by eliminating the ring transition. Further, such gains can be achieved without compromising protection. This is made possible by the use of a subtle trick involving segmentation and Task State Segments (TSS).

We propose an operating system design in which – (1) there is no protected kernel as perceived in current operating systems, (2) Operating system services are accessed without a ring transition, (3) all processes and the operating system execute at the unprivileged level and, (4) each process has its own private address space and virtual memory mapping and functions independent of all other processes in the system. KLOS, a Kernel-Less Operating System is a realization of this design.

We primarily target this design towards personal computers where performance takes precedence over stability. A point to be reinforced is that, regular runaway processes do not pose a threat to the stability of the operating system built on this design. Processes that are specifically engineered to thwart the stability of the system are contained with a very high probability of success.

KLOS is very much a work in progress. However, we have had very promising preliminary results that validate our design and therefore prompted us to share it with the research community at large. The following sections of the paper discuss the design, implementation and performance study in detail.

2. BACKGROUND

Early operating systems relied on a monolithic kernel design. The application processes were launched in the unprivileged mode that accessed services through the traditional system call barrier, which involved a transition from the user space to the kernel space. Examples include UNIX, BSD and, Windows.

With Microkernels, most components of the operating system were isolated from one another whilst maintaining a very minimal core that provided services (threads, messaging, RPC/LPC etc.) to the components. Examples include Spring [2,5], uChoices [3], SPIN [4]. They performed better [7] in comparison to traditional monolithic kernels, but still relied on the system call barrier for protection.

Exokernels [6] provided a minimal kernel with only message passing and moved most of what was in the microkernel out into the user space. Again, they relied on a ring transition to provide protection between the kernel and the application.

Component-based OSs involve a Nanokernel that manages component interaction as seen in operating systems like Pebbles [9] and Go! [10]. The Go! Operating system, which matches the performance of KLOS, launches all the components in the most privileged level, by using a static technique called code scanning where the executables are pre-scanned for illegal/privileged instructions [8].

There were other methods proposed to increase the performance of operating systems while at the same time ensuring protection by employing the segmentation capability of x86-based processors [1,11,12]. Our design also makes use of this capability, doing away with the down-call to the protected kernel. Lastly, our method is dynamic, which means there is no code pre-scanning or other procedures that need to be applied on application code before it is executed.

3. DESIGN

Our architecture, at its heart consists of an event core that is responsible for acting upon external events (hardware interrupts and processor generated exceptions). Events are the only means of vertical up-calls to the unprivileged domain. There are no down-calls to the privileged domain resulting in no context switches during normal program flow. All the components of a typical OS like the memory manager, process manager, device drivers etc. run in the unprivileged domain and have a horizontal mode of interaction.

Event Core

The Event Core is the heart of the KLOS architecture. The amount of processing done in the event core is minimal and restricted to transferring control to the unprivileged domain and performing an optional Translation Look-aside Buffer (TLB) flush. It is important to note that the Event Core is not just another name for the "kernel" as viewed in traditional OSs. Unlike traditional kernels, in KLOS there are no "down-calls" to the Event Core.

The TSS created during OS startup is initialized with support for the Event Core. This includes the event core stack that needs to be intact upon event triggering, as it is a part of the Trusted Computing Base (TCB) of the OS. The Event Core is composed of event stubs, one for every hardware interrupt or exception the underlying processor architecture supports. These event stubs are registered once at system initialization. The exact flow of how an event is handled is as depicted in figure 1.

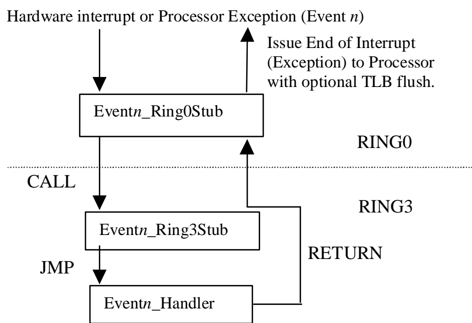

Figure 1: Event Core

Upon a hardware interrupt or exception, the processor switches the current context to the privileged level (Ring0) and transfers control to a vectored interrupt or exception handler corresponding to the interrupt or exception via the Interrupt Descriptor Table (IDT). In this case, each of the handlers will be pointing to an event stub “Event*n*_Ring0Stub”. Event*n*_Ring0Stub will reflect the interrupt or exception ‘n’, to the unprivileged level (Ring 3) stub “Event*n*_Ring3Stub” via a far CALL, switching privilege levels. This ring 3 stub in turn executes a JMP to a registered event handler, “Event*n*_Handler” at the same unprivileged level.

Once “Event*n*_Handler” has completed its execution, control returns back to “Event*n*_Ring0Stub” via a far return, switching back to the privileged level (Ring 0). “Event*n*_Ring0Stub” then performs an optional TLB flush. Event Handlers in subsystems such as the scheduler or the memory manager modify virtual memory mappings. Committing changes to such mappings requires a TLB flush. Finally, “Event*n*_Ring0Stub” issues an end of interrupt or exception signal to the processor.

Protection

To achieve a similar level of protection as with a ring-transition mechanism, we make use of a combination of segmentation and TSS. Below, we discuss the various kinds of protection that KLOS provides.

Memory protection

There are two pairs of unique data segment selectors – one for the application data access (DSAPP), which is limited to the application address space and the other for the protected OS data space (DSOS), encompassing the entire 4GBytes address space. All segment descriptors have a Descriptor Privilege Level (DPL) of 3 (unprivileged). There is only one code segment selector that is shared between the application and the OS. The code segment descriptor is constructed so that it is only executable and not readable.

The DSAPP selector prevents the application from tampering with critical OS structures, as it is restricted only to the application address space. However, the code segment selector spans the entire addressing space since the application needs to execute code in the OS. But, the fact that the code segment selector is execute-only prevents the application from either disassembling or reading OS code or data in order to discover the DSOS selector.

IO Protection

KLOS achieves I/O protection by making use of the 80x86 I/O Bitmap structure in the TSS and the DSAPP selector addressing limitations. Protection in case of memory mapped I/O is achieved by mapping I/O addresses in the OS data region. This means that the applications cannot do memory mapped I/O by themselves without going through the OS since any access to a memory location outside of the limits of the DSAPP selector will result in a general protection fault that is handled by the Event Core.

To understand how KLOS achieves legacy I/O protection, it is important to know about the TSS I/O Bitmap and how the Intel 80x86 uses it. The CPU has provision for a bit-mask array (called the I/O Bitmap) referenced by the TSS. Each bit in this array corresponds to an I/O port. If the bit is a 1, access is disallowed and an exception occurs whenever access to the corresponding port is attempted. If the bit is a 0, direct and unhampered access is granted to that particular port.

Any process with a Current Privilege Level (CPL) that is numerically greater than the I/O Privilege Level (IOPL) must go through the above described I/O protection mechanism when attempting port I/O. KLOS makes use of this prominent feature of the 80x86 class of CPUs, by setting the IOPL < CPL and controlling the ports accessed by using the I/O Bitmap. If there is a general protection fault due to an illegal I/O access, the Event Core dispatches the fault to the appropriate OS fault handler that then decides either to give the application a chance to recover from the fault or to terminate it altogether.

The section of the OS code that wishes to do legacy I/O must obtain permissions to set the corresponding bit in the TSS I/O Bitmap to a 0 so that access to that particular port is allowed. A point to be noted is that OS code can access the TSS I/O Bitmap of the current process, since the DSOS segment selector encompasses the complete memory addressing range. OS code can do memory mapped I/O without any prior setup as memory mapped I/O does not depend on the TSS I/O Bitmap.

The CPU stores the pointer to the memory location of the I/O Bitmap in the TSS. In KLOS this pointer points to a virtual address that lies exactly on a page boundary. Each process has a separate I/O Bitmap area allocated to it. Switching to a new set of I/O privileges upon a scheduler pre-emption, is accomplished by simply re-mapping the page tables to make the TSS pointer point to the I/O Bitmap area of the process being switched to.

Other Protection mechanisms

The Intel 80x86 (and compatibles) class of processors support instructions which can provide the location of certain system tables such as the GDT, the LDT/IDT, the TSS etc.

The problem arises from the fact that instructions such as the SGDT/SIDT/SLDT/STR can be executed at CPL=3 (unprivileged). So an application designed to peek or poke into the system tables will make use of such instructions to locate the area in memory where these structures are stored and manipulate them according to their will.

KLOS however is immune to attacks involving such instructions to locate and possibly manipulate critical system structures. KLOS stores these critical system structures in memory that is not accessible using the DSAPP selector. This means that though nothing is done to prevent the execution of these instructions, the application can at most know the location of the structures but can never read or write to them.

The most dangerous attack against KLOS is the attack involving locating the DSOS selector. Access to this selector empowers an application with read write privileges, which allow it to manipulate any critical OS structure in memory at will. However, KLOS thwarts this attack by relocating the DSOS selector upon sensing a potential attack.

We note that segment selectors in the Intel 80x86 are integers from 0 through 65528 in increments of 8 (one set each for the GDT and LDT). A potential attack involving guessing the DSOS selector could proceed along two lines. An application could use a brute-force method loading its data selector with values in the range of selectors possible, and each time trying to reference a memory region outside of its space. If there is an exception during the access, an application exception handler that is registered, gets the callback and increments the index. When there is no exception for a particular index, the application knows it has got hold of the DSOS selector for that session. Another variation to this theme is to create multiple threads each randomly trying to locate the magic selectors using the same technique as described.

To make locating the DSOS selector hard (but not impossible) KLOS relocates the DSOS selector to a new random location on every three "segment not present fault" exceptions. This will entail patching the service trampoline to reflect the relocated DSOS selector value. When a "segment not present fault" is generated, KLOS assumes that there is an attempt to break in and terminates the application. Three such successive faults trigger a relocation.

Considering that we use both the GDT and the LDT for relocation, there are 16382 (65528*2/8) possible values. Therefore, the probability of the first guess being correct is 1/16382, if the first guess fails the second guess has a probability of success of 1/16381 and the third has a success of 1/16380. Therefore the probability of success is 1/16382+1/16381+1/16380 (either the first is a success or the second or the third). This comes up to 1.83*10^-4. That is around 2 in 10000 chance of a correct guess.

Service Calls

We came up with a couple of designs for a service call implementation in KLOS. Both designs include a prologue and an epilogue code area. The service call prologue code is responsible for loading the data segment selector with the DSOS selector prior to executing the actual service call in the OS address space. Upon return from the service call, the data segment selector is reloaded with the DSAPP selector and execution is resumed. The necessity of the prologue/epilogue code is to – (1) establish the DSOS selector to address the entire addressing space and (2) to prevent the application from seeing the value of the DSOS selector.

The first design is as shown in figure 2(a). When an application makes a service call, it transfers control to the KLOS service trampoline area where the actual service call is executed by means of a Service Table. The Service Table is merely an array of 32-bit pointers to the various system calls. The second design is as shown in figure 2(b). In this, each of the service calls has their own prologue and epilogue code eliminating the need for a Service Table.

4. PERFORMANCE

To validate our design we implemented a prototype of KLOS. For test purposes, an AMD Athlon XP 1.3 GHz and an Intel Pentium III 1 GHz processor were used. The test environment consisted of a DOS shell executing in real mode and a DOS executable running on top of it. The executable switches from real to protected mode and performs a NULL system call. The choices for the NULL system call implementation are, (i) a traditional trap/interrupt based mechanism, (ii) KLOS Design A: using the service trampoline area, and (iii) KLOS Design B: using a separate prologue and epilogue for each service call.

In order to keep the protected-mode environment simple and the service call method predictable, IRQ handlers and other modules like memory manager, scheduler etc., were not programmed. A couple of TSSs were setup. One was set to execute at the privileged level while the other was set to execute at the unprivileged level. This was done to simulate a ring transition thereby measuring the latency of a traditional trap/interrupt based service call.

We used processor clock cycles as the performance metric for comparison. This metric is chosen, as it does not vary across processor speeds. The RDTSC instruction was used to measure the clock cycles. The timings for the best-case were measured by bringing the code being executed into the processor's L1 cache using tight loops. The worst-case timings were measured by executing the code only once (ensuring its absence from the L1 cache). The average-case was measured by executing the same copy of the code a few times in sequence. The performance measurements are shown in Table 1. From the performance numbers, one can observe that the system call implementation in KLOS is more efficient than the current widely implemented methods of system calls.

A few words on the likelihood of the above-cited cases is in order. The best-case of Design A is no doubt the fastest of all, but in practice turns out not being the most likely case (we have a logical explanation for this but space restrictions prevent us from discussing it here). The average-case of Design B, from our tests, was the most likely scenario. The point to note is that both these cases are much faster than the best-case measure of the traditional trap/interrupt based mechanism. In summary, both KLOS designs A and B perform an order of magnitude better (even in the worst-case) than traditional service-call mechanisms.

5. CONCLUSION

The results show that the KLOS service-call design provides better performance than current ring-transition

oriented operating systems. At the same time it provides the protection facilities as available in contemporary operating systems. The target environment for KLOS is desktop computing and multimedia operating systems where security is not of utmost important.

Our next step is complete system integration with all components such as the scheduler and the memory manager. The idea of our service call method may open up the mechanism of doing away with the traditional buffer copy mechanism between user and kernel mode, which if successful, will further improve the overall performance of the OS. Further, we need to account for other forms of latency such as the context switch during scheduling and hardware interrupt latency with the current design of our event core. We are working towards doing away with the event core and handling interrupts and exceptions at the unprivileged level.

6. BIBLIOGRAPHY

[1] J. Keedy. Paging and small segments: A memory management model. 8th World Computer Congress, Melbourne, 1980.

[2] G. Hamilton and P. Kougiouris., The Spring nucleus: A microkernel for objects. USENIX 1993, pages 147-159.

[3] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer., SPIN - an extensible microkernel for application specific operating system services. 1994 European SIGOPS Workshop, 1994.

[4] R. H. Campbell and S. M. Tan, μChoices: An Object-Oriented Multimedia Operating System. In Fifth Workshop on Hot Topics in Operating Systems, Orcas Island, WA, May 1995.

[5] S. Radia, G. Hamilton, P. Kessler, and M. Powell. The Spring object model. USENIX Conf. on Object-Oriented Technologies, Monterey CA (USA), June 1995.

[6] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating sytem architecture for application-level resource management. 15th ACM Symposium on Operating System Principles, pages 251 - 266, 1995.

[7] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. The performance of μ-Kernel-based systems. ACM 16th Symposium on Operating Systems Princpiles, pages 66 - 77, 1997.

[8] G. C. Necula. Proof-carrying code. 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997.

[9] E. Gabber, J. Bruno, J. Brustoloni, A. Silberscatz, and C. Small. The pebble component-based operating system. 1999 USENIX Technical Conference, Monterey, CA, June 1999.

[10] G. Law, J. McCann. A new protection model for component based operating systems. IEEE IPCCC 2000.

[11] F. J. Ballesteros, R. Jimenez, M. Pati˜, F. Kon, S. Arevalo and R. Campbell. Using interpreted CompositeCalls to improve operating system services. S/W. Pract. and Exp. 2000. 30:589–615.

[12] T. Shinagawa, K. Kono, T. Masuda. Fine-grained Protection Domain based on Segmentation Mechanism. Japan Society for Software Science and Technology 2000.

Service call method	Processor	Best Case	Worst Case	Average Case
Regular INT XX	AMD	145	750	145
Regular INT XX	Intel P3	217	1103	217
KLOS Design A	AMD	15	707	31
KLOS Design A	Intel P3	28	1057	72
KLOS Design B	AMD	25	170	37
KLOS Design B	Intel P3	38	220	58

Table 1: Performance metrics comparing traditional trap/interrupt based service call and KLOS service calls.

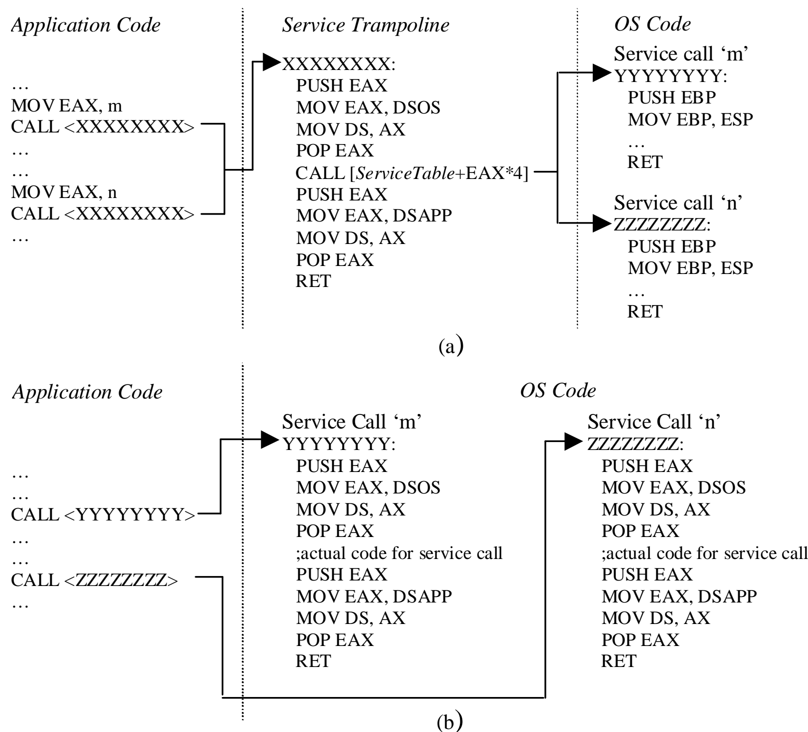

Figure 2: KLOS Service-Call Designs