A High Performance Kernel-Less Operating System Architecture

Amit Vasudevan, Ramesh Yerraballi, Ashish Chawla
Department of Computer Science and Engineering
University of Texas at Arlington
416 Yates, 300 Nedderman, Arlington TX 76019-0015
vasudeva@cse.uta.edu, ramesh@cse.uta.edu, 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 processor 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 unprivileged. KLOS is a Kernel-Less Operating System built on the basis of such a design. The KLOS service call mechanism is an order of magnitude faster than the current widely implemented mechanisms for service or system calls with a 4x improvement over the traditional trap/interrupt and a 2x improvement over the Intel SYSENTER/SYSEXIT fast system call models.

Keywords: Operating System, High Performance, Kernel-Less Architecture.

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 processor architectures that support segmentation (x86 class), 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 kernel as perceived in current operating systems, (2) operating system services are accessed without a ring transition, and (3) all processes and the operating system execute at the unprivileged level, while at the same time supporting features available in contemporary operating systems such as private addressing space for every process, per-process virtual memory mappings, inter-process communication facilities etc. KLOS, a Kernel-Less Operating System is a realization of this design.

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. We primarily target this design towards multimedia workstations, desktop computers, gaming consoles, portable digital assistants etc., where performance is the overriding design goal. In such systems it is acceptable to trade off stability for performance.

KLOS is able to achieve high performance due to its architecture and its service call mechanism. The service call mechanism of KLOS results in a general 4x improvement over the traditional trap (interrupt) and a 2x improvement over the Intel SYSENTER/SYSEXIT fast system call models. KLOS is still undergoing significant revisions, but the preliminary performance studies that we conducted have prompted us to share our work with the research community at large.

This paper is organized as follows: In Section 2, we discuss the architecture of KLOS in its present version. In Section 3, we present preliminary performance values for the service call mechanism employed in KLOS and compare it against the currently existing mechanisms. In Section 4, we consider related work on operating system architectures and compare them with KLOS. We conclude the paper in Section 5 summarizing our contributions to date with directions for future work.

2 Design

Traditional operating systems comprise of a kernel, that is responsible for providing the core services of the operating system and a shell or applications that reside on top of the kernel using its services and providing an interface to the user. In KLOS, there is no kernel per se. Instead, the entire operating system is made available to each application as a part of its execution space, running at the same unprivileged level. Figure 1 shows the architecture of KLOS in its present version. Our architecture, at its heart consists of an event core that is responsible for acting upon external events (hardware interrupts and processor generated exceptions).

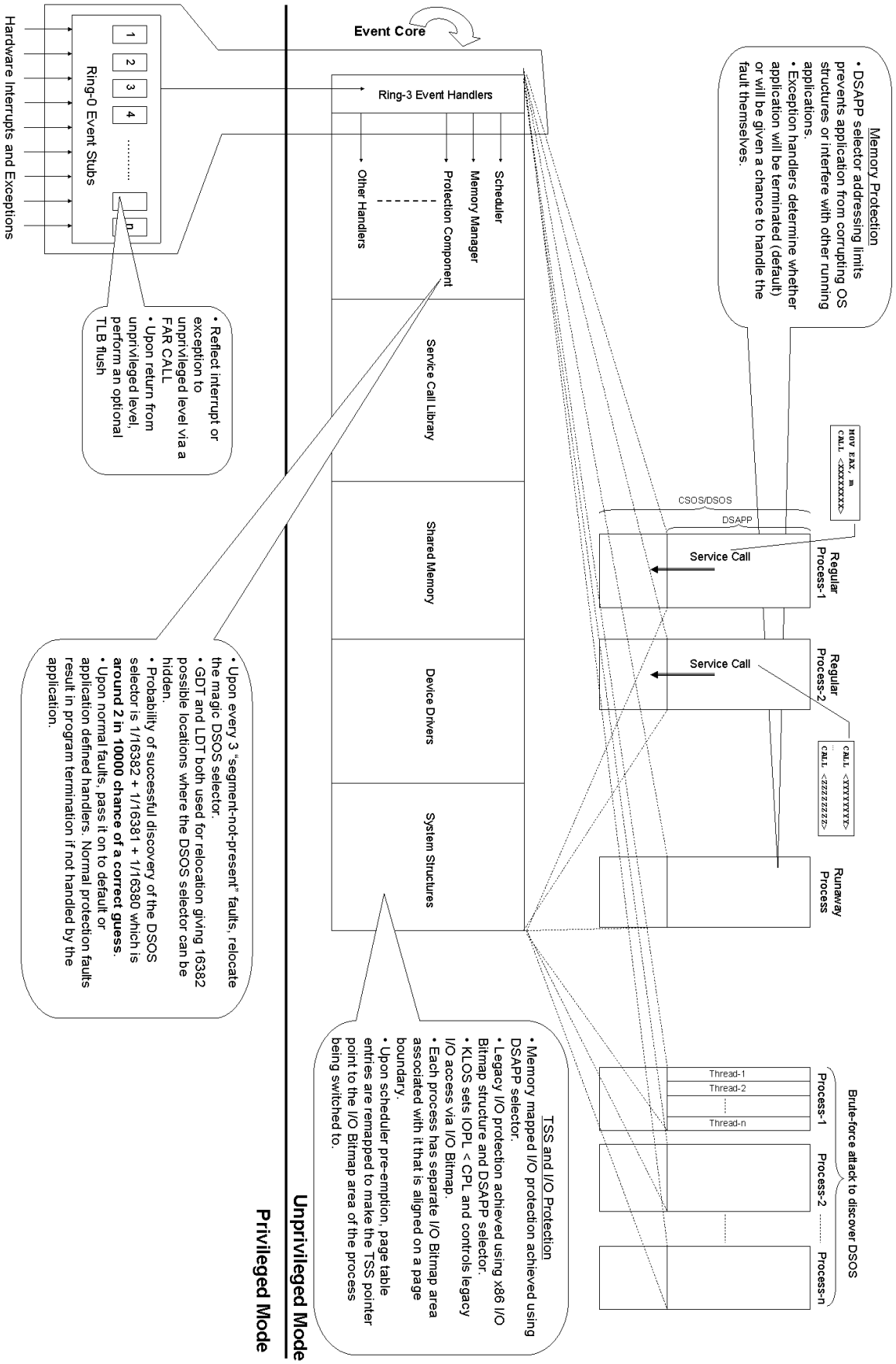

Figure 1: Architecture of KLOS

Events are the only means of vertical up-calls to the unprivileged level. There are no down-calls to the privileged level eliminating a protection domain transfer during normal program flow. All the components of a typical operating system like the memory manager, scheduler, process manager, device drivers etc. run in the unprivileged level and have a horizontal mode of interaction.

2.1 Event Core

The Event Core is the heart of the KLOS architecture. The amount of processing itself is minimal and restricted to transferring control to the unprivileged level 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 operating systems. A kernel in the traditional sense is a provider of services. Application and system processes access these services provided by the kernel via a down-call resulting in a protection domain transfer. Unlike traditional kernels, in KLOS there are no "down-calls" to the Event Core.

In KLOS, a TSS is created during the operating system start-up phase. This TSS is initialized with support for the Event Core. This includes the event core stack that needs to be intact upon event triggering. The Event Core is the only component of KLOS that executes within the privileged level and hence forms the Trusted Computing Base (TCB) of the operating system. The Event Core is composed of event stubs, one for every hardware interrupt or exception that the underlying processor supports. These event stubs are registered once during system initialization. The exact flow of how an event is handled is as depicted in figure 2.

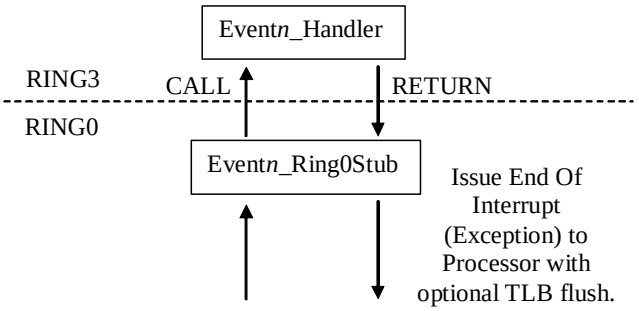

Figure 2: Event Core

Upon a hardware interrupt or exception, the processor switches the current context to the privileged level (Ring 0) 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 "Eventn_Ring0Stub". Eventn_Ring0Stub will direct the interrupt or exception 'n', to the registered event handler, "Eventn_Handler" at the unprivileged level (Ring 3) via a FAR CALL. Once "Eventn_Handler" has completed its execution, control returns back to "Eventn_Ring0Stub" via a FAR RETURN, switching back to the privileged level (Ring 0). "Eventn_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, "Eventn_Ring0Stub" issues an end of interrupt or exception signal to the processor.

2.2 Process Address Space

As seen from figure 1, each process under KLOS is given its own virtual address space which spans the entire range of addressable memory (typically 4 GB). A part of this is mapped to the operating system and other structures like shared memory regions etc, while the rest of it is available for use by the process.

In the x86 segmented architecture there are segment descriptors that define the characteristics of a segment and there are segment selectors which reference these descriptors. The segment descriptors go into the Global Descriptor Table (GDT) or the Local Descriptor Table (LDT) while the segment selectors are basically numeric indices into these tables. Each segment descriptor has a numeric privilege associated with it, called the Descriptor Privilege Level (DPL) that decides whether the segment can be referenced or used from within the Current Privilege Level (CPL). The value for the DPL ranges from 0 (highest privileged) to 3 (least privileged). The privilege levels are also commonly called rings of protection (from Ring 0 through Ring 3).

Though the x86-architecture allows for 4 different privilege levels (Ring 0 through Ring 3) as discussed, KLOS only makes use of two of them. They are Ring 0 for the privilege mode and Ring 3 for the unprivileged mode. In KLOS, there are two unique data segment selectors - one for the application data access (DSAPP) and the other for the operating system data access (DSOS). There is only one code segment selector (CSOS) that is shared between the application and the operating system. All segment descriptors are setup to have a DPL of 3 (unprivileged).

A part of a process virtual address space is mapped to the operating system as stated before. However, a process is prevented from directly addressing the areas where the operating system is mapped, since the DSAPP segment selector is limited to the application address space which does not include the mapped operating system area. On the other hand, the DSOS segment selector encompasses the entire virtual address space. This ensures that the operating system code can access any area within the currently executing virtual address space. As seen from the above discussion, there is a need to switch between the two data segment selectors depending upon whether the code being executed is within the application or the operating system address spaces. This mechanism is achieved via prologue-epilogue stubs which are described later.

The CSOS segment selector for executing code also encompasses the entire virtual address space and the code segment descriptor is constructed so that it is only executable and not readable. The rationale behind making

the code segment execute-only, is to prevent any form of malicious attack to thwart the protection provided by the operating system. This is further described in section 2.3.1.

Considering the code segment is shared between the operating system and the application, and the fact that it is execute-only, result in situations that deserve some explanation. One issue would be as to whether a privileged component is needed to load the processes in the first place. However, this is not the case. Since the CSOS and the DSOS selectors encompass the entire virtual address space, the DSOS selector can be considered as a data alias to the code segment and can be used to load processes while at the unprivileged level.

Another issue would be as to what happens to code that try to read data using the CSOS selector as an overriding prefix. This would definitely not be permitted with our operating system design as the CSOS selector is execute-only. We do not see this to be a problem however, as 99% of application code that we studied never made use of such esoteric data access and the 1% that did belonged to the category of programs which were malicious in some form or the other. In other words, in the worst case that a regular application does follow such a form of overriding data access with the code segment selector, the system will detect it and refuse access and will eventually result in the termination of the process.

2.3 Protection

Protection is a mechanism to prevent faulty processes from inadvertently destabilizing the operating system or malicious processes from purposefully tampering with the system in any way. Protection can be broadly classified into memory and I/O protection. In a traditional ring-transition oriented operating system, achieving memory and I/O protection is trivial as the operating system and the application run at different privilege levels. Most if not all of the work is done by the processor itself to ensure that protection is in place. However, KLOS does not use a ring-transition mechanism to isolate the operating system from the other processes.

Thus, to achieve a similar level of protection as with a ring-transition mechanism, we make use of segmentation and TSS, both of which are features provided by all x86-based processors. Below, we discuss the various kinds of protection that KLOS provides.

2.3.1 Memory Protection

The idea of memory protection is to isolate the address space of one process from the other. This is so that if a process malfunctions, the worse it can do is to bring itself down and not interfere with the other executing processes or the operating system itself.

Memory protection under KLOS is easily achieved using the way selectors are defined in the operating system. As we saw before, the addressing limits of the DSAPP selector prevents the application from tampering with critical operating system structures. However, the DSOS selector which is at the same privilege level encompasses the entire virtual address space. Access to this selector will empower an application to access any part of the virtual address space including the operating system itself. However, the DSOS selector can only be discovered in two possible ways. As we will see in section 2.4, there are prologue/epilogue stubs which switch between the DSAPP and the DSOS selectors acting as entry points into operating system code.

Thus, if an application can disassemble the prologue/epilogue stubs, it can easily discover the DSOS selector. However, the prologue/epilogue stubs lie outside the addressable limits of the DSAPP selector. Hence, the only other way for an application to disassemble the stubs would be to use the CSOS selector. But, the fact that the CSOS selector is execute-only prevents the application from either disassembling or reading operating system code or data in order to discover the DSOS selector. The second method of DSOS discovery and its antidote is described in detail in section 2.3.3.

2.3.2 I/O Protection

KLOS achieves I/O protection by making use of the x86 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 operating system data region. This means that the applications cannot do memory mapped I/O without going through the operating system since the DSAPP selector does not encompass the memory regions containing operating system code or data. Thus, access to a memory location outside of the limits of the DSAPP selector results in a general protection fault (that is handled by the Event Core) ensuring memory mapped I/O protection.

To understand how KLOS achieves legacy I/O protection, it is important to know about the TSS I/O Bitmap and how the x86-based processors use it. In the x86 (and compatibles), 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, access is granted to that particular port.

Any process with a 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 x86 class of CPUs, by setting the IOPL < CPL and forcing port access by the use of 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 operating system fault handler that decides whether to give the application a chance to recover from the fault or to terminate it altogether.

The section of the operating system 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 operating system code can access the TSS I/O Bitmap of the current process, since the DSOS

selector encompasses the entire virtual address space. Also, since one does not need cache invalidation to cement changes to the TSS IO bitmap on the x86 class of processors, port access is simply controlled by setting/resetting access bits for the particular port. Operating system 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. KLOS maintains a separate I/O Bitmap area for every process. 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 point to the I/O Bitmap area of the process being switched to. The idea of aligning the IO Bitmap on a page boundary thus keeps the latency incurred, in switching to the IO privileges for a different process, minimal.

2.3.3 Other Protection Mechanisms

The x86 class of processors support instructions, which can provide the location of certain system tables such as the GDT, the LDT, the TSS etc. The problem arises from instructions such as, SGDT/SIDT/SLDT/STR that provide access to these tables. These instructions can be executed at a CPL of 3 (unprivileged). So an application designed to read or write into the system tables may 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 there is nothing 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 attacks against KLOS are the attacks involving locating the DSOS selector. As we saw in section 2.3.1, access to this selector empowers an application with read write privileges, which allows it to manipulate any critical operating system structure in memory at will. We also saw one of the two possible methods of DSOS discovery and how that is prevented in section 2.3.1. In this section we will discuss the second method of possible DSOS discovery. In our opinion the DSOS discovery, is the only proven method by which the OS system structures are exposed to an application in our OS. It is also the only difference between a ring-transition oriented operating system and ours when it comes to security.

We note that the x86-class of processors provide a maximum of 8192 possible segment descriptor definitions within a descriptor table. Since there are two possible descriptor tables in the LDT and the GDT, there are thus a total of 16384 possible segment descriptor definitions. The segment selectors, which are indices into the descriptor tables, are 16-bit values comprising of a 2-bit Requestor Privilege Level (RPL) ranging from 0 to 3, a 1-bit table indicator (0 for the GDT and 1 for the LDT) and a 13-bit index within the specified table. The index values range from 0 through 8191 inclusive.

The attack involving guessing the DSOS selector makes use of certain facts. (1) KLOS allows for application level vectored and structured exception handling (where the application is given a chance to recover from a fault) on the lines of those offered in commercial desktop operating systems like windows, os/2 etc. (2) on the x86 class of processors, nothing prevents an application from reloading its data segment selector with another value in the unprivileged level and (3) on the x86 class of processors, access to a memory location using a segment selector that has no corresponding segment descriptor definition in either the LDT or the GDT causes a “segment not present fault”.

Armed with the above information, a potential attack could proceed as follows:

1. An application registers an exception handler call-back in structured exception handler chain.
2. The application then executes one of the SGDT/SLDT/SIDT instructions to get hold of the address of either of these system tables in the operating system data area.
3. The application then uses a brute-force method loading its data selector with the 13-bit index value ranging from 0 through 8191 for both the LDT and the GDT (by loading the 1-bit table indicator with a 0 or 1). In other words, the application iterates over all the 16384 possible segment descriptors definitions combining the LDT and the GDT, each time attempting to read a byte from the memory area obtained in step 2. If there is an exception during the read operation, the exception handler registered by the application in step 1 gets control with the appropriate exception code. If the exception code indicates a "segment not present fault", then the application merely proceeds after ignoring the fault with the next index value. However, if there is no exception during the read operation, the application knows that the current index value is that of the DSOS selector.

The easiest way to prevent such an attack would be to prevent delivery of "segment not present fault" codes to application level exception handlers. However, there might be instances where a runaway process could cause a "segment not present fault" too. To prevent "segment not present faults" from reaching application level exception handlers would mean that we risk the condition of not providing a regular runaway process a chance to recover during certain conditions. Thus, we devised a probabilistic prevention technique that we describe in the following paragraphs.

To make locating the DSOS selector hard (but not impossible) KLOS relocates the DSOS selector to a new random location within either the LDT or the GDT on every three "segment not present fault" exceptions.

Relocating the DSOS selector entails patching the prologue code in the prologue/epilogue stubs (discussed later). The rationale behind choosing to relocate on every three "segment not present fault", was to minimize the latency due to a relocation of DSOS, while at the same time ensuring that a potential attack is sensed as quickly as possible and thwarted.

Considering that we use both the GDT and the LDT for a relocation, there are 16384 possible selector values that the application has to guess from. However, assuming that the application has ruled out the obvious possibilities of the DSAPP and the CSOS, it has 16384 - 2 = 16382 values to guess the DSOS from. Since the application has three tries before a relocation is performed, 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 if the second guess fails the third has a success of 1/16380. Therefore, the probability of successfully guessing the DSOS selector is 1/16382 + 1/16381 + 1/16380 (either the first is a success or the second or the third), which comes up to $1.83*10^{-4}$. Note that this argument holds true even if one considers a variation of the attack where an application employs multiple threads or processes to simultaneously guess the DSOS (shown in figure 1). This is because (1) the relocation process is carried out globally and is not thread specific and (2) KLOS performs pre-emptive multithreading, resulting every thread being presented a different problem as far as trying to locate the DSOS is concerned.

As far as issues concerning other exploits like buffer overflow, etc. to bring certain services (or the system) down, we believe that our system is no better but at the same time no worse than other similar operating systems.

2.4 Service Calls

A Service or System call is a mechanism using which application and system processes access the services provided by the operating system. A service call in almost all cases involves a ring-transition to the kernel where the required function is then performed. Traditionally this is achieved by using the trap (interrupt) mechanism which transitions into the kernel to execute the required function. Intel also introduced the SYSENTER/SYSEXIT based fast system call mechanism for handling the ring-transition oriented system calls more efficiently. However, in case of KLOS, there is no ring-transition during service calls as discussed earlier.

Below, we discuss two possible designs for service call implementation in KLOS. Both designs include a prologue and an epilogue code area called prologue/epilogue stubs. 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 stubs is to - (1) load the data segment selector with DSOS, so that the operating system can address the entire virtual address space and (2) to prevent the application from seeing the value of the DSOS selector. The two service call designs are depicted in figures 3(a) and 3(b).

In the first design (figure 3a), 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 service calls. The service call itself is identified by a unique number and control is transferred to the appropriate call via the Service Table as shown. In the second design (figure 3b), each of the service calls have their own prologue and epilogue stubs eliminating the need for the Service Trampoline Area and the Service Table. The rationale behind the second design was to observe if the Service Trampoline Area and the Service Table contributed to the latency of a service call in any way.

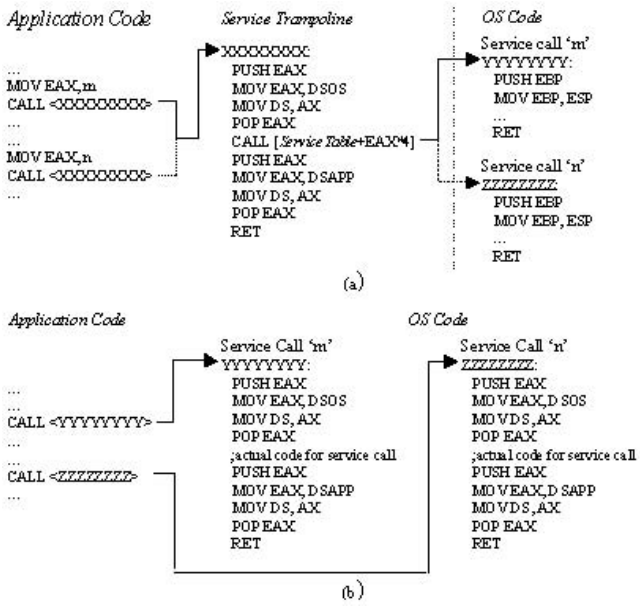

Figure 3: (a) KLOS Service Call Design-A using ServiceTable (b) KLOS Service Call Design-B without ServiceTable

3 Performance Evaluation

To validate our design we have 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. Our aim was to obtain a preliminary performance study of our service call mechanisms. This study would in turn enable us to decide on the service call mechanism we would go with in the final version of KLOS. We used a stripped down version of KLOS devoid of IRQ handlers and subsystems like the scheduler, memory manager etc. While excluding such subsystems affects the absolute performance measure, they have little impact on the relative measure. As our goal is to compare the service call design alternatives it is the relative measure that is of concern. Further, the absence of such subsystems allows for a deterministic test methodology keeping the service call mechanisms predictable with respect to measurements.

For test purposes, all executed service calls were very simple functions which had no processing and simply returned to the caller. The choices for the service call implementation are, (1) a traditional trap (interrupt) based mechanism, (2) the Intel SYSENTER/SYSEXIT fast system call mechanism (3) KLOS Design A, using the service trampoline area, and (4) KLOS Design B, using a separate prologue and epilogue for each service call. The first two choices were implemented by setting up a couple of TSSs. 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) and the Intel SYSENTER/SYSEXIT based service calls.

Readings were taken at various points in the service call handler and the unprivileged application code both before and after execution of the 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 and also since it is a standard in literature related to micro benchmarks. The RDTSC (ReaD Time Stamp Counter) 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 invalidating the processor internal caches using the cache invalidate instructions. We measured what we consider to be the average-case, by having 50 different service calls (each of them not doing any processing and simply returning to the caller) and calling them at random in a section of application code. This section of application code was then repeated a few times in sequence. This amounted to some of the code being in the processor L1 cache.

In both the worst and the average case measurements the number of runs over which the measurements were averaged was such that we have a confidence level of 98% that the percentage of error in our measurements is ±2%. We have measured the system call performance by accommodating the TLB and the cache. In other words, our mechanism does not turn off or set anything special as far as the TLB or the cache is concerned. Thus, we used the same environment (TLB/cache effects) on all the 3 types of system call mechanisms (trap, SYSENTER/SYSEXIT and KLOS).

The performance measurements are shown in figure 4. From the performance numbers, one can observe that the service call mechanism of KLOS is more efficient than the current widely implemented mechanisms of service calls. The mean of the performance metrics for both the processors in the best, average and worst cases establish a general 4x improvement over traditional trap/.interrupt and a 2x improvement over the Intel SYSENTER/SYSEXIT fast system call models. We also observe that the best-case measure of either of the service call mechanisms of KLOS (32 and 22 clock cycles for Design B and Design A respectively) are much faster than the best-case measure of the traditional trap/interrupt based mechanism (181 clock cycles) or the Intel SYSENTER/SYSEXIT based fast system call mechanism (92 clock cycles). More specifically, one can see that service call Design B consistently performs better than its counterparts in all of the best, worst and the average cases. Also from the graph in figure 4 and the metrics we obtained, one can notice that the AMD processor clocks significantly lesser cycles than the Intel during the measurements. We speculate this to be a result of AMDs advanced dynamic execution unit and its cache subsystem which seem to outperform that of Intel's.

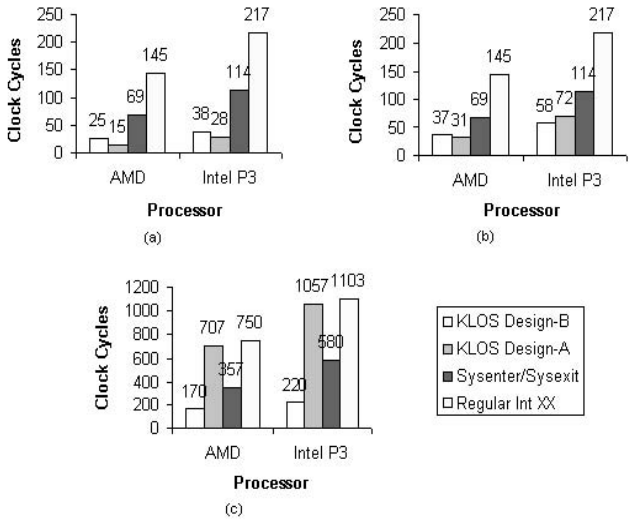

Figure 4: Comparison of Service Call Performance; (a) Best-Case (b) Average-Case and, (c) Worst-Case

A few words on the likelihood of the above-cited cases are in order. The best-case of Design A is no doubt the fastest of all, but in practice happens infrequently. The average-case of Design B, from our tests, was the most likely scenario. This happens due to the fact that Design A uses a Service Table to transfer control to the appropriate service call. In the best-case, when the entire service trampoline is in the processor L1 cache the Service Table has very little effect on the latency. However, we found out that when the service trampoline was not in the processor L1 cache, the latency of the service call is dependent on the relative distance of the target service from the point where the call to the target service is made using the Service Table. We speculate this to be an artifact of code-prefetching. Design B eliminates this latency by including a separate prologue and epilogue code area for each service call and aligning each of the prologue code on a page boundary.

In summary, KLOS service call Design-B performs an order of magnitude better (even in the worst-case) than traditional service call mechanisms.

4 Related Work

Early operating systems relied on a monolithic kernel design. The application processes were launched in the unprivileged level that accessed services provided by the kernel through the traditional system call barrier, which involved a transition from the unprivileged level to the privileged level. Examples include UNIX and its variants, Windows, OS/2, Clouds (P. Dasgupta, R. LeBlanc, M.

Ahamad, and U. Ramachandran 1992) etc. There were operating systems based on the object oriented design such as Choices (R. Campbell, G. Johnston, and V. Russo 1987) that paved the way for the movement away from the monolithic genre.

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 (G. Hamilton and P. Kougiouris 1993, S. Radia, G. Hamilton, P. Kessler, and M. Powell 1995), µChoices (R. Campbell, and S. Tan 1995), SPIN (B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer 1994), EMERALDS (K. Zuberi, P. Pillai, and K. Shin 1999), EROS (J. Shapiro, J. Smith, and D. Farber 1999) etc. They performed better in comparison (H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter 1997) to traditional monolithic kernels since majority of the services provided by the kernel were moved to the unprivileged level which resulted in lesser overhead while accessing system services. However they still relied on the system call barrier for inter-component protection and component support services. A detailed performance analysis of various popular microkernel based operating systems can be found in (G. Law, and J. McCann 2000).

Exokernels (D. Engler, M. Kaashoek, and J. O'Toole 1995) provided a minimal kernel with only message passing and moved most of what was in the microkernel out into the unprivileged level. Again, the message passing relied on a ring transition to provide protection rendering the exokernels to perform only marginally better than the microkernels.

Component-based operating systems involve a Nanokernel that manages component interaction as seen in operating systems like Pebbles (E. Gabber, J. Bruno, J. Brustoloni, A. Silberscatz, and C. Small 1999) and Go! (G. Law, and J. McCann 2000). The Go! operating system, which matches the performance of KLOS, launches all the components in the privileged level, by using a static technique called code scanning where the executables are pre-scanned for illegal/privileged instructions (G. Necula 1997). An alternative to code scanning was the model-carrying untrusted code execution mechanism (R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney 2003). However, this method was unsuitable for general purpose executables as it required some high level description of the code being executed.

There were some hardware and software based methods (C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang 1995, T. Li, L. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio 2002, D. Lie, C. Thekkath, and M. Horowitz 2003) proposed to increase the performance of operating systems but were unpopular due to their dependence on an already existing operating system architecture (hence limited) or ad-hoc hardware platforms that were not widespread. On the other hand, some researches aimed at increasing the performance of operating systems by employing the segmentation capability of processors based on the widespread x86 architecture (J. Keedy 1980, T. Chiueh, G. Venkitachalam, and P. Pradhan 1999, F. Ballesteros, R. Jimenez, M. Patil, F. Kon, S. Arevalo, and R. Campbell 2000, T. Shinagawa, K. Kono, and T. Masuda 2000).

While segmentation does provide protection, the penalty to switch to a segment (code) with a different protection level is still going to be in the order of traditional system call mechanisms which would be the case with the OS and applications sharing different code segments. KLOS on the other hand uses just a single code segment for both the application and the OS, doing away with the down-call to the kernel and eliminating the penalty. As seen, segmentation is used as an aid to our methodology since its original intended capability does not solve the latency problem with system calls in any case.

Palladium (T. Chiueh, G. Venkitachalam, and P. Pradhan 1999) was another operating system which also made use of the segmentation ability of the underlying processor but was all about safe kernel extensions, where kernel modules are loaded safely so that they don't bring the kernel down in case of bugs. There is still the user/kernel mode separation for normal programs. Palladium is based on Linux kernel 2.x series and uses the trap based mechanism for accessing system services.

There were operating systems based on a single address space architecture (SAOSs) like UW's Opal (J. Chase, H. Levy, E. Lazowska, and M. Baker-Harvey 1992, J. Chase, H.Levy, M. Feeley, E. Lazowska 1994). Opal has no conventional programs: all code exists as procedures residing in a shared space. Thus, threads work in a protection domain that is established depending on a trust relationship between the callee and the caller. This relationship has to be established before execution. Clearly this is not suitable for general purpose operating environment where you can potentially have any executable running and not have code existing as procedures.

Anonymous RPCs (C. Yarvin, R. Bukowski, and T. Anderson) on the other hand use virtual mappings to act as capabilities. This relies on the execute-only page permission ability of the underlying processor allowing only for memory protection. Also, the authors admit that it has a probabilistic failure rate.

KLOS also follows a probabilistic prevention of a known technique (specific to KLOS) that can be exploited by a malicious process to gain access to critical operating system structures. However, KLOS exploits the segmentation ability and the TSS of the underlying processor to implement memory as well as IO protection, but applying it to the entire general purpose operating system. Our method is also dynamic, which means there is no code pre-scanning or other procedures that need to be applied on application code before it is executed.

5 Conclusion

The results show that the service call mechanism of KLOS provides better performance than current ring-transition oriented service call mechanisms. At the same

time it provides the protection facilities close to what is available in contemporary operating systems. The target environment for KLOS is desktop computing, multimedia operating systems, gaming stations (consoles), portable digital assistants and any system where security is not of utmost importance. Also the fact that current ring-transition oriented operating systems are being compromised in any case, with the speedup that we achieved without the ring-transition and that we are able to contain errant processes completely, the only problem of containing malicious processes in a probabilistic manner seemed something we could afford considering the performance benefits achieved.

The idea of our service call mechanism may also open up the possibility 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 operating system.

Our main aim in this paper was to establish a methodology for fast system calls while at the same time containing errant processes completely and malicious processes with a probabilistic measure. Hence, we have only provided the micro benchmarks related to the system call performance. We however, do understand that our study of the system is incomplete without the complete integration performance benchmarks and I/O. KLOS is still evolving and we are working towards these issues and hope to address them in the near future. 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.

KLOS is highly scalable as it uses only 3 segment descriptors for the entire operating system in terms of the CSOS, DSOS and DSAPP. Considering portability, the service call mechanism we have developed relies on the segmentation ability of the underlying processor architecture. Though the majority of the processors out there (x86 compatible) support segmentation, some of the new generation 64-bit processors such as the IA-64 lack the support for it (R. Zahir, J. Ross, D. Morris, and D. Hess 2000). We strongly believe that the lack of such a support is an oversight on the part of the designers and hope this paper substantiates our view regarding the same.

6 References

J. Keedy (1980): Paging and small segments: A memory management model. *8th World Computer Congress.* Melbourne.

R. Campbell, G. Johnston, and V. Russo (1987): Choices (class hierachical open interface for custom embedded systems). *Operating Systems Review 21*, pages 9-17.

P. Dasgupta, R. LeBlanc, M. Ahamad, and U. Ramachandran (1992): The clouds distributed operating system. *IEEE Computer 24.*

J. Chase, H. Levy, E. Lazowska, and M. Baker-Harvey (1992): Opal: A Single Address Space System for 64-bit Architectures. In *Proc. IEEE Workshop in Workstation Operating Systems.*

G. Hamilton and P. Kougiouris (1993): The Spring nucleus: A microkernel for objects. *USENIX*, pages 147-159.

J. Chase, H.Levy, M. Feeley, E. Lazowska (1994): Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems*, 12(4).

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

R. Campbell and S. Tan (1995): µChoices: An Object-Oriented Multimedia Operating System. In *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA.

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

D. Engler, M. Kaashoek, and J. O'Toole (1995): Exokernel: An operating sytem architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 251 - 266.

C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang (1995): Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 314–324, Copper Mountain Resort, CO.

H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter (1997): The performance of µ-Kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66 - 77.

G. Necula (1997): Proof-carrying code. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France.

E. Gabber, J. Bruno, J. Brustoloni, A. Silberscatz, and C. Small (1999): The pebble component-based operating system. *USENIX Technical Conference*, Monterey, CA.

T. Chiueh, G. Venkitachalam, P. Pradhan (1999): Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, Kiawah Island, SC.

K. Zuberi, P. Pillai, and K. Shin (1999): EMERALDS: a small-memory real-time microkernel. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 277–291, Kiawah Island, SC.

J. Shapiro, J. Smith, D. Farber (1999): EROS: a fast capability system. In *Proceedings of the 17th ACM*

Symposium on Operating Systems Principles, pages 170-185, Charleston, SC.

G. Law, J. McCann (2000): A new protection model for component based operating systems. *IEEE IPCCC*.

F. Ballesteros, R. Jimenez, M. Patiˇ, F. Kon, S. Arevalo and R. Campbell (2000): Using interpreted CompositeCalls to improve operating system services. *S/W. Pract. and Exp.*, pages 589-615.

T. Shinagawa, K. Kono, T. Masuda (2000): Fine-grained Protection Domain based on Segmentation Mechanism. *Japan Society for Software Science and Technology*.

R. Zahir, J. Ross, D. Morris, and D. Hess (2000): OS and compiler considerations in the design of the IA-64 architecture. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 212-221, Cambridge, MA.

T. Li, L. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio (2002): Understanding and improving operating system effects in control flow prediction. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems*, pages 68-80, San Jose, CA.

R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney (2003): Model-carrying code: a practical approach for safe execution of untrusted applications. *In Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 15–28, Bolton Landing, NY.

D. Lie, Chandramohan A. Thekkath, and M. Horowitz (2003): Implementing an untrusted operating system on trusted hardware. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 178-192, Bolton Landing, NY.

C. Yarvin, R. Bukowski, and T. Anderson: Anonymous RPC: Low-latency protection in 64-bit address space. *Technical report*, University of California at Berkeley.