Adaptive Domain Environment for Operating Systems

Karim Yaghmour
Opersys inc.
www.opersys.com
karym@opersys.com

Abstract

Most modern operating systems have been designed to operate without restriction on the hardware they use. This is a very limited environment for both application programmers and system administrators as extension and operative modifications are impossible.

The Adaptive Domain Environment for Operating Systems (Adeos) introduced here provides an extensible and adaptive environment which can be used to enable the sharing of hardware resources among multiple operating systems or among multiple instances of the same operating system.

The implementation details of Adeos on the ix86 using the Linux OS as the host are presented as are fields of application. For each application field, a suggested deployment architecture is presented.

1 Introduction

Operating systems were born out of a desire to encompass system libraries and hardware abstraction software in order to offer users with a standardized application programming interface. In time, operating systems came to automate the sharing of hardware resources amongst multiple user programs. Hence, users no longer had control or direct access to the hardware they use. This has its advantages as it becomes possible to write portable programs and force users to abide by some rules imposed by the operating system administrators. On the other hand, all hardware accesses being channelled through the operating system, programmers became trapped in the operating system for which they programmed. In turn, users became trapped in the operating system they chose and could not use applications developed for other operating systems.

As many different and incompatible operating systems, both source-wise and binary- wise, became adopted, the gap between the programmers and users of one OS and the programmers and users of another OS became wider and wider. The propagation of computerized systems in different specialty fields did nothing to alleviate this divide. In the current state of the OS field this is somewhat unfortunate as there are often cases where the functionalities or applications of a given OS may be very useful to the users of another OS, but these users may not profit from such capabilities as the OS they use has not been designed to share its resources. The same can be said about system developers as they have no way to control the hardware outside of the capabilities provided by the running OS. Although this level of control may be desirable for debugging and performance measurement.

Current solutions to provide such a capability rely heavily on simulation or on limited-simulation supported by run-time mechanisms. Hence, the simulated OS has no direct access to the hardware and is treated as hostile by the simulator which, itself, runs on top of another operating system. This stacking of OSes becomes rapidly inefficient and limited. Other solutions attempt to build a miniature OS on top of which other OSes may come to be implemented and offer user capabilities. This, though, is not a practical approach as most users and programmers don't want new OSes, they simply want their current ones to interact more seamlessly.

What is desirable is to have an equal and trusted status among already existing OSes that use the same hardware in order to give control back to the application programmers and system administrators.

In section 2, previous work is presented and discussed. Section 3 covers the architectural concepts behind Adeos. Section 4 discusses the implementation details of Adeos on the ix86 using Linux as a

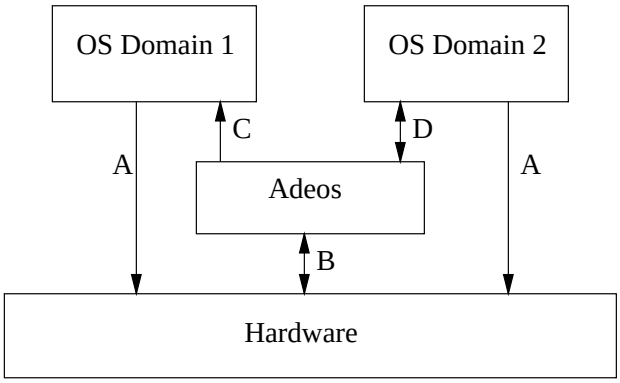

Figure 1: Adeos architecture.

host to startup the hardware. Section 5 discusses example applications of Adeos.

2 Previous work

As stated in the previous section, there are two categories of existing solutions that enable multiple operating systems to run on the same system. The first are simulation-based and provide a virtual environment for which to run additional operating systems. The second suggest the usage of a nano-kernel layer to enable hardware sharing.

In the simulation category we find tools such as VMWare [1], Plex86 [2], VirtualPC [3] and SimOS [10]. Both VMWare and Plex86 provide a virtual environnement on which ix86-based OSes can be used on top of other ix86-based OSes. VirtualPC enables Macintosh users to run ix86-based applications. Although these solutions are useful for users who desire to run applications foreign to their base OS, they provide no control whatsoever over the base OS to the user. SimOS was developed to analyze the performance and behavior of commercial operating systems under realistic workloads. Hence, it was never meant to be used in a production environment.

In the nano-kernel category we find projects such as SPACE [9, 8, 7], Cache kernel [4] and Exokernel [5]. All of these suggest building miniature hardware management facilities which can thereafter be used to build production operating systems. The problem of this approach is that it does not address the issue of currently existing operating systems and their user base. That being said, SPACE bares an important contribution to Adeos as it implements the concept of domains where each domain has its own set of characteristics.

Adeos addresses the requirements of both categories of application by providing a simple layer that is inserted under an unmodified running OS and thereafter provides the required primitives and mechanisms to allow multiple OSes to share the same hardware environment. Adeos does not attempt to impose any restrictions on the hardware's usage, by the different OSes, more than is necessary for Adeos' own operation. Instead, the actual restrictions are to be imposed by the system administrator or the system programmer. This, of course, renders the system subject to mismanagement, but the idea behind Adeos is precisely to give back control to system administrators and programmers. How this increased level of control should be used is outside the scope of this discussion.

3 Architectural concepts

As Adeos has to ensure equal and trusted access to the hardware it must take control of some hardware commands issued by the different OSes, but must not intrude too much on the different OSes' normal behavior. In order to achieve this fragile equilibrium the architecture presented in figure 1 has been adopted. It is a generalized form of the architecture presented in [7] and is not limited to providing what is referred to as portals in SPACE since it is also used to insure that every OS has a safe environment to operate in.

Each OS is encompassed in a domain over which it has total control. This domain may include a private address space and software abstractions such as processes, virtual memory, file-systems, etc. As will be covered shortly, these resources do not have to be exclusive since OSes that recognize Adeos and are able to interact with it may be able to share resources with or access the resources of other domains.

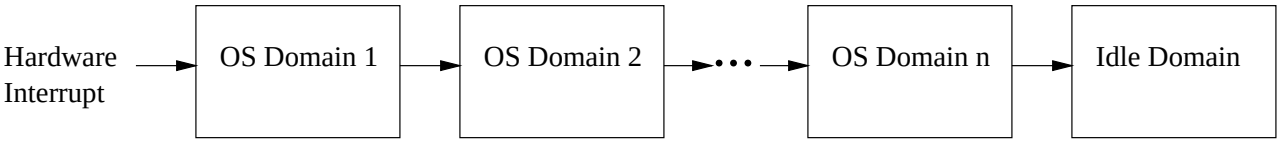

Figure 2: Adeos' interrupt pipe.

Again, Adeos does not attempt to impose any policy on the usage of the hardware except as needed for its own survival. The task of determining policy is left to the system architect.

In the type of environment provided by Adeos there are 4 broad categories of communication methods as illustrated in figure 1. Category A is a general usage of the hardware made by the different domains. This may involve any degree of memory access, hardware access and traps. Since successful memory and hardware accesses don't cause any faults, each domain operates as if Adeos wasn't present. Category B involves Adeos receiving control from the hardware because of a software or hardware interrupt (Adeos' interrupt management is discussed below). It also includes all hardware commands issued by Adeos to control the hardware. Category C involves invoking an OS's interrupt handler upon the occurrence of a trap while providing it the required information regarding the interrupt. If the domain is not aware of Adeos' presence, this may involve setting up the stack to resemble what the OS's interrupt handler expects. Category D involves two-way communication between a domain and Adeos. This is only possible if the OS in the domain is aware of Adeos' presence. This type of communication provides for maximum usage of Adeos' capabilities as it can be asked by a domain OS to grant it complete or partial access to resources belonging to other domains or even to grant it priority over other domains during interrupt handling. This may be used to provide cross-domain communication facilities as provided by SPACE.

Figure 2 presents the way interrupt handling is done in Adeos. Adeos uses an interrupt pipe to propagate interrupts through the different domains running on the hardware. As some domains may prefer to be the first to receive hardware interrupts, Adeos provides a mechanism for domains to have access to priority interrupt dispatching. In effect, Adeos places the requesting domain's interrupt handler and accompanying tables, which may be called as an interrupt mechanism in SPACE terminology, at the first stages of the interrupt pipeline. This, again, resembles SPACE's handling of interrupts as shown in figure 2 of [8], but behaves differently as to the way domains can control the interrupts they receive. That is, domains can control whether they accept, ignore, discard or terminate interrupts. Each of these has a different effect and is controlled differently.

Accepting interrupts is the normal state of a domain's interrupt mechanism. When Adeos encounters a domain who is accepting interrupts it summons its interrupt handler after having set the required CPU environment and stack content for the interrupt handler to operate correctly. The OS then may decide to operate any number of operations including task scheduling. Once the OS is done, the pipeline proceeds as planned by propagating interrupts down the pipeline. As some OSes do not recognize Adeos, their idle task is modified to call on Adeos when it becomes scheduled. This is possible since the process tables of most operating systems are accessible from a device driver or module standpoint. Taking control of the idle task is done by way of loading a device driver or a module, finding the idle task's entry in the task table and replacing it with a function that issues a special trap that Adeos recognizes as a signal from the OS in the domain that it is done using the CPU. Of course, this special interrupt isn't propagated in the pipeline and is therefore invisible to all domains. If the OS recognizes Adeos, then he only needs to tell Adeos that he is now dormant. This informs Adeos that the interrupt can proceed onto the other stages of the pipeline. Of course from one pipeline stage to the next, Adeos needs to save the state of the domain that was running so as to restore it in the next pipeline traverse.

When an OS in a domain does not want to be interrupted, for any type of reason, it asks Adeos to stall the stage its domain occupies in the interrupt pipeline. By doing so, interrupts go no further in the pipeline and are stalled at the stage occupied by the domain. When the OS is done wanting to be uninterrupted, it asks Adeos to unstall the pipeline and thereafter all the interrupts that were stalled at the corresponding stage follow their route to the other stages of the pipeline. Note that if new interrupts make their way through the pipeline while a stage is stalled they will remain at that

stage and will be provided as new interrupts to the domain occupying that stage once the stage is unstalled. As some OSes do not recognize Adeos' presence, a way must be found to communicate the usual "disable interrupt"/"enable interrupt" or "lower interrupt level"/"raise interrupt level" instructions to calls of "stall stage"/"unstall stage".

When a domain is discarding interrupts, the interrupt passes over the stage occupied by the domain and continues onto the other stages. When a domain terminates interrupts then the interrupts that are terminated by it are not propagated to latter stages. Interrupt discarding and termination is only possible when the OS in a domain recognizes Adeos. The interrupt behavior for each domain and for individual interrupts may be changed dynamically.

Since some OSes do not recognize Adeos, it is possible to create a domain which only serves as a handler for that OS. Hence, during interrupt pipelining, this stage always precedes the handled domain's stage and may take actions for that domain with Adeos in order to provide the handled domain's OS with the illusion of normal system operation.

Once Adeos is done traversing the pipeline it checks if all domains are dormant. If that is case it then calls on its idle task. This task remains active until the occurrence of the next interrupt. If all the domains aren't dormant it restores the processor to the state it had prior the interrupt entering the pipeline and execution continues where it had left.

Since Adeos is very much hardware dependent, many details pertain to one of its particular implementations. The next section covers these details for the ix86 using Linux as the base OS.

4 Implementation details

Implementing an OS from scratch is a long and risky process. There are many design philosophies available, many programming languages, many tools and other parameters that may be taken into account when wanting to design and implement an operating system starting from nothing. Implementing Adeos in such a context would require making many choices and spend more time getting the system to a functional state than actually implementing the components necessary to obtain the required behavior. Hence, the decision was made to use an unmodified already functional OS as a host for Adeos' implementation. As Linux is an OS that is widely available and for which source code is available it was chosen as the host to be used by Adeos and since Linux is well supported on the ix86 and that that architecture is widely available, ix86 was chosen as Adeos' hardware platform. That being said, the following discussion may apply to any other OS and any other platform, with the required changes to adapt to the environment composed of the chosen OS and the chosen hardware platform. In any case, a good knowledge of the hardware's operation and the OS's architecture will be fundamental since the main task of Adeos will be to take control of the hardware while the OS is running and without it sensing any difference.

In the chosen platform, Linux on an ix86, the task will be quite straightforward. In order to have the capability of even touching the hardware without Linux knowing, we need to use loadable modules. Hence, Adeos is a loadable module that is dynamically loaded and that will take control of the hardware. Taking control of the hardware on an unmodified Linux kernel on the ix86 may seem impossible since the kernel was compiled with ix86 instructions such as cli/sti which are the "disable interrupt"/ "enable interrupt" respectively on the ix86. Hence, even if we were to modify the interrupt descriptor table register using the "lidt" instruction, Linux would still be able to stop Adeos from operating in the intended fashion.

To bypass this shortcoming we use one of the ix86's own capabilities against the OS. Remember that the ix86 posses 4 privilege levels, 0 being the highest and 3 being the lowest. Linux normally operates at PL 0 with its applications running at PL 3. With these privilege levels come different capabilities. Most importantly, software running at PL 0, often called ring-zero, will be able to do what it may wish to do with the hardware. On the other end, software running at ring 3 is not permitted to do certain operations that may modify the overall system's behavior and hinder the OS's job of protecting the different processes running on the system. The privileges provided to a level or another may be controlled using the I/O privilege level (IOPL) mechanisms provided on the ix86.

An example of interest, Plex86 runs the "virtualized [1]" OS at ring 3, as explained in [6]. In the same document, K. Lawton speculates that it may be possible to virtualize an OS at ring 1 instead of ring 3. This would enable for more native execution of the different page protection mechanisms used on the ix86, hence accelerating system virtualization. In our case, virtualization is of no interest since we trust the OSes we are running on top of Adeos.

[1] offer a virtual environment in Plex86 nomenclature.

The solution therefore is to push Linux out of ring-zero and into ring-one. By doing so, all privileged instructions such as cli and sti aren't permitted and will generated a fault by the processor. These instructions will then be caught by a Linux handling domain (LHD), that precedes the Linux domain (LD) in the interrupt pipeline and that will convert these instructions into a stall/unstall LD pipeline stage. Using this technique, Linux is still in control of all the page management and Adeos doesn't necessarily need to take it away unless we plan to run another general-purpose OS side-by-side with Linux. This will be covered in section 5.

The downside to this technique is that other important instructions that we would like Linux to proceed with will be caught. The following instructions will not be permitted to Linux once it is placed in ring-one (as in the ix86 manuals provided by intel):

- clts, clear task-switched flag
- hlt, halt processor
- cli, clear interrupt flag
- sti, set interrupt flag
- in, input from port
- out, output to port
- ins, input string from port
- outs, output string from port
- invd, invalidate cache
- invlpg, invalidate
- lgdt, load GDT register
- lidt, load IDT register
- lldt, load LDT register
- lmsw, load machine status register
- ltr, load task register
- mov to/from CRn, move to control register n
- mov to/from DRn, move to debug register n
- wbinvd, writeback and invalidate cache
- rdmsr, read model-specific registers
- wrmsr, write model-specific registers
- rdpmc, read performance-monitoring counter
- rdtsc, read time-stamp counter

Most of these instructions do not execute very often. The critical ones are in/out/ins/outs as these may be heavily used by device drivers. In order not to generate faults each time these instructions are called, the task state segment bitmaps may be altered in order to permit ring-one IO. That done, device drivers will be able to operate as designed without Adeos seeing any traps for the in/out/ins/outs instructions.

Still, there remains a substantial amount of other instructions that still need to be taken care of. There are two ways to take care of those: emulation and single-stepping. With emulation, the contents of the memory and registers are modified to let Linux believe that he actually executed the instruction. Single-stepping involves setting the single-stepping bit for Linux and lowering the IOPL to 1. Hence, when Linux goes to run, he gets to run the instruction but is cutoff right afterwards and the LHD resets the IOPL back to 0 and removes the single-stepping bit. With this, Linux can continue operating at will.

There are cases that plex86 takes care of which we don't need to take care of or that we omit voluntarily to treat because we know Linux's behavior. Self-modifying code is one of these cases. Here, we assume that Linux will not attempt to modify its own binary code. This is a safe assumption given the current kernel components.

There's the case where we want additional loaded modules to have access to Adeos and we may even want some modules loaded by a ring-one Linux to become ring-zero elements. This may be provided for using a special software interrupt which traps into Adeos and which Adeos may use to modify the mappings of these modules to become part of ring-zero. This is possible since Linux's module mappings are accessible in kernel space.

This whole scheme means that Adeos itself needs to be running at ring-zero. This is achievable by adding extra entries in the GDT which can be used by Adeos and all other modules who want to become ring-zero. The inverse of this is that removing Linux from ring-zero is done by modifying the GDT entries that pertain to its code and data to have descriptor privilege levels of 1 instead of 0. Hence, the GDT register isn't modified, only the content of the global descriptor table pointed to by the GDT register is modified. The IDT register, on the other hand, is completely overwritten by Adeos to point to its own tables. Prior to overwriting it, though, Adeos keeps a pointer to Linux's initial table in order to invoke the correct entry when Linux's turn comes while going through the interrupt pipeline.

These modifications are then propagated to the rest of the elements recognized by the hardware in order to reflect the change of privilege level. This may involve changing the task-state segments maintained by Linux for every task it runs. As Linux may create new tasks while Adeos is active, he may create TSSes that are not expected by Adeos. This may be detected because of traps caused by the unexpected behavior and then fixed by the LHD.

The rest of the implementation is not hardware specific and may be implemented as described in section 3.

"Bootstrapping" Adeos on Linux/ix86 would involve loading the Adeos module first. This module's *init_module()* would not do much except return a value to inform the caller that loading succeeded. The reason Adeos can't go any further is that the Linux domain handler isn't loaded yet. It is this module's loading that will generate the chain-reaction that will result in Linux loosing control of the hardware. Once loaded, the LHD will initialize itself and call on Adeos to complete hardware hijacking.

Unloading Adeos involves unloading all modules that use it and returning Linux to ring-zero. Unloading ring-zero modules may be done using another special software interrupt recognized by Adeos. Thereafter the Adeos kernel module may be unloaded. Notice that a similar discussion may have been done with FreeBSD instead of Linux on the same hardware platform with the same end result.

5 Applicability

Having explained the architectural details of Adeos and provided details about its implementation, this section will cover some applications which can be developed under Adeos.

5.1 General-purpose operating system resource sharing

This is one of the main objectives of Adeos, to provide an environment which enables multiple general-purpose OSes to share the same hardware. As such, the implementation details provided in section 4 only explain how to install Adeos under Linux. In order to run another OS side-by-side with Linux some other implementation details will have to be covered. It remains that whichever OS gets to run with Linux it too will be trusted. In the following we will take an example where we'd like to run NetBSD beside Linux. Choosing an OS for which we do not have the source code may be a risky business though.

The first step to get an OS going is to boot it. Since this process involves probing and initializing all the hardware, we cannot permit NetBSD to boot in any form on Adeos since Linux has already booted an initialized all the hardware. Although once booted, a Linux and a NetBSD may be made to share the same hardware granted they don't reside in the same physical space and use different I/O devices. Some of the latter may be hard to share, but using appropriate domain handlers, this should be feasible. The division of physical memory may be made by instructing Linux at boot time to use only a portion of the physical memory for its operation, the rest being intended for NetBSD.

We still need to obtain a booted image of NetBSD. This may be performed in one of two ways. First, we may use plex86 to boot NetBSD in a virtualized mode and then transfer the booted OS into physical memory under control of Adeos. Second, we may boot NetBSD and then obtain an image of the running system which will thereafter be loaded beside Linux on top of Adeos. Both methods represent a fair amount of work, but the first solution is very close to K. Lawton's discussion in [6] about virtualization of OS operation at ring-one.

5.2 Operating system development

Developing OSes is usually a complicated process which sometimes requires extra hardware such as In-Circuit Emulators to probe the hardware on which the OS is running. Using Adeos, OS development is eased since any undesired behavior may be controlled by an appropriate domain handler. It would even be conceivable to provide a default domain handler for OS development under which developers may have controlled direct access to the hardware they are meant to control. As Adeos is itself a kernel-module, such development domain handlers may be developed independently from Adeos.

To start OS development under a Linux/ix86-operated Adeos, one would load a Linux kernel module which would request its own domain. Thereafter, it would have full unrestricted access to the hardware available. Since developers may wish to use a safe environment for such development, the loading of such a module may be preceded by the loading of appropriate domain handler. As development advances, it should be possible to reduce the checking of the domain handler in order to provide more control over to the developed OS and its domain. At the latest stages of OS development, the interaction between the developed OS, Adeos and the host OS

would resemble more and more closely the process described in the previous subsection to finally yield a fully functional OS.

5.3 Patch-less kernel debuggers and probers

The subject of kernel debuggers has been a very pointy one on the Linux kernel development mailing list. Each time it has been raised, it has caused a lot of tumult but no debugger ever made it into the kernel. Alleviating all these troubles, Adeos provides for a way for kernel debuggers and probers to take control of Linux without it ever being modified. As other Adeos domains, these facilities would load as normal kernel modules and would thereafter request a ring-zero domain from Adeos. Once that is done, they may request priority interrupt dispatching in the interrupt pipeline. Hence, before Linux gets to handle any interrupts, they will be able to intercept those interrupts and carry out the requested debugging tasks. This can also be extended to performance profilers and other such development tools.

6 Conclusion

In this paper we have presented the Adaptive Domain Environment for Operating Systems as a solution for sharing hardware resources amongst multiple operating systems. We have presented its architecture and suggested an implementation method for the ix86 using Linux as the base OS. We have also defined areas of applicability. Although the implementation discussion has centered around on the ix86 using Linux, the concepts presented may be extended to other architectures and other base operating systems in order to provide the same capabilities.

Given the current state of the operating system market and the research field, Adeos may be used to provide a bridge between both fields and promote the development of more flexible and cooperative operating systems. This would provide system administrators and programmers with the flexibility needed to develop user-friendly operating environments and applications that are not limited by the choice of a single operating system.

References

[1] VMWare, http://www.vmware.com/.

[2] Plex86, http://www.plex86.org/.

[3] Virtual PC, http://www.connectix.com/products/vpc.html.

[4] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proc. Symp. on Operating Systems Design and Implementation*, pages 179–194, Monterey CA (USA), 1994.

[5] D. Engler, M. Kaashoek, and J. Jr. Exokernel: an operating system architecture for application-specif ic resource management. December 1995.

[6] K. Lawton. Running multiple operating systems concurrently on an ia32 pc using virtualization techniques, November 1999. Plex86 documentation: plex86/docs/txt/paper-19991129a.txt.

[7] D. Probert and J. Bruno. Efficient cross-domain mechanisms for building kernel-less operating systems.

[8] D. Probert and J. Bruno. Building fundamentally extensible application-specific operating systems in space, March 1995.

[9] D. Probert, J. Bruno, and M. Karzaorman. Space: a new approach to operating system abstraction. In *International Workshop on Object Orientation in Operating Systems*, pages 133–137, October 1991.

[10] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM Transactions on Modeling and Computer Simulation*, volume 7, pages 78–103, January 1997.