KERNEL OPERATING SYSTEM

Manjeet Saini[1], Abhishek Jain[2], Ashish Chauhan[3]
Department Of Computer Science And Engineering, Dronacharya College Of Engineering
Khentawas, Farrukh Nagar, Gurgaon, Haryana, (India)

ABSTRACT

The central module of an operating system (OS) is the Kernel. It is the part of the operating system that loads first, and it remains in main memory. It is necessary for the kernel to be very small while still providing all the essential services needed by other parts of the OS because it stays in the memory. To prevent kernel code from being overwritten by programs or other parts of the operating system it is loaded into a protected area of memory. The presence of an operating system kernel is not a necessity to run a computer. Directly loading and executing the programs on the "bare metal" machine is possible, provided that the program authors are willing to do without any OS support or hardware abstraction. Many video game consoles and embedded systems still constitute the “bare metal” approach. But in general, newer systems use kernels and operating systems.

Keywords: Scalability, Multicore Processors, Message Passing

I. INTRODUCTION

In computing, the kernel is a computer program that manages input/output requests from software, and translates them into data processing instructions for the central processing unit and other electronic components of a computer. When a computer program (in this context called a process) makes requests of the kernel, the request is called a system call. Various kernel designs differ in how they manage system calls and resources.

For example, a monolithic kernel executes all the operating system instructions in the same address space in order to improve the performance of the system. A microkernel runs most of the operating system's background processes in user space, to make the operating system more modular and, therefore, easier to maintain. Because of its critical nature, the kernel code is usually loaded into a protected area of memory, which prevents it from being overwritten by other, less frequently used parts of the operating system or by application programs. The kernel performs its tasks, such as executing processes and handling interrupts, in kernel space, whereas everything a user normally does, such as writing text in a text editor or running programs in a GUI (graphical user interface), is done in user space. This separation is made in order to prevent user data and kernel data from interfering with each other and thereby diminishing performance or causing the system to become unstable (and possibly crashing).

II. FUNCTIONS OF KERNEL

The kernel's primary function is to mediate access to the computer's resources, including:

This central component of a computer system is responsible for running or executing programs. The kernel takes responsibility for deciding at any time which of the many running programs should be allocated to the processor or processors.

Random-access memory is used to store both program instructions and data. Typically, both need to be present in memory in order for a program to execute. Often multiple programs will want access to memory, frequently demanding more memory than the computer has available. The kernel is responsible for deciding which memory each process can use, and determining what to do when not enough is available. I/O devices include such peripherals as keyboards, mice, disk drives, printers, network adapters, and display devices. The kernel allocates requests from applications to perform I/O to an appropriate device and provides convenient methods for using the device (typically abstracted to the point where the application does not need to know implementation details of the device). Kernels also usually provide methods for synchronization and communication between processes called inter-process communication (IPC). The kernel has full access to the system's memory and must allow processes to safely access this memory as they require it. Often the first step in doing this is virtual addressing, usually achieved by paging and/or segmentation. Virtual addressing allows the kernel to make a given physical address appear to be another address, the virtual address. Virtual address spaces may be different for different processes; the memory that one process accesses at a particular (virtual) address may be different memory from what another process accesses at the same address. This allows every program to behave as if it is the only one (apart from the kernel) running and thus prevents applications from crashing each other. On many systems, a program's virtual address may refer to data which is not currently in memory. The layer of indirection provided by virtual addressing allows the operating system to use other data stores, like a hard drive, to store what would otherwise have to remain in main memory (RAM). As a result, operating systems can allow programs to use more memory than the system has physically available. When a program needs data which is not currently in RAM, the CPU signals to the kernel that this has happened, and the kernel responds by writing the contents of an inactive memory block to disk (if necessary) and replacing it with the data requested by the program. The program can then be resumed from the point where it was stopped. This scheme is generally known as demand paging. Virtual addressing also allows creation of virtual partitions of memory in two disjointed areas, one being reserved for the kernel (kernel space) and the other for the applications (user space). The applications are not permitted by the processor to address kernel memory, thus preventing an application from damaging the running kernel. This fundamental partition of memory space has contributed much for current designs of actual general-purpose kernels and is almost universal in such systems, although some research kernels (e.g. Singularity) take other approaches. To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers. A device driver is a computer program that enables the operating system to interact with a hardware device. It provides the operating system with information of how to control and communicate with a certain piece of hardware. The driver is an important and vital piece to a program application. The design goal of a driver is abstraction; the function of the driver is to translate the OS-mandated function calls (programming calls) into device-specific calls. In theory, the device should work correctly with the suitable driver. Device drivers are used for such things as video cards, sound cards, printers, scanners, modems, and LAN cards.

The common levels of abstraction of device drivers are:

1. **On The Hardware Side**:

- Interfacing directly.
- Using a lower-level device driver (file drivers using disk drivers).
- Using a high level interface (Video BIOS).
- Simulating work with hardware, while doing something entirely different.

2. On The Software Side:

- Allowing the operating system direct access to hardware resources.
- Implementing only primitives.
- Implementing an interface for non-driver software (Example: TWAIN).
- Implementing a language, sometimes high-level (Example PostScript).

For Example, to show the user something on the screen, an application would make a request to the kernel, which would forward the request to its display driver, which is then responsible for actually plotting the character/pixel. A kernel must maintain a list of available devices. This list may be known in advance (e.g. on an embedded system where the kernel will be rewritten if the available hardware changes), configured by the user (typical on older PCs and on systems that are not designed for personal use) or detected by the operating system at run time (normally called plug and play). In a plug and play system, a device manager first performs a scan on different hardware buses, such as Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB), to detect installed devices, then searches for the appropriate drivers. As device management is a very OS-specific topic, these drivers are handled differently by each kind of kernel design, but in every case, the kernel has to provide the I/O to allow drivers to physically access their devices through some port or memory location. Very important decisions have to be made when designing the device management system, as in some designs accesses may involve context switches, making the operation very CPU-intensive and easily causing a significant performance overhead.

1. In computing, a system call is how a program requests a service from an operating system's kernel that it does not normally have permission to run. System calls provide the interface between a process and the operating system. Most operations interacting with the system require permissions not available to a user level process, e.g. I/O performed with a device present on the system, or any form of communication with other processes requires the use of system calls. A system call is a mechanism that is used by the application program to request a service from the operating system. They use a machine-code instruction that causes the processor to change mode. An example would be from supervisor mode to protected mode. This is where the operating system performs actions like accessing hardware devices or the memory management unit. Generally the operating system provides a library that sits between the operating system and normal programs. Usually it is a C library such as Glibc or Windows API. The library handles the low-level details of passing information to the kernel and switching to supervisor mode. System calls include close, open, read, wait and write. To actually perform useful work, a process must be able to access the services provided by the kernel. This is implemented differently by each kernel, but most provide a C library or an API, which in turn invokes the related kernel functions. The method of invoking the kernel function varies from kernel to kernel. If memory isolation is in use, it is impossible for a user process to call the kernel directly, because that would be a violation of the processor's access control rules.

A few possibilities are:

- Using a software-simulated interrupt. This method is available on most hardware, and is therefore very common.
- Using a call gate. A call gate is a special address stored by the kernel in a list in kernel memory at a location known to the processor. When the processor detects a call to that address, it instead redirects to the target location without causing an access violation. This requires hardware support, but the hardware for it is quite common.

- Using a special system call instruction. This technique requires special hardware support, which common architectures (notably, x86) may lack. System call instructions have been added to recent models of x86 processors, however, and some operating systems for PCs make use of them when available.
- Using a memory-based queue. An application that makes large numbers of requests but does not need to wait for the result of each may add details of requests to an area of memory that the kernel periodically scans to find requests.

III. TYPES OF KERNELS

Naturally, the above listed tasks and features can be provided in many ways that differ from each- other in design and implementation. The principle of separation of mechanism and policy is the substantial difference between the philosophy of micro and monolithic kernels. Here a mechanism is the support that allows the implementation of many different policies, while a policy is a particular "mode of operation". For instance, a mechanism may provide for user log-in attempts to call an authorization server to determine whether access should be granted; a policy may be for the authorization server to request a password and check it against an encrypted password stored in a database. Because the mechanism is generic, the policy could more easily be changed (e.g. by requiring the use of a security token) than if the mechanism and policy were integrated in the same module. In minimal microkernel just some very basic policies are included, and its mechanisms allows what is running on top of the kernel (the remaining part of the operating system and the other applications) to decide which policies to adopt (as memory management, high level process scheduling, file system management, etc.). A monolithic kernel instead tends to include many policies, therefore restricting the rest of the system to rely on them. Mr Per Brinch Hansen presented arguments in favor of separation of mechanism and policy. The failure to properly fulfill this separation is one of the major causes of the lack of substantial innovation in existing operating systems a problem common in computer architecture. The monolithic design is induced by the "kernel mode"/"user mode" architectural approach to protection (technically called hierarchical protection domains), which is common in conventional commercial systems; in fact, every module needing protection is therefore preferably included into the kernel. This link between monolithic design and "privileged mode" can be re-conducted to the key issue of mechanism-policy separation; in fact the "privileged mode" architectural approach melts together the protection mechanism with the security policies, while the major alternative architectural approach, capability-based addressing, clearly distinguishes between the two, leading naturally to a microkernel design (see Separation of protection and security). While monolithic kernels execute all of their code in the same address space (kernel space) microkernels try to run most of their services in user space, aiming to improve maintainability and modularity of the code base. Most kernels do not fit exactly into one of these categories, but are rather found in between these two designs. These are called hybrid kernels. More exotic designs such as Nano and Exo-kernels are available, but are seldom used for production systems. The Xen hypervisor, for example, is a Exo-kernel.

3.1 Monolithic Kernels

In a monolithic kernel, all OS services run along with the main kernel thread, thus also residing in the same memory area. This approach provides rich and powerful hardware access. The main disadvantages of monolithic kernels are the dependencies between system components — a bug in a device driver might crash the entire system — and the fact that large kernels can become very difficult to maintain.

Monolithic kernels, which have traditionally been used by Unix-like operating systems, contain all the operating system core functions and the device drivers (small programs that allow the operating system to interact with hardware devices, such as disk drives, video cards and printers). A monolithic kernel is one single program that contains all of the code necessary to perform every kernel related task. Every part which is to be accessed by most programs which cannot be put in a library is in the kernel space: Device drivers, Scheduler, Memory handling, File systems, Network stacks. Many system calls are provided to applications, to allow them to access all those services. A monolithic kernel, while initially loaded with subsystems that may not be needed can be tuned to a point where it is as fast as or faster than the one that was specifically designed for the hardware, although more in a general sense. Modern monolithic kernels, such as those of Linux and FreeBSD, both of which fall into the category of Unix-like operating systems, feature the ability to load modules at runtime, thereby allowing easy extension of the kernel's capabilities as required, while helping to minimize the amount of code running in kernel space. In the monolithic kernel, some advantages hinge on these points:

- Since there is less software involved it is faster.
- As it is one single piece of software it should be smaller both in source and compiled forms.
- Less code generally means fewer bugs which can translate to fewer security problems.

This design has several flaws and limitations:

Coding in kernel can be challenging, in part because one cannot use common libraries (like a full-featured libc), and because one needs to use a source-level debugger like gdb. Rebooting the computer is often required. This is not just a problem of convenience to the developers. When debugging is harder, and as difficulties become stronger, it becomes more likely that code will be "buggier". Bugs in one part of the kernel have strong side effects; since every function in the kernel has all the privileges, a bug in one function can corrupt data structure of another, totally unrelated part of the kernel, or of any running program. Kernels often become very large and difficult to maintain. Even if the modules servicing these operations are separate from the whole, the code integration is tight and difficult to do correctly. Since the modules run in the same address space, a bug can bring down the entire system. Monolithic kernels are not portable; therefore, they must be rewritten for an each new architecture that the operating system is to be used on. In the microkernel approach, the kernel itself only provides basic functionality that allows the execution of servers, separate programs that assume former kernel functions, such as device drivers, GUI servers, etc.

3.2 Microkernels

Microkernel) is the term describing an approach to Operating System design by which the functionality of the system is moved out of the traditional "kernel", into a set of "servers" that communicate through a "minimal" kernel, leaving as little as possible in "system space" and as much as possible in "user space". A microkernel that is designed for a specific platform or device is only ever going to have what it needs to operate. The microkernel approach consists of defining a simple abstraction over the hardware, with a set of primitives or system calls to implement minimal OS services such as memory management, multitasking, and inter-process communication. Other services, including those normally provided by the kernel, such as networking, are implemented in user-space programs, referred to as servers. Microkernels are easier to maintain than monolithic kernels, but the large number of system calls and context switches might slow down the system because they typically generate more overhead than plain function calls. Only parts which really require being in a privileged mode are in kernel space: IPC (Inter-Process Communication), Basic scheduler, or scheduling primitives, Basic memory handling, Basic I/O primitives. Many critical parts are now running in user space: The complete scheduler, Memory handling, File systems, and Network stacks. Micro kernels were invented as a reaction to traditional "monolithic" kernel design, whereby all system functionality was put in a one static program running in a special "system" mode of the processor. In the microkernel, only the most fundamental of tasks are performed such as being able to access some (not necessarily all) of the hardware, manage memory and coordinate message passing between the processes. Some systems that use micro kernels are QNX and the HURD. In the case of QNX and Hurd user sessions can be entire snapshots of the system itself or views as it is referred to. The very essence of the microkernel architecture illustrates some of its advantages:

- Maintenance is generally easier.
- Patches can be tested in a separate instance, and then swapped in to take over a production instance.
- Rapid development time and new software can be tested without having to reboot the kernel.
- More persistence in general, if one instance goes hay-wire, it is often possible to substitute it with an operational mirror.

Most micro kernels use a message passing system of some sort to handle requests from one server to another. The message passing system generally operates on a port basis with the microkernel. As an example, if a request for more memory is sent, a port is opened with the microkernel and the request sent through. Once within the microkernel, the steps are similar to system calls. The rationale was that it would bring modularity in the system architecture, which would entail a cleaner system, easier to debug or dynamically modify, customizable to users' needs, and more performing. Other services provided by the kernel such as networking are implemented in user-space programs referred to as servers. Servers allow the operating system to be modified by simply starting and stopping programs. For a machine without networking support, for instance, the networking server is not started. The task of moving in and out of the kernel to move data between the various applications and servers creates overhead which is detrimental to the efficiency of micro kernels in comparison with monolithic kernels.

Disadvantages in the microkernel exist however. Some are:

- Larger running memory footprint
- More software for interfacing is required, there is a potential for performance loss.
- Messaging bugs can be harder to fix due to the longer trip that they have to take versus the one off copy in a monolithic kernel.
- Process management in general can be very complicated.
- The disadvantages for micro kernels are extremely context based. As an example, they work well for small single purpose (and critical) systems because if not many processes need to run, then the complications of process management are effectively mitigated.

So, what the bare minimum that MicroKernel architecture recommends in kernel space?

- Managing memory protection
- Process scheduling
- Inter Process communication (IPC)

A microkernel allows the implementation of the remaining part of the operating system as a normal application program written in a high-level language, and the use of different operating systems on top of the same unchanged kernel. It is also possible to dynamically switch among operating systems and to have more than one active simultaneously.

3.3 Hybrid (Or Modular) Kernel

These kernels represent a compromise that was implemented by some developers before it was demonstrated that pure micro kernels can provide high performance. These types of kernels are extensions of micro kernels with some properties of monolithic kernels. Unlike monolithic kernels, these types of kernels are unable to load modules at runtime on their own. Hybrid kernels are micro kernels that have some "non-essential" code in kernel-space in order for the code to run more quickly than it would were it to be in user-space. Hybrid kernels are a compromise between the monolithic and microkernel designs. This implies running some services (such as the network stack or the file system) in kernel space to reduce the performance overhead of a traditional microkernel, but still running kernel code (such as device drivers) as servers in user space. Hybrid kernels are used in most commercial operating systems such as Microsoft Windows NT 3.1, NT 3.5, NT 3.51, NT 4.0, 2000, XP, Vista, 7, 8, and 8.1 . Apple Inc's own Mac OS X uses a hybrid kernel called XNU which is based upon code from Carnegie Mellon's Mach kernel and FreeBSD's monolithic kernel. They are similar to micro kernels, except they include some additional code in kernel-space to increase performance. Many traditionally monolithic kernels are now at least adding (if not actively exploiting) the module capability. The most well-known from these kernels is the Linux kernel. The modular kernel essentially can have parts of it that are built into the core kernel binary or binaries that load into memory on demand. It is important to note that a code tainted module has the potential to destabilize a running kernel. Many people become confused on this point when discussing micro kernels. It is possible to write a driver for a microkernel in a completely separate memory space and test it before "going" live. When a kernel module is loaded, it accesses the monolithic portion's memory space by adding to it what it needs, therefore, opening the doorway to possible pollution. A few advantages to the modular (or) Hybrid kernel are:

- Faster development time for drivers that can operate from within modules. No reboot required for testing (provided the kernel is not destabilized).
- On demand capability versus spending time recompiling a whole kernel for things like new drivers or subsystems.

Modules, generally, communicate with the kernel using a module interface of some sort. The interface is generalized (although particular to a given operating system) so it is not always possible to use modules. Often the device drivers may need more flexibility than the module interface affords. Essentially, it is two system calls and often the safety checks that only have to be done once in the monolithic kernel now may be done twice. Some of the disadvantages of the modular approach are:

- With more interfaces to pass through, the possibility of increased bugs exists (which implies more security holes).
- Maintaining modules can be confusing for some administrators when dealing with problems like symbol differences.

1. Nano-kernels:-

Nano-kernel delegates virtually all services — including even the most basic ones like interrupt controllers or the timer — to device drivers to make the kernel memory requirement even smaller than a traditional microkernel.

2. Exo-kernels:-

Exo-kernels are a still experimental approach to operating system design. They differ from the other types of kernels in that their functionality is limited to the protection and multiplexing of the raw hardware, providing no hardware abstractions on top of which to develop applications. This separation of hardware protection from hardware management enables application developers to determine how to make the most efficient use of the available hardware for each specific program. Exo-kernels in themselves are extremely small. However, they are accompanied by library operating systems, providing application developers with the functionalities of a conventional operating system. A major advantage of Exo-kernel-based systems is that they can incorporate multiple library operating systems, each exporting a different API, for example one for high level UI development and one for real-time control.

IV. MONOLITHIC KERNELS VS. MICROKERNELS

As the computer kernel grows, a number of problems become evident. One of the most obvious is that the memory footprint increases. This is mitigated to some degree by perfecting the virtual memory system, but not all computer architectures have virtual memory support. To reduce the kernel's footprint, extensive editing has to be performed to carefully remove unneeded code, which can be very difficult with non-obvious interdependencies between parts of a kernel with millions of lines of code. Due to the various shortcomings of monolithic kernels versus microkernels, monolithic kernels were considered obsolete by virtually all operating system researchers. As a result, the design of Linux as a monolithic kernel rather than a microkernel was the topic of a famous debate between Linus Torvalds and Andrew Tanenbaum. There is merit on both sides of the argument presented in the Tanenbaum–Torvalds debate.

V. PERFORMANCE

Monolithic kernels are designed to have all of their code in the same address space (kernel space), which some developers argue is necessary to increase the performance of the system. Some developers also maintain that monolithic systems are extremely efficient if well-written. The monolithic model tends to be more efficient through the use of shared kernel memory, rather than the slower IPC system of microkernel designs, which is typically based on message passing. Studies that empirically measured the performance of these microkernels did not analyze the reasons of such inefficiency. The explanations of this data were left to "folklore", with the assumption that they were due to the increased frequency of switches from "kernel-mode" to "user-mode" to the increased frequency of inter-process communication and to the increased frequency of context switches.

The reasons for the poor performance of microkernels might as well have been:

(1) An actual inefficiency of the whole microkernel approach,

(2) The particular concepts implemented in those micro-kernels, and

(3) The particular implementation of those concepts. Therefore it remained to be studied if the solution to build an efficient microkernel was, unlike previous attempts, to apply the correct construction techniques. On the other end, the hierarchical protection domains architecture that leads to the design of a monolithic kernel has a significant performance drawback each time there's an interaction between different levels of protection (i.e. when a process has to manipulate a data structure both in 'user mode' and 'supervisor mode'), since this requires message copying by value. The hybrid kernel approach combines the speed and simpler design of a monolithic kernel with the modularity and execution safety of a microkernel.

VI. CONCLUSION

A monolithic OS kernel is faster due to small source and compiled code size. Less code means also less bugs and security issues. Bug fixing or adding new features requires the compilation of the whole source code which is a time and resource consuming process. The idea behind microkernel OS is to reduce the kernel to only basic process communication and IO control and let other system services run in user space just like any other normal processes. Easy and faster integration with 3d party modules is an advantage of Microkernel OS. The conclusion is no OS architecture is better than the other in the general sense because every monolithic operating system meets different needs.

VII. FUTURE SCOPE

The increased resemblance of multi core machines with complex networked systems can prove to be helpful in proposing the multi kernel architecture as a way forward. Current OS structure is tuned for a coherent shared memory with a limited number of homogeneous processors, and is not made for efficiently managing the diversity and scale of future hardware architectures. Viewing the OS as a distributed system rather than centralized system can help in scaling to a network like environment of a modern or future machine.

REFERENCES

[1] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R.Isaacs. Your computer is already a distributed system. Why isn't your OS? In Proceedings of the 12th Workshop on Hot Topics in Operating Systems, May 2009.

[2] S. Borkar. Thousand core chips: a technology perspective. In Proceedings of the 44th Annual Design Automation Conference, pages 746–749, 2007.

[3] P. Charles, C. Grotho_, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Proceedings of the ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications, pages 519–538, 2005.

[4] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. IEEE Micro, 26(3):10–23, 2006.

[5] SQLite database engine. http://www.sqlite.org/.