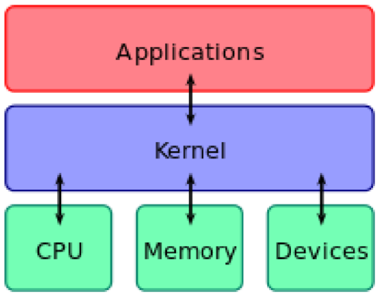
WIKIPEDIA

Kernel (operating system)

The **kernel** is a computer program that is the core of a computer's operating system, with complete control over everything in the system.[1] On most systems, it is one of the first programs loaded on start-up (after the bootloader). It handles the rest of start-up as well as input/output requests from software, translating them into data-processing instructions for the central processing unit. It handles memory and peripherals like keyboards, monitors, printers, and speakers.

The critical code of the kernel is usually loaded into a protected area of memory, which prevents it from being overwritten by applications or other, more minor parts of the operating system. The kernel performs its tasks, such as running processes and handling interrupts, in kernel space. In contrast, everything a user does is in user space: writing text in a text editor, running programs in a GUI, etc. This separation prevents user data and kernel data from interfering with each other and causing instability and slowness.[1]



A kernel connects the application software to the hardware of a computer.

The kernel's interface is a low-level abstraction layer. When a process makes requests of the kernel, it is called a system call. Kernel designs differ in how they manage these system calls and resources. A monolithic kernel runs all the operating system instructions in the same address space for speed. A microkernel runs most processes in user space,[2] for modularity.[3]

Contents

Functions

The kernel's primary function is to mediate access to the computer's resources, including:[4]

The central processing unit (CPU)
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 (each of which can usually run only one program at a time).

Random-access memory (RAM)
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 memory is available.

Input/output (I/O) devices
I/O devices include such peripherals as keyboards, mice, disk drives, printers, USB devices, 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).

Resource Management

Key aspects necessary in resource management are the definition of an execution domain(address space) and the protection mechanism used to mediate access to the resources within a domain.[4] Kernels also provide methods for synchronization and inter-process communication (IPC). These implementations may be within the Kernel itself or the kernel can also rely on other processes it is running. Although the kernel must provide inter-process communication in order to provide access to the facilities provided by each other. Kernels must also provide running programs with a method to make requests to access these facilities.

Memory management

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.[5]

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 to the current designs of actual general-purpose kernels and is almost universal in such systems, although some research kernels (e.g. Singularity) take other approaches.

Device management

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 high level interface (Video BIOS).
- Using a lower-level device driver (file drivers using disk drivers).
- 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.[5]

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.

System calls

In computing, a system call is how a process 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.[6]

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.

Kernel design decisions

Issues of kernel support for protection

An important consideration in the design of a kernel is the support it provides for protection from faults (fault tolerance) and from malicious behaviours (security). These two aspects are usually not clearly distinguished, and the adoption of this distinction in the kernel design leads to the rejection of a hierarchical structure for protection.[4]

The mechanisms or policies provided by the kernel can be classified according to several criteria, including: static (enforced at compile time) or dynamic (enforced at run time); pre-emptive or post-detection; according to the protection principles they satisfy (e.g. Denning[7][8]); whether they are hardware supported or language based; whether they are more an open mechanism or a binding policy; and many more.

Support for hierarchical protection domains[9] is typically implemented using CPU modes.

Many kernels provide implementation of "capabilities", i.e. objects that are provided to user code which allow limited access to an underlying object managed by the kernel. A common example occurs in file handling: a file is a representation of information stored on a permanent storage device. The kernel may be able to perform many different operations (e.g. read, write, delete or execute the file contents) but a user level application may only be permitted to perform some of these operations (e.g. it may only be allowed to read the file). A common implementation of this is for the kernel to provide an object to the application (typically called a "file handle") which the application may then invoke operations on, the validity of which the kernel checks at the time the operation is requested. Such a system may be extended to cover all objects that the kernel manages, and indeed to objects provided by other user applications.

An efficient and simple way to provide hardware support of capabilities is to delegate the MMU the responsibility of checking access-rights for every memory access, a mechanism called capability-based addressing.[10] Most commercial computer architectures lack such MMU support for capabilities.

An alternative approach is to simulate capabilities using commonly supported hierarchical domains; in this approach, each protected object must reside in an address space that the application does not have access to; the kernel also maintains a list of capabilities in such memory. When an application needs to access an object protected by a capability, it performs a system call and the kernel then checks whether the application's capability grants it permission to perform the requested action, and if it is permitted performs the access for it (either directly, or by delegating the request to another user-level process). The performance cost of address space switching limits the practicality of this approach in systems with complex interactions between objects, but it is used in current operating systems for objects that are not accessed frequently or which are not expected to perform quickly.[11][12] Approaches where protection mechanism are not firmware supported but are instead simulated at higher levels (e.g. simulating capabilities by manipulating page tables on hardware that does not have direct support), are possible, but there are performance implications.[13] Lack of hardware support may not be an issue, however, for systems that choose to use language-based protection.[14]

An important kernel design decision is the choice of the abstraction levels where the security mechanisms and policies should be implemented. Kernel security mechanisms play a critical role in supporting security at higher levels.[10][15][16][17][18]

One approach is to use firmware and kernel support for fault tolerance (see above), and build the security policy for malicious behavior on top of that (adding features such as cryptography mechanisms where necessary), delegating some responsibility to the compiler. Approaches that delegate enforcement of security policy to the compiler and/or the application level are often called *language-based security*.

The lack of many critical security mechanisms in current mainstream operating systems impedes the implementation of adequate security policies at the application abstraction level.[15] In fact, a common misconception in computer security is that any security policy can be implemented in an application regardless of kernel support.[15]

Hardware-based or language-based protection

Typical computer systems today use hardware-enforced rules about what programs are allowed to access what data. The processor monitors the execution and stops a program that violates a rule (e.g., a user process that is about to read or write to kernel memory, and so on). In systems that lack support for capabilities, processes are isolated from each other by using separate address spaces.[19] Calls from user processes into the kernel are regulated by requiring them to use one of the above-described system call methods.

An alternative approach is to use language-based protection. In a language-based protection system, the kernel will only allow code to execute that has been produced by a trusted language compiler. The language may then be designed such that it is impossible for the programmer to instruct it to do something that will violate a security requirement.[14]

Advantages of this approach include:

- No need for separate address spaces. Switching between address spaces is a slow operation that causes a great deal of overhead, and a lot of optimization work is currently performed in order to prevent unnecessary switches in current operating systems. Switching is completely unnecessary in a language-based protection system, as all code can safely operate in the same address space.
- Flexibility. Any protection scheme that can be designed to be expressed via a programming language can be implemented using this method. Changes to the protection scheme (e.g. from a hierarchical system to a capability-based one) do not require new hardware.

Disadvantages include:

- Longer application start up time. Applications must be verified when they are started to ensure they have been compiled by the correct compiler, or may need recompiling either from source code or from bytecode.
- Inflexible type systems. On traditional systems, applications frequently perform operations that are not type safe. Such operations cannot be permitted in a language-based protection system, which means that applications may need to be rewritten and may, in some cases, lose performance.

Examples of systems with language-based protection include JX and Microsoft's Singularity.

Process cooperation

Edsger Dijkstra proved that from a logical point of view, atomic lock and unlock operations operating on binary semaphores are sufficient primitives to express any functionality of process cooperation.[20] However this approach is generally held to be lacking in terms of safety and efficiency, whereas a message passing approach is more flexible.[21] A number of other approaches (either lower- or higher-level) are available as well, with many modern kernels providing support for systems such as shared memory and remote procedure calls.

I/O devices management

The idea of a kernel where I/O devices are handled uniformly with other processes, as parallel co-operating processes, was first proposed and implemented by Brinch Hansen (although similar ideas were suggested in 1967[22][23]). In Hansen's description of this, the "common" processes are called *internal processes*, while the I/O devices are called *external processes*.[21]

Similar to physical memory, allowing applications direct access to controller ports and registers can cause the controller to malfunction, or system to crash. With this, depending on the complexity of the device, some devices can get surprisingly complex to program, and use several different controllers. Because of this, providing a more abstract interface to manage the device is important. This interface is normally done by a Device Driver or Hardware Abstraction Layer. Frequently, applications will require access to these devices. The Kernel must maintain the list of these devices by querying the system for them in some way. This can be done through the BIOS, or through one of the various system buses (such as PCI/PCIE, or USB). When an application requests an operation on a device (Such as displaying a character), the kernel needs to send this request to the current active video driver. The video driver, in turn, needs to carry out this request. This is an example of Inter Process Communication (IPC).

Kernel-wide design approaches

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.[24][25] 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,[25] 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.).[4][21] A monolithic kernel instead tends to include many policies, therefore restricting the rest of the system to rely on them.

Per Brinch Hansen presented arguments in favour of separation of mechanism and policy.[4][21] The failure to properly fulfill this separation is one of the major causes of the lack of substantial innovation in existing operating systems,[4] a problem common in computer architecture.[26][27][28] 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;[29] in fact, every module needing protection is therefore preferably included into the kernel.[29] This link between monolithic design and "privileged mode" can be reconducted to the key issue of mechanism-policy separation;[4] 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[4] (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 codebase.[30] 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 nanokernels and exokernels are available, but are seldom used for production systems. The Xen hypervisor, for example, is an exokernel.

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. Some developers, such as UNIX developer Ken Thompson, maintain that it is "easier to implement a monolithic kernel"[31] than microkernels. 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. This is the traditional design of UNIX systems. 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 relevant 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:

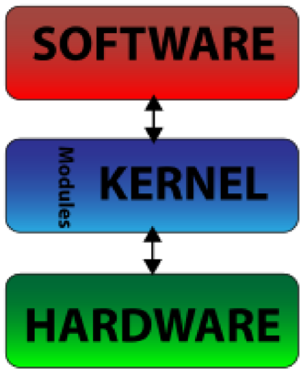


Diagram of a monolithic kernel

- 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.

Most work in the monolithic kernel is done via system calls. These are interfaces, usually kept in a tabular structure, that access some subsystem within the kernel such as disk operations. Essentially calls are made within programs and a checked copy of the request is passed through the system call. Hence, not far to travel at all. The monolithic Linux kernel can be made extremely small not only because of its ability to dynamically load modules but also because of its ease of customization. In fact, there are some versions that are small enough to fit together with a large number of utilities and other programs on a single floppy disk and still provide a fully functional operating system (one of the most popular of which is muLinux). This ability to miniaturize its kernel has also led to a rapid growth in the use of Linux in embedded systems.

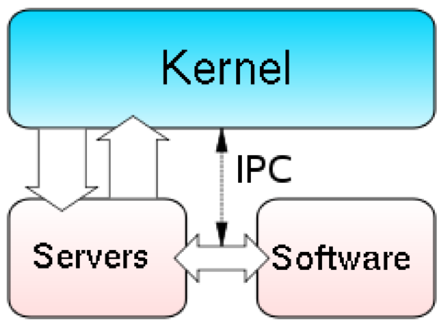
These types of kernels consist of the core functions of the operating system and the device drivers with the ability to load modules at runtime. They provide rich and powerful abstractions of the underlying hardware. They provide a small set of simple hardware abstractions and use applications called servers to provide more functionality. This particular approach defines a high-level virtual interface over the hardware, with a set of system calls to implement operating system services such as process management, concurrency and memory management in several modules that run in supervisor mode. 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 each new architecture that the operating system is to be used on.

Microkernels

Microkernel (also abbreviated μK or uK) 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:



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.

- 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. They are part of the operating systems like AIX, BeOS, Hurd, Mach, macOS, MINIX, QNX. Etc. Although micro kernels are very small by themselves, in combination with all their required auxiliary code they are, in fact, often larger than monolithic kernels. Advocates of monolithic kernels also point out that the two-tiered structure of microkernel systems, in which most of the operating system does not interact directly with the hardware, creates a not-insignificant cost in terms of system efficiency. These types of kernels normally provide only the minimal services such as defining memory address spaces, Inter-process communication (IPC) and the process management. The other functions such as running the hardware processes are not handled directly by micro kernels. Proponents of micro kernels point out those monolithic kernels have the disadvantage that an error in the kernel can cause the entire system to crash. However, with a microkernel, if a kernel process crashes, it is still possible to prevent a crash of the system as a whole by merely restarting the service that caused the error.

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 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.

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.[21] It is also possible to dynamically switch among operating systems and to have more than one active simultaneously.[21]

Monolithic kernels vs. microkernels

As the computer kernel grows, so grows the size and vulnerability of its trusted computing base; and, besides reducing security, there is the problem of enlarging the memory footprint. This is mitigated to some degree by perfecting the virtual memory system, but not all computer architectures have virtual memory support.[32] 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.

By the early 1990s, 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.[33] There is merit on both sides of the argument presented in the Tanenbaum–Torvalds debate.

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.[34] Some developers also maintain that monolithic systems are extremely efficient if well written.[34] The monolithic model tends to be more efficient[35] through the use of shared kernel memory, rather than the slower IPC system of microkernel designs, which is typically based on message passing.

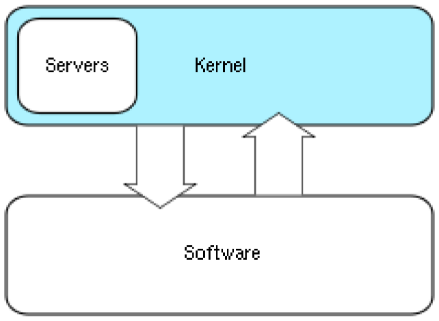
The performance of microkernels was poor in both the 1980s and early 1990s.[36][37] However, studies that empirically measured the performance of these microkernels did not analyze the reasons of such inefficiency.[36] The explanations of this data were left to "folklore", with the assumption that they were due to the increased frequency of switches from "kernel-mode" to "user-mode",[36] to the increased frequency of inter-process communication[36] and to the increased frequency of context switches.[36]

In fact, as guessed in 1995, 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 microkernels, and (3) the particular *implementation* of those concepts.[36] Therefore it remained to be studied if the solution to build an efficient microkernel was, unlike previous attempts, to apply the correct construction techniques.[36]

On the other end, the hierarchical protection domains architecture that leads to the design of a monolithic kernel[29] 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.[38]

By the mid-1990s, most researchers had abandoned the belief that careful tuning could reduce this overhead dramatically, but recently, newer microkernels, optimized for performance, such as L4[39] and K42 have addressed these problems.

Hybrid (or modular) kernels



The hybrid kernel approach combines the speed and simpler design of a monolithic kernel with the modularity and execution safety of a microkernel.

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, 8.1 and 10. Apple Inc's own macOS uses a hybrid kernel called XNU which is based upon code from OSF/1's Mach kernel (OSFMK 7.3)[40] and FreeBSD's monolithic kernel. They are similar to micro kernels, except they include some additional code in kernel-space to increase performance. 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 filesystem) 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.

Many traditionally monolithic kernels are now at least adding (if not actively exploiting) the module capability. The most well known of 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.
- Faster integration of third party technology (related to development but pertinent unto itself nonetheless).

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.

Nanokernels

A nanokernel 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.[41]

Exokernels

Exokernels 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.

Exokernels in themselves are extremely small. However, they are accompanied by library operating systems (see also unikernel), providing application developers with the functionalities of a conventional operating system. A major advantage of exokernel-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.

History of kernel development

Early operating system kernels

Strictly speaking, an operating system (and thus, a kernel) is not *required* to run a computer. Programs can be directly loaded and executed on the "bare metal" machine, provided that the authors of those programs are willing to work without any hardware abstraction or operating system support. Most early computers operated this way during the 1950s and early 1960s, which were reset and reloaded between the execution of different programs. Eventually, small ancillary programs such as program loaders and debuggers were left in memory between runs, or loaded from ROM. As these were developed, they formed the basis of what became early operating system kernels. The "bare metal" approach is still used today on some video game consoles and embedded systems,[42] but in general, newer computers use modern operating systems and kernels.

In 1969, the RC 4000 Multiprogramming System introduced the system design philosophy of a small nucleus "upon which operating systems for different purposes could be built in an orderly manner",[43] what would be called the microkernel approach.

Time-sharing operating systems

In the decade preceding Unix, computers had grown enormously in power – to the point where computer operators were looking for new ways to get people to use their spare time on their machines. One of the major developments during this era was time-sharing, whereby a number of users would get small slices of computer time, at a rate at which it appeared they were each connected to their own, slower, machine.[44]

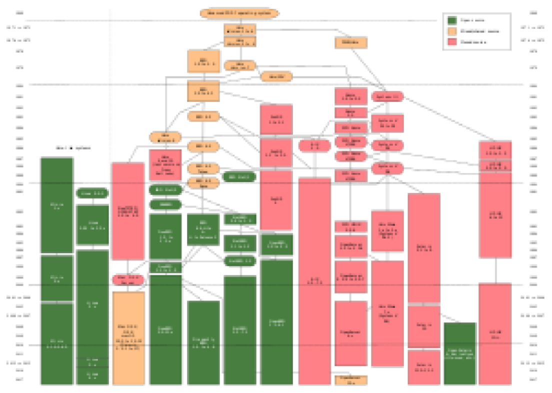
The development of time-sharing systems led to a number of problems. One was that users, particularly at universities where the systems were being developed, seemed to want to hack the system to get more CPU time. For this reason, security and access control became a major focus of the Multics project in 1965.[45] Another ongoing issue was properly handling computing resources: users spent most of their time staring at the terminal and thinking about what to input instead of actually using the resources of the computer, and a time-sharing system should give the CPU time to an active user during these periods. Finally, the systems typically offered a memory hierarchy several layers deep, and partitioning this expensive resource led to major developments in virtual memory systems.

Amiga

The Commodore Amiga was released in 1985, and was among the first – and certainly most successful – home computers to feature an advanced kernel architecture. The AmigaOS kernel's executive component, *exec.library*, uses a microkernel message-passing design, but there are other kernel components, like *graphics.library*, that have direct access to the hardware. There is no memory protection, and the kernel is almost always running in user mode. Only special actions are executed in kernel mode, and user-mode applications can ask the operating system to execute their code in kernel mode.

Unix

During the design phase of Unix, programmers decided to model every high-level device as a file, because they believed the purpose of computation was data transformation.[46]


A diagram of the predecessor/successor family relationship for Unix-like systems

For instance, printers were represented as a "file" at a known location – when data was copied to the file, it printed out. Other systems, to provide a similar functionality, tended to virtualize devices at a lower level – that is, both devices *and* files would be instances of some lower level concept. Virtualizing the system at the file level allowed users to manipulate the entire system using their existing file management utilities and concepts, dramatically simplifying operation. As an extension of the same paradigm, Unix allows programmers to manipulate files using a series of small programs, using the concept of pipes, which allowed users to complete operations in stages, feeding a file through a chain of single-purpose tools. Although the end result was the same, using smaller programs in this way dramatically increased flexibility as well as ease of development and use, allowing the user to modify their workflow by adding or removing a program from the chain.

In the Unix model, the *operating system* consists of two parts; first, the huge collection of utility programs that drive most operations, the other the kernel that runs the programs.[46] Under Unix, from a programming standpoint, the distinction between the two is fairly thin; the kernel is a program, running in supervisor mode,[47] that acts as a program loader and supervisor for the small utility programs making up the rest of the system, and to provide locking and I/O services for these programs; beyond that, the kernel didn't intervene at all in user space.

Over the years the computing model changed, and Unix's treatment of everything as a file or byte stream no longer was as universally applicable as it was before. Although a terminal could be treated as a file or a byte stream, which is printed to or read from, the same did not seem to be true for a graphical user interface. Networking posed another problem. Even if network communication can be compared to file access, the low-level packet-oriented architecture dealt with discrete chunks of data and not with whole files. As the capability of computers grew, Unix became increasingly cluttered with code. It is also because the modularity of the Unix kernel is extensively scalable.[48] While kernels might have had 100,000 lines of code in the seventies and eighties, kernels of modern Unix successors like Linux have more than 13 million lines.[49]

Modern Unix-derivatives are generally based on module-loading monolithic kernels. Examples of this are the Linux kernel in its many distributions as well as the Berkeley Software Distribution variant kernels such as FreeBSD, DragonflyBSD, OpenBSD, NetBSD,and macOS. Apart from these alternatives, amateur developers maintain an active operating system development community, populated by self-written hobby kernels which mostly end up sharing many features with Linux, FreeBSD, DragonflyBSD, OpenBSD or NetBSD kernels and/or being compatible with them.[50]

Mac OS

Apple first launched its classic Mac OS in 1984, bundled with its Macintosh personal computer. Apple moved to a nanokernel design in Mac OS 8.6. Against this, the modern macOS (originally named Mac OS X) is based on Darwin, which uses a hybrid kernel called XNU, which was created combining the 4.3BSD kernel and the Mach kernel.[51]

Microsoft Windows

Microsoft Windows was first released in 1985 as an add-on to MS-DOS. Because of its dependence on another operating system, initial releases of Windows, prior to Windows 95, were considered an operating environment (not to be confused with an operating system). This product line continued to evolve through the 1980s and 1990s, with the Windows 9x series adding 32-bit addressing and pre-emptive multitasking; but ended with the release of Windows Me in 2000.

Microsoft also developed Windows NT, an operating system with a very similar interface, but intended for high-end and business users. This line started with the release of Windows NT 3.1 in 1993, and was introduced to general users with the release of Windows XP in October 2001—replacing Windows 9x with a completely different, much more sophisticated operating system. This is the line that continues with Windows 10.

The architecture of Windows NT's kernel is considered a hybrid kernel because the kernel itself contains tasks such as the Window Manager and the IPC Managers, with a client/server layered subsystem model.[52]

IBM Supervisor

Supervisory program or supervisor is a computer program, usually part of an operating system, that controls the execution of other routines and regulates work scheduling, input/output operations, error actions, and similar functions and regulates the flow of work in a data processing system.

Historically, this term was essentially associated with IBM's line of mainframe operating systems starting with OS/360. In other operating systems, the supervisor is generally called the kernel.

In the 1970s, IBM further abstracted the supervisor state from the hardware, resulting in a hypervisor that enabled full virtualization, i.e. the capacity to run multiple operating systems on the same machine totally independently from each other. Hence the first such system was called *Virtual Machine* or *VM*.

Development of microkernels

Although Mach, developed at Carnegie Mellon University from 1985 to 1994, is the best-known general-purpose microkernel, other microkernels have been developed with more specific aims. The L4 microkernel family (mainly the L3 and the L4 kernel) was created to demonstrate that microkernels are not necessarily slow.[39] Newer implementations such as Fiasco and Pistachio are able to run Linux next to other L4 processes in separate address spaces.[53][54]

Additionally, QNX is a microkernel which is principally used in embedded systems,[55] and the open-source software MINIX, while originally created for educational purposes, is now focussed on being a highly reliable and self-healing microkernel OS.

See also

- Comparison of operating system kernels
- Inter-process communication

Notes

1. "Kernel" (http://www.linfo.org/kernel.html). *Linfo*. Bellevue Linux Users Group. Retrieved 15 September 2016.
2. cf. Daemon (computing)
3. Roch 2004
4. Wulf 1974 pp.337–345
5. Silberschatz 1991
6. Tanenbaum, Andrew S. (2008). *Modern Operating Systems* (3rd ed.). Prentice Hall. pp. 50–51. ISBN 0-13-600663-9. ". . . nearly all system calls [are] invoked from C programs by calling a library procedure . . . The library procedure . . . executes a TRAP instruction to switch from user mode to kernel mode and start execution . . ."
7. Denning 1976
8. Swift 2005, p.29 quote: "isolation, resource control, decision verification (checking), and error recovery."
9. Schroeder 72
10. Linden 76
11. Stephane Eranian and David Mosberger, Virtual Memory in the IA-64 Linux Kernel (http://www.informit.com/articles/article.aspx?p=29961), Prentice Hall PTR, 2002
12. Silberschatz & Galvin, Operating System Concepts, 4th ed, pp. 445 & 446
13. Hoch, Charles; J. C. Browne (July 1980). "An implementation of capabilities on the PDP-11/45" (http://portal.acm.org/citation.cfm?id=850701&dl=acm&coll=&CFID=15151515&CFTOKEN=6184618) (PDF). *ACM SIGOPS Operating Systems Review*. **14** (3): 22–32. doi:10.1145/850697.850701 (https://doi.org/10.1145%2F850697.850701). Retrieved 2007-01-07.
14. A Language-Based Approach to Security (http://www.cs.cmu.edu/~rwh/papers/langsec/dagstuhl.pdf), Schneider F., Morrissett G. (Cornell University) and Harper R. (Carnegie Mellon University)
15. P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. *The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments (http://www.jya.com/paperF1.htm) Archived (https://web.archive.org/web/20070621155813/http://jya.com/paperF1.htm) 2007-06-21 at the* Wayback Machine.. In Proceedings of the 21st National Information Systems Security Conference, pages 303–314, Oct. 1998. [1] (http://csrc.nist.gov/nissc/1998/proceedings/paperF1.pdf).
16. J. Lepreau et al. *The Persistent Relevance of the Local Operating System to Global Applications (http://doi.acm.org/10.1145/504450.504477)*. Proceedings of the 7th ACM SIGOPS Eurcshelf/book001/book001.html Information Security: An Integrated Collection of Essays, IEEE Comp. 1995.
17. J. Anderson, *Computer Security Technology Planning Study (http://csrc.nist.gov/publications/history/ande72.pdf), Air Force Elect. Systems Div., ESD-TR-73-51, October 1972.*
18. * Jerry H. Saltzer; Mike D. Schroeder (September 1975). "The protection of information in computer systems" (http://web.mit.edu/Saltzer/www/publications/protection/). *Proceedings of the IEEE*. **63** (9): 1278–1308. doi:10.1109/PROC.1975.9939 (https://doi.org/10.1109%2FPROC.1975.9939).
19. Jonathan S. Shapiro; Jonathan M. Smith; David J. Farber (1999). "EROS: a fast capability system" (http://portal.acm.org/citation.cfm?doid=319151.319163). *Proceedings of the seventeenth ACM symposium on Operating systems principles*. **33** (5): 170–185. doi:10.1145/319344.319163 (https://doi.org/10.1145%2F319344.319163).
20. Dijkstra, E. W. *Cooperating Sequential Processes*. Math. Dep., Technological U., Eindhoven, Sept. 1965.
21. Brinch Hansen 70 pp.238–241
22. "SHARER, a time sharing system for the CDC 6600" (http://portal.acm.org/citation.cfm?id=363778&dl=ACM&coll=GUIDE&CFID=11111111&CFTOKEN=2222222). Retrieved 2007-01-07.
23. "Dynamic Supervisors – their design and construction" (http://portal.acm.org/citation.cfm?id=811675&dl=ACM&coll=GUIDE&CFID=11111111&CFTOKEN=2222222). Retrieved 2007-01-07.
24. Baiardi 1988
25. Levin 75
26. Denning 1980

27. Jürgen Nehmer *The Immortality of Operating Systems, or: Is Research in Operating Systems still Justified? (http://portal.acm.org/citation.cfm?id=723612)* Lecture Notes In Computer Science; Vol. 563. Proceedings of the International Workshop on Operating Systems of the 90s and Beyond. pp. 77–83 (1991) ISBN 3-540-54987-0 [2] (https://link.springer.com/chapter/10.1007%2FBFb0024528#page-1) quote: "The past 25 years have shown that research on operating system architecture had a minor effect on existing main stream [*sic*] systems."
28. Levy 84, p.1 quote: "Although the complexity of computer applications increases yearly, the underlying hardware architecture for applications has remained unchanged for decades."
29. Levy 84, p.1 quote: "Conventional architectures support a single privileged mode of operation. This structure leads to monolithic design; any module needing protection must be part of the single operating system kernel. If, instead, any module could execute within a protected domain, systems could be built as a collection of independent modules extensible by any user."
30. Roch 2004
31. Open Sources: Voices from the Open Source Revolution (http://oreilly.com/catalog/opensources/book/appa.html)
32. Virtual addressing is most commonly achieved through a built-in memory management unit.
33. Recordings of the debate between Torvalds and Tanenbaum can be found at dina.dk (http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html) Archived (https://web.archive.org/web/20121003060514/http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html) 2012-10-03 at the Wayback Machine., groups.google.com (http://groups.google.com/group/comp.os.minix/browse_thread/thread/c25870d7a41696d2/f447530d082cd95d?tvc=2#f447530d082cd95d), oreilly.com (http://www.oreilly.com/catalog/opensources/book/appa.html) and Andrew Tanenbaum's website (http://www.cs.vu.nl/~ast/reliable-os/)
34. Matthew Russell. "What Is Darwin (and How It Powers Mac OS X)" (http://oreilly.com/pub/a/mac/2005/09/27/what-is-darwin.html?page=2). O'Reilly Media. quote: "The tightly coupled nature of a monolithic kernel allows it to make very efficient use of the underlying hardware [...] Microkernels, on the other hand, run a lot more of the core processes in userland. [...] Unfortunately, these benefits come at the cost of the microkernel having to pass a lot of information in and out of the kernel space through a process known as a context switch. Context switches introduce considerable overhead and therefore result in a performance penalty."
35. Operating Systems/Kernel Models (https://en.wikiversity.org/wiki/Operating_Systems/Kernel_Models#Monolithic_Kernel)
36. Liedtke 95
37. Härtig 97
38. Hansen 73, section 7.3 p.233 "*interactions between different levels of protection require transmission of messages by value*"
39. The L4 microkernel family – Overview (http://os.inf.tu-dresden.de/L4/overview.html)
40. https://www.youtube.com/watch?v=ggnFoDqzGMU
41. KeyKOS Nanokernel Architecture (http://www.cis.upenn.edu/~KeyKOS/NanoKernel/NanoKernel.html) Archived (https://web.archive.org/web/20110621235229/http://www.cis.upenn.edu/~KeyKOS/NanoKernel/NanoKernel.html) 2011-06-21 at the Wayback Machine.
42. Ball: Embedded Microprocessor Designs, p. 129
43. Hansen 2001 (os), pp.17–18
44. BSTJ version of C.ACM Unix paper (http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html)
45. Introduction and Overview of the Multics System (http://www.multicians.org/fjcc1.html), by F. J. Corbató and V. A. Vissotsky.
46. "The Single Unix Specification" (https://www.unix.org/what_is_unix/single_unix_specification.html). *The open group*.
47. The highest privilege level has various names throughout different architectures, such as supervisor mode, kernel mode, CPL0, DPL0, ring 0, etc. See Ring (computer security) for more information.
48. Unix's Revenge by Horace Dediu (http://www.asymco.com/2010/09/29/unixs-revenge/)
49. Linux Kernel 2.6: It's Worth More! (http://www.dwheeler.com/essays/linux-kernel-cost.html), by David A. Wheeler, October 12, 2004
50. This community mostly gathers at Bona Fide OS Development (http://www.osdever.net), The Mega-Tokyo Message Board (http://www.mega-tokyo.com/forum) and other operating system enthusiast web sites.
51. XNU: The Kernel (http://www.kernelthread.com/mac/osx/arch_xnu.html) Archived (https://www.webcitation.org/6187NYSIz?url=http://osxbook.com/book/bonus/ancient/whatismacosx//arch_xnu.html) 2011-08-22 at WebCite
52. Windows History: Windows Desktop Products History (http://www.microsoft.com/windows/WinHistoryDesktop.mspx)
53. The Fiasco microkernel – Overview (http://os.inf.tu-dresden.de/fiasco/overview.html)
54. L4Ka – The L4 microkernel family and friends (http://www.l4ka.org)
55. QNX Realtime Operating System Overview (http://www.qnx.com/products/rtos/microkernel.html)

References

- Roch, Benjamin (2004). "Monolithic kernel vs. Microkernel" (https://web.archive.org/web/20061101012856/http://www.vmars.tuwien.ac.at/courses/akti12/journal/04ss/article_04ss_Roch.pdf) (PDF). Archived from the original (http://www.vmars.tuwien.ac.at/courses/akti12/journal/04ss/article_04ss_Roch.pdf) (PDF) on 2006-11-01. Retrieved 2006-10-12.
- Silberschatz, Abraham; James L. Peterson; Peter B. Galvin (1991). *Operating system concepts* (http://portal.acm.org/citation.cfm?id=95329&dl=acm&coll=&CFID=15151515&CFTOKEN=6184618). Boston, Massachusetts: Addison-Wesley. p. 696. ISBN 0-201-51379-X.
- Ball, Stuart R. (2002) [2002]. *Embedded Microprocessor Systems: Real World Designs* (first ed.). Elsevier Science. ISBN 0-7506-7534-9.
- Deitel, Harvey M. (1984) [1982]. *An introduction to operating systems* (http://portal.acm.org/citation.cfm?id=79046&dl=GUIDE&coll=GUIDE) (revisited first ed.). Addison-Wesley. p. 673. ISBN 0-201-14502-2.
- Denning, Peter J. (December 1976). "Fault tolerant operating systems" (http://portal.acm.org/citation.cfm?id=356680&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618). *ACM Computing Surveys*. **8** (4): 359–389. doi:10.1145/356678.356680 (https://doi.org/10.1145%2F356678.356680). ISSN 0360-0300 (https://www.worldcat.org/issn/0360-0300).
- Denning, Peter J. (April 1980). "Why not innovations in computer architecture?" (http://portal.acm.org/citation.cfm?id=859506&coll=&dl=ACM&CFID=15151515&CFTOKEN=6184618). *ACM SIGARCH Computer Architecture News*. **8** (2): 4–7. doi:10.1145/859504.859506 (https://doi.org/10.1145%2F859504.859506). ISSN 0163-5964 (https://www.worldcat.org/issn/0163-5964).
- Hansen, Per Brinch (April 1970). "The nucleus of a Multiprogramming System" (http://portal.acm.org/citation.cfm?id=362278&dl=ACM&coll=GUIDE&CFID=11111111&CFTOKEN=2222222). *Communications of the ACM*. **13** (4): 238–241. doi:10.1145/362258.362278 (https://doi.org/10.1145%2F362258.362278). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).
- Hansen, Per Brinch (1973). *Operating System Principles* (http://portal.acm.org/citation.cfm?id=540365). Englewood Cliffs: Prentice Hall. p. 496. ISBN 0-13-637843-9.
- Hansen, Per Brinch (2001). "The evolution of operating systems" (http://brinch-hansen.net/papers/2001b.pdf) (PDF). Retrieved 2006-10-24. included in book: Per Brinch Hansen, ed. (2001). "1" (http://brinch-hansen.net/papers/2001b.pdf) (PDF). *Classic operating systems: from batch processing to distributed systems* (http://portal.acm.org/citation.cfm?id=360596&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618). New York,: Springer-Verlag. pp. 1–36. ISBN 0-387-95113-X.
- Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, Jean Wolter *The performance of μ-kernel-based systems (http://os.inf.tu-dresden.de/pubs/sosp97/#Karshmer:1991:OSA)*, Härtig, Hermann; Hohmuth, Michael; Liedtke, Jochen; Schönberg, Sebastian (1997). "The performance of μ-kernel-based systems". *Proceedings of the sixteenth ACM symposium on Operating systems principles - SOSP '97*. p. 66. doi:10.1145/268998.266660 (https://doi.org/10.1145%2F268998.266660). ISBN 0897919165. ACM SIGOPS Operating Systems Review, v.31 n.5, p. 66–77, Dec. 1997
- Houdek, M. E., Soltis, F. G., and Hoffman, R. L. 1981. *IBM System/38 support for capability-based addressing (http://portal.acm.org/citation.cfm?id=800052.801885)*. In Proceedings of the 8th ACM International Symposium on Computer Architecture. ACM/IEEE, pp. 341–348.
- Intel Corporation (2002) *The IA-32 Architecture Software Developer's Manual, Volume 1: Basic Architecture (http://www.intel.com/design/pentium4/manuals/24547010.pdf)*
- Levin, R.; Cohen, E.; Corwin, W.; Pollack, F.; Wulf, William (1975). "Policy/mechanism separation in Hydra" (http://portal.acm.org/citation.cfm?id=806531&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618). *ACM Symposium on Operating Systems Principles / Proceedings of the fifth ACM symposium on Operating systems principles*. **9** (5): 132–140. doi:10.1145/1067629.806531 (https://doi.org/10.1145%2F1067629.806531).
- Levy, Henry M. (1984). *Capability-based computer systems* (http://www.cs.washington.edu/homes/levy/capabook/index.html). Maynard, Mass: Digital Press. ISBN 0-932376-22-3.
- Liedtke, Jochen. *On μ-Kernel Construction (http://i30www.ira.uka.de/research/publications/papers/index.php?lid=en&docid=642)*, *Proc. 15th ACM Symposium on Operating System Principles (SOSP)*, December 1995
- Linden, Theodore A. (December 1976). "Operating System Structures to Support Security and Reliable Software" (http://portal.acm.org/citation.cfm?id=356682&coll=&dl=ACM&CFID=15151515&CFTOKEN=6184618). *ACM Computing Surveys*. **8** (4): 409–445. doi:10.1145/356678.356682 (https://doi.org/10.1145%2F356678.356682). ISSN 0360-0300 (https://www.worldcat.org/issn/0360-0300)., "Operating System Structures to Support Security and Reliable Software" (http://csrc.nist.gov/publications/history/lind76.pdf) (PDF). Retrieved 2010-06-19.
- Lorin, Harold (1981). *Operating systems* (http://portal.acm.org/citation.cfm?id=578308&coll=GUIDE&dl=GUIDE&CFID=2651732&CFTOKEN=19681373). Boston, Massachusetts: Addison-Wesley. pp. 161–186. ISBN 0-201-14464-6.

- Schroeder, Michael D.; Jerome H. Saltzer (March 1972). "A hardware architecture for implementing protection rings" (http://portal.acm.org/citation.cfm?id=361275&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618). *Communications of the ACM*. **15** (3): 157–170. doi:10.1145/361268.361275 (https://doi.org/10.1145%2F361268.361275). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).
- Shaw, Alan C. (1974). *The logical design of Operating systems* (http://portal.acm.org/citation.cfm?id=540329). Prentice-Hall. p. 304. ISBN 0-13-540112-7.
- Tanenbaum, Andrew S. (1979). *Structured Computer Organization*. Englewood Cliffs, New Jersey: Prentice-Hall. ISBN 0-13-148521-0.
- Wulf, W.; E. Cohen; W. Corwin; A. Jones; R. Levin; C. Pierson; F. Pollack (June 1974). "HYDRA: the kernel of a multiprocessor operating system" (http://www.cs.virginia.edu/papers/p337-wulf.pdf) (PDF). *Communications of the ACM*. **17** (6): 337–345. doi:10.1145/355616.364017 (https://doi.org/10.1145%2F355616.364017). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).
- Baiardi, F.; A. Tomasi; M. Vanneschi (1988). *Architettura dei Sistemi di Elaborazione, volume 1* (http://www.pangloss.it/libro.php?isbn=882042746X&id=4357&PHPSESSID=9da1895b18ed1cda115cf1c7ace9bdf0) (in Italian). Franco Angeli. ISBN 88-204-2746-X.
- Swift, Michael M.; Brian N. Bershad; Henry M. Levy. *Improving the reliability of commodity operating systems* (http://nooks.cs.washington.edu/nooks-tocs.pdf) (PDF).
- "Improving the reliability of commodity operating systems" (http://doi.acm.org/10.1145/1047915.1047919). Doi.acm.org. doi:10.1002/spe.4380201404 (https://doi.org/10.1002%2Fspe.4380201404). Retrieved 2010-06-19.
- "ACM Transactions on Computer Systems (TOCS), v.23 n.1, p. 77–110, February 2005".

Further reading

- Andrew Tanenbaum, *Operating Systems – Design and Implementation (Third edition)*;
- Andrew Tanenbaum, *Modern Operating Systems (Second edition)*;
- Daniel P. Bovet, Marco Cesati, *The Linux Kernel*;
- David A. Peterson, Nitin Indurkhya, Patterson, *Computer Organization and Design*, Morgan Koffman (ISBN 1-55860-428-6);
- B.S. Chalk, *Computer Organisation and Architecture*, Macmillan P.(ISBN 0-333-64551-0).

External links

- Detailed comparison between most popular operating system kernels (http://widefox.pbwiki.com/Kernel%20Comparison%20Linux%20vs%20Windows)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Kernel_(operating_system)&oldid=830585090"