Kernel : An Operating system is a well organized collection of a basic set of program.

The most important program in the collection is called kernel of Operating system (the kernel is also called supervisor monitor or nucleus). The kernel is the minimal OS program that is loaded in the main memory when computer is initialized during bootstrapping. It resides permanently in the main memory until the system is shutdown. It typically resides in the low memory address part of the main memory. The term operating system is defined as the notion of the kernel (the case).

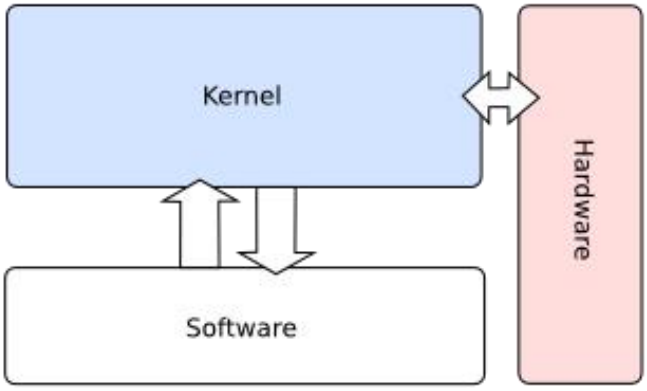
The supporting hardware management software, the various system libraries and the specialized system application is called utilities.

The kernel is the controlling authority and provides care Operating system functionalities. It typically manages hardware components and export services such as memory management and I/O device management. It provides the generic interface for the rest of Operating system programs. The non-kernel parts of Operating system are loaded into and unloaded from the main memory as the situation demands during run time(In some system these other part are supported the rough system applications).

The kernel contains all the critical functionalities of an Operating system which are the needed for the effective and smooth function of a computer. However in modern monolithic system the demarcation between the Operating system and the kernel is blushed unless stated otherwise we will use the terms kernel and Operating system to mean the same.

Monolithic Kernel

A monolithic kernel includes all (or at least, most) of its services in the kernel proper. This reduces the amount of context switches and messaging involved making the concept faster than a Microkernel. On the downside, the amount of code running in kernel space makes the kernel more prone to fatal bugs.



Basic overview of a monolithic kernel.

The word "monolithic" by itself means a single piece (mono) that is of or like stone (lithic), however when applied to kernels the exact meaning is more general. Most people consider that a kernel where device drivers and services run as part of the kernel is a monolithic kernel, regardless of whether parts are dynamically loaded "kernel modules" or if everything is a true single unchangeable binary. For this reason I draw a distinction between "monolithic" and "pure monolithic".

Modern versions of Linux are a well-known example of a monolithic kernel - while drivers are shipped as dynamically loaded "kernel modules" they are still loaded into and running in kernel space. Monolithic kernels are common for the 80x86/PC architecture.

Examples of "pure monolithic" kernels are rare for the 80x86/PC architecture (but more common in embedded systems). This is because of the wide variety of devices, hardware and CPU features that may be present within a

modern PC - a pure monolithic kernel would need to be far too large or compiled specifically for the computer before use.

In general most OS's aren't "pure monolithic" or "pure micro-kernel", but fall somewhere between these extremes in order to make use of the advantages of both methods. In a Monolithic Kernel, all OS services run along with the main Kernel thread, thus also residing in the same memory area. All thread and process are running on the Kernel space. This approach provides rich and powerful hardware access. Monolithic systems are easier to design and implement then other systems, and extremely efficient if well-written.

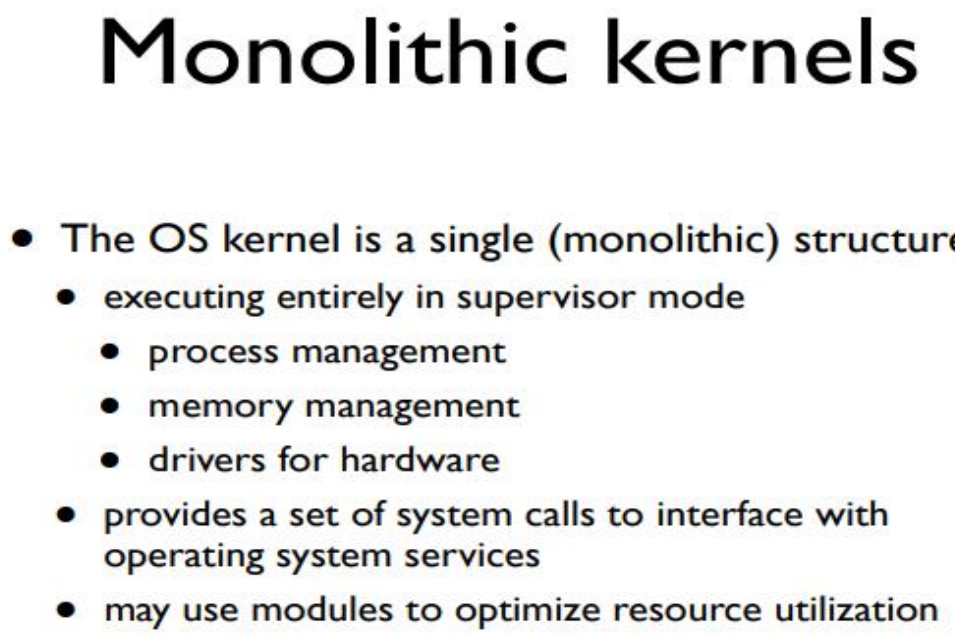
The main disadvantage of monolithic kernels are the dependencies between system components - a bug in a device driver might crash the entire systems - and the fact that large kernels can become very difficult to maintain.

Monolithic architecture examples

- Unix kernels
- Linux
- MS-DOS
- Microsoft Windows 9x series (95, 98, Windows 98SE, Me)
- OpenVMS
- XTS-400

Amiga

The Commodore Amiga was released in 1985, and was among the first (and certainly most successful) home computers to feature a microkernel operating system. The Amiga's kernel, *exec.library*, was small but capable, providing fast pre-emptive multitasking on hardware similar to the cooperatively-multitasked Apple Macintosh, and an advanced dynamic linking system that allowed for easy expansion.



• easy to design and develop
• hard to evolve without risking growing pains

Typical examples • Traditional UNIX kernels (System V and BSD flavours) • FreeBSD • Solaris • Linux

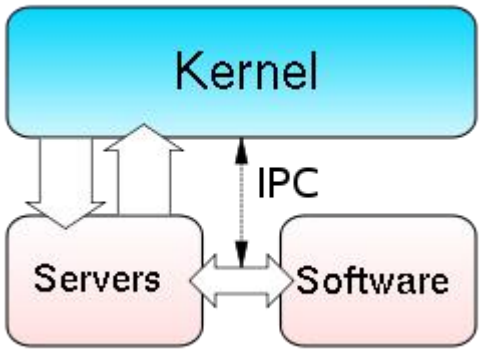
Advantages

• Efficiently using function calls between kernel modules
• Requires a low number of context switches

Drawbacks

• Limitations in robustness
• No fault or privilege isolation

Hybrid (or) Modular kernels:



Hybrid kernels are used in most commercial operating systems such as Microsoft Windows NT, 2000, XP, Vista, 7 and 8. Apple Inc's own Mac OS X uses a hybrid kernel called XNU which is based upon code from Carnegie Mellon's Mach kernel andFreeBSD'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.

Nano kernels:

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.

Nano kernel: **Nanokernel** is hardware access without IPC s sometimes. Or a microkernel that is really micro as opposed to MINO kernels (Micro In Name Only). Kind of a marketing term.

Nanokernels are relatively small kernels which provide hardware abstraction, but lack system services. They provide the very basic OS functionalities and rest is implemented as applications

A nanokernel is a small kernel that offers hardware abstraction, but without system services. Larger kernels are designed to offer more features and manage more hardware abstraction. Modern microkernels lack system services as well, hence, the terms microkernal and nanokernal have become analogous.

- A kernel in which the total volume of kernel code, that is, the code being executed in the hardware's privileged mode, is quite small.
- A virtualization layer beneath an operating system, which is more precisely called a hypervisor.
- A hardware abstraction layer (HAL), which forms the lowest level portion of a kernel.
- Occasionally, the term nanokernel is used to describe a kernel that supports a nanosecond clock resolution.

The term **nanokernel** is used to describe a specific **type of kernel**. The prefix "pico-", or "nano-", "micro-" is usually denoting the "size" of the kernel.. **Bigger kernels** are more built with more features, and **handle more hardware abstraction**. Nanokernels are relatively small kernels which **provide hardware abstraction**, but **lack system services**. Modern microkernels also lack system services, so the terms have become analogous.

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.

The "nanokernel" of Symbian OS is an example of this, or the Apple example given in the article.

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, 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 forreal-time control.

- A special case of microkernels
- Reduction of mechanisms in supervisor space
- Moves all abstractions to user-space
- Limited to virtualization
- Exposes the hardware to user-space programs
- Currently available as research prototypes only

Reentrant Kernels

A *reentrant* program is one that does not modify itself and any global data. Multiple processes or threads can execute reentrant programs concurrently without interfering one another. They can share reentrant programs, but have their own private data. A *reentrant kernel* is one where many processes/ threads can execute the same kernel programs concurrently without affecting one another. In non-reentrant kernels, a process does not modify kernel programs, but can modify global kernel data. Consequently, if a process is executing operating system programs, no other processes may be allowed to execute the programs, nor may the system start another kernel path execution (due to an interrupt or exception) when the kernel accesses the global data. Consequently, interrupts from I/O devices may not be handled immediately by the operating system.

Generally, an operating system is composed of both reentrant and non-reentrant functions. The reentrant functions may modify local (on-stack) data, but they do not modify any global data. Concurrent executions of reentrant function(s) do not affect the behaviour of one another. The operating system needs to make sure that non-reentrant functions are executed mutually exclusively by kernel paths so that the function executions do not modify global data at the same time.

All Unix kernels are reentrant. This means that several processes may be executing in Kernel Mode at the same time. Of course, on uniprocessor systems, only one process can progress, but many can be blocked in Kernel Mode when waiting for the CPU or the completion of some I/O operation. For instance, after issuing a read to a disk on behalf of a process, the kernel lets the disk controller handle it and resumes executing other processes. An interrupt notifies the kernel when the device has satisfied the read, so the former process can resume the execution.

One way to provide reentrancy is to write functions so that they modify only local variables and do not alter global data structures. Such functions are called reentrant functions . But a reentrant kernel is not limited only to such reentrant functions (although that is how some real-time kernels are implemented). Instead, the kernel can include nonreentrant functions and use locking mechanisms to ensure that only one process can execute a nonreentrant function at a time.

Difference between Non re-entrant and Re-entrant Kernel

In Non re-entrant kernel mode, every single process acts on its own set of memory locations and thus cannot interfere with the others.

On a "Re-entrant Kernel", when a hardware interrupt occurs, it is able to suspend the current running process,even if the process is in Kernel Mode. This improves the throughput of the device controllers that issue interrupts. What exactly happens is this: When a device issues an interrupt, it waits for the CPU to acknowledge. If the Kernel answers quickly, the device controller will be able to perform other tasks while the CPU handles the interrupt.

Booting process

In computing, **booting** (also known as **booting up**) is the initial set of operations that a computer system performs after electrical power to the CPU is switched on or when the computer is reset. The process begins when a computer is turned on for the first time, is re-energized after being turned off, when it is reset or when the operator invokes a LOAD function from the console, and ends when the computer is ready to perform its normal operations. On modern general purpose computers, this can take tens of seconds and typically involves performing a power-on self-test, locating and initializing peripheral devices, and then finding, loading and starting an operating system. Many computer systems also allow these operations to be initiated by a software command without cycling power, in what is known as a soft reboot, though some of the initial operations might be skipped on a soft reboot. A **boot loader** is a computer program that loads the main operating system or runtime environment for the computer after completion of the self-tests.

The computer term *boot* is short for *bootstrap* or *bootstrap load* and derives from the phrase *to pull oneself up by one's bootstraps*. The usage calls attention to the requirement that, if most software is loaded onto a computer by other software already running on the computer, some mechanism must exist to load initial software onto the computer. Early computers used a variety of ad-hoc methods to get a small program into memory to solve this problem. The invention of read-only memory (ROM) of various types solved this paradox by allowing computers to be shipped with a start up program that could not be erased. Growth in the capacity of ROM has allowed ever more elaborate start up procedures to be implemented.

On general purpose computers, the boot process begins with the execution of an initial program stored in boot ROMs or read in another fashion. In some older computers, the initial program might have been the application to run, if no operating system was used, or the operating system. In other computers, the initial program is a boot loader that may then load into random-access memory (RAM), from nonvolatile secondary storage (such as a hard disk drive) or, in some older computers, from a medium such as punched cards, punched tape, or magnetic tape, the binary code of an operating system or runtime environment and then execute it. If the boot loader is limited in its size and capabilities, it may, instead, load a larger and more capable secondary boot loader, which would then load the operating system or runtime environment. Some embedded systems do not require a noticeable boot sequence to begin functioning and when turned on may simply run operational programs that are stored in ROM.

Dynamic linker

In computing, a **dynamic linker** is the part of an operating system (OS) that loads (copies from persistent storage to RAM) and links (fills jump tables and relocates pointers) the shared libraries needed by an executable at run time, that is, when it is executed. The specific operating system and executable format determine how the dynamic linker functions and how it is implemented. Linking is often referred to as a process that is performed at compile time of the executable while a dynamic linker is in actuality a special loader that loads external shared libraries into a running process and then binds those shared libraries dynamically to the running process. The specifics of how a dynamic linker functions is operating-system dependent.