**Chapter 02: Operating System Structures**

Table of Contents

[2.1 Operating System Services 4](#_Toc83032770)

[2.2 User and Operating System Interface 7](#_Toc83032771)

[Command Interpreter 7](#_Toc83032772)

[Graphical User Interface 7](#_Toc83032773)

[Choice of Interface 8](#_Toc83032774)

[2.3 System Calls 9](#_Toc83032775)

[Application Programming Interfaces 10](#_Toc83032776)

[System Call Interfaces 10](#_Toc83032777)

[System Call Parameters 11](#_Toc83032778)

[2.4 Types of System Calls 13](#_Toc83032779)

[Process Control 13](#_Toc83032780)

[File Management 14](#_Toc83032781)

[Device Management 14](#_Toc83032782)

[Information Maintenance 14](#_Toc83032783)

[Communication 15](#_Toc83032784)

[Protection 15](#_Toc83032785)

[2.5 System Programs 16](#_Toc83032786)

[2.6 Operating System Design and Implementation 18](#_Toc83032787)

[Design Goals 18](#_Toc83032788)

[2.7 Operating System Structure 19](#_Toc83032789)

[Layered Approach 19](#_Toc83032790)

[Microkernels 20](#_Toc83032791)

[Modules 21](#_Toc83032792)

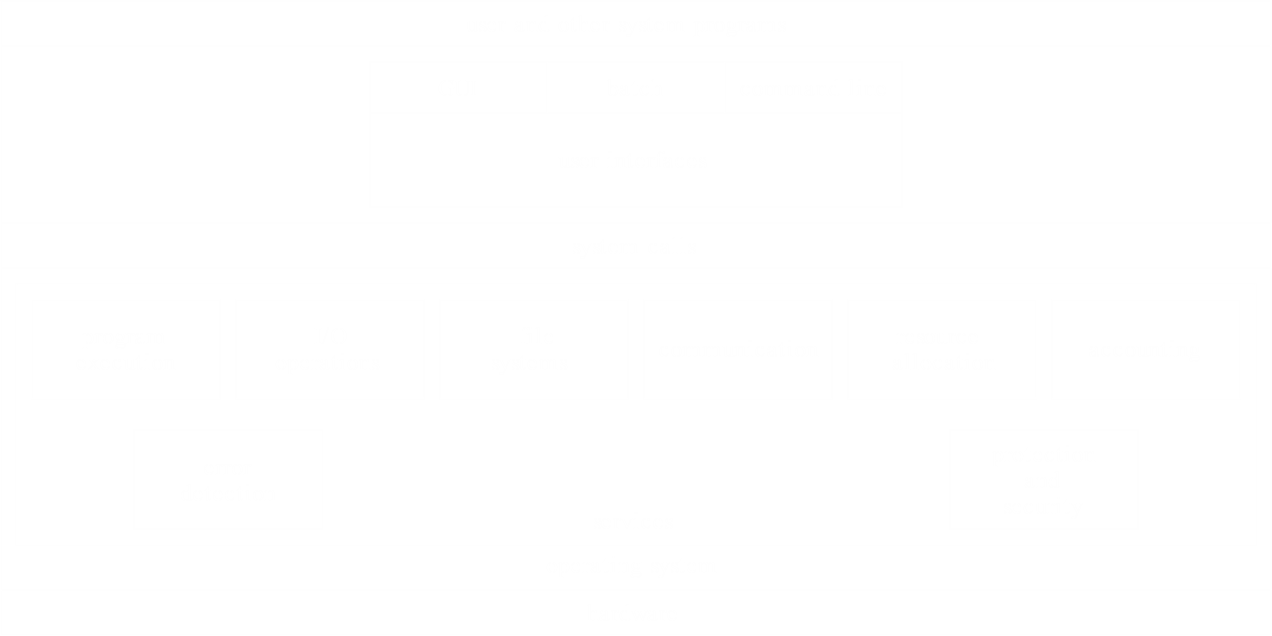
[2.9 Operating System Generation 23](#_Toc83032793)

[2.10 System Boot 25](#_Toc83032794)

The design of an operating system varies greatly based on what **goals** it is meant to achieve. These goals form the basis for choices between various **algorithms** and **strategies**. We can view these from different perspectives, based on what **services** the system provides or on the **interfaces** available to users and programmers or on the **components and interconnections**.

## 2.1 Operating System Services

Operating systems provide a range of services to programs and to users to make **programming easier**. The exact services vary, but a few common ones can be identified.



One set of services exist to help the **user**:

* **User Interface** – This can be of several types. A **command-line** interface uses text commands. A **batch** interface executes files that contain the commands. A **graphical user interface**, which takes input with a mouse and keyboard, allows users to choose from menus, make selections and enter text. An operating system may have one or more of these interfaces.
* **Program Execution** – Programs can be **loaded** into memory and **executed**. The programs can end execution normally, or end abruptly due to an error.
* **I/O Operations** – Programs must be able to use I/O operations, either with **I/O devices** or with **files**. Some devices, like recording to CDs, may even require special functions. Users cannot be allowed to do this themselves to keep the system efficient and protected.
* **File-System Manipulation** – Programs must be able to **read** and **write** files and directories and **create**, **delete** and **search** for them by name. Some operating systems also provide **permissions** to allow or deny specific actions based on user.
* **Communications** – Processes, either on the same computer or computers tied by a network, need to be able to **exchange information**. This is done using either a **shared memory**, or **message passing**, where packets of information in a pre-defined format are moved between processes.
* **Error Detection** – Errors can occur in the CPU, memory hardware, I/O devices or user programs. For each type of error, the OS must be able to take appropriate **action**, sometimes by halting the process or even the system. It may also return an error code to the process which caused the error.

Another set of services exist to make the **system** efficient:

* **Resource Allocation** – For **multiple users** running multiple tasks, the OS needs to allocate computing resources in such a way so as to achieve maximum efficiency. This involves many considerations, such as CPU speed, tasks being executed, number of registers available, etc. Even peripherals may need to be allocated, though sometimes this is done with a simple request and release process.
* **Accounting** – The OS must keep track of which users are using which resources and how intensively. This could be used for **billing** purposes or just for **usage statistics**, which are valuable if we want to improve the system.
* **Protection and Security** – Processes should not be able to interfere with each other or with the OS itself. This means all access to computing resources needs to be controlled. Security from outside interference is also important, such as by user authentication and recording invalid access attempts.

## 2.2 User and Operating System Interface

### Command Interpreter

A command interpreter, or a **command-line interface (CLI)**, either exists in the **kernel** or a special program that is running right from user login. If there are multiple interpreters in a system, they are called **shells**. There are even third-party shells available. Most shells are similar and the choice between them is mostly about user preference.

The main job of the CLI is to get the next user-specified command and execute it. Many of these commands involve files, creating, deleting, listing, printing, copying, executing, etc. These can be implemented in two ways. In one approach, the CLI itself contains the code needed to execute the commands. This means the number of acceptable commands determines the size of the CLI. In another approach, the CLI uses the command to identify a file in memory that it executes. That file contains the code required to perform the required action. This is the approach taken by UNIX. It allows new commands to be added in the future without having to change and enlarge the CLI.

### Graphical User Interface

**GUI**s are far more **user friendly**. They use a mouse-based window-and-menu system, commonly called a **desktop**. The user moves the mouse pointer to icons on the screen, clicking items which causes programs to execute, files or directories to open or other system functions to execute depending on what is being clicked.

For smartphones and other mobile operating systems, **touchscreens** are used instead of mice. They depend on **gestures**, such as swipes or presses, to perform different tasks. Earlier smartphones had physical keyboards, but newer ones use a virtual keyboard.

UNIX systems have traditionally used CLIs, but GUIs also exist, Linux being one of the most famous.

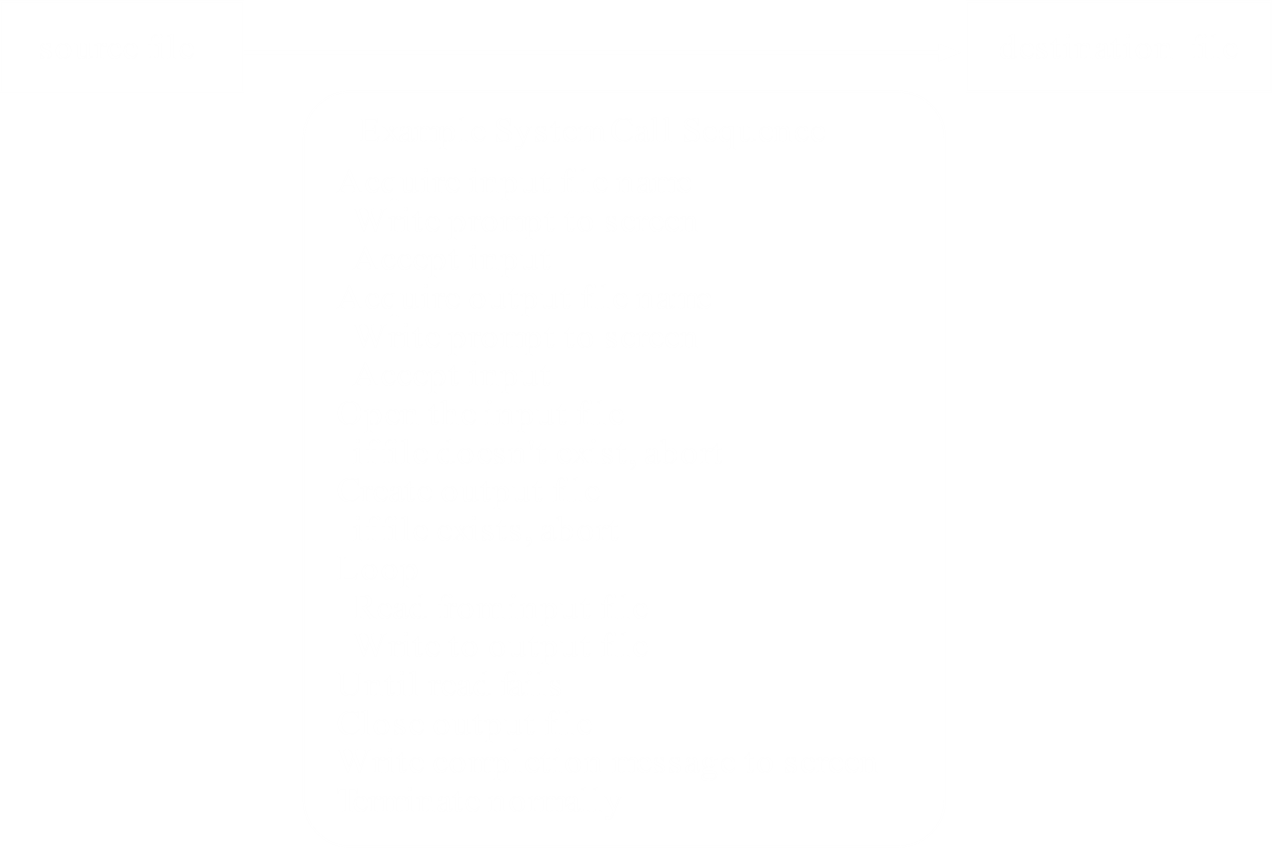
### Choice of Interface

The choice between CLIs and GUIs is mostly about personal preference. System administrators with deep knowledge of the system may prefer **CLI**s, since those are more **efficient** to use. CLIs also tend to make **repetitive tasks** easier, for example by using **shell scripts**, which are files with the required commands. General users however, may prefer **GUI**s, which, though far less efficient and perhaps not even containing all the possible commands, are admittedly **easier** to use. The design of a user-friendly GUI has nothing to do with the actual OS.

## 2.3 System Calls

System calls provide an **interface** to the services made available by an OS. These are generally available as **routines** written in C or C++, although some low-level tasks may use Assembly Language. Note that the system calls we will be seeing are **generic**. Actual system calls vary from system to system.

If we want to write a simple program that reads data from one file and copies it to another, we will first need to provide the names of the files. In an interactive system, just this step will involve several I/O system calls. At the very least, a prompt will need to be displayed asking for the file names and the input will need to be recorded. On a modern system, a menu may need to be displayed with the different directories and file names from where the user can select the required files using their mouse. The entire program will most likely involve a huge number of system calls.



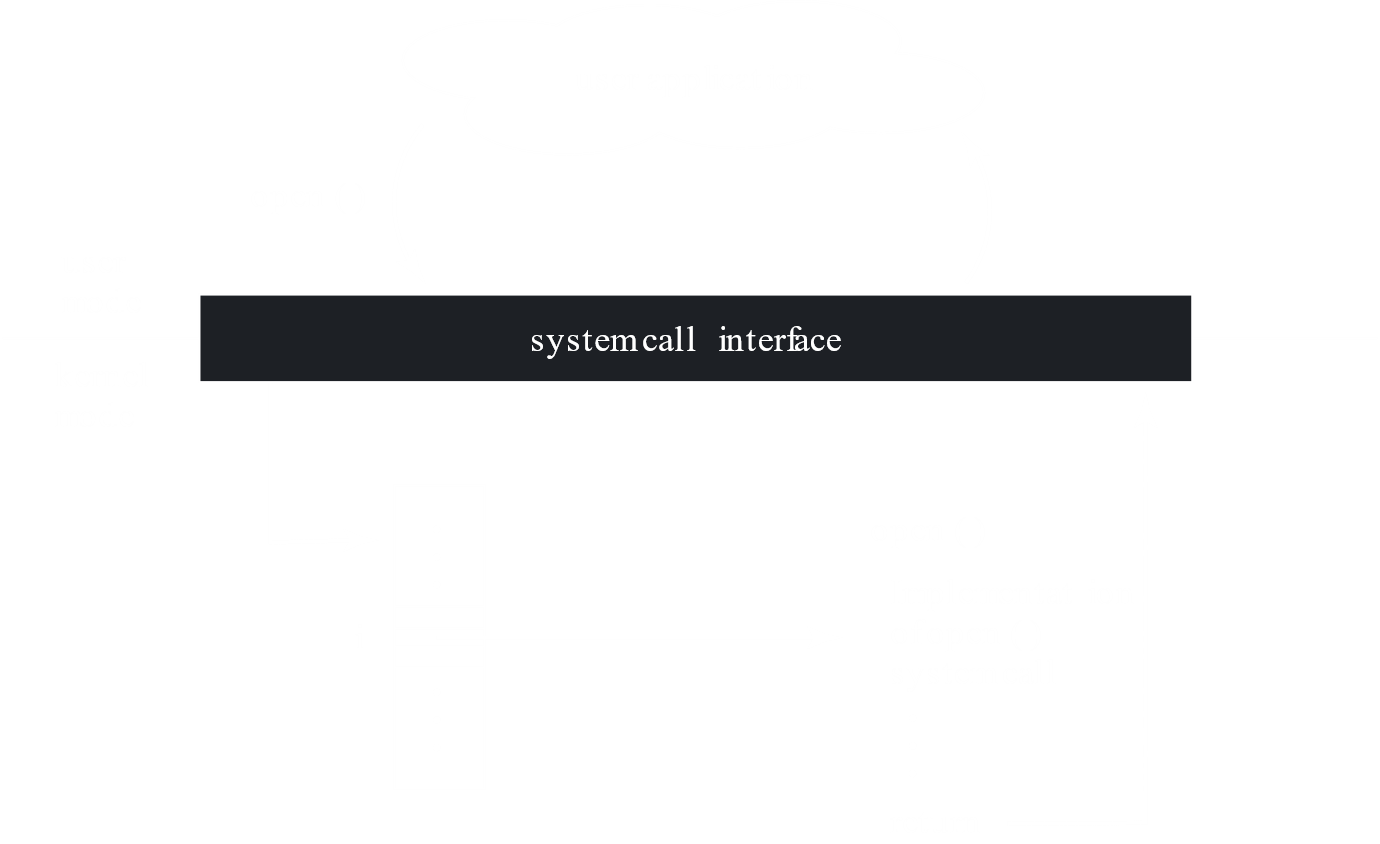
### Application Programming Interfaces

Systems frequently execute **thousands** of system calls each second. Most programmers however, use an **application programming interface (API)**, which specifies a set of functions available to the programmer. The API in turn makes the actual system calls. APIs are accessed via a **library** of code provided by the OS. For UNIX and Linux programs written in C, the library is called libc.

One of the benefits of using an API instead of making the system calls ourselves is **portability**. If we use an API, any OS that supports the API will be able to compile and execute the program we write. Also, APIs are **easier** to deal with than system calls, although there are a lot of cases where the API functions and the system calls are actually very similar.

### System Call Interfaces

For most programming languages, the **run-time support system** (a set of functions built into libraries in the compiler) provides a **system call interface**. This serves as a link to the system calls available in the OS. The system call interface accepts function calls in the API, invokes the necessary system calls within the OS kernel and returns the status of the system call as well as any return values. Typically, a **number** is associated with each system call and the system call interface maintains an index.



All of this means the programmer does not need to know how the system call works or what it does. They only need to obey the API and understand what the OS will do as a result of executing a certain API function. Most of the details of the OS interface are hidden by the API and managed by the run-time support library.

### System Call Parameters

System calls are more complicated than just calling a function. They often need to know **more information**. For example, taking input requires information about whether to take input from a file or from a device as well as the address and size of the buffer into which the input will be taken. Of course, some of this information may be implicit, but others may need to be passed as **parameters**.

There are three approaches to passing parameters:

1. **Registers** – Parameters can simply be stored in registers. However, there may be more parameters than registers, which will cause this approach to fail.
2. **Memory** – If there are too many parameters, they could be stored in a block or a table in memory and the address of the block passed as a parameter in a register. This approach is taken by Linux.
3. **Stack** – Parameters can also be pushed onto a stack by the program and popped from the stack by the OS.

The latter two approaches are preferred by many operating systems since they do not limit the number or length of parameters.

## 2.4 Types of System Calls

There are **six** types of system calls:

1. Process Control
2. File Manipulation
3. Device Manipulation
4. Information Maintenance
5. Communications
6. Protection

### Process Control

**Process control** system calls include basic things like ending a program, either normally or abnormally, dealing with any errors, moving between processes and so on. Errors are generally indicated with some **error code**, with higher values indicating more serious errors.

Regarding shared memory between processes, process control system calls may allow a process to **lock** the shared memory while it is being used so that other processes cannot mess things up.

Regarding the switching of processes, multiple approaches could be taken. For example, control could go back to the previous program, or move onto the next one. Multiple programs may even be able to run concurrently on some systems.

### File Management

**File management** system calls include all the basic file management activities, such as creating, reading and writing files, but also involve editing attributes or metadata related to the files.

### Device Management

A device may refer to physical devices, but they could also refer to files in a scenario where files cannot immediately be accessed. Essentially, any resource that forces the user program to first make a **request** may be considered a device. Once allocated, the device can be used similarly to files, read and writing data. Once it is no longer being used, the device must be **released**. All of these actions are done using **device management** system calls.

### Information Maintenance

**Information maintenance** system calls exist to simply **exchange information**. For example, the user program may need to know the number of active users and would ask the OS for this information. This would involve an information maintenance system call. Memory dumps after errors and accounting information also involve these system calls.

### Communication

Messages between processes are exchanged in two ways, **message-passing** or **shared-memory**.

In message-passing, a common **mailbox** is essentially opened, be it a process on the same system or a process on a different system, communicating through a network.

In the shared-memory model, regions of memory are owned by **both processes**. This involves bypassing the operating systems insistence on protecting the memory used by one process from other processes, by both processes agreeing to lift this restriction. At this point, the OS lets them be. The processes are on their own and need to ensure things like a common data format and not writing to the same location simultaneously.

The prior method is easier with smaller amounts of data and for intercomputer communication, while the latter is faster. **Communication** system calls deal with system calls related to these actions.

### Protection

**Protection** system calls involve reading and writing **permissions** for users and processes to access specific resources.

## 2.5 System Programs

System programs, or system utilities, provide a convenient environment for program development and execution. Some are just user interfaces to system calls, while others are more complex. Users see the OS through these programs, along with the application programs. They do not see system calls.

* **File Management** – Programs to create, delete, copy, rename, print, dump, list and generally manipulate files and directories.
* **Status Information** – Some just ask the OS for basic information like the date or time, while others are more complex, providing performance, logging and debugging information. Usually, these programs present the information in a terminal or a GUI window or output it to a file. Some systems also keep a **registry**, which stores configuration information.
* **File Modification** – There may be several text editors to modify the contents of files, search through them or transform them in different ways.
* **Programming Language Support** – Compilers, assemblers, debuggers and interpreters for common programming languages are often provided by the OS or available for download.
* **Program Loading and Execution** – There are programs that load user programs into memory and execute them.
* **Communications** – Specific programs are used to create virtual connections between processes, users and computer systems to allow sending messages between users, browsing web pages, using email, to log in remotely or to transfer files.
* **Background Services** – There are certain processes that are constantly running while the computer is running. These are called services, subsystems or daemons. For example, there could be a service that constantly listens to attempts to access the system via a network.

Most operating systems also have several **application programs** that can provide solutions to general tasks, such as Web Browsers, Word Processors, etc.

## 2.6 Operating System Design and Implementation

### Design Goals

The first step to designing an OS is to define the **goals** and **specifications**. At the highest level, this depends on what **hardware** we use and the **type** of system we choose to make, batch, time-sharing, single-user, multi-user, distributed, real-time or general purpose.

Beyond this though, requirements are harder to define. We can still divide requirements into two parts, **user goals** and **system goals**.

User goals define what end-users want from the system. This includes things like convenience, speed, safety, reliability, etc. These requirements do not help with system design due to how vague they are. There are multiple ways to achieve each of these requirements and things like what qualifies as ‘fast’ are debatable.

A similar set of requirements can be found from those who will create and maintain the system. The system should be easy to design and maintain, flexible, error-free, efficient, etc. Again, the requirements are vague.

All of this proves that there is no way to decide upon the requirements. This can be seen in the large range of systems in existence that solve common problems in a variety of ways. Large, multiuser, multi-access systems used on mainframes are wildly different from the average Windows OS. Some general principles on design exist in the field of software engineering, but overall, the process is highly creative.

## 2.7 Operating System Structure

It should be obvious by now that a modern OS is very large and complex. Such a system needs to be built carefully to make sure it functions properly and can be modified easily. A common approach to doing this is to partition the OS into small components, or **modules**.

Breaking down the OS into modules allows for **greater control** over the computer as well as more **freedom** in changing the inner workings of each module. The overall functionality and features can be determined and divided into components, and each of those components can be implemented however the programmer sees fit. As long as the external interface is uniform and the module does what it is meant to, the inner workings are no one else’s concern. Essentially, each module can be a well-defined portion of the system with carefully defined inputs, outputs and functions.

### Layered Approach

Under the **layered approach**, the OS is divided into layers. The exact number varies, but the bottom layer is **hardware** and the top-layer is the **user interface**. Each layer consists of some **data structures** and a set of **routines** to operate on the data structures. Higher layers can invoke operations on lower layers without knowing how those operations work.

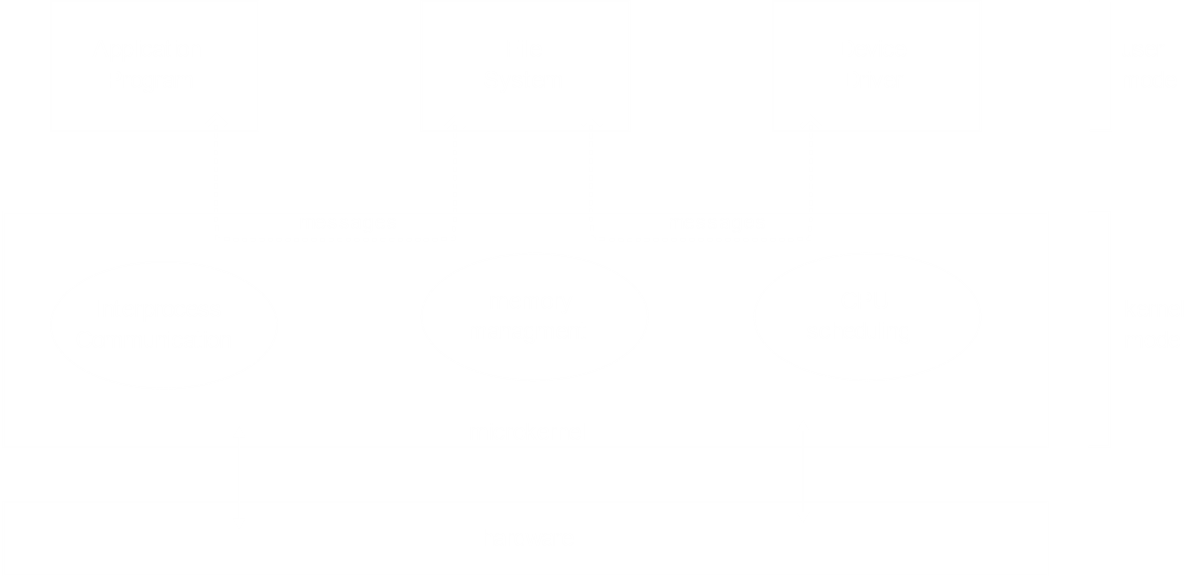
The main advantage to this approach is the **simplicity** of construction and debugging. Each layer uses only the services of the layers below it, so if we start debugging from the bottom, any error at a given layer is guaranteed to be caused at that layer, since everything below it has already been debugged and proven to be working properly.

The major issue to this approach is that it requires **careful planning**. The order in which we place layers is important and can be difficult to figure out. Another issue is that this approach is relatively **less efficient**. Every action by the user needs to go down to the hardware level layer by layer. At each layer, functions are called, data is manipulated and parameters are passed. This causes a significant amount of overhead.

### Microkernels

As UNIX expanded, the kernel became large and difficult to manage. To solve this, an OS called Mach was created that modularized the kernel using the **microkernel** approach.

In this method, all **non-essential components** are removed from the kernel and implemented as system and user-level programs. Exactly which components should remain is slightly debatable, but generally, minimal process and memory management is kept, as well as a communication facility.



The main function of the microkernel is actually to allow the client program and the various services running in the user space to communicate. This is achieved through **message passing**. The client program and the service never interact directly. Unfortunately, this system-function overheadcan affect **performance**.

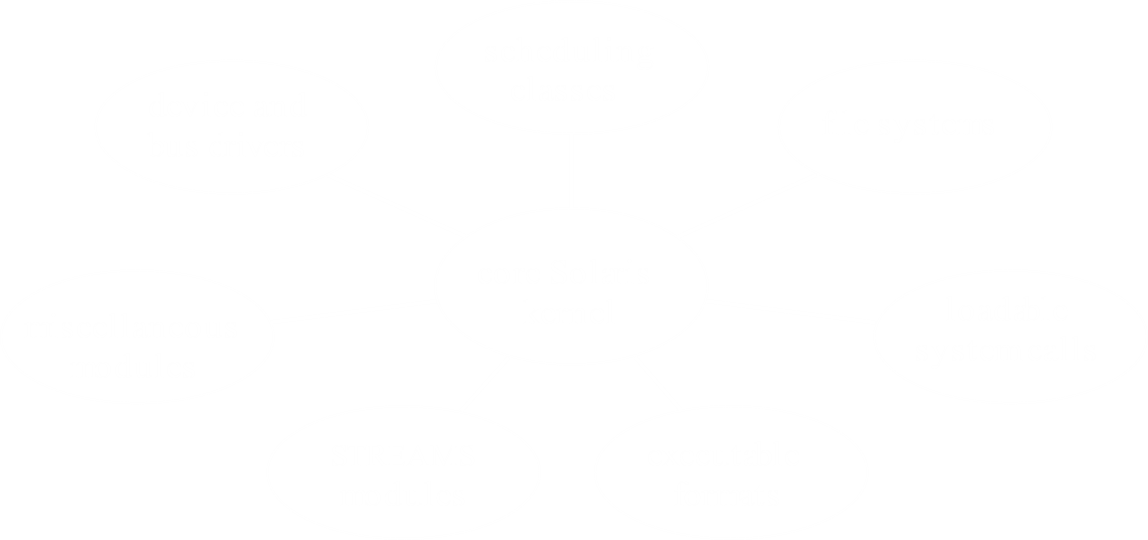
Benefits to this approach include:

* **Extending the OS** is easier. New services are just added to the user space without modifying the actual kernel. On the rare occasion that the kernel is modified, there are fewer changes to be made.
* The OS created is **easier to port** from one hardware design to another.
* **Security** is increased, since most services are not running on the kernel.
* **Reliability** is increased, since if one service fails, the OS can continue operating.

### Modules

The best methodology for OS design is to use **loadable kernel modules**. The kernel has a set of core components and loads in additional services via modules, either during boot time or during run time.

The idea is that non-core services can be implemented **dynamically** using the modules as the kernel is running. This is preferable to adding features to the kernel directly, which would require recompiling the kernel every time a change was made. The overall result is similar to a **layered approach**, in that each kernel section has defined, protected interfaces, but more flexible than a layered approach. The feature of the **microkernel** approach, where only the core functions are kept in the kernel, is also present, while the overhead of message passing for communication is removed.



The diagram above shows the approach taken by the Solaris OS.

## 2.9 Operating System Generation

It is possible to design, code and implement an OS for just one specific machine. More commonly though, operating systems are designed to run on any of a class of machines at different places with different peripheral configurations. Consider how the Windows OS needs to be able to operate on thousands of different configurations of computers. In this case, the OS must be generated for each computer when it is being installed onto the computer. This process is called **system generation**, or **SYSGEN**.

SYSGEN is a special program. It reads from a given file, or asks the operator of the system, or directly probes hardware to determine what **components** are present in the system. It needs answers to a range of questions such as:

* What CPU is to be used?
* How will the boot disk be formatted?
* How much memory is available?
* What devices are available?
* What operating-system options are desired?

Once this information is determined, it can be used in several ways.

* The system administrator may choose to use it to **modify** the OS to their preference. This causes the OS to be completely recompiled.
* Modules could be selected from a **precompiled** library and linked together to generate the OS. In this way, device drivers for all supported I/O devices are present in the library, but only those required are linked to the OS. This is faster, but the system may be overly general.
* It is possible to have all of the code constantly present, with selection occurring during execution. The actual system generation is very fast, since it just involves creating **tables** that describe the system.

The differences between these approaches are about size, generality and ease of modification when hardware is changed.

## 2.10 System Boot

Once an OS is generated, it still needs to be made available for use by the hardware. The process of loading the kernel is called **booting** the system. On most computers, this is done by a small program called the **bootstrap program** or the **bootstrap loader**. It locates the kernel and loads it into the main memory. On some computers, like PCs, this is a two-step process, where a simple bootstrap program loads a more complex one from disk, which in turn loads the kernel.

When a CPU is **reset**, either by powering it up or by restarting it, the instruction register is set to a **predefined memory location** from where execution starts. The initial bootstrap program is at that location.

The program itself is on **ROM**, which is convenient since it does not need to be initialized and cannot be infected by viruses. The program does a variety of tasks, from performing diagnostics to initializing different parts of the system. Finally, it loads the OS.

Some systems, like phones, store the **entire OS** on ROM. This is only suitable for a small OS with simple hardware. A problem with this approach is to change to OS, we would need to change ROM **hardware chips**. A workaround is to use **EPROM**.

All forms of ROM are called **firmware**, since they are a mixture of hardware and software. A general problem with firmware is that it is **slower** to execute code there than it is on RAM. Some systems get around this by keeping a copy of the OS on RAM as well. A final issue is that it is relatively **expensive**.

For large operating systems, just the bootstrap loader is stored in firmware while the actual OS is stored on **disk**. Once the bootstrap has run diagnostics, it reads a single block at a **fixed location** from disk. This block is called the **boot block**. It is loaded onto memory and executed.

The program in the boot block could be sophisticated enough to load the entire OS into memory and begin execution. More commonly, it is a simple program that just knows the address on disk and the length of the rest of the bootstrap program. Once the entire bootstrap program is loaded, it can find the OS kernel, load it to memory and start execution. At this point, we can say that the system is ‘running’.

All of the disk-bound bootstrap and the OS itself can be easily changed by writing a new version to disk. A disk that has a boot partition is called a **boot disk** or a **system disk**.