Operating System Structure

50.005 Computer System Engineering

Materials taken from SGG: Ch. 2.1-2.6, 2.7.1-2.7.3, 2.9.2

Operating System as a Service

In the previous week, we learned an introduction to the roles of an operating system kernel. Kernel is just one part of an operating system. The entire operating system software itself is a much bigger one, and **it provides various apps and functionalities to help users use the computer system.**

We also learned that user programs can make **system calls** whenever it needs help from the Kernel to access the hardware or I/O devices.

In other words, the OS provides services for user programs — the goals of the OS that we learned last week are made with the purpose of allowing users to use the computer system in an easier and more efficient manners.

Below are the brief list of operating system services that are usually provided to the users:

1. Operating system **interface** : terminal and GUI — they provide **access** to other OS services listed below.

2. Basic **support** for computer system usage:
    a. **Program execution:** The system must be able to load a program into memory and to run that program upon request. The program must be able to end its execution, either normally or abnormally (indicating error).

    b. **I/O Operations:** Being able to interrupt programs and manage asynchronous I/O requests.

    c. **File-system manipulation:** programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some programs include permissions management to allow or deny access to files or directories based on file ownership.

    d. **Process communication:** Processes run in virtual environment. Communications may be implemented via shared memory or through message passing, in which packets of information are moved between processes by the operating system.

    e. **Error detection:** The operating system needs to be constantly aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices, etc. For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

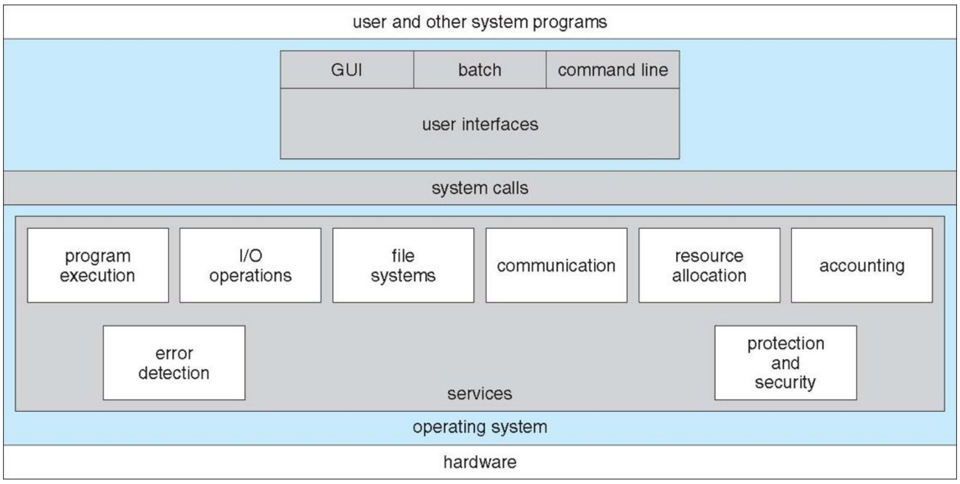
3. **Handlers** for **system calls** relating to six major categories that are related to the points in (2): **process control, file manipulation, device manipulation, information maintenance, communications**, and **protection**.

4. Computer usage **efficiency**:
    a. **Resources sharing** matter: When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system: CPU cycles, main memory, file storage, I/O device routines

    b. **Resource accounting**: This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.

5. **Protection and security against external threats**: Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. It extends to **defending** external I/O devices, including **modems and network adapters**, from invalid access attempts and to **record** all such connections for detection of break-ins.

An overview of OS services is illustrated in the diagram below:



Users can utilise the OS services in two general ways:

1. **Using the Operating system GUI or CLI (User Interface)**
2. **By writing code and making system calls within the code (Programming Interface)**

Operating System User Interface

The OS interface gives users a **convenient access** to various OS services. They are programs that can execute specialised commands and help users perform appropriate **system calls** in order to navigate and utilise the computer system.

There are two general ways for users to conveniently access OS services:

1. Using Graphical User Interface (**GUI**)
2. Using Command Line Interface (**CLI**)

The OS GUI

The OS GUI is what we usually call as our "home screen" or "desktop". It characterise the ***feel*** and ***look*** of an operating system. We use our mouse and keyboard everyday to interact with the OS GUI and make various **system calls**:

1. Opening or closing an app
2. File creation or deletion
3. Get attached device input or output
4. Install new programs, etc

When interacting with the OS through the GUI, users employ a mouse-based window-and-menu system characterized by a desktop metaphor:

- The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions.
- Depending on the mouse pointer's location, clicking a button on the mouse can launch a program,
- Select a file or directory—known as a folder—or pull down a menu that contains *commands*.

In fact, anything that task performed by the user that involves **resource allocation, memory management, access to I/O and hardware,** and **system security** requires system calls. Operations to perform various system calls are made easier with the OS interface and more convenient with the OS GUI. In other words, the GUI is made such that general-purpose computers are **user friendly.**

The Command Line Interface

The OS CLI is what we usually know as the "terminal" or "command line".

A command-line interface (CLI) is a **means of interacting with a computer program** where the user issues successive commands to the program in the form of **text**. The program which handles this interface feature is called a command-line interpreter.

Command Line Interpreter

In UNIX systems, the particular program that acts as the interpreters of these commands are known as shells[1]. Users typically interact with a Unix shell via a terminal emulator, or by directly writing a shell script that contains a bunch of successive commands to be executed.

For a system that comes with multiple command line interpreters, a user may choose among several different shells, including the Bourne shell, C-shell, Bourne-Again shell, Korn shell, and others. Common shells that we may have encountered:

- Bourne-Again shell (bash): written as part of the GNU Project to provide a superset of Bourne Shell functionality. This shell can be found installed and is the default interactive shell for users on most Linux and macOS systems.
- Z shell (zsh) is a relatively modern shell that is backward compatible with bash. It's the default shell in macOS since 10.15 Catalina.
- PowerShell – An object-oriented shell developed originally for Windows OS and now available to macOS and Linux.

In short, the shell primarily interprets a command from user and executes it:

1. Sometimes these commands are built-in — meaning that the command interpreter itself contains the code to execute the command.
   - For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call.
   - In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.
2. An alternative approach—used by UNIX, among other operating systems —**implements most commands through system programs**. In this case, the command interpreter does not understand the command in any way; it merely uses the command to **identify a file to be loaded into memory and be executed**. *In Unix systems, you can type `echo $PATH` on your terminal to find out possible places on where are these system programs.*

[1] The most generic sense of the term shell means any program that users employ to type commands. A shell hides the details of the underlying operating system and manages the technical details of the operating system kernel interface, which is the lowest-level, or "inner-most" component of most operating systems.

Terminal Emulator

A terminal emulator is a **text-based user interface (UI)** to provide easy access for the users to issue *commands*. Examples of **terminal emulator** that we may have encountered before are iTerm, MacOS terminal, Terminus, and Windows Terminal.

While the OS GUI seems more user friendly, sometimes it is more **efficient** to use the CLI that allows us to **give direct command entry.**

For example, if we want to **delete a file** using the OS GUI, we need to perform the following steps:

1. Hover our mouse to click folder after folder until we arrive at a final folder where the target file resides
2. Right click, press delete

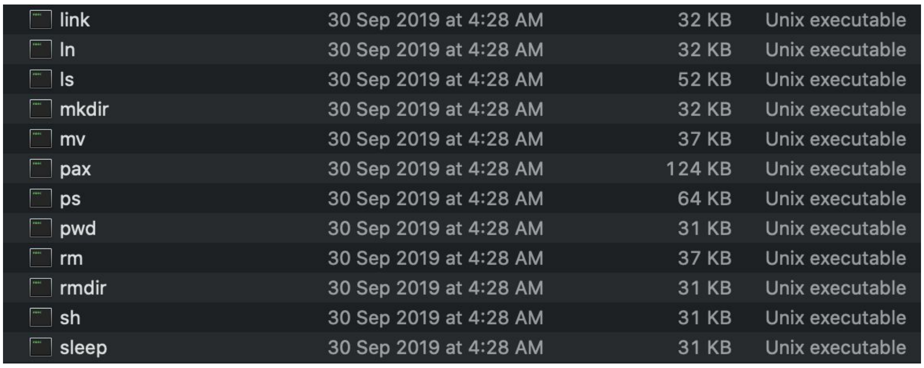
Equivalently, we can perform the same action using the CLI. In UNIX systems, we can write the following commands in our terminal emulator:

```
1. >>> cd {location}
2. >>> rm {filename.format}
```

How they work:

1. **The first line is implemented within the shell program itself (changing directory with `chdir` system call).**
2. The second line will tell the shell to **search for a system program called** `rm` and executes it with parameter of `<filename.format>`.
   - The function associated with the rm command (removing a file) would be defined completely **within the code in the program called `rm`.**
   - In this way, **programmers can add new commands to the system** easily by creating new files with the proper names.
   - The **command-interpreter program**, which can be **small**, does not have to be changed for new commands to be added.

A sample screenshot below shows various [system programs] (Unix executable type) that can be found at `./bin` that are invoked upon issuance of its corresponding *commands* through the terminal.

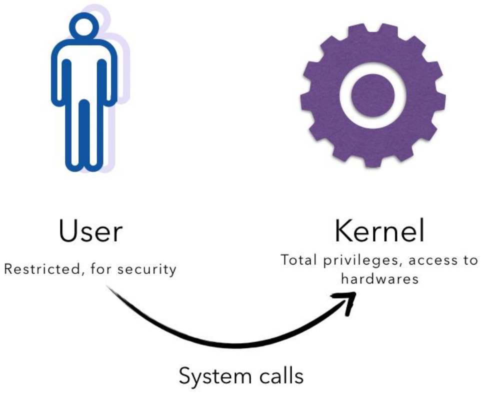


link	30 Sep 2019 at 4:28 AM	32 KB	Unix executable
ln	30 Sep 2019 at 4:28 AM	32 KB	Unix executable
ls	30 Sep 2019 at 4:28 AM	52 KB	Unix executable
mkdir	30 Sep 2019 at 4:28 AM	32 KB	Unix executable
mv	30 Sep 2019 at 4:28 AM	37 KB	Unix executable
pax	30 Sep 2019 at 4:28 AM	124 KB	Unix executable
ps	30 Sep 2019 at 4:28 AM	64 KB	Unix executable
pwd	30 Sep 2019 at 4:28 AM	31 KB	Unix executable
rm	30 Sep 2019 at 4:28 AM	37 KB	Unix executable
rmdir	30 Sep 2019 at 4:28 AM	31 KB	Unix executable
sh	30 Sep 2019 at 4:28 AM	31 KB	Unix executable
sleep	30 Sep 2019 at 4:28 AM	31 KB	Unix executable

Recall that **system programs** are simply programs that come with the OS to help users use the computer. They are run in **user mode** and will help users make the appropriate **system calls** based on the tasks given by the users.

Alternatively, we can access OS services using its programming interface (system calls) — meaning that we develop our own programs that rely on OS services to utilise the computer system's hardware and I/O devices.

System Calls



System calls are programming interface provided by the OS Kernel for users to **access kernel services.** When application programs make system calls, its execution is temporarily suspended and the Kernel mode takes over. **System calls are the only entry point to the kernel system,** meaning that we **cannot** make our program such that our PC directly executes arbitrary part of the kernel code.

This is usually prevented by :

1. Hardware, i.e: PC cannot perform JMP to code with raw RAM addresses starting with MSB of '1'
2. Virtualisation: the address space that the program is in is not the real address space.

System calls are mostly accessed by programs through APIs (application program interface), although we can certainly make system calls directly in assembly.

System calls through API

One of the most common ways to make system calls is through an intermediary called API. We can write a program in a particular language and conveniently perform several system calls through the appropriate APIs as needed.

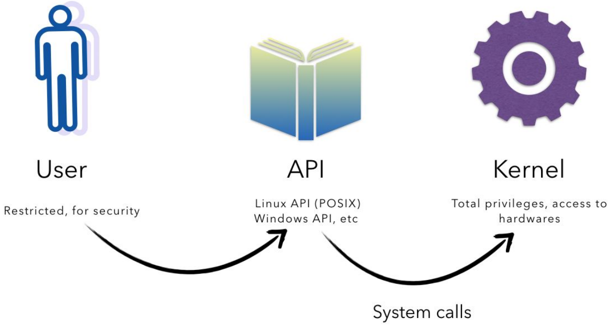
About API: API is a type of library that provides **wrapper functions** for the system calls, often named the same as the system calls they invoke. It specifies:

- a set of functions that are available to an application programmer
- the parameters that are passed to each function and
- the return values the programmer can expect

3 most common APIs available:

1. Win32 API for Windows systems -- written in C++
2. POSIX API for POSIX-based system (all versions of UNIX) -- mostly written in C. You can find the functions supported by the API here https://pubs.opengroup.org/onlinepubs/9699919799/
3. Java API for programs running on Java Virtual Machine (JVM)

Behind the scenes, the functions that make up an API **invoke the actual system calls** on **behalf** of the application programmer.



Benefits of APIs:

- It adds another layer of abstraction hence simplifies the process of application development[2]
- Supports program portability[3]

Note: Making the system call directly in the application code is possible, ***but more complicated*** *and may require embedded assembly code to be used (in C and C++) as well as knowledge of the low-level binary interface for the system call operation, which may be subject to change over time and thus not be part of the application binary interface;* ***the library functions are meant to abstract this away*****.**

To see more examples about *direct* system calls, you can find out more about Linux[4] system calls in http://man7.org/linux/man-pages/man2/syscalls.2.html

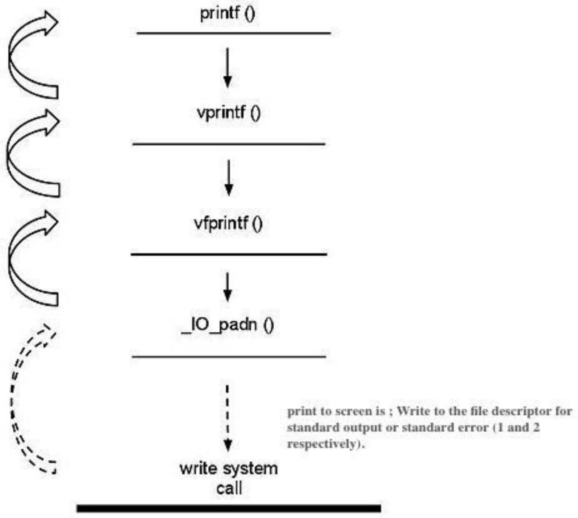
[2] Actual system calls can often be more detailed and difficult to work with than the API available to an application programmer.

[3] An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API (although in reality, architectural differences often make this more difficult than it may appear)

[4] Linux is a Unix clone written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX compliance.

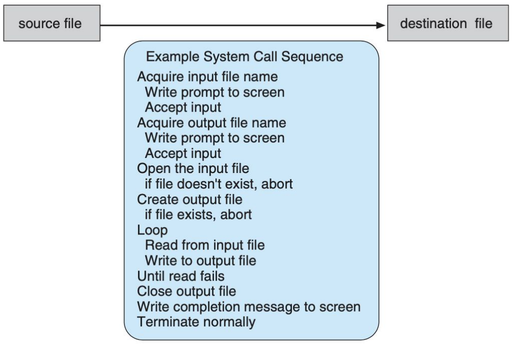
Example:

1. We always conveniently call `printf()` whenever we want to display our output to the console. `printf` itself is a system call API. This function requires kernel service as in involves access to hardware, in particular the display unit. The API is actually making several other function calls to ***prepare the required resources for this operation*** and finally make the *actual* system call that invokes the kernel's help to display the output to the display.



print to screen is ; Write to the file descriptor for standard output or standard error (1 and 2 respectively).

2. We also always conveniently copy one file into another location (be it programmatically or through the GUI). In Win32 API, this is supported by the `CopyFile`[5] function. The *actual* instruction **sequence** that is made by the API to complete the entire copy operation is actually pretty lengthy, involving **multiple** system calls (in this case alone system calls are made for writing to screen, opening files, obtaining file name, read from file, termination, etc).



```
Example System Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

[5] https://docs.microsoft.com/en-gb/windows/win32/api/winbase/nf-winbase-copyfile?redirectedfrom=MSDN

System Call Implementation

An API helps user make appropriate system calls by providing convenient wrapper functions. Here's the sequence of what happens in the nutshell after we invoke API functions involving system calls:

1. For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) provides a **system call interface** that serves as the **link** to **system calls made available by the operating system.**

2. The **system-call interface intercepts function calls in the API** and **invokes** the **necessary system calls** within the operating system.

3. Typically, **a number** is associated with **each system call**, and the system-call interface maintains a **table** indexed according to these numbers. For example, Linux system call tables and its associated numbers can be found here: https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl

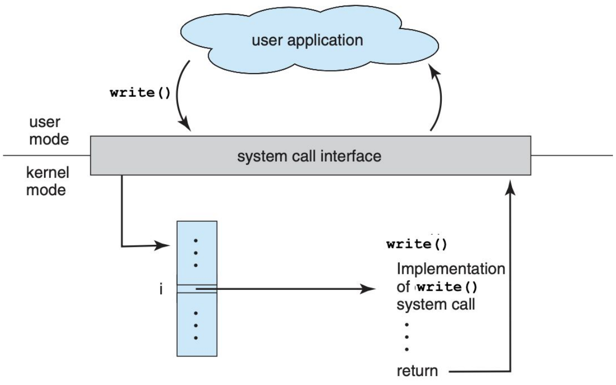
4. The system call interface then **invokes** the intended system call in the operating-system **kernel** by interrupting itself and invoking the appropriate handler:
   a. For example: if a system call has an id of X, the system call interface has to trap itself to the appropriate handler -- such as ILLOP handler and leave X onto R0.
   b. The ILLOP handler will examine R0 and extract X. It refers to the standard system call table and branch onto the address in the Kernel space that handles system call with id X.

5. Kernel mode takes over to handle this system call

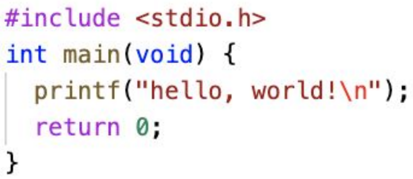
6. After the kernel **returns the status of the system call** and any requested **return** values, the system call interface resumes the program execution.

In a nutshell, the function calls **within the API** reached the system call, the execution of the user application is temporarily suspended, since a system call is essentially a **software interrupt.**

Note that the system call ids are HARDWARE DEPENDENT. It varies from systems to systems that utilise different hardwares. The relationship between application program, API, System call interface, and the kernel is shown below:

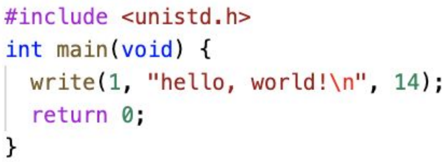


Example: making system call in C using API vs directly using assembly to print to screen



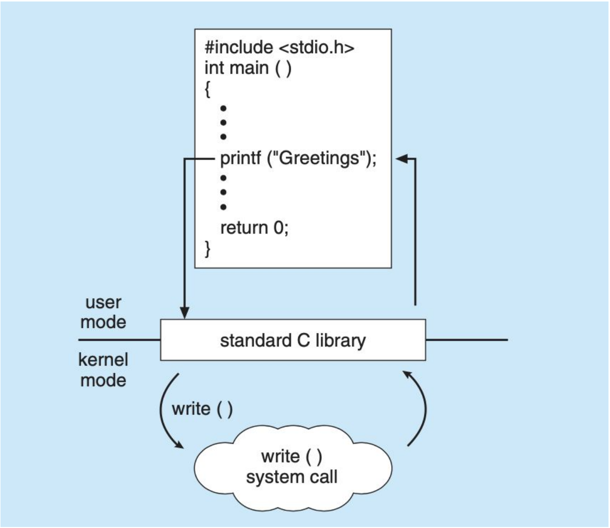
```
#include <stdio.h>
int main(void) {
  printf("hello, world!\n");
  return 0;
}
```

Typically we will do the above to print a simple hello, world!



```
#include <unistd.h>
int main(void) {
  write(1, "hello, world!\n", 14);
  return 0;
}
```

We can also do the same thing by using `write`. In fact, `write` is implemented within `printf` in C standard library as shown in the figure below. For obvious reasons, `printf` is more convenient because we don’t have to care about arguments like “1” and “14” and

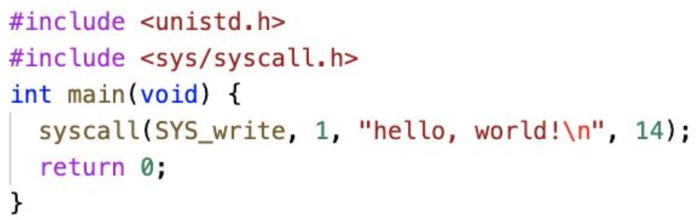


having to read the manual page for your hardware to find out what these means inside `write`.

But realise `write` here is just another C function call, not a system call. The figure in the previous page simply shows that a system call will be made within function `write`.

`write` is actually a convenient and more human-friendly wrapper around the system call `SYS_write`, and its implementation varies depending on the OS. The program above works on Linux and on macOS for this reason.

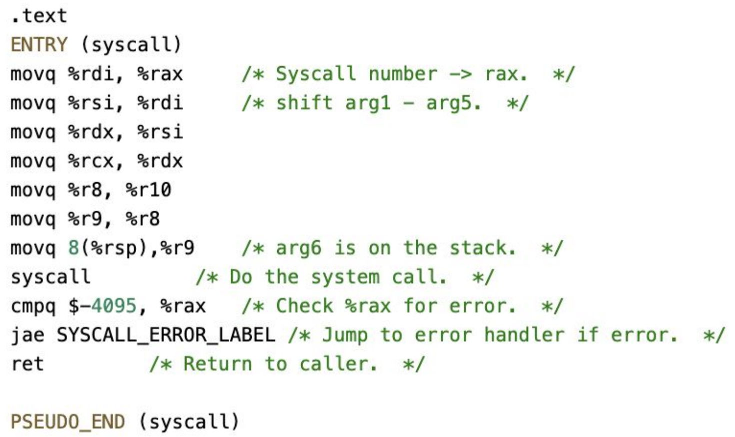
Many wrappers in APIs are named after the system call itself, just like `write` here that's meant to invoke `system call write`. This is the reason why many will not know whether they are making a direct system call or simply using the API to make a system call.



```c
#include <unistd.h>
#include <sys/syscall.h>
int main(void) {
  syscall(SYS_write, 1, "hello, world!\n", 14);
  return 0;
}
```

`write` is implemented by invoking another function called `syscall`. So we can do the same thing by calling `syscall` instead and **passing** the **appropriate parameters**. `SYS_write` here is actually the `id` of the system call that performs this write-to-console operation.

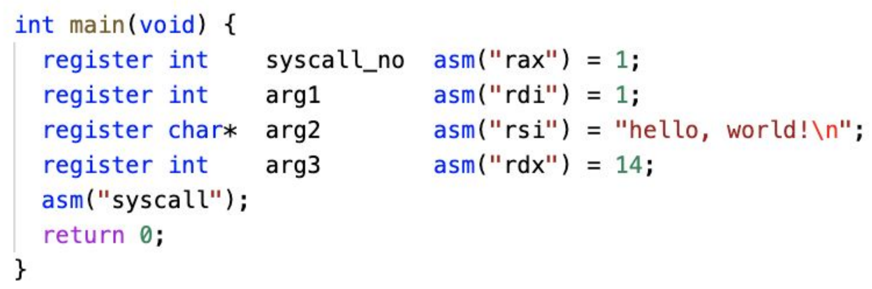
How is `syscall` implemented? `main` function puts its arguments in the right registers for the system call, and branch to `syscall` which is essentially composed of assembly instruction:



```
.text
ENTRY (syscall)
movq %rdi, %rax     /* Syscall number -> rax.  */
movq %rsi, %rdi     /* shift arg1 - arg5.  */
movq %rdx, %rsi
movq %rcx, %rdx
movq %r8, %r10
movq %r9, %r8
movq 8(%rsp),%r9    /* arg6 is on the stack.  */
syscall         /* Do the system call.  */
cmpq $-4095, %rax   /* Check %rax for error.  */
jae SYSCALL_ERROR_LABEL /* Jump to error handler if error.  */
ret         /* Return to caller.  */

PSEUDO_END (syscall)
```

In this light, you can completely make the system call for printing purpose **directly** using *C-inline assembly*. It is first done by putting the right arguments to the registers such as: `rax` -- the register to pass the system call id, `rdi` -- the register that should contain file handler (1 for `stdout` in your terminal), etc. **Note that these registers are machine specific.** These commands will work on 64-bit Linux run x86 architecture at the time of writing only. *It will compile on Mac systems but will not do anything since the registers are not correct for its hardware.*



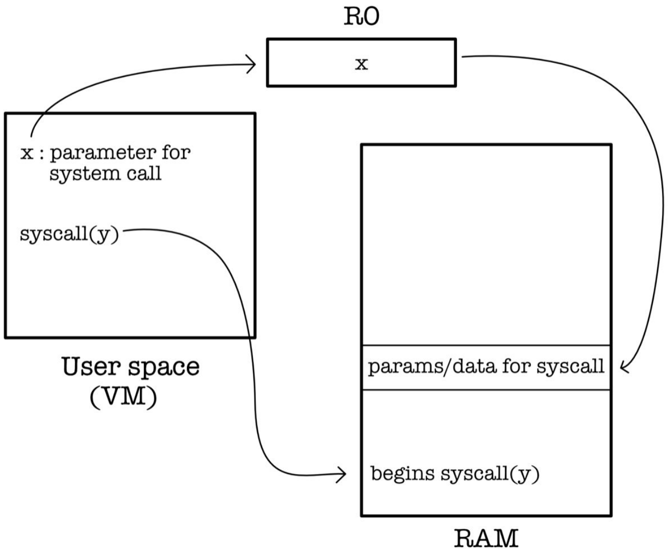
```
int main(void) {
  register int    syscall_no  asm("rax") = 1;
  register int    arg1        asm("rdi") = 1;
  register char*  arg2        asm("rsi") = "hello, world!\n";
  register int    arg3        asm("rdx") = 14;
  asm("syscall");
  return 0;
}
```

System Call Parameter Passing

System call is just like common function, implemented in the kernel space. It requires parameters. For example, if we request a `write`, one of the most obvious **parameters** required are **the strings to write**.

There are three general ways to pass the parameters required for system calls to the OS Kernel:

1. Pass parameters in Registers:
    - For the example of `write` system call, Kernel examines certain special registers for characters to print
    - **Pros**: Simple and fast access
    - **Cons**: There might be more parameters than registers
2. Push parameters to the program Stack:
    - Pushed to the stack by user program, then invoke `syscall(i)`
    - Kernel pops the arguments from the user program's stack
3. Pass parameters that are stored in a *block* or *table* in RAM and pass the pointer to the OS through registers:
    - As illustrated below, `x` represents the address of the parameters for the system call.
    - When system call `y` is made, the kernel examines certain registers, in this example is `R0` to obtain the address to the parameter
    - Given an address, Kernel can find the parameter for the system call in the RAM



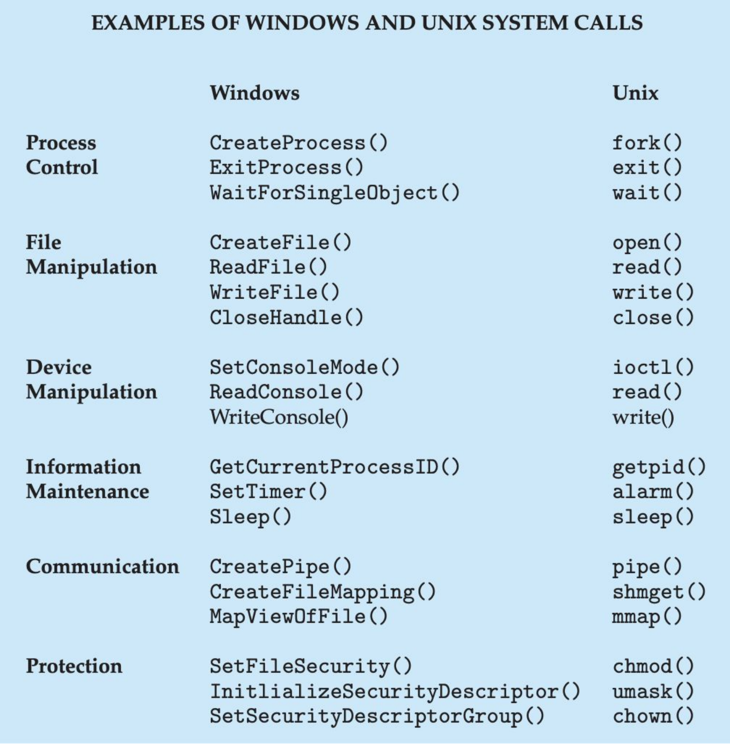
Types of System Calls

In general, each OS will provide a list of system calls that can be made. System calls can be grouped (but not limited to) roughly into six major categories:

1. **Process control:** end, abort, load, execute, create and terminate processes, get and set process attributes, wait for time, wait for event, signal event, allocate, and free memory
2. **File manipulation:** create, delete, rename, open, close, read, write, and reposition files, get, and set file attributes
3. **Device manipulation:** request and release device, read from, write to, and reposition device, get and set device attributes, logically attach or detach devices
4. **Information maintenance:** get or set time and date, get or set system data, get or set process, file, or device attributes
5. **Communications:** create and delete pipes, send or receive packets through network, transfer status information, attach or detach remote devices, etc
6. **Protection:** set network encryption, protocol

If you are curious about Linux-specific system calls, you can find the list here.

The screenshot below shows several examples of Windows and UNIX API functions that perform system calls:



EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix

Process Control	CreateProcess()
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

File Manipulation	CreateFile()
	ReadFile()
	WriteFile()
	CloseHandle()
		open()
		read()
		write()
		close()

Device Manipulation	SetConsoleMode()
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

Information Maintenance	GetCurrentProcessID()
	SetTimer()
	Sleep()
		getpid()
		alarm()
		sleep()

Communication	CreatePipe()
	CreateFileMapping()
	MapViewOfFile()
		pipe()
		shmget()
		mmap()

Protection	SetFileSecurity()
	InitlializeSecurityDescriptor()
	SetSecurityDescriptorGroup()
		chmod()
		umask()
		chown()

Process Control

In this section we choose to explain process control type of system calls with a little bit more depth.

1. **System calls to end or abort a program:**

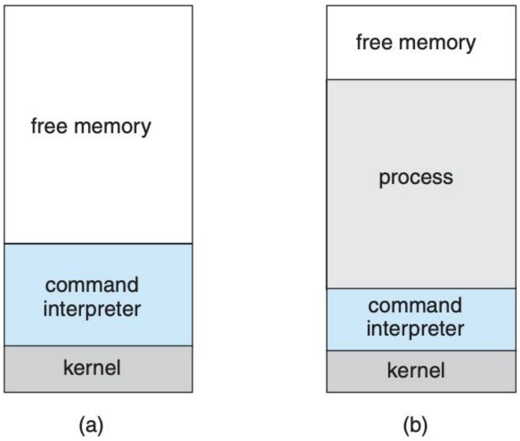
   A running process can either terminate **normally** (end) or **abruptly** (abort).
   If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a **debugger** -- a type of system program. It is assumed that the user will issue an appropriate command to respond to any error.

2. **System calls to load and execute another program and to support process communication:**

   It is possible for a user program to **call upon the execution of another user program**, such as creating background processes, etc. Having created new jobs or processes, we may need to **wait** for them to finish their execution. We may want to wait for a certain amount of time to pass `(wait time)`; more probably, we will want to wait for a specific event to occur `(wait event)`. The jobs or processes should then signal when that event has occurred `(signal event)`. Quite often, two or more processes share data and multiple processes need to communicate. All these features to `wait, signal event`, and communicate are done by making system calls.

There are so many facets of and variations in process and job control that we need to clarify using examples:

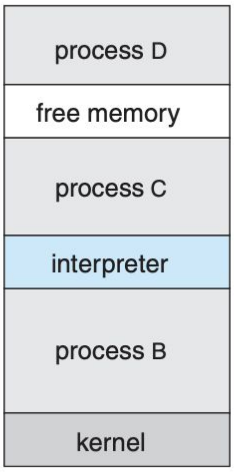
1. Single-tasking system (e.g: MS-DOS):



- The MS-DOS operating system is an example of a **single-tasking system**.

- It has a **command interpreter** (recall: part of OS interface) that is invoked when the computer is started as shown in the figure above, labeled as (a).
- Upon opening a new program, it **loads** the program into memory, **writing over most of itself (note the shrinking portion of command interpreter codebase)** to give the program as much memory as possible as shown in (b) above.
- Next, it sets the instruction pointer to the first instruction of the program.
  - The program then runs, and either an error causes a trap, or the program executes a **system call** to terminate.
  - In either case, the error code is saved in the system memory for later use.
- Following this action, the small portion of the command interpreter that was not overwritten resumes execution:
  - Its first task is to **reload the rest of the command interpreter** from **disk**.
  - Then the command interpreter makes the previous error code available to the user or to the next program
  - It stands by for more input command from the user

2. Multi-tasking system (e.g: FreeBSD)



The FreeBSD operating system is a multi-tasking OS that is able to create and manage multiple processes at a time:

- When a user logs on to the system, the shell of the user's choice is run.
- This shell is similar to the MS-DOS shell in that it **accepts** commands and **executes** programs that the user requests.
- However, since FreeBSD is a multitasking system, the command interpreter may **continue running** while another program is executed:
  - The possible state of a RAM with FreeBSD OS is as shown in the figure above
  - To start a new process, the shell executes a `fork()` system call.

    - Then, the selected program is loaded into memory via an `exec()`[6] system call, and the program is executed normally until it executes `exit()` system call to end normally or `abort` system call.
- Depending on the way the command was issued, the shell then either **waits** for the process to **finish** or **runs** the process **"in the background."**
    - In the latter case, the shell immediately requests *another command.*

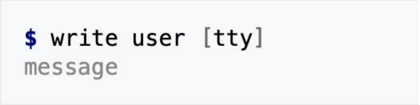
[6] We will learn about these system calls in the latter weeks and in lab

System Programs

Apart from the Kernel and the user interface (GUI and/or CLI), a modern operating system also comes with **system programs**. Most users' view of an operating system is defined by the **system programs**, not the actual system calls *because they are actually hidden from us (through API)*.

System programs, also known as **system utilities, provide a convenient environment for program development and execution**:

1. They are **basic tools** used by many users for common low-level activities.
2. It runs on **user mode, just like any other user-level applications.** When it requires kernel services, they **make system calls** just like any other user programs.
3. **These tools are very generic,** thus can be considered as part of the "system" instead of individual user apps that we typically install
4. *Note that sometimes system programs and system calls have the **same name**, but they are nowhere the same. For example:*
    a. `write` as a **command** that can be typed on the terminal (type `man write` to find out what these arguments are.)[7]

    ```
    $ write user [tty]
    message
    ```

    b. `write` **system call** as we have seen above

Like system calls, they can also be divided into the following categories:

- **File management:**
    - These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
    - For example: *all commands that you can enter in CLI that involves file management in UNIX systems is actually the **name** of system programs*

- **Status information:**
    - Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information.
    - Others are more complex, providing detailed performance, logging, and debugging information.
    - Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.

[7] `tty` itself is a command in Unix and Unix-like operating systems to print the file name of the terminal connected to standard input. If you open multiple terminal windows in your UNIX-based system and type `tty` on each of them, you will be returned with different ids. You may use write to communicate across terminal windows

- **File modification:**
    - Several text editors may be available to create and modify the content of files stored on disk or other storage devices.
    - There may also be special commands to search the contents of files or perform transformations of the text.

- **Programming-language support:**
    - **Compilers**, **assemblers**, **debuggers**, and **interpreters** for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download.

- **Program loading and execution:**
    - Once a program is assembled or compiled, it must be loaded into memory to be executed.
    - The system may provide **absolute loaders**, **relocatable loaders**, **linkage** editors, and **overlay loaders**.
    - **Runtime debugging systems** for either higher-level languages or machine language are needed as well.

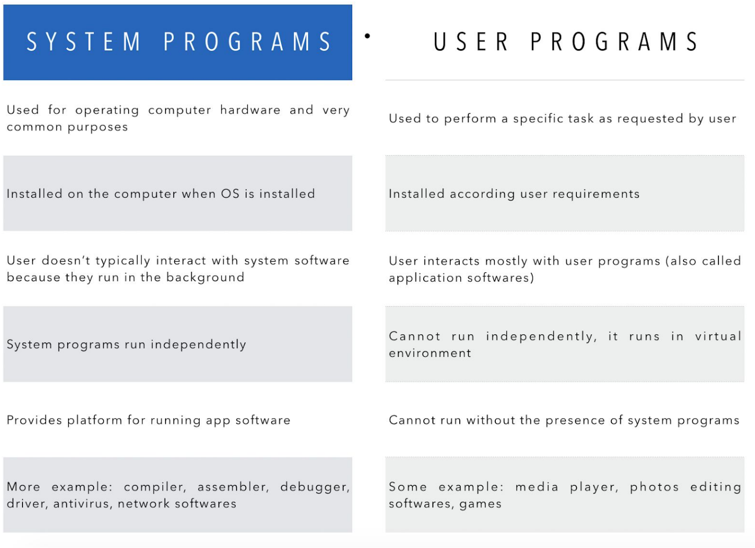
- **Communications:**
    - These programs provide the mechanism for **creating virtual connections** among processes, users, and computer systems.
    - They allow users to send messages to one another's screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.
    - *For example: ssh, pipe*

- **Background services:**
    - All general-purpose systems have methods for launching certain **system-program processes at boot time (upon startup)**: network-related system programs, some device drivers (although there are drivers that run in kernel mode, these are *not* system programs), etc
    - Constantly running system-program processes are known as services, subsystems, or **daemons**. One example is the **network daemon**:
        - A system needed a service to listen for network connections in order to connect those requests to the correct processes.
    - Other daemon examples include:
        - The `init` process (systemd in Linux)
        - Process schedulers that start processes according to a specified schedule,
        - System error monitoring services,
        - Print servers, etc
    - **Typical systems have dozens of daemons.** In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities.

Application Programs

Along with system programs, most operating systems are supplied with **programs** that are useful in solving common problems or performing common operations. Such application programs include **Web browsers**, **word processors** and **text formatters**, **spreadsheets, database systems, compilers, plotting and statistical-analysis.packages,** and **games.**

The table below summarises the differences between system programs and application programs (user programs)



SYSTEM PROGRAMS	USER PROGRAMS
Used for operating computer hardware and very common purposes	Used to perform a specific task as requested by user
Installed on the computer when OS is installed	Installed according user requirements
User doesn't typically interact with system software because they run in the background	User interacts mostly with user programs (also called application softwares)
System programs run independently	Cannot run independently, it runs in virtual environment
Provides platform for running app software	Cannot run without the presence of system programs
More example: compiler, assembler, debugger, driver, antivirus, network softwares	Some example: media player, photos editing softwares, games

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls. For example, when a user's computer is running the Mac operating system, the user might see the GUI, featuring a mouse-and-windows interface.

Alternatively, or even in one of the windows, the user might have a command-line UNIX shell. **Both use the same set of system calls, but the system calls look different and act in different ways.**

OS design and Implementation

There's no known ultimate solution when it comes to OS design. Internal structures of known operating systems can vary widely.

When one design an OS, it might be helpful to consider a few known things:

1. **Start by defining goals:**
    1. **User goals:** OS should be convenient to use, easy to learn, reliable, safe, fast
    2. **System goals:** The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient.
2. **Know the difference between policy and mechanism and separate them**:
    1. **Policy**: determines what will be done
    2. **Mechanism:** determines how to do something
    3. The separation of policy and mechanism is important for **flexibility**:
        - Policies are likely to change across places or over time.
        - In the worst case, each change in policy would require a change in the underlying mechanism.
        - **A general mechanism insensitive to changes in policy** would be more **desirable**. A change in policy would then require *redefinition of only certain parameters of the system*.
    4. For example: a mechanism for giving priority to certain types of programs over others. If the mechanism is properly separated from policy, it can be used either to
        - Support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones
        - Or support the opposite policy whenever appropriate.

Once an operating system is **designed**, it must be **implemented**. Because operating systems are *collections of many programs: kernel, system programs, interface, etc*, written by **many people over a long period of time**, it is difficult to make general statements about how they are implemented[8].

Example of OS structures

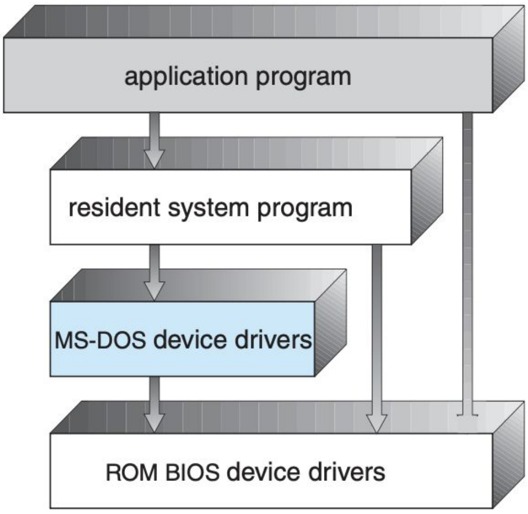
Simple (Monolithic) Structure

A monolithic kernel is an operating system architecture where the entire operating system is working in kernel space. There are only three components: user, kernel, and hardware.

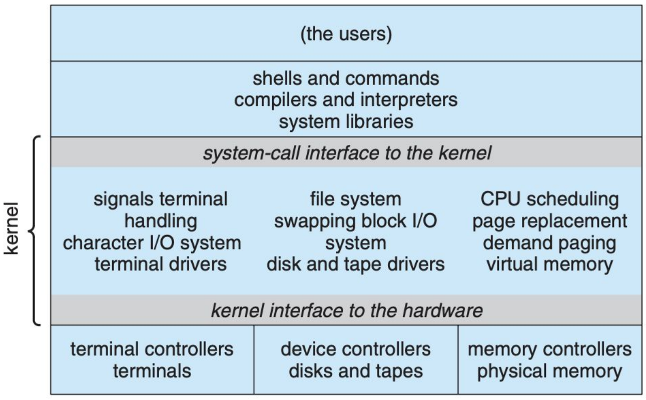
The figure below shows the structure of MS-DOS, one of the simplest OS made in the early years:

[8] Early operating systems were written in assembly language. Now, although some operating systems are still written in assembly language, most are written in a higher-level language such as C or an even higher-level language such as C++. **Actually, an operating system can be written in more than one language:** (1) The lowest levels of the kernel might be assembly language, and then (2) higher-level routines might be in C, and finally (3) system programs might be in C or C++, in interpreted scripting languages like PERL or Python, or in shell scripts. In fact, a given Linux distribution probably includes programs written in all of those languages.

- The interfaces and levels of functionality are not well separated (all programs can access the hardware)
- For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives.
- Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.



The early UNIX OS was also simple in its form as shown below. In a way, it is **layered** to a minimal extent with very simple structuring.



The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

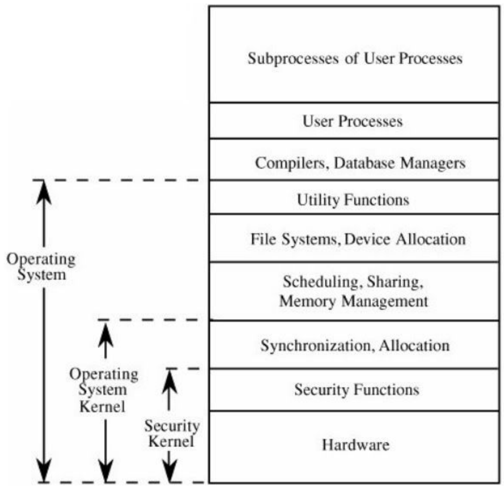
Taken in sum, that is an enormous amount of functionality to be combined into one level.

Pros: *distinct performance advantage* because there is very little overhead in the system call interface or in communication within the kernel.
Cons: *difficult to implement and maintain.*

Layered Approach

The operating system is broken into a many number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

Figure below shows a layered approach. The programs in layer N relies on services only from the layer below it.


Subprocesses of User Processes
User Processes
Compilers, Database Managers
Utility Functions
File Systems, Device Allocation
Scheduling, Sharing, Memory Management
Synchronization, Allocation
Security Functions
Hardware
Operating System
Operating System Kernel
Security Kernel

Pros -- **simplicity of construction and debugging**:

- Each layer is implemented only with operations provided by lower-level layers.
- A layer does not need to know how these operations are implemented; it needs to know only what these operations do.
- This abstracts and hides the existence of certain data structures, operations, and hardware from higher-level layers.

Cons:

- Appropriately defining the various layers, and **careful planning** is necessary.
    - If we are met with bugs in our program, we debug our program and not our compiler
    - We mostly assume that the layers beneath us are already *made correct*
    - Sometimes, this assumption is not always true and difficult to maintain with the growing size of the OS. Some OS is shipped with bugs on its lower layers that are very difficult for users to debug.
    - Patches and updates are periodically given to fixed these bugs.

- **They tend to be less efficient than other types.**
    - For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware.

- At each layer, the parameters may be modified, data may need to be passed, and so on.
- **Each layer adds overhead to the system call.**
- The net result is a system call that takes longer than does one on a non layered system.

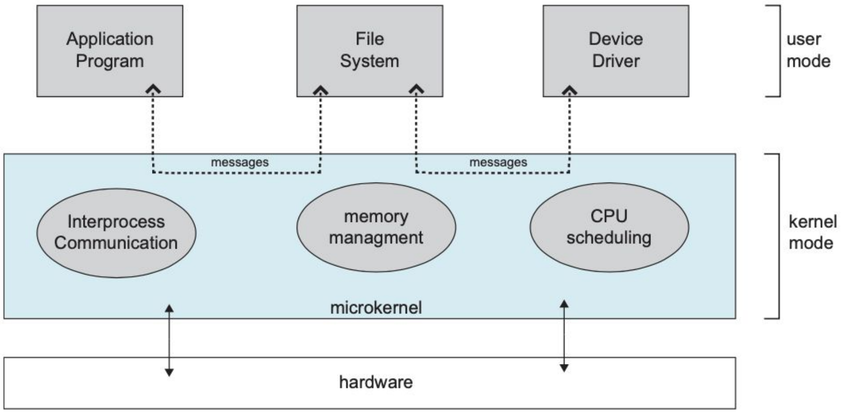
Example: most modern OS

Microkernel structures

Microkernel is a very small kernel that provide minimal process and memory management, in addition to a communication facility.

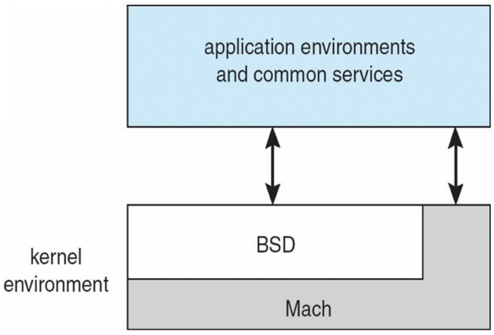
This method structures the operating system by r**emoving all nonessential** components from the kernel and **implementing them as system and user-level programs**. The result is a *smaller kernel*.

For example: if the client program wishes to access a file, it must interact with the file server. The client program and service **never interact directly**. Rather, they **communicate** indirectly by **exchanging messages with the microkernel** as illustrated below:



Pros: extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.
Cons: suffer in performance increased system-function overhead.

Example: Mach BSD, Windows NT (first release was a microkernel), Mac OSX (shown below) was partly based on microkernel approach



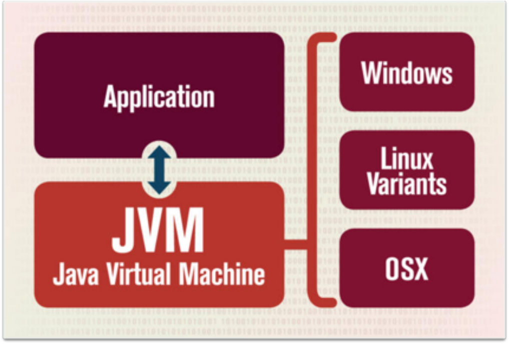
Java Operating System (JX)

The JX OS is written almost entirely in Java. Such a system, **known as a language-based extensible system,** and **runs in a single address space (no virtualisation, no MMU)**, as such it will face difficulties in maintaining memory protection that is usually supported by hardwares in typical OS.

Language-based systems instead rely on type-safety features of the language. As a result, language-based systems are desirable on small hardware devices, which may lack hardware features that provide memory protection.

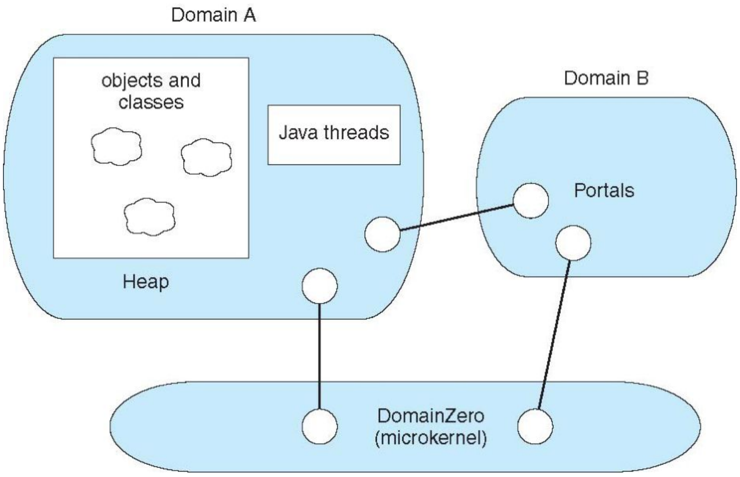
The architecture of the JX system is illustrated below:

- JX organizes its system according to **domains**.
- Each domain represents an independent **JVM (Java Virtual Machine)**:
  - JVM is an abstract virtual machine that can run on any OS
  - JVM provides **portable execution environment** for Java-based apps
  - It maintains a heap used for allocating memory during object creation and threads within itself, as well as for garbage collection.



- Domain zero is a **microkernel** responsible for *low-level details* such as system initialization and saving and restoring the state of the CPU.
  - Domain zero is written in C and assembly language; all other domains are written entirely in Java.
  - Communication between domains occurs through a specific mechanism called ***portals***

- **Protection** within and between domains **relies on the type safety** of the Java language. **However, since domain zero is not written in Java, it must be considered trusted.**


Domain A
objects and classes
Java threads
Heap
Domain B
Portals
DomainZero (microkernel)