UNIX is CUI operating system. Operating system is an interface between hardware and applications software’s. It serves as the operating system for all types of computers, including single-user personal computers and engineering workstations, multi-user microcomputers, mini-computers, mainframes, and super computers as well as special-purpose devices.

**History of UNIX: -**

**Multiuser: -** Multi user operating system means more than one user shares the same system resources (hard disk, memory, printer, application software etc., ) at the same time.

**Multi-tasking: -** Another highlight of UNIX is that it is Multitasking, implying that it can carry out more than one job at the same time. Depending on the priority the task, the operating system appropriately allots small time slots to each foreground and background task. Programming

**Facility: -** UNIX operating system provides shell. Shell works like a programming language. It provides commands and keywords. By running these two, user can prepare efficient program.

**Portability: -** One of the main reasons for the universal popularity of Unix is that it can be ported to almost any computer system, with only the bare minimum of adoptions to suit the given computer architecture.

**Communication: -** UNIX provides electronic mail. The communication may be within the network of a single main computer, or between two are more such computer networks. The user can easily exchange mail data, programs through such networks.

**Security: -** UNIX provides three levels of security to protect data. The first is provided by assigning passwords and login names to individual users ensuring that not anybody can come and have access to your work. At the file level, there are read, write, and execute permissions to each file which decide who can access a particular file, who can modify it and who can execute it. Lastly, there is file encryption. This utility encodes your file into an unreadable format, so that even if someone succeeds in opening it, your secrets are safe.

**Open System**: - The source code for the UNIX system and not just the executable cede, has been made available to users and programmers. Because of this many people have been able to adapt the UNIX system in different ways.

**System Calls**: - Programs interact with the kernel through approximately 100 system calls. System calls tell the kernel to carry out various tasks for the program, such as opening a file, writing a file, obtaining information about a file, executing a program, terminating a process, changing the priority of a process and getting the time of day. Different implementations of Unix system have compatible system calls with each call having the same functionality.

**System Calls:**

The most common system calls used on Unix system calls, Unix-like, and other POSIX-compliant operating systems are open, read, write, close, wait, exec, fork, exit, and kill.

A system call is a function that a user program uses to ask the operating system for a particular service. User programmers can communicate with the operating system to request its services using the interface that is created by a system call.

System calls serve as the interface between an operating system and a process. System calls can typically be found as assembly language instructions. They are also covered in the manuals that the programmers working at the assembly level use. When a process in user mode needs access to a resource, system calls are typically generated. The resource is then requested from the kernel via a system call.

**Important System Calls Used in OS**

**1. wait()**

In certain systems, a process must wait for another process to finish running before continuing. When a parent process creates a child process, the parent process's execution is suspended until the child process has finished running. With a wait() system call, the parent process is automatically suspended. Control returns to the parent process once the child process has completed its execution.

**2. fork()**

The fork system call in OS is used by processes to make copies of themselves. By using this system, the Call parent process can create a child process, and the parent process's execution will be halted while the child process runs.

**3. exec()**

When an executable file replaces an earlier executable file within the context of an active process, this system call is executed. The original process identifier is still present even though a new process is not created; instead, the new process replaces the old one's stack, data, head, data, etc.

**4. kill()**

The OS uses the kill() system call to urge processes to terminate by sending them a signal. A kill system call can have different meanings and is not always used to end a process.

**5. exit()**

An exit system call is used when the programme must be stopped. When the exit system call is used, the resources that the process was using were released.

**Why Do You Need System Calls in Operating System?**

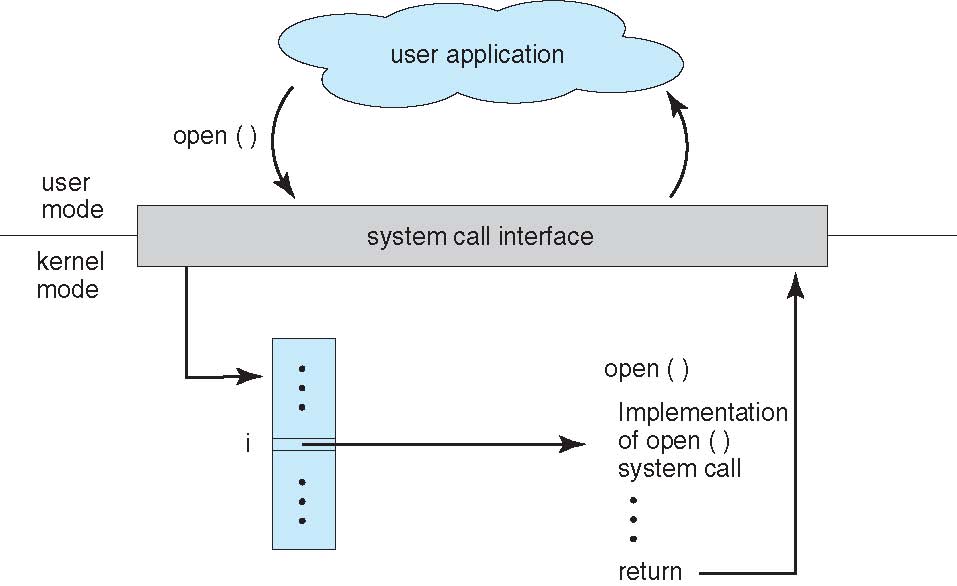
* System calls are necessary for reading and writing from files.
* System calls are necessary for a file system to add or remove files.
* New processes are created and managed using system calls.
* System calls are required for packet sending and receiving over network connections.
* A system call is required to access hardware devices like scanners and printers.

**System Call Implementation:**

* Typically, a number is associated with each system call
  + **System-call interface** maintains a table indexed according to these numbers
* The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
* The caller need not know a thing about how the system call is implemented
  + Just needs to obey the API and understand what the OS will do as a result call
  + Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

**System Call -- OS Relationship:**

The handling of a user application invoking the open() system call



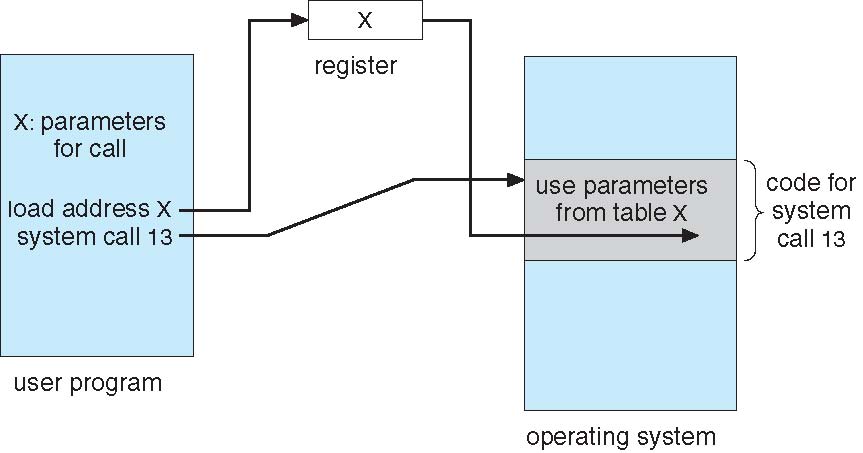
System Call Parameter Passing:

* Often, more information is required than simply identity of desired system call
  + Exact type and amount of information vary according to OS and call
* Three general methods used to pass parameters to the OS
  + Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  + Parameters stored in a block*,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  + Parameters placed, or **pushed***,* onto the **stack**by the program and **popped**off the stack by the operating system.
  + Block and stack methods do not limit the number or length of parameters being passed.

**Parameter Passing via Table:**

x points to a block of parameters. x is loaded into a register

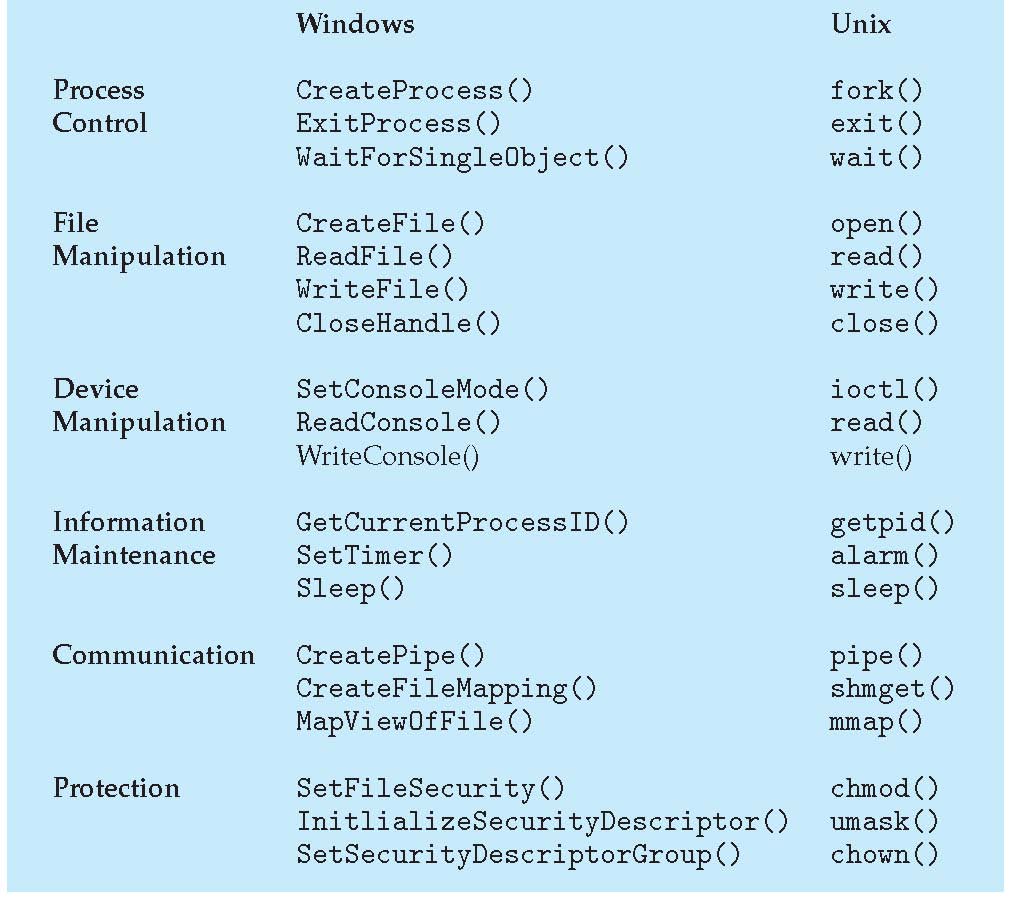
A system call is a way for a user program to interface with the operating system. The program requests several services, and the OS responds by invoking a series of system calls to satisfy the request. A system call can be written in assembly language or a high-level language like **C** or **Pascal**. System calls are predefined functions that the operating system may directly invoke if a high-level language is used.



**Types of System Calls:**

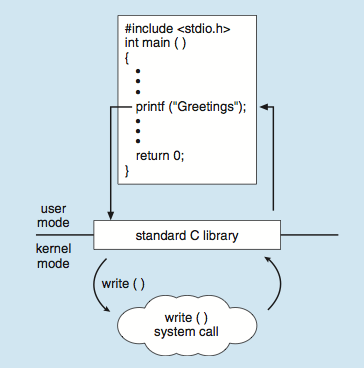
* **Process control**
  + create process, terminate process
  + end, abort
  + load, execute
  + get process attributes, set process attributes
  + wait for time
  + wait event, signal event
  + allocate and free memory
  + Dump memory if error
  + **Debugger** for determining **bugs, single step** execution
  + **Locks** for managing access to shared data between processes
* **File management**
  + create file, delete file
  + open, close file
  + read, write, reposition
  + get and set file attributes
* **Device management**
  + request device, release device
  + read, write, reposition
  + get device attributes, set device attributes
  + logically attach or detach devices
* **Information maintenance**
  + get time or date, set time or date
  + get system data, set system data
  + get and set process, file, or device attributes
* **Communications**
  + create, delete communication connection
  + send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  + **Shared-memory model** create and gain access to memory regions
  + transfer status information
  + attach and detach remote devices
* **Protection**
  + Control access to resources
  + Get and set permissions
  + Allow and deny user access

**Examples of Windows and Unix System Calls:**



**Example -- Standard C Library:**

C program invoking printf() library call, which calls write() system call



**File Structure Related System Calls:**

* The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices.
* To a process then, all input and output operations are synchronous and unbuffered.
* All input and output operations start by opening a file using either the "creat()" or "open()" system calls.
  + These calls return a file descriptor that identifies the I/O channel.

**File descriptors:**

* Each UNIX process has 20 file descriptors at it disposal, numbered 0 through 19.
* The first three are already opened when the process begins.
  + 0: The standard input
  + 1: The standard output
  + 2: The standard error output
* When the parent process forks a process, the child process inherits the file descriptors of the parent.

## **What is the Purpose of System Calls in OS?**

The purpose of system calls serves as the interface between an operating system and a process. System calls can typically be found as assembly language instructions. They are also covered in the manuals that the programmers working at the assembly level use. When a process in user mode needs access to a resource, system calls are typically generated. The resource is then requested from the kernel via a system call.

**System calls are required in in following circumstances:**

* If a file system needs files to be created or deleted. A system call is also necessary for reading and writing from files.
* development and administration of new procedures.
* System calls are also required for network connections. This also applies to packet sending and receiving.
* A system call is necessary to access hardware such as a printer, scanner, etc.

When a system call is made, the process transitions from user mode to kernel mode. Once the system call has finished running, the control is returned to the user mode process. Sending the kernel, a trap signal causes it to read the system call code from the register and carry out the system call. Process control, file management system calls in OS, device management, information maintenance, and communication are the main types of system calls in OS. Several crucial system calls used by our computer system include wait(), fork(), exec(), kill(), and exit().

How to provide the illusion of many CPUs?

* CPU virtualizing
  + The OS can promote the illusion that many virtual CPUs exist.
  + **Time sharing**: Running one process, then stopping it and running another
    - The potential cost is performance.

**A Process:**

A Process is a running program. A process is basically a program in execution. The execution of a process must progress in a sequential procedure.

* Comprising of a process:
  + Memory (address space)
    - Instructions
    - Data section
  + Registers
    - Program counter
    - Stack pointer

**Process API:**

* These APIs are available on any modern OS.
  + **Create**
    - Create a new process to run a program
  + **Destroy**
    - Halt a runaway process
  + **Wait**
    - Wait for a process to stop running
  + **Miscellaneous Control**
    - Some kind of method to suspend a process and then resume it
  + **Status**
    - Get some status info about a process

**Process Creation:**

1. **Load** a program code into memory, into the address space of the process.
   * Programs initially reside on disk in *executable format*.
   * OS perform the loading process lazily.
     + Loading pieces of code or data only as they are needed during program execution.
2. The program’s run-time **stack** is allocated.
   * Use the stack for *local variables*, *function parameters*, and *return address*.
   * Initialize the stack with arguments 🡪 argc and the argv array of main() function
3. The program’s **heap** is created.

Used for explicitly requested dynamically allocated data.

Program request such space by calling malloc() and free it by calling free().

1. The OS do some other initialization tasks.

input/output (I/O) setup

Each process by default has three open file descriptors.

Standard input, output and error

1. **Start the program** running at the entry point, namely main().

The OS *transfers control* of the CPU to the newly-created process.

Loading: From Program To Process:

Diagram

Description automatically generated

Process States:

* A process can be one of three states.
  + **Running**
    - A process is running on a processor.
  + **Ready**
    - A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
  + **Blocked**
    - A process has performed some kind of operation.
    - When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

**Process State Transition:**

Diagram

Description automatically generated

**Data structures:**

* The OS has some key data structures that track various relevant pieces of information.
  + **Process list**
    - Ready processes
    - Blocked processes
    - Current running process
  + **Register context**
  + PCB(Process Control Block)
  + A C-structure that contains information about each process.

**API (Application Programming Interface)** is used to establish connectivity among devices and applications. However, it is an interface which takes the requests from the user and informs the system about what must be done and returns the response back to the user.

Each process has a name; in most systems, that name is a number known as a **process ID (PID).**

Table

Description automatically generated

**What API does the OS provide to user programs?**

• API = Application Programming Interface

= functions available to write user programs

• API provided by OS is a set of “system calls” – System call is a function call into OS code that runs at

a higher privilege level of the CPU

– Sensitive operations (e.g., access to hardware) are

allowed only at a higher privilege level – Some “blocking” system calls cause the process to be

blocked and de-scheduled (e.g., read from disk)

**The fork() system call is used to create a new process,**

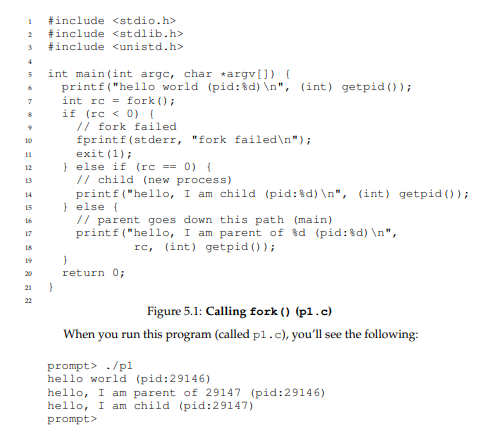
The fork() System Call:

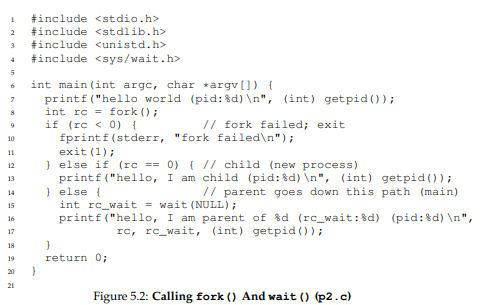
* Create a new process
  + The newly-created process has its own copy of the **address space**, **registers**, and **PC**.

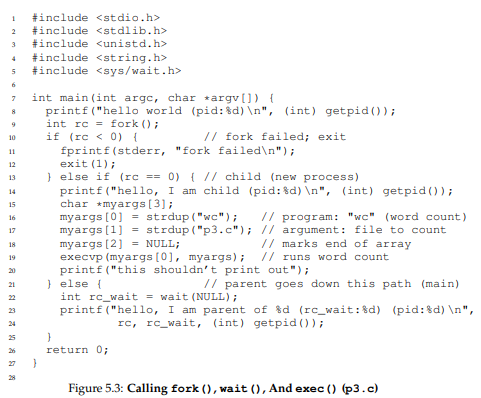
Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc (program counter), same CPU registers, same open files which use in the parent process.

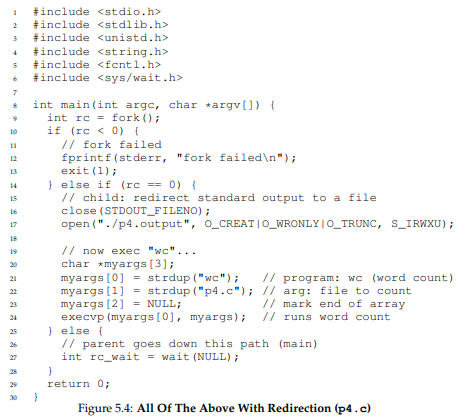
It takes no parameters and returns an integer value. Below are different values returned by fork().

***Negative Value*:** creation of a child process was unsuccessful.  
***Zero*:** Returned to the newly created child process.  
***Positive value*:** Returned to parent or caller. The value contains process ID of newly created child process.









**Text & Reference books:**

1. Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci- Dusseau, Arpaci-Dusseau Books, May, (2014).