CSE 344 System Programming

Introduction, POSIX and fundamental concepts

Spring 2021-2022

Erchan Aptoula

Institute of Information Technologies

Office: 253, eaptoula@gtu.edu.tr

Introduction

The aim of the course is to familiarize students with the system related calls of a (mostly) POSIX compliant operating system.

What can “system calls” do ?

- Talk to the Operating System (O/S)
- Access Hardware
- Perform low level programming

During this course we will

- Study the Standard C Library and how we can use it
- Study the standards for O/S level programming
- Study how the theory of O/Ss is applied to the practice of actual system programming

Programming interface

Programming interfaces allow applications to perform tasks such as:

• File I/O, creating and deleting files and directories, creating new processes, executing programs, setting timers, communicating between processes and threads on the same computer, and communicating between processes residing on different computers connected via a network.

This set of low-level interfaces is also known as the *system programming interface*

POSIX

What is POSIX?

It's the Portable Operating System Interface: a family of IEEE standards to ensure compatibility between operating systems.

Started in 1988, the latest version is from 2017.

POSIX-certified: AIX, HP-UX, macOS, and more

Formerly POSIX certified: Solaris, and more

Mostly POSIX compliant: most Linux distros, Android, and more

POSIX for MS Windows: Cygwin (mostly compliant), Windows Subsystem for Linux (WSL), and more.

The Core of the Operating System: The Kernel

The term *operating system* is commonly used with two different meanings:

- To denote the entire package consisting of the **central software managing a computer's resources** and all of the accompanying **standard software tools**, such as **command-line interpreters**, **graphical user interfaces**, **file utilities**, and **editors**.

- More narrowly, to refer to the central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).

The term ***kernel*** is often used as a synonym for the second meaning.

Tasks performed by the kernel

Among other things, the kernel performs the following tasks :

- Process scheduling
- Memory management
- Provision of a file system
- Creation and termination of processes
- Access to devices
- Networking
- Provision of an application programming interface (API) through system calls.

Kernel Tasks: Process Scheduling

A computer has one or more central processing units (CPUs), which execute the instructions of programs.

Most UNIX systems, are *preemptive multitasking* operating systems

- **Multitasking** means multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s).

- **Preemptive** means the rules governing which processes receive use of the CPU and for how long are determined by the kernel process scheduler (rather than by the processes themselves)

Kernel Tasks: Memory Management

Physical memory (RAM) is a limited resource that the kernel must share among processes in an equitable and efficient fashion.

Most modern operating systems, employ **virtual memory management**, a technique that confers two main advantages:

- Processes are **isolated from one another and from the kernel**, so that one process can't read or modify the memory of another process or the kernel.

- Only **part of a process needs to be kept in memory**, thereby lowering the memory requirements of each process and allowing more processes to be held in RAM simultaneously. This leads to better CPU utilization, since it increases the likelihood that, at any moment in time, there is at least one process that the CPU(s) can execute.

Kernel Tasks

Provision of a file system:

The kernel provides a file system on disk, allowing files to be created, retrieved, updated, deleted, and so on.

Creation and termination of processes:

The kernel can load a new program into memory, providing it with the resources (e.g., CPU, memory, and access to files) in order to run it. Such an instance of **a running program is termed as a "*process*"**.

Once a process has completed execution, the kernel ensures that the resources it uses are freed for subsequent reuse by later programs.

Kernel Tasks

Accessing to devices:

The devices attached to a computer allow communication of information between the computer and the outside world, permitting input / output.

The kernel provides programs with an **interface that standardizes and simplifies access to devices**, while at the same time arbitrating access by multiple processes to each device.

Networking:

The kernel transmits and receives network messages (packets) on behalf of user processes. This task includes routing of network packets to the target system.

Kernel Tasks

Multiuser operating systems provide users with the abstraction of a virtual private computer; that is, each user can log on to the system and operate largely independently of other users.

Users can run programs, (each gets a share of the CPU) and operates in its own virtual address space

All programs can independently access devices and transfer information over the network

The kernel resolves potential conflicts in accessing hardware resources, so users and processes are generally unaware of the conflicts.

Kernel mode and the user mode

Modern processor architectures typically allow the CPU to operate in at least two different modes: **user mode and kernel mode**. Hardware instructions allow switching from one mode to the other.

Correspondingly, **areas of virtual memory can be marked as being part of user space or kernel space**. When running in user mode, the CPU can access only memory that is marked as being in user space; attempts to access memory in kernel space result in a hardware exception. When running **in kernel mode, the CPU can access both user and kernel memory space**.

Certain operations can be performed only while the processor is operating in kernel mode.

Process versus kernel views of the system

A running system typically has numerous processes. For a process, many things happen asynchronously. An executing process **doesn't know when it will next time out**, which other processes will then be scheduled for the CPU (and in what order), or when it will next be scheduled.

The delivery of **signals** and the occurrence of **interprocess communication** events are mediated by the kernel, and can occur at any time for a process.

Many things happen transparently for a process. A process doesn't know **where it is located in RAM** or, whether a particular part of its memory space is currently resident in memory or held in the swap area.

Similarly, a process doesn't know where on the disk drive the files it accesses are being held; it simply refers to the files by name.

Process versus kernel views of the system

A process operates in isolation; it **can't directly communicate with another process**.

A process **can't itself create a new process** or even end its own existence.

A process can't communicate directly with the input and output devices attached to the computer.

The **kernel decides which process will next obtain access to the CPU**, when it will do so, and for how long.

Process versus kernel views of the system

The **kernel maintains data structures containing information about all running processes** and updates these structures as processes are created, change state, and terminated.

The **kernel maintains all of the low-level data structures** that enable the file names used by programs to be translated into physical locations on the disk.

The **kernel also maintains data structures that map the virtual memory of each process into the physical memory** of the computer and the swap area(s) on disk.

All communication between processes is done via mechanisms provided by the kernel.

The Shell

A shell is a special-purpose program designed to read commands typed by a user and execute appropriate programs in response to those commands.

Such a program is also known as a **command interpreter**

The term **login shell** is used to denote the process that is created to run a shell when the user first logs in.

On some operating systems the command interpreter is an integrated part of the kernel, **on UNIX systems, the shell is a user process**. Many different shells exist, and different users on the same computer can simultaneously use different shells.

Users and Groups

Each user on the system is uniquely identified, and users may belong to groups

Users

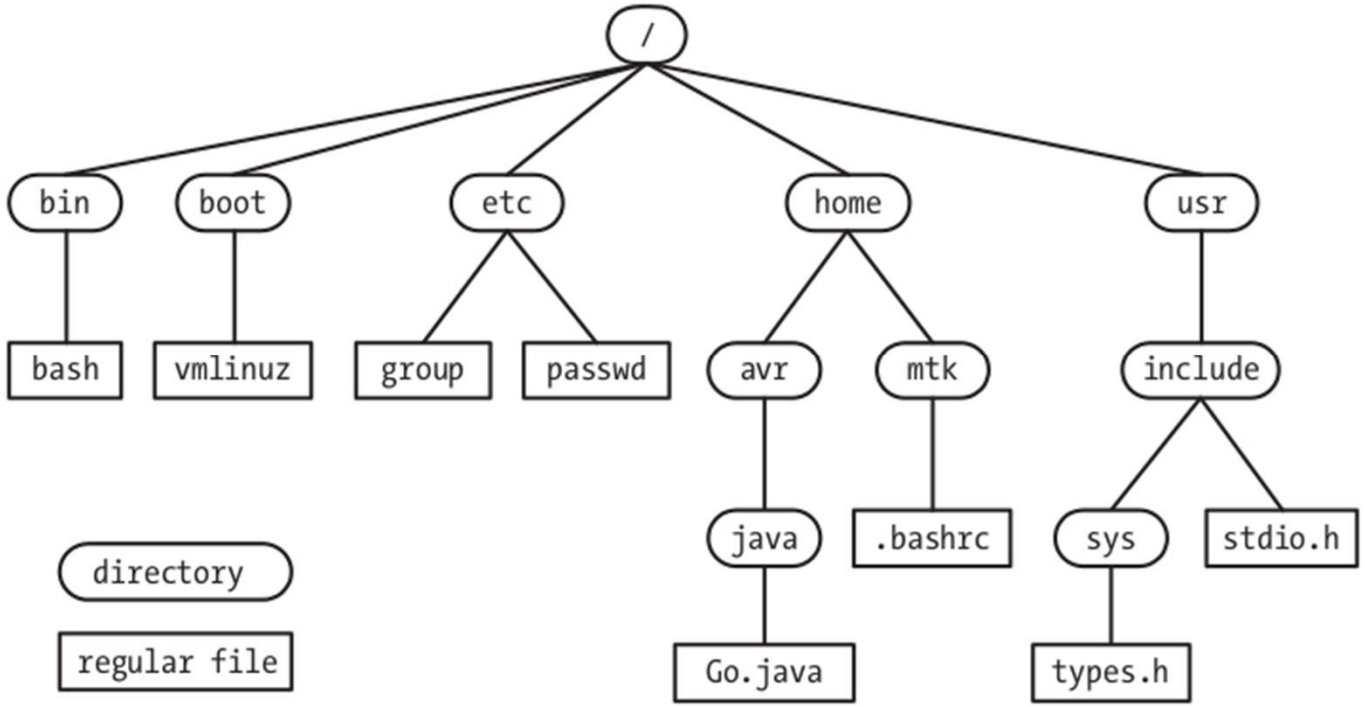
Every user of the system has a unique login name (*username*) and a corresponding numeric user ID (UID).

Groups

For administrative purposes (for controlling access to files and other system resources) it is useful to organize users into groups. Each group is identified by its *group name*, *Group ID* (GID) and *user list*

Single Directory Hierarchy, Directories, Links, and Files

The kernel maintains a hierarchical directory structure to organize all files in the system where each disk device has its own directory hierarchy.



Single Directory Hierarchy, Directories, Links, and Files

At the base of this hierarchy is the **root directory**, named / (slash).

All files and directories are children or further removed descendants of the root directory

Within the file system, **each file is marked with a *type***, indicating what kind of file it is.

File types include **devices, pipes, sockets, directories, and symbolic links**.

A *directory* is a special file whose contents take the form of a table of filenames coupled with references to the corresponding files.

Single Directory Hierarchy, Directories, Links, and Files

The filename-plus-reference association is called a *link*, and files may have multiple links, and thus multiple names, in the same or in different directories

Every directory contains at least two entries:
. (dot), which is a link to the directory itself, and
.. (dot-dot), which is a link to its parent directory, the directory above it in the hierarchy (except the root directory)

For the root directory, the dot-dot entry is a link to the root directory itself (thus, */..* equates to */*).

Directories, links and files

A *pathname* is a string consisting of an optional initial slash (/) followed by a series of filenames separated by slashes.

All but the last of these component filenames identifies a directory

A pathname is read from left to right; each filename resides in the directory specified by the preceding part of the pathname.

An *absolute pathname* begins with a slash (/) and specifies the location of a file with respect to the root directory.

A *relative pathname* specifies the location of a file relative to a process's *current working directory*

File I/O

One of the distinguishing features of the I/O model on UNIX systems is the concept of ***universality of I/O***.

The **same system calls** (*open()*, *read()*, *write()*, *close()*, ...) are used to perform I/O on all types of files, including devices.

Many applications and libraries interpret the *newline* character (ASCII code 10 decimal) as terminating one line of text and commencing another.

File descriptors

The I/O system calls refer to open files using a *file descriptor*, a non-negative integer. A file descriptor is typically obtained by a call to *open()*, which takes a pathname argument specifying a file upon which I/O is to be performed.

A process inherits three open file descriptors when it is started by the shell:

descriptor 0 is standard input, the file from which the process takes its input;

descriptor 1 is standard output, the file to which the process writes its output;

and **descriptor 2 is standard error**, the file to which the process writes error messages and notification of exceptional or abnormal conditions

Processes

When the process terminates, all such resources are released for reuse by other processes. Other resources, such as the CPU and network bandwidth, are renewable, but must be shared equitably among all processes.

A process is logically divided into the following parts, known as segments:

Text: the instructions of the program.

Data: the static variables used by the program.

Heap: an area from which programs can dynamically allocate extra memory.

Stack: a piece of memory that grows and shrinks as functions are called and return. Also used to allocate storage for local variables and function call linkage information.

Process creation and program execution

A process can create a new process using the ***fork()*** system call. The process that calls *fork()* is referred to as the parent process, and the new process is referred to as the child process. The kernel creates the child process by making a duplicate of the parent process.

The child inherits copies of the parent's data, stack, and heap segments, which it may then modify independently of the parent's copies

The child process goes on either to execute a different set of functions in the same code as the parent, or, frequently, to use the ***execve()*** system call to load and execute an entirely new program. An *execve()* call destroys the existing text, data, stack, and heap segments, replacing them with new segments based on the code of the new program

Process creation and program execution

Process ID and parent process ID

Each process has a unique integer process identifier (PID). Each process also has a parent process identifier (PPID) attribute, which identifies the process that requested the kernel to create this process. (PIDs wrap when maxed)

Process termination and termination status

A process can terminate in one of two ways: by requesting its own termination using the *_exit()* system call, or by being killed by the delivery of a signal.

The process yields a termination status, a small nonnegative integer value that is available for inspection by the parent process using the *wait()* system call.

Daemon processes

A **daemon** is a special-purpose process that is created and handled by the system in the same way as other processes, but is distinguished by the following characteristics:

It is long-lived. A daemon process is often started at system boot and remains in existence until the system is shut down

It runs in the background, and has no controlling terminal from which it can read input or to which it can write output.

Examples of daemon processes include *syslogd*, which records messages in the system log, and *httpd*, which serves web pages via the Hypertext Transfer Protocol

Environment List

Each process has an environment list, which is a set of environment variables that are maintained within the user-space memory of the process.

Each element of this list consists of a name and an associated value. When a new process is created via *fork()*, it inherits a copy of its parent's environment. Thus, the environment provides a mechanism for a parent process to communicate information to a child process.

When a process replaces the program that it is running using *exec()*, the new program either inherits the environment used by the old program or receives a new environment specified as part of the *exec()* call

Memory mapping

The **mmap() system call** creates a new *memory mapping* in the calling process's virtual address space.

Mappings fall into two categories:

A **file mapping** maps a region of a file into the calling process's virtual memory. Once mapped, the file's contents can be accessed by operations on the bytes in the corresponding memory region. The pages of the mapping are automatically loaded from the file as required.

By contrast, an *anonymous mapping* doesn't have a corresponding file. Instead, the pages of the mapping are initialized to 0

Memory mapping

The memory in one process's mapping may be shared with mappings in other processes.

This can occur either because:

- two processes map the same region of a file
- or because a child process created by *fork()* inherits a mapping from its parent.

Memory mappings **serve a variety of purposes**, including initialization of a process's text segment from the corresponding segment of an executable file, allocation of new (zero-filled) memory, file I/O (memory-mapped I/O), and inter process communication (via a shared mapping)

Static and shared libraries

- An *object library* is a file containing the compiled object code for a (usually logically related) set of functions that may be called from application programs

- Placing code for a set of functions in a single object library eases the tasks of program creation and maintenance

- Modern UNIX systems provide two types of object libraries *static libraries* and *shared libraries*.

Static libraries

- **Static libraries** were the only type of library on early UNIX systems. A static library is essentially **a structured bundle of compiled object modules**.

- To use functions from a static library, we specify that library in the link command used to build a program. After resolving the various function references from the main program to the modules in the static library, **the linker extracts copies of the required object modules from the library and copies these into the resulting executable file**.

Static libraries

- The fact that each statically linked program includes its **own copy** of the object modules required from the library creates a number of **disadvantages**.

- One is the **duplication** of object code in different executable files wastes disk space. A corresponding waste of memory occurs when statically linked programs using the same library function are executed at the same time; each program requires its own copy of the function to reside in memory.

- Additionally, if a library function requires **modification**, then, after recompiling that function and adding it to the static library, all applications that need to use the updated function must be **relinked** against the library.

Shared libraries

Shared libraries were designed to address the problems with static libraries.

- If a program is linked against a shared library, then, instead of copying object modules from the library into the executable, the linker just writes a record into the executable to indicate that at run time the executable needs to use that shared library.

- When the executable is loaded into memory at run time, a program called the **dynamic linker ensures** that the shared libraries required by the executable are **found** and **loaded** into memory, and performs **run-time linking** to resolve the function calls in the executable to the corresponding definitions in the shared libraries.

- They have commonly ".so" in their filename, the environment variable LD_LIBRARY_PATH contains the paths of shared libraries

IPC and synchronization

A running operating system consists of numerous processes, many of which operate independently of each other.

- Some processes, however, cooperate to achieve their intended purposes, and these processes **need methods of communicating** with one another and **synchronizing** their actions.

- One way for processes to communicate is by reading and **writing information in disk files**. However, for many applications, this is too slow and inflexible

Interprocess communication (IPC)

Modern UNIX implementations provides a rich set of mechanisms for interprocess communication, including the following:

- **signals** (to indicate that an error has occurred)
- **pipes** and FIFOS (to transfer data between processes)
- **sockets** (to transfer data between processes on the same or even on different computers)
- **file locking** (so a process can lock a region of the file and prevents others not to use it)
- **message queues** (exchange packets of data between processes)
- **semaphores** (to synchronize the actions of processes)
- **shared memory** (so that different proocesses can share the same memory page)

Signals

- Signals are often described as "**software interrupts**." The arrival of a signal informs a process that some event or exceptional condition has occurred.

- There are various types of signals, each of which identifies a different event or condition. Each signal type is identified by a different integer, defined with symbolic names of the form **SIGxxxx**

- Signals are sent to a process **by the kernel, by another process** (with suitable permissions), or **by the process itself**

- Within the shell, the **kill command** can be used to send a signal to a process.

- The **kill() system call** provides the same facility within programs

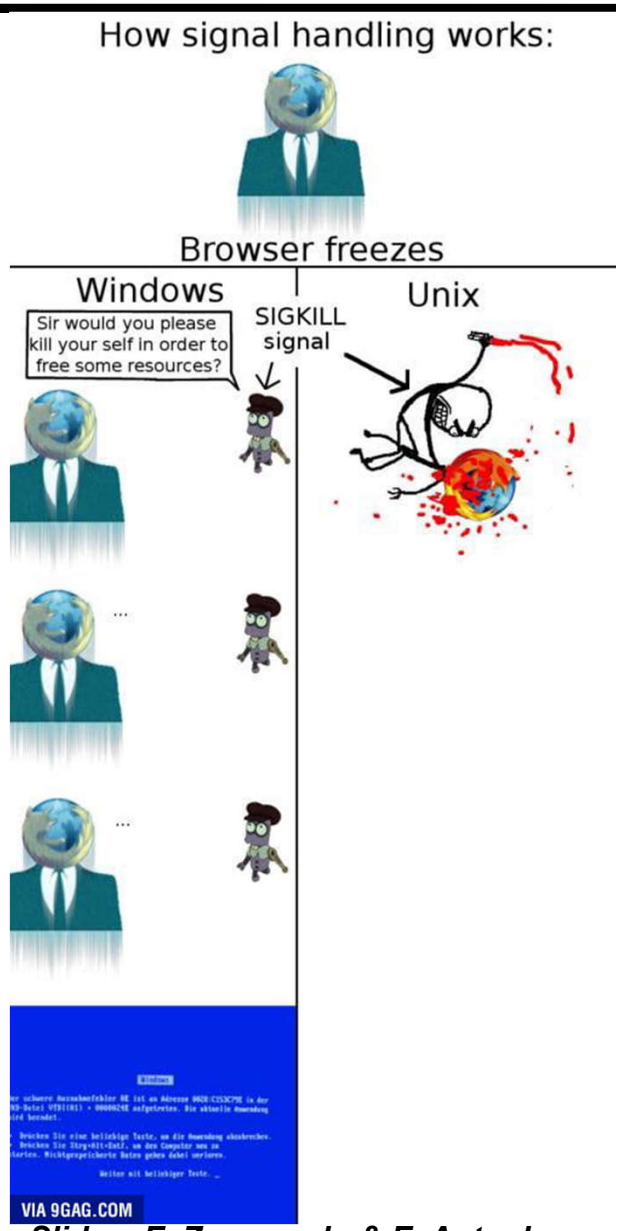
Signals

When a process receives a signal, it takes one of the following actions, depending on the signal:

- it **ignores** the signal;
- it is **killed** by the signal;
- it is **suspended** until later being resumed by receipt of a special-purpose signal

For most signal types, instead of accepting the default signal action, a program can choose to ignore the signal or to establish a **signal handler**.

A signal handler is a programmer-defined function that is automatically invoked when the signal is delivered to the process. This function performs some action appropriate to the condition that generated the signal



Threads

- In most modern operating systems each process can have **multiple threads of execution**.

- One way of envisaging threads is as a set of processes that share the same virtual memory.

- Each thread is executing the same program code and **shares the same data area and heap**. However, **each thread has it own stack** containing local variables and function call linkage information

- Threads can communicate with each other **via the global variables that they share**. The threading API provides **condition variables and mutexes**, which are primitives that enable the threads of a process to communicate and synchronize their actions, in particular, their use of shared variables

Threads

- The primary **advantages of using *threads*** are that they make it **easy to share data** (via global variables) between cooperating *threads* and that some algorithms transpose more naturally to a *multithreaded* implementation than to a *multiprocess* implementation.

- A *multithreaded* application can transparently take advantage of the possibilities for parallel processing on multiprocessor hardware.

Client-server architecture

A client-server application is one that is broken into two component processes:

- **a client**, which asks the server to carry out some service by sending it a request message
- **a server**, which examines the client's request, performs appropriate actions, and then sends a response message back to the client.

Typically, the client application interacts with a user, while the server application provides access to some shared resource. There are multiple instances of client processes communicating with a few (if not only one) instances of the server process

Real time

Realtime applications are those that need to respond in a timely fashion to input. Frequently, such input comes from an external sensor or a specialized input device, and output takes the form of controlling some external hardware.

The defining factor is that **the response is guaranteed to be delivered within a certain deadline** time after the triggering event.

The provision of realtime responsiveness, especially where short response times are demanded, requires support from the underlying operating system

Examples of applications with realtime response requirements include **automated assembly lines, bank ATMs, and aircraft navigation systems.**