Chapter 2: Operating System Structures

Andrés Ponce

September 30, 2020

> An *operating system* allows us to allocate resources to a machine. We can use either a graphical environment or all from the *command line*.

Operating System Services

The OS has some key services that it provides:

- **User Interface**: How does the user interact with the system? There have traditionally been two ways, **command-line interface**, where the user types the commands it wants the computer to execute. There is also the option for a **user interface**, where the user clicks icons and opens graphical programs to run commands and operate the computer.
- **Program Execution**: One of the main functions of an operating system is to load programs into memory and run those programs. One of the main abstractions that the OS provides is to load/execute programs.
- **I/O operations**: For safety reasons, the user seldom interacts directly with I/O devices, but the computer has to communicate with the outside. Writing to a network interface or talking with the filesystem maybe should not be left to the user.....
- **Communication**: Sometimes programs need to communicate with each other, maybe about error detection through sockets.
- **Error Detection**: When there is an error allocating the resources, memory or I/O error, the OS has to be there to detect and correct the error, or to halt the system operation.
- **Resource Allocation**: If there are multiple processes, the CPU has to manage the CPU scheduling routines for each process. There are some routines to manage the CPU schedule to manage multiple processes.
- **Logging**: If there is an error in your system, then the OS will write what happened to some files. Then we can know what is happening in each process.

The way we interact with the operating system is also different. In Linux, the main way to interact with the computer is through the **command line interface**, where the user types the commands the

computer is to execute. Other systems such as Windows and MacOs intend for the user to to use a graphical environment with icons and graphical folders.

System Calls

When we want our system to perform some action, we will usually specify the filename to run, and provide it with any arguments necessary. For example, if we were to type

```
cp foo.txt bar.txt
```

in our terminal, then our OS would know what commands we wanted to run and on which files to do it. In this example, the `cp` will *copy* a file called `foo.txt` and copy to another file on the same directory called `bar.txt`. Even in this simple command, there are multiple system calls going on, for example we have to open or create the files, then enter a loop which copies the lines, which requires even more system calls.

Usually the way that these calls are implemented is through an **Application Programming Interface**. The shell program might make a request to the API which then makes the system call. The reason for this is mostly to provide a standard format for systems using the same interface. For example, systems all using the POSIX standard can all expect similar functionality from its function calls.

When our API runs a command, how do we pass the information that the OS requires? There are two common approaches: **register method** and **bock method**. On Linux, if there are 5 or fewer parameters, we store the individual parameters in registers. For more arguments, we use the block method. In this method we store all the parameters in a block in memory and pass the address of the block. We can also use a stack to pass the arguments, since stacks don't care about the size or number of the arguments.

Types of System Calls

There are six major categories of system calls: **process control**, **file management**, **device management**,**information maintenance**, **communications** and **protection**.

1. **Process Control**: This are the calls responsible for running programs. If the program terminates normally or abnormally, we will generate an appropriate error message. The system might generate a memory dump and place the results in a file for the program to check.

In programs involving system calls, we have to call direct system calls from inside our program. For example, if we have a `printf()` instruction, we might have to call the equivalent `write()` system call. This type of processes could also apply when we want to **lock** a certian resource, or prevent its modification until a later point in time.

2. **File Management**: When we interact with files, we might also need to create, close, modify, move, etc... files around the directory if we have one.

3. **Device Management**: When we have to interact with another device such as disk, we have to ask for control of the device first, then we can read the data, and finally close the connection. Since these functions are similar to the ones used for files, some OSes (Linux) combine the two into one. This means that devices are treated as files. The data from one device might be available directly somewhere on the file system!

4. **Information Maintenance**: There are also system calls for getting information about the system. For example, we can get the `date()` or `time()`, how much free space there is on the system, etc... These sytem calls are useful for debugging and knowing what order of system calls are being executed. We can use a CPU's **single step** mode to find the order of instructions being executed.

5.

6.