CMPSCI 577 Operating Systems Design and Implementation **Spring 2020**

Lecture 6: Feb 6th, 2020

Lecturer: ***Prashant Shenoy*** *Scribe:* ***Serena Chan***

6.1 Introduction to System Calls in Linux & Minix

A system call is essentially an interface that the OS exposes for the user. However, this often is not invoked directly by the programmer; instead, an Application Program Interface (API) is used in place in order to provide an even higher level of abstraction. The API invokes the system call in its implementation in order to enable the programmer to indirectly invoke the system call by calling the functions provided by libraries. There are a few common well-defined APIs for system calls. UNIX systems (as well as Linux and OS X) use the POSIX API so they will be the main focus. However, there are other examples - for instance, Windows uses the Win32 API, and the JVM runs on the Java API.

As an example, we can look at the `printf()` function, which is part of the standard C I/O library. The `printf` function abstracts away the system call `write()` that the library implementation invokes. The Win32 equivalent behaves similarly.

6.2 System Calls Implementation

6.2.1 General Implementation

When a system call happens, the TRAP instruction is invoked, and the process is now running in kernel mode (assuming a monolithic kernel). The unique integer code of the system call is used to index into the trap vector, or a table of function addresses, which correspond to the memory address of the relevant system call. The kernel then jumps to the memory location, and starts executing. However, consistent

In order to pass arguments to a system call, the stack-based approach becomes more complicated, since the user process has a different address space from the kernel. An obvious choice would be to use registers - this suffices for system calls with a smaller number of arguments, but there could be situations where the number of arguments exceeds the number of registers. Instead, operating systems like Linux and Solaris will allocate a block in memory that stores the relevant parameters and write the address of the block to the register. Another option is to continue using the stack, and have the operating system keep track of extra information indicating which user process's stack it should obtain parameters from.

6.2.2 POSIX API

The POSIX API is an IEEE-defined standard, and any operating system that implements the POSIX API is said to be **POSIX-compliant**. The significance of POSIX-compliance (or API compliance in general) is portability - an application that performs POSIX API calls can be compiled and run across on any POSIX-compliant OS. Without a well-defined API, code that runs on a machine would need to be modified in order to run on other systems, due to different system calls or libraries present on the OS. The POSIX API does not only include system calls, but a wide variety of essentials like networking (socket communication) and POSIX threads (commonly known as `pthread`).

6.3 Minix System/Kernel Calls

Note: MINIX is POSIX-compliant.

While MINIX loosely follows the general implementation, an important distinction to make is that MINIX differentiates between system and kernel calls. Therefore, when a system call is made, it is first handled by a system-level process (e.g. the reincarnation server or the file system), unlike in other operating systems, where system calls go directly to the kernel. If the system call can be handled directly by one of the processes, it is handled and the result is returned. If the system call requires the kernel (e.g. `fork`), it is handled with the typical system call routine. For some particular functions, like `fork`, there are even more additional layers of complexity that comes with the need to manipulate more parts of the operating system (e.g. memory).

In MIINIX, the list of system calls can be found in `/minix/include/minix/callnr.h` - these are the functions that can be directly invoked by user processes. It is worth noting that, in addition to kernel calls, IPC is also restricted to system processes. In order to limit the privileges a server or function has, the `/usr/system.conf` file is used to explicitly state permissions.

It may seem, upon first glance, that every kernel call must have a corresponding system call, since the path for users to invoke a kernel call must necessarily pass through a system call. However, this is not true - for instance, the `sys_devio()` call does not have a system call counterpart.