Operating Systems 2

System Calls

Arkadiusz Chrobot

Department of Information Systems

April 8, 2025

Outline

System Calls Background

System Calls are kernel functions that can be called by user-space processes. They create an interface between user-space software and the operating system. For safety reasons modern kernels do not allow user processes to directly interact with resources, such as hardware, or even with themselves (the shared memory is an exception). However, to complete their tasks the processes need these resources and they can access them indirectly, using services provided by the operating system with the help of system calls. That has several advantages. The system calls simplify usage of resources, especially hardware resources, for the user-space software. Programmers who create such programs don't have to know or worry about details of using hardware. They just delegate the work to the operating system. Moreover, system calls are crucial for implementing *virtualization* of resources, like a CPU or a memory.

Application Programming Interface

User processes usually do not directly invoke system calls. Instead they use something which is called an *Application Programming Interface* (an API, for short). It is a set of functions, variables and other software constructs that allow the processes to perform the most needed operations and also to cooperate with the operating system. Linux has an API defined by the POSIX and SUS standards, which is inherited from the Unix system. It is implemented (mostly) in a form of the *Standard C Library* called *glibc* (*libc* in Unix). Some of the functions from this library do not use any system calls (like the `strcpy()` function), some of them are simple wrappings for a system call (like the `write()` function) and some of them perform other operations besides invoking a system call (like the `printf()` function).

Invoking System Calls

A user process cannot invoke system calls like regular functions because they are part of the kernel-space. Modern CPUs provide a special instruction generally called a *software interrupt*[1], which can be performed by a user-space software and which causes the CPU to switch to the system mode and give the control to the kernel, just like when serving a hardware interrupt or an exception. The instruction has many names, depending on the CPU. In case of the PowerPC processors it is called `sc`. In case of 32-bit Intel and AMD processors it is called `int` and is followed by an operand (an argument) which is the number of the interrupt (`0x80` — 128 in decimal, in case of Linux). Newer 32-bit Intel processors, starting from Pentium II provide the `sysenter` instruction. The 64-bit AMD and Intel processors have the `syscall` instruction.

[1] This name is ambiguous in Linux terminology, so be careful when using it.

Invoking System Calls

Intel And AMD 32-Bit CPUs

The software interrupt is handled by a special kernel function named `system_call()`. Its implementation and behaviour depends on the CPU on which the kernel is running. For the 32-bit Intel and AMD CPUs it is defined in a file called `entry_32.S` file (the `.S` extension indicates that this file contains assembly code). The `system_call()` function is also a *call gate* for all of the system calls. After being invoked it first checks the content of the `eax` register. It should contain the *system call number*. Each system call has a unique number which is also an index in the `sys_call_table` array. This array contains addresses of all system calls and it is defined in the `syscall_table_32.S` file which is included in the `entry_32.S` file. This means that the array is also implemented in assembly code. If the number in `eax` register is incorrect — bigger than expected — the `system_call()` function returns the `-ENOSYS` value, signaling that the user process tried to call a non-existing system call and exits.

Invoking System Calls

Intel And AMD 32-Bit CPUs

If however the system call number in the `eax` register is correct then the `system_call()` function pushes onto the process kernel stack the content of the `ebx`, `ecx`, `edx`, `edi`, `esi` registers. These registers store arguments for the system calls. Next, it multiplies the system call number by 4 (the size of the 32-bit address expressed in bytes) and uses the result as an offset in the `sys_call_table` to locate the address of the system call. Finally, the `system_call()` function uses the address to invoke the system call using regular `call` instruction. On exit the system call returns a result by storing it in the `eax` register and passes the control back to the `system_call()` function.

Invoking System Calls

Intel And AMD 64-Bit CPUs

In case of Intel and AMD 64-bit CPUs the `system_call()` function is defined in the `entry_64.S` file and the `sys_call_table` array in the `unistd_64.h` file which is included in the `syscall_64.c` file, which in turn is included in the `entry_64.S` file. The system call number is stored in the `rax` register, where also the system call returns its result. The system call number is multiplied by 8 (64-bit address size expressed in bytes) instead of 4. The arguments for the system call are stored in the `rdi`, `rsi`, `rdx`, `r10`, `r9` and `r8` registers, however they **are not pushed** onto the process kernel stack.

Defining of System Calls

System Call Header

System calls are regular kernel functions running in the process context, but they are defined differently. Names of these functions start with the `sys_` prefix, so for example, the name of the function that implements the `write()` system call is actually `sys_write()`. Every system call returns a `long` type number that usually, but not always, indicates a success or a failure. In that case zero usually means that the system call completed its task successfully and a negative number describes the reason of the failure. However, for some system calls the meaning of this number can be different. For 32-bit x86 CPUs the system call can have up to 5 parameters and for 64-bit x86 CPUs, up to 6. These parameters are used for passing the arguments originally stored in registers. The system call header contains the `asmlinkage` macro which informs the compiler that arguments for the function are passed using the process kernel stack. If the CPU (like the 64-bit x86 CPUs) does not require this, then the `asmlinkage` macro is empty.

Defining System Calls

System Call Header

To simplify creating headers of system calls, Linux kernel programmers added macros called `SYSCALL_DEFINE`n, where n is the number of parameters the system call requires, and ranges from 0 to 5 or 6, for Intel and compatible CPUs. If the system call does not need any parameters the `SYSCALL_DEFINE0` macro can be applied for building its header. If the system call requires 2 parameters then the `SYSCALL_DEFINE2` macro can be used for creating its header. However, each of the macros requires $2 \cdot n + 1$ arguments, where the n is the number of system call parameters. The first argument is the name of the system call (without the `sys_` prefix) followed (if needed) by a pair or pairs consisting of the parameter type and name.

Defining System Calls

System Call Body — General Remarks

Almost every system call has a side effects, which means that it changes the state of the kernel. Since they run in the process context they get access to the descriptor of the process that invoked them using the `current` macro and can set that process in one of the waiting states. In Linux terminology it is called "putting a process to sleep". If the system call requires more than 5 or 6 arguments, or the arguments do not fit in registers, then they values can be stored in some memory area and the starting address of this area can be passed by one of the registers. Each system call that gets arguments has to verify them before using. Information that comes from the user-space cannot be trusted. Arguments that are pointers have to be especially carefully checked.

Defining System Calls

System Call Body — General Remarks

The verification of a pointer is performed in three steps:

1. Does the pointer point to a memory area in the user-space?
2. Does the pointer point to a memory area that belongs to the process which invoked the system call?
3. Depending on the operation that is to be performed the system call checks if the invoking process has permissions to read, write or execute for the memory area.

The system call may also have to check the permissions of the invoking process for other resources. Since the 2.6 version of the kernel it can use the `capable()` function to this end. The function is defined in the `linux/capability.h` header file together with constants that can be applied as its arguments and which describe different permissions. If the process has a given permission then the function returns a non-zero value, otherwise it returns zero. Before the function was introduced, system calls only checked if the invoking process belonged to a privileged user by calling the `suser()` function.

Defining System Calls

System Call Body — General Remarks

If data have to be copied from a user-space memory area to a kernel-space memory area then the `copy_from_user()` function can be used. If the data have to be copied in the opposite direction then the `copy_to_user()` function can be applied. Both functions take three arguments: the starting address of the destination memory area, the starting address of the source memory area and the number of bytes that have to be copied. Also the meaning of the return value is the same for both of them. On success they return zero, on failure the return the number of bytes left to copy.

There is a special system call, named `sys_ni_call()`. It only returns the `-ENOSYS` value. This function is used for handling system calls that were not implemented for a given hardware (for example they are available for x86 processors, but not for PowerPC CPUs), or that were for some reason removed. Fortunately, the latter happens not very often in the Linux kernel source code.

Defining System Calls

The Return Value

When the system call exits, its result is stored in the `eax` or `rax` register. From there it eventually gets to the `errno` variable in the user-space. If the system call fails to complete its task then the number can be used by such a function like `perror()` to print on the screen a human-readable description of the cause of the failure.

Adding New System Call

Adding a new system call is relatively easy. For example for the 32-bit x86 CPUs it requires defining the system call in one of the kernel source code files associated with the part of the kernel related to the system call, adding a new entry in the array of system calls located in the `syscall_table_32.S` file, specifying the number of the system call in the `include/asm/unistd_32.h` header file and finally compiling and installing the kernel. The steps are similar for the 64-bit x86 CPUs, but adding the new entry to the system call array and specifying the system call number requires only a modification of the `include/asm/unistd_64.h` header file.
In case of more recent kernel versions, adding a new system call is even easier. It only requires adding a new entry in the `syscall_64.tbl` or `syscall_32.tbl` file, and a system call prototype to the `syscalls.h` header file.

Adding New System Call

Invoking New System Call

Since no API function knows the new system call, no one will call it. There is however a way of invoking such a system call from the user-space. Before the 2.6.18 version of the kernel, there were available macros called `_syscall`n, where n specifies the number of arguments taken by the system call. Each of the macros required $2 \cdot n + 2$ of its own arguments. The arguments where as follows: type of the system call return value, name of the system call and, optionally, a pair or pairs consisting of argument type and value. However, those macros were not available for all hardware supported by Linux, and since 2.6.18 version of the kernel they have been replaced by the `syscall()` function, which takes one mandatory argument — the system call number and any number of arguments required by the system call. The function returns the same value as the invoked system call.

Adding New System Call

Pros And Cons of Adding New System Call

Advantages:

1. System calls can be relatively easy implemented and used.
2. System calls in Linux are very efficient.

Disadvantages:

1. A system call number has to be assigned to the new system call which must be accepted by all Linux kernel programmers.
2. The interface of the system call (the number and order of its arguments) cannot be changed in the future.
3. A system call may have to be defined for all hardware supported by the Linux kernel.
4. A system call should not be used as a communication device for user-space processes.
5. A system call cannot be defined as a part of a kernel module.

Summary

The original Unix operating system had about 100 system calls. Linux has about 400 of them. The number differs for different hardware supported by the kernel, but generally it does not change very often. Most of system calls follow the philosophy of "doing one thing, but doing it well". As a counterexample the `ioctl()` system call can be given. Adding new system calls should be avoided. Some of the issues that initially are thought to be only tackled by introducing a new system call to the kernel, can be solved with the help of the properties of the device handling subsystem or filesystems.

Questions

?

The End

Thank You for Your attention!