System Calls

Hello, and welcome to this brief exploration of system calls.

We already know that the system calls are the API for the user to access operating system functions. We will briefly discuss what types of system calls there are. We will see that today's operating systems have quite a few system calls!

We will then spend most of the rest of this lesson investigating the implementation of system calls. They do look like function calls, but we will see that actually they are not.

If they are not function calls, what are they then? The difference is in how they are called. In fact, we will see how system calls are invoked through software interrupts rather than the jump-to-subroutine mechanism that we would expect.

In addition to being somewhat complicated and quite counter-intuitive, this way of calling operating system functions is awfully expensive. We will learn why this is the case and why system calls are implemented the way they are.

Operating System Interfaces: System Calls

Let's briefly recap. What are system calls?

When a user application wants to access a function provided by the operating system, for example to read from a device, the application has to do this through a well-defined API: the system call interface of the operating system.

The application invokes a system call.

Types of System Calls

Today's operating systems support hundreds of system calls, which may be grouped as follows:

- Process Control: System calls related to process control deal with loading and execution of programs and creation and termination of processes. They also deal with memory allocation and synchronization.
- File management system calls handle the creation, access, and deletion of files and of file systems.
- Device Management: System calls that deal with device management control the access to and configuration of devices.
- Information Maintenance: There is a number of system calls that provide system information to the caller.
- Communication: And finally, there are quite a number of system calls that deal with communication. They manage connections, send and receive messages, and manage communication devices.

This all sums up to quite a number of system calls. Linux for example has currently about 350 different calls, and the number is growing.

System Calls - They *look like* Functions . . .

The normal user is often not aware when they invoke a system call. For example, the following program writes the string "Hello world" to file descriptor 1, which happens to be standard out.

The operation "write" is a function call, which takes three arguments, the number of the file descriptor, the character array representing the string, and the length of the string. So, the write operation looks like a function call.

. . . but they *are not*!

But appearances can be misleading.

A function call is typically implemented by pushing the return address and arguments onto the stack, and then calling some form of jump-to-subroutine instruction.

This is not the case for system calls, as the following example of the implementation of system calls for 32-bit Linux illustrates. In this case, a system call is issued in three steps

1. First, the number of the system call is stored in register A. Each system call has a specific number, which ranges from zero to about the total number of supported system calls.
2. The arguments are stored in the following registers, registers B, C, and D, and some additional registers if needed.
3. Now we invoke the operating systems system call handler by triggering a Hex 80 software interrupt. The interrupt service routine for interrupt Hex 80 looks at the value of register A and calls the intended function in the kernel.

Once the function returns, the interrupt service routine loads the return value into register A and returns with a return-from-interrupt instruction.

For 64-bit Linux the approach is similar, with three main differences:

1. First, the names of the registers are different, reflecting the register set of 64-bit architectures.
2. Second, the numbering of the system calls is different as well, for some reason.
3. And third, the invocation using software interrupt Hex 80 is replaced by a so-called syscall fast-system-call instruction, which is a more efficient implementation of the previous interrupt mechanism. We will look at a detailed example in a minute.

System Call Numbers

Here we see a list of a few system calls for 32-bit Linux, including their number, and the arguments. This list is just the beginning of over 300 system calls.

In a minute we will be looking at an example that uses the write system call, which in 32-bit Linux is identified as system call number 4. It takes three arguments, namely the file descriptor, a pointer to the beginning of the character array to be written, and a length of the array.

System Calls - Example

Let's look at an example, where we implement the Hello World program in three ways:

- First we implement Hello World in C, using the UNIX standard library.
- Then we implement it using the 32-bit Linux system calls directly, in assembler.
- Finally, we do the same, but using 64-bit Linux system calls.

Let's start.

In this example we write the string "Hello World" to standard out using a C program. First we include UNIX standard which gives us access to the UNIX system calls. Here is the main function. Then we use the function write to write to standard out the string "Hello world". The string is 13 characters long. The return is zero. And . . . we exit the editor, compile, and run the resulting a.out. And we get the expected output.

Now let's do the same using Linux 32-bit system calls. In this example we write the string "Hello world" to standard out using 32-bit style Linux system calls to write the code in Assembler. The exported entry point is called start, and here comes the executable. Here is the start of our program, and here we define a string variable message that contains the string "Hello world".

Now we set up the registers for the invocation of the write system call. The write system call is number 4, so we load this into register A. We want to write to file descriptor number 1, so we load this into register B, we load the pointer to the message into register C, and finally we load the length of the message we want to write, so 13 in this case, into register D. We invoke the system call by triggering interrupt number 80, which is the system call interrupt. After returning from the write system call, we need to call D. Exit the system call to return from the program. The exit system call is number 1. We load this into register A. We want to return the value zero, so we load that into register B, and we trigger interrupt number 80.

We leave the editor, we compile the program, and we tell the compiler that we don't want standard libraries, and we run the program. And here is the "Hello world".

As a side note, if you look at the size of the executable, you will notice that it is a tiny 928 byte. In fact this compares well to the more than the 8 kilobyte of the C implementation. This is because the Assembler implementation does not need to link in libraries.

Now let's do the same using Linux 64-bit calls. For completeness sake, let's repeat this for 64-bit Linux. There are 2 differences between 32-bit Linux system calls and 64-bit Linux system calls. First, because 64-bit architectures have a different register set, we will be using different registers to load the parameters for the system call. Second, 64-bit architectures have a special system call instruction, which replaces interrupt Hex 80. Also, the numbers assigned to the system calls have changed. In our case the number of the write system call is changed from 4 to 1, so we load 1 into register A, we load the file descriptor number 1 in what used to be register B, now it is rdi register, we move the message pointer into what used to be register C, now rsi register, and we move the length of the message string, 13, into the D register. Now instead of invoking software interrupt Hex 80 to make the system call, we use the instruction syscall. We can think of this instruction as being a fast implementation of the Hex 80 interrupt.

Similarly to before, we use the exit system call to terminate the program in an orderly fashion, exit system call is number 60. We load the return value zero into the rdi register and we call the system call instruction. We compile the program similarly to the 32-bit version. We execute the program and here is the result.

System Calls are Expensive!

System calls using software interrupts are not only somewhat complicated and rather counter-intuitive. They are also very expensive.

This is because software interrupts, much like all other interrupts, are themselves expensive. Why is this?

Each interrupt requires two context switches, one when the interrupt occurs and we have to save all the registers, and the other when we return from the interrupt and we have to restore all the registers. This saving/storing business consumes quite a lot of cycles.

In addition, after each context switch the cache is stale.

The same holds for the TLB entries, and any pre-fetching in the CPU becomes obsolete as well.

To make matters worse, it is very difficult for the compiler to optimize away this overhead. The same would be rather trivial if the system calls were actual function calls.

Why Interrupts or syscall?

So, why do we bother with this convoluted and expensive way to implement system calls? There are four main reasons for this.

1. First, the user program does not know where a particular system function is. If the operating system were a library, then we would solve this problem by linking the OS to the user program. This of course

cannot be done in practice. The software interrupt approach solves this problem because the user program specifies the system call number as one of the arguments and invokes the software interrupt, which causes the system to call the system-call interrupt service routine. This routine reads the system call number in the argument and calls the intended function. As a result, the user calls the intended function without knowing where it it is.

2. Second, CPUs allow to switch the stack as part of a context switch. In this way we can have a user stack and a separate kernel stack. The stack of the user program does not get co-mingled with the stack of the operating system kernel, which helps protect the kernel.
3. Third, most CPUs switch from user mode to supervisor mode whenever an interrupt occurs. During a system call the CPU is therefore in supervisor mode. It returns back to user mode when the system call returns.

If interrupts are the only way to change to supervisor mode, then the only way for a user to issue privileged instructions is to make a system call and have the handler for the system call issue the instructions on the user's behalf.

Note that system security is naturally enforced, as the handler for the system call will check whether the calling user program is authorized to make the requested call. For example, a user wanting to call the HALT instruction to halt the system would have to call some sort of a shutdown system call, which would switch the system to supervisor mode, but would also check if the owner the user program is authorized to shut down the system. If not, the system call would simply return with an error.

4. Finally, most CPUs automatically mask interrupts whenever an interrupt occurs. Interrupts can be unmasked inside the interrupt service routine whenever appropriate. This generally allows for a flexible way of dealing with nested interrupts. Operating system designers use this to enforce mutual exclusion in the kernel.

Reason 4: Mutual Exclusion in Kernel

Let's see how this works.

Here we have a closer look at the boundary between user and kernel space.

At some point a thread makes a system call.

The system call is invoked through a software interrupt, which in turn masks all other interrupts. Now, the thread is executing in the kernel.

If at some later point in time another thread wants to make a system call and enter the kernel, it will not be able to do so because interrupts are masked, and the software interrupt for the new thread's system call either gets queued or dropped.

When the first thread returns from the system call, the system call interrupt service routine returns with a return-from-interrupt instruction. The caller continues executing where it left off. In addition, the return-from-interrupt instruction typically unmasks any mask interrupts, including the system calling interrupt.

Now other threads can again make system calls.

In practice, things are a bit more complicated, primarily for two reasons.

- First, blocking of interrupts to enforce mutual exclusion works great on a single processor, but does not work in a multicore system.
- Second, in some cases it is better to not strictly enforce mutual exclusion in the kernel, for example when a system call causes a thread to block. In such cases it could be beneficial or even necessary to allow other threads into the kernel

Summary: System Calls

Let's summarize what we learned in this lesson about system calls.

1. First, system calls are the user's Application Programming Interface (API) to the operating system. The only other way to invoke operating system functions is for devices through device drivers.
2. We spent quite some time investigating how the system call invocation mechanism is implemented. System calls look like normal functions, but they are really not. They are just normally hidden behind system call libraries to make them simpler to use.
3. We learned how systems calls are implemented using software interrupts. On newer architectures they are also implemented using specialized fast-system-call instructions.
4. We also concluded that software interrupts are very expensive, which begged the question why an operating system designer would use them to implement something that is used as often as system calls. We learned four common reasons for why operating designers insist on using this expensive mechanism.

We sincerely hope that you enjoyed this presentation on the nature and implementation of system calls.

Thank you for watching!