System Calls

System Calls are used to call a kernel service from user land. The goal is to be able to switch from user mode to kernel mode, with the associated privileges. Provided system calls depend on the nature of your kernel.

Contents

Possible methods to make a System Call
- Interrupts
- Sysenter/Sysexit (Intel)
- Trap
- Call Gates (Intel)

Passing Arguments
- Registers
- Stack
- Memory

Security/safety implications

On the user land side

Strategies Conclusion

See Also
- Threads
- External Links

Possible methods to make a System Call

Interrupts

The most common way to implement system calls is using a software interrupt. It is probably the most portable way to implement system calls. Linux traditionally uses interrupt 0x80 for this purpose on x86. Other systems may have a fixed system call vector (e.g. PowerPC or Microblaze).

To do this, you will have to create your interrupt handler in Assembly. This is because your system call ABI will likely not correspond to the normal ABI the compiler supports. It is therefore necessary to translate from one to the other.

For example, on i386, the Linux kernel gets its arguments in `eax, ebx, ecx, edx, esi, edi, and ebp` in that order. The ABI however places all arguments in reverse order on the stack. Linux proceeds to construct a `pt_regs` structure on the stack and passes a pointer to it to a C function to handle the call itself. This can be simplified into something like this:

```
Int128Handler:
    ; already on stack: ss, sp, flags, cs, ip.
    ; need to push ax, gs, fs, es, ds, -ENOSYS, bp, di, si, dx, cx, and bx
    push eax
    push dword gs
    push dword fs
    push dword es
    push dword ds
    push dword -ENOSYS
    push ebp
    push edi
    push esi
    push edx
    push ecx
    push ebx
    push esp
    call do_syscall_in_C
    add esp, 4
    pop ebx
    pop ecx
    [...]
    pop es
    pop fs
    pop gs
    add esp, 4
    iretd
```

Many protected mode OSes use EAX to hold the function code. DOS uses the AX register to store the function code — AH for the service and AL for functions of the service, or AH for the functions if there are no services. For example, let's say you have read() and write(). The codes are 1 for read() and 2 for write() from the interrupt 0A9h (an arbitrary choice, possibly wrong). You can write

```
 IntA9Handler:
     CMP AH, 1
     JNE .write
     CALL _read
     JMP .done
 .write:
     CMP AH, 2
     JNE .badcode
     CALL _write
     JMP .done
 .badcode:
     MOV EAX, 0FFFFFFFFh
 .done:
     IRETD
```

However, if all function codes are small contiguous numbers, a better option might be a function table, such as:

```
dispatch_syscall:
    cmp eax, NR_syscalls
    ja .badcode
    jmp [syscall_table+4*eax]
.badcode:
    mov eax, -ENOSYS
    ret
```

Note that this assumes the syscall table to be NULL free. If there is a hole in the table, fill it with a pointer to a function returning an error code!

Sysenter/Sysexit (Intel)

Main article: Sysenter

On Intel CPU, starting from the Pentium II, a new instruction pair sysenter/sysexit has appeared. It allows a faster switch from user mode to kernel mode, by limiting the overhead of changing mode. The sysenter entry point will have the kernel stack set already. However, sysenter does absolutely no state saving, so the user stack pointer and return address both have to either be well-known values, or have to be saved by the user-space code leading up to the sysenter. Also, besides unconditionally clearing the interrupt and VM flags, sysenter will not modify any flags.

A similar instruction pair has been created by AMD: Syscall/Sysret. However the behaviour of these instructions are different from Intel's. The syscall entry point will still have the user space stack loaded, and will have to save it and load the kernel stack. The only reasonable way to do this is by way of a CPU-local variable: By way of the `swapgs` instruction, a CPU-local pointer can be loaded, behind which the user stack pointer can be saved, before overwriting the stack pointer with the kernel value (which also can be saved among the CPU-local variables). In 32-bit mode you get a bit of a chicken-and-egg scenario: You can't save any register to stack, since the stack in question is user stack and thus not to be trusted or modified, and in an SMP system, you can't use any global state, either. And you need to save pretty much all registers, so you can't modify them. So, possibly avoid syscall in 32-bit mode.

In 64-bit mode, the flags register can be modified by way of the SFMASK MSR. The original RFLAGS value will be saved in r11.

Note that, although these instructions did appear in pairs, there is no actual need to keep these instructions paired. With a properly constructed stack-frame, a system call that was started with `syscall` can be ended with `iret`.

The kernel can specify which registers are preserved and which registers are lost on SYSENTER or SYSCALL (with the exception of r11 in 64-bit mode, which is always lost) as part of its syscall ABI. It then does not need to save all registers but only those specified as being preserved. Most commonly the C calling conventions in use are followed. By using a tiny assembler stub that calls SYSENTER or SYSCALL the C compiler will safeguard caller saved registers. The kernel entry point for SYSENTER or SYSCALL can then be another small assembler stub that avoids changing any callee saved register before calling a C function for the syscall. That way only the user space stack pointer (and r11 in 64-bit mode) need to be saved as everything else is either preserved by the C compiler or allowed to be destroyed.

Note that for security reasons the kernel should zero all the registers that are not preserved across SYSENTER or SYSCALL so no information is accidentally leaked from kernel to userspace.

Trap

Some OSes implement system calls by triggering a CPU Trap in a determined fashion such that they can recognize it as a system call. This solution is adopted on some hardware by Solaris, by L4, and probably others.

For example, L4 use a "LOCK NOP" instruction on x86. Since it is not permitted to perform a lock on the "NOP" instruction a trap is triggered. The problem with this approach is that there is no guarantee the "LOCK NOP" will have the same behavior on future x86 CPU. They should probably have used the "UD2" instruction, since it is defined for this purpose.

Call Gates (Intel)

The 80386 family of processors offer various call gates as part of the GDT. The call gate is a far pointer that can be called similar to calling a normal function. Very few operating systems use call gates.

To use a call gate in 32-bit mode, an entry has to be added to the GDT. Assuming the standard two-tier architecture (kernel is ring 0, user is ring 3, and rings 1 and 2 are unused), the DPL of that segment needs to be 3, the first two bytes and last two bytes of the descriptor (i.e. the limit, flags, and high base fields) need to be set to a pointer to the handler function, the second two bytes (the low base field) needs to be set to the kernel code segment selector, the mid-base field can contain a parameter count (up to 31 DWORDs can be copied from user to kernel stack), and the access byte has to be set up like this: Pr must be 1, Privl must be 3, the bit the GDT page claims must be 1 (which is called S in the AMD documentation) must actually be 0, and below that a value of 1100 causes the segment to be seen as a 32-bit call-gate.

To use the gate, user-space code must use a far-call instruction. The offset will be ignored. Assuming the gate is the first entry in the GDT, segment 0x0b will have to be requested (offset 8 and RPL 3):

```
call far 0x0b:0
```

In 64-bit mode, the descriptor size is doubled, with the high half of the handler address directly after the rest of the descriptor described above. Also, the argument count has to be zero, and the second DWORD of the second descriptor has to be all zeros. Otherwise, no changes.

Passing Arguments

Registers

The easiest way to pass arguments to a System Call handler are the registers. The BIOS takes arguments this way.

Pros:

- very fast

Cons:

- limited to the number of available registers
- caller has to save/restore the used registers if it needs their old values after the System Call
- insecure (if the caller passes more/less arguments than the callee assumes to get)

Stack

It is also possible to pass arguments through the stack.

Pros:

- nested System Calls are possible
- it is easy to implement a System Call handler in C because C uses the stack to pass arguments to functions, too
- not limited

Cons:

- insecure (if the caller passes more/less arguments than the callee assumes to get)

Memory

The last common way to pass arguments is to store them in memory. Before making the System Call the caller must store a pointer to the argument's location in a register for the System Call handler. (assuming this location is not fixed)

Pros:

- not limited
- secure

Cons:

- one register is still needed
- nested System Calls are not possible without copying arguments
- insecure (if the caller passes more/less arguments than the callee assumes to get)

Security/safety implications

Since the kernel is running at higher privilege than the user mode code calling it, it is imperative to check everything. This is not merely paranoia for fear of malicious programs, but also to protect your kernel from broken applications. It is therefore necessary to check all arguments for being in range, and all pointers for being actual user land pointers. The kernel can write anywhere, but you would not want a specially crafted `read()` system call to overwrite the credentials of some process with zeroes (thus giving it root access).

As for making sure that pointers are in range, checking if they point to user or kernel memory can be difficult to do efficiently unless you are writing a Higher Half Kernel. For checking all user space accesses for being valid, you can either check with your Virtual Memory Manager to see if the requested bytes are mapped, or else you can just access them and handle the resulting page faults. Linux switched to doing the latter from version 2.6 onwards.

On the user land side

While the developer can trigger a system call manually, it is probably a good idea to provide a library to encapsulate such call. Therefore you will be able to switch the system call technique without impacting user applications.

Another way is to have a stub somewhere in memory that the kernel places there, then once your registers are set up, call that stub to do the actual system call for you. Then you can swap methods at load time rather than compile time.

Note that whatever library you provide, you cannot assume the user to call the system with that stub. They can, and will, call the system directly if given half the chance.

Strategies Conclusion

The system call strategy depends on the platform. You may want to use different strategy depending on the architecture, and even switch strategy depending on the hardware performance. You might also need more parameter copying on a microkernel than you will need on a monolithic one.

See Also

Threads

- System call implementation

External Links

- System Call on Wikipedia.
- sandpile.org - Re: SYSENTER SYSCALL (http://www.sandpile.org/post/msgs/20003633.htm) - An explanation on using sysenter / sysexit / syscall.
- Asking the kernel how to make a syscall (http://www.pagetable.com/?p=9) - Just the same, with notes on how the L4 microkernel is impacted by this.
- Sysenter Based System Call Mechanism in Linux 2.6 (http://manugarg.appspot.com/systemcall inlinux2_6.html) - Explains the how and why of why Linux changed their System Call procedure for Pentium II+ machines.
- FreeBSD Developers' Handbook - System Calls (http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/x86-system-calls.html) - Discusses System Calls in FreeBSD from the usermode perspective.