

Networking In the Linux Kernel

Lecture 4
System Calls

Announcements

- If you don't submit Assignment 0 by the end of Friday...
 - Talk to me/email me as soon as possible
 - Remember there are no late days in this course
 - This is an important assignment so may be flexible
- Assignment 1 will be posted Monday
- Monday will be an “in class exercise” – bring your computer
 - Will be part of your team assignments score
 - Will help with Assignment 1
 - You can work with 1 other student on In Class Exercises

Announcements

- A gradeable for In Class Exercise 1 will be posted on Monday as well
 - Due at the end of Wednesday
- Individual assignments:
 - Assignment 0, Assignment 1, Assignment 2, peer reviews for Assignments 1 & 2, peer reviews for Projects 1 & 2
- Team assignments:
 - Project 1, Project 2, 3 “in class” exercises

Sources for Today

- Bovet, Daniel P. and Cesati, Marco. Understanding the Linux Kernel, 3rd edition. O'reilly Media Inc., 2006.
 - If you really want to know how the kernel handles the gory details
 - ...you probably want to read Chapter 4 in this book first, then Chapter 10
- Love, Robert. Linux Kernel Development, 3rd edition. Pearson Education Inc., 2010.
 - More geared towards our course
 - Chapter 5

What is a System Call?

- We've answered this quite a few times, but we haven't really gone into any details

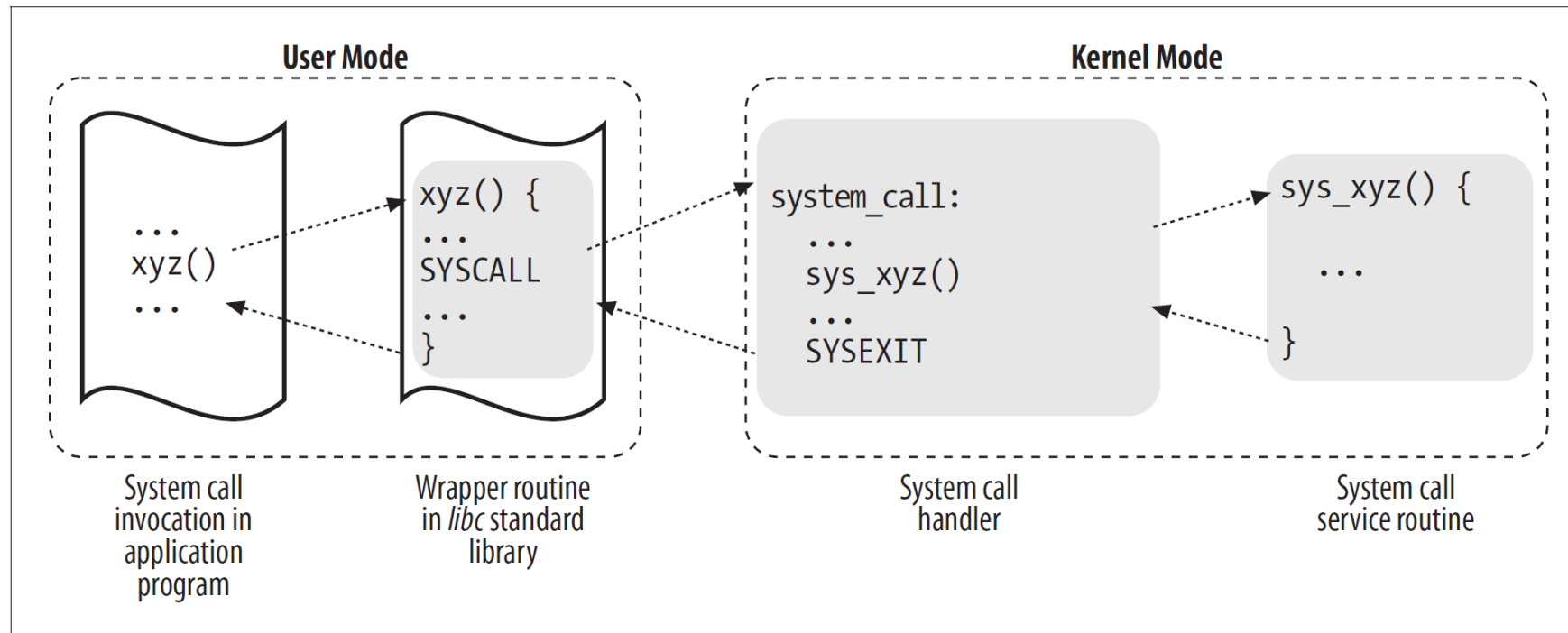


Figure 10-1. Invoking a system call

Key Points

- The user space calls a wrapper
- The compiler handles the wrapper, which may generate additional calls or have logic
- The compiler makes system calls, doing black magic to the stack and registers as needed.
- The kernel handler does black magic as needed to finish the transition to kernel-space, may run logic, and calls the “system service routine”
- The service routine runs and returns
- The kernel handler does black magic as needed and returns to the wrapper (user space again)

Why Have System Calls?

- Abstract hardware interface (for user space)
 - User just calls `read()` regardless of where the file is
 - Or if the file is even really a file or a socket or an OS object...
- Security and stability
 - Resource allocation
 - Permissions
- Virtualized System for Processes
 - Kernel needs full control of resources to do memory management, multitasking, etc.

Why Have an API?

- Removes the need for a 1:1 match between compilers and kernel code

- Example:

main.c printf()

gcc, libc, or other printf()

gcc, libc, or other write()

kernel space write()



Where Do We Need To Care?

- API and library don't matter from the kernel's perspective
- The kernel provides abstracted interfaces
- It does **not** provide rules about how those interfaces should be used.
- If a library's write() does a bunch of extraneous read() calls as well, that's not our problem
- We'll focus on the kernel side

Example: getpid()

[SYSCALL_DEFINE0\(getpid\)](#)

```
{  
    return task_tgid_vnr(current); //returns current->tgid;  
}
```

- Let's not drill down too far, this will bury us in the process/scheduler code very quickly if we delve.
- End result is returning the PID [well, the TGID (thread group id)]

SYSCALL_DEFINE0

```
#define SYSCALL_DEFINE0(sname)
```

```
    SYSCALL_METADATA(_##sname, 0);
```

```
    asm linkage long sys_##sname(void)
```

- Defines a system call with no parameters
 - Calls `sys_getpid()`
 - Return long instead of int for 32/64-bit compatibility
- Let's not think too hard about the assembly
 - What we learned last time is sometimes it's best to trust the macros
 - Understanding what the macro is supposed to do as opposed to the details is the best course of action in most cases
- Architecture-specific x86 version: [[here](#)]

Syscalls With Arguments

```
#define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, _##name, __VA_ARGS__)\n#define SYSCALL_DEFINE2(name, ...) SYSCALL_DEFINEx(2, _##name, __VA_ARGS__)\n#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)\n#define SYSCALL_DEFINE4(name, ...) SYSCALL_DEFINEx(4, _##name, __VA_ARGS__)\n#define SYSCALL_DEFINE5(name, ...) SYSCALL_DEFINEx(5, _##name, __VA_ARGS__)\n#define SYSCALL_DEFINE6(name, ...) SYSCALL_DEFINEx(6, _##name, __VA_ARGS__)
```

- Tricky question: Why only up to 6 arguments?
 - We'll cover that at a very high level later in the lecture

Syscall Numbers

- Every system call gets a **syscall number**
 - Unique to the call
 - Used instead of system call names
- If the call number is invalid, -ENOSYS will be returned via *sys_ni_syscall()*.
 - `errno.h` has definitions for errors
- Syscalls can be architecture specific!
 - *sys_call_table* contains the calls
 - Used to live in `arch/x86/kernel/syscall_64.c`

Syscall Numbers

- Let's not get bogged down in details but...
- arch/x86/entry/syscall_64.c

```
const sys_call_ptr_t sys\_call\_table[] _____cacheline_aligned = {  
    /*  
     * Smells like a compiler bug -- it doesn't work  
     * when the & below is removed.  
     */  
    [0 ... __NR_syscall_max] = &sys_ni_syscall,  
#include <asm/syscalls_64.h>  
};
```

Automatic Generation

- A lot of architecture specific “stuff” gets generated automatically
- This includes vDSO which we won’t discuss
- However, older references refer to an entry.S, that doesn’t exist anymore
 - There is still arch/x86/entry_32.S and entry_64.S
 - If you look in them, there’s no syscall definitions, just entry-point code
- We now look in arch/x86/entry/syscalls/syscall_64.tbl and arch/x86/entry/syscalls/syscall_32.tbl

Syscall Numbers

- A lot of calls are generic, and live in places like `<uapi/asm-generic/unistd.h>`
 - This depends on the architecture
 - UAPI was supposed to take all the "user API" calls and stick them in one place. x86 usually still goes into arch though.
 - If you add to UAPI, need to update the `__NR_syscalls` constant
 - Don't be a slob - put your syscall number ABOVE this constant. (Why?)
- Look at the further reading at the end of the lecture for a good current guide on how to register system calls.
 - Mainly just make sure you update the arch-specific table or the `unistd.h` entry.
 - In this course should be able to just change arch-specific code.

System Call Handler

- The kernel is in a protected “kernel space”, a chunk of memory that user space applications can’t touch
 - If they could, Linux would be extremely insecure, and probably unstable
- A software interrupt signals to switch from user mode to kernel mode
 - User Exception -> Kernel Exception Handler -> Kernel Mode
 - Which exception handler do we use?
 - *int \$0x80* (DEC 128) for x86
 - *sysenter* (Pentium II / AMD legacy) *and* *syscall* (x86_64/AMD) can and should be used when possible, but legacy might not allow for it
- Let’s stick to x86 right now, workflow is similar for x86_64

How To Get the Right System Call

- All the calls enter the kernel the same way
- How do we know which call caused the interrupt?
- %eax has the system call number, put there while still in user space

/*

** This call instruction is handled specially in stub_ptregs_64.*

** It might end up jumping to the slow path. If it jumps, RAX*

** and all argument registers are clobbered.*

**/*

`call *sys_call_table(, %rax, 8)`

- Note: see <https://lkml.org/lkml/2018/1/25/491>

What about on 32-bit?

System Call Table (x86) Types

- From arch/x86/include/asm/syscall.h

```
#ifdef CONFIG_X86_64
```

```
typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs *);
```

```
#else
```

```
typedef asmlinkage long (*sys_call_ptr_t)(unsigned long, unsigned long,  
                                           unsigned long, unsigned long,  
                                           unsigned long, unsigned long);
```

```
#endif /* CONFIG_X86_64 */
```

```
extern const sys_call_ptr_t sys_call_table[];
```

How To Get the Right System Call (cont'd)

- All the calls enter the kernel the same way
- Another place to look is arch/x86/include/asm/ptrace.h at the struct pt_regs definition [\[link\]](#):

```
/*
```

```
 * On syscall entry, this is syscall#. On CPU exception, this is error code.
```

```
 * On hw interrupt, it's IRQ number:
```

```
*/
```

```
    unsigned long orig_ax;
```

- Interesting comment style, `//__i386__` is at the `#else`, but this is `#ifdef __i386__ / #else...` and the `#endif` has `//!__i386__`

Passing Parameters

- We have a SYSCALL_DEFINEx macro
 - What is it doing (fundamentally)?
- Read parameters from registers!
 - How many registers can we use?
 - %ebx, %ecx, %edx, %esi, %edi
 - <https://lwn.net/Articles/18419/>
 - *“I'm a disgusting pig, and proud of it to boot.
Linus”*
- What if we need more?
 - (void*) actually_a_struct_or_union_or_something

Again, Where Do We Care?

- We don't have to really care about the system call handler's behavior
- We need to know limitations like number of parameters
- Mainly we just need to write our service routine and register the call with the system call handler
- So, let's get into the topic of actually adding a system call.

Modern x86 Syscall Definitions

- Let's look at the SYSCALL_DEFINE0 [\[link\]](#)
 - Looks a little different from before!
- What is X64_SYS_STUB0? [\[link\]](#)
 - Another macro invocation
- Finally, what is __SYS_STUB0() [\[link\]](#)
 - This naming scheme is very important! Follow the existing table entries **carefully** on Monday and whenever you make a new syscall entry.
- These three pieces, the table, and the build system make it so we don't explicitly have to do all the messy asmlinkage etc. manually

Design Considerations

- Why are we making a new system call? Can an existing call be adapted?
- System calls should not be multipurpose. Don't multiplex like *ioctl()* (or really *do_vfs_ioctl()*) does.
 - Challenge: Find *sys_ioctl()* implementation.
- Arguments, return values, error codes
 - If we introduce new error codes, we're expanding *errno.h*
 - Minimize arguments
 - Don't change semantics, ideally this call should still look the same 5, 10, or even 30 years down the road!
 - Can we provide backwards compatibility? (see further reading, kernel.org)

Parameter Verification

- Check for...
 - Type and value correctness in parameters
 - Resources are accessible and valid
 - User space pointers were okay (what if we could just give ANY pointer in user space to the kernel and the system call worked?)
 - In user space
 - In the process's space
 - Read/write permissions if needed
 - Since we're in the kernel we *can* access anything, but we should make sure we only access what the process should access.

Writing/Reading

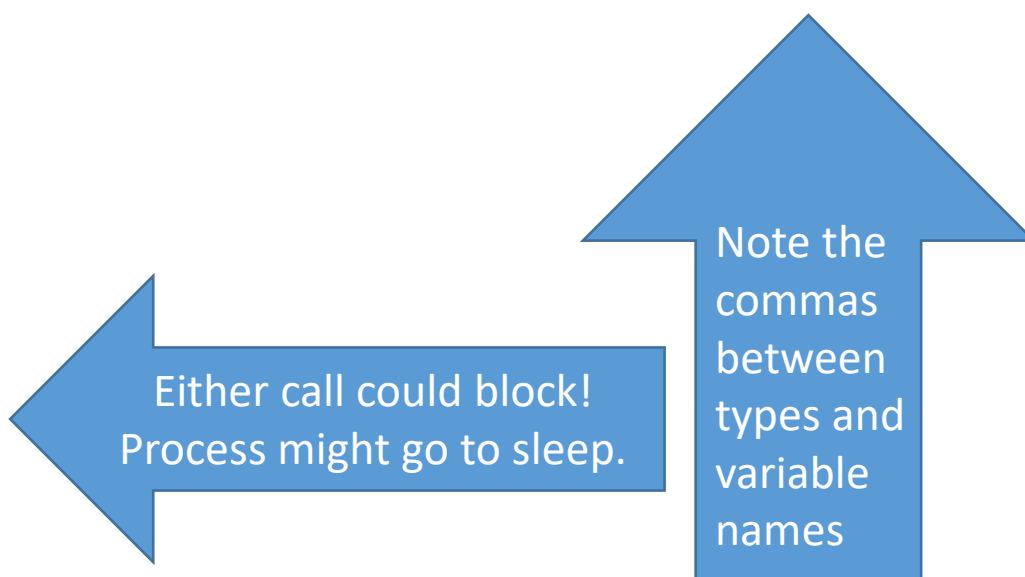
- *copy_to_user()*: writing to user space
- *copy_from_user()*: reading from user space
- Both have three parameters: destination, source, and size (in bytes)
- If an error happens, number of bytes uncopied is returned, otherwise 0.
- A handler typically should return -EFAULT if one of these calls fails.

Love example: *silly_copy()*

```
SYSCALL_DEFINE3(silly_copy, unsigned long *, src, unsigned long *, dest, unsigned long, len)
{
    unsigned long buf;
    if(copy_from_user(&buf, src, len))
        return -EFAULT;

    if(copy_to_user(dest, &buf, len))
        return -EFAULT;

    return len;
}
```



Either call could block!
Process might go to sleep.

Note the
commas
between
types and
variable
names

Permissions

- There used to be a function called *suser()*
 - Guesses on what it did?
 - Checks if the process has superuser privileges (e.g. is *root*)
- Now we have *capable(flags)*
 - Flags defined in [<uapi/linux/capability.h>](http://uapi/linux/capability.h)
 - Take a peek at it sometime
 - Some are more relevant to us, like CAP_NET_BIND_SERVICE
 - Others less so, like CAP_SYS_BOOT
 - Call `capable(FLAGS)`, get true or false back

Process Context

- When a system call is running, the process with the system call currently has context (remember scheduling terms?)
- This means that the kernel code can sleep
 - Could happen from blocking
 - Could call *schedule()* [[2005 article on it if you're curious](#)]
- Interesting problem:
 - Let's suppose our *sys_foo()* needs to sleep because the disk is busy
 - Another process becomes active, and also calls *sys_foo()*
 - Are we re-entrant?

Seeing the CallFrom User Space

- C library provides support for existing calls
- If we make a new call, libc won't know about it
- We used to have to call `_syscallN()` where N was the number of arguments
 - But what we do in `unistd.h` or our arch-specific entry with `_syscall()` is the modern version of this, so we're already done!
- We don't need to understand why this works, just that it's `_syscall(call_number, service_function, type1, var1, type2, var2, ... , typeN, varN)`
- May need to add an `asm` linkage to `<linux/syscalls.h>`

Invocation From Userspace

- Remember, normally glibc uses wrappers
- Ultimately the only thing we can really pass is an interrupt with a register specifying the syscall number
- Manually have to define your own number
- You can use *syscall(2)* to make system calls.

```
#define _GNU_SOURCE      /* See feature_test_macros(7) */
```

```
#include <unistd.h>
```

```
#include <sys/syscall.h> /* For SYS_XXX definitions */
```

```
long syscall(long number, ...);
```

Gross Details

- How do we support both syscall/sysenter and int \$0x80 in the same code base? [check arch/x86/entry/entry_64_compat.S]
- What is a vsyscall?
- What does syscall/sysenter do?
 - MSR (Model Specific Registers), %eip, %esp...
- From a “big picture” standpoint, what’s going on with memory and stacks?
- If you’re curious...
 - Read code, read the Bovet book chapter, read online
 - Not stuff we’re going to tackle in this course

When (Not) To Use Syscalls

- System calls should do just one thing
- They should not be too similar to an existing system call in purpose
- You need a number, so this can get messy when you try to merge into the main kernel tree (again, see kernel.org link in additional reading)
- It's hard to work with system calls using scripts / browsing the filesystem
- They should be super-duper-future-proofed

Additional Reading (optional)

- If you're curious about tracing through the system calls, Andries Brouwer has some notes that are now 18 years old. The process might still be helpful though. Just keep in mind the kernel has been reorganized since then, so there may be subtle differences...

<https://www.win.tue.nl/~aeb/linux/lk/lk-4.html>

Or Next Time

- We'll cover interrupts, tasklets, etc. next Thursday, and we'll cover timers a week from Monday.
- Assignment 1 will be out on Monday so you have ~2 weeks (due Friday September 24th). It will involve designing and writing a new system call. The in-class exercise will help.
- In-class exercise may take longer, but is designed to fit within lecture period (110 minutes) or less. Kernel development can be slow!
- See further reading if you're curious. kernel.org article is a really good one for doing your own system calls!

Further Syscalls Reading

- <https://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls/>
- <https://www.kernel.org/doc/html/v4.14/process/adding-syscalls.html>
- <https://lwn.net/Articles/507794/>
- <https://www.linuxjournal.com/content/creating-vdso-colonels-other-chicken> (about vDSOs, note it's talking about 2.6.37)