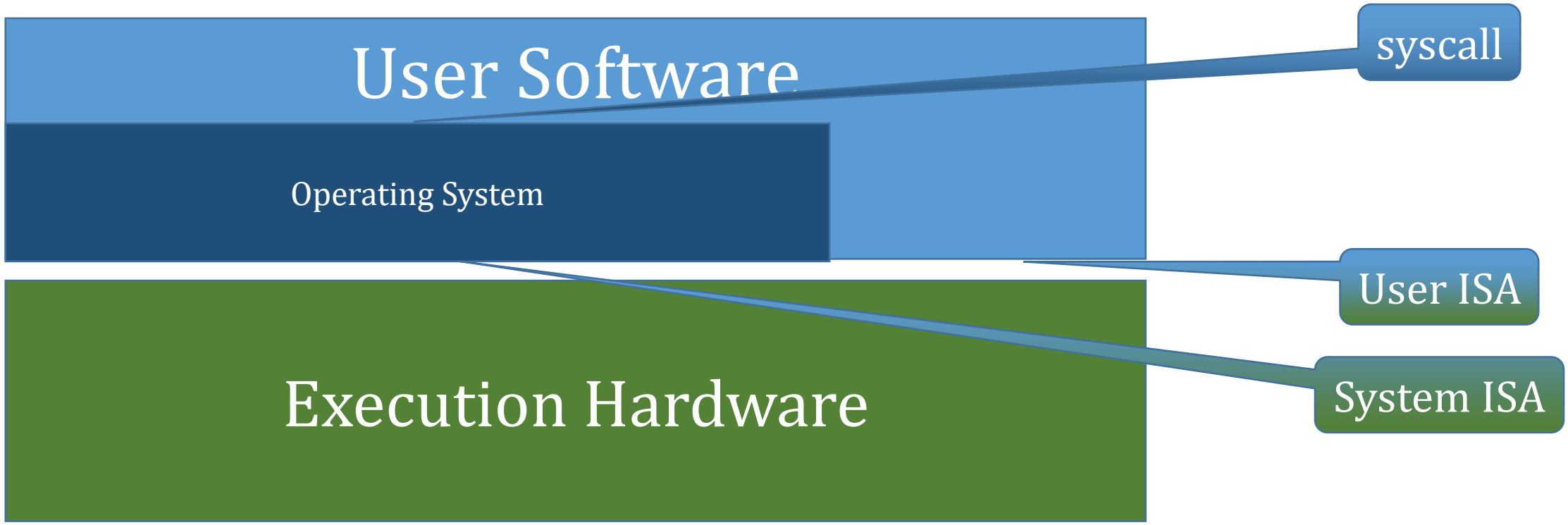
System Calls

Modern Operating Systems, by Andrew Tanenbaum

Chap 1.6.3, 10.3.2, 10.7.2

Interfaces: Operating System

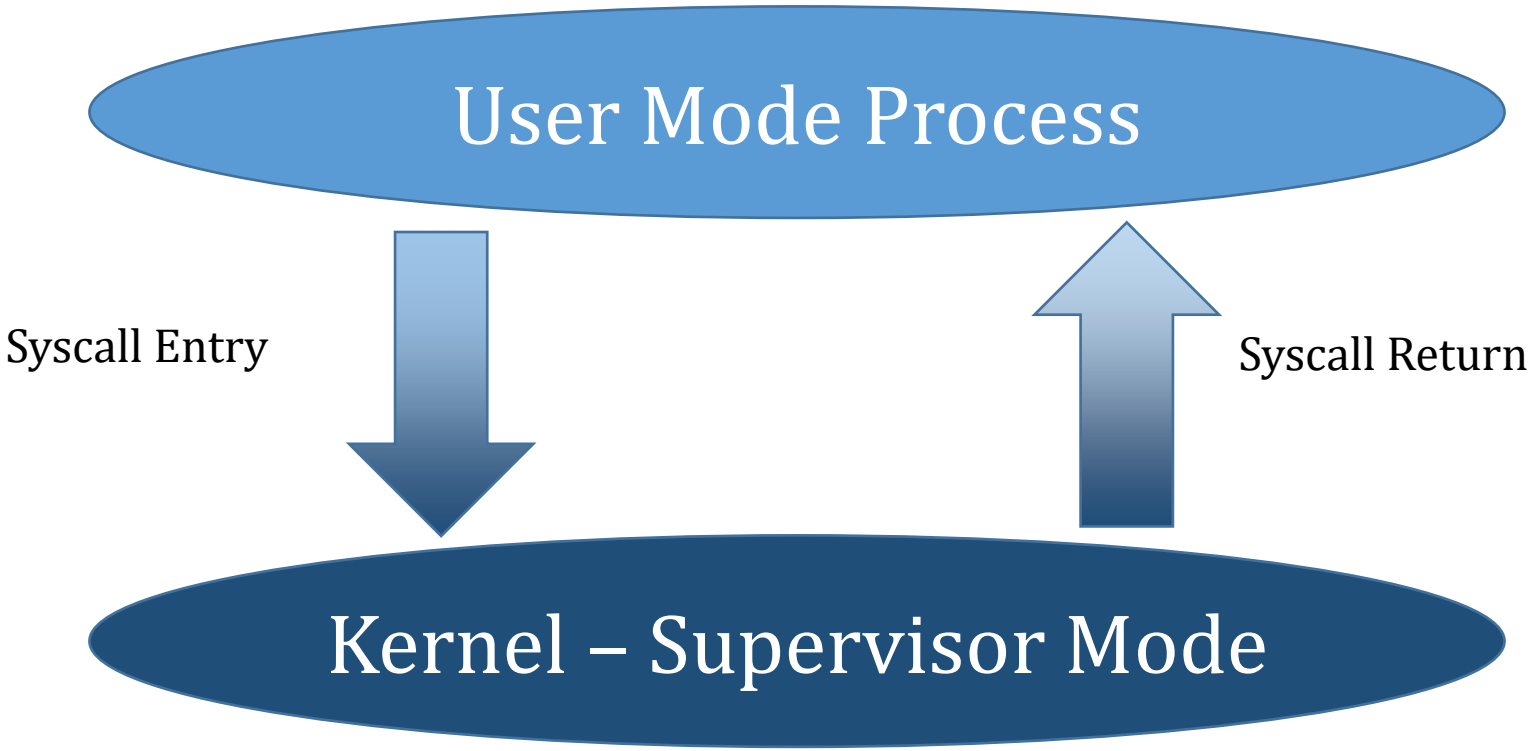


System Calls (Defined)

- User level code - limited privilege code in a user address space
- Kernel code – full privilege code in the kernel address space

- Question: How do we transfer control to execute kernel functions on behalf of a user?
  - Enable kernel mode code to run
  - Enable kernel full privilege
  - Enable kernel access to user address space as required
  - Prevent user from accessing full privilege
  - Return to user once kernel function is complete

System Calls

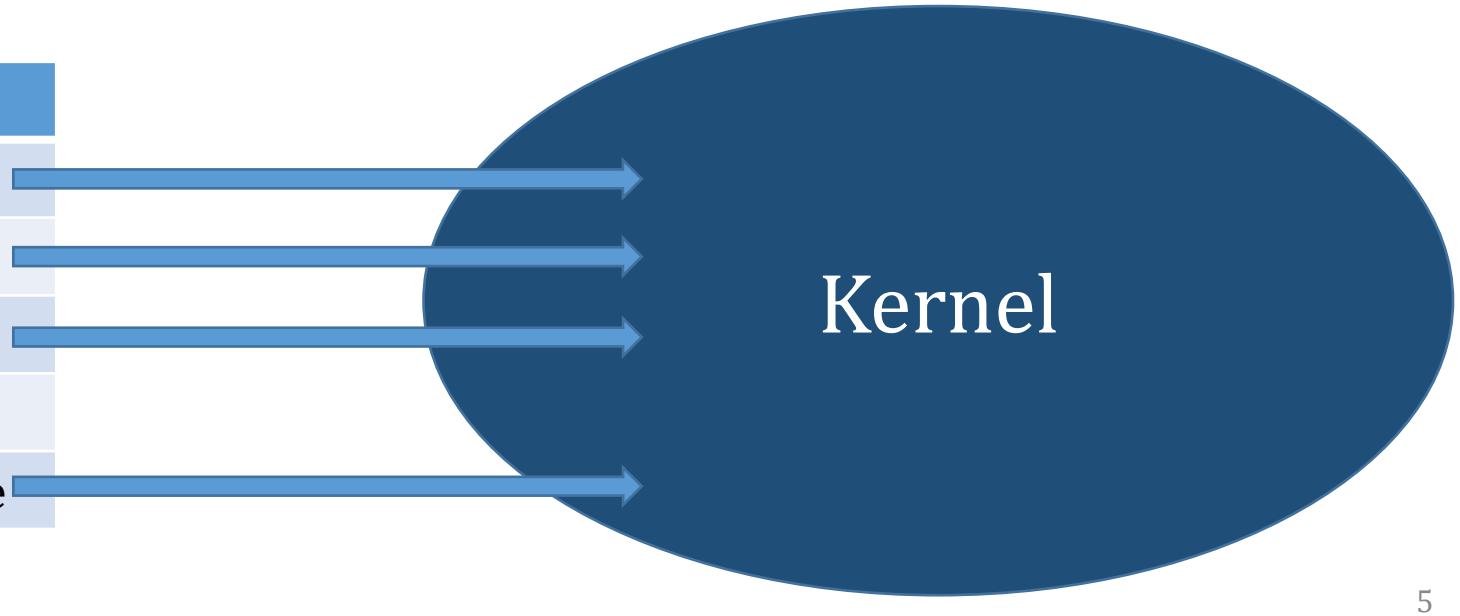


System Call Table

- Pre-defined list of kernel functions available to the user
  - Restrict user access to kernel functions to just this list!

- Each kernel function is numbered – index into the system call table

Index	Function
0	read
1	write
2	open
…	…
313	finit_module



System Call Invocation

Entry

1. System call invoked via special SYSCALL instruction
   - SYSENTER/int 0x80/lcall7/lcall27 etc.
   - Syscall number and arguments pass via registers and optionally stack
2. CPU saves process execution state
3. CPU switches to higher privilege mode & jumps to kernel entry point
4. OS: invokes function at system_call_table[syscall_number]
   - For performance, usually in the execution context of the calling process, but sometimes in a separate context for better security
5. If syscall involves blocking, calling process may be blocked

Return

6. When syscall is complete, the calling process is moved to ready state
7. Saved process state is restored
8. CPU switches back to user privilege using SYSEXIT/iret instructions
9. Process returns from system call and continues

System Library Wrappers

- OS Writers normally provide a library of system call wrappers
  - e.g. libc, glibc, etc.

- Wrapper functions hides the low level details of:
  - Preparing arguments
  - Passing arguments to kernel
  - Switching to supervisor mode
  - Fetching and returning results to application

- Reduce OS dependency – increase portability

Implementing System Calls

Writing the System Call Handler

- Write the system call as a kernel function
  - Be careful when reading/writing user space – use copy_to_user() or copy_from_user() routines which check for you
- Use the "asmlinkage" macro to write code in C
  - Example with integer return value and no arguments

```
asmlinkage int sys_foo(void) {
    printk( KERN_ALERT "I am foo. UID is %d\n",current->uid);
    return current->uid;
}
```

  - Example with integer return value and one primitive argument

```
asmlinkage int sys_foo(int arg) {
    printk( KERN_ALERT "This is foo. Argument is %d\n",arg);
    return arg;
}
```

Example System Call Handler

```
asmlinkage long sys_close(unsigned int fd) {
        struct file *filp;
        struct files_struct * files = current->files;
        struct fdtable *fdt;
        spin_lock(&files->file_lock);
        fdt = files_fdtable(files)
        if (fd >=fdt->max_fds) goto out_unlock;
        filp = fdt->fd[fd]
        if (!filp) goto out_unlock;
        …
out_unlock:
        spin_unlock(&files->file_lock);
        return -EBADF;
}
```

current is a pointer to caller info

Verify argument passed in from user!

Return a negative errno if there is a problem

Example System Call Handler

```
asmlinkage ssize_t sys_read (unsigned int fd, char ___user *buf, size_t count) {
    …
    if (!access_ok( VERIFY_WRITE, buf, count)) return -EFAULT;
    …
}
```

call-by-reference argument user-space pointer

Verify argument passed in from user!

Return a negative errno if there is a problem

Example System Call Handler

```
asmlinkage int sys_foo (void) {
        static int count = 0;
        printk( KERN_ALERT "Hello World! count=%d\n",count++);
        return -EFAULT;
}

EXPORT_SYMBOL(sys_foo);
```

Don't forget to export sys_foo!

What happens to this return value?

Update Kernel's Syscall Table

- Create an entry in the kernel's system call table
  - Kernel's system call table is built from syscall_64.tbl
  - In the Linux source at arch/x86/entry/syscalls/syscall_64.tbl
  - Each entry has 4 fields:
    - Syscall number – pick one greater than the last one there
    - application binary interface (abi) : "common"/"64", or "x32" (we will use "common")
    - Name of the entry – e.g. foo
    - function pointer of the module function that implements the system call – e.g. sys_foo

Example syscall_64.tbl

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read            __x64_sys_read
1       common  write           __x64_sys_write
2       common  open            __x64_sys_open
3       common  close           __x64_sys_close
…
546     x32     preadv2         __x32_compat_sys_preadv64v2
547     x32     pwritev2        __x32_compat_sys_pwritev64v2
# and our new entry…
548     common  foo             sys_foo
```

User invocation of syscall

- Use the syscall(…) library function (do a "man syscall" for details)
- For instance:

```
#define __NR_sys_foo 548
ret = syscall(__NR_sys_foo); // For no argument foo
or
ret = syscall(__NR_sys_foo,arg); // For one argument foo
```

See IBM Developer System Call Tutorial

Example User Program

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <linux/unistd.h>
#define __NR_sys_foo 548
int main(void) {
        int ret;
        while(1) {
                ret = syscall(__NR_sys_foo);
                printf("ret = %d errno = %d\n", ret, errno);
                sleep(1);
        }
        return 0;
}
```

Define the syscall number
Standard syscalls are in linux/unistd.h

Making the system call

syscall sets errno!