System calls — Kernel and userspace frontier

Meeting 5

LKCAMP - Round 3 - Apr 23, 2019

Index 1

Kernelspace × userspace

For security reasons, code is either run on *Userspace* or *Kernelspace*.

Kernelspace

- Only the kernel runs here
- Access to the whole instruction set
- Access to all of the machine's resources (e.g. files, memory, devices)

Userspace

- All of the other programs run here
- Access to a subset of the instruction set
- Access only to its memory space

Services for userspace programs

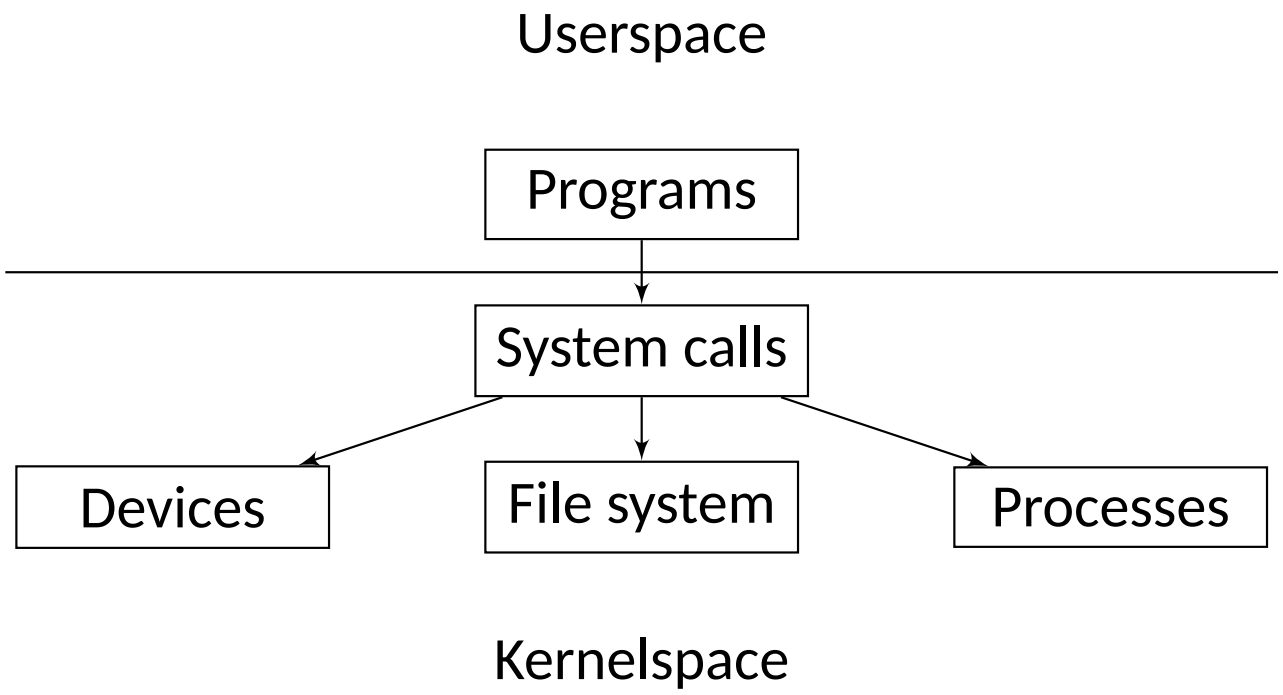
But programs frequently need access to things only allowed for kernelspace:

- Reading a file
- Allocating more memory
- Writing data to a device
- Checking the current time from the clock

Therefore, there needs to exist mechanisms to request those from the kernel.
The main one is the *system call* (a.k.a. syscall).

Index 2

Userspace ⟶ Kernelspace



Making system calls

In order to request a service from the kernel, a process needs to:

1. Pass some parameters to the kernel (i.e. what it wants and how it wants it)
2. Pass the execution to the kernel

System call arguments

The arguments for a syscall can be separated in:

- ID
  - Identifies the type of system call (i.e. what is being requested)
- Other arguments
  - Specify the necessary data for the system call
  - Quantity and type are specific to the ID

The arguments are passed by setting registers.

System calls IDs (x86_64)

File arch/x86/entry/syscalls/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
4       common  stat                    __x64_sys_newstat
5       common  fstat                   __x64_sys_newfstat
6       common  lstat                   __x64_sys_newlstat
7       common  poll                    __x64_sys_poll
8       common  lseek                   __x64_sys_lseek
```

System calls arguments (x86_64)

File arch/x86/entry/entry_64.S

```
/*
 * 64-bit SYSCALL instruction entry. Up to 6 arguments in registers.
 *
```

⋮

```
 * Registers on entry:
 * rax  system call number
 * rcx  return address
 * r11  saved rflags (note: r11 is callee-clobbered register in C ABI)
 * rdi  arg0
 * rsi  arg1
 * rdx  arg2
 * r10  arg3 (needs to be moved to rcx to conform to C ABI)
 * r8   arg4
 * r9   arg5
 * (note: r12-r15, rbp, rbx are callee-preserved in C ABI)
 *
 * Only called from user space.
```

Setting the registers — wrapper functions

Registers are dependent on the architecture of the machine.
Code for accessing registers also needs to be architecture-specific (using assembly).
Solution: wrapper functions provided by libraries

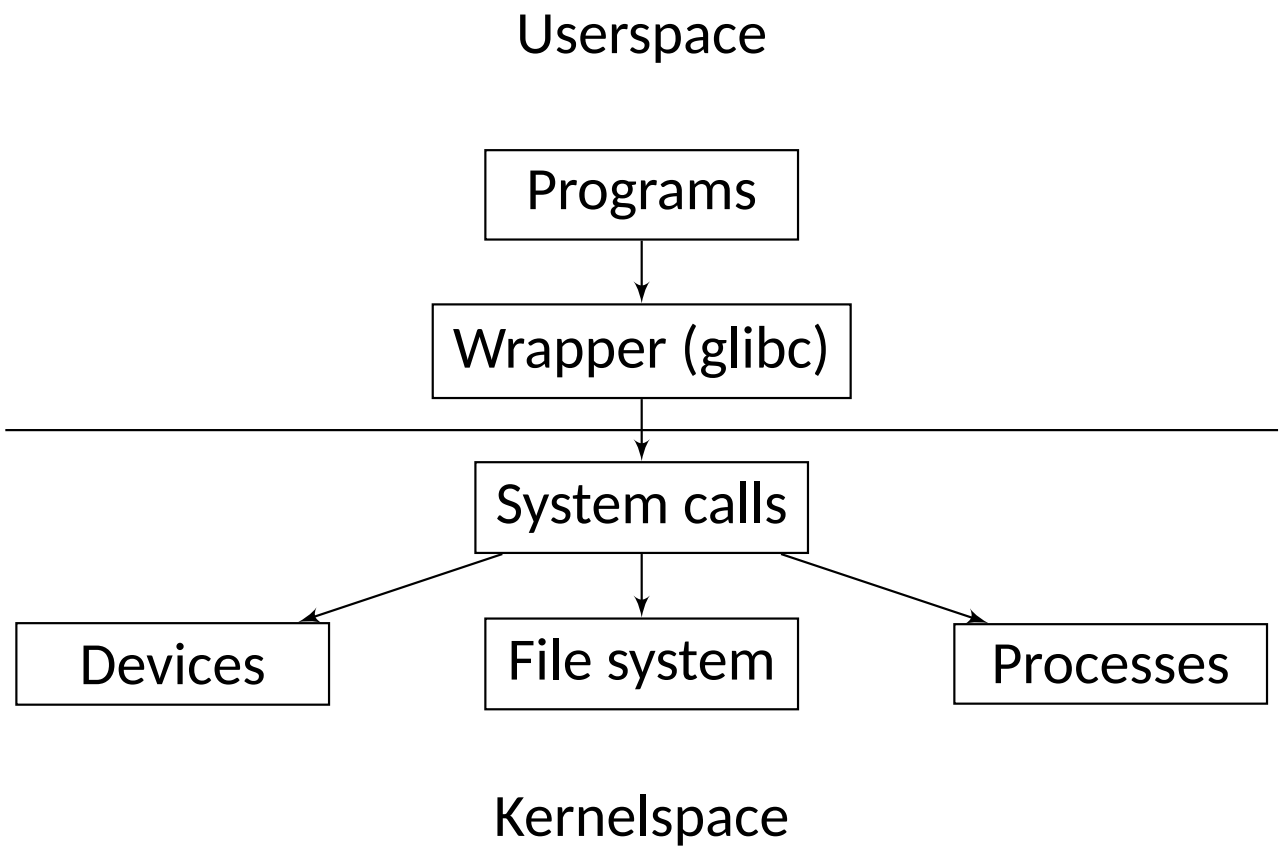
- Encapsulates the architecture-specific part, so the program calling it can be architecture-independent
- Makes it possible for code to use syscalls and still be portable

Most common: GNU C Library — *glibc*.

Syscalls arguments with wrappers (x86_64)

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count
2	sys_open	const char *filename	int flags	int mode
3	sys_close	unsigned int fd		
4	sys_stat	const char *filename	struct stat *statbuf	
5	sys_fstat	unsigned int fd	struct stat *statbuf	
6	sys_lstat	fconst char *filename	struct stat *statbuf	
7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs
8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin
...				
10	sys_mprotect	unsigned long start	size_t len	unsigned long prot
11	sys_munmap	unsigned long addr	size_t len	
12	sys_brk	unsigned long brk		
...				
15	sys_rt_sigreturn	unsigned long __unused		
16	sys_ioctl	unsigned int fd	unsigned int cmd	unsigned long arg

Userspace ⟶ Kernelspace (with wrappers)



Syscall execution

Since kernel code is in another memory space, it can’t be directly called.

One way to make the kernel execute is by causing an interrupt.

Interrupts

Interrupts are signals received by the processor that change the CPU execution flow.

They can be classified into:

- Asynchronous
  - Unrelated to the running code
  - Generated by an external event (hardware)
  - E.g. keystroke on the keyboard, incoming data from hard drive
- Synchronous
  - Related to a specific instruction in the code
  - E.g. division by zero, invalid access to memory, syscall instruction

Interrupt vector table

No matter the nature, interrupts are treated by using the interrupt vector table.
The interrupt vector table is composed of:

- A fixed position in memory
- An index for each type of interrupt
- The pointer to the interrupt handler (a function that handles the interrupt) for each interrupt type

The addresses of the interrupt handlers are configured by the kernel.

Syscalls for configuring drivers

It is now clear why syscalls are needed and how they work, so let's now consider how to configure a driver using them.
Since drivers are very diverse, a special system call is used to configure them: IOCTLs.

Index 3

IOCTLs

IOCTLs are syscalls for Input Output Control.

They are associated to a file e.g. `/dev/video0`, which acts as a communication point to the driver.

IOCTL syscall parameters:

%rax	System call	%rdi	%rsi	%rdx
16	sys_ioctl	unsigned int fd	unsigned int cmd	unsigned long arg

- `fd` ⟶ considered a “driver id”
- `cmd` and `arg` ⟶ free interpretation by the driver

IOCTL libc example

Example:

```
fd = open("/dev/example0", O_RDWR);
ioctl(fd, 8, "open first");
```

The driver is free to choose:

- What command 8 does
- What the argument means
  - If it is a pointer to a string, an integer or a more complex structure

IOCTLs API

If each driver defined its own API, it would be chaos.
For this reason, since drivers in each subsystem should be relatively similar, an IOCTL API is defined per subsystem e.g.

- V4L2/Media
- DRM
- Block
- Joystick

IOCTLs drawbacks

Although IOCTLs work great, they also have some drawbacks:

- Meant to return immediately
- Initiated from userspace only
- Polling to check for state changes

Solution: Netlink

Index 4

Netlink — Socket based

Netlink is socket based, which means it has the same features as sockets:

- Full-duplex
- Uses socket interface
  - `sendto()`
  - `recvfrom()`
  - `poll()`
- Waits for new messages (avoid polling)
- Asynchronous execution → queue command, answer later
- Multicast groups
- UDP

Netlink

Notifications from the kernel, e.g.

- Hotplug events (UEVENT)
- Network interfaces (create/delete/up/down)
- Errors

Userspace async requests:

- Create/delete route in the route table

TCP x Netlink socket comparison

TCP

```
struct sockaddr_in sa

int fd = socket(AF_INET, SOCK_STREAM, 0);

sa.sin_family = AF_INET;
sa.sin_port = htons(1234);
sa.sin_addr.saddr = INADDR_ANY;
bind(fd, (struct sockaddr *) &sa, sizeof(
      sa));
```

Netlink

```
struct sockaddr_nl sa

int fd = socket(AF_NETLINK, SOCK_RAW,
      NETLINK_ROUTE);

sa.nl_family = AF_NETLINK;
sa.nl_pid = get_pid();

bind(fd, (struct sockaddr *) &sa, sizeof(
      sa));
```

Socket creation arguments:

```
int socket (int domain, int type, int protocol);
```

IOCTLs and Netlink alternative

IOCTLs and Netlink require coding.

There is an alternative that doesn't require coding, relying on the filesystem hierarchy instead: Configfs.

Index 5

Configfs

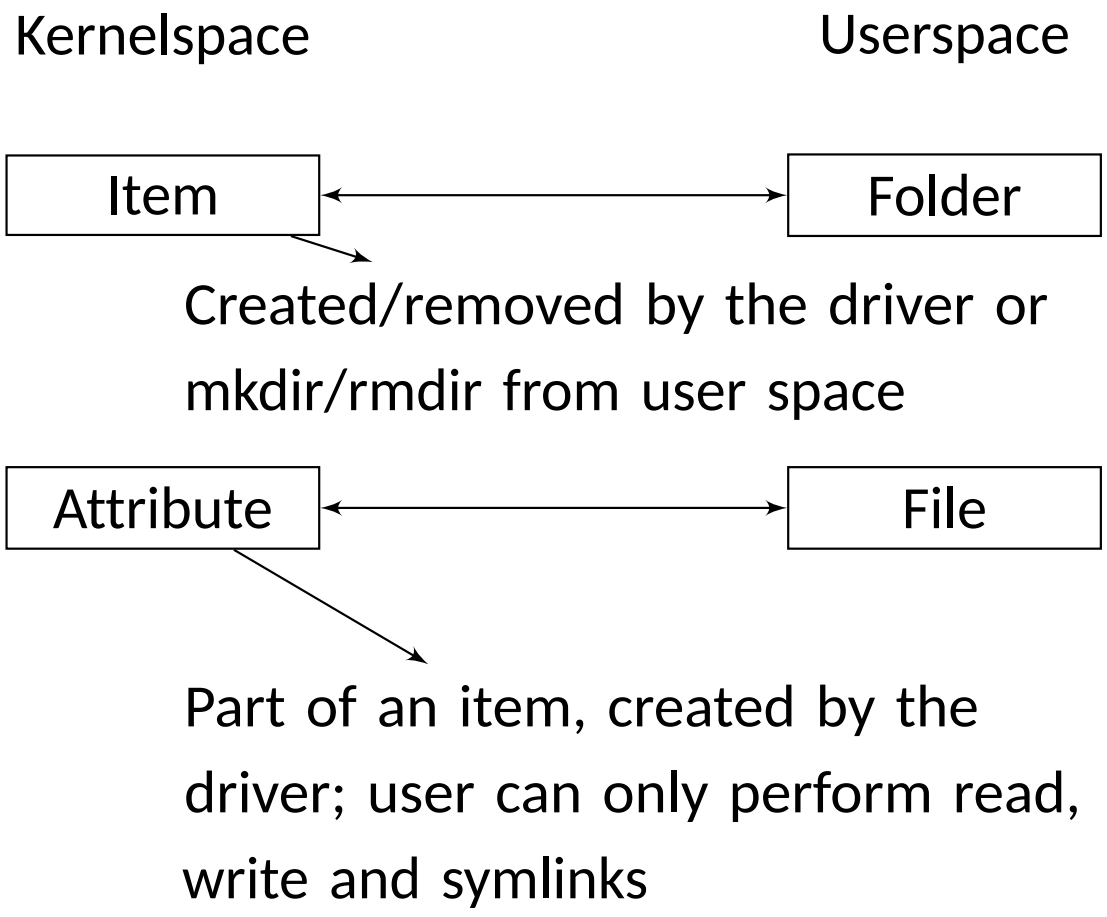
Drivers are configured by *items* and *attributes* exposed through filesystem.
By using the filesystem, well known tools can be used for configuring the drivers:

- `mkdir/rmdir` for manipulating *items*
- `echo/cat` for manipulating *attributes*

Organization through filesystem hierarchy provides:

- Arbitrarily deep tree
- Parent-child relationship for *items*
- Symbolic links for other relationships
- Interpretation of *item* name by the driver

Configfs anatomy



Configfs example: Fakevi

Our fake video driver (fakevi) has the following characteristics:

- Emulates a real webcam
- Outputs a colored bar image
- Useful for application developers who don't own a real webcam
  - Test if the application detects new webcams (hotplug uevents)
  - Test if the application detects the image brightness correctly

Configfs example: Fakevi commands

```
> ls /dev/video0
ls: cannot access '/dev/video0': No such file or directory
> mkdir /configfs/fakevi/my_fake_device_0
> ls /dev/video0
/dev/video0
> tree /configfs/fakevi/my_fake_device_0/
.
|_ image/
|   |_ bright
|   |_ saturation
|   |_ hue
|   |_ shade
|_ filter
> cat /configfs/fakevi/my_fake_device_0/image/bright
22
> echo 42 > /configfs/fakevi/my_fake_device_0/image/bright
> cat /configfs/fakevi/my_fake_device_0/image/bright
42
> echo "sepia" > /configfs/fakevi/my_fake_device_0/filter
> rmdir /configfs/fakevi/my_fake_device_0
```

Filesystems similar to configfs

There are other filesystems in the kernel that are similar to configfs:

- Procfs (/proc/)
  - Processes information
  - Cpu information
    - Memory usage
    - Network
    - Cached buffers
- Sysfs (/sys/)
  - Devices
  - Modules
  - Buses
- Debugfs (/debugfs/)
  - For debugging, no specific rules
  - E.g. dynamic debugs, enable specific debug prints

Bibliography

[1] *LKCAMP M5 - Round 2*. Available at https://www.youtube.com/watch?v=ZW2vWBlhyUg.

[2] *System call - Wikipedia*. Available at https://en.wikipedia.org/wiki/System_call

[3] *User space - Wikipedia*. Available at https://en.wikipedia.org/wiki/User_space

[4] *Configfs - an introduction*. Available at https://lwn.net/Articles/148973/

[5] *ioctl - Wikipedia*. Available at https://en.wikipedia.org/wiki/Ioctl