Linux Kernel Programming

Redha Gouicem and Julien Sopena

Lecture 04:

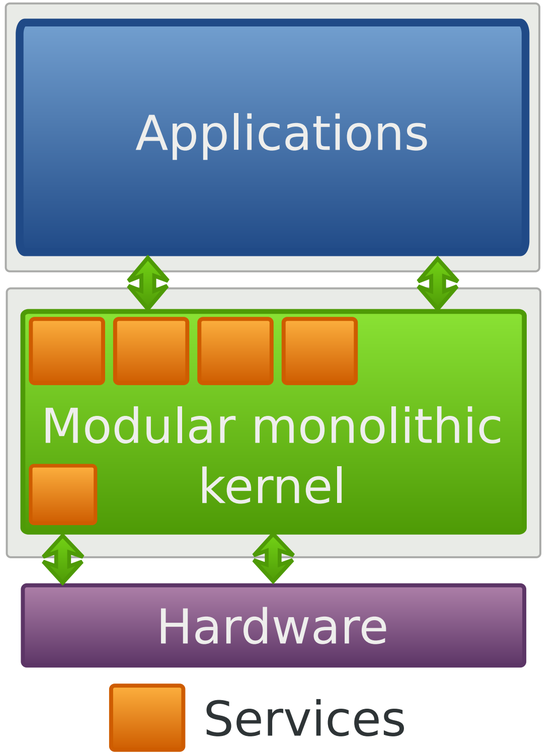
User-kernel Communication

Overview

In monolithic architectures, kernel and user programs are run in different permission levels: **kernel space** and **user space**.

There needs to be communication between both of these spaces.

Multiple mechanisms are available in the Linux kernel, and choosing the right one is a common discussion on the mailing list.



Communication mechanisms:

- Module parameters
- Pseudo file systems
- Sockets
- System Calls
- ioctl

Pseudo File Systems

In-Memory File Systems

An in-memory file system, or **ramfs**, is a file system not backed by a storage device.
The data stored on it is completely in memory or computed when accessed.

The Linux kernel provides a set of **pseudo file systems** that are ramfs representing kernel data or configuration.
They *usually* have a semantic where one file represents one value.
Each pseudo file system has slightly different semantics and answer to different needs: *procfs, sysfs, configfs, debugfs, …*

Pros: User space programs can access these files with the standard POSIX file API: `read` and `write`.
You can thus use regular shell programs such as `cat` and `echo` to read/write from/to them.

Cons: These mechanisms are synchronous from user to kernel, but asynchronous in the other direction,
i.e., user space applications cannot be notified when the value represented by a file changes in memory.

Using Pseudo File Systems

A pseudo file system is a component of the kernel that has to be enabled and mounted before being used.

1. Check that your kernel has been built with the needed pseudo file system

```
cat .config | grep CONFIG_DEBUG_FS
CONFIG_DEBUG_FS=y
```

2. Mount the pseudo file system

```
mount -t debugfs none /sys/kernel/debug
```

3. Read a value by reading from a file

```
cat /sys/kernel/debug/sched/wakeup_granularity_ns
4000000
```

4. Modify a value by writing to a file

```
echo 3000000 > /sys/kernel/debug/sched/wakeup_granularity_ns
```

> **Tip**
>
> You might need to have *root* privileges to read/write to/from these special files.
>
> - Reading: `sudo cat <file>`
> - Writing: `echo <value> | sudo tee <file>`

procfs

The oldest pseudo file system, mounted in `/proc`.

Goal: Export information about processes.

Since its creation, it has been used for more than that, exporting kernel data from various subsystems.

Using it is now discouraged for anything unrelated to processes.

Pro: most widely documented

Con: no real structure enforced

The *procfs* provides two APIs:

- **Legacy procfs API**: simple, but each file is limited to one page (`PAGE_SIZE`, usually 4 KB)
- `seq_file` API: more complex, but allows larger data to be exported with a list of buffers

procfs: Example

Defining a `read` function and declaring a `struct file_operations`:

```
 1 static int system_enabled;
 2
 3 static ssize_t system_state_read(struct file *file, char __user *buf,
 4                                  size_t count, loff_t *ppos)
 5 {
 6     const char *tmp = system_enabled ? "The system is enabled\n"
 7                         : "The system is disabled\n";;
 8     return simple_read_from_buffer(buf, count, ppos, tmp, strlen(tmp));
 9 }
10
11 static const struct file_operations system_state_fops = {
12     .open = simple_open,
13     .read = system_state_read,
14     .llseek = noop_llseek,
15 };
16 static struct proc_dir_entry *system_state_proc_dir;
```

Instantiate a *procfs* file at `/proc/my_state`:

```
18 static int system_state_init(void)
19 {
20     system_state_proc_dir = proc_create("my_state", 0, NULL,
21                                         &system_state_fops);
22     return 0;
23 }
24 module_init(system_state_init);
25
26 static void system_state_exit(void)
27 {
28     remove_proc_entry("my_state", NULL);
29 }
30 module_exit(system_state_exit);
```

sysfs

Successor to the *procfs*, mounted in `/sys`.

Goal: Store information about subsystems, hardware devices, drivers, …

This should be the default choice!

Pros:

- Hierarchical topology, information is arranged logically, by subsystem, component, etc…
- Provides a set of mechanisms to free memory and recursively destroy directories

Cons:

- Complex
- Each file represents one single piece of data
- A piece of data cannot be larger than one page (`PAGE_SIZE`)

sysfs: Directories

The `struct kobject` is at the heart of the *sysfs*:

- Each directory corresponds to a *kobject*
- A file in the sysfs **is not** a kobject

Most important fields:

```
1 struct kobject {
2     const char               *name;      // name of the directory
3     struct kobject           *parent;    // kobject of the parent directory
4     struct kset              *kset;      // the collection of kobjects this object belongs to
5     struct kref              kref;       // reference counter, used to free the memory properly
6     const struct kobj_type   *ktype;     // type of the object, with functions pointers to manipulate it
7     /* ... */
8 };
```

To create a *kobject* and add it to the *sysfs*, you can use this functions:

```
struct kobject *kobject_create_and_add(const char *name, struct kobject *parent);
```

Don't forget to cleanup your kobjects and their files when they are not needed anymore with:

```
void kobject_put(struct kobject *kobj);
```

Caution

This works for simple cases (most likely what you want to do). For more complex scenarios, there are other functions to initialize, create and register *kobjects*. Check out the documentation for more information!

sysfs: Files

Kobject attributes

Each file in the *sysfs* corresponds to one single value and is associated with an instance of a `struct kobj_attribute`.

```
1 struct kobj_attribute {
2     struct attribute attr;  // file information (name, permissions)
3     ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
4     ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count);
5 };
```

Kobject attributes can be created with the following macro:

```
#define __ATTR(_name, _mode, _show, _store)
```

You can also create group of attributes to have multiple files in the same directory:

```
1 static struct attribute *attrs[] = {
2     &foo_attribute.attr,
3     &bar_attribute.attr,
4     NULL,
5 };
6
7 static struct attribute_group attr_grp = {
8     .attrs = attrs,
9 };
```

Files in the sysfs

Files can be created with the following functions:

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
int sysfs_create_group(struct kobject *kobj, const struct attribute_group *grp);
```

sysfs: Example 1, a Read-only Variable

We want to export the state of our system represented by this `int system_enabled` global variable in a human-readable form in `/sys/kernel/my_state/system_enabled`.

```
1 static int system_enabled;
2
3 static ssize_t system_state_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf)
4 {
5     return snprintf(buf, PAGE_SIZE, "The system is %srunning\n", system_enabled ? "" : "not ");
6 }
7
8 static struct kobj_attribute system_state_attribute = __ATTR(system_enabled, 0400, system_state_show, NULL);
9 static struct kobject *my_state_kobj;
```

sysfs: Example 1, a Read-only Variable (2)

Now, we need to instantiate the *sysfs* file when loading our module and destroy it when unloading.

```
11 static int __init my_state_init(void)
12 {
13     int retval;
14
15     my_state_kobj = kobject_create_and_add("my_state", kernel_kobj);
16     if (!my_state_kobj)
17         goto error_init_1;
18
19     retval = sysfs_create_file(my_state_kobj, &system_state_attribute.attr);
20     if (retval)
21         goto error_init_2;
22
23     return 0;
24
25 error_init_2:
26     kobject_put(my_state_kobj);
27 error_init_1:
28     return -ENOMEM;
29 }
30 module_init(my_state_init);
31
32 static void __exit my_state_exit(void)
33 {
34     kobject_put(my_state_kobj);
35 }
36 module_exit(my_state_exit);
```

sysfs: Example 2, Let’s Add an RW File

```
 1 static int system_enabled;
 2 static u64 clock;
 3
 4 static ssize_t system_state_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf)
 5 {
 6     return snprintf(buf, PAGE_SIZE, "The system is %srunning\n", system_enabled ? "" : "not ");
 7 }
 8
 9 static ssize_t clock_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf)
10 {
11     return snprintf(buf, PAGE_SIZE, "%llu\n", clock++);
12 }
13
14 static ssize_t clock_store(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count)
15 {
16     u64 val;
17     int rc = sscanf(buf, "%llu", &val);
18     if (rc != 1 || rc < 0)
19         return -EINVAL;
20
21     clock = val;
22     return count;
23 }
24
25 static struct kobj_attribute system_state_attribute = __ATTR(system_enabled, 0400, system_state_show, NULL);
26 static struct kobj_attribute clock_attribute = __ATTR(clock, 0600, clock_show, clock_store);
27
28 static struct attribute *attrs[] = {
29     &system_state_attribute.attr,
30     &clock_attribute.attr,
31     NULL,
32 };
33 static struct attribute_group attr_grp = { .attrs = attrs };
```

sysfs: Example 2, Let’s Add an RW File (2)

```
35 static struct kobject *my_state_kobj;
36
37 static int __init my_state_init(void)
38 {
39     int retval;
40
41     my_state_kobj = kobject_create_and_add("my_state", kernel_kobj);
42     if (!my_state_kobj)
43         goto error_init_1;
44
45     retval = sysfs_create_group(my_state_kobj, &attr_grp);
46     if (retval)
47         goto error_init_2;
48
49     return 0;
50
51 error_init_2:
52     kobject_put(my_state_kobj);
53 error_init_1:
54     return -ENOMEM;
55 }
56 module_init(my_state_init);
57
58 static void __exit my_state_exit(void)
59 {
60     kobject_put(my_state_kobj);
61 }
62 module_exit(my_state_exit);
```

configfs

configfs is another ram-based file system offering the converse functionality to *sysfs*, mounted in `/sys/kernel/config`.

While the *sysfs* is a view of kernel objects, *configfs* is a manager of kernel objects,
i.e., it allows to create/destroy kernel objects from user space.

Example: Allowing user programs to create and configure a virtual network devices by doing an `mkdir` in the configfs directory of a driver.

Pro:

- Allows user space programs to manage kernel objects

Cons:

- Complex to set up
- One file equals one value

debugfs

The **debugfs** offers a very flexible and simple API targeted at simplifying kernel development and debugging.

It is mounted in `/sys/kernel/debug`.

Pros:

- Very flexible, no size limit for files
- Very high level and simple API

Cons:

- Only for debugging purposes!

Quick non-exhaustive API tour:

```
struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);
void debugfs_create_u32(const char *name, umode_t mode, struct dentry *parent, u32 *value);
void debugfs_create_str(const char *name, umode_t mode, struct dentry *parent, char **value);
```

debugfs: Example

```
 1 static int system_state;
 2
 3 static struct dentry *my_state_dir;
 4
 5 static int my_state_init(void)
 6 {
 7     struct dentry *new_file;
 8
 9     my_state_dir = debugfs_create_dir("my_state", NULL);
10     if (my_state_dir == 0)
11         return -ENOTDIR;
12
13     new_file = debugfs_create_u8("system_state", 0444, my_state_dir, (u8 *) &system_state);
14     if (new_file == 0) {
15         debugfs_remove_recursive(my_state_dir);
16         return -EINVAL;
17     }
18
19     return 0;
20 }
21 module_init(my_state_init);
22
23 static void my_state_exit(void)
24 {
25     debugfs_remove_recursive(my_state_dir);
26 }
27 module_exit(my_state_exit);
```

Synchronous Communication Mechanisms

Overview

Pseudo file system-based mechanisms are:

- **Synchronous** from user to kernel:
  When a user writes a value, it is changed in the kernel memory when the user program returns to user space;
- **Asynchronous** from kernel to user:
  When an value in kernel memory changes, the user space is **not** notified of this change until the next read.

The kernel provides various synchronous communication mechanisms:

- System calls
- ioctls
- Sockets

System Calls

System calls are the most classic user-kernel communication mechanism.

They allow user space applications to execute privileged kernel code:

- I/Os (disk, network)
- Resource management (threads, memory)
- Communication (signals, IPCs)
- Access to specific hardware

They are the core API of the kernel, with some limitations:

- Each system call is identified by a number defined at kernel compile time
  You cannot add new ones dynamically
- They are a “universal” API, so they need to be the same on all systems [1]

There are two ways of making system calls:

- The “old” interrupt way (x86)
- The “new” `syscall` instruction way (x86, amd64, ARM64)

1. As long as it is on the same architecture, and maybe some kernel configuration options… Don’t quote me on that.

System Calls: The Interrupt Way

System calls are a software interrupt like the others:

1. Place the system call number in a register
2. Place the arguments in the proper registers and/or on the stack
3. Trigger the “system call” interrupt
4. Jump to the “system call” interrupt handler
5. Load the system call table and jump to the index given by the syscall number
6. Execute the system call handler
7. Return the result to user space

System Calls: The Instruction Way

Some architectures provide a specific instruction (`syscall`, `svc`, `sysenter`, ...):

1. Place the system call number in a register
2. Place the arguments in the proper registers and/or on the stack
3. Use the system call instruction
4. Jump to the index given by the syscall number in the syscall table
5. Execute the system call handler
6. Return the result to user space

One level of indirection is bypassed!

> **Tip**
>
> You can find a very detailed (with less omissions) explanation of how syscall work in Linux here!

System Calls: The Linux Application Binary interface

API: *Application Programming Interface*
High-level interface for programmers (function prototypes, data types, …)

ABI: *Application Binary Interface*
Low-level interface for compilers/OS (calling conventions, architecture-specific)

Architecture	syscall#	retval	arg1	arg2	arg3	arg4	arg5	arg6	arg7
Arm EABI	r7	r0	r0	r1	r2	r3	r4	r5	r6
arm64	w8	x0	x0	x1	x2	x3	x4	x5	
mips	v0	v0	a0	a1	a2	a3	a4	a5	
riscv	a7	a0	a0	a1	a2	a3	a4	a5	
x86-64	rax	rax	rdi	rsi	rdx	r10	r8	r9	

System Calls: How to Add One

> **Note**
>
> We'll see that a bit later, when you'll need to do it for an exercise.

ioctl

ioctls are a way to provide custom system calls, mainly for device drivers.

They are called from user space by the `ioctl` system call and provide an additional level of indirection, with each ioctl having a number.

Pros:

- Easy to extend
- Provide a syscall-like interface, *i.e., an arbitrary function call*

Cons:

- Numbers have to stay “forever”, as changing them might break existing user space applications

An *ioctl* is tied to a file. It is registered as a file operation similar to `read` or `write` in the kernel.

For a given device, each *ioctl* has a number generated through a set of macros.

> **Note**
>
> We won’t see the API in details here, you will have a task solely on that in the next exercise.

Sockets

The kernel also provide a socket-based interface with user space called **netlink**.

It is similar to regular sockets in user space, but with the `AF_NETLINK` socket family.

Pros:

- Symmetrical communication

Cons:

- Implementation is a bit more complex than simple read/write semantics