Introduction to the Kernel

An overview of the lower levels of programming

Tazio Ceri

Cynny

May 3, 2017

Presentation plan

An anecdotal definition of Lateral knowledge

The dawn of space exploration

NASA had to design the first space suit for the Apollo missions. They worked for six months, then the whole space suit team went to the museum. They found a room where there were many medieval armors. They were really surprised: armor's joints where somewhat similar to space suit's joints. If they had gone to the museum before the project started, by looking at the armors in advance they would have saved three months of work.

What distinguishes us from the animals?

Animals can have emotions. They can get drunk if they find alcohol. They are also capable of thinking and communicating. Of course, our language allows us to make things complex through abstractions. That's the point! What makes us different is our capability to abstract the shape of a problem and apply that abstraction to other problems, totally different but with a common "shape".

Lateral knowledge and us

- When approaching to a new problem, one never knows what's needed;
- when approaching to an unknown thing or concept, one never knows in advance wheter it will be useful and for what;
- even if your problem has a good and tested solution, you are not able to find it if you don't know what to search;
- that's what lateral knowledge is;
- that's what these talks are for!
- Just be curious, don't be a goat, be a human!

Human or goat?

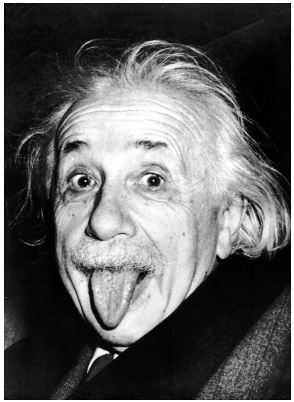

Figure: Human

Figure: Goat

A very simple Unix utility: *cat*

How it works

- It concatenates file's contents and print them to the *stdout*;
- *stdout* is normally the user's monitor.
- How can this happen?
- *cat* knows nothing about monitors, pipes or what goes around.
- What does it use?

System calls

cat example

```
open("/usr/lib64/gconv/gconv-modules.cache", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=26244, ...}) = 0
mmap(NULL, 26244, PROT_READ, MAP_SHARED, 3, 0) = 0x7f55a66bf000
close(3)
```

- *strace* shows as usual what is happening;
- system calls are how normally a process communicates with the kernel;
- the kernel is where all the I/O goes on;
- syscalls are the standard way to jump into the kernel space.
- What is kernel space exactly?

Virtual memory

Paging and interrupt

- Modern CPUs do not use physical memory address;
- every process has an address space, virtual and private, pointed by *CR3*;
- memory is managed in pages, that form a tree;
- that enables swapping and paging, and also **ACL for memory pages**;
- every process shares the kernel space;
- no process normally can access directly kernel memory, even if it's shared;
- kernel memory can be accessed only when a process switch to kernel context;
- usually via a syscall, that is not a normal function call;
- it uses an interrupt (**int 80h**), or **sysenter**, and different call conventions.
- Every process uses a different stack in kernel space.
- Virtual address is mapped to physical address by TLB.

Gates and interrupts

Interrupt context

- An interrupt causes execution of a "function" pointed from the interrupt table;
- can be used to switch from different privilege level;
- the interrupt handler gets executed in a *interrupt context*;
- it has a different stack on some architecture;
- or it can use the stack of the current process;
- the interrupt handler must be totally reentrant;
- it is also called a *atomic context*, at least for *fast interrupts*
- An interrupt source is usually called IRQ.
- Handlers usually dispatch jobs to bottom halves.
- handlers are registered with *request_irq* and unregistered with *free_irq*.

Example of interrupt handler (rtc)

```
static irqreturn_t rtc_interrupt(int irq, void *dev) {
    spin_lock(&rtc_lock);
    /* update the timer */
    spin_unlock(&rtc_lock);
    spin_lock(&rtc_task_lock);
    if (rtc_callback) rtc_callback->func(rtc_callback->private_data);
    spin_unlock(&rtc_task_lock);
    wake_up_interruptible(&rtc_wait);
    kill_fasync(&rtc_async_queue, SIGIO, POLL_IN);
    return IRQ_HANDLED;
}
```

- *wake_up_interruptible* awakes all blocked process on this device;
- *kill_fasync* delivers SIGIO to the process interested in reading rtc.

Implementation of a *syscall*

Syscall generalities

- Arguments are passed through registers;
- memory and buffers are usually copied (because of different context);
- userspace stack, and most ALU registers, are saved;
- kernel space stack is popped;
- *asmlinkage* modifier is needed: we don't want the compiler to optimize arguments;
- syscall is selected by *rax* value;
- that value is an index to a table;
- the handler calls then the selected function.

Dummy Syscall implementation

```
SYSCALL_DEFINE3(silly_copy, unsigned long * src, unsigned long * dst,
    unsigned long len) {
  unsigned long buf[len];
  if (copy_from_user(&buf, src, len)) return -EFAULT;
  if (copy_to_user(dst, &buf, len)) return -EFAULT;
  return len;
}
```

- *copy_from_user* copies memory from userspace to kernelspace;
- *copy_to_user* copies memory from kernelspace to userspace;
- *SYSCALL_DEFINEx* are macros that hide real definition;

Syscall table

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
...
```

- The array of function pointers is generated by a bash script that takes *syscall_64.tbl* as input.
- The syscall handler issues a *call *sys_call_table(,%rax,8)*

Final information

- No syscall can trust user's data, special functions are needed to copy user data;
- Syscalls cannot be implemented as modules;
- Syscall multiplexing is simply bad design.

Starting

- Learn how to install and compile a kernel;
- have a directory with your sources correctly configured;
- install your distribution's sources to have all dependencies working;
- you need more than gcc and GNU make to work with it.

Makefile

```
obj-m+=tcmod.o

all:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

Devices

Two inhabitants of /dev

```
brw-rw---- 1 root disk 8, 1  2 mag 20.52 /dev/sda1
crw--w---- 1 root tty  4, 6  2 mag 20.52 /dev/tty6
```

- A block device (*/dev/sda1*)
- and a char device (*/dev/tty6*)
- a common population of the /dev directory;
- not every kind of device lives there though;
- for example, network interfaces use another abstraction, called **NETLINK** socket.
- Block devices manage I/O in blocks (surprise!), typically they are disks;
- char devices manage I/O in single bytes.

What a char device module need

- When it gets loaded, one (or more) device inside /dev must be created;
- an implementation of functions that get called when we operate on the devices;
- it must have a unique major number that identifies the correct driver;
- it must have one or more minor numbers that identify the devices managed by the driver;
- it must have an implementation of needed functions that get called on the devices.

Module generic code

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tazio Ceri");
MODULE_DESCRIPTION("A simple Linux driver for char devices");
MODULE_VERSION("0.1");

static char *name = "tcmod";
module_param(name, charp, S_IRUGO);
/// Param desc. charp = char ptr, S_IRUGO can be read/not changed
MODULE_PARM_DESC(name, "The name to display in /var/log/kern.log");

static long size_l = 1;
module_param(size_l, long, S_IRUGO);
MODULE_PARM_DESC(size_l, "Size of the queue");
```

Module load/unload functions

Code

```
static int __init tcmod_init(void) { ... }
static void __exit tcmod_exit(void) { ... }

module_init(tcmod_init);
module_exit(tcmod_exit);
```

The *__init* macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point. The *__exit* macro is the same, but at exit time.

Just two readable macros

```
#define __init  __attribute__ ((__section__ (".text.init")))
#define __exit  __attribute_used__ __attribute__ (( __section__(".text.exit")))
```

Load function: allocate numbers

```
#define DEVICENAME "tqueue"
static dev_t dev; // An int that will take major/minor number information

// in tcmod_init
   printk(KERN_INFO "Hello from %s tcmod LKM!\n", name);
   result = alloc_chrdev_region( &dev, 0, 1, DEVICENAME );
   if (result<0)
   {
      printk(KERN_ALERT DEVICENAME" failed to register a major number\n");
      return result;
   }
   printk(KERN_INFO DEVICENAME
      ": registered correctly with major number %d\n", 0);
   majorNumber = MAJOR( dev );
```

Load function: files under */sys* will appear now

```
tqueueClass = class_create(THIS_MODULE, CLASS_NAME);
if ( IS_ERR( tqueueClass ) )
{
   printk(KERN_ERR "Failed class creation\n");
   goto fail_chardev;
}
printk(KERN_INFO DEVICENAME": class ok\n");
ddev = MKDEV(majorNumber, 0);
tqueueDevice = device_create(tqueueClass, NULL, ddev, NULL, DEVICENAME);
if ( IS_ERR( tqueueDevice ) ) goto fail_classdestroy;

printk(KERN_INFO "Device happily created!\n");
```

These functions will make appear the appropriate information (class and device) under */sys* appear.

Load function: */sys* and */dev*

/sys

```
ls /sys/class/classy/tqueue/
    dev  power  subsystem  uevent
```

/dev

```
crw------- 1 root root 247, 0  1 mag 00.18 /dev/tqueue
```

Everything works automagically because of udev, that listens to events under */sys* and create the appropriate device automatically. You could also create them by hand using *mknod* on every filesystem of your choice.

Linking the system to our driver - Part 1

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
```

Linking the system to our driver - Part 2

```
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, loff_t, loff_t, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, in
unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned lo
   unsigned long, unsigned long);
int (*check_flags)(int);
int (*flock) (struct file *, int, struct file_lock *);
ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
   size_t, unsigned int);
ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
   size_t, unsigned int);
```

Linking the system to our driver - Part 3

```
  int (*setlease)(struct file *, long, struct file_lock **, void **);
  long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
  void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
   unsigned (*mmap_capabilities)(struct file *);
#endif
   ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
      loff_t, size_t, unsigned int);
   int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t, u64);
   ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *, u64);
};
```

- Every "method" of the virtual interface has a default;
- you need to implement only what you need;
- every method "maps" to a syscall.

Linking the system to our driver

```
static struct cdev* cdev_s;

// in tcmod_init
cdev_s = cdev_alloc();
cdev_init( cdev_s, &ops ); // ops is our file_operation instance
cdev_s->owner = THIS_MODULE;
result = cdev_add( cdev_s, ddev, 1);
if ( result )
{
   printk(KERN_ERR "Failed cdev add\n");
   goto fail_cdev;
}
```

Concurrency and the kernel

- A normal process has only a fistful of threads;
- and they are all known;
- from a kernel point of view, every single process or thread is unknown;
- and is a source of concurrency.
- Then a number of kernel threads (processes that do not have a userspace) adds up.

Concurrent kernel programming: semaphores

Declaration of a semaphore as Mutex

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

Initialization of a semaphore as Mutex

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

Usage of a semaphore

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
void up(struct semaphore *sem);
```

Completions

```
#include <linux/completion.h>
DECLARE_COMPLETION(name);
init_completion(struct completion *c);
INIT_COMPLETION(struct completion c); // Reinit only

void wait_for_completion(struct completion *c);

void complete(struct completion *c);
void complete_all(struct completion *c);

void complete_and_exit(struct completion *c, long retval);
```

Launching jobs and waiting for their completion asynchronously. It should sound familiar! The kernel is much asynchronous, quite every I/O operation happens asynchronously, from interrupt context to botton halves and other tools.

Reading from the user

```
static ssize_t dev_read(struct file * f, char * b, size_t s, loff_t * off)
{
   long r;
   if ( )down_interruptible( &sx ) ) return -ERESTARTSYS;
   ...
   r = copy_to_user(b, queue, s );
   ...
   up( &sx );
   return (ssize_t)r;
}
```

Important!

copy_to_user and *copy_from_user* are the only functions that are allowed to write/read from userspace memory! On modern x86-64 systems access is otherwise denied by **Supervisor mode access prevention**, on ancient ones the bug remains silent. **Note:** these functions may sleep!

Blocking I/O and wait queues

```
static DECLARE_WAIT_QUEUE_HEAD(rq);
static DECLARE_WAIT_QUEUE_HEAD(wq);

static ssize_t dev_read(struct file * f, char * b, size_t s, loff_t * off) {
   if(down_interruptible(&sx)) return -ERESTARTSYS;
   while ( /* nothing to read */ ) {
      up(&sx);
      if (f->f_flags & O_NONBLOCK) return -EAGAIN;
      if (wait_event_interruptible(wq, ( /*something to read*/ ) ) )
         return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
      /* otherwise loop, but first reacquire the lock */
      if (down_interruptible(&x))  return -ERESTARTSYS;
   }
...
   wake_up_interruptible(&wq); // Awake writers
```

wake up!

```
wake_up(wait_queue_head_t *queue);
wake_up_interruptible(wait_queue_head_t *queue);

// Up to nr exclusive waiters
wake_up_nr(wait_queue_head_t *queue, int nr);
wake_up_interruptible_nr(wait_queue_head_t *queue, int nr);

wake_up_interruptible_all(wait_queue_head_t *queue);
wake_up_interruptible_sync(wait_queue_head_t *queue);
```

wait!

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

Notes on DMA

- Memory coherency is not only a concurrency problem;
- memory is not used only by CPUs with virtual addresses,
- but also by devices with DMA.
- Devices always use physical addresses.
- Device drivers are supposed to go to the limits of memory coherency, for performance reasons!

Memory and overcommit

- Overcommit is the ability to feed a process, that is requesting more memory than it will use, just virtual addresses and to create the physical memory mapping late.
- This allows a better usage of system memory, as normally processes request much more than needed. Sometimes 5 times more, when a GC is involved.
- Normally, this makes malloc never fail!
- */proc/sys/vm/overcommit_memory* lets you control the systemś behaviour.
- Default value (0) overcommits but denies excesses;
- value (1) overcommits very much, useful for applications that use large sparse matrixes;
- value (2) no overcommit! The whole system cannot use more than physical memory.

Debug

- Linus Torvalds does not believe in debuggers!
- One of the main ways to debug your code is *printk*!
- *strace* can help you to see the raw data that are going inside and out your code;
- *perf* can help you to profile your internals;
- *gdb* can be used to debug a kernel, with some limitation.
- *kgdb* has arrived in 2.6.26 and allows to use gdb from remote to debug your kernel.

References

- Linux Kernel Development;
- Linux Device Drivers;
- Linux Networking Internals.