Lecture 19: Linux kernel modules

CS 3281

Daniel Balasubramanian,
Shervin Hajiamini, Sandeep Neema, and Bryan Ward

Overview

- The Linux operating system is considered the most successful open source project
  - Supports many architectures
  - Thousands of contributors
  - It's the OS used by Netflix to stream TV to everyone!
    - A good talk by a Netflix engineer: https://www.youtube.com/watch?v=FJW8nGV4jxY
- We're going to learn about it by:
  - Building a kernel
  - Adding functionality

History...

- 1969: Unix invented by Dennis Ritchie and Ken Thompson at Bell Labs
- Many derivatives of Unix since then, including:
  - FreeBSD, OpenBSD, macOS (Darwin)
- But Unix was expensive
  - Bell Labs estimated its IP in Unix at ~$250 million in the early 90s
- 1991: Linus Torvalds creates his own OS
  - Frustrated by Minix
  - Unix-like, but not Unix

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)

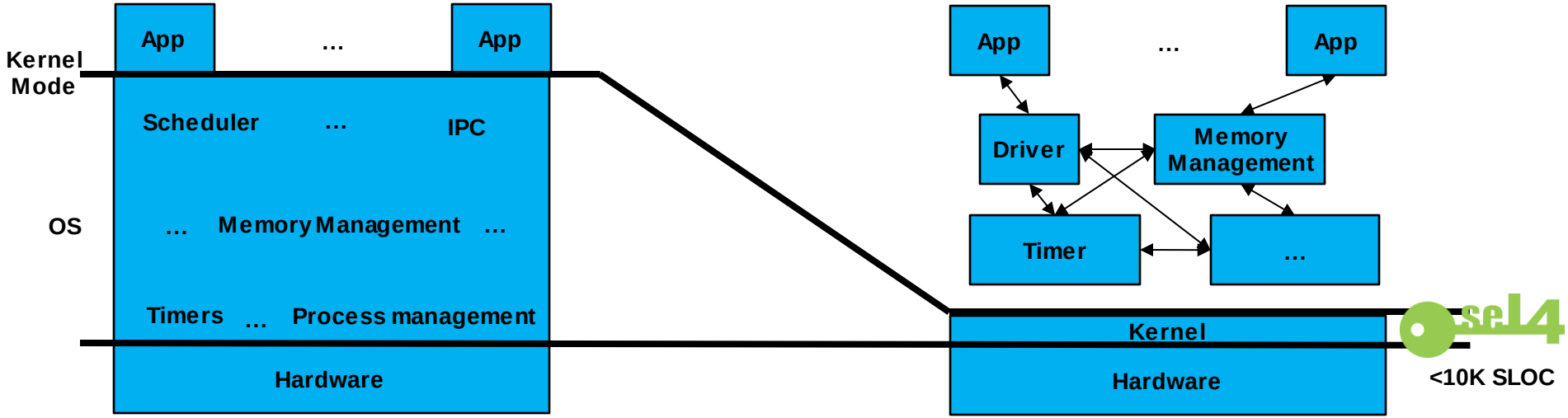
PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

— Linus Torvalds[15]

Monolithic vs microkernels

- Linux is a *monolithic* kernel
  - Translation: it's one big process running in one big address space
  - Good things: runs fast, simplifies implementation
  - Bad things: if one bad thing happens, the whole OS can crash
    - Including drivers: if a driver does something bad, the whole OS can crash
- This is in contrast to a *microkernel*
  - Functionality broken down into separate processes
  - These processes communicate with each other
  - Good things: a "cleaner" design, isolation between subsystems, some formally verified
  - Bad things:
    - Overhead with the communication between processes (though fast now)
    - Processes need to acquire many permissions to do relatively simple things,
    - Very difficult to program for – you must implement things you take for granted in Linux

Monolithic vs microkernels



Monolithic Operating System (E.g., Linux)

Microkernel

25M+ SLOC!

"A concept is tolerated inside the u-kernel only if moving it outside the kernel would prevent the implementation of the system's required functionality."

SLOC: Source Lines of Code

Overview of the source directories

- This table explains the directories:
- Question: where's the majority of the code?

Table 2.1 **Directories in the Root of the Kernel Source Tree**

Directory	Description
arch	Architecture-specific source
block	Block I/O layer
crypto	Crypto API
Documentation	Kernel source documentation
drivers	Device drivers
firmware	Device firmware needed to use certain drivers
fs	The VFS and the individual filesystems
include	Kernel headers
init	Kernel boot and initialization
ipc	Interprocess communication code
kernel	Core subsystems, such as the scheduler
lib	Helper routines
mm	Memory management subsystem and the VM
net	Networking subsystem
samples	Sample, demonstrative code
scripts	Scripts used to build the kernel
security	Linux Security Module
sound	Sound subsystem
usr	Early user-space code (called initramfs)
tools	Tools helpful for developing Linux
virt	Virtualization infrastructure

*Figure from *Linux Kernel Development* by Robert Love

Other notes about the kernel

- The kernel isn't a user-space application
  - Thus no libc! For example, no printf or malloc
- There are often kernel-specific equivalents
  - printk or kmalloc
- Linux is heavily tied to gcc
  - Many gcc specific extensions needed to build kernel
  - There are efforts to make it build with clang/llvm
- Fixed-size stacks
  - Architecture specific, but in the kernel, your stack is usually 8kB (32-bit) or 16kB (64-bit)
- No memory protection like userspace
  - Example: userspace processes get a SIGSEGV signal when dereferencing a null pointer
- Portability is important
  - https://en.wikipedia.org/wiki/List_of_Linux-supported_computer_architectures

A short look: processes

- Each process is represented by a `struct task_struct` defined in `<linux/sched.h>`

```
ender@HiveQueen:~/work/linux-5.3/include/linux$ ls -l sched.h
-rw-rw-r-- 1 ender ender 57169 Sep 15 16:19 sched.h
ender@HiveQueen:~/work/linux-5.3/include/linux$
```

  - Kernel keeps a doubly linked list of these called the task list
  - Relatively large: ~1.7kB

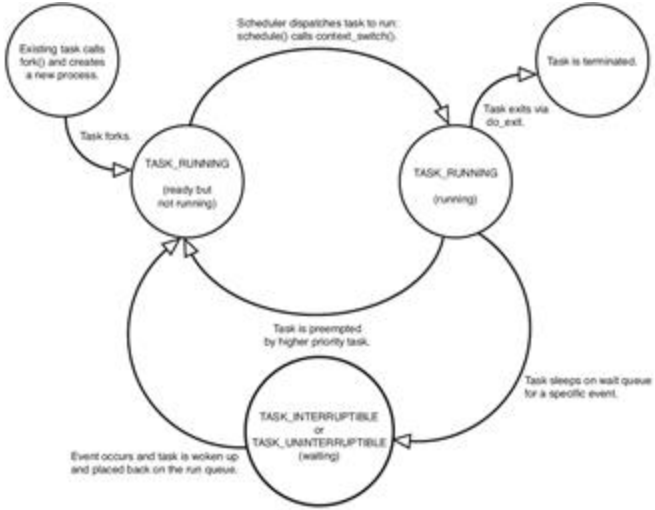


Figure 3.3 Flow chart of process states.

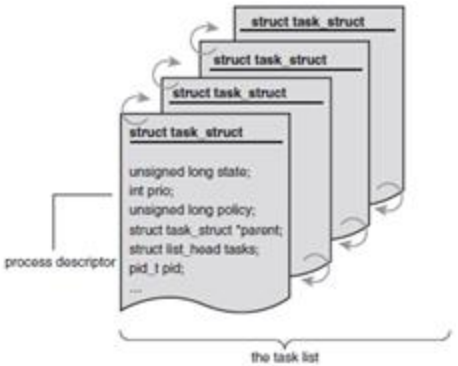


Figure 3.1 The process descriptor and task list.

*Figure from *Linux Kernel Development* by Robert Love

Modules

- Linux is monolithic
  - The whole kernel runs in a single address space
- But code can be dynamically inserted and removed at runtime with *modules*
  - Module: a loadable kernel object
- Allows the base kernel image to be relatively small
  - Insert modules for extra functionality on-demand
  - In-tree: modules (drivers) that come with the linux kernel source
  - Out-of-tree: modules that don't ship with the linux kernel source
  - Maintainer has to make sure it works with each kernel version
- `lsmod` and `modinfo`
  - Get info about loaded modules

```
ender@HiveQueen:~$ lsmod
Module                  Size  Used by
dcdbas                 16384  0
joydev                 20480  0
snd_intel8x0           36864  2
input_leds             16384  0
snd_ac97_codec        106496  1 snd_intel8x0
ac97_bus               16384  1 snd_ac97_codec
serio_raw              16384  0
intel_powerclamp       16384  0
snd_pcm                94208  2 snd_ac97_codec,snd_intel8x0
snd_seq_midi           16384  0
snd_seq_midi_event     16384  1 snd_seq_midi
snd_rawmidi            28672  1 snd_seq_midi
snd_seq                57344  2 snd_seq_midi_event,snd_seq_midi
snd_seq_device         16384  3 snd_seq,snd_rawmidi,snd_seq_midi
snd_timer              32768  2 snd_pcm,snd_seq
ipc_ich                20480  0
```

```
ender@HiveQueen:~$ modinfo dcdbas
filename:       /lib/modules/4.4.0-140-generic/kernel/drivers/firmware/dcdbas.ko
alias:          dmi:*:[bs]vnD[Ee][Ll][Ll]*:*
license:        GPL
author:         Dell Inc.
version:        5.6.0-3.2
description:    Dell Systems Management Base Driver (version 5.6.0-3.2)
srcversion:     2487624BA6DC5F530702CAD
depends:
retpoline:      Y
intree:         Y
vermagic:       4.4.0-140-generic SMP mod_unload modversions 686 retpoline
ender@HiveQueen:~$
```

Kernel Module

- A kernel module is a portion of kernel functionality that can be dynamically loaded into the operating system at run-time
- Example
  - USB drivers
  - File-system drivers
  - Disk drivers
  - Cryptographic libraries

Why not just compile everything into the kernel?

- Each machine only needs a certain number of drivers
  - For example, should not have to load every single motherboard driver
- Load only the modules you need
  - Smaller system footprint
- Dynamically load modules for new devices
  - Camera, new printer, etc.

Sample Kernel Module: hello.c

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
   printk(KERN_ALERT "Hello, world!\n");
   return 0;
}

static void hello_exit(void)
{
   printk(KERN_ALERT "Goodbye, sleepy world.\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Sample Kernel Module: hello.c

Module headers

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
   printk(KERN_ALERT "Hello, world!\n");
   return 0;
}

static void hello_exit(void)
{
   printk(KERN_ALERT "Goodbye, sleepy world.\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Sample Kernel Module: hello.c

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
   printk(KERN_ALERT "Hello, world!\n");
   return 0;
}

static void hello_exit(void)
{
   printk(KERN_ALERT "Goodbye, sleepy world.\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Initialization function, runs when module loaded

Tells kernel which function to run on load

Sample Kernel Module: hello.c

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
   printk(KERN_ALERT "Hello, world!\n");
   return 0;
}

static void hello_exit(void)
{
   printk(KERN_ALERT "Goodbye, sleepy world.\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Exit function, runs when module exits

Tells kernel which function to run on exit

Sample Kernel Module: Makefile

```
ifneq ($(KERNELRELEASE),)
   obj-m := hello.o

else
   KERNELDIR ?= \
   /lib/modules/`uname -r`/build/
   PWD := `pwd`
default:
   $(MAKE) -C $(KERNELDIR) \
      M=$(PWD) modules
endif

clean:
   rm -f *.ko *.o Module* *mod*
```

Compile the Kernel Module

```
/usr/src/hello$> make
```

- Creates hello.ko – This is the finished kernel module!

Inserting and Removing the Module

- insmod – insert a module

```
/usr/src/hello$> sudo insmod hello.ko
```

- rmmod – remove a module

```
/usr/src/hello$> sudo rmmod hello.ko
```

Listing Modules

- lsmod – lists all running modules

```
/usr/src/hello$>lsmod
```

Where is it printing?

- Look inside /var/log/syslog
- Hint – to watch syslog in real time, issue the following command in a second terminal:

```
$> sudo tail –f /var/log/syslog
```