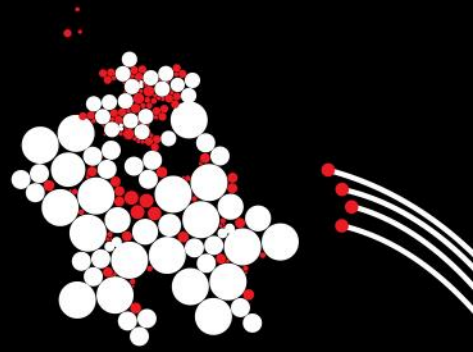
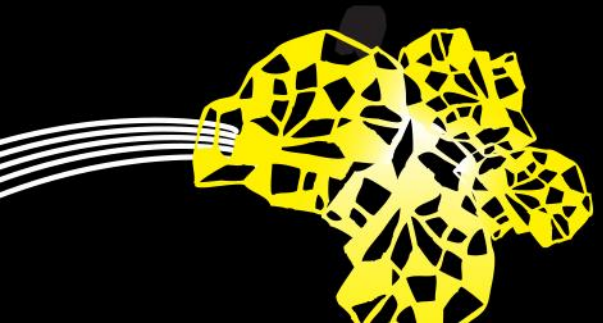
UNIVERSITY OF TWENTE.



OPERATING SYSTEMS

KERNEL

ERIK TEWS <e.tews@utwente.nl>



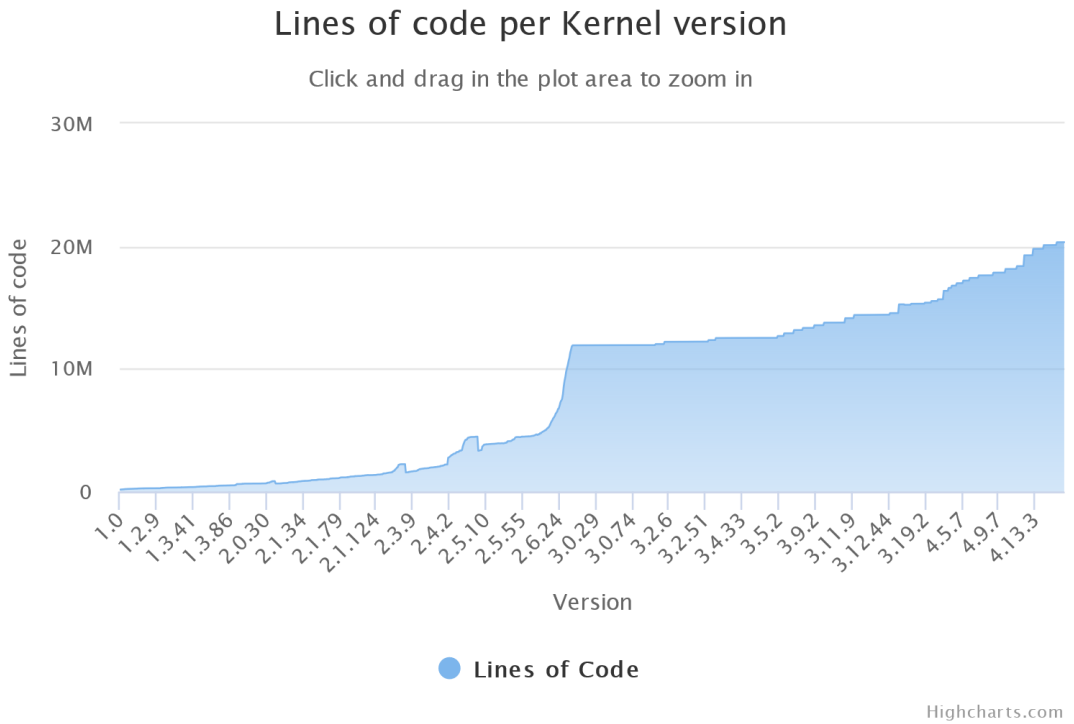
SO FAR

- We looked at programs running on Linux
  - How to request various resources from the kernel
  - How to instruct the kernel to optimize the execution of the program
- We looked at the kernel
  - Which algorithms are implement
  - How Linux performs certain tasks
- Now we are looking into the Kernel

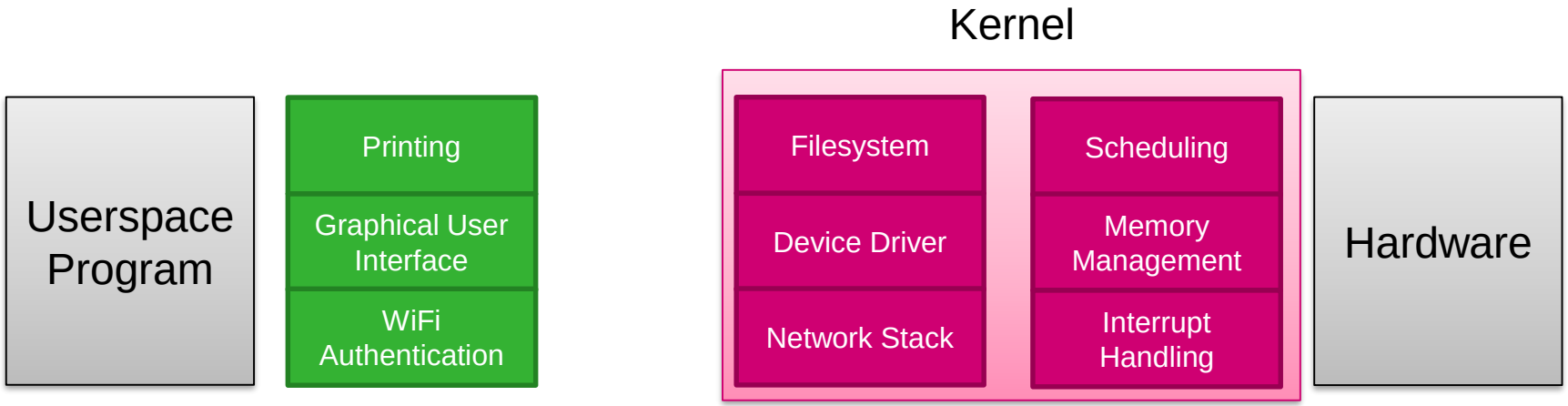
LINUX KERNEL OVERVIEW

- Written in C (and a bit of assembly)
- 20.000.000 lines of code (at the moment)
- Open Source
  - Even more, it’s GPL code
- Several architectures supported (~24 + sub architectures)
- Supports modules
- Monoolitic design

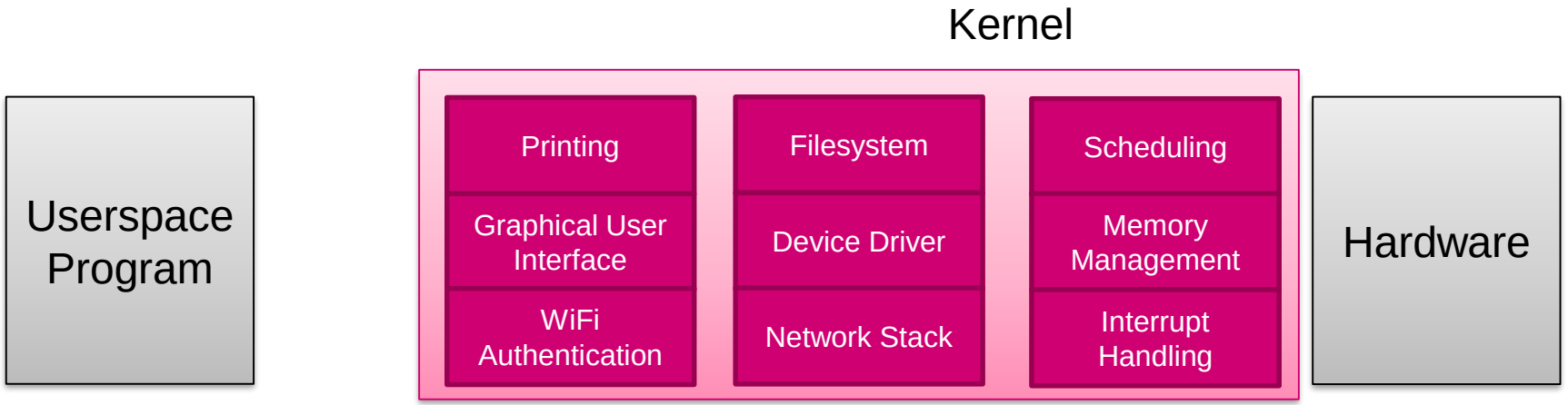
LINUX KERNEL OVERVIEW



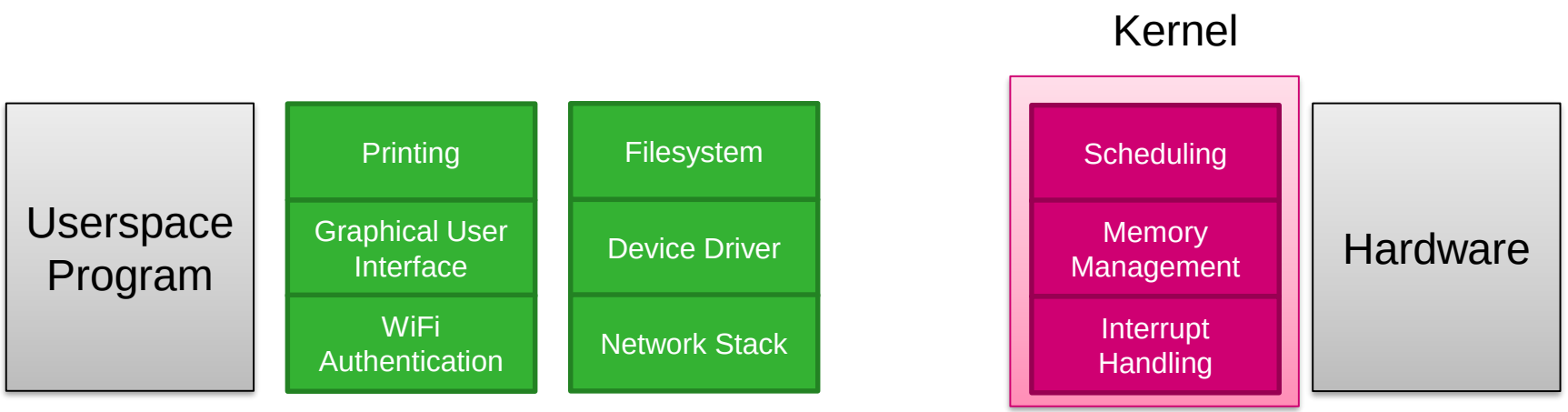
MONOTOLITIC VS MICROKERNEL



MONOTOLITIC VS MICROKERNEL



MONOTOLITIC VS MICROKERNEL



LINUX KERNEL DEVELOPMENT

- Mostly done by Linus
  - Supported by a few subsystem maintainers
  - Supported by many maintainers
  - Supported by even more contributors

LINUX KERNEL STRUCTURE

- Architecture specific parts
  - Arm, x86, s390
- Subsystem specific parts
  - Memory Management, Network, IPC, Sound…

LINUX KERNEL APIS

- External
  - Syscalls
  - Very stable
- Internal
  - Used by Linux modules
  - Very unstable

GETTING LINUX

- You can use git to clone the current Linux development tree
  - **git clone https://git.kernel.org/**
- But the RPI runs a modified kernel
  - **git clone https://github.com/raspberrypi/linux.git**
- Optionally, run with the **--depth 1** argument
- Git is a great tool for kernel development
  - Do you know why?
- And you may also need this:
  sudo apt-get install git bc bison flex libssl-dev

THE KERNEL ON YOUR DISK

pi@raspberrypi:/boot $ ls -la kernel*

-rwxr-xr-x 1 root root 4934264 Sep 27 15:23 kernel7.img

-rwxr-xr-x 1 root root 4683800 Sep 27 15:23 kernel.img

KERNEL MODULES

```
pi@raspberrypi:/boot $ ls -la /lib/modules/4.14.70-v7+/
total 1964
drwxr-xr-x  3 root root   4096 Sep 27 15:22 .
drwxr-xr-x  4 root root   4096 Sep 27 15:22 ..
drwxr-xr-x 11 root root   4096 Sep 27 15:22 kernel
-rw-r--r--  1 root root 505978 Sep 19 18:06 modules.alias
-rw-r--r--  1 root root 524355 Sep 19 18:06 modules.alias.bin
-rw-r--r--  1 root root  11137 Sep 19 18:06 modules.builtin
-rw-r--r--  1 root root  12165 Sep 19 18:06 modules.builtin.bin
-rw-r--r--  1 root root 152785 Sep 19 18:06 modules.dep
-rw-r--r--  1 root root 219986 Sep 19 18:06 modules.dep.bin
-rw-r--r--  1 root root    302 Sep 19 18:06 modules.devname
-rw-r--r--  1 root root  58458 Sep 19 18:06 modules.order
-rw-r--r--  1 root root    352 Sep 19 18:06 modules.softdep
-rw-r--r--  1 root root 217667 Sep 19 18:06 modules.symbols
-rw-r--r--  1 root root 267319 Sep 19 18:06 modules.symbols.bin
```

DEVICE TREE (ARM SPECIFIC)

- Allows you to compile one kernel that supports many different platforms
- Their difference is configured with the device tree files
- Found in /boot/*.dtb and /boot/overlays/

COMPILING THE LINUX KERNEL

- The kernel is (of course) always compiled for a specific architecture
  - Not simply a make
  - Many things can be configured
- Many configuration options are:
  - Y = Include that feature in the kernel
  - N = Do not include that feature in the kernel
  - M = Compile that feature as a module
- Controlled with a **.config** file

NOW COMPILE IT

```
KERNEL=kernel7l
make -j4 zImage modules dtbs

sudo make modules_install
sudo cp arch/arm/boot/dts/*.dtb /boot/
sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
sudo cp arch/arm/boot/zImage /boot/$KERNEL.img
```

CONFIGURING THE LINUX KERNEL

- Place a .config file in the current Linux kernel directory
  - **make menuconfig**
- You can query the currently running kernel for it‘s configuration
  - **modprobe configs**
  - **zcat /proc/config.gz > .config**
- Alternatively: **make bcm2711_defconfig**
- Configuration options are changed with new Linux versions
  - They often come with a useful default

EXTENDING THE LINUX KERNEL

- Patch the source code directly
  - Is sometimes required when fundamental new features are added
  - Or when existing features are modified
- Just implement a module
  - Often sufficient when a new device needs to be supported
  - Can be loaded during runtime
  - When you wanna do this on your Pi, run
    - sudo apt-get install raspberrypi-kernel-headers

PROGRAMMING IN THE LINUX KERNEL

- In theory just normal C code
- Except that there is no standard C library available
  - No printf, malloc, free, fopen, sleep…
- And there are a few things you are not allowed to do
  - Such as using floating point registers, we come to that later
- And no main function
  - Instead you use **module_init(your_start_function);** for your code

MODULE EXAMPLE (BLOG.SOURCERER.IO)

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Me");
MODULE_DESCRIPTION("A simple example Linux module.");

static int __init lkm_example_init(void) {
 printk(KERN_INFO, "Hello, World!\n");
 return 0;
}

static void __exit lkm_example_exit(void) {
 printk(KERN_INFO, "Goodbye, World!\n");
}

module_init(lkm_example_init);
module_exit(lkm_example_exit);
```

A FEW MORE THINGS ABOUT KERNEL CODE

- At any time each of the CPUs in a system can be:
  - not associated with any process, serving a hardware interrupt;
  - not associated with any process, serving a softirq or tasklet;
  - running in kernel space, associated with a process (user context);
  - running a process in user space.
- You can check that (in_task() for example)

https://www.kernel.org/doc/html/v4.19/kernel-hacking/

NO MEMORY PROTECTION

- If you corrupt memory, whether in user context or interrupt context, the whole machine will crash. Are you sure you can’t do what you want in userspace?

NO FLOATING POINT OR MMX

- The FPU context is not saved; even in user context the FPU state probably won't correspond with the current process: you would mess with some user process' FPU state. If you really want to do this, you would have to explicitly save/restore the full FPU state (and avoid context switches). It is generally a bad idea; use fixed point arithmetic first.

A RIGID STACK LIMIT

- Depending on configuration options the kernel stack is about 3K to 6K for most 32-bit architectures: it’s about 14K on most 64-bit archs, and often shared with interrupts so you can’t use it all. Avoid deep recursion and huge local arrays on the stack (allocate them dynamically instead).

THE LINUX KERNEL IS PORTABLE

- Let’s keep it that way. Your code should be 64-bit clean, and endian-independent. You should also minimize CPU specific stuff, e.g. inline assembly should be cleanly encapsulated and minimized to ease porting. Generally it should be restricted to the architecture-dependent part of the kernel tree.

ENOUGH ABOUT PROBLEMS

- There are many replacements for the common routines from the standard c library
- However they might have different semantics
- And you cannot use them in every context (user context is the easiest one)

PRINTING MESSAGES

printk() feeds kernel messages to the console, dmesg, and the syslog daemon. It is useful for debugging and reporting errors, and can be used inside interrupt context, but use with caution: a machine which has its console flooded with printk messages is unusable. It uses a format string mostly compatible with ANSI C printf, and C string concatenation to give it a first “priority” argument:

GET THE PROCESS FOR THE USER CONTEXT

current()

Defined in include/asm/current.h

This global variable (really a macro) contains a pointer to the current task structure, so is only valid in user context. For example, when a process makes a system call, this will point to the task structure of the calling process. It is not NULL in interrupt context.

To get the current program file, try: get_task_exe_file()

ATOMIC OPERATIONS

int atomic_inc_return(atomic_t *v);

int atomic_dec_return(atomic_t *v);

These routines add 1 and subtract 1, respectively, from the given atomic_t and return the new counter value after the operation is performed.

MEMORY ALLOCATION

void * kmalloc(size_t size, gfp_t flags) allocate memory

The flags argument may be one of:

- GFP_USER - Allocate memory on behalf of user. May sleep.
- GFP_KERNEL - Allocate normal kernel ram. May sleep.
- GFP_ATOMIC - Allocation will not sleep. May use emergency pools.
  For example, use this inside interrupt handlers.
- GFP_HIGHUSER - Allocate pages from high memory.
- GFP_NOIO - Do not do any I/O at all while trying to get memory.
- GFP_NOFS - Do not make any fs calls while trying to get memory.
- GFP_NOWAIT - Allocation will not sleep.
- __GFP_THISNODE - Allocate node-local memory only.
- ……

MODULE USAGE COUNT

try_module_get()/module_put()

Defined in include/linux/module.h

These manipulate the module usage count, to protect against removal (a module also can’t be removed if another module uses one of its exported symbols: see below). Before calling into module code, you should call try_module_get() on that module: if it fails, then the module is being removed and you should act as if it wasn’t there. Otherwise, you can safely enter the module, and call module_put() when you’re finished.

THE /PROC FILESYSTEM

proc_create() – To create an entry in the proc filesystem

single_open() / seq_printf() / seq_read() / seq_lseek() / single_release()
Useful utilities to export information to users

seq_path() – When you have to deal with files

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

MODULE_DESCRIPTION("My kernel module");
MODULE_AUTHOR("Me");
MODULE_LICENSE("GPL");

static struct proc_dir_entry* my_proc_file;
static int counter = 0;

#define procfs_name "helloworld"

static int
my_show(struct seq_file *m, void *v)
{
    seq_printf(m, "%s %d\n", "Hello World!", counter++);
    return 0;
}

static int
my_open(struct inode *inode, struct file *file)
{
    return single_open(file, my_show, NULL);
}
```

```c
static const struct file_operations my_fops = {
    .owner      = THIS_MODULE,
    .open       = my_open,
    .read       = seq_read,
    .llseek     = seq_lseek,
    .release    = single_release,
};

static int __init my_init(void)
{
  my_proc_file = proc_create(procfs_name, 0, NULL, &my_fops);

  if (!my_proc_file) {
      return -ENOMEM;
  }

  return 0;
}
static void __exit my_exit(void)
{
        remove_proc_entry(procfs_name, NULL);
        printk(KERN_INFO "/proc/%s removed\n", procfs_name);
}

module_init(my_init);
module_exit(my_exit);
```

ONE MORE THING TO BE AWARE OF

- The kernel API changes constantly
- When you find a tutorial, expect it to be outdated
- Except of course stuff from the kernel documentation itself
- The kernel itself is a nice template for source code

https://www.kernel.org/doc/html/v4.19/

https://www.linux-mm.org/