Distributed Systems Group
Department Software Technology
Faculty EEMCS
DELFT UNIVERSITY OF TECHNOLOGY

Operating Systems (TI2726-C)
Lab Session 4: Kernel

D.E. Schut, M. van de Ruit, O. W. Visser & J.S. Rellermeyer

March 14, 2019

Deadline: March 21 2019

Introduction

Session 4 of this lab will explore kernel modules, character device drivers, as well as the basics of Pulse Width Modulation (PWM). In order, the purposes of this assignment are to:

- Become familiar with writing small kernel modules for linux.
- Learn how the pi's gpio can be used in kernel space.
- Learn how character devices can be used to communicate with a kernel module.
- Learn how high-resolution timers can be used in kernel space.
- Get insight into how kernel modules are executed, and how to make this execution thread safe.
- Learn how PWM works and how it can be applied to dim LEDs.

As you may encounter a number of new concepts during these assignments, please make sure you read the explanatory texts before starting on them. Also note that it is possible for you to damage the kernel during this assignment. Do not panic! **If you damage the kernel or memory in any way during this assignment and your Raspberry Pi stops functioning correctly, just pull the power plug and reboot it!**

Loadable kernel modules

A (loadable) kernel module is a piece of code that can be loaded into and unloaded from the kernel on demand. Kernel modules extend the functionality of the kernel without requiring a reboot of the computer. They are executed in kernel space—as opposed to regular programs which are executed in user space—which provides them with a set of restrictions and privileges that you should be aware of before getting started.

The advantage of loadable kernel modules over modifying the kernel directly is vast. For one, you do not have to recompile the kernel after every modification. For another, as the kernel is built up of many small components that can be enabled and disabled individually, this greatly alleviates issues with debugging and performance.

Programming in kernel space requires you to accept some hard truths, such as:

- Functions such as `sleep()` and `printf()` are not available. However, `printk()` is available. This prints to the system logs. To display the system log, use the `dmesg` command. To shorten the output of `dmesg`, you can use a pipe with `tail`, `grep` or `less`.
- Many normal C-libraries are not available. Special kernel versions are available for some often used functions.
- You always have root access and your code will run in kernel mode on the CPU, which allows you to make highly destructive changes.
- Kernel modules typically are run at a higher priority than user space code. This might seem nice, but since kernel modules always run in the background you should keep in mind you don't want them to use too much CPU time, and you should never stall the entire system.
- Using floating point (FP) arithmetic is possible in theory, but it has some difficulties. As Linus Torvalds says: "In other words: the rule is that you really shouldn't use FP in the kernel". [1]
- Dynamically allocated memory is **not** automatically deallocated when a kernel module is unloaded. This may cause memory leaks that can only be solved by rebooting the Pi.
- Kernel modules do not run sequentially. Functions are called when specific events occur. Because of this you should keep in mind that your code can be interrupted, so you should make your code thread safe.
- You cannot execute kernel modules as if they were a program. You can load them into the kernel using the command `sudo insmod <module name>`, and remove them again using the command `sudo rmmod <module name>`. You can view all currently loaded kernel modules with the command `lsmod`.
- Compiling kernel modules requires a special Makefile. This Makefile will be provided on Brightspace!

For more information, we strongly recommend you to read the Linux Kernel Module Programming Guide. [2]. The first few tutorials are bound to help you get started with kernel programming in no time at all!

[1] For Linus' complete view on FP in the kernel: `http://yarchive.net/comp/linux/kernel_fp.html`
[2] Linux Kernel Module Programming Guide: `http://www.tldp.org/LDP/lkmpg/2.6/html/`

Kernel module information

To give some information about your kernel module several macros are available. To use some libraries it is needed to define your module license to be GPL. How to describe your module is shown below:

```
MODULE_LICENSE("GPL");  //Needed for using libraries such as gpio.h
MODULE_AUTHOR("John Doe");
MODULE_DESCRIPTION("Does nothing really");
MODULE_VERSION("0.1");
```

To view all currently loaded kernel modules you can use the `lsmod` command in the terminal. To view the information of one of the loaded kernel modules you can use the command `modinfo <module name>`.

Device files

To communicate with a kernel module from user space, device files can be used. Linux uses a synthetic file system [3] in some places, which means that not every file actually has to correspond to a data storage file, as you would normally use files. The files in the `/dev` folder of a Linux installation can be used to communicate with kernel modules that typically interface to hardware. There are two types of device files: *character devices* either directly handle a request or block it, and *block devices* buffer their requests, so requests are never blocked but they might be executed later and in a different order. In this lab session we will only consider character devices.

When you do something with a device file, a kernel module will respond. To define how the kernel module should respond to each type of file operation, you can define a callback function for each operation. For example, you could register a callback function to the file open operation that would print ”File is opened” to the kernel log every time the file is opened by an application.

To distinguish which kernel module is associated with which file, each kernel module is assigned a major number. When a device file is given the same major number as a kernel module, the file is controlled by that kernel module. Kernel modules can also control multiple device files, so each device file is also assigned a minor number.

You can register a character device with the following function:

```
int register_chrdev(unsigned int <major number>,
        const char * <device name>,
        const struct file_operations* <file operations callbacks>)
```

To make sure your kernel module uses a major number that is not already in use by some other kernel module, you can request an unused major number by passing 0 as the first argument. The function will return the chosen major number. You can pick a device name yourself. The file operation struct contains callbacks to all the file operations you want to support. When you do not specify a callback function for a file operation, the function pointer will keep its default value NULL, which means nothing happens on that operation. The complete specification of the file operation struct can be found on the documentation of the Linux virtual file system [4] under ”The File Object”.
The next step is to create a device class, with the following function:

[3] Synthetic file system: `https://en.wikipedia.org/wiki/Synthetic_file_system`
[4] VFS documentation: `https://www.kernel.org/doc/Documentation/filesystems/vfs.txt`

```
struct class* class_create (struct module* <owner>,
                            const char* <class name>);
```

You can pass THIS_MODULE as the first argument and you can pick a name yourself as the second argument.
The final step is to create the device itself with the following function call:

```
struct device* device_create (<class pointer>, NULL,
        MKDEV(<major number>, 0), NULL, <device name>);
```

When the kernel module is removed, you should also clean up your character device:

```
// remove the device
device_destroy(<class pointer>, MKDEV(majorNumber, 0));
// unregister the device class
class_unregister(<class pointer>);
// remove the device class
class_destroy(<class pointer>);
// unregister the major number
unregister_chrdev(majorNumber, <device name>);
```

Timers in kernel space

Software timers are a concept in Linux that allow you to schedule the execution of a function in your code at a later point in time. Timers can be used, for example, to regularly poll the status of a device that does not generate interrupts. Normally `linux/timers` more than suffices. This uses *jiffies*[5] (yes, we know) as a unit of time. On the Raspberry Pi, jiffies span somewhere slightly above 10ms. For the coming assignments, sadly, this is a bit too large!

Thankfully, `linux/hrtimer.h` exists, which provides high-resolution timers accurate down to nanoseconds. HrTimers works with a unit called KTime, which is located in `linux/ktime`. HrTimers are represented by a structure `hrtimer`. This structure has a number of properties, such as a callback function and an expiration time. Declaring one is simple:

```
static struct hrtimer hr_timer;
```

Further, a timer must be initialized and started. When no longer needed (on cleanup of the kernel module, for example), it must be canceled. A number of functions exists for this:

```
// Initialize a hrtimer
void hrtimer_init(struct hrtimer *timer,
                  clockid_t which_clock,
                  enum hrtimer_mode mode);

// Start a hrtimer
int hrtimer_start(struct hrtimer *timer,
                  ktime_t tim,
                  const enum hrtimer_mode mode);
```

[5] About Jiffies: `http://www.makelinux.net/books/lkd2/ch10lev1sec3`

```
// Cancel a hrtimer
int hrtimer_cancel(struct hrtimer *timer);
```

Finally, the most important part of a timer is its callback function! Code in this function is executed when the timer expires. The function returns whether it should be rescheduled for another interval.

```
enum hrtimer_restart hrTimerCallback(struct hrtimer *timer_for_restart) {
    ktime_t currtime, interval;
    currtime = ktime_get();

    // Set an interval of 1 millisecond, which is equal to 1*10^6 nanosecond
    interval = ktime_set(0, 1e6);

    // Move restart timer to a time in the future
    hrtimer_forward(timer_for_restart, currtime, interval);

    // restart, or return HRTIMER_NORESTART;
    return HRTIMER_RESTART;
}
```

For more information, which you'll need, we recommend starting in the hrtimers source. [6]

Mutexes in kernel space

Mutexes can be used in kernel space by including `linux/mutex.h`. They can be used as follows:

```
struct mutex myMutex; // defines a mutex

// Initializes the mutex, only call it once
mutex_init(&myMutex);

// Lock a mutex. If it is already locked, wait for it to be unlocked
mutex_lock(&myMutex);

//Lock a mutex. If it is already locked, return false, else return true
mutex_trylock(&myMutex);

// Unlock a mutex
mutex_unlock(&myMutex);
```

Pulse width modulation (PWM)

As you have already seen in Lab Session 2, PWM can be used to dim LEDs. Here we will explain what PWM is so you can implement it yourself! PWM is a technique to generate a signal that can be varied within a certain range by rapidly switching between an on and an off state. For example on the Raspberry Pi the GPIO pins can only be turned completely on or completely off. By switching very quickly between on and off you can generate values that appear to be in between on and off.

[6] linux/hrtimers.h: `http://lxr.free-electrons.com/source/include/linux/hrtimer.h`

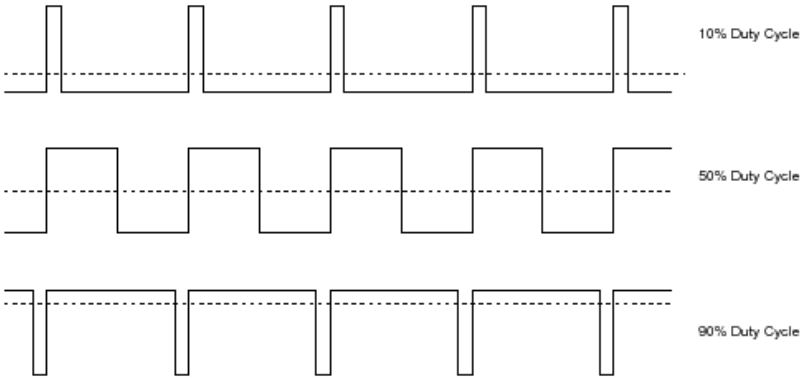

Figure 1: PWM waveforms for different duty cycles

In PWM a periodic signal is generated with a period that remains constant. In this lab session a period of 10ms will be used, so the PWM frequency will be 100Hz. This period is split into two parts: a part where the output is high and a part where the signal is low. The duty cycle defines what percentage of the PWM period the output will be high. This is illustrated in Figure 1. For example, when the period is 2 ms and the duty cycle is 75%, the LED will be on for 1.5ms and then off for 0.5ms, which will be repeated as long as the LED brightness should remain the same.

Some devices have the capability for hardware PWM. This way the switching is performed by hardware, so no CPU time is used for switching the outputs, but in this lab session you will implement PWM in software.

Assignments

For the assignments all code has to be written in C, and has to able to run on the Raspberry Pi. Only the code has to be submitted to CPM. Please use a separate .c file for each assignment and name it with the convention X_Y.c, where X is the number of the lab session and Y is the number of the assignment (use X_b.c for the bonus assignment). You can use the same makefile as last week. For the kernel module a special makefile is provided.

Please read the assignments carefully and make sure your code exactly implements the given descriptions. We will assess both the functionality and the clarity of your code and comments. Answers to questions are allowed in both Dutch and English.

4.1: Your first kernel module

In this assignment, you will create a tiny kernel module that controls the GPIO pins. For a thorough explanation on kernel module programming, please first read **Loadable kernel modules**, and **definitely** take a look at the provided source material.

In order for you to control the GPIO pins in kernel space, we recommend you to use the `gpio.h` header file, which can be included by adding `#include <linux/gpio.h>`. You should figure out the exact function calls to gpio.h yourself, but they are quite similar to wiringPi. Make sure to request the GPIO pins before using them and to free them when the kernel module is cleaned up.

Furthermore, if you use gpio.h, please add the following line to your code, otherwise your module will not load:

```
MODULE_LICENSE("GPL");
```

Your assignment is to:

1. Create a kernel module.
2. On initialization, have the kernel module request the GPIO pins with the LEDs, print something to the kernel logs, and turn on all the LEDs.
3. On removal, have the kernel module turn off all the LEDs, free the used GPIO pins, and print something to the kernel logs.

4.2 A simple character device driver

In this assignment, you will create a small character device driver. This character device driver registers itself as `/dev/klog` (if this is taken, choose another name!), accepts string input, and prints this input to the kernel log in a readable format. Use `printk()` to print to the kernel log.

Please read **Device files** before starting on this assignment.

Your assignment is to:

1. Create a character device driver that registers as `/dev/klog`.
2. Let strings written to this character device driver be printed to the kernel log.

3. Ensure the character device driver de-registers and unloads correctly.

4. Make sure that the `/dev/klog` file can only be opened by one application at the same time. When the file already is opened and another program tries to open it, `fopen()` should return an error code in that program. Make sure that this mechanism is thread safe.

You can test your device driver from the terminal using the following command:

```
sudo sh -c 'printf "Your message" > /dev/klog'
```

4.3: An introduction to timers

In this assignment, you will create a small kernel module that uses hrtimers to turn on and off the LEDs on your Pi in a certain pattern. Remember that you can use `linux/gpio.h` to control GPIO from kernel modules.

Please read **Timers** before starting on this assignment.

Your assignment is to:

1. Create a kernel module.

2. Use a hrtimer to turn the LEDs on for one second and then off for two seconds. Keep repeating this as long as the module is loaded.

4.4: Dimming a LED using PWM

In this assignment, you will create a character device driver that registers under `/dev/pwm`. When you write an integer between 0 and 100 to the file, it uses PWM with a duty cycle between 0% and 100% to dim all LEDs as a result. Use timers to implement your PWM.

Please read **Pulse Width Modulation** and **Mutexes in kernel space** before starting on this assignment.

Your assignment is to:

1. Create a character device driver that registers under `/dev/pwm`.

2. Let integers between 0-100 written to this character device change the duty cycle between 0% and 100%.

3. Every change in duty cycle should be printed to the kernel log.

4. Use PWM with the given duty cycle to set all LEDs to an intensity between 0% and 100%.

5. The period of one PWM cycle should be 10ms.

6. The timer callback function should not be called more than twice every PWM period.

7. Ensure the character device driver de-registers and unloads correctly.

Bonus assignment: prime number device

Linux allows developers to create device files that have no correspondence to hardware. Some of these devices are listed below:

/dev/null accepts and discards all input; produces no output (always returns an end-of-file indication on a read)

/dev/zero accepts and discards all input; produces a continuous stream of NULL (zero value) bytes

/dev/full produces a continuous stream of NULL (zero value) bytes when read, and returns a "disk full" message when written to

/dev/random and /dev/urandom produce a variable-length stream of pseudo-random numbers.

Interesting things happen when you try `cat /dev/urandom` or `sudo dd if=/dev/urandom of=/dev/mem` [7]. **Only try them at your own risk. Only try them on the Raspberry Pi. Don't try them on your personal computer.**

In this assignment you will create a new device `/dev/prime` that generates prime numbers. Starting at value zero, every read to the device file should return the next prime number followed by a newline character (\n).

Moreover, when you write a number to the device file it should start returning prime numbers from that value onwards. For example, when you write 8 to `dev/prime` the next read should return 11, after that the next read should return 13. The file may only be opened by one application at the same time, and the mechanism should be thread safe.

[7] `https://www.youtube.com/watch?v=GIu2vaIet6U`