ECE 373 – EMBEDDED OPERATING SYSTEMS AND DEVICE DRIVERS
Spring 2022

Midterm 5/4/2022

Name: ________James Ross____________

1. Define what an Embedded Operating System is. Describe some things that make it different from other types of OS's in the wild. (3)

An embedded operating system is an OS that has a small footprint on the device and controls specified HW in a system to do specific tasks. This is different than single user and multi user systems because those systems do many things and often support all kinds of hardware that could be introduced in the system whereas embedded systems are designed to be small and do specific tasks. This is also different from real time operating systems in a way since real time operating systems are not specific to embedded but an embedded operating system can also be designed to be real time.

2. If your code uses `printk(KERN_INFO "Huzzah, Hurrah!\n")`, list all of the places you can see the output. (2)

you can see the output in dmesg, journalctl and in sys log.

3. When thinking about memory-mapped I/O (MMIO): (6)
    a. Explain what a BAR is.

The BAR is the base address register. In memory-mapped I/O this is where the section of memory that is reserved to communicate with the hardware being set up. The BAR is the initial starting point of the memory-mapped section of memory to change registers on the device.

b. Explain the role that a BAR has with MMIO and a device.

The BAR holds the address of the start of the section for a devices memory that maps to the registers on the device. It is used in conjunction with the offset of a particular register to get to the correct memory locations that alter the registers in the device being used.

c. Explain the process to map a BAR for access.

In order to map the BAR for access you first need to find call pci_request_selected_regions() to get the access to the devices BARs in memory. Then you need to call pci_resource_start() and select which BAR of the device you want to use which gives you the MMIO starting position. After that we call pci_resource_len() to obtain the length of the memory resource for a devices given BAR.

Once that is complete we use ioremap with the start and length of the memory to obtain the BAR address in the memory to access the registers in the device.

4. Kernel modules and drivers require a certain access level to be loaded and unloaded in the OS. Explain why this is. And what command is useful to get this access level from the command line? (2)

Root access is required for loading and unloading kernel modules so the users who are not root cant obtain specific kernel information and possibly load modules that can break the system or unload modules that are required for the system to run properly. To get the access level you can use ls -la and see the permissions or use the stat command to see if it is readable, writable or executable.

5. Time is an important piece of any OS. In Linux, what is the wall-time-independent measure of time? How does one convert from wall time to this unit of time? (2)

The wall-time-independent measure of time is in jiffies which is a counter from the start of the unix operating system. We use HZ which is a measurement of jiffies per second to convert jiffies into wall time. For example, for 2 seconds wall time we would use jiffies+(2*HZ) which gives us the current time in jiffies plus 2 times jiffies per second. HZ is defined on the kernel build time and can be altered in the config file for the build.

6. Speaking of time… (6)

    a. Explain what the difference between `msleep(200)` and `mdelay(200)` is when using it inside your kernel module, and explain why it is so important.

msleep() will give up your timeslice in the kernel and it will go do other things with that time and come back after 200ms in this example. This is not a guarantee of 200ms exactly and can be slightly longer than that depending on when your module gets its timeslice back. mdelay() on the other hand will put itself in a loop and continue to execute something like a

NOP until 200ms is reached in this example. This will be more exact than msleep() but it causes the kernel to do nothing else but delay for your module so other work is not being done in the meantime.

This is important for things like interrupts and states that should not be giving its timeslice away. Leaving an interrupt and going back in can cause issues.

b. If you wanted to compute a delta between `jiffies` now and `jiffies` in the future, what's a good way to do that?

First we can record what jiffies is currently, then delay, sleep, or execute functions like time_before() while we wait for time to go by. Then we can obtain jiffies again and find the difference between them.

7. Let's discuss memory in Linux: (10)

a. What functions can we use to allocate memory in the kernel?

kmalloc, kzmalloc, kcalloc, vmalloc

b. Describe the various memory types Linux provides, the differences between them, and ways some can be allocated for use.

<u>user space</u>

In userspace we only see virtual memory. Virtual memory is memory that maps to a physical address. Most likely you will not have a virtual memory map to contiguous physical memory but instead it will map to different chunks in the physical memory space. Virtual memory does appear to be contiguous to the person who allocated it but behind the scenes it is not. We use malloc() in userspace to get our allocated memory from the OS which does the mapping. Virtual memory also does not have to fit into the physical memory space and can utilize things like a swap drive or file.

Kernel space

In the kernel there are a few memory types.

Kernel virtual memory is much like user space memory except it is allocated for the kernel to use. The same rules apply as the user space virtual memory as far as its contiguous nature behind the scenes and up front. This memory cannot be pinned like the other kernel memory types. we use vmalloc() to allocate this memory.

Kernel logical memory is mapped one to one with the physical memory. It is guaranteed to be contiguous and if it is not able to do that it will return from kmalloc() with a negative errno. This is the memory that all the kernel uses for its operations. This memory can be pinned and used for things like DMA.

Bus addresses are like the kernel logical except it is in its own domain. For now we see them as essentially the same in this class.

8. Consider this file: (6)

```
crw--w----  1 pjw  tty  136, 1 May  4 22:56 foobar
```

a. What type of file is this? How would you create this file in `/dev`?

It is a serial port, character driver. You would create this file in /dev utilizing the mknod command on the command line.

b. What are the major and minor numbers for this file, and what are their significance?

The major number is 136 and the minor number is 1. The major number lets the kernel know which module it is operating on and in general should be allocated dynamically. The minor number is how many devices can be connected to the driver and in this case it is 1, therefore it is safe to say it is the second device being driven by this driver.

9. Describe the differences between `register_chardev_region()`, and `alloc_chardev_region()`? Which one is typically preferable to use versus the other in most circumstances, and why? (5)

regiser_chardev_region() takes arguments that sets the major and minor numbers. This is not done dynamically and should be avoided other than small embedded systems where you control and know all the devices that will be placed in the system.

alloc_chardev_region() dynamically allocates the numbers and lets the kernel decide what the major number is going to be based on what's already on the system. This is useful in most situations where you don't know exactly which devices and their other major numbers are going to be. This will prevent collisions on declaring your major and minor numbers.

10. Consider the `/sys` and `/proc` filesystems. (9)

a. What do we call these filesystems?

These are pseudo file systems.

b. How do these filesystems differ from regular filesystems?

These files are not actually on disc but in kernel memory space and just present like files to gain access to functionalities like being able to cat and echo to the device.

c. How are the files in these filesystems useful?

These filesystems are useful because they can provice information about what is going on in the kernel. In /sys we can find all sorts of information such as the log file instead of using dmesg. It also holds module information in /sys/devices and you can see all the modules that are loaded and alter the parameters. in /proc we can find more information about the kernel as well. It gives the process ID's of the running OS and also information about interrupts, vmalloc information, DMA information all in its pseudo file system.

11. When we encounter a kernel panic (crash), a stack trace is generated. What command is useful to analyze this crash? (3)

objdump to find the piece of code around where the crash occurred based on the offsets found in the stack trace.

12. Explain what `BUG_ON(`*`expr`*`)` and `WARN_ON(`*`expr`*`)` do. Also explain what the major differences are between the two of these. (2)

BUG_ON() will crash the system and provide a stack trace and information about the register states of the CPUs if the expression is true. WARN_ON() does the same thing except it will not crash the system just produce the information in the sys log or dmesg.

13. Describe the relationships between `struct file_operations`, `struct cdev`, and a chardev referred to as a `dev_t` type, and why they're relevant to our driver. Also outline the sequence to put them all together. (8)

struct file_operations is a struct that holds function pointers to our defined functions. This is used for system calls such as read and write in userspace to guide the OS to the correct read and write function for the given driver. It is like a road map to the correct code to execute based on the system call from the userspace program.

struct cdev represents our character device. It holds device information and adds it to the kernel so the kernel knows what type of device we are using and has parameters for the kernel representing our device.

The dev_t type holds the packed major and minor numbers for our device. This is used to identify the device in the kernel. This is passed to alloc_chrdev_region() which uses the major and minor numbers and sets the dev_t node.

To put them all together you must set up your struct file_operations with the module name and the function pointers to the desired file system calls. Then you will have to call alloc_chrdev_region() to set the major and minor numbers. Once that is complete you call cdev_init() to initialize the cdev with the file_operations struct being passed to init. Once that is complete we call cdev_add() to add the cdev device to the kernel and set its major and minor number using the dev_t type and the device count.

14. Module parameters allow us to pass data into our driver when it is loaded into the kernel. Consider the following: (6)

    a. Write the code you would need to include a module parameter named "`exam_time`" that is an integer, and no flags.

static int exam_time = 0;

module_param(exam_time, int, 0);

    b. If you wanted to expose "`exam_time`" in `/sys` to allow read access, rewrite the code from "a" to allow this.

static int exam_time = 0;

module_param(exam_time, int, S_IRUSR);

15. Consider the following code: (10)

```
struct my_dev {
      int foo;
      struct timer_list my_timer;
} bar_dev;

static void timer_cb(struct timer_list *t)
```

```
{
      struct my_dev *dev = from_timer(dev, t, my_timer);
      printk(KERN_INFO "dev->foo is: %d\n”, dev->foo);
      dev->foo++;
      mod_timer(dev->my_timer, dev->foo * HZ + jiffies);
}

static int __init ece_timer_init(void) {
      bar_dev.foo = 1;
      timer_setup(&bar.my_timer, timer_cb, 0);
      mod_timer(&bar.my_timer, 1 * HZ + jiffies);
}
```

a. Outline what is happening in this code, from the init to the timer callback.

   In init it initializes the integer foo to 1 for use in the timer in bar_dev which holds the timer_list struct for the timer to be used in the kernel. From there timer_setup() adds the timer to a list and sets its call back function with flags set to 0. From there mod_timer arms the timer for functionality by passing the timer_list struct and the interval of time you want to execute the callback function on. In this case, the interval is set to 1 second past the current time in jiffies.

   In the timer_cd() function which is the call back function it acquires the my_dev struct using from_timer(). From there it prints the current value of foo which will be 1 on the first execution. After the print, it increments the foo integer by 1. Then is rearms the timer using mod_timer just like in init but with the new value of foo.

   In this case it will be 1 second longer, so the functionality will, after init, which is set to 1, it will execute the callback function 1 second later. After foo is incremented and the timer is armed again, it will be 1 second longer than the last interval. So each time the timer callback function is called it will be called again 1 second on top of the previous value. For example, after the first call to timer_cb() it will rearm the timer to 2 second, then on the next call, 3 second and so on.

b. What do the first three lines of output look like, and at what intervals are they printed?

dev->foo is: 1, printed after a 1 second interval

dev->foo is: 2, printed after a 2 second interval from first execution of timer_cb()

dev->foo is: 3, printed after a 3 second interval from the second execution of timer_cb()

16. Consider the following PCI probe code and register definition. Using MMIO access, write the code that does the following actions. Note that NASA had to stay within budget, so their onboard command processors are cheap and slow, and can only accept new commands every 400 microseconds. But the processors must stay running in between commands, and cannot sleep. While no commands are pending, solar arrays should be deployed to keep the batteries warm. Otherwise, they'll freeze in the depth of space. (20)

    a. First, write the code to retract the solar array, followed by a command to turn on the camera. After 2 seconds, turn off the camera, followed by turning the solar array back on.

b. Next, write the code to send data back to Earth for analysis. Kepler can send 1 second of captured data at a time before transmission must be stopped for one command cycle (400 microseconds). Ensure all 2 seconds of data is sent.
c. Finally, once all data is sent, re-engage the solar arrays.

You do not need to write `init()` or `exit()` functions, or the PCIe `remove()` function. Just any random function containing the register accesses will suffice.

```
static void *hw_addr;

static int kepler_probe(struct pci_dev *pdev,
                        const struct pci_device_id *ent)
{
      int err;
      err = pci_enable_device_mem(pdev);
      if (err)
            return err;
      err = pci_request_selected_regions(pdev,
                                    pci_select_bars(pdev, 0),
                                    DEVNAME);
      if (err) {
            pci_disable_device(pdev);
            return err;
      }

      pci_set_drvdata(pdev, &kepler_driver);
      hw_addr = ioremap(pci_resource_start(pdev, 0),
                        pci_resource_len(pdev, 0));

      return 0;
}
```

Planetary Hunting Astrophysics Telescope - PHATCTL (0x00420) (RW)

Field	Bit(s)	Init. Val	Description
SOLAR ARRAY DEPLOYMENT	0	0b	0b = retract solar array 1b = deploy solar array
Reserved	2:1	??	Reserved bits, do not write
CAMERA CONTROL	3	0b	0b = camera off 1b = camera on
Reserved	7:4	??	Reserved bits, do not write
E.T. SIGNAL PROCESSING	8	0b	0b = stop transmission 1b = phone home
Reserved	30:9	??	Reserved bits, do not write
RUN COMMAND	31	0b	0b = N/A 1b = Process current command (clear on write)

See attached text file, sorry i tried to copy and paste but format was way off, the newlines did not transition into the google doc.