ECE 373 – EMBEDDED OPERATING SYSTEMS AND DEVICE DRIVERS
Spring 2022

Final Exam – 6-June-2022

Name: ___________James Ross____________

1. List the three main types of kernel drivers. Explain main reasons why they're different. (3)

Character Driver, Block driver, network driver. The difference between them are the character driver works like a stream of data, a block driver works in transferring chunks of data at a time and the network drivers are connected to the network stack and handle incoming and outgoing packets.

2. What kernel communication mechanism is being used from userspace when a processor switches from protected mode (userspace) to privileged mode (kernel)? Give one example of one of these functions that can make this transition. (3)

The kernel mechanisms being used from userspace to kernel space are system calls. System calls are a set of functions with defined arguments that go down into the kernel space for the kernel to interpret and work with. One example is the open() system call which opens a file and returns a file descriptor which is associated with the open file. read() and write() are also system calls to pass or receive data to and from the kernel for the kernel to react to or operate on and return or write.

3. On what types of devices can you find embedded operating systems, and why would they be considered embedded? (4)

You can find embedded operating systems on small devices that do a specific set of operations in a usually controlled device environment. Small systems with limited resources are often found in embedded devices and can be found in things like traffic lights or custom devices like a smart watch or a device that contains sensors to read data and transfer/display the data to a desired location, online or on the device's potential flash memory and the like.

4. Your driver is running well, but keeps looking for data too soon from the sensor. You need to delay for 25 milliseconds to correct this. However, the code is running in an interrupt context. Write the bit of code you would insert into your driver to properly wait for 25 milliseconds. (5)

mdelay(25);

5. Consider the following code, where "`bar`" is 7 when passed into `my_func()`: (15)

```
static struct my_test {
   struct timer_list final_timer;
   int baz;
} my_oh_my;

void final_countdown(struct timer_list *t)
{
   struct my_test *wee = from_timer(wee, t, final_timer);
   int foo = wee->baz;

   printk(KERN_INFO "your foo is strong: %d\n", foo);

   foo--;
   if (foo > 0)
         mod_timer(&final_timer, (jiffies + (foo * HZ)));
}

void my_func(int bar)
{
   my_oh_my.baz = 4;

   timer_setup(&my_oh_my.final_timer, final_countdown, 0);
   mod_timer(&my_oh_my.final_timer, (jiffies + (bar * HZ)));
   return;
}
```

a. What is the output of this code snippet, and at what intervals?

after my_func is called, the timer is armed for 7 seconds after the current time in jiffies.
On first pass the output is - your foo is strong: 7
second pass - your foo is strong: 6
third pass - your foo is strong: 5
this trend of decreasing values continues until the last pass when foo gets set to 0.
on 7th pass foo is 0, output - your foo is strong: 1
then the timer does not get armed again since foo is decremented to 0 after the print statement.

b. What is the relationship of `jiffies` to `HZ` to actual wall clock time?

HZ is jiffies per second, jiffies is an internal count in the kernel to keep track of time (not related to wall clock time). Together they can be used to set jiffies to a later time that represents wall clock time. When adding jiffies to X*HZ, it will be X seconds farther in the future wall clock time in jiffies.

c. What code would you need to properly clean up the timer when your module is removed from the kernel?

```
del_timer_sync(my_oh_my.final_timer);
```

6. Explain what is the purpose of the BAR in a PCI device. Outline the basic steps to map the BAR for use in your kernel driver. (5)

The BAR is the base address register of the device. This is used to access the MMIO of the device with an offset address when added to the BAR is the location of a register in MMIO. This is used to alter the registers in the device and control it.

The basic steps of mapping the BAR is enabling device mem, getting the BARs from the device, requesting selected regions with the BARs, getting the mmio start and mmio len with the BAR number that holds the MMIO mappings, then using ioremap to set the memory for the final BAR in memory that we used to access the device.

Here is a code snippet of the process -

```
pci_enable_device_mem(pdev);
bars = pci_select_bars(pdev, IORESOURCE_MEM);
pci_requiest_selected_regions(pdev, bars, DEV_NAME);
mmio_start = pci_resource_start(pdev, MM_BAR0);
mmio_len = pci_resource_len(pdev, MM_BAR0);
dev->hwbar = ioremap(mmio_start, mmio_len);
```

This gives the BAR address used in MMIO for the device found on BAR0 of the device.

7. What is the difference in behavior between `WARN()` and `BUG()` inside kernel code? (5)

WARN() will dump a stack trace and register information on the CPU's along with other information as if there was a kernel panic or crash. It will not stop the driver from running and just outputs the contents into dmesg and other logging locations.

BUG() does the same thing as warn but crashes the kernel once it is done dumping information.

8. Explain what memory pinning is actually doing and why it is important. Then write code how you'd pin a chunk of memory named `msg_buf` already allocated from `kmalloc()`, and then unpin it. (10)

Memory pinning is taking a kernel address and setting it so the memory manager does not swap the data to another location while it is operating. It pins the physical memory. This is important so when data from things such as DMA will know for sure the location they are reading or writing to is going to be the right data. If the data switches from the memory manager and different data is there it could corrupt the memory by writing data to the wrong spot and potentially breaking your kernel. If it is reading the wrong information because the memory got swapped to another location it could badly affect the device's operation.

variables - buff_info (holds DMA address and size of pinned buffer)

```
buff_info->size = MSG_BUF_SIZE;
buff_info->dma = dma_map_single(&pdev->dev, msg_buf, buff_info->size,
DMA_FROM_DEVICE);

dma_unmap_single(&pdev->dev, buff_info->dma, buff_info->size, DMA_FROM_DEVICE);
```

9. Calvin wants his watterson driver included in the kernel build. The watterson driver depends on the pci and hobbes modules, and needs to have the option of being a loadable module or directly linked into the kernel. What should the Kconfig entry look like? (5)

```
config WATTERSON
    tristate "watterson driver"
    depends on PCI
    depends on hobbes
    help
        This driver is the watterson driver
```

10. Consider the following bit of **userspace** code and 16-bit register definition: (10)

BRW_CTL: 0x00630

Bits	**Description**
1-0	00b: LEDs off 01b: Right LED on 10b: Left LED on 11b: Reserved
5-2	Reserved
6	0b: Afterburner off 1b: Afterburner on
9-7	000b: Brewing off 001b: Drip coffee mode on 010b: K-cup mode on 100-111b: Reserved
15-10	Reserved

```
int fd = open("/dev/mem", O_RDWR);

void *mem = mmap(NULL, sizeof(PCI_BAR0), PROT_READ | PROT_WRITE, MAP_SHARED,
fd, BAR0_OFFSET);
```

a. Assuming Reserved fields are written with a 0b, what would the value be to turn on the Right LED while brewing K-cup coffee with the afterburner enabled?

0x0041

b. Using the `mmap()`'d segment above and the register definition above, write this value to the `BRW_CTL` register. Reminder: this is in userspace, not in-kernel.

```
#define HWREG_U32(bar, offset) ((u32*)((char*)bar + offset))
#define BRW_CTL 0x00630
#define R_LED_AFTERBURN_ON 0x0041
u32 *mem_reg = HWREG_U32(mem, BRW_CTL);
*mem_reg = R_LED_AFTERBURN_ON;
```

11. Consider the following two code threads operating on the same global variable x: (10)

<u>Thread A</u>

```
static int x = 6;

int y;
x = x + 4;
y = x + 2;
printk(KERN_INFO "y is %d\n", y);
```

<u>Thread B</u>

```
static int x = 6;

int y;
x = x + 3;
y = x + 3;
printk(KERN_INFO "y is %d\n", y);
```

a. Assuming both threads start at the same exact time, what would the output look like?
It would depend on which thread reaches x first i think. If they started at the exact same time, they can read the variable without issue but the writing is the issue. If they both read X at the exact same time before incrementing both threads would read x=6. Then they would increment at the same time which would be undefined behavior. It would depend on who gets access to the write first as i don't think you can literally write to the variable at the exact same time and know what the result would be.

If thread A writes first, it would set x to 10. Then thread B would get access to the write and overwrite X with 9 since they read the variable at the exact same time, both read x=6.

Assuming thread A got access to the write first, the output of Thread A would be 9 + 2 which would print - y is 11.

Then thread B would be 9 + 3 since thread B got the second write, which would output - y is 12.

b. Rewrite each thread to make them "better."
To fix this we can use mutex's

<u>Thread A</u>

```
struct mutex mlock; /* global */

int y;

mutex_init(&mlock);
mutex_lock(&mlock);

x = x + 4;
y = x + 2;

mutex_unlock(&mlock);
printk(KERN_INFO "y is %d\n", y);
```

Thread B
struct mutex mlock; /* same global seen in thread A example */

int y;

mutex_init(&mlock);
mutex_lock(&mlock);

x = x + 3;
y = x + 3;

mutex_unlock(&mlock);
printk(KERN_INFO, y is %d, y);

12. Consider the following interrupt handler. What is wrong with this code? (5)

```
static irqreturn_t phht_intr(int irq, void *data)
{
      struct pci_device *dev = data;
      struct phht_adapter *adapter = pci_getdrvdata(dev);
      unsigned char *buffer;

      mutex_lock(&adapter->my_mutex);
      adapter->sg_list = kmalloc(16, GFP_ATOMIC);
      mutex_unlock(&adapter->my_mutex);

      return IRQ_HANDLED;
}
```

It is using a mutex which can put the driver to sleep. This can't happen in an interrupt context since storing an interrupt state for switching back and forth to it isn't supported. The kmalloc is okay since it is using the GFP_ATOMIC flag.

13. Using the descriptor description and register below, answer the following questions: (20)

63 0
Buffer address

Reserved	Length	Command	Error	Status
63 56	55 40	39 24	23 8	7 0

Status flags

3-0	0000b: No activity 0011b: Descriptor Done 1100b: End of Packet
7-4	Reserved

Command flags

3-0	0000b: No activity 0011b: Load Descriptor 1100b: Clear Interrupt
7-4	0011: Start transmit 1100: Compute checksum
15-8	Reserved

a. Define the struct in C that would define the descriptor. Assume the device is little-endian.

```
struct my_desc {
        __le64 buffer_addr;
        union {
            __le64 data;
            struct {
                u8 status;
                __le16 error;
                __le16 cmd;
                __le16 len;
                __le8 reserved;
            } fields;
        } desc_data;
};
```

b. Write the code to fill out the descriptor with the following settings: a 1Kbyte buffer of pinned memory, set the length field, clear the status field, and start transmit along with computing the checksum.

See attached file for code. Done in VIM so format doesn't get messy on google docs.