EEL5733/EEL4732 Advanced Systems Programming

Final Exam

Student Name:

UFID:

Signature:

Important Note There are four questions. Please read all the questions carefully and write your name on all answer pages. Good luck.

Questions

1. (7 pts) In a multi-process application, a developer uses the `mmap` system call to let two processes in the application to communicate via shared memory without a backing file. For some reason, the communication does not work, i.e., one process does not see the values written by the other process and vice versa. Clearly describe with code snippets two different scenarios such that in each scenario a different type of programming mistake prevents correctly setting up memory sharing based inter-process communication. Please note that compilation errors would not be accepted. Also, make sure to include the mmap call(s) and any other relevant in your code snippets, and briefly explain the mistake for each case. *Hint:* Think about how to properly use the `mmap` system call to enable inter-process communication when there is no backing file and figure out how to introduce the relevant bugs.

   (a) Scenario 1

   (b) Scenario 2

2. (9 pts) This question is about the `short` driver that was discussed in the Interrupt Handling chapter of the Linux Device Drivers book (`https://static.lwn.net/images/pdf/LDD3/ch10.pdf`). The source code of the driver can also be found at `https://github.com/martinezjavier/ldd3/blob/master/short/short.c`.

   (a) What happens if the call to the `ioremap` function at line 21 in the `short_init` function gets commented out? Explain.

   (b) List all line no's that request some kernel resources in the code snippet of the `short_init` function below and briefly explain each kernel resource that you mention.

   (c) List the names of all top handlers as registered by the `short_init` function. Briefly explain how each of the top handler differs from the others.

```
 1 int short_init(void)
 2 {
 3   int result;
 4
 5   short_base = base;
 6   short_irq = irq;
 7
 8   if (!use_mem) {
 9      if (! request_region(short_base, SHORT_NR_PORTS, "short")) {
10         printk(KERN_INFO "short: can't get I/O port address 0x%lx\n",
11                                         short_base);
12         return -ENODEV;
13      }
14
15   } else {
16        if (! request_mem_region(short_base, SHORT_NR_PORTS, "short")) {
17                         printk(KERN_INFO "short: can't get I/O mem address 0x%lx\n",
18                                         short_base);
19                         return -ENODEV;
20        }
21       short_base = (unsigned long) ioremap(short_base, SHORT_NR_PORTS);
22   }
23   result = register_chrdev(major, "short", &short_fops);
24   if (result < 0) {
25      printk(KERN_INFO "short: can't get major number\n");
26                 release_region(short_base,SHORT_NR_PORTS);
27                 return result;
28   }
29   if (major == 0) major = result; /* dynamic */
30   short_buffer = __get_free_pages(GFP_KERNEL,0);
31   short_head = short_tail = short_buffer;
```

Name of the bottom handler

```
INIT_WORK(&short_wq, (void (*)(struct work_struct *)) short_do_tasklet);


if (short_irq < 0 && probe == 1)
   short_kernelprobe();

if (short_irq < 0 && probe == 2)
   short_selfprobe();

if (short_irq < 0)
      switch(short_base) {
                  case 0x378: short_irq = 7; break;
                  case 0x278: short_irq = 2; break;
                  case 0x3bc: short_irq = 5; break;
      }

if (short_irq >= 0 && share > 0) {
   result = request_irq(short_irq, short_sh_interrupt,
                                   IRQF_SHARED,"short",
                              short_sh_interrupt);
   if (result) {
      printk(KERN_INFO "short: can't get assigned irq %i\n", short_irq);
      short_irq = -1;
    }
    else {
        outb(0x10, short_base+2);
    }
    return 0;
}

 if (short_irq >= 0) {
    result = request_irq(short_irq, short_interrupt,
                                   0, "short", NULL);
    if (result) {
       printk(KERN_INFO "short: can't get assigned irq %i\n",
                                      short_irq);
       short_irq = -1;
    }
    else {
         outb(0x10,short_base+2);
    }
}

if (short_irq >= 0 && (wq + tasklet) > 0) {
   free_irq(short_irq,NULL);
   result = request_irq(short_irq,
                              tasklet ? short_tl_interrupt :
                              short_wq_interrupt,
                                0, "short-bh", NULL);
      if (result) {
         printk(KERN_INFO "short-bh: can't get assigned irq %i\n",
                                        short_irq);
         short_irq = -1;
      }
   }
   return 0;
}
```

3. (9 pts) This question is about the `scull` driver. You can find the full source code at
`https://github.com/martinezjavier/ldd3/tree/master/scull`.

   (a) In the `scull_init_module` function, what happens if `scull_setup_cdev(&scull_devices[i], i);` is executed as the first statement of the body of the `for` loop, e.g, switch the statements at lines 38 and 41?

   (b) What is the role of the `container_of` function that is used in the `scull_open` function (line 67)? Explain.

   (c) What would be the side-effect of commenting out the assignment statement `filp->private_data = dev;` at line 68 in the `scull_open` function? Explain.

```c
1  struct scull_dev {
2          struct scull_qset *data;  /* Pointer to first quantum set */
3          int quantum;              /* the current quantum size */
4          int qset;                 /* the current array size */
5          unsigned long size;       /* amount of data stored here */
6          unsigned int access_key;  /* used by sculluid and scullpriv */
7          struct mutex lock;     /* mutual exclusion semaphore     */
8          struct cdev cdev;          /* Char device structure                */
9  };
10 
11 int scull_init_module(void)
12 {
13         int result, i;
14         dev_t dev = 0;
15 
16         if (scull_major) {
17                 dev = MKDEV(scull_major, scull_minor);
18                 result = register_chrdev_region(dev, scull_nr_devs, "scull");
19         } else {
20                 result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs,
21                                 "scull");
22                 scull_major = MAJOR(dev);
23         }
24         if (result < 0) {
25                 printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
26                 return result;
27         }
28 
29         scull_devices = kmalloc(scull_nr_devs * sizeof(struct scull_dev), GFP_KERNEL);
30         if (!scull_devices) {
31                 result = -ENOMEM;
32                 goto fail;  /* Make this more graceful */
```

```
        }
        memset(scull_devices, 0, scull_nr_devs * sizeof(struct scull_dev));

        /* Initialize each device. */
        for (i = 0; i < scull_nr_devs; i++) {
                scull_devices[i].quantum = scull_quantum;
                scull_devices[i].qset = scull_qset;
                mutex_init(&scull_devices[i].lock);
                scull_setup_cdev(&scull_devices[i], i);
        }

        return 0; /* succeed */

  fail:
        scull_cleanup_module();
        return result;
}

static void scull_setup_cdev(struct scull_dev *dev, int index)
{
        int err, devno = MKDEV(scull_major, scull_minor + index);

        cdev_init(&dev->cdev, &scull_fops);
        dev->cdev.owner = THIS_MODULE;
        dev->cdev.ops = &scull_fops;
        err = cdev_add (&dev->cdev, devno, 1);
        if (err)
                printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}

int scull_open(struct inode *inode, struct file *filp)
{
        struct scull_dev *dev;

        dev = container_of(inode->i_cdev, struct scull_dev, cdev);
        filp->private_data = dev;

        if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
                if (mutex_lock_interruptible(&dev->lock))
                        return -ERESTARTSYS;
                scull_trim(dev); /* ignore errors */
                mutex_unlock(&dev->lock);
        }
        return 0;
}

loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
        struct scull_dev *dev = filp->private_data;
        loff_t newpos;

        switch(whence) {
          case 0: /* SEEK_SET */
                newpos = off;
                break;

          case 1: /* SEEK_CUR */
                newpos = filp->f_pos + off;
                break;

          case 2: /* SEEK_END */
                newpos = dev->size + off;
                break;

          default: /* can't happen */
                return -EINVAL;
        }
        if (newpos < 0) return -EINVAL;
        filp->f_pos = newpos;
        return newpos;
}
```

4. (10 pts) This question is about the `usbkbd` driver that we studied in class. The source code is provided with the line numbers below for your reference.

   (a) What would be the consequence of commenting out the call to the `usb_submit_urb` function in the `usb_kbd_open` function (lines 236 and 237)? Explain in terms of the `usb_kbd_irq` and the `usb_submit_led` functions.

   (b) Which buffer(s) in the driver may get accessed by the keyboard by transferring data to or transferring data from? Explain how you identify those buffer(s) and the type of operation (read or write) that gets performed by the device for each such buffer that you identify.

   (c) Why does the `usbkbd` driver does not define any Virtual File System entry points? Does it mean the keyboard device cannot be accessed through the `open` system call from the user space? Explain.

   (d) Specify the line number from which the very first LED urb get submitted. Also, explain what gets achieved with the submission of an LED urb?

```
struct usb_kbd {
        struct input_dev *dev;
        struct usb_device *usbdev;
        unsigned char old[8];
        struct urb *irq, *led;
        unsigned char newleds;
        char name[128];
        char phys[64];

        unsigned char *new;
        struct usb_ctrlrequest *cr;
        unsigned char *leds;
        dma_addr_t new_dma;
        dma_addr_t leds_dma;

        spinlock_t leds_lock;
        bool led_urb_submitted;

};

static void usb_kbd_irq(struct urb *urb)
{
        struct usb_kbd *kbd = urb->context;
        int i;

        switch (urb->status) {
        case 0:                 /* success */
                break;
        case -ECONNRESET:       /* unlink */
        case -ENOENT:
        case -ESHUTDOWN:
                return;
        /* -EPIPE:  should clear the halt */
        default:                /* error */
                goto resubmit;
        }

        for (i = 0; i < 8; i++)
                input_report_key(kbd->dev, usb_kbd_keycode[i + 224],
                           (kbd->new[0] >> i) & 1);

        for (i = 2; i < 8; i++) {

                if (kbd->old[i] > 3 &&
                         memscan(kbd->new + 2, kbd->old[i], 6) == kbd->new + 8) {
                        if (usb_kbd_keycode[kbd->old[i]])
                                input_report_key(kbd->dev,
                                                  usb_kbd_keycode[kbd->old[i]], 0);
                        else
```

```
                                hid_info(urb->dev,
                                         "Unknown key (scancode %#x) released.\n",
                                         kbd->old[i]);
                }

                if (kbd->new[i] > 3 && memscan(kbd->old + 2,
                                  kbd->new[i], 6) == kbd->old + 8) {
                        if (usb_kbd_keycode[kbd->new[i]])
                                input_report_key(kbd->dev,
                                          usb_kbd_keycode[kbd->new[i]], 1);
                        else
                                hid_info(urb->dev,
                                         "Unknown key (scancode %#x) pressed.\n",
                                         kbd->new[i]);
                }
        }

        input_sync(kbd->dev);

        memcpy(kbd->old, kbd->new, 8);

resubmit:
        i = usb_submit_urb (urb, GFP_ATOMIC);
        if (i)
                hid_err(urb->dev, "can't resubmit intr, %s-%s/input0, status %d",
                        kbd->usbdev->bus->bus_name,
                        kbd->usbdev->devpath, i);
}

static int usb_kbd_event(struct input_dev *dev, unsigned int type,
                         unsigned int code, int value)
{
        unsigned long flags;
        struct usb_kbd *kbd = input_get_drvdata(dev);

        if (type != EV_LED)
                return -1;

        spin_lock_irqsave(&kbd->leds_lock, flags);
        kbd->newleds = (!!test_bit(LED_KANA,    dev->led) << 3) |
                          (!!test_bit(LED_COMPOSE, dev->led) << 3) |
                       (!!test_bit(LED_SCROLLL, dev->led) << 2) |
                          (!!test_bit(LED_CAPSL,   dev->led) << 1) |
                       (!!test_bit(LED_NUML,    dev->led));

        if (kbd->led_urb_submitted){
                spin_unlock_irqrestore(&kbd->leds_lock, flags);
                return 0;
        }
        if (*(kbd->leds) == kbd->newleds){
                spin_unlock_irqrestore(&kbd->leds_lock, flags);
                return 0;
        }

        *(kbd->leds) = kbd->newleds;

        kbd->led->dev = kbd->usbdev;
        if (usb_submit_urb(kbd->led, GFP_ATOMIC))
                pr_err("usb_submit_urb(leds) failed\n");
        else
                kbd->led_urb_submitted = true;

        spin_unlock_irqrestore(&kbd->leds_lock, flags);

        return 0;
}

static void usb_kbd_led(struct urb *urb)
{
        unsigned long flags;
        struct usb_kbd *kbd = urb->context;

        if (urb->status)
                hid_warn(urb->dev, "led urb status %d received\n",
                         urb->status);

        spin_lock_irqsave(&kbd->leds_lock, flags);

        if (*(kbd->leds) == kbd->newleds){
                kbd->led_urb_submitted = false;
                spin_unlock_irqrestore(&kbd->leds_lock, flags);
                return;
        }

        *(kbd->leds) = kbd->newleds;

        kbd->led->dev = kbd->usbdev;
        if (usb_submit_urb(kbd->led, GFP_ATOMIC)){
                hid_err(urb->dev, "usb_submit_urb(leds) failed\n");
                kbd->led_urb_submitted = false;
        }
        spin_unlock_irqrestore(&kbd->leds_lock, flags);

}

static int usb_kbd_open(struct input_dev *dev)
{
	struct usb_kbd *kbd = input_get_drvdata(dev);

	kbd->irq->dev = kbd->usbdev;
	if (usb_submit_urb(kbd->irq, GFP_KERNEL))
		return -EIO;

	return 0;
}

static void usb_kbd_close(struct input_dev *dev)
{
	struct usb_kbd *kbd = input_get_drvdata(dev);

	usb_kill_urb(kbd->irq);
}

static int usb_kbd_alloc_mem(struct usb_device *dev, struct usb_kbd *kbd)
{
	if (!(kbd->irq = usb_alloc_urb(0, GFP_KERNEL)))
		return -1;
	if (!(kbd->led = usb_alloc_urb(0, GFP_KERNEL)))
		return -1;
	if (!(kbd->new = usb_alloc_coherent(dev, 8, GFP_ATOMIC, &kbd->new_dma)))
		return -1;
	if (!(kbd->cr = kmalloc(sizeof(struct usb_ctrlrequest), GFP_KERNEL)))
		return -1;
	if (!(kbd->leds = usb_alloc_coherent(dev, 1, GFP_ATOMIC, &kbd->leds_dma)))
		return -1;

	return 0;
}

static void usb_kbd_free_mem(struct usb_device *dev, struct usb_kbd *kbd)
{
	usb_free_urb(kbd->irq);
	usb_free_urb(kbd->led);
	usb_free_coherent(dev, 8, kbd->new, kbd->new_dma);
	kfree(kbd->cr);
	usb_free_coherent(dev, 1, kbd->leds, kbd->leds_dma);
}

static int usb_kbd_probe(struct usb_interface *iface,
			 const struct usb_device_id *id)
{
	struct usb_device *dev = interface_to_usbdev(iface);
	struct usb_host_interface *interface;
	struct usb_endpoint_descriptor *endpoint;
	struct usb_kbd *kbd;
	struct input_dev *input_dev;
```

```c
        int i, pipe, maxp;
        int error = -ENOMEM;

        interface = iface->cur_altsetting;

        if (interface->desc.bNumEndpoints != 1)
                return -ENODEV;

        endpoint = &interface->endpoint[0].desc;
        if (!usb_endpoint_is_int_in(endpoint))
                return -ENODEV;

        pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
        maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));

        kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
        input_dev = input_allocate_device();
        if (!kbd || !input_dev)
                goto fail1;

        if (usb_kbd_alloc_mem(dev, kbd))
                goto fail2;

        kbd->usbdev = dev;
        kbd->dev = input_dev;
        spin_lock_init(&kbd->leds_lock);

        if (dev->manufacturer)
                strlcpy(kbd->name, dev->manufacturer, sizeof(kbd->name));

        if (dev->product) {
                if (dev->manufacturer)
                        strlcat(kbd->name, " ", sizeof(kbd->name));
                strlcat(kbd->name, dev->product, sizeof(kbd->name));
        }

        if (!strlen(kbd->name))
                snprintf(kbd->name, sizeof(kbd->name),
                         "USB HIDBP Keyboard %04x:%04x",
                         le16_to_cpu(dev->descriptor.idVendor),
                         le16_to_cpu(dev->descriptor.idProduct));

        usb_make_path(dev, kbd->phys, sizeof(kbd->phys));
        strlcat(kbd->phys, "/input0", sizeof(kbd->phys));

        input_dev->name = kbd->name;
        input_dev->phys = kbd->phys;
        usb_to_input_id(dev, &input_dev->id);
        input_dev->dev.parent = &iface->dev;

        input_set_drvdata(input_dev, kbd);

        input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_LED) |
                BIT_MASK(EV_REP);
        input_dev->ledbit[0] = BIT_MASK(LED_NUML) | BIT_MASK(LED_CAPSL) |
                BIT_MASK(LED_SCROLLL) | BIT_MASK(LED_COMPOSE) |
                BIT_MASK(LED_KANA);

        for (i = 0; i < 255; i++)
                set_bit(usb_kbd_keycode[i], input_dev->keybit);
        clear_bit(0, input_dev->keybit);

        input_dev->event = usb_kbd_event;
        input_dev->open = usb_kbd_open;
        input_dev->close = usb_kbd_close;

        usb_fill_int_urb(kbd->irq, dev, pipe,
                         kbd->new, (maxp > 8 ? 8 : maxp),
                         usb_kbd_irq, kbd, endpoint->bInterval);
        kbd->irq->transfer_dma = kbd->new_dma;
        kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

        kbd->cr->bRequestType = USB_TYPE_CLASS | USB_RECIP_INTERFACE;
        kbd->cr->bRequest = 0x09;
        kbd->cr->wValue = cpu_to_le16(0x200);
        kbd->cr->wIndex = cpu_to_le16(interface->desc.bInterfaceNumber);
        kbd->cr->wLength = cpu_to_le16(1);

        usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),
                             (void *) kbd->cr, kbd->leds, 1,
                             usb_kbd_led, kbd);
        kbd->led->transfer_dma = kbd->leds_dma;
        kbd->led->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

        error = input_register_device(kbd->dev);
        if (error)
                goto fail2;

        usb_set_intfdata(iface, kbd);
        device_set_wakeup_enable(&dev->dev, 1);
        return 0;

fail2:
        usb_kbd_free_mem(dev, kbd);
fail1:
        input_free_device(input_dev);
        kfree(kbd);
        return error;
```

```
}

static void usb_kbd_disconnect(struct usb_interface *intf)
{
        struct usb_kbd *kbd = usb_get_intfdata (intf);

        usb_set_intfdata(intf, NULL);
        if (kbd) {
                usb_kill_urb(kbd->irq);
                input_unregister_device(kbd->dev);
                usb_kill_urb(kbd->led);
                usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
                kfree(kbd);
        }
}

static struct usb_device_id usb_kbd_id_table [] = {
        { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
                USB_INTERFACE_PROTOCOL_KEYBOARD) },
        { }                                             /* Terminating entry */
};

MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);

static struct usb_driver usb_kbd_driver = {
        .name =         "usbkbd",
        .probe =        usb_kbd_probe,
        .disconnect =   usb_kbd_disconnect,
        .id_table =     usb_kbd_id_table,
};

module_usb_driver(usb_kbd_driver);

```