CPSC 8220 FINAL EXAM, SPRING 2016

NAME: _____

Questions are short answer and multiple choice. For each of the multiple choice questions, put a CIRCLE around the letter of the ONE correct answer. Make sure that the circle encloses only ONE letter. DO NOT use check marks, Xs, or other notations.

1. (5) In our warm-up C code that was used to configure a NetApp file server, we noted that it was quite important to disable the default, echo-input on one of our file descriptors in order to avoid a cascading storm of error messages between the control system and the NetApp server. Which ioctl() call included this disable operation?

2. (5) Producing an SMP-safe driver for the Kyouko3 graphics card was complicated by the concurrent access to the circular DMA buffer pool on the part of the card interrupt handler and the driver's START_DMA code. A simple design to solve the problem uses a spinlock to serialize access and an additional boolean, *queuefull*, that indicates whether all buffers are in use (*queuefull == 1*) or not (*queuefull == 0*). The boolean is set/reset only while holding the lock and only when *fill==drain*. The driver's START_DMA code can then safely give up the lock and call:

   ```
   wait_event_interruptible(dma_snooze,!kyouko3.queuefull);
   ```

   In which routine should *queuefull* be set to 0, and in which routine should it be set to 1? Where in each?

   A. set to 1 in the interrupt handler, before advancing drain; set to 0 in the START_DMA code, before advancing fill;

   B. set to 1 in the interrupt handler, after advancing drain; set to 0 in the START_DMA code, after advancing fill;

   C. set to 0 in the interrupt handler, before advancing drain; set to 1 in the START_DMA code, before advancing fill;

   D. set to 0 in the interrupt handler, before advancing drain; set to 1 in the START_DMA code, after advancing fill;

   E. set to 1 in the interrupt handler, before advancing drain; set to 0 in the START_DMA code, after advancing fill;

   F. set to 0 in the interrupt handler, after advancing drain; set to 1 in the START_DMA code, before advancing fill;

G. set to 1 in the interrupt handler, after advancing drain; set to 0 in the START_DMA code, before advancing fill;

H. set to 0 in the interrupt handler, after advancing drain; set to 1 in the START_DMA code, after advancing fill;

3. (10) In the ioctl(BIND_DMA) call in our Kyouko3 graphics card driver, we needed to allocate and memory map NUM_DMA_BUFS dma buffers. Here is a loop to do that. You may assume a structure:

```
struct dma_buf {
        uint64_t pbase; // Buffer physical address.
        unsigned int *kbase; // Buffer kernel virtual address.
        unsigned int *ubase; // Buffer user virtual address.
        } k3_dma[NUM_DMA_BUFS];
```

Fill in the blanks below.

```
for(i=0;i<NUM_DMA_BUFS;i++){
        k3_dma[i].________________ = pci_alloc_consistent(NULL,BUFFER_SIZE,
                (dma_addr_t *) &k3_dma[i].________________);
        k3_dma[i].________________ = (unsigned int *) vm_mmap(fp,_____________,BUFFER_SIZE,
                PROT_READ|PROT_WRITE,MAP_SHARED,k3_dma[i].________________);
        }
```

4. (5) Suppose an active, singly-threaded process is not stopped, not traced, not exiting, and has no signals pending or blocked. If another process now sends it a SIGSTOP, give the sequence of function calls for the sending process, starting with *send_signal( )*, that will will set the target's TIF_SIGPENDING flag.

5. (5) In the previous question, now suppose the receiving process is multi-threaded. When it begins to handle the SIGSTOP, this receiving process calls *do_signal_stop( )*, which calls *task_participate_group_stop( )*. On this first call, does *task_participate_group_stop( )* return "true" or "false"?

6. (5) Can a process with state == TASK_DEAD call schedule()?

A. yes

B. no

7. (5) At the beginning of the call to

   ```
   static int kill_something_info(int sig, struct siginfo *info, pid_t pid)
   ```

   which variable holds the numerical process id of the process that is sending the signal?

   A. sig
   B. pid
   C. info→si_signo
   D. info→si_errno
   E. info→si_code
   F. info→si_pid
   G. none of these

8. (5) A process table entry in Linux is of what type?

   A. struct pid
   B. struct task_struct
   C. struct task
   D. struct process
   E. enum_pid_type
   F. struct_pid_task
   G. struct siginfo

9. (5) Which one of these will turn on the kill bit in the pending signals list for a non-threaded, current process?

   A. $current \rightarrow pending.signal.sig[0] = 1$
   B. $current \rightarrow pending.signal.sig[0] = SIGKILL$
   C. $current \rightarrow pending.signal.sig[0] \quad | = SIGKILL$
   D. $current \rightarrow pending.signal.sig[0] \quad | = (1 << (SIGKILL - 1))$
   E. $current \rightarrow pending.signal.sig[0] \quad | = (1 << SIGKILL)$
   F. $current \rightarrow pending.signal.sig[0] \quad | = (1 << (SIGKILL + 1))$
   G. $current \rightarrow signal \rightarrow shared_pending.signal.sig[0] = 1$
   H. $current \rightarrow signal \rightarrow shared_pending.signal.sig[0] = SIGKILL$
   I. $current \rightarrow signal \rightarrow shared_pending.signal.sig[0] \quad | = SIGKILL$
   J. $current \rightarrow signal \rightarrow shared_pending.signal.sig[0] \quad | = (1 << (SIGKILL - 1))$
   K. $current \rightarrow signal \rightarrow shared_pending.signal.sig[0] \quad | = (1 << SIGKILL)$
   L. $current \rightarrow signal \rightarrow shared_pending.signal.sig[0] \quad | = (1 << (SIGKILL + 1))$

10. (10) The Thermo Company manufactures a collection of PCI devices for PCs. The model Z is a card that provides users with the ability to monitor temperature and humidity inside the PC. The device holds three integer registers, each 32 bits. The first (lowest) is read-only and provides the current temperature. The second is read-only and provides the current humidity. The third is read-write and is used to turn the device on (1) or off (0). Fill in the blanks to complete this driver for the model Z. Open on the device should turn it on; close should turn it off. After calling mmap() on the device, users should be able to read temperature and humidity at offsets from the value returned. Thermo's vendor id is 0xd0be, and the device id for model Z is 0x007. The Linux major number is 411, and the minor number is 7. The base address of the three registers can be found in pci resource 0.

```
#include <linux/mman.h>
#include <asm/uaccess.h>
#include <linux/delay.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/irqflags.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/pci_ids.h>
#include <linux/spinlock.h>
#include <linux/sched.h>
#include <linux/ioctl.h>

MODULE_LICENSE("Proprietary");
MODULE_AUTHOR("ThermoZ Driver Author");

struct thermoZ_struct {
      unsigned int *p_control_base;
      unsigned int *k_control_base;
      unsigned int *u_control_base;
      } thermoZ;

int thermoZ_open(struct inode *inode, struct file *fp)
{
thermoZ._____=_____;
_____;
return 0;
}

int thermoZ_release(struct inode *inode, struct file *fp)
{
_____;
_____;
```

```
return 0;
}

int thermoZ_mmap(struct file *fp, struct vm_area_struct *vma)
{
_____;
return 0;
}

struct file_operations thermoZ_fops = {
        .open = _____,
        .release= _____,
        .mmap = _____,
        .owner = THIS_MODULE
        };

struct cdev thermoZ_cdev;
int thermoZ_probe(struct pci_dev *pci_dev, const struct pci_device_id *pci_id)
{
int result;
thermoZ.p_control_base = _____;
return 0;
}

void thermoZ_remove(struct pci_dev *pci_dev)
{
_____
return;
}

struct pci_device_id thermoZ_dev_ids[] = {
        {PCI_DEVICE(_____,_____)},
        {0}
        };

struct pci_driver thermoZ_pci_drv = {
        .name = "ThermoZ",
        .id_table = thermoZ_dev_ids,
        .probe = thermoZ_probe,
        .remove = thermoZ_remove,
        };

int init_TZ(void)
{
int result;
```

```
cdev_init(&_____, &_____);
thermoZ_cdev.owner = THIS_MODULE;
_____(&thermoZ_cdev, MKDEV(_____,_____) ,1);
result = pci_register_driver(&_____);
return 0;
}

void exit_TZ(void)
{
_____(&thermoZ_pci_drv);
_____(&thermoZ_cdev);
return;
}

module_init(_____);
module_exit(_____);
```

11. (5) What is the name of the function that *blk_finish_plug( )* calls to move the per-process plugged list of disk requests to the device request queue? In which function that we reviewed in class is it also called?

12. (10) For the noop scheduler, the function *__elv_next_request( )*, invoked in *scsi_request_fn( )* via a call to *blk_peek_request( )* will call *noop_dispatch( )* to move requests from the scheduler queue to the dispatch queue. Give the line of code from *__elv_next_request( )* that makes this call.

13. (5) The call to *scsi_dev_queue_ready()* is the principal call which determines whether we can queue another request or we've reached a hardware limit. Which of these inequalities does it check? To what value should we set *sdev→queue_depth* to ensure at most 1 request on the hardware at any given time?

14. (10) Suppose we have disk requests (1 block each) at starting locations 20, 82, 120, and 200, and the r/w head is at 100. If we assume that seek time is linear in seek distance (it is

nearly so), then the total waiting time to serve the requests in the order given would be W = 4*K*(100-20) + 3*K*(82-20) + 2*K*(120-82) + K*(200-120) = 662*K, where K is the proportionality constant. Thus the average waiting time would be 165.5*K. What would be the average waiting time if the requests were served in greedy order? What would be the average waiting time if the requests were served in this order: 120, 82, 20, 200?

15. (5) In your disk scheduler, you implemented functions *greedy_former_request( )* and *greedy_latter_request( )*. Which functions, called in *blk_queue_bio( )*, ultimately call these scheduler functions?

16. (5) Which line of code in *ext4_mpage_readpages( )* detects non-sequential physical disk blocks in building a struct bio and, as a result, submits the current bio in order to start a new one?