Drivers for exams

August 24, 2024

1 DRIVER K24

```c
// Write a character device driver with the structure below and these specifications
// Each write needs to return bytes equal to a single sizeof(patient_data)
// Sequential reads need to return sequential measurements only from when open happened and on
// If for any reason sequential reads can't happen, then the second read should return EOF
// after an EOF, for a new stream of data, the process needs to close() and open() again
// if a process reads all measurements, it sleeps and waits for new measurements through the interrupt

struct medical_dev {
    // DONE: Lock type ?
    spinlock_t lock;

    waitqueue_t wq;
    uint128_t cnt; /* Initialised to zero and will never wrap */

#define CIRC_BUF_SIZE (1024 * sizeof(struct patient_data))
    char circ_buffer[CIRC_BUF_SIZE];
} medical_dev;

void intr(void) {
    struct medical_dev *dev = &medical_dev;
    struct patient_data pd;

    get_patient_data_from_hw(&pd); /* Get data from real device */

    spin_lock_irq(&medical_dev->lock);
    memcpy(&dev->circ_buffer[dev->cnt % CIRC_BUF_SIZE],
       &pd, sizeof(struct patient_data));
    dev->cnt += sizeof(struct patient_data);
    spin_unlock_irq(&medical_dev->lock);

    wake_up_interruptable(&dev->wq);
}

struct chrdev_state {
    // DONE: Lock type ?
    struct semaphore lock;

    struct medical_dev *medical_dev;

    char local_buf[sizeof(struct patient_data)];
    uint128_t local_cnt; /* Suppose it will never wrap */
}

static int medical_chrdev_open(struct inode *inode, struct file *filp) {
    struct chrdev_state *state;
    struct medical_dev *dev = &medical_dev;

    if ((ret = nonseekable_open(inode, filp)) < 0) {
        kfree(state);
        return -ENODEV;
    }

    state = kmalloc(sizeof(chrdev_state), GFP_KERNEL);
    if (!state) {
        return -ENOMEM;
    }

    state->local_cnt = dev->cnt;
    state->medical_dev = dev;
    filp->private_data = state;
    sema_init(&state->lock,1);

    return ret;
}

static ssize_t medical_chrdev_read(struct file *filp, char __user *usrbuf,
        size_t cnt, loff_t *f_pos) {

    struct chrdev_state *state;
    struct medical_dev *dev;

    state = filp->private_data;
    dev = state->medical_dev;

    uint32_t bytes_to_copy = sizeof(patient_data);

    if (down_interruptible(&state->lock)) {
        return -ERESTARTSYS;
    }

    // Do we need to fetch a new measurement
    if (dev->cnt == state->local_cnt) {
        if (wait_event_interruptable(dev->wq, dev->cnt > state->local_cnt) > 0) {
            up(&state->lock);
            return -ERESTARTSYS;
        }
    }

    if (*f_pos == 0) {
        if (dev->cnt - state->local_cnt >= CIRC_BUF_SIZE) {
            f_pos = 0;
            return 0;
        }
    }

    // Copy data from circular buffer
    memcpy(state->local_buf,
            &dev->circ_buffer[state->local_cnt % CIRC_BUF_SIZE],
            bytes_to_copy);

    // Send the data to user
    if (copy_to_user(usrbuf, state->local_buf, bytes_to_copy)) {
        up(&state->lock);
        return -EFAULT;
    }

    state->local_cnt += bytes_to_copy;
    *f_pos += bytes_to_copy;

    up(&state->lock);
    return bytes_to_copy;
}
```

2 DRIVER K23

```
typedef struct {} wait_queue_head_t;
//               status                dca_request
//                              FREE, RESERVED, FINISHED
typedef enum {
    DCA_REQ_FREE,
    DCA_REQ_RESERVED,
    DCA_REQ_FINISHED,
} dca_status_t;

//                buffer                      input       output
#define MAX_DATA_SZ (4 << 20)
typedef struct {
    int input[MAX_DATA_SZ];
    int result[MAX_DATA_SZ];
    dca_status_t status;
    wait_queue_head_t request_wq;
} dca_request_t;

#define MAX_REQS (64*sizeof(dca_request_t))
typedef struct {
    spinlock_t lock;
    wait_queue_head_t slots_wq;
    dca_request_t reqs[MAX_REQS];
} dca_dev_t;

dca_dev_t dca_dev;

//                                                state             request
void dca_req_set_status(dca_request_t *req, dca_status_t status) {
    req->status = status;
}

dca_request_t find_free_request(dca_dev_t *cdev) {
    int i;

    for (i=0; i<MAX_REQS; i++) {
        if (cdev->reqs[i].status == DCA_REQ_FREE) {
            return cdev->reqs[i];
        }
    }
    return NULL;
}

dca_request_t *dca_is_req_finished(dca_request_t *req) {
    return (req->status == DCA_REQ_FINISHED);
}

// Implemented elsewhere
void dca_notify_device(dca_request_t *req);

void dca_intr(void) {
    dca_dev_t *dev = &dca_dev;
    int i;
    // .. ? .. lock
    spin_lock_irq(&dev->lock);
    for (i=0; i<MAX_REQS; i++) {
        if (dca_is_req_finished(&dca_dev->reqs[i])) {
            wake_up_interruptable(&cdev->reqs[i].request_wq);
            dca_req_set_status(&cdev->req[i], DCA_REQ_FREE);
        }
    }
    spin_unlock_irq(&dev->lock);
    // .. ? .. unlock
}

static int dca_chrdev_open(struct inode *inode, struct file *filp) {
    int ret = 0;
    dca_dev_t *dev = &dca_dev;

    if((ret = nonseekable_open(inode,filp)) < 0) {
        ret = -ENODEV;
        goto out;
    }

    filp->private_data = dev;
    return ret;
}

typedef struct {
    uint8_t input[MAX_DATA_SZ];
    uint8_t result[MAX_DATA_SZ];
} dca_user_request_t;

#define DCA_MAGIC 'D'
#define DCA_SUBMIT_REQ _IORW(DCA_MAGIC, 0, cda_user_request_t);

static long dca_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {
    dca_user_request_t __user *argp = (dca_user_request_t __user *) arg;
    dca_dev_t *dev = filp->private_data;
    int ret = -ENOTTY;
    int retry = -ERESTARTSYS;
  if(_IOC_TYPE(cmd) != DCA_MAGIC) return ret;

    switch(cmd) {
        case DCA_SUBMIT_REQ:

            //                                                  slot        device,

            spin_lock_irq(&dev->lock);
            dca_request_t req = find_free_request(&dev);
            if (!req) {
                wait_event_interruptible(dev->slots_wq,find_free_request(&dev));
                req = find_free_request(&dev);
            }

            //                                              user space             request
            if (copy_from_user(&req->input, &argp->input, sizeof(argp->input))) {
                return -EFAULT;
            }

            //               status        request      reserved             request
    user
            dca_req_set_status(req, DCA_REQ_RESERVED);
            spin_unlock_irq(&dev->lock);

            //
            dca_notify_device(req);
            //                                request                 finished
            wait_event_interruptible(req->request_wq, req->status == DCA_REQ_FINISHED);
            if (copy_to_user(argp->result, req->result, sizeof(req->result))) {
                return -EFAULT;
            }
            break;

        default:
            return -EINVAL;
    }

}
```

3 DRIVER K22

```c
// Create a device driver that computes sparse to dense vectors and the other way around
// user space connects to the device via cvec_ioctl
// statuses for each slot in the buffer of the device: FREE, OCCUPIED, PROCESSED
// the driver only processes buffer slots that are OCCUPIED and an interrupt happens when
// the processing of a buffer slot is done to change the status of the slot to PROCESSED
// if there is no FREE slot in the buffer, than the process sleeps until there is one
#define DENSE_TO_SPARSE 0
#define SPARSE_TO_DENSE 0

// =======================
//                                  device
// =======================
//
// static int __init cvec_device_init(void) {
//     int i;
//     printk(KERN_INFO "Initializing cvec device\n");
//
//     // Initialize the device lock
//     mutex_init(&cvec_dev.lock);
//
//     // Initialize the wait queue
//     init_waitqueue_head(&cvec_dev.wq);
//
//     // Initialize buffer elements
//     for (i = 0; i < BUF_LEN; i++) {
//         cvec_dev.buffer[i].cvdesc = NULL;
//         cvec_dev.buffer[i].status = FREE;
//         cvec_dev.buffer[i].conversion_mode = DENSE_TO_SPARSE; // Default conversion mode
//     }
//
//     return 0;
// }
//
// module_init(cvec_device_init);

// Assume these two are implemented
int get_free_slot(struct cvec_device *cvdev);
int get_processed_slot(struct cvec_device *cvdev);

struct cvec_state {
    // TODO: what kind of lock here?
    struct semaphore lock;
    // TODO: done
    int conversion_mode;
}

struct cvec_descriptor{
    int len;
    int *input;
    int *output;
}

struct cvec_device {
    #define BUF_LEN 1024
    struct {
        cvec_descriptor *cvdesc;
        int conversion_mode;
        int status;
    } buffer[BUF_LEN];
    // TODO: what kind of lock here?
    spinlock_t lock;
    // TODO: done
    wait_queue_head_t wq;
} cvec_dev;

void open(struct inode *inode, struct file *filp) {
    int ret = 0;
    struct cvec_state *state;
    struct cvec_device *cvdev = &cvec_dev;
    if ((ret = nonseekable_open(inode,filp)) < 0) {
        ret = -ENODEV;
        goto out;
    }
    state = kmalloc(sizeof(struct cvec_state), GFP_KERNEL);
    // TODO:                                 structs
    state->conversion_mode = DENSE_TO_SPARSE; // Default
    filp->private_data = state;
    sema_init(&cvec_state->lock,1);
    // TODO: Done

    return ret;
}

void intr(unsigned int intr_mask) {
    struct cvec_device *cvdev = &cvec_dev;
    // TODO: lock?
    spin_lock_irq(&cvdev->lock);
    // TODO: change status to PROCESSED in slot that was recently processed
    int slot = get_processed_slot(cvdev);
    if (slot != -1) {
        cvdev->buffer[slot].status = PROCESSED;
    }
    // TODO: unlock?
  spin_unlock_irq(&cvdev->lock);
    wake_up_interruptable(&cvdev->wq);
}

static ssize_t cvec_ioctl(struct file *filp, unsigned int cmd, unsigned long uarg) {
    struct cvec_device *cvdev = &cvec_dev;
    struct cvec_descriptor *cvdesc;
    // TODO: standard ioctl stuff
  // File descriptor not associated with character special device, or the request does not
    apply
  // to the kind of object the file descriptor references. (ENOTTY)
    int ret = -ENOTTY;
    int retry = -ERESTARTSYS;
  if(_IOC_TYPE(cmd) != CVEC_IOC_MAGIC) return ret;
  if(_IOC_NR(cmd) > CVEC_IOC_MAXNR) return ret;
    // TODO: Done with standard ioctl stuff

    switch(cmd){
        case CONVERT_VECTOR:
            cvdesc = kzalloc(sizeof(*cvdesc), GFP_KERNEL);

            // TODO: case of CONVERT_VECTOR
            // .. initialize structs & copy data from user space
            struct cvec_state *state;
            state = filp->private_data;
            if (down_interruptible(&state->lock)) return retry;
            if (copy_from_user(&cvdesc, (cvec_descriptor __user *) uarg, sizeof(
    cvec_descriptor))) {
                return -EFAULT;
            }

            // .. check if there is a free slot in the buffer otherwise sleep
            int slot = get_free_slot(cvdev);
            if (slot == -1) {
                wait_event_interruptible(cvdev->wq, get_free_slot(cdev));
                slot = get_free_slot(cdev);
            }
            up(&state->lock);
            // .. submit the computation and sleep until it finishes
            spin_lock_irq(&cdev->lock);
            cvdev->buffer[slot].cvdesc = cvdesc;
            cvdev->buffer[slot].status = OCCUPIED;
            spin_unlock_irq(&cdev->lock);
            wait_event_interruptible(cvdev->wq, cvdev->buffer[slot].status == PROCESSED);
            // .. copy data to user space
            if (copy_to_user((char *)cvdesc.output, cvdesc.output, cvdesc.len)) {
                return -EFAULT;
            }
            // .. update the buffer
            break;
        case SET_CONVERSION:
            // TODO: change conversion mode
            struct cvec_state *state = filp->private_data;
            if (down_interruptible(&state->lock)) return retry;
            if (state->conversion_mode == DENSE_TO_SPARSE) {
                state->conversion_mode = SPARSE_TO_DENSE;
            } else {
                state->conversion_mode = DENSE_TO_SPARSE;
            }
            up(&state->lock);
            break;
        default:
            ret = -EINVAL;
            break;
    }

    return ret;
}
```

4 DRIVER K11

```
struct input_data {
    spinlock_t lock;
    wait_queue_head_t wq;
    uint32_t cnt;

#define CIRC_BUF_SIZE (1024 * sizeof(measurement_t))
    char circ_buffer[CIRC_BUF_SIZE];

} input_data;

void intr(void) {
    struct input_data *inp = &input_data;
    spin_lock_irq(&inp->lock);
    memcpy(&inp->circ_buffer[inp->cnt % CIRC_BUF_SIZE],
            device_memory, sizeof(measurement_t));
    //      cnt

    //                          ,

    //

    //                                               CIRC_BUFFER        LRU
    inp->cnt += sizeof(measurement_t);
    spin_unlock_irq(&inp->lock);
    //                 processes                             data
    wake_up_interruptable(&inp->wq);
}

struct chrdev_state {
    struct semaphore lock;
    struct input_data *inp;
    //                                              ;
}

#define wait_event_interruptible(waitqueue, condition)
unsigned long copy_to_user(void __user *dst, const void *src, unsigned long len);

static int chrdev_open(struct inode *inode, struct file *filp) {
    struct chrdev_state *state = kmalloc(sizeof(*state), GFP_KERNEL);
    filp->private_data = state;
    //
    int ret = -ENODEV;

    if ((ret = nonseekable_open(inode,filp)) < 0) {
        kfree(state);
        return ret;
    }
    sema_init(&state->lock,1);
out:
    return 0;
}

static ssize_t chrdev_read(struct file *filp, char __user *usrbuf,
        size_t cnt, loff_t *f_pos) {
    struct chrdev_state *state = filp->private_data;
    struct input_data *inp = state->inp;
    //
    uint32_t bytes_to_copy = sizeof(measurement_t);
    uint32_t available = inp->cnt;

    if (down_interruptible(&state->lock)) {
        return -ERESTARTSYS;
    }

    while(cnt > 0) {
        if (inp->cnt == 0) {
            if (wait_event_interruptible(inp->wq, inp->cnt > 0)) {
                up(&state->lock);
                return -ERESTARTSYS;
            }
        }

        if (copy_to_user(usrbuf, &inp->circ_buffer[(CIRC_BUF_SIZE - available) % CIRC_BUF_SIZE
    ],
                    bytes_to_copy)){
            return -EFAULT;
        }

        *f_pos += bytes_to_copy;
        usrbuf += bytes_to_copy;
        cnt -= bytes_to_copy;
        available -= bytes_to_copy;


    }

    up(&state->lock);
    return 0;
}
```

5 LUNIX

```c
/*
 * lunix-chrdev.h
 *
 * Definition file for the
 * Lunix:TNG character device
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 */

#ifndef _LUNIX_CHRDEV_H
#define _LUNIX_CHRDEV_H

/*
 * Lunix:TNG character device
 */
#define LUNIX_CHRDEV_MAJOR  60  /* Reserved for local / experimental use */
#define LUNIX_CHRDEV_BUFSZ      20      /* Buffer size used to hold textual info */

/* Compile-time parameters */

#ifdef __KERNEL__

#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/module.h>

#include "lunix.h"

/*
 * Private state for an open character device node
 */
struct lunix_chrdev_state_struct {
  enum lunix_msr_enum type;
  struct lunix_sensor_struct *sensor;

  /* A buffer used to hold cached textual info */
  int buf_lim;
  unsigned char buf_data[LUNIX_CHRDEV_BUFSZ];
  uint32_t buf_timestamp;

  struct semaphore lock;

  /*
   * Fixme: Any mode settings? e.g. blocking vs. non-blocking
   */
  int cmd_arg;
};

/*
 * Function prototypes
 */
int lunix_chrdev_init(void);
void lunix_chrdev_destroy(void);

#endif  /* __KERNEL__ */

#include <linux/ioctl.h>

/*
 * Definition of ioctl commands
 */
#define LUNIX_IOC_MAGIC     LUNIX_CHRDEV_MAJOR
#define LUNIX_IOC_EXAMPLE   _IOR(LUNIX_IOC_MAGIC, 0, void *)

#define LUNIX_IOC_MAXNR     0

#endif  /* _LUNIX_H */
```

```c
/*
 * lunix-chrdev.c
 *
 * Implementation of character devices
 * for Lunix:TNG
 *
 * < Your name here >
 *
 */

#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/list.h>
#include <linux/cdev.h>
#include <linux/poll.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/ioctl.h>
#include <linux/types.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mmzone.h>
#include <linux/vmalloc.h>
#include <linux/spinlock.h>

#include "lunix.h"
#include "lunix-chrdev.h"
#include "lunix-lookup.h"

/*
 * Global data
 */
// struct cdev:

struct cdev lunix_chrdev_cdev;

/*
 * Just a quick [unlocked] check to see if the cached
 * chrdev state needs to be updated from sensor measurements.
 */
/*
 * Declare a prototype so we can define the "unused" attribute and keep
 * the compiler happy. This function is not yet used, because this helpcode
 * is a stub.
 */

// State need Refresh
//                              refresh        timestamp

static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
{
  struct lunix_sensor_struct *sensor;

  WARN_ON ( !(sensor = state->sensor));
  /* ? */
  debug("exiting state need refresh");
  return state->buf_timestamp != sensor->msr_data[state->type]->last_update;
}

/*
 * Updates the cached state of a character device
 * based on sensor data. Must be called with the
 * character device state lock held.
 */


// Update State
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
  struct lunix_sensor_struct *sensor = state->sensor;
  long int proper_data;
  uint32_t raw_time = sensor->msr_data[state->type]->last_update;
  uint32_t raw_value = sensor->msr_data[state->type]->values[0];
  char sign;
  debug("entering state update");


  // debug("leaving\n");

  /*
   * Grab the raw data quickly, hold the
   * spinlock for as little as possible.
   */
  /* ? */
  spin_lock_irq(&sensor->lock);
  raw_time = sensor->msr_data[state->type]->last_update;
  raw_value = sensor->msr_data[state->type]->values[0];
  spin_unlock_irq(&sensor->lock);
  /* Why use spinlocks? See LDD3, p. 119 */

  /*
   * Any new data available?
   */
  /* ? */
  if (lunix_chrdev_state_needs_refresh(state)) {
    spin_lock_irq(&sensor->lock);
    raw_time = sensor->msr_data[state->type]->last_update;
    raw_value = sensor->msr_data[state->type]->values[0];
    spin_unlock_irq(&sensor->lock);
  }
  else {
    // -EAGAIN = no data available right now, try again later
    return -EAGAIN;
  }

  /*
   * Now we can take our time to format them,
   * holding only the private state semaphore
   */

  switch(state->type){
    case BATT:
      proper_data = lookup_voltage[raw_value];
      break;
    case TEMP:
      proper_data = lookup_temperature[raw_value];
      break;
    case LIGHT:
      proper_data = lookup_light[raw_value];
      break;
    default:
      return -EAGAIN;
  }

  if (proper_data >= 0) {
    sign='+';
  }
  else {
    sign='-';
    proper_data = (-1)*proper_data;
  }


  state->buf_lim = snprintf(state->buf_data,
      LUNIX_CHRDEV_BUFSZ, "%c%ld,%ld",
      sign, proper_data/1000,
      proper_data%1000);
  state->buf_timestamp = raw_time;

  /* ? */

  debug("leaving\n");
  return 0;
}

/*************************************
 * Implementation of file operations
 * for the Lunix character device
 *************************************/

// Open System Call
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
  /* Declarations */
  /* ? */
  //        lunix-chrdev.h                       struct
  //
  //
  //                                                        struct
  struct lunix_chrdev_state_struct *lunix_chrdev_state;
  int ret;
  int minor = iminor(inode);
  int sensor = minor >> 3;
  int type = minor%8; //??

  debug("entering\n");
  ret = -ENODEV;
  if ((ret = nonseekable_open(inode, filp)) < 0)
    goto out;

  /*
   * Associate this open file with the relevant sensor based on
   * the minor number of the device node [/dev/sensor<NO>-<TYPE>]
   */

  /* Allocate a new Lunix character device private state structure */
  /* ? */
  //                                       allocate

  //             struct state
  lunix_chrdev_state = kzalloc(sizeof(*lunix_chrdev_state), GFP_KERNEL);
  //      GFP_KERNEL flag                            allocation
    process
  //
  //                        struct

  //                              minor
  //      struct                            :
  //          -type {BATT, TEMP, LIGHT, N_LUNIX_MSR}
  //          -sensor,
    sensors
  //                            minor/8.
  //          -buf_lim
  //          -buf_data
  //          -lock
  //          -buf_timestamp
  //          -raw_data

  lunix_chrdev_state->sensor = &lunix_sensors[sensor];
  lunix_chrdev_state->type = type;
  lunix_chrdev_state->buf_lim = 0;
  lunix_chrdev_state->buf_timestamp = 0;
  // struct file: void *private_data;
  // The open system call sets this pointer to NULL before calling the open method for the
    driver.
  // The driver is free to make its own use of the field or to ignore it.
  // The driver can use the field to point to allocated data, but then must free memory
  // in the release method before the file structure is destroyed by the kernel.
  // private_data is a useful resource for preserving state information across system calls
  // and is used by most of our sample modules.
  filp->private_data = lunix_chrdev_state;

  // Initialize lock
  sema_init(&lunix_chrdev_state->lock,1);
out:
  debug("leaving, with ret = %d\n", ret);
  return ret;
}

// Release System Call (free allocated memory of file)
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
  /* ? */
  kfree(filp->private_data);
  return 0;
}

// Device Specific Commands
static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
  // File descriptor not associated with character special device, or the request does not
```

```c
    apply
  // to the kind of object the file descriptor references. (ENOTTY)
  int ret = -ENOTTY;
  int retry = -ERESTARTSYS;
  struct lunix_chrdev_state_struct *state;

  // https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch05.html
  if(_IOC_TYPE(cmd) != LUNIX_IOC_MAGIC) return ret;
  if(_IOC_NR(cmd) > LUNIX_IOC_MAXNR) return ret;
  state = filp->private_data;

  switch(cmd) {
    case LUNIX_IOC_EXAMPLE:
      if(down_interruptible(&state->lock)) return retry;
      debug("in LUNIX_IOC_EXAMPLE area");
      up(&state->lock);
      break;
    default: return ret;
  }
  /* Why? */
  // return -EINVAL; Invalid Argument
  debug("ioctl done my guy ");
  return 0;
}

static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf, size_t cnt, loff_t *
    f_pos)
{
  ssize_t ret = 0;
  int retry = -ERESTARTSYS;

  struct lunix_sensor_struct *sensor;
  struct lunix_chrdev_state_struct *state = filp->private_data;
  WARN_ON(!state);

  sensor = state->sensor;
  WARN_ON(!sensor);

  /* Lock? */
    //         semaphore                                                        process
                          waitqueue
  if(down_interruptible(&state->lock)){
    debug("down interruptable failed in read");
    return retry;
  }
  /*
   * If the cached character device state needs to be
   * updated by actual sensor data (i.e. we need to report
   * on a "fresh" measurement, do so
   */
  if (*f_pos == 0) {
    // Nothing to read
    while (lunix_chrdev_state_update(state) == -EAGAIN) {
      /* ? */
      /* The process needs to sleep */
      /* See LDD3, page 153 for a hint */
      up(&state->lock);
      if (wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state))){
        debug("wait interruptable failed in read");
        return -ERESTARTSYS;
      }
      if (down_interruptible(&state->lock)){
        debug("down interruptable failed in read");
        return -ERESTARTSYS;
      }
    }
  }

  /* End of file */
  /* ? */

  /* Determine the number of cached bytes to copy to userspace */
  /* ? */
  //         user
                        ,
  //
  if (*f_pos + cnt >= state->buf_lim) {
    cnt = state->buf_lim - *f_pos;
  }

  if (copy_to_user(usrbuf, state->buf_data, cnt)) {
    ret=-EFAULT;
    goto out;
  }

  //                                                                              offset
  *f_pos += cnt;
  ret = cnt;


  /* Auto-rewind on EOF mode? */
  /* ? */
  if (*f_pos == state->buf_lim){*f_pos = 0;}

out:
  /* Unlock? */
  up(&state->lock);
  return ret;
}

// Mmap
static int lunix_chrdev_mmap(struct file *filp, struct vm_area_struct *vma)
{
  return -EINVAL;
}

//                                          cdev
static struct file_operations lunix_chrdev_fops =
{
  .owner          = THIS_MODULE,
  .open           = lunix_chrdev_open,
  .release        = lunix_chrdev_release,
  .read           = lunix_chrdev_read,
  .unlocked_ioctl = lunix_chrdev_ioctl,
  .mmap           = lunix_chrdev_mmap
};

// Initialization
int lunix_chrdev_init(void)
{
  /*
   * Register the character device with the kernel, asking for
   * a range of minor numbers (number of sensors * 8 measurements / sensor)
   * beginning with LINUX_CHRDEV_MAJOR:0
   */
  int ret;
  dev_t dev_no;
  unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;

  //                       sensors * 2^3
           minor numbers

  debug("initializing character device\n");
  cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
  //                                              cdev
  lunix_chrdev_cdev.owner = THIS_MODULE;

  dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
  //                      device ID
  /* ? */
  /* register_chrdev_region? */
  //                                                                  device numbers
  //                                                                                  (
                   Major Number),
  //                                           ,
  ret = register_chrdev_region(dev_no, lunix_minor_cnt, "lunix");
  if (ret < 0) {
    debug("failed to register region, ret = %d\n", ret);
    goto out;
  }
  /* ? */
  /* cdev_add? */
  //                                                                   cdev_add
  //                     :          cdev,

  //                            minor numbers.
  ret = cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt);
  if (ret < 0) {
    debug("failed to add character device\n");
    goto out_with_chrdev_region;
  }
  debug("completed successfully\n");
  return 0;

out_with_chrdev_region:
  unregister_chrdev_region(dev_no, lunix_minor_cnt);
out:
  return ret;
}

// Destroy
void lunix_chrdev_destroy(void)
{
  dev_t dev_no;
  unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;

  debug("entering\n");
  dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
  cdev_del(&lunix_chrdev_cdev);
  unregister_chrdev_region(dev_no, lunix_minor_cnt);
  debug("leaving\n");
}
```