Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
7ο εξάμηνο, Ακαδημαϊκή περίοδος 2021 - 2022
http://www.cslab.ece.ntua.gr/

Εργαστήριο Λειτουργικών Συστημάτων

Ομάδα 20

Μέλη Ομάδας: Βρεττός Βασίλειος el18126
Τσάφος Αλέξανδρος el18211

2η Εργαστηριακή Άσκηση: Lunix:TNG

Σκοπός της άσκησης ήταν η υλοποίηση ενός οδηγού συσκευής για ένα ασύρματο δίκτυο αισθητήρων θερμοκρασίας, τάσης και φωτεινότητας. Το κύκλωμα συνδέεται σε έναν σταθμό μέσω Serial over USB και ακούει στην εικονική θύρα /dev/ttyUSB1.
Το σύστημα, γενικά, έχει 2 κύρια μέρη:

α)Λαμβάνονται δεδομένα από τον σταθμό βάσης, προωθούνται μέσω του Serial over USB στο σύστημα, επεξεργάζονται κατάλληλα μέσω ενός φίλτρου και αποθηκεύονται σε κατάλληλο buffer ανά σένσορα.

β)Λαμβάνονται δεδομένα από τους προηγουμένως αναφερόμενους buffer, δέχονται επεξεργασία ώστε τα δεδομένα να είναι σε δεκαδική μορφή (μορφής float) και εξάγονται στον χώρο χρήστη.

Η υλοποίηση του πρώτου μέρους μας δίνεται έτοιμη, δική μας ευθύνη είναι η υλοποίηση του δεύτερου. Το μόνο αρχείο στο οποίο γράψαμε δικό μας κώδικα από τα ήδη έτοιμα ήταν το **lunix-chrdev.c**. Επίσης υλοποιήθηκαν και 2 πολύ απλά test προγράμματα για να δούμε την λειτουργία του οδηγού χωρίς να κάνουμε κάθε φορά cat το αντίστοιχο dev/file. Αργότερα, η ανάκτηση των μετρήσεων από τον χώρο χρήστη θα γίνεται είτε μέσω της κλήσης σύστηματος read() ή απευθείας μέσω απεικόνισης μνήμης μεταξύ πυρήνα και χώρου χρήστη μέσω της κλήσης συστήματος mmap().

Lunix_chrdev_state_needs_refresh:
Έλεγχος για νέα δεδομένα σε κάποιο device. Γίνεται σύγκριση του timestamp της τελευταίας ενημέρωσης του state buffer και της τελευταίας ενημέρωσης των δεδομένων μετρήσεων της συσκευής. Αν είναι ίδια, επιστρέφεται η τιμή 0. Αλλιώς επιστρέφεται η τιμή 1, σηματοδοτώντας την ανάγκη για ανανέωση του state.

```
/*
```

```
 * Just a quick [unlocked] check to see if the cached
 * chrdev state needs to be updated from sensor measurements.
 */
static int lunix_chrdev_state_needs_refresh(struct
lunix_chrdev_state_struct *state)
{
      struct lunix_sensor_struct *sensor;
      int ret;

      debug("state_needs_refresh got called");
      WARN_ON ( !(sensor = state->sensor));
      ret = (sensor->msr_data[state->type]->last_update !=
state->buf_timestamp);
      /* ? */
      return ret;

//state->type is an enum that is the type of the sensor. (lunix-module.c,
lunix.h)
//last_update is the timestamp of the last time the sensor was updated
through lunix_sensor_update (lunix-sensors.c)
//buf_timestamp (lunix-chrdev.h) is the last time the state buffer got
"filled" with new data
//if the two timestamps are different, then, the state needs a refresh, so
we return 1

      /* The following return is bogus, just for the stub to compile */
      //return 0; /* ? */
}
```

Lunix_chrdev_state_update:
Ανανέωση ενός state, αν χρειάζεται. Με τη χρήση spinlocks, καλούμε την lunix_chrdev_state_needs_refresh() και σώζουμε τα καινούρια δεδομένα σε προσωρινές μεταβλητές, με σκοπό να τις επεξεργαστούμε εκτός του spinlock. Ο λόγος είναι πως θέλουμε να περιορίσουμε το μέγεθος του κώδικα που εκτελείται μέσα στα κλειδώματα, προς αποφυγή πιθανών deadlocks. Επίσης, χρησιμοποιούμε τις spin_lock_irqsave() και spin_unlock_irqrestore(), για να αποθηκεύουμε και να ανακτούμε την κατάσταση της διακοπής. Στη συνέχεια, έχοντας εξέλθει από το κρίσιμο σημείο, ανανεώνουμε (ή οχι) το state struct και επιστρέφουμε την τιμή 0 αν έγινε ανανέωση ή την τιμή -EAGAIN αν δεν χρειάστηκε.

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

```
 * character device state lock held.
 */
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct
*state)
{
      struct lunix_sensor_struct *sensor;
      unsigned long state_flags;
      long result, result_dec, *lookup[N_LUNIX_MSR]= {lookup_voltage,
lookup_temperature, lookup_light};
      uint32_t temp_timestamp;
      uint16_t temp_values;
      int refresh, ret;

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

      /*
       * Grab the raw data quickly, hold the
       * spinlock for as little as possible.
       */

      /* Why use spinlocks? See LDD3, p. 119 */
      debug("lunix_chrdev_state_update got called");
      spin_lock_irqsave(&sensor->lock, state_flags);

      //No spinlocks after reading :P
      //Code runs in intterrupt context, We need to disable intterrupts
      //We save the interrupt state. Better be safe than sorry :)

      /*
       * Any new data available?
       */
      /* ? */

      if (refresh = lunix_chrdev_state_needs_refresh(state)) {
            //if yes, store them, so no more race conditions occur (less
spinlocks)
            temp_values = sensor->msr_data[state->type]->values[0];
            temp_timestamp = sensor->msr_data[state->type]->last_update;
      }
      else {
            spin_unlock_irqrestore(&sensor->lock, state_flags);
            debug("state needs refresh: %d\n", refresh);
            ret = -EAGAIN;
```

```
            goto out;
      }

      spin_unlock_irqrestore(&sensor->lock, state_flags);
      debug("state needs refresh: %d\n", refresh);
      /*
       * Now we can take our time to format them,
       * holding only the private state semaphore
       */

       if (refresh) {
             result = lookup[state->type][temp_values];
             result_dec = (result%1000 < 0) ? -result%1000 : result%1000;
             state->buf_timestamp = temp_timestamp;
             //Warning: result is XXYYY but should be XX.YYY
             state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ,
"%ld.%ld\n", result/1000, result_dec);
             debug("Value %ld.%ld of sensor %d printed to state buffer",
result/1000, result_dec, state->type);
       }

       ret = 0;

      /* ? */
      out:
      debug("leaving state update\n");
      return ret;
}
```

Lunix_chrdev_open:
Συσχετισμός αρχείου με private_state struct. Σκοπός της open είναι να συσχετίσει ορθά τα αρχεία που ανοίγουμε με τον σωστό σένσορα. Αυτό γίνεται μέσω των ορισμάτων minor number και inode. Για κάθε συσκευή-αρχείο που ανοίγουμε, δημιουργούμε και ένα private state struct με αρχικές τιμές ανάλογα τον αισθητήρα. Για να γίνει αυτό, δεσμεύουμε μνήμη κάνοντας χρήση την kzalloc().

```
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
      /* Declarations */
      /* ? */
      int ret, minor, sensor_type, sensor_nb;
```

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

   //nonseekable_open = open but for subsystems that do not want seekable
file descriptors.
       //inode represents file on disk

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

       minor = iminor(inode); //get minor number from C function --> inode
gets us the /dev/sensor info

       sensor_type = minor % 8; //
       if (sensor_type >= N_LUNIX_MSR) goto out;

       sensor_nb = minor / 8; //
       debug("inode from /dev/sensor associated");

       //buf_lim = int, buf_data = unsigned char (sizeof(20)),
buf_timestamp = uint32_t
       //lock = struct semaphore
      /* Allocate a new Lunix character device private state structure */
      struct lunix_chrdev_state_struct *p_state;
      //allocate memory on kernel space for p_state struct --> GFP_KERNEL
(observed from lunix-sensors.c)
      p_state = kzalloc(sizeof(struct lunix_chrdev_state_struct),
GFP_KERNEL);
      if (!p_state) {
            ret = -ENOMEM; //out of memory error
            printk(KERN_ERR "Failed to allocate memory for Lunix
sensors\n");
            goto out; //skip private struct initialization
      }
      //initialize p_state struct with values

      p_state->type = sensor_type;
      p_state->sensor = &(lunix_sensors[sensor_nb]);
      p_state->buf_timestamp = get_seconds(); //current timestamp
```

```
	p_state->buf_data[LUNIX_CHRDEV_BUFSZ - 1]='\0'; //initialised
	p_state->buf_lim = strnlen(p_state->buf_data, LUNIX_CHRDEV_BUFSZ);

	//initialize a semaphore with 1 as initial value
	sema_init(&p_state->lock,1);


	filp->private_data = p_state;
	//State struct must be private

	ret = 0; //everything is ok
	debug("State of type %d and sensor %d successfully associated\n",
sensor_type, sensor_nb);

out:
	debug("leaving, with ret = %d\n", ret);
	return ret;
}
```

Lunix_chrdev_release:
Αποδέσμευση μνήμης που αντιστοιχεί στο state struct. Χρησιμοποιείται η συνάρτηση kfree().

```
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
//done!
{
	debug("freed memory via lunix_chrden_release");
	kfree(filp->private_data); //free previous kzalloc memory allocation
- also could have written
	//kfree(p_state);
	return 0;
}
```

Lunix_chrdev_read:
Διάβασμα ενός dev file. Εδώ χρησιμοποιούνται σημαφόροι για το κλείδωμα των κρίσιμων σημείων. Συγκεκριμένα, χρησιμοποιούμε την down_interruptible() για να επιτρέψουμε τις διακοπές από τους sensors. Αν η διεργασία δεν καταφέρει να αποκτήσει πρόσβαση μέσω του σημαφόρου, επιστρέφεται η τιμή -ERESTARTSYS. Αλλιώς τρέχουμε συνεχώς την lunix_chrdev_state_needs_refresh() μέσω της wait_event_interruptible(). Μπαίνουμε σε έναν κύκλο περιμένοντας νέα δεδομένα από αισθητήρα για τον οποίο καλέσαμε την read(). Με το που δούμε νέα δεδομένα, συνεχίζουμε την διαδικασία του διαβάσματος. Στη συνέχεια, με τη χρήση

της συνάρτησης copy_to_user(), αντιγράφουμε στον user buffer τα δεδομένα που βρίσκονται στον state buffer, δίνοντας προσοχή στο offset (f_pos) και στον αριθμό των χαρακτήρων (cnt) που θα αντιγραφούν. Τέλος, ελέγχουμε αν η copy_to_user() εκτελέστηκε επιτυχώς. Αν ναι, ανανεώνουμε τις μεταβλητές f_pos και cnt και επιστρέφουμε την τιμή 0, αφου ξεκλειδώσουμε τον σημαφόρο με την up(). Αν προκύψει κάποιο σφάλμα, τότε επιστρέφουμε την ανάλογη τιμή, αφού πάλι ξεκλειδώσουμε τον σημαφόρο.

```
static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf,
size_t cnt, loff_t *f_pos)
{
	ssize_t ret;
	//ssize_t count_bytes;
	struct lunix_sensor_struct *sensor;
	struct lunix_chrdev_state_struct *state;
	unsigned long check;
	static int update;

	state = filp->private_data;
	WARN_ON(!state);

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

	//down_interruptible allows a user-space process that is waiting on a
semaphore
	//to be interrupted by the user
		/* Lock? */
	debug("locked read");
	if (down_interruptible(&state->lock)){
		return -ERESTARTSYS;
	}

	/*
	 * 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
	 */


	 //file position == 0??
	 //while() code HEAVILY inspired by LDD3 page 153
	if (*f_pos == 0) {
		while (lunix_chrdev_state_update(state) == -EAGAIN) {
```

```
                  /* The process needs to sleep */
                  /* See LDD3, page 153 for a hint */
                  up(&state->lock);
                  update = lunix_chrdev_state_update(state);
                  debug("state updated --> go copy to user");
                  if (wait_event_interruptible(sensor->wq,(update !=
-EAGAIN))){ //needs to be filled
                        return -ERESTARTSYS;
                  }
                  if (down_interruptible(&state->lock)){
                        return -ERESTARTSYS;
                  }
            }
      }

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

      /* Determine the number of cached bytes to copy to userspace */
      /* ? */
      cnt = ((state->buf_lim - *f_pos) <= cnt) ? (state->buf_lim - *f_pos)
: cnt;

      check = copy_to_user(usrbuf, (state->buf_data + *f_pos) ,cnt);
      //if number of bytes that could not be copied > 0 --> copy_to_user
      //basically failed
      debug("copy to user successful");
      if (check > 0){
            ret = -EFAULT;
            goto out;
      }

      *f_pos += cnt;
      ret = cnt;

      /* Auto-rewind on EOF mode? */
      if (*f_pos == state->buf_lim){
            *f_pos = 0; //return file_position to 0 if EOF has been reached
      }


out:
      up(&state->lock); //unlock on out. NOTE:copy_to_user CAN sleep but
```

```
will not bring
	//us in a deadlock state.
	debug("read complete with ret returnd: %zu", ret);
	return ret;
}
```

Lunix_chrdev_init:
Εγγραφή των character devices στο kernel με χρήση του οδηγού για τα Minor και Major numbers. Με τις συναρτήσεις cdev_init(), register_chrdev_region() και cdev_add() ζητάει από το kernel περιοχή από minor number και τοποθετεί την αντίστοιχη συσκευή εκεί. Αν αποτύχει η πρόσθεση, καλείται η unregister_chrdev_region() για να αφαιρεθεί η λάθος αρχικοποιημένη character device από τον πυρήνα. Μαζί με την init, υπάρχει και η destroy η οποία κάνει την διαγραφή και αποδέσμευση των συσκευών χαρακτήρων με τον ακριβώς αντίστροφο τρόπο. (cdev_del(), unregister_chrdev_region()).

```
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;

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

	dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
	ret = register_chrdev_region(dev_no, lunix_minor_cnt, "lunix");
	// returns a negative error value on failure
	if (ret < 0) {
		debug("failed to register region, ret = %d\n", ret);
		goto out;
	}

	ret = cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt);
	// returns a negative error value on failure
	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;
}
```

Lunix_chrdev_mmap:
Δημιουργία εικονικού χώρου μνήμης από kernelspace σε userspace. Πριν υλοποιήσουμε την mmap(), υλοποιήσαμε 2 υπερ-απλουστευμένες συναρτήσεις για άνοιγμα και κλείσιμο Virtual Memory σελίδων. Αυτές είναι οι lunix_chrdev_vma_open και η lunix_chrdev_vma_close. Προσθέτουμε αυτές τις συναρτήσεις στο struct vm_operations_struct. Γενικά αυτό το βήμα είναι προεραιτικό καθώς δεν αξιοποιούνται (τουλαχιστον η close δεν αξιοποιείται). Όπως και να έχει, γραφτήκαν καθώς είχαμε οδηγό το βιβλίο LDD3.

Η mmap() ξεκινάει φτιάχνοντας έναν pointer για το Virtual Address που υπάρχουν τα δεδομένα που λάβαμε από τον σένσορα. Έπειτα, μέσω της page_address, επιστρέφεται η εικονική διεύθυνση αυτής της σελίδας. Μετατρέπουμε την εικονική διεύθυνση σε φυσική μέσω της __pa() και κάνοντας right shift όσο έχουμε θέσει στο PAGE_SHIFT.

Τέλος κάνουμε remap την συγκεκριμένη kernel memory στο userspace μέσω της remap_pfn_range(). Εδώ σημειώνουμε ότι καθώς έχουμε ορίσει το page size να είναι vma->vm_end - vma->vm_start, το μέγεθος μνήμης που μπορεί να μεταφέρει η συγκεκριμένη υλοποίηση της mmap() είναι πάντα μόνο 1 σελίδα. Με εξυπνότερη χρήση της remap_pfn_range() και έλεγχο, θα μπορούσε να επεκταθεί για να αξιοποιούνται παραπάνω σελίδες, αλλά για την συγκεκριμένη άσκηση, δεν χρειάζεται τίποτα περισσότερο.

```
//Page 422 of FDD3
void lunix_chrdev_vma_open(struct vm_area_struct *vma)
{
      printk(KERN_NOTICE "Simple VMA open, virt %lx, phys %lx\n",
vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void lunix_chrdev_vma_close(struct vm_area_struct *vma)
{
      printk(KERN_NOTICE "Simple VMA close.\n");
```

```
}

static struct vm_operations_struct lunix_chrdev_vm_ops = {
      .open = lunix_chrdev_vma_open,
      .close = lunix_chrdev_vma_close,
};

//NOTE: this function only returns ONE page, the specific application does not need more, it can be imporved
static int lunix_chrdev_mmap(struct file *filp, struct vm_area_struct *vma)
{
      struct lunix_chrdev_state_struct *state;
      struct lunix_sensor_struct *sensor;

      unsigned long *kmap_return;
      struct page *kernel_page;
      state = filp->private_data;

      sensor = state->sensor;

      //pointer of VA's page from values receieved
      kernel_page = virt_to_page(sensor->msr_data[state->type]->values);
      //VA of page with values receieved
      kmap_return = page_address(kernel_page);
      //convert VA to Physical Address
      vma->vm_pgoff = __pa(kmap_return) >> PAGE_SHIFT;

      //map device memory to user address space -- page size is vm_end - vm_start = 1 Page.
      //function is safe if mm semaphore is HELD.
      if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,vma->vm_end - vma->vm_start, vma->vm_page_prot)){
            return -EAGAIN;
      }
      //link to struct and use vma_open
      vma->vm_ops = &lunix_chrdev_vm_ops;
      lunix_chrdev_vma_open(vma);
      //return 0 on success
      return 0;
}
```