Device Driver Assignment

John Abendroth, Ryan Tucker, Austin Seyboldt

Goals

The goal of this program is to create a FreeBSD device driver that interfaces with the GPIO pins to send and receive strings inputted by the user. We included extensive error checking to ensure that kernel panics do not occur.

Notes on our implementation

We decided to build our driver on top of gpiobus which provides a simple interface for setting up interrupts, aquiring pins, and setting the values of pins. We did this by registering our driver as a child of gpiobus which is done in the DRIVER_MODULE macro.

Writes

For writing, we utilize buf_rings and taksqueues to implement I/O queueing. Whenever we receive a write() call from the user, we copy their buffer into kernel space using uio_move(). We do not use a shared buffer for all write calls. Instead, we malloc a buffer of appropriate size whenever a user calls write(). We then add this buffer to the buf_ring and add a task to the global taskqueue called taskqueue_swi. The function we register with the task is called write_to_laser() which pulls a buffer from the buf_ring and writes it out to the LED. With this approach, we are able to immediately return to the user once they make a write call while.

Our algorithm for flashing the LED is fairly simple. For each character, we reference a huffman dictionary array which contains conversion data. For each huffman bit in a character, we turn on the LED and sleep for a certain amount of milliseconds based on whether it is a 0 or a 1. Then we turn off the LED and sleep for x_time which defaults to 20ms. The locks we are using around the write loop allow us to sleep so it seemed like the simplest implementation.

Reads

For reading, we set up interrupts for the read pin and have an interrupt generated whenever the pin changes value (e.g. from low to high and from high to low). We process this interrupt first from an interrupt filter where we record the current time and get the value from the pin. We then schedule an ithread to run which does more processing such as calculate the interval, descend the huffman tree, and possibly copy the character into the read buffer. We opted to overwrite existing read buffer data if it is full, however this is an arbitrary design decision and could be easily changed. The use of an ithread was desirable to us as it allows us to grab a mutex which protects the read buffer.

Non-Blocking Behavior

We implemented non-blocking reads according to what was mentioned in section early on, which is slightly different than mentioned in piazza post https://piazza.com/class/km2ijrdlqlm18r?cid=70. Here's how the driver handles non-blocking reads.

1. If we get a read request and the read buffer is empty, by default (without flag `O_NONBLOCK`) the driver will block until the requested amount is available in full, instead of returning the first available data even if it is smaller than requested. Thus, on a default blocking read call, the driver blocks until the exact requested amount is available to return.

2. If the flag `O_NONBLOCK` is given to the read function, it will return `EAGAIN` if the full data amount requested isn't available, instead of returning data if there is any available at all. Thus, if `O_NONBLOCK` is supplied it only returns data if the read buffer has at least the requested amount available.

3. If we get a read request for more data than currently available in the read buffer, the driver blocks until the requested amount is available, up until the complete read buffer size. Thus, it only returns when the full requested amount is ready instead of whatever is available.

IOCTL

Designing around the IOCTL functions introduced some synchronization issues that we solved with locks. For example, the IOCTL function SET_DICTIONARY is protected by two locks: a write_pin_lock and a huffman_lock. The write_pin_lock is locked when flashing a message on the LED which ensures that only one message is being displayed at once, that the output pin cannot be changed mid message, and that the huffman conversion cannot be changed mid message. The huffman_lock is locked when decoding a single flash, e.g. when we are performing a single node traversal of the huffman tree. This ensures that the interrupt handler does not end up with an invalid pointer to a huffman node that does not exist. However, it is generally a bad idea to set a new dictionary while the program is reading because it is entirely possible that a new tree could be set before the huffman traversal has hit a leaf node, in which case all subsequent reads will be messed up.

Additional notes

- The minimum value for x_time is 20ms. This is because going any lower can cause interrupts to be incorrectly recorded and thus reads fail to work properly.

Data Structures

struct for storing all data about a given read/write pair

```
struct cdev* led_cdev;
    device_t dev;        // this device
    device_t bus_dev;    // the bus
    gpio_pin_t read_pin;
    gpio_pin_t write_pin;
    struct buf_ring* br;      // holds write buffers
    struct task tsk;
    bool read_busy;
    int read_request_size;
    char* read_buf;
    int read_buf_length;
    int read_buf_index;    // Where to start reading from circular read buffer
    int write_index;       // Where to start copying into circular read buffer
    struct mtx buf_ring_mtx;
    struct mtx read_buf_mtx;
    struct mtx read_busy_lock;
    struct cv read_is_free;
    struct cv read_buf_ready;
    struct sx write_pin_lock;    // used to synchronize updating of pin with writer
    struct sx read_pin_lock;
    struct node* huffman_root;
    struct node* huffman_current;
    uint64_t x_time;
    uint64_t* write_dict;    // array for storing write dictionary
```

Holds data for interrupts.

```
struct intr_pin_value {
    time_t sec;
    time_t recent_sec;
    suseconds_t micro_sec;
    suseconds_t recent_micro_sec;
    int led_value;
    int recent_value;
};
```

Struct to hold resources for interrupt handler.
Holds references to the interrupt handler resources and read pin needed during interrupts.

```
struct led_pin_intr {
    struct led_softc* sc;
    gpio_pin_t read_pin;
    int intr_rid;
    struct resource* intr_res;
    void* intr_cookie;
    struct intr_pin_value* led;
};
```

Array to store Huffman Dictionary Information

```
static uint64_t dict[DICT_SIZE];
```

Character device switch to register our functions.

```
static struct cdevsw led_cdevsw = {
    .d_version = D_VERSION,
    .d_open    = led_open,
    .d_close   = led_close,
    .d_read    = led_read,
    .d_write   = led_write,
    .d_ioctl   = led_ioctl,
    .d_name    = DEVICE_NAME,
};
```

Huffman Tree Implementation

```
struct node {
    struct node* left;
    struct node* right;
    char data;
};
```

Driver Setup and Autoconfiguration

```
gpio_led_identify(driver_t* driver, device_t parent )
    device_t child;
    child = device_find_child(parent, DEVICE_NAME, -1);
    if (!child) {
        child = BUS_ADD_CHILD(parent, 0, DEVICE_NAME, -1);
    }
```

```
gpio_led_probe(device_t dev)
    device_set_desc(dev, "GPIO led printer");
    return (BUS_PROBE_DEFAULT);
```

```
gpio_led_attach(device_t dev)
    get read and write pins
    set flags for input and output on pins
    setup_intr(sc)
    initialize mutexes, read buffers, buf_ring, and task
    construct initial huffman tree based on default encoding
    make_dev_s()
```

```
gpio_led_detach(device_t dev)
    free all buffers we malloc'd
    destroy all mutexes
    release read and write pins
    tear down interrupts
    release interrupt resource
    destroy_dev_sched(sc->led_cdev)
```

Write Functionality

Interface between user process and device driver.

```
d_write():
    copy data into kernel buffer using uio_move
    acquire buf_ring mutex
    enqueue kernel buffer into buf_ring
    enqueue task to system taksqueue (taskqueue_swi)
    release buf_ring mutex
```

Writes out each character one-by-one.

```
write_to_laser(buffer, len):
    lock buf_ring mutex
    dequeue write_buf from buf_ring
    unlock buf_ring mutex
    lock write_pin_lock
    For each char in write_buf:
        write_char(char)
    unlock write_pin_lock
    free write_buf
```

Writes a single character to the laser.
Note for timing: we sleep for 5 MS longer than specified
in order to account for some variance in interrupt timing.

```
write_char(char):
    encoding = dict[(int)c];
    length   = (encoding >> 8) & 0xFF;
    code     = encoding >> 17;
    for i = (length - 1) to 0:
        bit = (code >> i) & 1;
        if (bit == 0)
            set write_pin to 1
            sleep for (x_time / 2) + 5
            set write_pin to 0
        else
            set write_pin to 1
            sleep for (x_time * 3) + 5
            set write_pin to 0
```

Read Functionality

Interrupt handler is triggered when the read pin changes state, with the interrupt installed to fire for any edge change. First the filter handler executes, doing the smallest amount neccessary by storing the current time and the read pin logical state. This is so that the interrupt isn't too disruptive to the overall driver or any user programs. Then the ithread handler is scheduled to complete the interrupt handling. There it calculates the interval between the current interrupt and the previous, and processess the interval as a bit in the huffman tree. This is done by `decode_flash()`, which actually traverses the huffman tree, and if it reaches a leaf in tree places the decoded chracter into the read buffer and signals there's data to be read.

```
filter_handler():
    interrupt_info_struct->sec <- get_system_uptime()  // record number of miliseconds the system has been up

    bool pin_status <- get_readpin_value()
    interrupt_info_struct->led_value <- pin_status

    return(FILTER_SCHEDULE_THREAD) // schedule ithread handler
```

```
ithread_handler():
    if first interrupt fired:
        // move current interrupt interval time and pin value to recent fields and wait for next interrupt
        interrupt_info_struct->recent_sec <- interrupt_info_struct->sec

        interrupt_info_struct->recent_led_value <- interrupt_info_struct->led_value

    this interrupt ms <- (micro_sec / 1000) + (sec * 1000)
    previous interrupt ms <- (previous micro_sec / 1000) + (previous sec * 1000)
    interval <- this interrupt ms - previous interrupt ms

    if led_value is 0:
        // falling edge, just finished dot/dash
        if interval between x_time / 2 and x_time * 1.5:
            huffman_bit <- 0
        else if interval greather than x_time * 3:
            huffman_bit <- 1
        decode_flash(huffman_bit)
    else if led_value is 1:
        // nothing to do since led just turned on
        if interval is less than the minimum x_time:
            print warning that the interrupt happened too quick for some reason, and decoding may be off

    // store all current interrupt values in recent fields for next interrupt to access and utilize
    interrupt_info_struct->recent_sec <- interrupt_info_struct->sec
    interrupt_info_struct->recent_led_value <- interrupt_info_struct->led_value

    return from thread handler
```

Traverse down huffman tree based on given node and bit value

```
traverse_tree(struct node* nd, val):
    If (val == 0):
        return node_move_left(nd)
    else if (val == 1):
        return node_move_right(nd)
    else:
        return NULL
```

```
decode_flash(led_softc* sc, huffman_bit):
    mtx_lock(huffmann tree lock)
    // get pointer to current pos in huffman tree base on recent led value
    sc->huffman_current <- traverse_tree(sc->huffman_current, huffman_bit)
    if sc->huffman_current is NULL:
        // move current huffman tree pointer back to root
        sc->huffman_current = sc->huffman_root
        mtx_unlock(sc->huffman_lock)
    else if is_huffman_leaf(sc->huffman_current):
        character <- get_huffman_node_char(sc->huffman_current)
        mtx_unlock(huffman_lock)

        mtx_lock(sc->read_buf_mtx)
        index_to_place <- (read_buf_index + read_buf+len) % BUFFER_LEN
        read_buf[index_to_place] = character

        if read_buf_length >= read_request+size:
            cv_signal(sc->read_buf_ready)
        mtx_unlock(sc->read_buf_mtx)
    else:
        mtx_unlock(sc->huffman_lock)
```

```
write_char_to_buffer(char to_write):
    // Not sure which lock we are using specifically, but some lock or mutex to control access
    //  to the read buffer
    mtx.aquire()
    //Make sure there's room left in the buffer to write
    If ((write_index == read_index)
        //buffer is full, throw input out / return error to user?
    Read_buffer[write_index] = to_write;
    Write_index = (write_index + 1) % BUFFER_LEN
    mtx.release()
```

IOCTL functionality

```
led_ioctl(struct cdev* dev, u_long cmd, caddr_t data, int flag, struct thread* td) {
    struct led_softc* sc = dev->si_drv1;

    switch (cmd) {
        case GET_DICTIONARY:
            // getting a dictionary: just copy it back to the user

            user_data = (led_ioctl_data*)data
            for i = 0 to DICT_SIZE:
                user_data[i] = dict[i]

        case SET_DICTIONARY:
            // setting a dictionary: need to save the array,
            // free the old huffman tree, build a new one

            acquire write lock and huffman lock

            populate_write_dict(data->dict)
            node* new_huff_tree = construct_tree(dict, DICT_SIZE)
            destroy the old huffman tree
            huffman_root    = new_huff_tree
            huffman_current = huffman_root

            release write lock and huffman lock

        case SET_OUTPUT_PIN:
            // need to attempt to get new pin and only
            // release old one once we have it

            acquire the output pin lock
            int err = 0;

            new_pin = data;
            if new_pin < 0 or greater than 40:
                return invalid

            gpio_pin_t temp_pin = write_pin;
            try to acquire the new pin
            if it fails, error out, release lock

            try to configure the pin
            if it fails, error out, release lock, reset the input pin to temp_pin

            release temp_pin

            release the output pin lock

        case SET_INPUT_PIN:
            // need to acquire the new pin, release the old one,
            // deallocate the old interrupts, allocate new ones

            new_pin = data
            if new_pin == current pin:
                return
            if new_pin is less than 0 or more than 40:
                return error

            gpio_pin_t temp_pin = sc->read_pin

            try to acquire and configure the new pin for output
            if it fails, release any resources and error out
            else if it was succesful, deallocate interrupt on the old pin

            release the temp pin

            try to allocate the interrupt for new_pin and return failure if it doesn't work

        }
        case SET_MIN_TIME:
            // just set the time if its valid
            int new_time = *(int*)data
            if new_time < 20:
                return invalid

            acquire write lock
            sc->x_time = (uint64_t)new_time
            release write lock

        default:
            break
}
```