SSE3052: Embedded Systems Practice

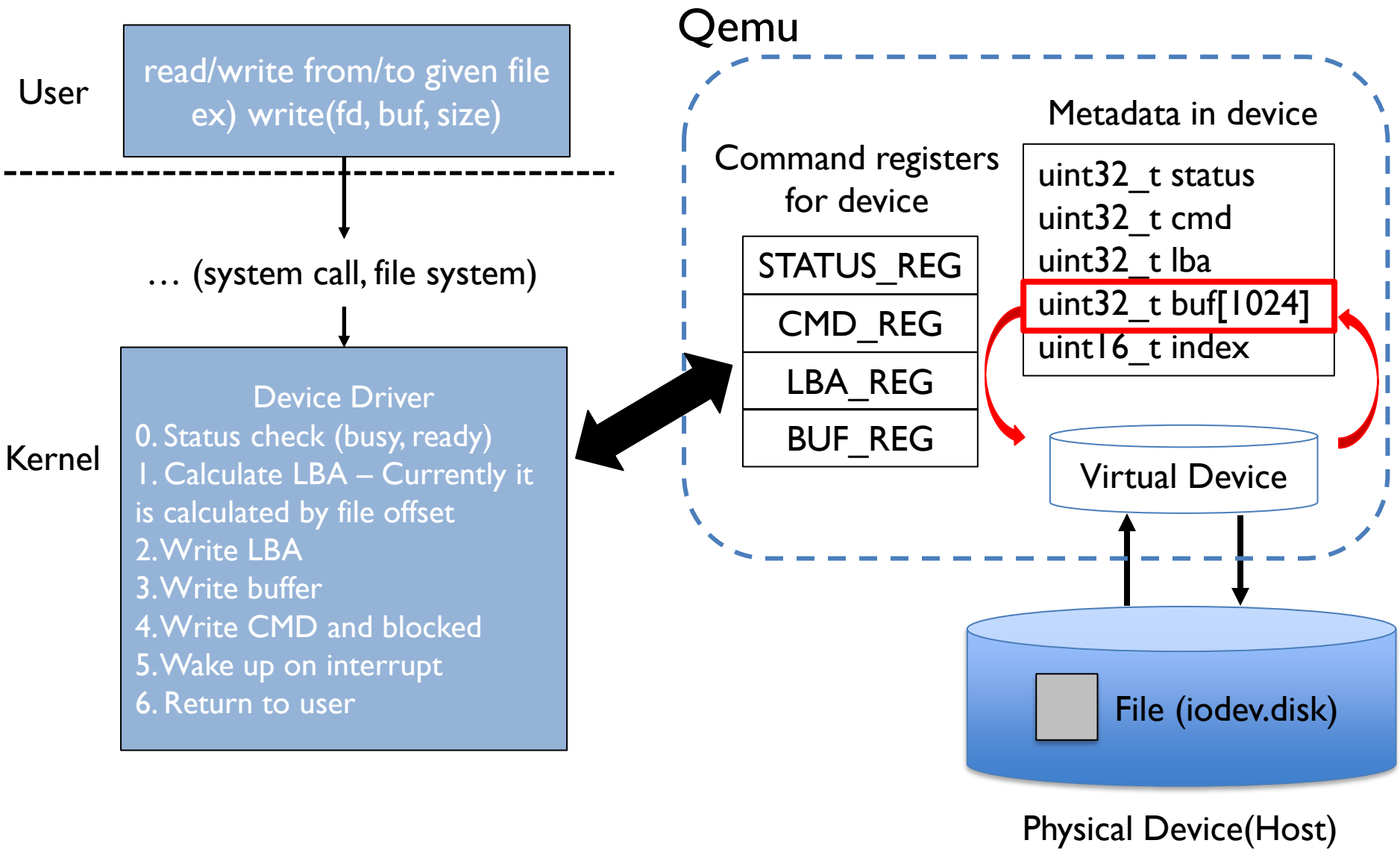
Jinkyu Jeong
jinkyu@skku.edu
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Today's goal

- Add new soft block device (no skin)

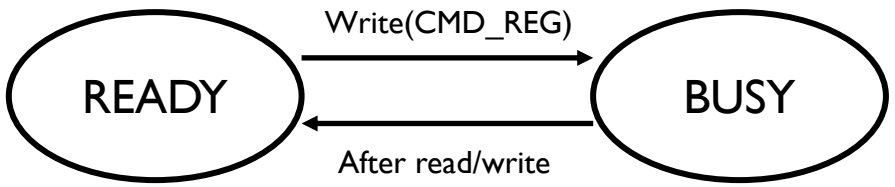
- Implement interrupt based IO

Overview


User
read/write from/to given file
ex) write(fd, buf, size)
… (system call, file system)
Kernel
Device Driver
0. Status check (busy, ready)
1. Calculate LBA – Currently it is calculated by file offset
2. Write LBA
3. Write buffer
4. Write CMD and blocked
5. Wake up on interrupt
6. Return to user
Qemu
Command registers for device
STATUS_REG
CMD_REG
LBA_REG
BUF_REG
Metadata in device
uint32_t status
uint32_t cmd
uint32_t lba
uint32_t buf[1024]
uint16_t index
Virtual Device
File (iodev.disk)
Physical Device(Host)

The New Soft Device

- 4KB read/write block device
- Four device registers
  - STATUS_REG (read-only)
    - DEV_BUSY: device busy
    - DEV_READY: ready to receive a new command
  - CMD_REG (write-only)
    - CMD_READ / CMD_WRITE
  - LBA_REG (write-only)
    - Logical block number in 4KB
  - BUF_REG (read/write): 4KB buffer
    - 1024 * 4B writes fill in the buffer
    - 1024 * 4B reads fetch data from the buffer



Protocols

- Description of interface btw. goldfish and device
- Following two slides are protocols of read/write
- Template code of QEMU and device driver are op ened on iCampus

Write Protocol

1. Buffer copy from user
2. Status check
   - Busy-wait for device's state become DEV_READY
3. Write LBA
4. Fill in the device buffer
   – 1024 * 4B writes to BUF_REG
5. Write command (CMD_WRITE)
6. Wait for interrupt

Read Protocol

1. Status check
   - Busy-wait for device's state become DEV_READY
2. Write LBA
3. Write command (CMD_READ)
4. Wait for interrupt
5. Read device buffer
   – 1024 * 4B reads to BUF_REG
6. Buffer copy to user

QEMU Protocol

- STATUS_REG (read-only)
  - Return current state of device (CMD_READY, CMD_BUSY)
- CMD_REG (write-only)
  - Switch current state to CMD_BUSY
  - Control written operation (READ_CMD, WRITE_CMD)
    - Value of lba is used
  - Raise interrupt to host
- LBA_REG (write-only)
  - Assign written value to lba
- BUF_REG (read/write)
  - Write: assign written value to device buffer
  - Read: return value of device buffer

Modify your QEMU

- Refer to "qemu_devices_with_led.patch" (From resources on week 4)
  - Make new file for emulate new soft device
    - e.g., hw/android/goldfish/iodev.c
  - Edit Makefile.qemu1-target.mk
    - To compile hw/android/goldfish/iodev.c
  - Call initialization of new device in hw/i386/pc.c
    - goldfish_iodev_init(void)
  - Include initialization in include/hw/android/goldfish/device.h
    - void goldfish_iodev_init(void)

Modify your Goldfish

- Refer to "goldfish_kernel_with_led.patch" (From resources on week 4)
  - Add kernel configuration for new soft device
    - drivers/misc/Kconfig
  - Edit Makefile
    - drivers/misc/Makefile
  - Write device driver for new soft device
    - e.g., drivers/misc/goldfish_iodev.c

(Goldfish) goldfish_iodev_probe (1)

```
static int goldfish_iodev_probe(struct platform_device *pdev)
{
    int ret;
    struct resource *r;
    int error;

    printk(KERN_ERR "iodev probe started\n");
    data = devm_kzalloc(&pdev->dev, sizeof(*data), GFP_KERNEL);
    if (data == NULL)
        return -ENOMEM;

    r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (r == NULL) {
        dev_err(&pdev->dev, "platform_get_resource failed\n");
        return -ENODEV;
    }

    if(request_mem_region(r->start, resource_size(r), "7iodev")==NULL){
        printk(KERN_INFO "register 7iodev fail\n");
        return -EBUSY;
    }

    misc_register(&iodev_dev);

    data->reg_base = devm_ioremap(&pdev->dev, r->start, resource_size(r));
    if (data->reg_base == NULL) {
        dev_err(&pdev->dev, "unable to remap MMIO\n");
        return -ENOMEM;
    }

    error = devm_request_irq(&pdev->dev, platform_get_irq(pdev, 0), goldfish_iodev_handler, 0, "goldfish_iodev", NULL);
    init_waitqueue_head(&wait_q);
    spin_lock_init(&wait_q_lock);
    condition = 0;
    printk(KERN_ERR "iodev probe finished\n");

    return 0;
}
```

(Goldfish) goldfish_iodev_probe (2)

```
static const struct of_device_id goldfish_iodev_of_match[] = {
    { .compatible = "generic,goldfish_iodev", },
    {},
};
MODULE_DEVICE_TABLE(of, goldfish_iodev_of_match);

static const struct acpi_device_id goldfish_iodev_acpi_match[] = {
    { "GFSH0001", 0 },
    { },
};
MODULE_DEVICE_TABLE(acpi, goldfish_iodev_acpi_match);


static struct platform_driver goldfish_iodev_device = {
    .probe      = goldfish_iodev_probe,
    .remove     = goldfish_iodev_remove,
    .driver = {
        .name = "goldfish_iodev",
        .of_match_table = goldfish_iodev_of_match,
        .acpi_match_table = ACPI_PTR(goldfish_iodev_acpi_match),
    }
};
module_platform_driver(goldfish_iodev_device);
```

(Goldfish) Add New Interrupt

- We should call below function inside device_probe()
  - static inline int devm_request_irq(struct device *dev, \
    unsigned int irq, \
    irq_handler_t handler, \
    unsigned long irqflags, \
    const char *devname, \
    void *dev_id)

*Example of calling devm_request_irq()

```
data->reg_base = devm_ioremap(&pdev->dev, r->start, resource_size(r));
if (data->reg_base == NULL) {
    dev_err(&pdev->dev, "unable to remap MMIO\n");
    return -ENOMEM;
}

error = devm_request_irq(&pdev->dev, platform_get_irq(pdev, 0), goldfish_iodev_handler, 0, "goldfish_iodev", NULL);
```

(Goldfish) Interrupt Handler

```
static irqreturn_t goldfish_iodev_handler(int irq, void *dev_id)
{
    spin_lock(&wait_q_lock);
    condition = 1;
    wake_up(&wait_q);
    spin_unlock(&wait_q_lock);
    return IRQ_HANDLED;
}
```

(Goldfish) Edit File Operations

- Exercise 1) You should fill the lines of 2, 3, 4 according to the comments

```
static ssize_t iodev_write (struct file *file, const char __user *buf, size_t size, loff_t *loff) {
    int i;
    uint32_t lpn;

    if (*loff & (PAGE_SIZE - 1) || size != PAGE_SIZE )
        return -EINVAL;

    lpn = (uint32_t)(*loff >> PAGE_SHIFT);
    copy_from_user((char*)kbuf, buf, size);

    /*(iodev_write)
        1. Read status register (until not DEV_READY)
        2. Write lpn to LBA_REG
        3. Write kbuf repeatedly (size of 4 bytes)f to BUF_REG
        4. Write WRITE_CMD to CMD_REG
        5. Wait for condition variable (condition)
        6. Increase offset
        7. Return size
     */
    while (readl(data->reg_base + STATUS_REG) != DEV_READY);    //1

    spin_lock_irq(&wait_q_lock);
    condition = 0;

    //2

    for (i = 0; i < PAGE_SIZE / sizeof(uint32_t); i++ ) {
        //3
    }

    //4

    wait_event_lock_irq(wait_q, condition, wait_q_lock);    //5
    spin_unlock_irq(&wait_q_lock);

    (*loff) += size;    //6

    return size;    //7
}
```

(QEMU) Add New Interrupt

- We should set irq count to 1 and irq number to 15

```
void goldfish_iodev_init(void)
{
    struct goldfish_iodev_state *s;

    s = (struct goldfish_iodev_state *)g_malloc0(sizeof(*s));
    s->dev.name = "goldfish_iodev";
    s->dev.base = 0;
    s->dev.size = 0x1000;
    s->dev.irq_count = 1;
    s->dev.irq = 15;
```

Only IRQ number 15 is left for new interrupt

(QEMU) goldfish_iodev_init

```
void goldfish_iodev_init(void)
{
    struct goldfish_iodev_state *s;

    s = (struct goldfish_iodev_state *)g_malloc0(sizeof(*s));
    s->dev.name = "goldfish_iodev";
    s->dev.base = 0;
    s->dev.size = 0x1000;
    s->dev.irq_count = 1;
    s->dev.irq = 15;
    s->iodev_fd = open("./iodev.disk", O_RDWR);

    goldfish_device_add(&s->dev, goldfish_iodev_readfn, goldfish_iodev_writefn, s);


    register_savevm(NULL,
            "goldfish_iodev",
            0,
            IODEV_STATE_SAVE_VERSION,
            goldfish_iodev_save,
            goldfish_iodev_load,
            s);
}
```

(QEMU) goldfish_iodev_write

```
static void goldfish_iodev_write(void* opaque, hwaddr offset, uint32_t value)
{
    struct goldfish_iodev_state* s = (struct goldfish_iodev_state*)opaque;
    int status;

    if ( offset < 0 ) {
        cpu_abort(cpu_single_env, "iodev_dev_read: Bad offset %" HWADDR_PRIx "\n", offset);
        return;
    }

    switch (offset) {
        case IODEV_CMD_REG:
            s->status = IODEV_BUSY;
            s->cmd = value;
            switch(s->cmd) {
                case READ_CMD:
                    status = goldfish_iodev_data_read();
                    break;
                case WRITE_CMD:
                    status = goldfish_iodev_data_write((void *)opaque);
                    break;
                default:
                    cpu_abort(cpu_single_env, "iodev_cmd: unsupported command %d\n", s->cmd);
                    return;
            }
            if ( status == 0 ) {
                s->status = IODEV_READY;
                goldfish_device_set_irq(&s->dev, 0, 1);
            } else {
                cpu_abort(cpu_single_env, "iodev_cmd: command:%d failed %d\n", s->cmd);
                return;
            }
            break;
        case IODEV_LBA_REG:
            s->lba = value;
            break;
        case IODEV_BUF_REG:
            s->iodev_buf[s->iodev_buf_pos] = value;
            s->iodev_buf_pos = (s->iodev_buf_pos+1) % 1024;
            break;
    };
}
```

(QEMU) goldfish_iodev_read

```
static uint32_t goldfish_iodev_read(void* opaque, hwaddr offset)
{
    struct goldfish_iodev_state* s = (struct goldfish_iodev_state*)opaque;

    if ( offset < 0 ) {
        cpu_abort(cpu_single_env, "iodev_dev_read: Bad offset %" HWADDR_PRIx "\n", offset);
        return 0;
    }

    switch (offset) {
        case IODEV_STATUS_REG:
            return s->status;
    };

    return 0;
}
```

Exercise

- Write Goldfish and QEMU code to "iodev" can emulate read and write requests from user
  - You can test your softdevice with 4k_write.c from iCampus