Linux Device Driver

Prof. Yan Luo

For UMass Lowell 16.480/552

Outline

- Overview
- Device driver example
- Polling vs interrupt
- Lab 3

Linux and device drivers

- Linux, an open OS
  - Open source
  - Modular
  - Extensible
- Device driver
  - Black boxes that hide details of a piece of hardware
  - Provide well defined programming interfaces to others
  - Plugged in as needed
  - Necessary for new hardware
  - Writing a good device driver is art ;)

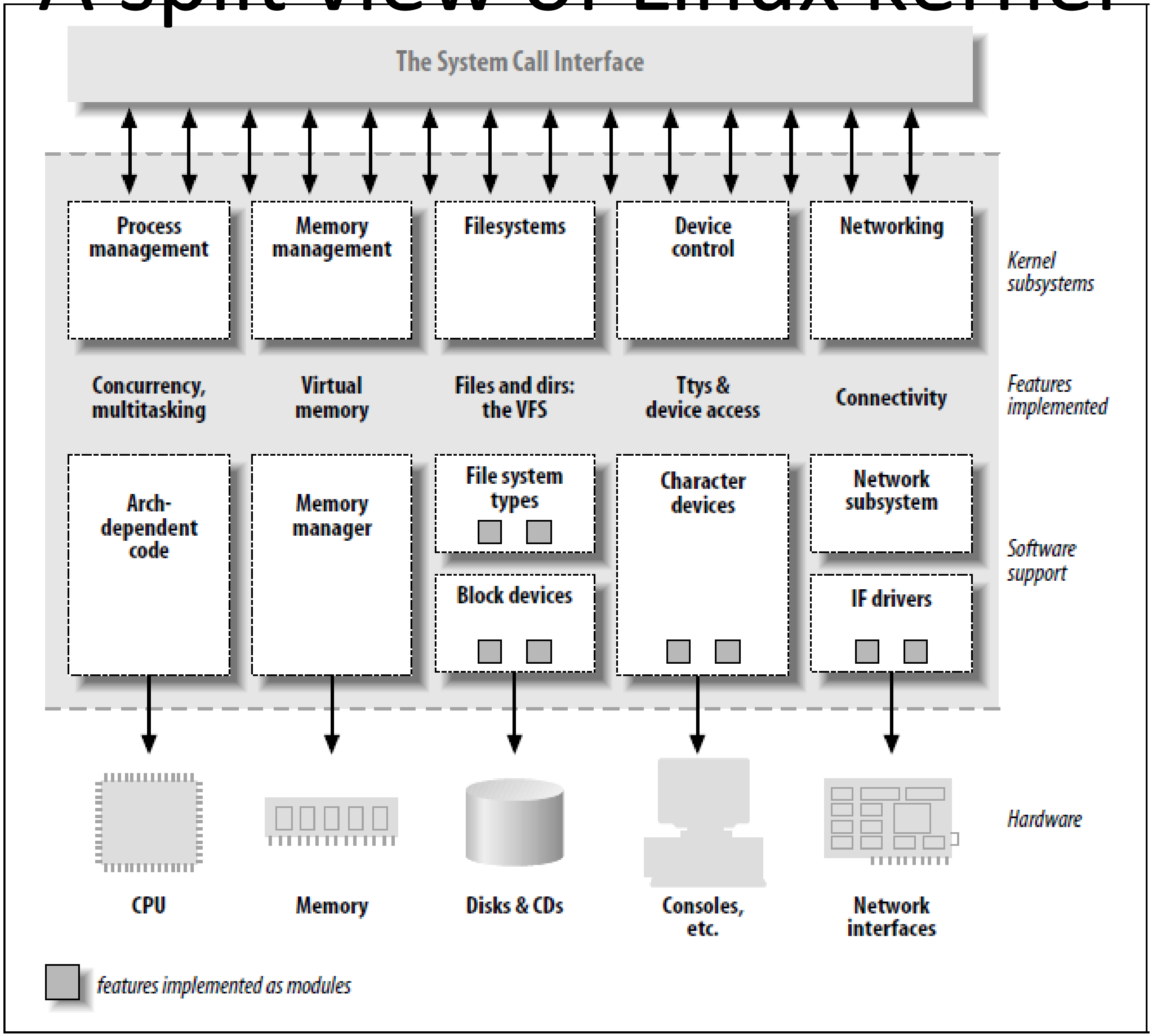
The Role of Device Driver

- Device driver is a layer between application and actual device
- Providing mechanism, NOT policy
- Example: management of graphic display
  - X server: knows the h/w, and offers programming interfaces to user
  - Window/session manager: implements a policy without the need of knowing about the h/w
  - So, users can use the same window session manager on different hardware, or different window/session on the same hardware.

Characteristics of device driver

- Support both sync and async operations
- Can be opened multiple times
- Exploit hardware capabilities
- Do not provide policy related operations
- Simple

A split view of Linux kernel



Classes of Device Drivers

- Char device
  - Access as a stream of bytes
  - Open(), close(), read(), write()
  - Accessed by file system nodes, e.g. /dev/console, /dev/ttyS0
- Block device
  - Transfer data in blocks
- Network device
  - Exchange data over network
  - Knows about packets, but not others things like “connections’
  - NOT mapped to a node in file system
- USB device

Building and running modules

- Device driver is designed, loaded, unloaded as kernel modules
- Get your development system ready
  - Kernel source
  - Compilers
  - Check your linux distribution about how to setup
- Example:
  - Hello world module

Hello World Module

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Compile and run

```
% make
make[1]: Entering directory `/usr/src/linux-2.6.10'
  CC [M]  /home/ldd3/src/misc-modules/hello.o
  Building modules, stage 2.
  MODPOST
  CC      /home/ldd3/src/misc-modules/hello.mod.o
  LD [M]  /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

Hands-on Exercise [30 min]

- Download a copy VirtualBox
- Download a Linux virtual machine
- Create a helloworld.c
- Compile and run

Kernel module vs user apps

- Kernel module needs to do init and exit very carefully
- No printf and other because no libc or other libraries
- Linux kernel header files (e.g. <include/linux>)
- Bugs in device driver may crash kernel
- Runs in kernel space
- Much greater concurrency in kernel modules
  - Interrupt and interrupt handler
  - Timers,
  - SMP support
- So, device driver must
  - be *reentrant, handle concurrency and avoid race conditions*.

A parallel port device

- Figure: PIC based sensor <-> parallel port

Device driver for the PIC-based sensor, connected to parallel port

- The mission
  - Write a char device driver for this sensor device
    - Device driver module named pp_adc
    - Operations: open(), close(), read(), write()
    - Using device file named /dev/pp_adc0
  - Allow user apps to send commands to the sensor
    - reset, ping, enable, disable, set_in_between, set_outside, get

Walk through the Skeleton Code [40 min]

- pp_adc.c

Interrupt

- Slides from Brey's text book

Polling vs Interrupt, the big picture

- Polling
  - Keep reading
  - Consume CPU cycles
  - Suitable for ultra high speed I/O
- Interrupt
  - Asynchronous
  - Need based service
  - Best for slow speed I/O
- Example scenario
  - User app needs to sound an alarm if the sensor reading is between 5 and 10 units

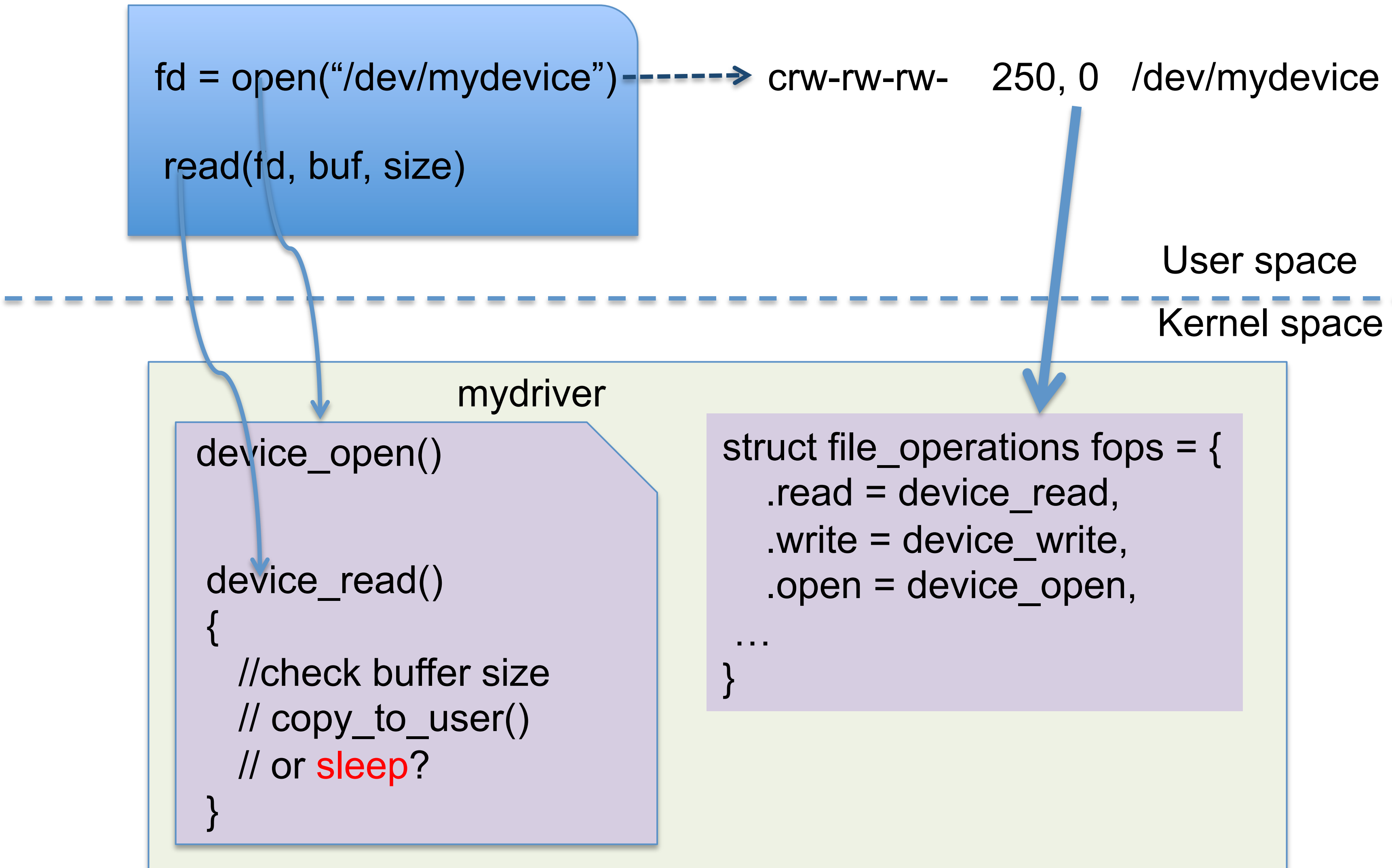
Interrupt handling

- Interrupt handler (ISR)
  - Some work needs to be done when interrupt from the device happens
  - The amount of work depends on the actual device
- Restrictions on ISR
  - Is not executed in the context of a process
    - Thus cannot transfer data to or from user space
  - Must NOT sleep!
    - Must not call anything like wait_event, locking a semaphore, or scheduler
- The role
  - Give feedback to device about interrupt reception
  - Read/write data
  - Clear INT bit
  - Awaken processes sleeping on the device or waiting for some events

Blocking I/O

- What if a driver cannot immediately satisfy the request? E.g.
  - Read when no data is available
  - Write when the device is not ready to accept data
- The calling process does not care about such issues
  - Programmer simply calls read or write
  - Have the call return after necessary work is done
- The driver should
  - "*block*" the process
  - Put it to sleep until the request can proceed

System Call from a User Application



Sleeping

- What does it mean for a process to "sleep"?
  - Marked as being in a special state
  - Removed from scheduler's run queue
  - Will not run until some future event happens
- Rules of sleeping
  - Never sleep when in atomic context (holding a lock, disabled interrupts, etc.)
  - Cannot assume the state of the system after waking up (e.g. resources may not be available)
  - Make sure some other processes can wake up

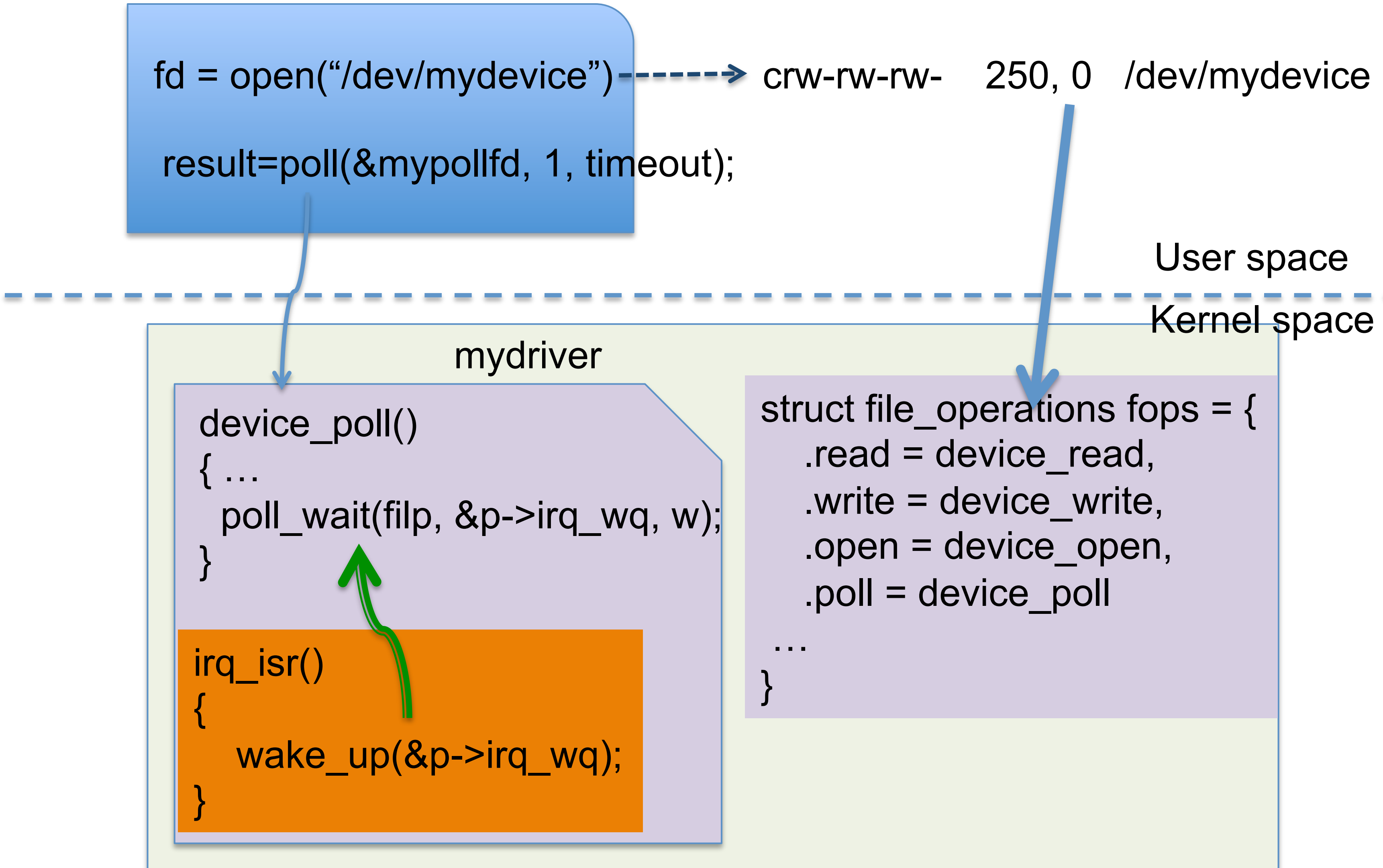
Wait Queue

- A kernel structure
  - A list of processes all waiting for a specific event
  - Make it possible for your sleeping process to be found
- Managed by "wait queue head"
  - wait_queue_head_t, is defined in <linux/wait.h>.
  - be defined and initialized statically with:
    - DECLARE_WAIT_QUEUE_HEAD(name);
  - or dynamicly as follows:
    - wait_queue_head_t my_queue;
    - init_waitqueue_head(&my_queue);

Select/poll system call

- The select/poll system call
  - allows userspace applications to wait for data to arrive on one or more file descriptors.
  - call the f_ops->poll method of all file descriptors.
  - Each ->poll method should return whether data is available or not.
  - If no file descriptor has any data available, then the poll/select call has to wait for data on those file descriptors.
  - It has to know about all wait queues that could be used to signal new data.

select/poll from a User Application



Example

```
unsigned int example_poll(struct file * file,
poll_table * pt) {
   unsigned int mask = 0;
   if (data_avail_to_read) mask |= POLLIN | POLLRDNORM;
   if (data_avail_to_write) mask |= POLLOUT |
POLLWRNORM;
   poll_wait(file, &read_queue, pt);
   poll_wait(file, &write_queue, pt);
   return mask;
}
```

Then, when data is available again the driver should call:

```
   data_avail_to_read = 1;
   wake_up(&read_queue);
```

- Review of pp_adc.c [20 min]