

# Operating Systems

## Devices

**Irvin Aloise**

`ialoise@diag.uniroma1.it`

Department of Computer Control and Management Engineering  
Sapienza University of Rome

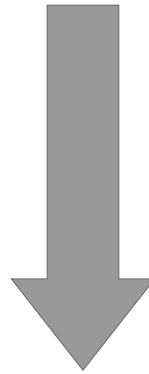
# Devices in Linux (1)

Computers have to interact with many different devices

- Storage devices (e.g. disks)
- Transmission devices (e.g. network, Bluetooth, ...)
- Generic IO devices (e.g. keyboard, joystick, audio and video capture, ...)

# Devices in Linux (2)

How do we interact with all those guys?



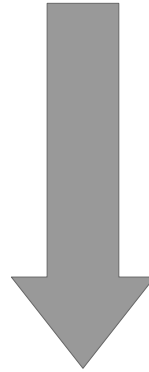
We stick to the Linux  
mantra that says:

*“Everything is a file”*

# DMA (1)

Some devices have to transfer a big amount of data all together

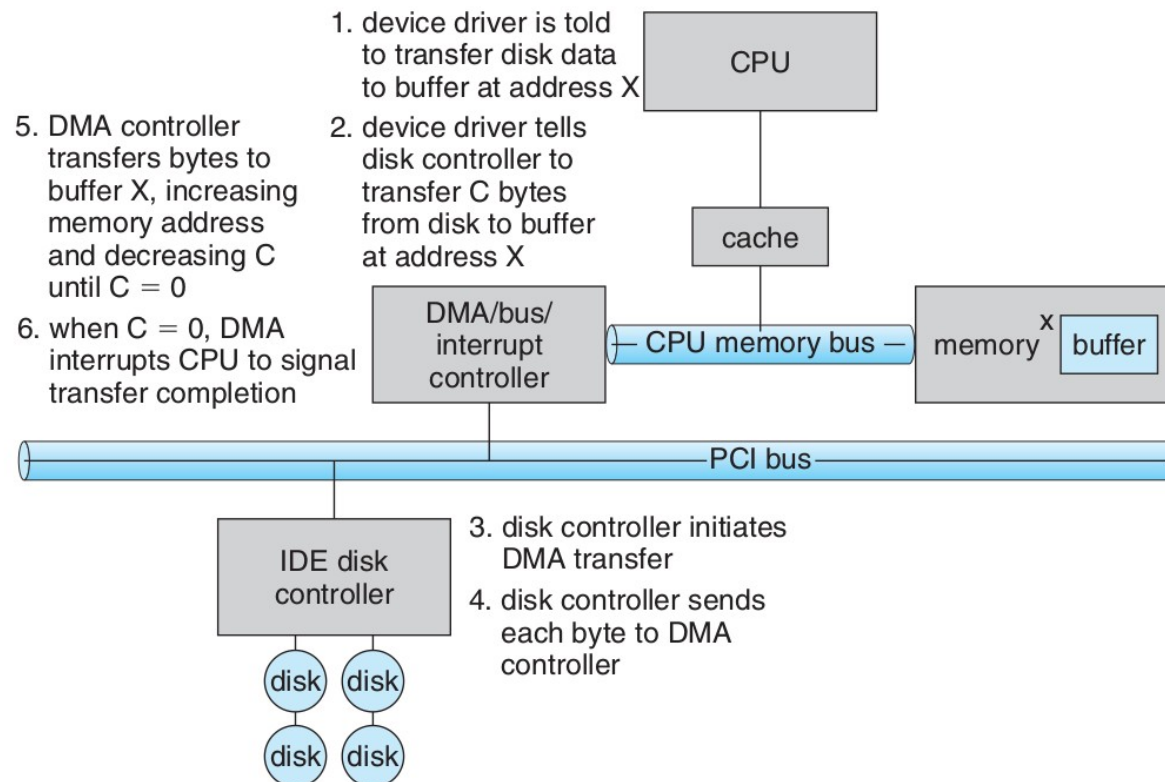
How to do this without bothering the CPU too much?



*Direct Memory Access*

# DMA (2)

Basic concept: the device puts its data in a reserved memory area and generates a **single interrupt** – only when the transfer is *completed*.



# Study Case: Joystick

Joystick is relatively easy to handle in Linux

- Most of joysticks are natively supported



- The joystick is a *device*

```
int fd = open ("/dev/input/js0", O_RDONLY);
```

- Joystick reading is event-based

```
struct js_event e;  
read (fd, &e, sizeof(e)); // do it forever
```

# Study Case: Serial Port (1)

Dealing with a **serial** device requires to setup specific parameters relative to this type of communication.

- Serial is seen as a file
- Linux provides APIs to deal with such class of devices – called `termios`
- `termios` allow us to setup communication parameters (e.g. comm speed)
- Once we setup the parameters, we can interact with it as a simple file – using `read` and `write`

# Study Case: Serial Port (2)

## How to deal with a serial in practice

1) Open serial and get a file descriptor

```
int fd = open (name, O_RDWR | O_NOCTTY | O_SYNC );
if (fd < 0) {
    printf ("error %d opening serial, fd %d\n", errno, fd);
}
```

2) Setup parameters using termios

```
int serial_set_interface_attribs(int fd, int speed, int parity) {
    struct termios tty;
    memset (&tty, 0, sizeof tty);
    if (tcgetattr (fd, &tty) != 0) {
        printf ("error %d from tcgetattr", errno);
        return -1;
    }
    switch (speed){
        case 57600:
            speed=B57600;
            break;
        case 115200:
            speed=B115200;
            break;
        default:
            printf("cannot set baudrate %d\n", speed);
            return -1;
    }
    cfsetospeed (&tty, speed);
    cfsetispeed (&tty, speed);
    cfmakeraw(&tty);
    // enable reading
    tty.c_cflag &= ~(PARENB | PARODD);           // shut off parity
    tty.c_cflag |= parity;
    tty.c_cflag = (tty.c_cflag & ~CSIZE) | CS8;   // 8-bit chars

    if (tcsetattr (fd, TCSANOW, &tty) != 0) {
        printf ("error %d from tcsetattr", errno);
        return -1;
    }
    return 0;
}
```

3) Read/write

```
int n=read (cl->fd, &c, 1);
ssize_t res = write(cl->fd,&c,1);
```

4) Close serial

```
close(cl->fd);
```



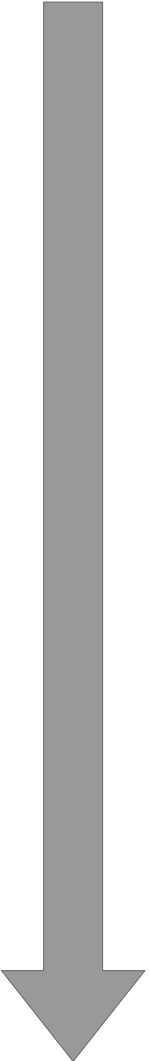
# Study Case: Camera (1)

Cameras require to transfer many bytes all together – i.e. images – and have slightly more complex drivers to deal with.

- Obviously it is seen as a *file* :)
- Linux provides APIs to control camera through V4L (now V4L2) library
- DMA is preferred (used with `mmap`)
- Device must be probed to understand how it works and its capabilities – through the `ioctl` black magic

# Study Case: Camera (2)

Generic workflow to acquire raw images using V4L2.

- 
- 1) Open the camera as a file
  - 2) Query the device to gather its property
  - 3) Setup DMA and buffers
  - 4) Request buffers from device
  - 5) Capture frames and save them on disk
  - 6) Deallocate memory and close device

# Study Case: Camera (3)

```
int main(int argc, char** argv)
{
    if (argc != 2) {
        printf("usage: <executable> <number_of_frames> - eg ./camera_capture 100\n");
        return -1;
    }

    const int num_frames = atoi(argv[1]);
    if (num_frames < 0) {
        printf("error, invalid number of frames - it must be positive :)\n");
        printf("usage: <executable> <number_of_frames> - eg ./camera_capture 100\n");
        return -1;
    }

    camera_t* camera = camera_open("/dev/video0", 640, 480);
    camera_init(camera);
    camera_start(camera);

    struct timeval timeout;
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;
    char image_name[1024];
    for (int i = 0; i < num_frames; ++i) {
        if (camera_frame(camera, timeout) > 0) {
            sprintf(image_name, "image-%05d.pgm", i);
            printf("acquiring frame %d\n", i);
            savePGM(camera, image_name);
        }
    }
    camera_frame(camera, timeout);

    camera_stop(camera);
    camera_finish(camera);
    camera_close(camera);
    return 0;
}
```