DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING

[INQ0091623] AA 2021-22

Lab 3

Working with devices 2

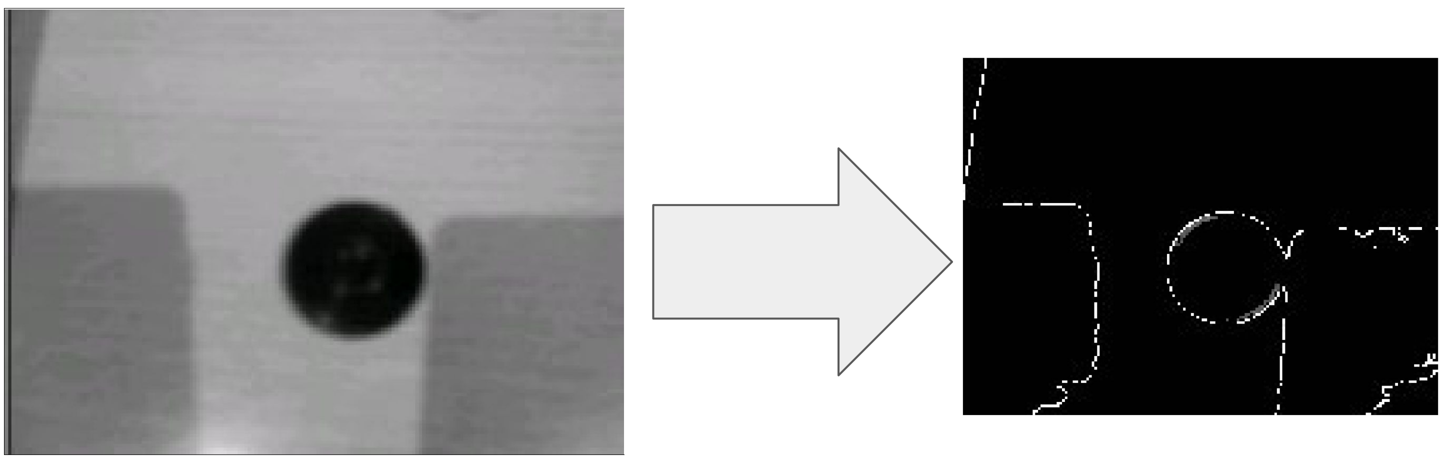
Video 4 Linux Application Example

Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

Tasks

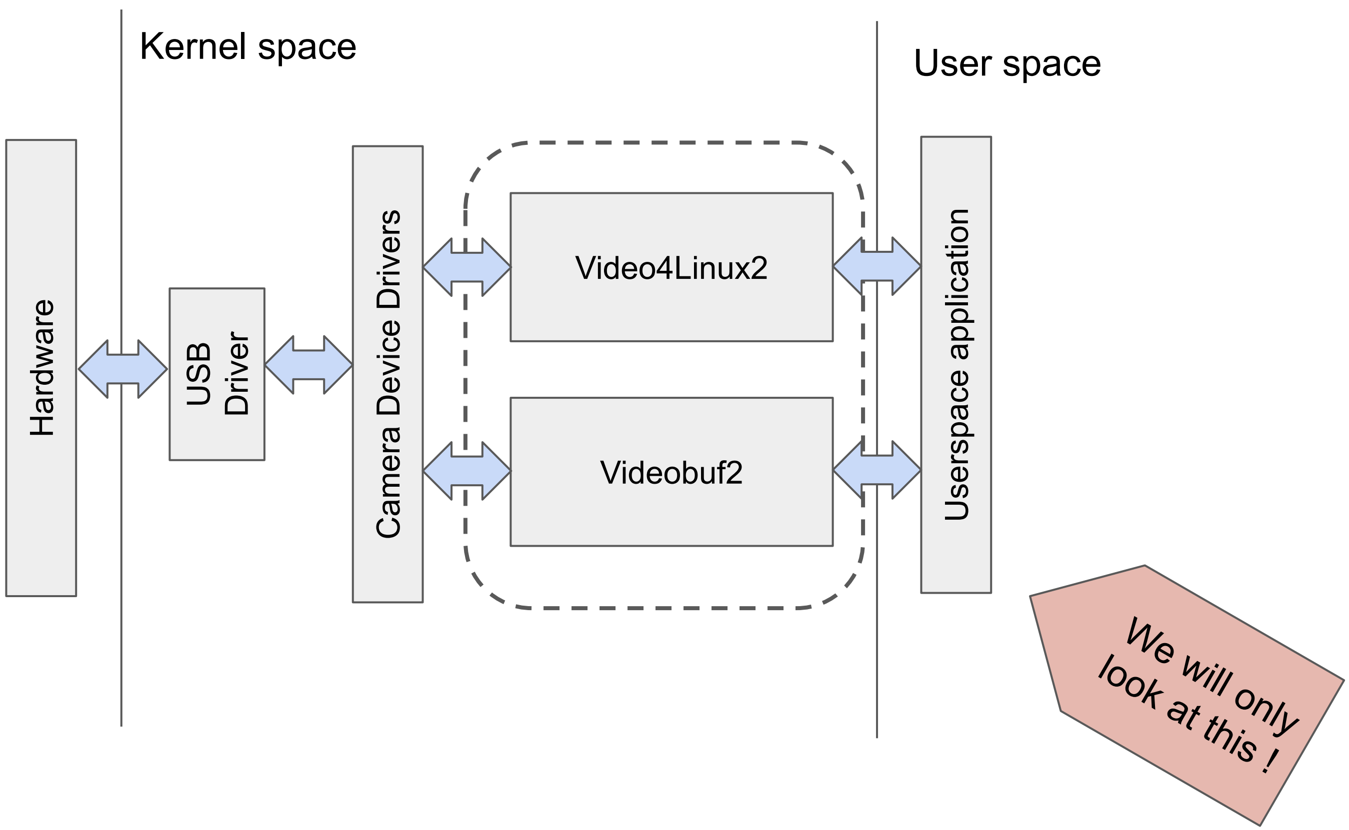
- A Linux application to acquire WebCam images
- Process image to find edges
- We shall use this example to summarize several facts about the Operating System and as a replay for some basic techniques in C programming



I/O abstraction: A Unified interface for camera devices

(Wikipedia)

Video4Linux (V4L for short) is a collection of **device drivers** and an **API** for supporting realtime video capture on Linux systems. It supports many USB webcams, TV tuners, and related devices, standardizing their output, so programmers can easily add video support to applications.



I/O abstraction: A Unified interface for camera devices

- V4L2 (Video for Linux Two), defines a set of ioctl operations and associated data structures.
  - They are general enough to be adapted for all the available camera devices of common usage.
- An important feature is the availability of **query operations** for discovering the supported functionality of the device.
  - To adapt the great variety of camera devices
- V4L2 provides a way to handle a stream of video buffers to pass acquired frames in userspace application.

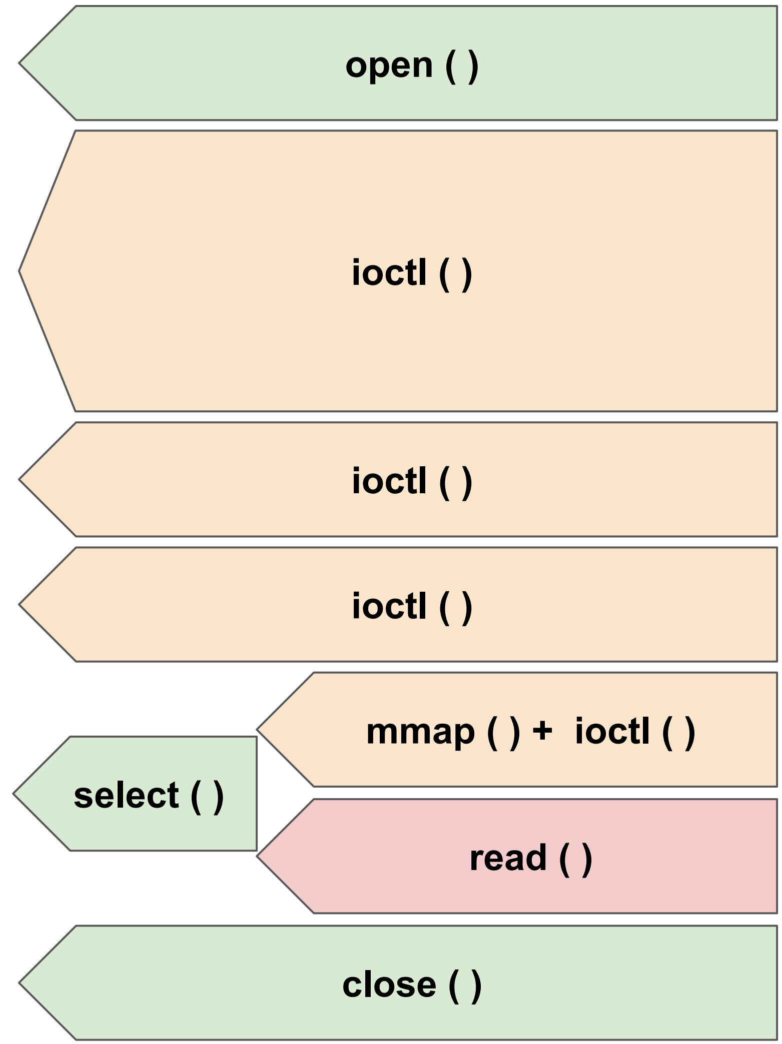
https://www.linuxtv.org/downloads/v4l-dvb-apis-new/userspace-api/v4l/v4l2.html

Common V4L2 API elements

Programming a V4L2 device consists of these steps:

- **Open the device**
- **Change device properties:**
    - selecting a video and audio input
    - video standard
    - picture brightness
    - ...
- **Negotiate a data format**
- **Negotiate an input/output method**
- **Write the actual input/output LOOP**
- **Close the device**

FILE OPERATIONS



Opening and Closing Devices

V4L2 drivers are implemented as kernel modules, loaded manually by the system administrator or automatically when a device is first discovered.

The driver modules plug into the videodev kernel module.

Each driver registers one or more device nodes with **major number 81**.
Minor numbers are allocated dynamically.

```
$ cd /dev/
$ ls -la video*
crw-rw----+ 1 root video 81, 0 Oct 14 04:03 video0
crw-rw----+ 1 root video 81, 1 Oct 14 04:03 video1
```

Default device node name	Usage
`/dev/videoX`	Video and metadata for capture/output devices
`/dev/vbiX`	Vertical blank data (i.e. closed captions, teletext)
`/dev/radioX`	Radio tuners and modulators
`/dev/swradioX`	Software Defined Radio tuners and modulators
`/dev/v4l-touchX`	Touch sensors
`/dev/v4l-subdevX`	Video sub-devices (used by sensors and other components of the hardware peripheral)[1]

Opening and Closing Devices

```
$ v4l2-ctl --list-devices

UVC Camera (046d:081b)
(usb-0000:00:1a.0-1.3):
        /dev/video0
        /dev/video1
```

```c
fd = open("/dev/video0", O_RDWR);

  ...

close(fd);
```

Which camera is attached to a specific dev file ??

```
$ cd dev/
$ tree v4l

v4l
|-- by-id
|   |-- usb-046d_081b_1F59AD20-video-index0 -> ../../video0
|   `-- usb-046d_081b_1F59AD20-video-index1 -> ../../video1
`-- by-path
    |-- pci-0000:00:1a.0-usb-0:1.3:1.0-video-index0 -> ../../video0
    `-- pci-0000:00:1a.0-usb-0:1.3:1.0-video-index1 -> ../../video1
```

Querying Capabilities

Because V4L2 covers a wide variety of devices not all aspects of the API are equally applicable to all types of devices. Furthermore devices of the same type have different capabilities and this specification permits the omission of a few complicated and less important parts of the API.

The ioctl VIDIOC_QUERYCAP ioctl is available to check if the kernel device is compatible with this specification, and to query the functions and I/O methods supported by the device.

V4L2_CAP_READWRITE	0x01000000	The device supports the **read()** and/or **write()** I/O methods.
V4L2_CAP_ASYNCIO	0x02000000	The device supports the asynchronous I/O methods.
V4L2_CAP_STREAMING	0x04000000	The device supports the streaming I/O method.

For example in our application we might be interested in checking for streaming input

```
/* Step 2: Check streaming capability */
   status = ioctl(fd, VIDIOC_QUERYCAP, &cap);
   CHECK_IOCTL_STATUS("Error querying capability")
   if(!(cap.capabilities & V4L2_CAP_STREAMING))
   {
       printf("Streaming NOT supported\n");
       exit(EXIT_FAILURE);
   }
```

v4l2-ctl - device info

Also the ***v4l2-ctl*** command is able to query many informations from the device:

```
$ v4l2-ctl --info --device /dev/video0
Driver Info:
	Driver name      : uvcvideo
	Card type        : UVC Camera (046d:081b)
	Bus info         : usb-0000:00:1a.0-1.3
	Driver version   : 5.10.14
	Capabilities     : 0x84a00001
		Video Capture
		Metadata Capture
		Streaming
		Extended Pix Format
		Device Capabilities
	Device Caps      : 0x04200001
		Video Capture
		Streaming
		Extended Pix Format
```

STEP 3 - Query Formats

The word ***format*** in the V4L context is used to identify how the image is represented in memory

The V4L2 API was primarily designed for devices exchanging **image data** with applications. The struct [v4l2_pix_format]() and struct [v4l2_pix_format_mplane]() structures define the format and layout of an image in memory.

Image formats are negotiated with the [VIDIOC_S_FMT]() **ioctl**.

```c
/* Step 3: Check supported formats */
    yuyvFound = FALSE;
    for(idx = 0; idx < MAX_FORMAT; idx++)
    {
        fmt.index = idx;
        fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        status = ioctl(fd, VIDIOC_ENUM_FMT, &fmt);
        if(status != 0) break;
        if(fmt.pixelformat == V4L2_PIX_FMT_YUYV)
        {
            yuyvFound = TRUE;
            break;
        }
    }
    if(!yuyvFound)
    {
        printf("YUYV format not supported\n");
        exit(EXIT_FAILURE);
    }
```

Query formats using **v4l2-ctl**

```
$ v4l2-ctl --list-formats --device /dev/video0

ioctl: VIDIOC_ENUM_FMT
        Type: Video Capture

        [0]: 'YUYV' (YUYV 4:2:2)
        [1]: 'MJPG' (Motion-JPEG, compressed)
```

Standard image formats

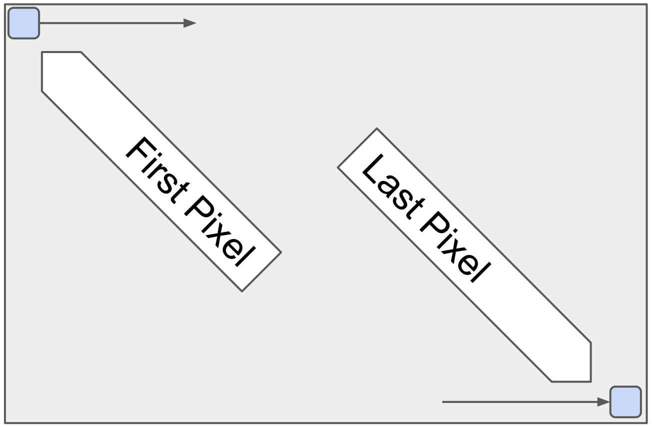
To exchange images between the driver and the application, it is necessary to have standard image data formats which both sides will interpret the same way.

V4L2 includes several standard formats already defined. In V4L2 each format has an identifier which looks like **PIX_FMT_XXX**, defined in the videodev2.h header file.

NOTE: Custom driver-specific formats are possible. In that case the application may depend on a codec to convert images to one of the standard formats when needed. For example, a device may support a proprietary compressed format. Applications can still capture and save the data in the compressed format, saving much disk space, and later use a codec to convert the images to the X Windows screen format when the video is to be displayed.

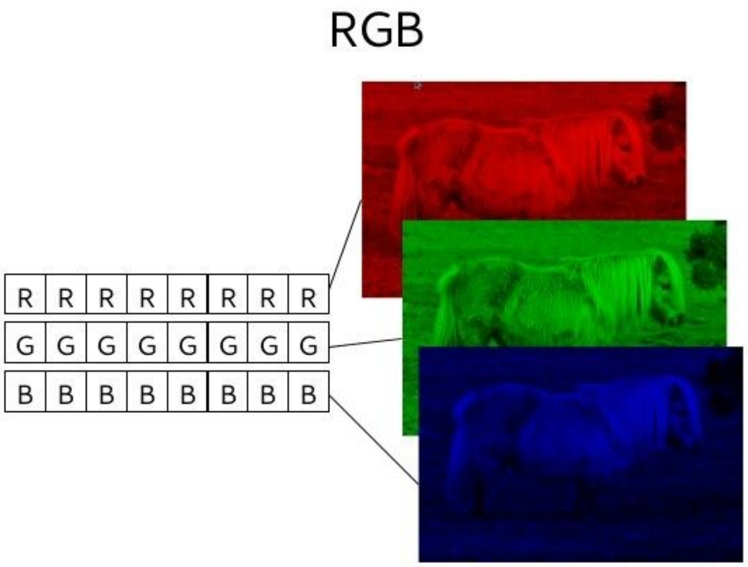
The V4L2 standard formats are mainly uncompressed formats.

- The pixels are always arranged in memory **from left to right**, and **from top to bottom**.
- The first byte of data in the image buffer is always for the leftmost pixel of the topmost row.



RGB

These formats encode each pixel as a triplet of RGB values. They are packed formats, meaning that the RGB values for one pixel are stored consecutively in memory and each pixel consumes an integer number of bytes. When the number of bits required to store a pixel is not aligned to a byte boundary, the data is padded with additional bits to fill the remaining byte.

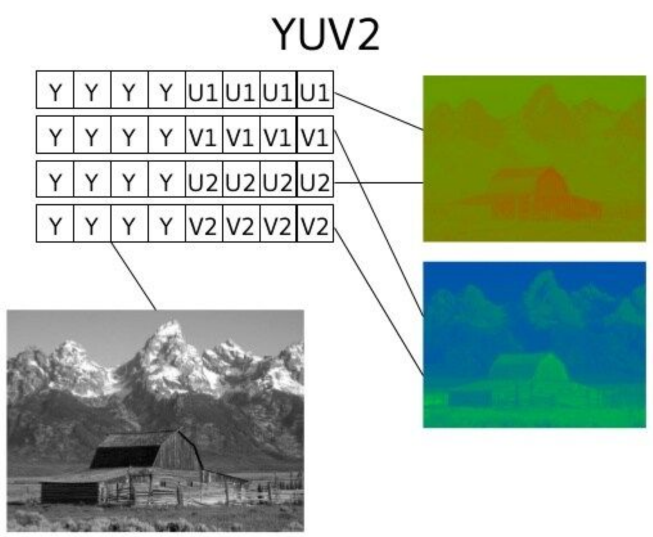


YUV

YUV is the format native to TV broadcast and composite video signals. It separates the **luminance** brightness information (Y) from the **chrominance** color information (U and V or Cb and Cr). The color information consists of red and blue color difference signals, this way the green component can be reconstructed by subtracting from the brightness component.

U = B′ − Y′ (blue − luma)
V = R′ − Y′ (red − luma)

NOTE: YUV was chosen because early television would only transmit brightness information. To add color in a way compatible with existing receivers a new signal carrier was added to transmit the color difference signals.



STEP 4-5-6 - Request Format

CODING EXAMPLE

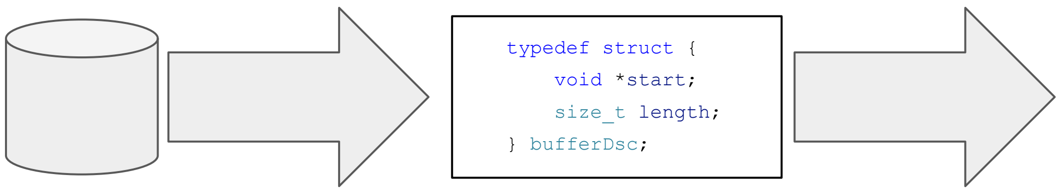
CODING EXAMPLE

```
/* Step 4: Read current format definition */
   memset(&format, 0, sizeof(format));
   format.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
   status = ioctl(fd, VIDIOC_G_FMT, &format);
   CHECK_IOCTL_STATUS("Error Querying Format")

/* Step 5: Set format fields to desired values: YUYV coding,
   480 lines, 640 pixels per line */
   format.fmt.pix.width = 640;
   format.fmt.pix.height = 480;
   format.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;

/* Step 6: Write desired format and check actual image size */
   status = ioctl(fd, VIDIOC_S_FMT, &format);
   CHECK_IOCTL_STATUS("Error Setting Format");
   width = format.fmt.pix.width;                       //Image Width
   height = format.fmt.pix.height;                     //Image Height
   //Total image size in bytes
   imageSize = (unsigned int)format.fmt.pix.sizeimage;
```

Streaming buffers

Streaming is an I/O method where **only pointers to buffers are exchanged between application and driver**, the **data itself is not actually copied**.

Memory mapping is primarily intended to map buffers in device memory into the application's address space. Device memory can be for example the video memory on a graphics card with a video capture add-on.



First of all we need to ask V4L and the driver to allocate some frames with the **ioctl VIDIOC_REQBUFS**.

struct v4l2_requestbuffers

__u32	`count`	The number of buffers requested or granted.
__u32	`type`	Type of the stream or buffers, this is the same as the struct **`v4l2_format`** `type` field. See **`v4l2_buf_type`** for valid values.
__u32	`memory`	Applications set this field to `V4L2_MEMORY_MMAP`, `V4L2_MEMORY_DMABUF` or `V4L2_MEMORY_USERPTR`. See **`v4l2_memory`**.
__u32	`capabilities`	Set by the driver. If 0, then the driver doesn't support capabilities. In that case all you know is that the driver is guaranteed to support `V4L2_MEMORY_MMAP` and *might* support other **`v4l2_memory`** types. It will not support any other capabilities. If you want to query the capabilities with a minimum of side-effects, then this can be called with `count` set to 0, `memory` set to `V4L2_MEMORY_MMAP` and `type` set to the buffer type. This will free any previously allocated buffers, so this is typically something that will be done at the start of the application.
__u32	`reserved[1]`	A place holder for future extensions. Drivers and applications must set the array to zero.

STEP 7 - Request Buffer

```
/* Step 7: request for allocation of 4 frame buffers by the driver */
   reqBuf.count = 4;
   reqBuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
   reqBuf.memory = V4L2_MEMORY_MMAP;
   status = ioctl(fd, VIDIOC_REQBUFS, &reqBuf);
   CHECK_IOCTL_STATUS("Error requesting buffers")
/* Check the number of returned buffers. It must be at least 2 */
   if(reqBuf.count < 2)
   {
       printf("Insufficient buffers\n");
       exit(EXIT_FAILURE);
   }
```

CODING EXAMPLE

CODING EXAMPLE

STEP 8 - Map Buffers

To get informations about the actual buffer that the device driver is exposing a special structure is exchanged by the **ioctl** [**VIDIOC_QUERYBUF**]

```c
struct v4l2_buffer {
    __u32           index;
    __u32           type;
    __u32           bytesused;
    __u32           flags;
    __u32           field;
    struct timeval      timestamp;
    struct v4l2_timecode    timecode;
    __u32           sequence;
```

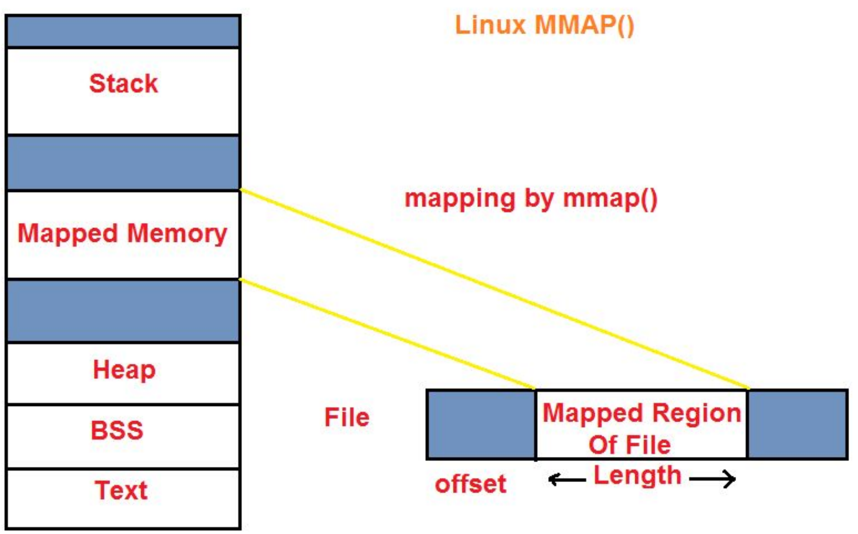
```c
    /* memory location */
    __u32           memory;
    union {
        __u32           offset;
        unsigned long   userptr;
        struct v4l2_plane *planes;
        __s32       fd;
    } m;
    __u32           length;
    __u32           reserved2;
    union {
        __s32       request_fd;
        __u32       reserved;
    };
};
```

STEP 8 - mmap buffers

CODING EXAMPLE

CODING EXAMPLE

```
struct v4l2_buffer buf;            //Buffer setup structure
buffers = calloc(reqBuf.count, sizeof(bufferDsc));

 for(idx = 0; idx < reqBuf.count; idx++)
 {
     buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
     buf.memory = V4L2_MEMORY_MMAP;
     buf.index = idx;
/* Get the start address in the driver space of buffer idx */
     status = ioctl(fd, VIDIOC_QUERYBUF, &buf);
     CHECK_IOCTL_STATUS("Error querying buffers")
/* Prepare the buffer descriptor with the address in user space
   returned by mmap() */
     buffers[idx].length = buf.length;
     buffers[idx].start = mmap(NULL, buf.length,
         PROT_READ | PROT_WRITE, MAP_SHARED,
         fd, buf.m.offset);
     if(buffers[idx].start == MAP_FAILED)
     {
         perror("Error mapping memory");
         exit(EXIT_FAILURE);
     }
 }
```

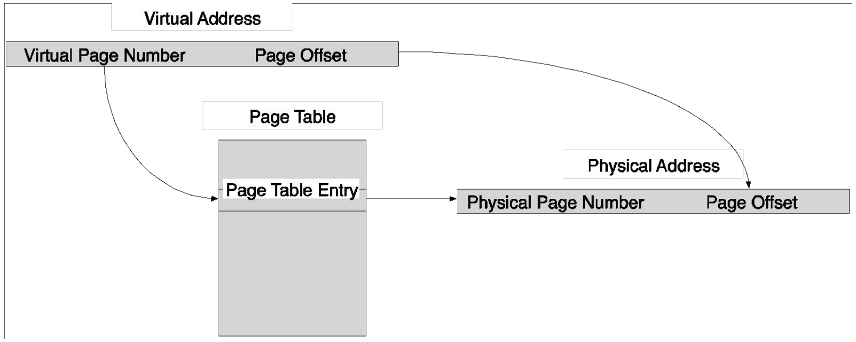


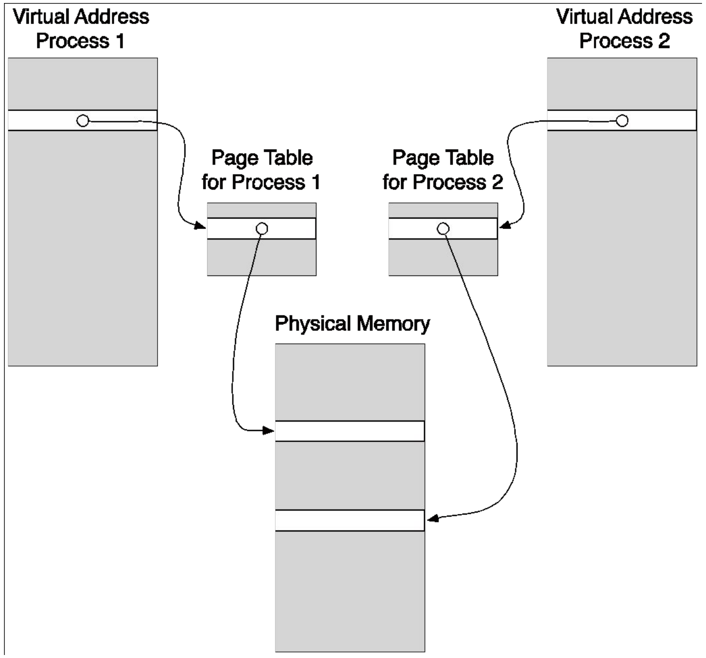
Virtual Memory (REPLAY)

- Provides flexible mapping between the process address space and the physical address space

- Allows different processes run the same program

- May represent an overhead in real-time applications because the context switch required updating Page Table





Double buffering

- In order of avoiding losing frames it is necessary to guarantee that the frame is read before the next one is acquired
- Not feasible in practice, especially for non real-time systems
  - In practice many real-time systems may experience occasional delays
- Using Double buffering , the device parks incoming frames in buffers which are then read by the program
  - In this way, frames are not lost
- Double buffering programming may face the address space problem

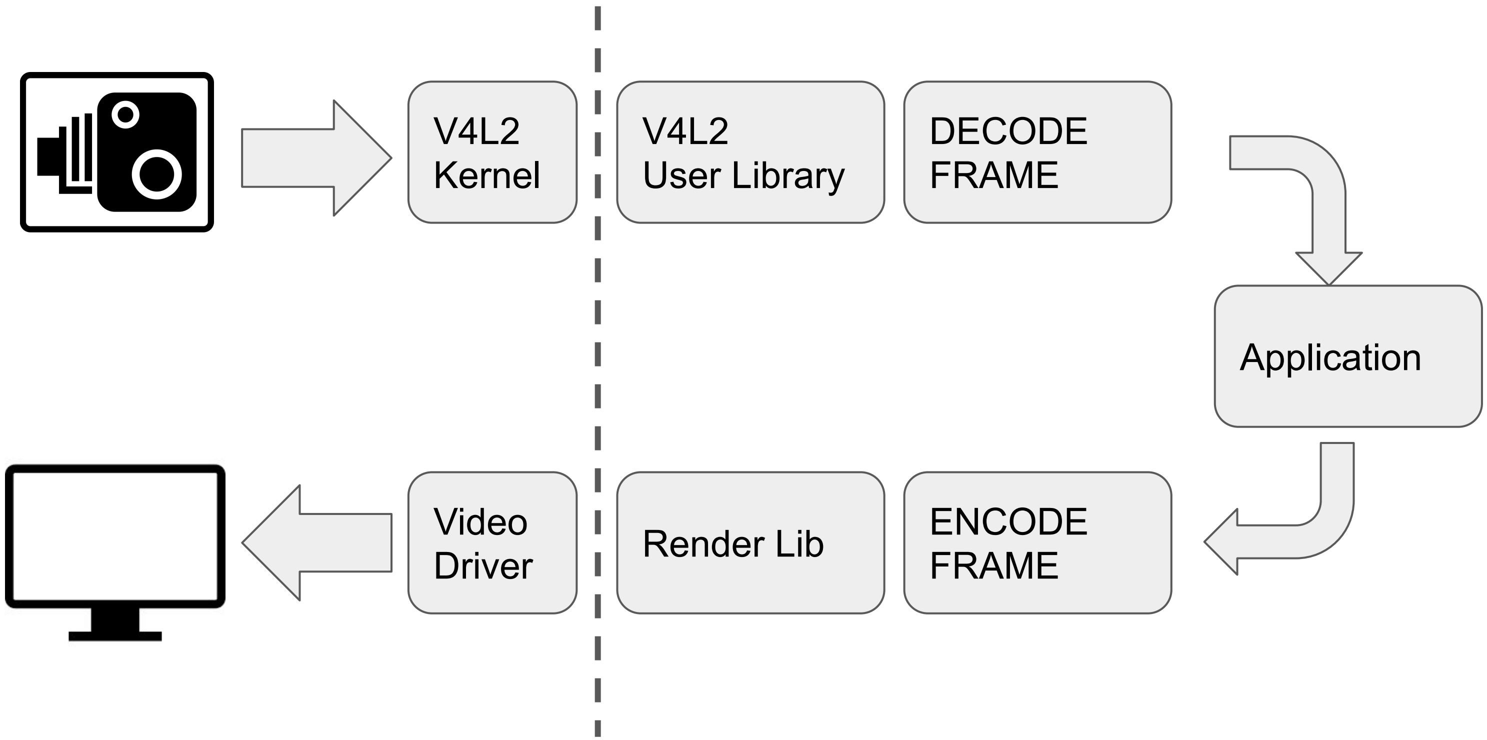
Steps in handling double buffering for camera acquisition

1. Request the device to allocate (in kernel space) a given number of buffers. The buffer offset is returned in the structure associated with ioctl() call
2. Map the buffers into process address space using mmap().
3. request the driver to enqueue all the buffers in a circular list
4. Start frame acquisition
5. Wait for buffer ready using select()
6. Dequeue the current buffer
7. Use it
8. Enqueue the buffer again

V4L_example

Stream of data in a typical video application

Look at lab3 files



Processing image example: Edge detection

- Carried out by calculating the gradient Lx and Ly over X and Y, respectively and selecting the points for which the overall gradient approximated as abs(Lx) + abs(Ly) is above a given threshold
- X and Y gradient are computed for every pixel by considering a 3x3 matrix (Sobel Matrix)
- A straight implementation uses pointer and memory allocated matrixes to carry out computation

Sobel Operator

The Sobel operator, sometimes called the Sobel–Feldman operator or Sobel filter, is used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasising edges. It is named after Irwin Sobel and Gary Feldman, colleagues at the Stanford Artificial Intelligence Laboratory (SAIL).

If we define A as the source image, and Gx and Gy are two images which at each point contain the horizontal and vertical derivative approximations respectively

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Where * is the discrete convolution operator

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x+dx, y+dy),$$

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$