DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING

[INQ0091623] AA 2021-22

Lab 2

Working with devices

I/O Examples

Gabriele Manduchi <gabriele.manduchi@unipd.it>

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

Power on !

In the previous lesson we learned how the CPU perform the operation through the cycles of instruction that are fed to the processor pipelines and activate the internal logics.

Here we look at an higher abstraction that is WHAT ACTUALLY RUNS IN THE CPU.

What happens when a CPU starts? which is it’s PC value?

When you activate the "**reset**" signal in the CPU, it will initialise its registers to certain hard-coded values including the **Program Counter**. Then the CPU starts to execute instructions as usual: fetch, decode, execute … fetch, decode, execute … etc

The actual initial value of PC depends on the specific CPU. It can be that the CPU wakes up with PC=0, or it can be that PC=0xFFFF_FFF0 (as happens with Intel's X86 family).

This address to start the machine is usually pointing to some memory stored in ROM (non-volatile memory) at a fixed address X. This is an immutable program called **Basic Input/Output System (BIOS).**

The main purpose of the BIOS is to make the system able to access the minimal resources that are needed to start reading the program in the selected media (for example to start booting from HD).

Improve your knowledge

Bare machines

The BIOS eventually ends its execution starting a program.

This can be your program ! It is called a BARE MACHINE (or **Bare Metal Machine**)

bare-metal programming is very complex because all the services provided by the operating system have to be re-implemented.

boot process, memory management (segmenting the code and the data, mapping hardware resources and peripherals), interruptions handling, task scheduling, peripherals management, error management ...

On the other hand you have the complete control of the processor.

This is the nearest thing to implement a completely deterministic program.

Does it worth the effort ? Usually NOT ... but it depends

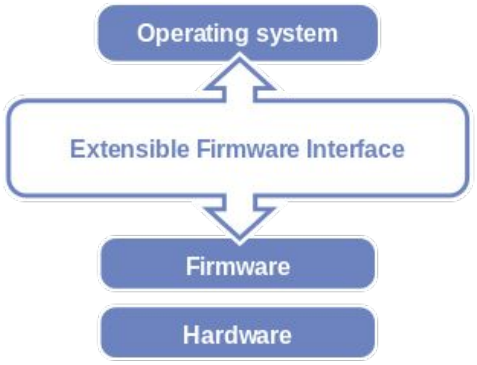
Operative system - the bootloader

If it is not your program, the BIOS starts a program called **bootloader** (in Gnu-Linux for example LILO, GRUB, GRUB2 … etc)

This is a further program that aims at loading all other complex resources that are needed to eventually start the OPERATIVE SYSTEM.

NOTE:

In recent years the BIOS became more and more complete systems including a direct access to files stored into the hard drives. In particular with all firmwares that adhere to the Unified Extensible Firmware Interface (UEFI).



Operative system

An operating system is a program itself (usually called kernel) on which application programs are executed.

It acts as a communication bridge (interface) between the **user** and the **computer hardware**.

1. **Security –** user permissions, access to programs and user data in memory.

2. **Control over system performance –**

   Monitors overall system health to help improve performance. records the response time between service requests and system response to having a complete view of the system health. This can help improve performance by providing important information needed to troubleshoot problems.

3. **Job accounting –**

   Operating system Keeps track of time and resources used by various tasks and users, this information can be used to track resource usage for a particular user or group of users.

4. **Error detecting aids –**

   The operating system constantly monitors the system to detect errors and avoid the malfunctioning of a computer system.

5. **Coordination between other software and users –**

   Operating systems also coordinate and assign interpreters, compilers, assemblers, and other software to the various users of the computer systems.

Operative system

6. **Memory Management –**

   Keeps track of primary memory, i.e., which bytes of memory are used by which user program. The memory addresses that have already been allocated and the memory addresses of the memory that has not yet been used. In multiprogramming, the OS decides the order in which processes are granted access to memory, and for how long. It Allocates the memory to a process when the process requests it and deallocates the memory when the process has terminated or is performing an I/O operation.

7. **Processor Management –**

   The OS decides the order in which processes have access to the processor, and how much processing time each process has. This function of OS is called process scheduling. An Operating System performs the following activities for processor management.

   Keeps track of the status of processes. The program which performs this task is known as a traffic controller. Allocates the CPU that is a processor to a process. De-allocates processor when a process is no more required.

Operative system

8. **Device Management –**

   An OS manages device communication via their respective drivers. It performs the following activities for device management. Keeps track of all devices connected to the system. designates a program responsible for every device known as the Input/Output controller. Decides which process gets access to a certain device and for how long. Allocates devices in an effective and efficient way. Deallocates devices when they are no longer required.

9. **File Management –**

   A file system is organized into directories for efficient or easy navigation and usage. These directories may contain other directories and other files. An Operating System carries out the following file management activities. It keeps track of where information is stored, user access settings and status of every file, and more… These facilities are collectively known as the file system.

Operative system - Services

1. **Program Execution**: The Operating System is responsible for the execution of all types of programs whether it be user programs or system programs. The Operating System utilizes various resources available for the efficient running of all types of functionalities.
2. **Handling Input/Output Operations**: The Operating System is responsible for handling all sorts of inputs, i.e, from the keyboard, mouse, desktop, etc. The Operating System does all interfacing in the most appropriate manner regarding all kinds of Inputs and Outputs.

   For example, there is a difference in the nature of all types of peripheral devices such as mice or keyboards, the Operating System is responsible for handling data between them.
3. **Manipulation of File System**: The Operating System is responsible for making decisions regarding the storage of all types of data or files, i.e, floppy disk/hard disk/pen drive, etc. The Operating System decides how the data should be manipulated and stored.
4. **Error Detection and Handling**: The Operating System is responsible for the detection of any type of error or bugs that can occur while any task. The well-secured OS sometimes also acts as a countermeasure for preventing any sort of breach to the Computer System from any external source and probably handling them.
5. **Resource Allocation:** The Operating System ensures the proper use of all the resources available by deciding which resource to be used by whom for how much time. All the decisions are taken by the Operating System.
6. **Accounting:** The Operating System tracks an account of all the functionalities taking place in the computer system at a time. All the details such as the types of errors that occurred are recorded by the Operating System.
7. **Information and Resource Protection:** The Operating System is responsible for using all the information and resources available on the machine in the most protected way. The Operating System must foil an attempt from any external resource to hamper any sort of data or information.

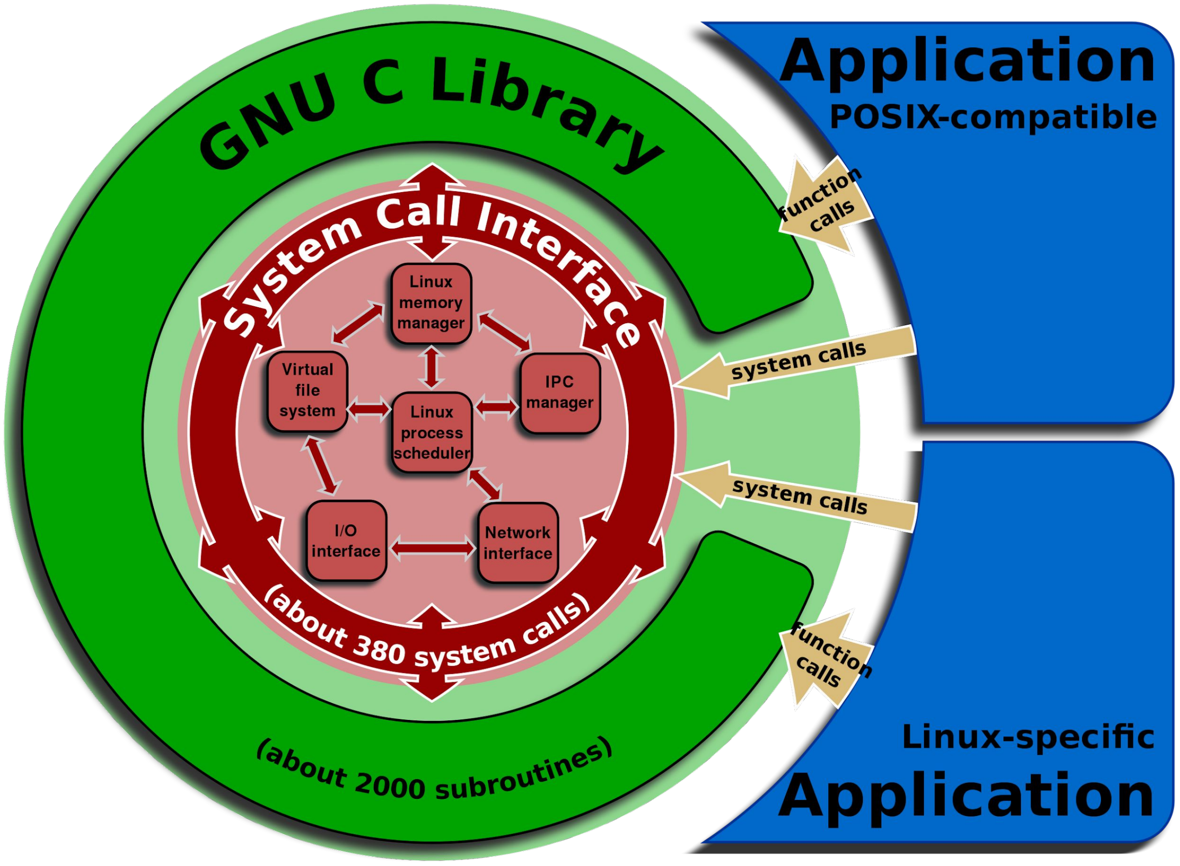
Talking with the Kernel

Kernel <~> User communication

When the Operative system is running all other applications are simple programs that interchange **signals** and **messages** with the kernel.

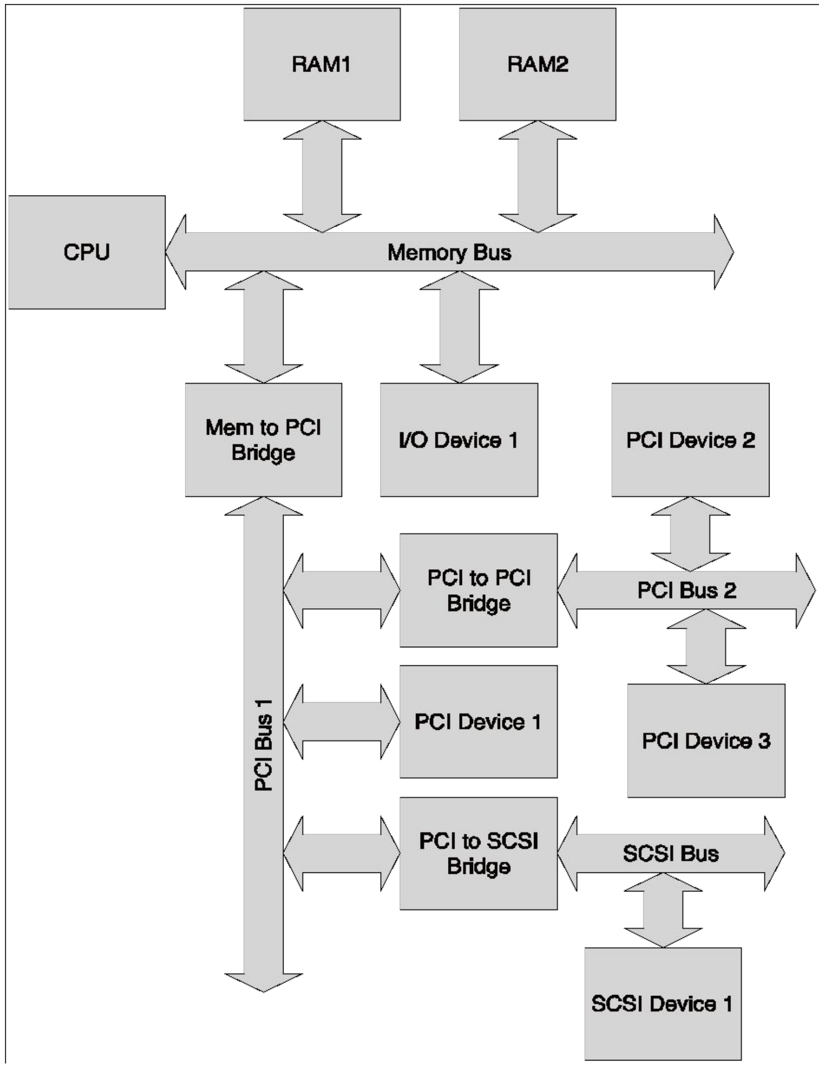
The existence of a running program within the system is called **PROCESS**.

How it works in GNU-Linux:



Working with I/O devices

- Every computer application does some I/O at a certain point
- Devices are seen as a set of registers
- In Memory Mapped architectures registers are mapped against memory addresses
- PCI bridges are used in practice to handle proper data redirection against I/O buses
- Bridges are programmed in the BIOS before the system boot

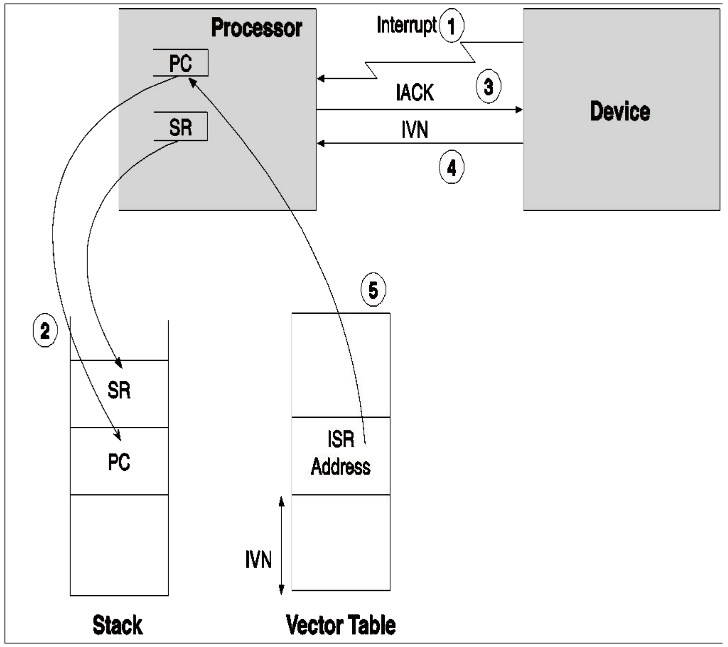


Operating System I/O abstraction: user and kernel modes

- **I/O operations are always shielded** by the Operating System
- All normal operative systems **never expose critical memory areas** and instructions
- Kernel and User mode supported by CPU HW
  - **NOTE:** In real-time Operating Systems user mode can be sometime disabled
- **Software Interrupts represent the mechanisms to enter in kernel mode**
- Device specific code is implemented by pluggable Device Drivers
- A device driver is a piece of code, normally executed in kernel mode which interacts with the device registers and presents a given, OS dependent interface
  - It becomes a component of the Operating System

I/O Synchronization Techniques

- Polling
  - Repeatedly check a status register
- Interrupt
  - Do actions only when requested by the device
- DMA
  - Let device perform data transfer and interrupt when transfer terminated



Polling often used in real-time systems

Possibly in conjunction with DMA

The history of a printf() call (REPLAY)

1. The program calls routine printf(), provided by the C run-time library.
2. The printf code carries out the required formatting of the passed string and the other optional arguments, and then calls the operating system specific system service for writing the formatted string on the screen;
3. The system routine executes initially in user mode, makes some preparatory work and then switches to kernel mode issuing a software interrupt
4. The ISR is eventually activated by the processor in response to the software interrupt.
5. After some work to prepare the required data structures, the ISR routine will interact with the output device. To do this, it will call specific routines of the device driver;
6. The activated driver code will write appropriate values in the device registers to start transferring the string to the video device. In the meantime the calling process is put in wait state
7. A sequence of interrupts will be likely generated by the device to handle the transfer of the bytes of the string to be printed on the screen;
8. When the whole string has been printed on the screen, the calling process will be resumed by the operating system and printf() will return.

Device driver interface

In GNU-Linux most of the drivers exposes their API as a file.

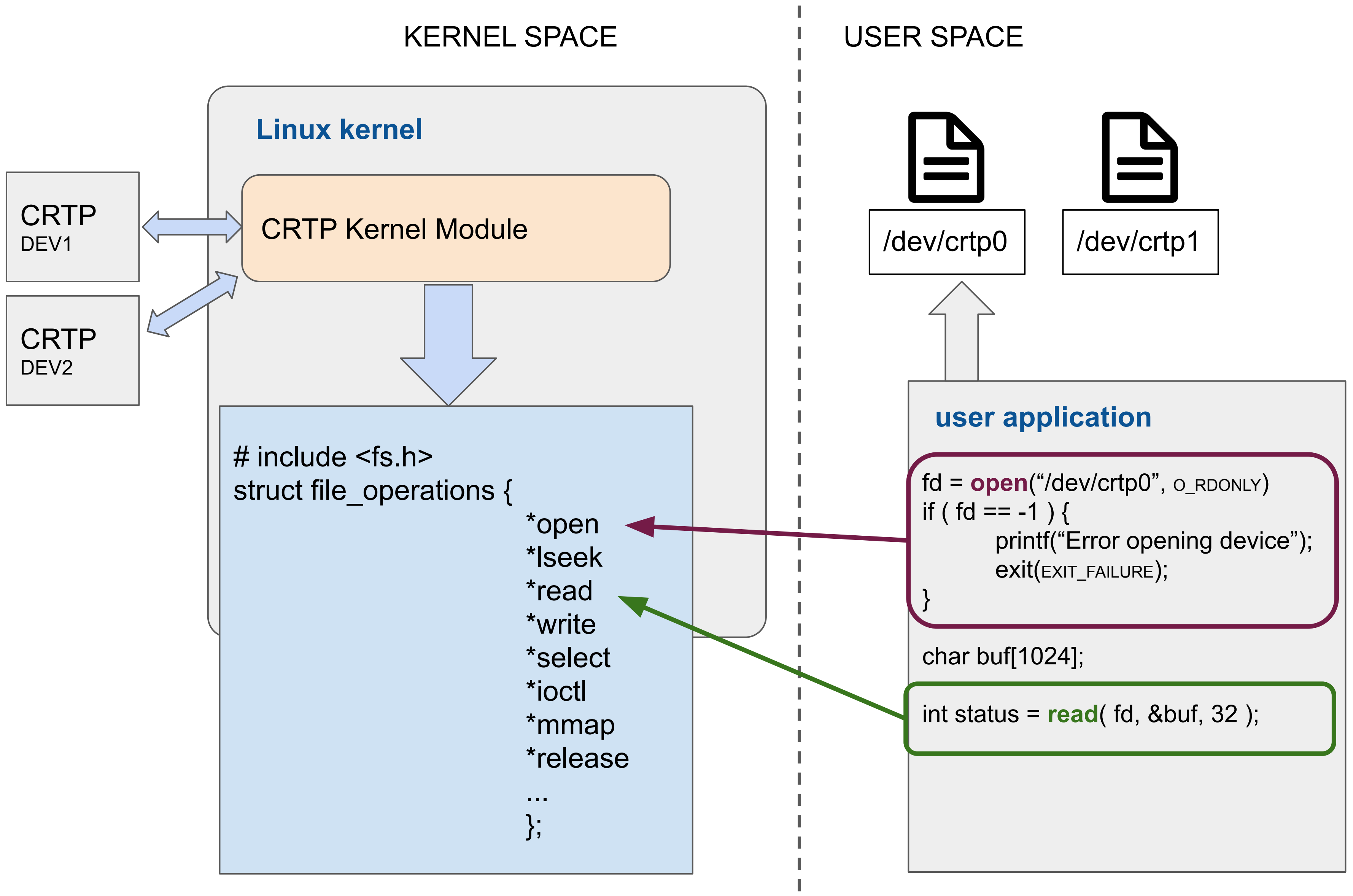
Everything in Linux can be viewed as a file, including: regular files, linked files, Sockets, device drivers, etc...

In the first versions of Linux, all filesystem access went straight into routines which understood the minix filesystem. To make it possible for other filesystems to be written, filesystem calls had to pass through a layer of indirection which would switch the call to the routine for the correct filesystem. This was done by some generic code which can handle generic cases and a structure of pointers to functions which handle specific cases.

The **Virtual Filesystem Switch** (VFS), is the mechanism which allows Linux to mount many different file-systems at the same time.

This structure is called **file_operations** and became the main structure of function that implements all possible kinds of communication from userspace and the kernel module that implements the device driver.

```
struct file_operations {
   int  (*lseek)   (struct inode *, struct file *, off_t, int);
   int  (*read)    (struct inode *, struct file *, char *, int);
   int  (*write)   (struct inode *, struct file *, char *, int);
   int  (*readdir) (struct inode *, struct file *, struct dirent *, int count);
   int  (*select)  (struct inode *, struct file *, int, select_table *);
   int  (*ioctl)   (struct inode *, struct file *, unsigned int, unsigned int);
   int  (*mmap)    (struct inode *, struct file *, unsigned long, size_t, int, unsigned long);
   int  (*open)    (struct inode *, struct file *);
   void (*release) (struct inode *, struct file *);
};
```



```
# include <fs.h>
struct file_operations {
        *open
        *lseek
        *read
        *write
        *select
        *ioctl
        *mmap
        *release
        ...
};
```

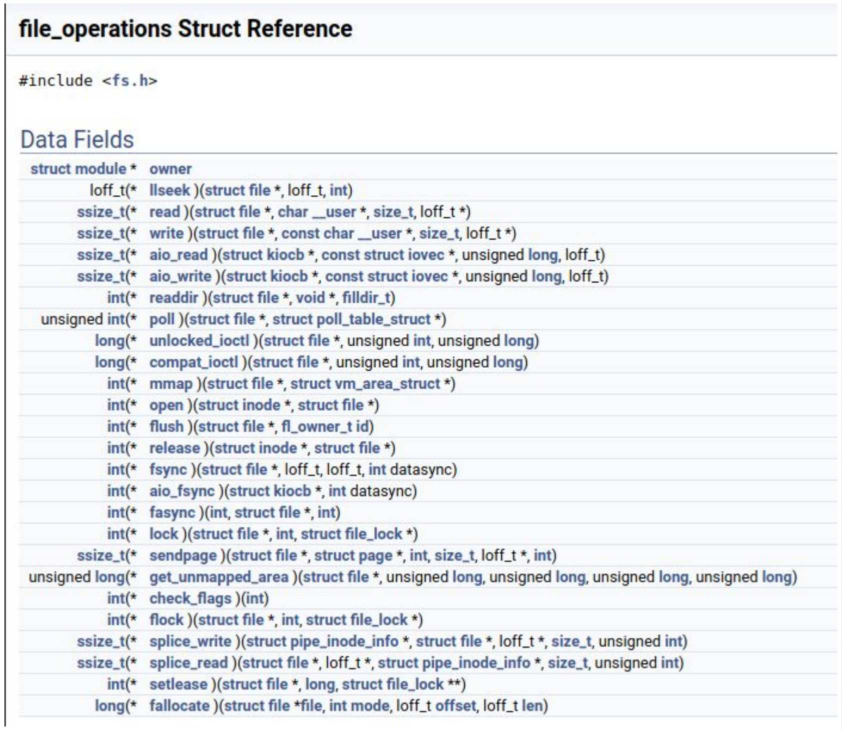
```
fd = open("/dev/crtp0", O_RDONLY)
if ( fd == -1 ) {
        printf("Error opening device");
        exit(EXIT_FAILURE);
}

char buf[1024];

int status = read( fd, &buf, 32 );
```

The I/O file abstraction

During the evolution of the linux kernel the structure has been provided with many more function pointers that respond to any possible need of user interaction.



file_operations Struct Reference

```
#include <fs.h>
```

Data Fields

struct module *	owner
loff_t(*	llseek)(struct file *, loff_t, int)
ssize_t(*	read)(struct file *, char __user *, size_t, loff_t *)
ssize_t(*	write)(struct file *, const char __user *, size_t, loff_t *)
ssize_t(*	aio_read)(struct kiocb *, const struct iovec *, unsigned long, loff_t)
ssize_t(*	aio_write)(struct kiocb *, const struct iovec *, unsigned long, loff_t)
int(*	readdir)(struct file *, void *, filldir_t)
unsigned int(*	poll)(struct file *, struct poll_table_struct *)
long(*	unlocked_ioctl)(struct file *, unsigned int, unsigned long)
long(*	compat_ioctl)(struct file *, unsigned int, unsigned long)
int(*	mmap)(struct file *, struct vm_area_struct *)
int(*	open)(struct inode *, struct file *)
int(*	flush)(struct file *, fl_owner_t id)
int(*	release)(struct inode *, struct file *)
int(*	fsync)(struct file *, loff_t, loff_t, int datasync)
int(*	aio_fsync)(struct kiocb *, int datasync)
int(*	fasync)(int, struct file *, int)
int(*	lock)(struct file *, int, struct file_lock *)
ssize_t(*	sendpage)(struct file *, struct page *, int, size_t, loff_t *, int)
unsigned long(*	get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long)
int(*	check_flags)(int)
int(*	flock)(struct file *, int, struct file_lock *)
ssize_t(*	splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int)
ssize_t(*	splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int)
int(*	setlease)(struct file *, long, struct file_lock **)
long(*	fallocate)(struct file *file, int mode, loff_t offset, loff_t len)

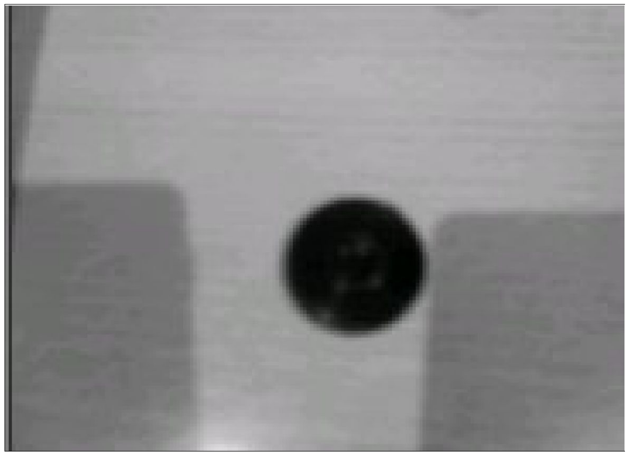
We will see these

- open()
  - to initialize the device;
- close()
  - call (* release) to release the device;
- read()
  - to get data from the device;
- write()
  - to send data to the device;
- ioctl()
  - explicitly call an internal software interrupt and trigger a callback passing a code to identify which request to handle.
- mmap()
  - request to directly map a portion of the device registers memory into userspace.

Video4Linux

An introductory application

- A Linux system to acquire WebCam images
- 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

- 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 of V4L2 is the availability of query operations for discovering the supported functionality of the device.
  - To adapt the great variety of camera devices

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

Querying the device

```
/* Step 1: Open the device */
   fd = open("/dev/video0", O_RDWR);

/* 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);
   }

/* 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;
       }
   }
```

NOTE:

YUYV is the format commonly available which uses 4 bytes to code 2 pixels with the format pattern: Yi, Cbi, Yi+1, Cri.

- Cb: Blue Difference Chrominance
- Cr: Red Difference Chrominance

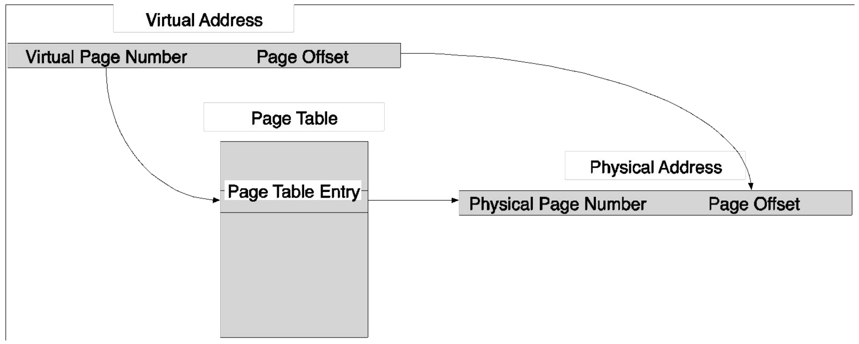
CODING EXAMPLE CODING EXAMPLE

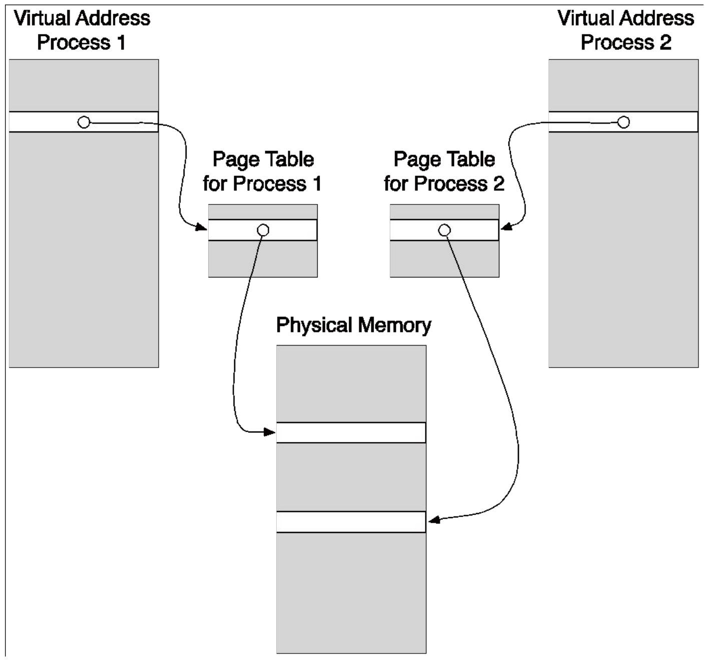
Double buffering

- 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

Virtual Memory (REPLAY)

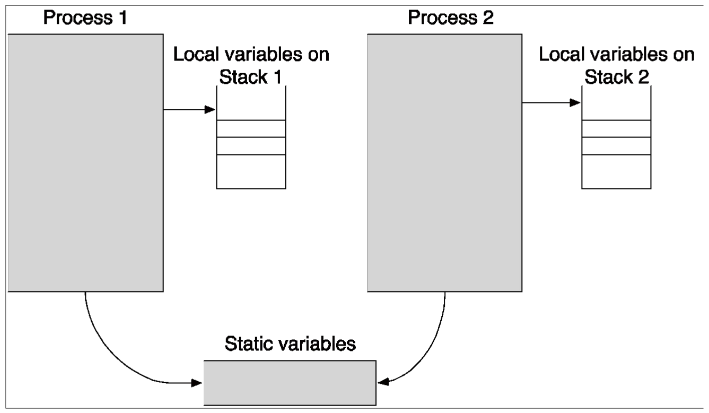
- Provides flexible mapping between the process address space and the physical address space
- Allows different processes run the same program
- May represent an overhead in real-time applications because the context switch required updating Page Table

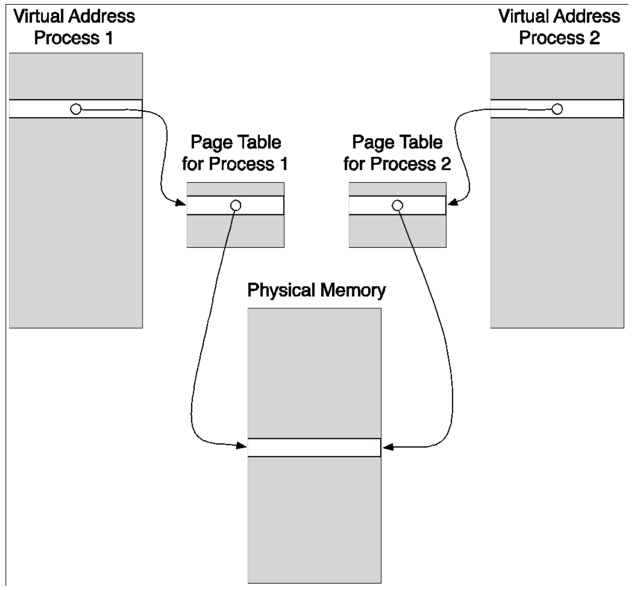




Virtual Memory (cont.)

- It is possible to let the same program be executed by different tasks even without virtual memory
  - In this case variables are held in the stack and static variables are shared
- Memory sharing can be achieved with virtual memory, too.





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