T4-Input/Output Management

SO-Grade 2013-2014 Q2

License

1.2

Outline

- I/O Management
  - Overview
  - Requirements
  - Physical devices, Logical devices, Virtual devices (file descriptors)
- Data structures
  - **i-node**. i-node table
  - **Open file's**. Open file's table
  - **File descriptor's**. File descriptor's table
- Basic System calls: open, read, write, close
- I/O for process communications: pipes
- Redirection

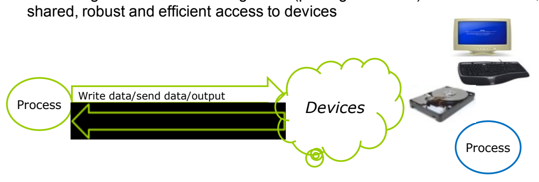
1.3

I/O MANAGEMENT

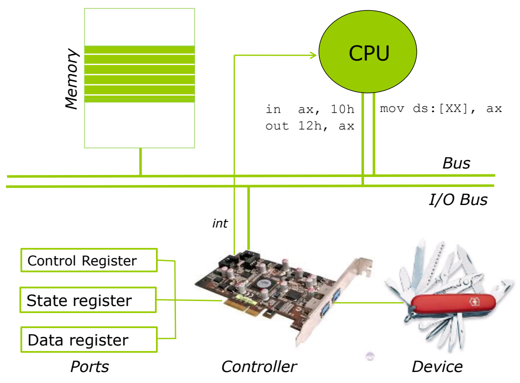
To provide a robust, usable and efficient device access

Input/Output manegement

- Data movement between processes and devices, always from the point of view of the process
  - Input: From the device to the process
  - Output: From the process to the device
- It is one of the key activities of processes
- I/O management: Device management (privileged access) to offer a usable, shared, robust and efficient access to devices



Accessing physical devices



```
in  ax, 10h
out 12h, ax
```

```
mov ds:[XX], ax
```

Too many device types

- To interact with users: display, keyboard, mouse
- To store data: hard disk, DVD,
- To transfer data: modem, network, WI-FI

- Too many possible characterizations

- Device management requirement design: to provide a standard API able to support very different types of devices and to include new ones (non existing yet)

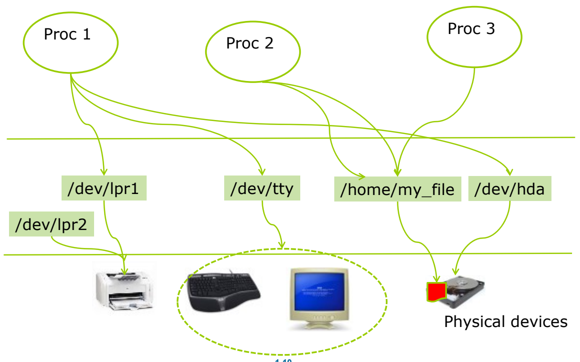
Goal: I/O operations design independent from device characteristics

Design requirements

- Uniform I/O operations: Same system calls for all the devices
- Virtual devices: Programs doesn't specify physical devices, only an abstraction provided by the O.S.
- It allows to connect virtual devices to different physical devices
  - The same program, using the same virtual device, can be accessing different physical devices
  - The same program, using different virtual devices, can be accessing the same physical device

```
% programa < disp1 > disp2
```

Virtual, logical and physical devices



Three levels of abstractions

- Physical, logical and virtual
- Each level has:
  - A set of functions that implements device operations (initialize, read, write, configure, etc)
  - A name space, a way to refer and identify the device (physical, logical or virtual)

Physical devices : operations

- Physical devices are managed by the kernel by defining a set of common operations that all the manufacturers has to provide: device drivers
- Device drivers code
  - Follows kernel specifications
  - Low level code to initialize, read, write, configure (etc) the device
  - Part of it is usually write in assembler
  - API is a superset, manufacturers can provide some of them and define the rest as NULL functions
- Device drivers code are part of the kernel code (executed in privileged mode)
  - To add a new driver we need to regenerate the kernel code
    - By recompiling the kernel (and then rebooting the system)
    - By using kernel provided tools to dynamically add code to the kernel without recompile and reboot (**kernel modules**)
      - We will see in the lab!!!!

Physical devices: names

- When adding a new device, system administrator has to specify (user root adds news devices):
  - Type: Character or Block device
  - Two numbers (physical device identifier): Major and Minor
    - Major (identifier): identifies the device. Must be unique per each type (character or block)
    - Minor: identifies a particular device of type major
- These two numbers are specified and reserved when inserting the device driver code into the kernel
- In case of physical devices, they are named by these two numbers

Logical devices

- Logical devices are abstractions of physical devices (the device driver must be previously inserted into the kernel)
- One logical device can represent
  - One physical device
  - Many physical devices combined emulating a new one
  - One physical device configured with specific case to facilitate usability
  - No physical devices, being a device implemented only in memory
- Operations to access logical devices have a common part, implemented in the kernel (by kernel programmers) and a specific part, implemented like calls to device driver specific operations
- Logical devices are offered to users (processes) as objects in the file system, that is, file names associated with physical devices (special files)
- Logical device names are the ones that users can see

1.14

Logical devices: names

- The system administrator creates new special files by associating a file name with physical devices (identified by a device type (block/Character), major, and minor)
  - Mknod command (or system call) creates a special file
  - For instance: mknod /dev/mydevice c X Y (where X Y are two numbers that corresponds to the major and minor of a valid character device)
- Once we have a file name, it can be accessed by users (processes) through commands or/and system calls
  - Open
  - Read
  - Write
  - Close
  - etc

1.15

Virtual devices (file descriptors)

- Processes ask to the kernel specific accesses to a file name (logical device)
  - With the system call open
  - Specific accesses can be: read/write/read and write
  - The kernel grants access to logical devices by creating one new (dynamic and local to the process) virtual device for each particular request
    - We will refer to virtual devices as *file descriptors (fd)*
  - File descriptors “names” are numbers (unsigned integers). Subsequent operations MUST refer to these numbers to access logical devices
    - The access mode is restricted to the one specified previously
- These three levels of abstractions provides great flexibility and usability of devices
  - One logical device can be accessed by many processes at the same time
  - One process can access one logical device with many file descriptors

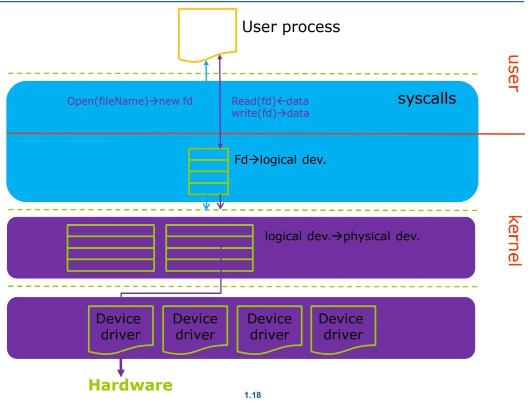
1.16

Standard file descriptors

- All the processes have created, by default, three file descriptors
  - 0 : standard input. Read access is allowed. By default it is associated with the console (stdin)
  - 1 : standard output. Write access is allowed. By default it is associated with the console (stdout)
  - 2 : standard error output. Write access is allowed. By default it is associated with the console (stderr)
- Processes usually use these tree standard file descriptors, we will see how to change the logical device associated with them (Redirection)

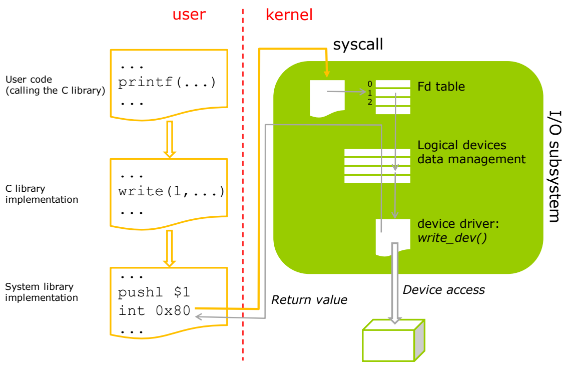
1.17

Accessing devices



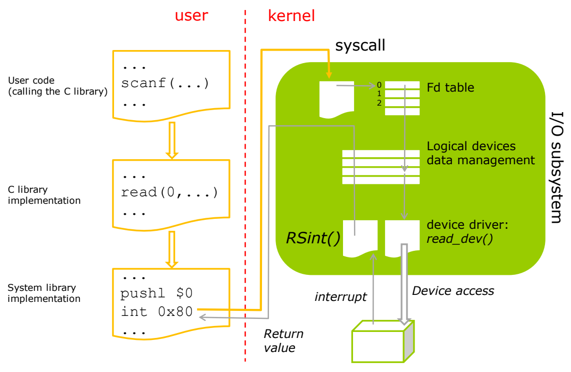
1.18

Example: Writing to a device



1.19

Example: Reading from a device



Accessing slow devices

- Devices have slow access times compared with CPU or memory → it is normal that processes have to wait for data
- Operations are classified in three types:
  - Synchronous with process blocking: They block processes if data transfer is not available at the moment
  - Synchronous without process blocking: The process never blocks
  - Asynchronous: The I/O operation is programmed and the process doesn't block

Synchronous with process blocking

- The process asks for a data transfer of N bytes (input or output)
  - If some data transfer is immediately available, data transfer is performed and the process returns to user mode
    - Not necessarily all the bytes requested by the user
  - If no data transfer is available (0 bytes availables), the process blocks
    - The process leaves the CPU and it goes to BLOCKED state
    - The scheduling policy is applied to select the next process to run
      - In case of round robin, the first process of the ready queue is selected
    - When the I/O is finished, the process is unblocked and it is moved to the ready queue
- The system call always returns the number of bytes transferred (read or write)

Synchronous without process blocking

- The process asks for a data transfer of N bytes (input or output)
  - The kernel performs the data transfer available and the process returns to user mode
  - Even though no data transfer is available
- The system call always returns the number of bytes transferred (read or write)

1.23

Asynchronous operations

- The process asks for a data transfer of N bytes (input or output) and it returns to user mode immediately
  - It is useful in those cases were data is not required at that moment, it's a pre-fetch of data
- Data transfer finalization is notified to the process in two ways:
  - By modifying a global variable of the process
  - With a software interrupt (similar to a signal processing)
- If data is required before the transfer finish, the process can execute a blocking system call to wait for data

- In our case, we will always use the first case: synchronous with process blocking

1.24

SOME DEVICE CHARACTERISTICS

1.25

Some device characteristics: console

- Console (/dev/tty in the labs)
  - It is a logical device that abstracts two physical devices together: keyboard and display
  - The standard input/output/output_error fd's of processes is associated with the console
  - Read: If there are characters in the keyboard buffer, it gets them, otherwise the process is blocked until the user press as many keys as requested and press enter (or ^D which represents the end of input data for the console)
  - Write: The message is written in the display without blocking
  - It is an ascii device, input data is received in ascii format and output must be in ascii format

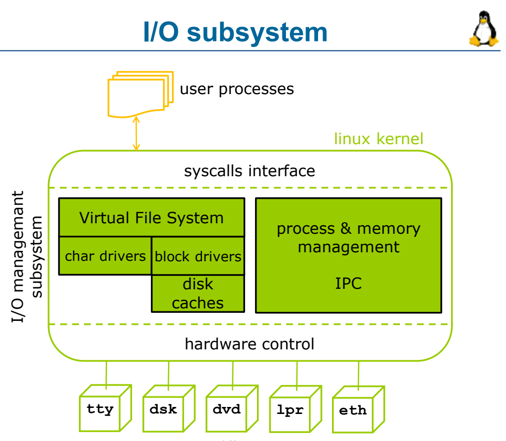
Some device characteristics: regular file

- Regular (ordinary) file
  - Logical device that represents a sequence of bytes stored at *disk*
  - This device offers sequential access, each open file has associated a current position in the file
  - Read: The process blocks if required
    - If current offset is equal to file size at the beginning of the read, the read returns 0
  - Write: The process blocks (or not) until the writing is finished
    - Only in case the disk is full the writing will return a number of bytes different from the requested ones

1. Main data structures
2. System calls

LINUX MANAGEMENT

I/O subsystem



1.29

i-node

- The i-node is the data structure that represents a logical device (which is the object managed by the kernel)
- It contains all the attributes of a logical device (not all of them are used by all the devices). Some basics are:
  - i-node identifier (i-node number)
  - User ID of the owner
  - Group
  - File size, if it is a regular file
  - Device ID
  - Some timestamps
  - Device driver operations (pointers to)
  - …
- It is to device management the same than the PCB to process management
- File name's are decouple from i-node data.
- Users uses file name's, which are more user friendly than i-node numbers.
  - We will see in File System management how it is defined this relationship

1.30

Accessing logical files

- Processes accesses logical devices (files) creating a new file descriptor that grants access to that logical file with specific access flags
- This is done through the open system call

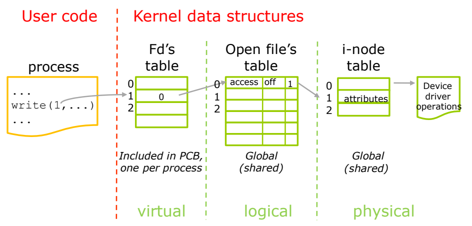
```
new_fd open(file_name, access_flags)
```

- Each open system call creates one new open_file object (a new data structure), with the dynamic link between inode number, access flags, and current offset
  - The offset is only used if the device provides a sequential access
- Open_file's are organized in an global open_file table (shared by all the processes)
- Each open also creates a new file descriptor that simply links with the newly created open_file object
  - This new level of abstratcion will provide processes with a high flexibility

1.31

Data structures

- 3 levels of data corresponding with the three levels of abstractions:
  - **File descriptors table**: Each process has N created file descriptors. The table is indexed by the FD number.
  - **Open file's table**: Files currently in use with specific access flags
    - Each open allocates one new open_file entry
  - **i-node table** : in memory copy of i-nodes currently in use



1.32

Data structures fields

- Each **i-node entry** of the i-node table has:
  - Number of references: defines how many open_file entries are linked with the i-node entry
  - Copy of i-node (in disk) attributes, that includes a link to i-node device driver operations
    - One i-node
- Each **open_file entry** of the open_file table has:
  - Number of references: defines how many file descriptors entries are linked with the open_file entry
  - Access mode: READ/WRITE/READ_WRITE
  - Current offset: 0..(*file_size*-1)
  - i-node entry number
- Each **fd entry** of the fd table has:
  - Open_file entry number

1.33

SYSTEM CALLS

1.36

Basic system calls

System call	description
Open	• Creates a new open_file that links logical device with specific access flags and offset. • It also creates a new fd (return value) that grants access the process to that open_file object
Read	Reads bytes from a fd to memory
Write	Write bytes from memory to fd
close	Removes the fd

1.37

Basic system calls for I/O:open

- `fd = open(file_name, flags);`
- It creates a new open_file object that grants access to that file with this specific access mode and offset.
  - The file MUST previously exist!!
- The open_file object is a kernel object, the process has access to it through a new file descriptor
- The open system call implements security checks, once created the fd, permissions are not validated again
- Following accesses will be performed using the fd
- Valid flags MUST include one of the following:
  - O_RDONLY: read access is requested
  - O_WRONLY: write access is requested
  - O_RDWR: read and write access is requested
- We will explain later!

1.38

Basic system calls for I/O:open

- Open (cont): Modifications in kernel data structures
  - If i-node is not in memory, the kernel loads it from disk and adds it to the in memory i-node table (references=1), otherwise references++
  - One new entry in the open_file's table is reserved and initialized
    - i-node entry points to the entry number of i-node table
    - Current offset set to zero
    - Access_mode set to acces_mode argument
    - References set to 1
  - One new fd entry (the first one available in sequential order) in the fd's table is reserved and initialized
    - Open_file entry set to the entry number of the open_file's table

1.39

Basic system calls for I/O:open

- Example: How are kernel data structures updated with these two system calls??

```
fd=open("file",O_WRONLY);
write(fd,"hola",4);
```

The file MUST previously exist??

- For special files, we must previously create them with the mknod command (or mknod system call)
- For regular files, we can create and open the file with the open system call with a different format

```
fd = open(file_name, flags, permissions);
```

- Flags can be bitwise-*or*'d in flags with one of the following:
  - O_CREAT: If the regular file doesn't exists, create it, otherwise it hasn't effect
  - O_TRUNC: truncates the content of the file, the file size is set to zero.
- Permissions specifies the permissions to use in case a new file is created

  Example:

```
fd = open("f1", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR );
If the file f1 doesn't exists, it is created with read and
write permissions for the owner. It opens it for writing.
If the file f1 exists, it just opens it for writing
```

Basic system calls for I/O:read

- `n = read(fd, buffer, count);`

- Asks the kernel to read count bytes from the device pointed by fd and to store them in the memory area pointed by buffer
  - At the position defined by the offset
  - Count is the desired maximum number of bytes to read, it usually matches the buffer size
- It could block the process depending on the device
- It always returns the number of bytes read:
  - If n>0 and n<= count, the read has been ok
  - If n==0 it means the end of the input data. That "situation" depends on the device
  - If n<0 some error has occurred
- The current offset of the open_file pointed by fd is advanced by n

Basic system calls for I/O: write

- `n = write(fd, buffer, count);`

- Asks the kernel to write count bytes from the memory area pointed by buffer to the device pointed by fd
    - At the position defined by the offset
- Count is the desired maximum number of bytes to write, it usually matches the buffer size
- It could block the process depending on the device
- It always returns the number of bytes written:
    - If n>0 and n<= count, the write has been ok
    - If n==0 indicates nothing was written
    - If n<0 some error has occurred
- The current offset of the open_file pointed by fd is advanced by n

1.43

Basic system calls for I/O:close

- It releases the fd entry from the fd table
- It propagates the close request to the open_file entry pointed by fd and, if required, to the i-node entry
    - It depends on the number of references of each entry

1.44

Basic system calls for I/O

- Example of reading a file and writing it to the standard output

1.45

I/O and process management

- The fd's table is included in the PCB, what happens at fork and execlp?
- Fork: The fd table of the child is a private copy of the fd table of the parent
  - It allows to share the open_file entry (including the current offset
  - References of the open_file entries must be updated
- Execlp: It doesn't affect the fd table

1.46

Example

- What happens internally when executing the following code?

```
fd=open("f1",o_rdonly);
fork();
fd1=open("f2",o_wronly);
```

1.47

I/O and signal management

- What happens if the process is blocked in a read or write and it receives a signal?
- It depends on the kernel
  - UNIX systems: The process unblocks, the signal is processed, and the system calls returns error (-1). The errno variable is set to -EINTR
    - Checking that value, the programmer can decide to retry the I/O operation or not

```
while (((n=read(...))== -1) && (errno == -EINTR));
```

  - Linux systems: The signal is processes and the I/O operation continues automatically
    - The behavior can be modified to act as a UNIX system with the siginterrupt system call
    - *For instance: siginterrupt(SIGUSR1, 1)*

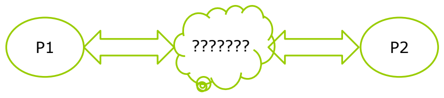
1.49

I/O FOR PROCESS COMMUNICATION

1.50

I/O for process communication

- How to proceed to read/write data from/to other processes?



- The kernel provides processes with a specific device for process communication: Pipes
- We can create a special file (FIFO) using mknod or we can create a temporal pipe

1.51

Pipes

- FIFO buffer where bytes are removed as soon as they are read (they cannot be read two times)
- It is designed for process communication. Processes must be in the same computer.
  - (Unnamed) Pipe. (It is a special case)
    - It's a temporal device without name in the system file.
    - It is created totally in memory and only accessible by the creator or by heritage (It doesn't have a name to refer to it)
    - Created with the pipe system call
  - Named Pipe: Special device with a name in the file system
    - Created with the mknod system call (or command)
    - Opened with the open system call

1.52

Pipes

- It is a blocking device
- Read: If some bytes are available, they are read (even partially). Otherwise (empty pipe), the process blocks until new bytes become available.
    - If there aren't writers, this is interpreted like the end of the communication and blocked processes (readers) will be unblocked
- Write: If the pipe is not full, the write is performed (even partially). Otherwise (full pipe), the process blocks until some space is available
    - If there aren't readers, it is is interpreted like the end of the communication and blocked processes (writers) will be unblocked. In that case, the kernel will send the SIGPIPE signal to readers

(unnamed) Pipes creation

```
int fd_pipe[2];
pipe(fd_pipe);
```

- It creates one new temporal device of type pipe. The i-node is only created in the i-node table, it doesn't exist in the file system
- It creates two new open_file entries in the open_file table, one with read access and the other with write access.
- It creates two new fd's, linked with these two new open_file entries. These two fd's are returned to the process in fd_pipe[0] and fd_pipe[1] respectively
    - fd_pipe[0] contains the fd number with read permissions
    - fd_pipe[1] contains the fd number with write permissions
- The kernel uses the number of "readers" and "writters" to interpret the end of the communication→ take care with unused fd's since they affect the references of open_file's entries.
- NOTE: Named pipes are accessed using open, refer to open to see data structures modifications

Redirection

- It consists of changing the open file linked with a specific file descriptor
- When running programs that use standard file descriptors, it is required a mechanism to change open files pointed by standard input, output and error output
- We can do it with the dup or dup2 system calls, they copy an existing fd entry in a new entry in the fd table
    - The "normal" usage is to duplicate a previously created fd in entries 0, 1 or 2

Redirection

- `newfd = dup(fd);`
  - It duplicates an fd entry (fd is the entry number) in the first fd entry available in the fd's table
  - The fd entry number allocated is returned
  - If ok, after the system call we will have two entries with the same information
- `newfd = dup2(fd, newfd);`
  - It duplicates an fd entry (fd is the entry number) in the entry number specified by newfd
  - If ok, after the system call we will have two entries with the same information
- In both system calls, the reference of the open_file pointed by fd must be updated

Processes in different computers: socket

- **Socket**
  - Logical device implemented only in memory (there isn't a physical device named socket)
  - It offers a FIFO buffer where bytes are removed as soon as they are read (they cannot be read two times)
  - It is designed for process communication. Processes CAN be at different computers with a network in the middle.
  - The behavior is very similar to a pipe but internal implementation is much more complicated (network protocols must be implemented)

Socket

- **Socket**: Example (*pseudo-code*)
  - Client
  - Server

```
                                    ...
                                    sfd = socket(...)
                                    bind(sfd, ...)
...                                 listen(sfd, ...)
sfd = socket(...)
connect(sfd, ...)                   nfd = accept(sfd, ...)

write/read(sfd, ...)                read/write(nfd, ...)
```

Modifying device and file characteristics

- It is possible to modify the default behavior of file descriptors and devices
- Fcntl: Manipulates file descriptor default characteristics
  - Example: close on exec flag. Specifies that fd must be closed if the process mutates
  - `int fcntl(fd, cmd, [, args]);`
- Ioctl: Manipulates the underlying device parameters of special files
  - Example: Echo or not echo for keyboard (canonical)
  - `int ioctl(fd, cmd [, ptr]);`