# T4-Input/Output

## License



## Contents

- Basic concepts of I/O
- Devices: virtual, logical, physical
- Linux I/O management
- Kernel data structures
- Basic system calls
- Examples
- File system
- Relationship between system calls and data structures



# BASICS CONCEPTS OF I/O

# What's I/O?

- **Definition**: information transfer between a process and the outside.
  - Data Input: from the outside to the process
  - Data Output: from the process to the outside

  (always from the process point of view)

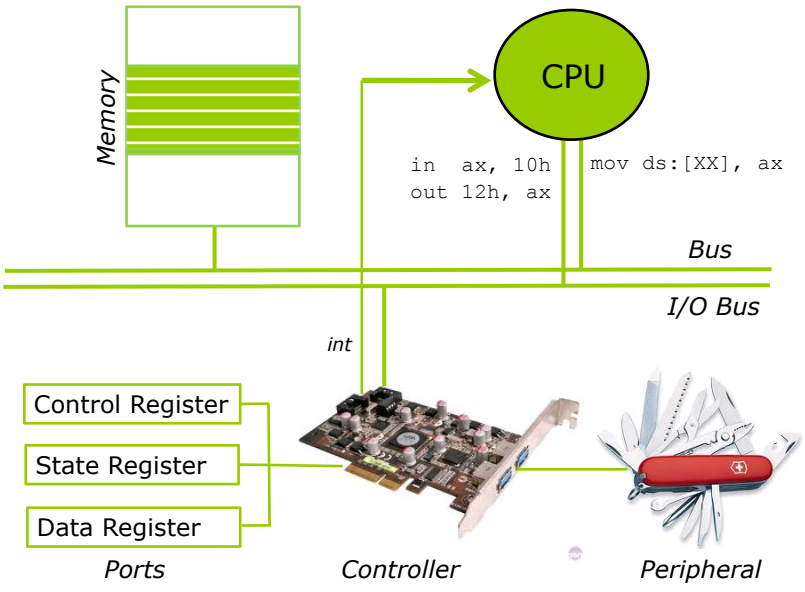
- In fact, basically, processes perform computation and/or I/O
- Sometimes, even, I/O is the main task of the process:
  for instance, web browsing, shell, word processor

- **I/O management**: Device (peripherals) management to offer an usable, shared, robust and efficient access to resources

1.5

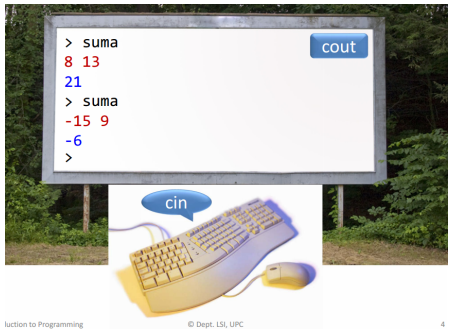
# I/O Devices

## HW view : Accessing physical devices




1.7

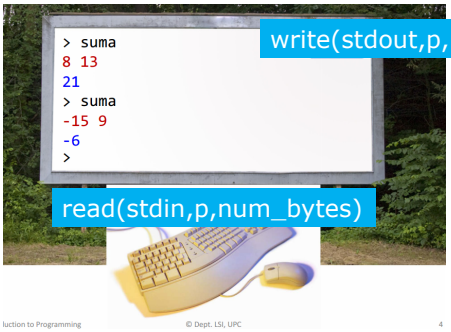
## User view (till now) (PRO1)




Input: **cin**. Read data, process them in order to adapt them to the data type
Output: **cout**. Process data and write them

1.8

## In this course we'll see what's in the middle




- No so easy as in C or C++
- Data conversion are up to the user (programmer) (that's the duty of C and C++ libraries)

1.9

# DEVICES: VIRTUAL, LOGICAL, PHYSICAL

1.10

## Device types

- To interact with users: display, keyboard, mouse. To store data: hard disk, Bluray, pendrive. To transfer data: modem, network, WI-FI or even more specialized (plane controllers, sensors, robots) … many possible characterizations
- Classification criteria:
  - Device type: logical, physical, network
  - Access speed: keyboard vs hard disk
  - Access flow: mouse (byte) vs DVD (block)
  - Access exclusivity : disk (shared) vs printer (dedicated)
  - And so on …
- Trade-off: standardization vs new device types

**CONCEPT: Device independence**

1.11

## Independence: principles of design

- The goal is that processes (code mainly) be independent of the device that is being accessed
- **Uniform** I/O operations
  - Access to any device with the **same system calls**
  - Increase portability and simplicity of user's processes
- Use of **virtual devices**
  - Process does not specifies the physical device, but it uses an identifier and a later translation.
- **Device redirection**: use different devices with no code changes
  - The Operating System allows a process to change the allocation of its virtual devices

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

1.12

# Independence: design principles

- Hence, usually, design in three levels: **virtual**, **logical** and **physical**
- First level gives us independence, the **process works on virtual devices** and does not need to know what's behind.
- The second level gives us **device sharing**. Different concurrent accesses to the same device.
- Therefore, it's possible to write programs performing I/O on (virtual) devices without specify which (logical) ones
- In execution time, process dynamically determines on which devices it is working. It can be a program argument or "inherited" from its parent.
- Third level separates operations (software) from implementation. This code is quite low-level, most of times in assembler.

1.13

# Virtual Device

- **Virtual level**: isolates user from the complexity of managing physical devices.
  - It sets correspondence between symbolic name (filename) and user application, using a virtual device.
    - A symbolic name is the representation inside of the Operating System
      - `/dev/dispX` or `.../dispX`
    - A virtual device represents a device in use by a process
      - Virtual Device = channel = file descriptor. It is a **number**
      - Processes have 3 standard file descriptors
        - Standard Input file descriptor 0 (stdin)
        - Standard Output file descriptor 1 (stdout)
        - Standard Error descriptor 2 (stderr)

  - I/O System calls use the identifier of the virtual device

1.14

## Logical Device

- **Logical level**:
  - It sets correspondence between virtual device and physical(?) device
  - It manages devices with or without physical representation
    - For instance, a virtual disk (on memory) or a null device
  - It deals with independent size data blocks
  - It brings a uniform interface to physical level
  - It offers shared access (concurrent) to physical devices that represents (if so)
  - In this level permissions, such as access rights, are checked.
  - Linux identifies a logical device with a file name

1.15

## Physical Device

- **Physical Level**: implements logical level operations in low-level.
  - Translates parameters from the logical level to specific parameters
    - For instance, on a hard disk, translates file offset to cylinder, platter, sector and track
  - Device initialization. Check if it is free, otherwise it enqueues a request
  - It performs the request programming operation
    - It could mean state checking, hard disk engine init, ...
  - Waits (or not) the operation ending
  - If successful, return the results or report any possible error
  - In Linux, a physical device is identified by three parameters:
    - Type: **Block/Character**
    - And two numbers: major/minor
      - **Major**: tells the kernel which family device to use (DD)
      - **Minor**: tells the kernel which one inside the family

1.16

## Device Drivers

- In order to offer independence, a set of operations is defined for all devices
  - It is a superset of operations that could be offered to access to a physical device.
  - Not all devices can offer all operations
  - In translation time (from virtual to logical) the available operations is set

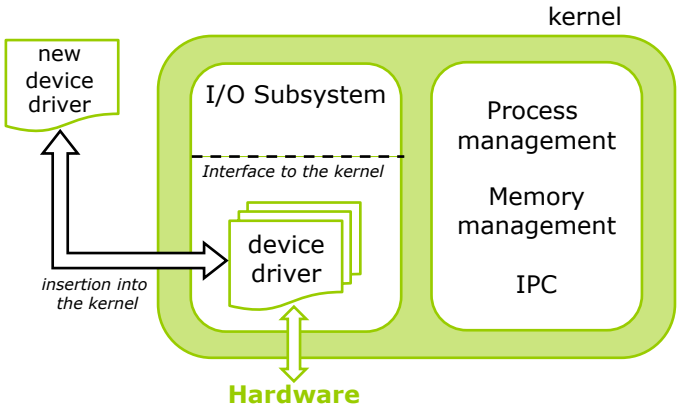


## Device Drivers

- OS programmers can't generate code for all devices, models, etc.
- Manufacturers should provide the low-level routines set that implements device functionality
  - The code + data to access device is known as Device Driver
  - It follows the interface specification of accessing to I/O operations defined by the OS
- To add a new device
  1. Option 1: with kernel recompilation
  2. Option 2: without kernel recompilation
     - OS must offer a mechanism to add kernel code/data dynamically
       - *Dynamic Kernel Module Support*, *Plug & Play*

# Device Driver

- Common operations (interface) are identified and specific differences are encapsulated inside OS modules: Device Driver.
  - Isolates the kernel from device management complexity
  - Protects the kernel from external code




1.19

# Device Driver

- **Device Driver (generic)**

Routines set that manages a device, allowing a program interact with the device.

- It follows the interface specification defined by the OS (*open*, *read*, *write*, ...)
  - Each OS defines its own interface
- It implements device dependent tasks
  - Each device performs specific tasks
- Usually, it contains low-level code
  - I/O ports access, interrupt management,...
- Functionalities are encapsulated in a binary file

1.20

## Dynamic insertion code: Linux modules

- Modern kernels offer mechanisms to add data+code to the kernel **without kernel recompilation**
  - Kernel recompilation could take hours ...
- Nowadays insertion is performed dynamically (at runtime)
  - *Dynamic Kernel **Module** Support* (linux) or *Plug&Play* (windows)
- Linux Module mechanism
  - File(s) compiled in a specific way that contain the data+code to be inserted into the kernel
  - A set of shell commands to load/unload/list modules
  - To be discussed in laboratory

1.21

## In the Lab…MyDriver1.c

- Device driver for a fake device. We install the driver using a module, so we don not need to recompile the kernel.

```
struct file_operations fops_driver_1 = {
                owner: THIS_MODULE,
                read: read_driver_1,
};
```

Operations provided by the driver

```
int read_driver_1 (struct file *f, char __user *buffer, size_t s, loff_t *off) {
…
return size;

}
```

In this case, only the read operation is implemented

```
static int __init driver1_init(void){
…
}
static void __exit driver1_exit(void){
…
}
```

Operations to load/unload driver into/from the kernel

```
module_init(driver1_init);
module_exit(driver1_exit);
```

Module operations to load/unload

1.22

# Device Driver (DD) in Linux: +details

- Internals of a Device Driver (DD)

  - **General information about the DD**: name, author, license, description, ...
  - **Implementation of the generic functions to access devices**
    - *open*, *read*, *write*, ...
  - Implementation of the specific functions to access devices
    - Device programming, access to ports, interruption management, …
  - Data struct with pointers to specific functions
  - **Init function**
    - Executed when the DD is installed
    - Registers the DD on the System, associating to a major
    - Associates the generic functions to the registered DD
  - **Exit function**
    - Unregisters the DD from the System and the associated functions

- Example of DD: see `myDriver1.c` and `myDriver2.c` in the lab docs

1.23

# I/O Linux Modules: +details

- Steps to create and start using a new device:

Phy.

- **Compile** DD, if needed, in kernel object format (.ko)
  - Type (block/character) and IDs are especified inside the code
- **Install** (insert) at runtime driver routines
  - insmod file_driver
  - The driver is related to a given major id

Log.

- **Create a logic device** (filename) and link it to the physical device
  - New file ←→block I character + major & minor
  - **Command** mknod
    - mknod /dev/mydisp c major minor

Virt.

- **Create the virtual device**
  - open("/dev/mydisp", ...);

1.24

## Examples of devices (1)

- Operation of some logical devices: terminal, file, pipe, socket

1. **Terminal**
   - Logical level object that represents a keyboard + screen set
   - "Usually" processes have it as data standard input and output
2. **Data file**
   - Logical level object that represents information stored on disk. It is interpreted as a sequence of bytes and the System deals with the offset inside the sequence.

1.25

## Examples of devices(2)

- **Pipe**
  - Logical level object that implements a temporary buffer operating as FIFO. Data written to a pipe disappear as are read them. It is used to unidirectional data flow among processes
    - **Unnamed** pipe**,** connects only processes with parenthood as it is accessible just by inheritance
    - **Named** pipe, allows connection to any process with enough permissions on the device
- **Socket**
  - Logical level object that implements a temporary buffer operating as FIFO. It is used to a two-way connection based byte streams between processes located in different hosts in a network.
  - Operation similar to pipes, although much more complex implementation depending on the semantics of communication

1.26

# LINUX I/O MANAGEMENT 



## Linux file types 

- In Linux, all devices are identified by a file (and there are several types of files)
  - block device
  - character device
  - directory
  - FIFO/pipe
  - symbolic link
  - **regular file → data files or "ordinary files"**
  - socket
- We called "special" files those which are not data files: pipes, links, etc.

## Creating files in Linux

- Almost all file types can be create with the syscall/command mknod. Exceptions:
  - Directories and soft links
  - Regular files. The mknod command (to be used in lab) **does not create data files**
- mknod name [c | b] major minor
  - name device name
  - c → character type devices, such as terminal and pseudo devices
  - b → If the device is a block type device such as a tape or disk drive which needs both cooked and raw special files, the type is b
  - p → FIFO (pipe, no major/minor is needed)
  - major identifying the class of the device.
  - minor identifying a specific instance of a device in that class

1.29

## File types

- Some examples:

| dev name | type | major | minor | description |
|---|---|---|---|---|
| /dev/fd0 | b | 2 | 0 | floppy disk |
| /dev/hda | b | 3 | 0 | first IDE disk |
| /dev/hda2 | b | 3 | 2 | second primary partition of first IDE disk |
| /dev/hdb | b | 3 | 64 | second IDE disk |
| /dev/tty0 | c | 3 | 0 | terminal |
| /dev/null | c | 1 | 3 | null device |

1.30

# KERNEL DATA STRUCTURES

1.31

## Kernel data structures: inode

- Data structure for storing file system metadata with pointers to its data. Each inode represents an individual file. It stores:
  - size
  - type
  - access permissions
  - owner and group
  - file access times
  - number of links (number of file names pointing to the inode)
  - pointers to data (multilevel indexation) → see below, at the end of this section
- All information about a file, except file names
- Stored on disk, but there is an in-core copy for access optimization

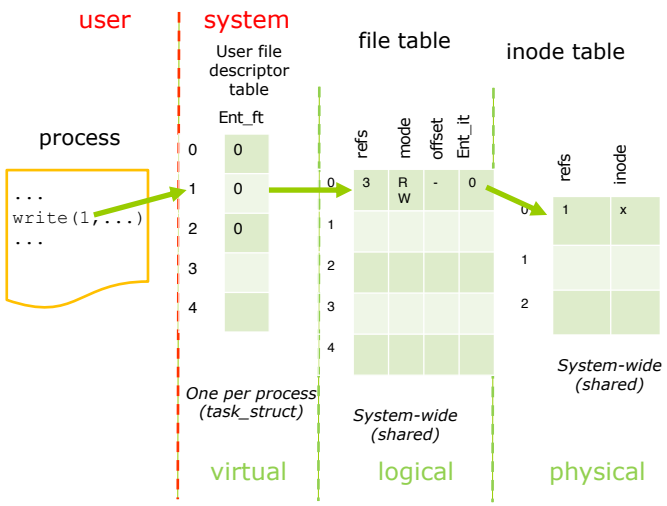
1.32

# Kernel data structures

- Each process
  - User Field Descriptor Table (FDT): per-process open-file table (saved in the task_struct, ie, PCB)
    - Records to which files the process is accessing
    - The file is accessed through the file descriptor, which is an index to the **FT**
    - Each file descriptor is a virtual device
    - Each field descriptor points to an entry in the Open File Table (FT)
    - Fields we'll assume: `num_entry_OFT`
- Global:
  - Open File Table (FT):
    - System-wide open-file management
    - One entry can be shared among several processes and one process can point to several entries.
    - One entry of FT points to one entry of the Inode Table (IT)
    - Fields we'll assume: `num_links, mode , offset, num_it_entry`
  - Inode Table (IT):
    - Active-inode table. One entry for each opened physical object. Including DD routines.
    - Memory (in-core) copy of the disk data for optimization purposes,
    - Fields we'll assume: `num_links, inode_data`

1.33

# Kernel data structures




1.34

# BASIC SYSTEM CALLS

1.35

## Blocking and non-blocking operations

- Some I/O operations are time consuming, a process cannot be idle in the CPU → OS blocks the process (RUN→BLOCKED)
- **Blocking** I/O: A process ask for a transfer of N bytes and waits for the call completes. Returns the number of bytes transferred.
  - If there are data available (even if the number of bytes is smaller than requested) transfer is made and process returns immediately
  - If there are not data available the **process is blocked**
    1. Process state changes from RUN to BLOCKED
       1. Process leaves CPU and is queued in a waiting processes list
       2. The first process from the READY queue is moved to RUN (if Round Robin)
    2. When data are available an interrupt arrives
       - The ISR (Interrupt Service Routine) sends data to the CPU and enqueue the blocked process in the READY list (if Round Robin)
    3. When the turn is over, the process will be put into RUN again

1.36

## Blocking and non-blocking operations (1)

- Non-blocking I/O operation
  - A process ask for a transfer of N bytes. Data available at that time are sent and immediately returns whether there are data or not.
  - Control returns immediately with the data have been transferred.



## BASIC I/O OPERATIONS

| Syscall | Description |
|---|---|
| open | Given a *pathname, flags* and *mode* returns an integer called the user *file descriptor* |
| read | **Reads N bytes from a device (**identified by the file descriptor) and saved in memory |
| write | Reads N bytes from memory and **writes them to the device** (identified by the file descriptor) |
| close | Releases the file descriptor and and leaves it free to be reused |
| dup/dup2 | Duplicates the file descriptor. Copies a file descriptor into the first free slot of the user file table. It increments the count of the corresponding file table entry, which now has one more fd entry that points to it. |
| pipe | Allows transfer of data between processes in a first-in-first-out manner |
| lseek | Changes the offset of a data file (an entry in the File Table pointed by the fd). |

Syscalls open, read & write are blocking

# Open

- So, how do you associate a name with a virtual device?
- **`fd = open(pathname, flags [, mode]);`**
  - **`open`** syscall links a device (file name) to a virtual device (field descriptor)
    - Is the first step that a process must take to access file data. It checks permissions. After correct completion, process can call *read/write* multiple times without check permissions again.
    - **`open`** returns a file descriptor. Other file operations, such as reading, writing, seeking and closing the file use the file descriptor.
  - **`pathname`** is a file name.
  - **`flags`** indicate the type of open. At least, one of them
    - `O_RDONLY` (reading)
    - `O_WRONLY` (writing)
    - `O_RDWR` (reading & writing)
  - **`mode`** gives the file permissions if the file is being created.

1.39

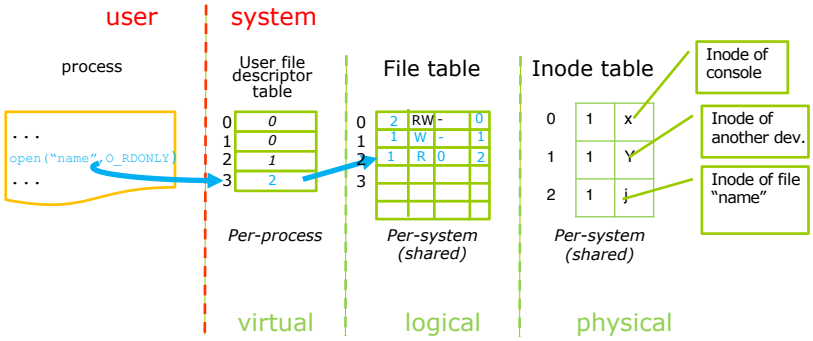
# Open: creating

- Regular files can be created with the system call `open` adding O_CREAT in the **`flags`** argument. The system call `mknod` creates special files.
- Argument **`mode`** is mandatory
  - It is an **OR** (|) among : S_IRWXU, S_IRUSR, S_IWUSR, etc.
- There is not a syscall for parcilly remove data in a file. However, file can be truncated to length 0 adding O_TRUNC in the **`flags`** argument.
- Examples:
  - Ex1: `open("X",O_RDWR|O_CREAT, S_IRUSR|S_IWUSR)` → If file X did not exist it's created, but otherwise `O_CREAT` has no effect.
  - Ex2: `open("X",O_RDWR|O_CREAT|O_TRUNC, S_IRWXU)` → If file X did not exist it's created, but otherwise file X is empty now.

1.40

# Open: data structure

- Open (cont): effects on the kernel data structures
  - The kernel allocates an entry in the file descriptor table. **It will always be the first free entry.** The kernel records the index of the File Table in this entry
  - The kernel **allocates an entry in the file table** for the open file. It contains a pointer to the in-core inode of the open file, and a field that indicates **the byte offset** in the file where the kernel expects the next read or write to begin.
  - The kernel associates these structures in the corresponding DD (`MAJOR` of the symbolic name). It may happen that different entries of the FT point to the same DD




1.41

# Read

Num. of bytes actually read

File descriptor returned by open

Address of a data structure in the user process

Num. of bytes the user wants to read

- `n = read(fd, buffer, count);`
  - Asks for reading *count* bytes (characters) from the device pointed by *fd*
    - If there is great or equal *count* bytes available, it reads *count bytes*
    - If there is less than *count* bytes, it reads all of them
    - If there is no bytes, it's up to the device behaviour:
      - Blocking process until data available
      - Returns 0 immediately
    - If *EOF*, returns 0 immediately
      - The meaning of *EOF* it's up to the device behaviour
  - Returns n, the number of bytes actually read
  - The kernel updates the offset in the file table to the n; consequently, successive reads of a file deliver the file data in sequence

1.42

## Write

Num. of bytes actually written

File descriptor returned by open

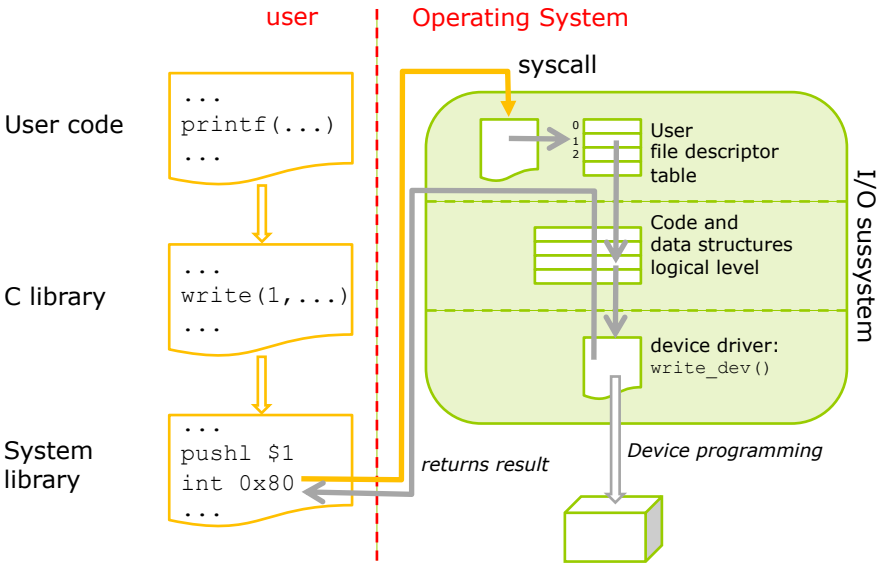
Address of a data structure in the user process

Num. of bytes the user wants to read

- `n = write(fd, buffer, count);`
  - Asks for writing *count* bytes (characters) to the device pointed by *fd*
    - If there is space on device for *count* bytes, it writes *count* (the kernel allocates a new block if the file does not contains a block that corresponds to the byte offset to be written)
    - If there is less, it writes what fits
    - If there is no space left on device, it's up to the device behaviour:
      - Blocking process until space available
      - Returns 0 immediately
  - Returns n, the number of bytes actually written
  - The kernel updates the offset in the file table to the n; consequently, successive writes of a file update the file data in sequence (when the write is complete, the kernel updates the file size entry in the inode if the file has grown larger)

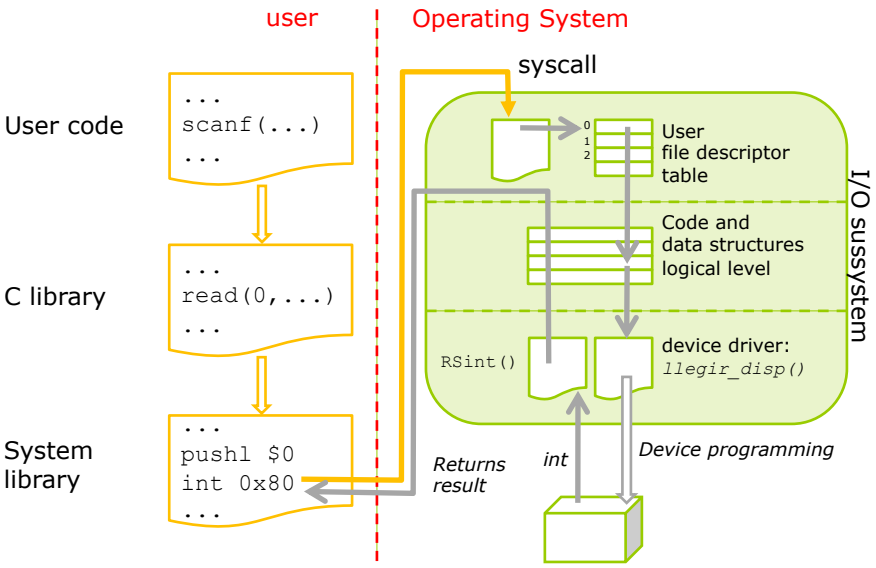
1.43

## Example: writing to a device




1.44

# Example: reading from a device






# Dup/dup2/close

- `newfd = dup(fd);`
  - Where *fd* is the file descriptor being duped and *newfd* is the new file descriptor that references the file.
  - Copies a file descriptor into the first free slot of the user file descriptor table.
  - Returns *newfd*
- `newfd = dup2(fd, newfd);`
  - Similar to *dup*, but the free slot is forced to be *newfd*
  - If *newfd* already refers to an open file, it is closed before duped
- `close(fd);`
  - Where *fd* is the file descriptor for the *open* file.
  - The kernel does the *close* operation by manipulating the file descriptor and the corresponding file table and inode table.
  - If the reference count of the file table entry is greater than 1 (*dup, fork*) then the kernel decrements the *count* and the *close* completes.
  - If the table reference count is 1, the kernels frees the entry and releases the in-core inode (If other processes still reference the inode, the kernel decrements the inode reference count but leaves it allocated).

# pipe

- Pipes allow transfer of data between processes in a first-in-first-out manner and they allow also synchronization of process execution.
  - `pipe(fd_vector); // Device for FIFO communications`
    - Creates an unnamed pipe. Returns 2 file descriptors `fd_vector[0]` for reading, `fd_vector[1]` for writing the pipe (and allocates corresponding File Table entries).
    - There is **no name** in the VFS, so there is no any call to `open`.
    - Only related processes, descendants of a processes that issued the pipe call can share access to unnamed pipes
  - Named pipes are identical, except for the way that a process initially accesses them
  - `mknod("my_pipe", S_IFIFO | 0600, 0);`
    - Creates a pipe, named "my_pipe", in the VFS and, hence, processes that are not closely related can communicate.
    - Processes use the `open` syscall for named pipes in the same way that they open regular files.
    - The kernel allocates 2 entries in the File Table and 1 in the Inode Table.

1.47

# pipe

- Usage
  - Processes use the `open` system call for named pipes, but the `pipe` system call to create unnamed pipes.
  - Afterwards processes use regular system calls for files, such as `read` and `write`, and `close` when manipulating pipes.
  - Pipes are bidirectional, but ideally each process uses it in just one direction. In this case the kernel manages synchronization of process execution.
- Blocking device:
  - Opening: a process that opens the named pipe for reading will sleep until another process opens the named pipe for writing, and vice versa.
  - Reading: if the pipe is empty, the process will typically sleep until another process writes data into the pipe.
  - If the count of writer processes drops to 0 and there are processes asleep waiting to read from the pipe, the kernel awakens them, and they return from their read calls without reading any data.
  - Writing: if a process writes a pipe and the pipe cannot hold all the data, the kernel marks the inode and goes to sleep waiting for data to drain from the pipe.
    - If there are no processes reading from the pipe, the processes that writes the pipe receives a signal SIGPIPE → the kernel awakens the sleeping processes
  - Processes should close all non-used files descriptors, otherwise -> Blocking!
- Data structures
  - 2 entries in the user File Descriptor Table (R/W)
  - 2 entries in the File Table (R/W)
  - 1 entry in the in-core Inode Table

1.48

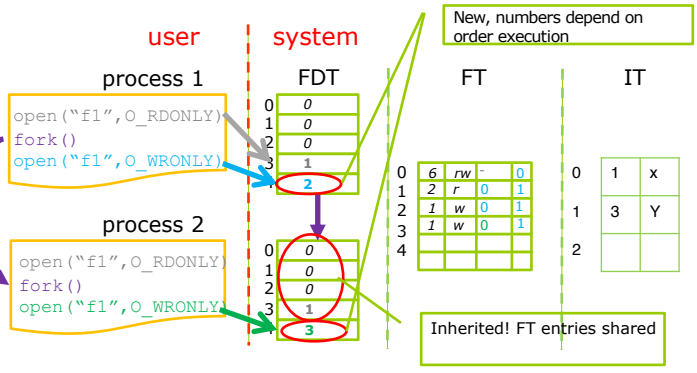
# lseek

- `lseek` changes the File Table byte offset (the read-write pointer) . It allows direct access by position in data files (or even sequential devices, like tapes).
  - Offset is 0 after an `open` system call (except with `APPEND` flag).
  - Offset is increased by `read` and `write` system calls.
  - Offset can be modified by the user with lseek system call
- `new = lseek(fildes, offset, origin)`
- The value of the pointer depends on `origin`:
  - `SEEK_SET`: pointer = `offset`. Set the pointer to offset bytes from the beginning of the file.
  - `SEEK_CUR`: pointer += `offset`. Increment the current value of the pointer by offset.
  - `SEEK_END`: pointer = file_size + `offset`. Set the pointer to the size of the file plus offset bytes.
  - `offset` can be negative.

1.49

# I/O and concurrent execution (1)

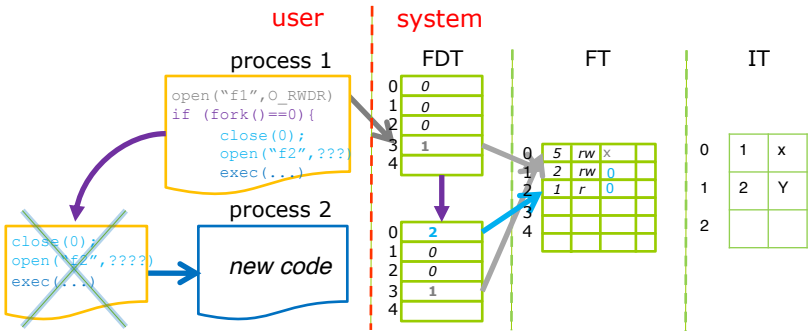
- I/O and *fork*
  - Child process inherits a **copy** of the parent process file descriptor table.
    - All open entries point to the same File Table entries
  - Parent and child sharing devices opened before *fork* system call
  - Next calls to open will be independent




1.50

# I/O and concurrent execution (2)

- I/O and *exec*
  - New process image **keeps** the same process' I/O internal structures
  - *fork+exec* allows I/O redirection before process image change






# I/O and concurrent execution (3)

- If a process is blocked in a I/O operation and it is interrupted by a signal there are two possible behaviours:
  - After the signal is handled, the kernel resumes the I/O operation (so the process remains blocked)
  - After the signal is handled, operation returns error and and sets `errno` to `EINTR`.
- Behaviour is up to
  - Signal programming:
    - If flag SA_RESTART in `sigaction` → operation is resumed
    - Otherwise → operation returns error
  - The operation itself (for instance, operations on sockets, on waiting signals, etc.)
- Example: how to protect a system call depending on behaviour:

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

## manipulating the underlying device parameters

- Although system calls are uniform, devices are not
- There are system calls to modified the specific characteristics of logical and virtual devices
- Logical device
  - `ioctl(fd, cmd [, ptr]);`
- Virtual device
  - `fcntl(fd, cmd, [, args]);`
- Arguments are quite generics to offer flexibility

1.53

## Characteristics of terminals

- Terminal
  - For each terminal the system has a buffer to keep characters typed in order to erase some of them, if necessary, before interpret them
  - POSIX defines special functionalities to some characters:
    - ^H: erases a character
    - ^U: erases a line
    - ^D: EOF (end-of-file) to indicate the end of input to the shell.
  - It could implement a writer buffer with cursor functionalities like “backwards n lines” or “forward in current line”
  - Each controller can implement a terminal characteristics as complex as it wants, for instance, it can modified text in the middle of the line.
  - Canonical/Non canonical: Character pre-processing or not before sending to the process (reading)

1.54

## Characteristics of terminals

- Terminal: operation (canonical by default)
  - Reading
    - **Buffer keeps chars until CR (carriage return) is pressed**
    - If there is a sleeping process waiting chars, it receives as chars as it can
    - Otherwise, chars are saved until a process ask for them
    - ^D means the end of the current reading, with the chars read since then, even if the buffer is empty.

```
while ( (n=read(0, &car, 1)) > 0 )
        write(1, &car, 1);
```

    - This is interpreted as end-of-file (EOF), by convention.
  - Writing
    - It writes blocks of chars
      - It could wait for the CR to be displayed on screen
    - Process is not blocked
  - Behaviour of devices, like blocking, can be modified by syscalls (`ioctl`)

1.55

## Characteristics of pipes

- **Pipe**
  - Reading
    - If there are data, a process reading from the pipe reads what it wants in a transient way
    - If there are no data, a process reading from the pipe is blocked until other process writes to the pipe.
    - If the pipe is empty and there are no writer processes (ie, all file descriptors opened for writing are now closed), reader process receives and EOF
    - Therefore, processes must close all unused file descriptors as soon as possible.
  - Writing
    - If there is room for the data to be written, the kernel writes the data (or as many as possible)
    - If the pipe is full, writer process goes to sleep waiting for data to drain from the pipe
    - If there are no more readers, writer process receives a signal *SIGPIPE*
  - Behaviour of pipes, like blocking, can be modified by syscalls (`fcntl`)

1.56

## Network devices

- Although network devices are I/O devices, network functionalities cannot be covered by the generic I/O operations
- Management of network devices is performed by different mechanisms
  - For instance, `/dev/eth0` has no inode, nor device driver
- There are specific system calls for networks and several interfaces
- They implement different network protocols (subject XC)

1.57

## Network devices

- **Socket**: operation
  - Mechanism, introduced by the BSD Unix, to provide common methods for interprocess communication and to allow use of network protocols.
  - Similar to pipes. It uses just one file descriptor for full duplex communication.
  - The socket system call establishes the end point of a communications link between two processes connected to the network. Each process executes `sd = socket(format, type, protocol);`
  - A process can ask for connection to a remote socket and detect if someone wants to connect to a local socket. The `send` and `recv` are socket specific system calls that transmit data over a connected socket. The `read` and `wite` syscalls are also valid.
  - The socket mechanism contains several system calls. They allow implement concurrent and, in general, distributed applications
  - In general, sockets use a client–server architecture

1.58

## Network devices

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

| Client | Server |
|---|---|
| `...` | `...` |
| `sfd = socket(...)` | `sfd = socket(...)` |
| | `bind(sfd, ...)` |
| | `listen(sfd, ...)` |
| `connect(sfd, ...)` | `nsfd = accept(sfd, ...)` |
| `write/read(sfd, ...)` | `read/write(nsfd, ...)` |

1.59

## EXAMPLES

1.60

# Byte-by-Byte access

- Reading from the standard input and writing to the standard output

```
while ((n = read(0, &c, 1)) > 0)
    write(1, &c, 1);
```

  - Note:
    - Reading while there are data (`n==0`), that's up to the device. The total amount of syscall depends on the number of bytes to be read
    - Processes conventionally have access to three files: its standard input (0), its standard output (1) and its standard error(2).
    - Processes executing at a terminal typically use the terminal for these three files.
    - But each may be "redirected" independently to any logical device that accepts the operations of reading and/or writing.
    - For instance:

```
#example1 →                    input=terminal, output=terminal
#example1 <disp1 →             input=disp1, output=terminal
#example1 <disp1  >disp2 →     input=disp1, output=disp2
```

1.61

# Buffer in user space access

- The same, but reading blocks if bytes (chars in this case)

```
char buf[SIZE];
...
while ((n = read(0, buf, SIZE)) > 0)
    write(1, buf, n);
```

  - Note:
    - You must write `n` bytes
      - Process is asking for `SIZE` bytes, however it reads `n` bytes
    - What about performance? How many system calls are executed?

1.62

## Data communication using pipes

- Program a process schema equivalent to the figure:


- 2 pipes
- P1 sends to pipe1 and receives from pipe2
- P2 the opposite symmetrically

```
void p1(int fdin,int fdout);
void p2(int fdin,int fdout);
```

```
1. int pipe1[2], pipe2[2],pidp1,pidp2;
2. pipe(pipe1);
3. pipe(pipe2);
4. pidp1=fork();
5. if (pidp1==0){
6.       close(pipe1[0]);
7.       close(pipe2[1]);
8.       p1(pipe2[0],pipe1[1]);
9.       exit(0);
10.}
11.close(pipe1[1]);
12.close(pipe2[0]);
13.pidp2=fork();
14.if (pidp2==0){
15.      p2(pipe1[0],pipe2[1]);
16.      exit(0);
17.}
18.close(pipe1[0]);close(pipe2[1]);
19.while(waitpid(-1,null,0)>0);
```

1.63

## Random access and size evaluation

- What does this code do?

```
fd = open("abc.txt", O_RDONLY);
while (read(fd, &c, 1) > 0) {
    write(1, &c, 1);
    lseek(fd, 4, SEEK_CUR);
}
```

*You can find this code at: ejemplo1.c*

- And this one?

```
fd = open("abc.txt", O_RDONLY);
size = lseek(fd, 0, SEEK_END);
printf("%d\n", size);
```

*You can find this code at : ejemplo2.c*

1.64

## pipes and blocking

```
int fd[2];
...
pipe(fd);
pid = fork();
if (pid == 0) {                              // child
    while (read(0, &c, 1) > 0) {
        // Reads, process and send data
        write(fd[1], &c, 1);
    }
}
else {                                       // parent
    while (read(fd[0], &c, 1) > 0) {
        // Receives, process and send data
        write(1, &c, 1);
    }
}
...
```

*This code is available at: pipe_basic.c*

- Be careful The parent process must close `fd[1]` if it does not want to be blocked!

1.65

## Sharing the read-writer pointer

- What does this code do?

```
...
fd = open("fitxer.txt", O_RDONLY);
pid = fork();
while ((n = read(fd, &car, 1)) > 0 )
       if (car == 'A') numA++;
sprintf(str, "El número d'As és %d\n", numA);
write(1, str, strlen(str));
...
```

*This code is available at: exemple1.c*

1.66

## Non shared read-write pointer

- What does this code do?

```
...
pid = fork();
fd = open("fitxer.txt", O_RDONLY);
while ((n = read(fd, &car, 1)) > 0 )
       if (car == 'A') numA++;
sprintf(str, "El número d'As és %d\n", numA);
write(1, str, strlen(str));
...
```

*This code is available at: exemple2.c*

1.67

## Redirection of standard input and output

- What does this code do?

```
...
pid = fork();
if ( pid == 0 ) {
    close(0);
    fd1 = open("/dev/disp1", O_RDONLY);
    close(1);
    fd2 = open("/dev/disp2", O_WRONLY);
    execv("programa", "programa", (char *)NULL);
}
...
```

1.68

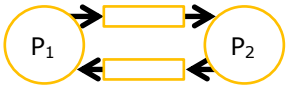
## Redirection and pipes

```
...
pipe(fd);
pid1 = fork();
if ( pid1 != 0 ) {                          // parent
    pid2 = fork();
    if ( pid2 != 0 ) {                      // parent
        close(fd[0]); close(fd[1]);
        while (1);
    }
    else {                                  // child 2
        close(0); dup(fd[0]);
        close(fd[0]); close(fd[1]);
        execlp("programa2", "programa2", NULL);
    }
}
else {                                      // child 1
        close(1); dup(fd[1]);
        close(fd[0]); close(fd[1]);
        execlp("programa1", "programa1", NULL);
}
```

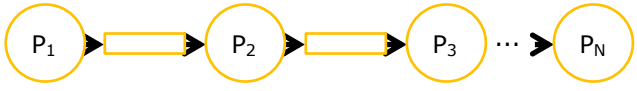
1.69

## Classroom exercises (1)

- Write a fragment of code that creates two processes $P_1$ and $P_2$, connect them by two pipes using the standard I/O file descriptors: in the first one $P_1$ writes and $P_2$ reads, in the second one $P_2$ writes and $P_1$ reads



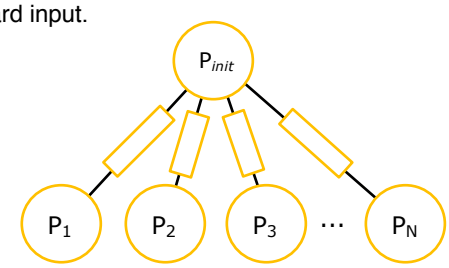

- Write a fragment of code that creates a sequence of N processes in chain: each process $P_i$ creates only one child $P_{i+1}$, until $P_N$. Each process communicates in chain with its parent and its child via a pipe in the standard I/O file descriptors, so that what the first process wrote arrives to the last process




1.70

## Classroom exercises (2)

- Write a fragment of code that creates N processes in sequence: the initial process creates all children, from $P_1$ until $P_N$. Each process communicates with the parent process via a pipe; the parent process must be able to write to all the pipes (using file descriptors 3..N+2) and children must read from its standard input.




1.71

# FILE SYSTEM

1.72

## File system tasks

- Duties of the file system regarding file management and data storage
  - Organize files in the system→ name space and directories schema
  - Guarantee correct access to files (access permissions)
  - Used/free space management (allocation/releasing) for data files
  - Search/storage data of files
- At all events, each file (of any kind) has a name and it must be stored and managed. File names are organized in directories.

1.73

## Name space: directories

- Directory: logical structure that organizes files.
- It is a kind of file (type directory) OS managed (users cannot open it or handled).
- It allows linking file name and the file attributes

In Linux, all this stuff is in the inode

- File attributes
  - File type: directory(d)/block(b)/character(c)/pipe(p)/link(l),socket(s), regular(-)
  - size
  - owner
  - permissions
  - file access times
  - ...
- Table of contents for the disk addresses of data in a file (in case of data files)

1.74

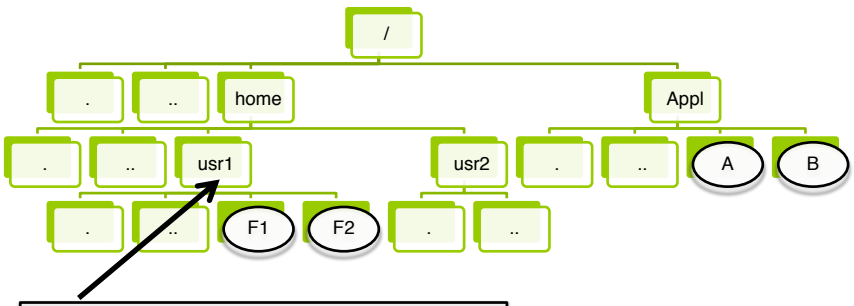
# Linux: user view

- Directories are organized in a hierarchical way (graph)
- Directories let users to classify their data
- The file system organizes the storage devices (each one with its own directory schema) in a single name space with an unique entry point
- The entry point is the root directory, that is, "/"
- Any directory has, at least, two (special) files (also directories)
  - **.** Dot: link to the current directory
  - **..** Dot-dot: link to the parent directory

1.75

# Linux: user view

- Each file can be searched in two ways:
  - Absolute path name: the path name starts with the slash character
  - Relative path name: starts from the current directory of the process




If you are here, you can search for f2 using:
Relative path name: f2
Absolute path name: /home/usr1/F2

1.76

# Linux: file names

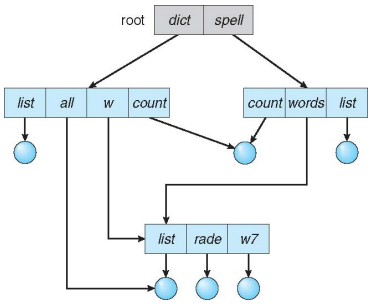
- File name is not among the file attributes (inode). Linux allows more than on file name with the same inode number.
- There two types of links between file name and inode.
  - Hard-link:
    - File name points to the inode that contains file attributes and data.
    - It's the most common
    - File attributes include a reference counter (how many file names for this inode).
      - Be careful: this ref. counter is different from the one stored in the in-core inode table!
    - Command line: `ln origin destination`
    - System call: `link(origin, destination);`
  - Soft-link
    - File name points to a inode that contains the path name of the destination file
    - It is not a regular file (l)
    - Command line: `ln -s origin destination`
    - System call: `symlink(origin, destination);`

1.77

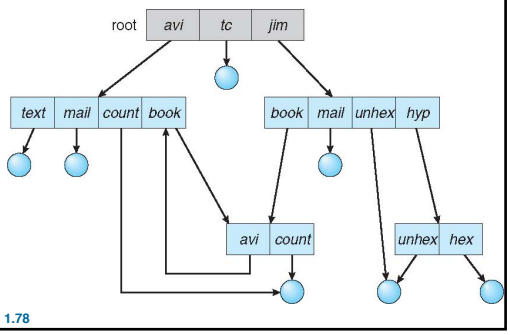
# Directory hierarchy implementation

- The existence of the two types of links influences the directory structure (graph)
  - No cycles are allowed with hard links
  - Cycles are allowed with soft links

**acyclic-graph**
FS checks that no cycles are created




**cyclic-graph**
FS checks for infinite loops




1.78

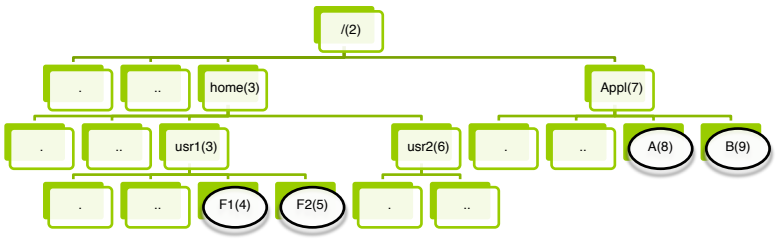
# Problems with directories in graph

- Backups
  - Do not backup a file twice
- Deleting files
  - Soft links
    - The kernel does not check for soft links to a file. In access time it will detect that the destination if file does not exist
  - Hard links
    - Inode is deleted when the reference count is zero

1.79

# Directories internal view

- A directory is a file whose data is a sequence of entries, each consisting of an inode number and the name of a file contained in the directory
- For instance:




| Nom | inodo |
|---|---|
| . | 2 |
| .. | 2 |
| home | 3 |
| Appl | 7 |

Directory /

Special case

| Nom | inodo |
|---|---|
| . | 2 |
| .. | 3 |
| F1 | 4 |
| F2 | 5 |

Directory /home/usr1

1.80

## File access permissions

- The FS lets assign different permissions to files
  - FS defines levels of access and operations that are allowed
- Linux:
  - Levels of access: owner, group, world
  - Operations: Read (r) , Write (w), Execute (x) → Be careful: this is not the flag access mode in the open() syscall!
  - Access permissions can be changed with a number in octal. Some typic values

| R | W | X | Valor numèric |
|---|---|---|---|
| 1 | 1 | 1 | 7 |
| 1 | 1 | 0 | 6 |
| 1 | 0 | 0 | 4 |

- See laboratory documentation

1.81

## System calls: name space and permissions

| Service | System call |
|---|---|
| Create / remove link to file/soft-link | link / unlink/symlink |
| Changing file permissions | chmod |
| Changing owner and group | chown / chgrp |
| query the status of files | stat, lstat, fstat |

- There are more system calls that allow processes to change :
  - Permissions
  - Characteristics
  - etc

1.82

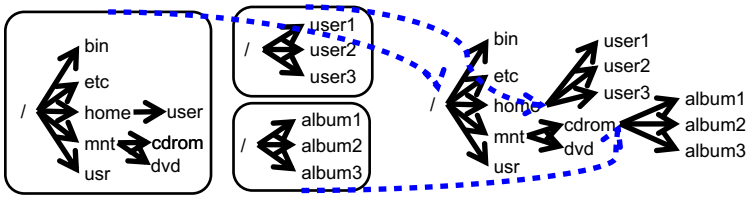
# Disk work units

- A storage device is divided in parts named sectors
- The OS allocation unit is the disk block (1 block corresponds to 1 or more sectors)
- Partition, volume or file system
  - a large **sequential** array of logical blocks, with an unique identifier (logical device). They are manged by the OS as a independent logical entity.
    - C:, D: (Windows); /dev/hda1, /dev/hda2 (Linux); /dev/disk1s1 (MacOS X)
  - Each partition has its own file system independent of the rest of partitions

1.83

# Accessing partitions

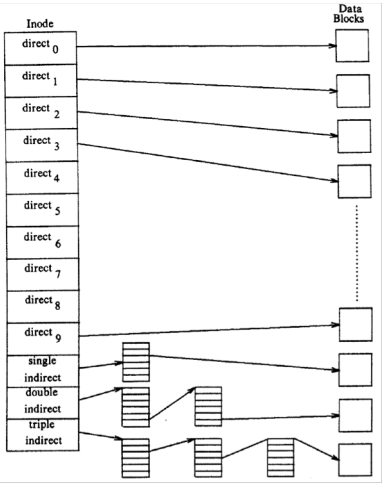
- A file system is accessible after execution of the mount system call or shell command (only root user can do that)
- Linux command line:
  - # mount –t ext2 /dev/hda1 /home
  - # umount /home
- Where “ext2” is the file system type, “dev/hda1” is the name of the device partition and “home” is the mount point, i.e., the location within the file structure where the file system is going to be attached.




1.84

# Managing used space

- For each file, OS must know where are located its blocks.
  - Indexed allocation: pointers to data blocks assigned to a file
  - Inode contains the indexes. How many?
    - multilevel index
- Indexes' blocks (B = 1KB, @ = 32b)
  - 10 direct blocks
    - ( 10 blocks = 10/KB)
  - 1 indirect block
    - (256 KB)
  - 1 double indirect block
    - (64 MB)
  - 1 triple indirect block
    - (16GB)
  - Classical AT&T System V UNIX approach (1983).




1.85

# Allocation of blocks

- Managing free space:
  - Lists of free disk blocks and free inodes
  - When the kernel wants to allocate a block from a file system it allocates the next available block in the list. Similar for inodes

1.86

## Metadata

- Persistent metadata: **stored on disk**
  - inodes and block list of the file
  - directories
  - a list of free blocks available on the file system
  - a list of free inodes in the file system,
  - … and any data that describes the filesystem such us: root inode, block size, etc.)
- Superblok: data block per partition that contains file system (redundant to be fault tolerant)
- Superblock is in memory in order to reduce disk I/O access. The kernel periodically writes the super block to disk if it had been modified so that it is consistent with the data in the file system.

1.87

## Memory and metadata

- Memory zone to save the last read inodes
- Memory zone to save the last read directories
- Buffer cache: memory zone to save the last read blocks
- Superblock: it is also in memory (in-core superblock)

1.88

# RELATIONSHIP BETWEEN SYSTEM CALLS AND DATA STRUCTURES



## Open

- The kernel searches the file system for the file name parameter and points to one entry in the the in-core inode table..
- Which inode? Read the directory entry. Two cases:
  - Directory is in the cache. Just access the directory in cache.
  - Directory is on disk. Read it and keep it in memory
    - Which blocks it must read? Check inode
    - Which inode? Check directory (repeat the algorithm)
- If the inode is found
  - The kernel checks permissions for opening the file and returns `errno` if applicable.
  - If it is a soft-link, it reads the inode path and searches for the file name (repeat algorithm)
  - The kernel allocates an entry in a private table in the process (user File Descriptor Table) and allocates an entry in the file table for the open file (File Table).
- If the inode is not found:
  - Error, except `O_CREAT`.
- At all events, `open` does **not** access to any data block of the file requested

## Open (cont.)

- If a process calls `open` to create a file (`O_CREAT`)
  - Get inode for the file name
    - Update list of free inodes in the Superblock
  - create new directory entry in the parent directory:
    - Directory data block: include new file name and newly assigned inode number;
    - Directory inode: update size

1.91

## Read

- Get File Table entry from the user file descriptor table
- Get inode from the File Table (lock inode)
- Get the offset from the File Table
- While count not satisfied
  - EOF?
  - calculate offset into block,
    - By dividing the value of the pointer by the block size, the kernel obtains the index of the first block to read
- If the blocks were not in memory, the kernel reads them from disk (and keep them in the buffer cache)

1.92

## Write

- Similar to that for reading a regular file
- In this case, if the file does not contain a block that corresponds to the byte offset to be written, the kernel allocates a new block
  - Superblock:
    - Update the list of free-block
  - Inode:
    - Assigns the block number to the correct position in the inode's table of contents
    - Update inode file size

1.93

## Close

- The kernel performs the close operation by manipulating the file descriptor and the corresponding file table and inode table entries.
- FDT: the user file descriptor table entry is empty.
- File Table
  - Decrements the file reference count. If zero, frees the entry and releases the in-core inode.
- Inode Table
  - Decrements the inode reference count. If zero, the inode is free for reallocation
- Inode:
  - Updates time stamps

1.94