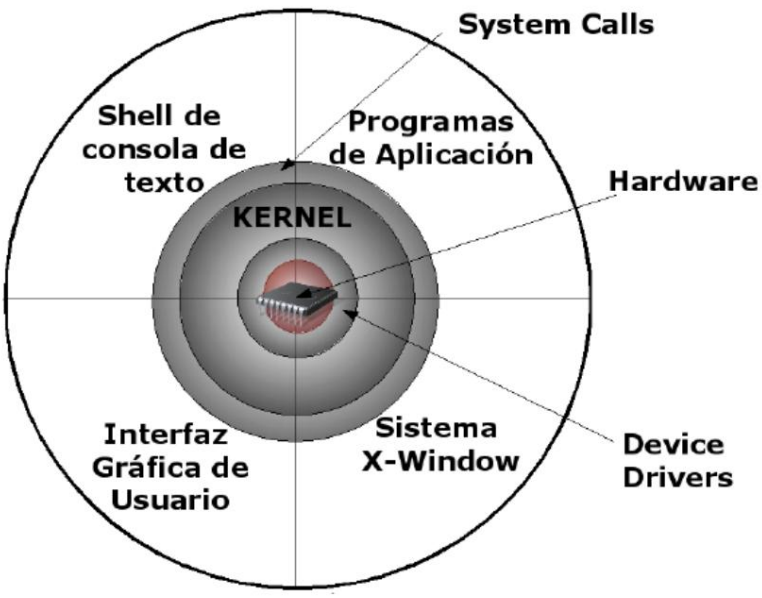
What is an operating system?: An operating system is a collection of programs that manage a computer's resources, providing users who may be logged into the system with the ability to use the system's hardware and software resources without needing to know any details about them and in a manner that is secure for themselves and others.

API (Application Programming Interface): A series of services or functions that the operating system offers to the programmer, allowing them to access the system's hardware resources. The implementation of APIs within the operating system is **known as system calls.**

Kernel: The set of programs that make up the operating system itself and the implementation of system calls is called the kernel. (Wiki NOTE: It is primarily responsible for providing different programs with secure access to the computer's hardware, or, more basically, it is in charge of managing resources. It is also responsible for deciding which program can use a hardware device and for how long.)

Shell: It is a user interface. The shell is a command interpreter that translates user requests and executes them by invoking the appropriate system calls. This interface is known as console mode. EXTRA: Every time we type a command, the shell interprets it as a binary executable or script. A binary executable is the result of editing, compiling, and linking a specific program. A script is a text file that contains binary commands or other scripts to be executed, and can also include flow control statements.

Device Drivers: To access hardware, the operating system has a very low-level interface called Device Drivers. These software components are part of the kernel and their mission is to access the system hardware directly. Applications CANNOT perform this access in modern operating systems, as they are multi-user environments in which the stability of the system and the applications running within it must be guaranteed.



File system: A file system is a set of policies defined for organizing information into files within a storage medium, allowing for the definition of their location within that medium and enabling simple access to the data contained within them. The file system manager is a piece of software in an operating system that implements the policies defined for the file system. Examples of file systems: hard drives, CDs, DVDs. The file system manager is responsible for organizing the sectors and tracks of disks, CDs, and so on into files, directories, links, etc., and for maintaining an up-to-date database of the sectors corresponding to each file, directory, link, etc.

Processes: A process is (trying to give a simplified definition) an instance of a program being executed. EXTRA: Processes are programs that are being executed at a given time. For example, the ls command. Several people can be executing ls simultaneously at any given time. Each instance of ls being executed is a different process. (The ps command displays the processes.) Every time we execute ls to view a directory, the system creates a process.

".h" files: These files contain definitions of functions, macros and variables.

Compilation: The compilation process involves four successive stages: preprocessing, compilation, assembly, and linking. To transform a human-written source program into an executable file, these four stages must be performed in succession.

1.ÿ **Preprocessing:** In this stage, the preprocessor directives are interpreted. Among other things, variables initialized with #define are replaced in the code by their value wherever their name appears. Preprocessing can be requested by running: gcc E circulo.c > circulo.pp and examined by running: more circulo.pp 2. **Compilation:** Compilation transforms the C code into the assembly language specific to our machine's processor. gcc S circulo.c performs the first two stages

by creating the file circulo.s; examining it with more circulo.s, the program can be seen in assembly language.

Assembly: Assembly transforms the program written in language 3.ÿ assembler to object code, a binary file in machine language executable by the processor. gcc c circle.c gcc c o circle.o circle.c gcc c circle.c o circle.o Either of these lines of code creates the object code file circle.o from circle.c (Perform the previous 2 steps first). The file type can be verified using the command $ file circle.o NOTE: In large programs, where many source files are written in C code, it is very common to compile each source file separately, and then link all the object modules created. These operations are automated by placing them in a file called a makefile, interpretable by the make command, which is responsible for making the minimum necessary updates whenever any portion of code is modified in any of

the source files.

It's **Linked:** The C/C++ functions included in our code, such as 4.ÿ Like printf() in the example, they are already compiled and assembled into existing libraries on the system. necessary to somehow incorporate the binary code for these functions into our executable (combine or link them). This is the linking stage, where one or more modules are assembled into object code with the code existing in the libraries. $ gcc or circle circle.o creates the circle executable from the object.

Library: A program library is nothing more than a file containing compiled and functional code and data, which will be incorporated into other programs when they require it. Code libraries facilitate code reuse, avoiding having to rewrite them every time we need them, and they also allow teamwork. A library is composed of: • Function prototypes (header file(s) .h) • Function definitions (source file(s) .c)

Types of libraries: 1. Static: These are simple collections of object programs grouped into a single file whose name typically ends in '.a'. When compiling a program that uses code contained in a static library, at the time of linking, the object code of the library is copied into our program, meaning that it is not necessary to distribute our program with the library since it is embedded in our new program. To generate a static library we use the ar utility. Before doing so, we must obviously have generated the object program, compiling with the c option. E.G.: $ gcc c holalib.c We create an object in that step..... $ ar rcs libhola.a holalib.o We create the library. Libraries generally have the prefix "lib" in their name. Therefore, for the purposes of the linker, the name is what follows "lib" and precedes ".a". **2.** Shared: These libraries are not linked to the program that calls the functions packaged within them. Instead, references are resolved at program startup time. If the necessary libraries are not already loaded into memory when the program is loaded, the dynamic loader will simultaneously load the program itself, which will compose the process triggered by the user, into memory, along with the libraries it needs. **3. Dynamically loaded (DL): These libraries are loaded at** different times during program loading and execution. Their main use is the implementation of modules or plugins, since these software elements are loaded when they are invoked during program execution. From a formatting perspective, they are no different in Linux than how shared libraries or object programs are built. However, there are differences in the code that must be written in the application to work with these libraries. This means that the application programmer must include specific functions that are not required in the models analyzed so far.

Basically we need to call four functions:

- dlopen(): Loads a library and prepares it for use.
- Dlerror (): Returns a string that describes the error generated by the other dynamic library management functions.
- Dlsym (): Searches for the value of a symbol present in a library already open with dlopen().
- dlclose(): Closes the library opened with dlopen().

Development of a program:

Make: make is a tool that allows you to run a sequence of processes.
It uses a script, commonly called a makefile. It is capable of determining automatically which steps in a sequence need to be repeated due to the change in some of the files involved in building an object, or in a operation, and which have not registered changes since the last time so it is not necessary to repeat them.

Make logic (according to program development): Each object program depends on its corresponding source program, and headers if any. A executable will depend on the object programs involved in your project, plus the libraries used in it.

We have three groups clearly differentiated by the colors of the arrows. When building a makefile, we need to have a clear dependency map.
Once the makefile is built and running, if we modify the bc file, regardless of the reason, make will detect the update, but instead of repeating all the compilations and links, it will only recompile the bc file, which will modify the bo object, and therefore it will have to relink the application to get to the executable version that contains the changes made in bc. The interesting thing is that it didn't touch the rest. That is, what doesn't change is left as is.
Previous Make example: _____

What we see at the beginning of the line as a name ending with ':' is called a rule.
If we don't specify anything and just type "make," make will assume that the rule to be executed is only the one on the first line with this characteristic. In our first example, executable.

If the rule following the ':' character has dependencies, these must correspond within the makefile to other rules written after the dependent rule. In our case, excecutable is dependent on ao and bo.
Therefore, there must necessarily be a rule for ao and another for bo written after the dependent rule. These lines are listed below with their respective dependencies.

Finally, we've written a rule that cleans up all files generated from the source code. This can be useful for general recompilation. It's executed by typing: "$ make clean" Ignoring the other rules and executing

only the clean rule. The general format of each rule is: **dependent:dependency command to generate the dependent from the dependency.**

Lines containing commands begin with a tab. Otherwise, it doesn't work.

Variables in make:

They are placed at the beginning of the code, equaling the variable constant to the command to be replaced, and the value is accessed by putting $ (VARIABLE).

Units basic:

- BIT[b]: Smallest unit of information, can be 0 or 1. • BYTE[B]: Set of 8 bits.

ÿ

The Von Newmann model: Proposes:

- Memory: Data and programs are stored in the same read-write memory.
  - ÿ The contents of this memory are addressed indicating their position regardless of their type.
- Execution: Sequential processing of instructions (unless otherwise indicated).
- Information: All this using Binary Data.

In the current version of stored-program computer architectures, they minimally meet the following characteristics: • Three hardware systems (interconnected by buses): ÿ Central Processing Unit (CPU). ÿ A main memory system. ÿ An input/output (I/O) system.

- Ability to perform processing sequentially. • A single path (physical or logical) between memory and CPU (bottleneck of the Von Newmann architecture).

Central Processing Unit (CPU): Its mission is to coordinate and control or perform all system operations. Its main parts are the following: • Control Unit: implements the state machine. • Arithmetic Logic Unit (ALU). • Central Memory (CM) or Registers (small memory area and the counter)

of program).

Main memory system: Stores data and programs.

Input/output system: Communication with the outside world.

The central processing unit, the main memory system and the input/output system are connected to each other by a system of interconnections (buses).

A typical complete cycle would be:

1) The control unit retrieves the “next” program instruction from the main memory (using the program counter: IP)
2) The instruction is decoded.
3) The operands following the instruction are taken from memory and placed in the records.
4) The ALU performs the requested operation and the result is placed in the registers or in memory.

SEE IMAGE BELOW.

A processor is a sequential machine that is in an infinite cycle like the one in the figure:

Decimal system:

- You need 10 symbols: (0 1 2 3 4 5 6 7 8 9).

• The value of the number is given by the sum of each digit multiplied by its "weight".

ÿ For example: 123.4=1 ÿ102 +2ÿ101 +3ÿ100 +4 ÿ10ÿ1 **Binary system:**

• I need 2 symbols: (0 1). • The value of the number is given by the sum of each digit multiplied by its "weight".

ÿ For example: 101102=1 ÿ2 ^{4}+0 ÿ2 ^{3}+1 ÿ2 ^{2}+1ÿ 2 ^{1}+0 ÿ2 0=16+0+4+2+0=2210

Octal system:

• I need 8 symbols (0 1 2 3 4 5 6 7). • The value of the number is given by the sum of each digit multiplied by its "weight".

ÿ For example: 3778=3 ÿ8 ^{2}+7 ÿ8 ^{1}+*7ÿ8* 0=192+56+7=255 -->(255=4ÿ8^{2} ÿ1)

Hexadecimal system:

• I need 8 symbols (0 1 2 3 4 5 6 7 8 9 A B C D E F). • The value of the number is given by the sum of each digit multiplied by its "weight".

ÿ For example: ABC8=10 ÿ162 +*11ÿ161* +12ÿ160=2560+176+12=274810

PASSAGES:

4410=1001002 :

The number to be converted is divided by the base to be converted until the quotient is less than the base.
The result is composed of the last quotient and the remainders taken in the reverse order of the quotient sequence.

In the case of decimals, multiply by the base and the integer part is the desired digit. 0.375x2=0.75 => 0.0
0.75x2=1.5

=> 0.01 0.5x2=1 => 0.011

2.37510=10.0112 **Binary to octal:**

Binary to hexa:

If I exceed the maximum number of digits that I can handle when adding, I will have problems, like this:
It is called “carry”:

- Carry Flag (CF) (Bit 0) Carry. Indicates an unsigned overflow with a 1. E.g., bytes 255 + 1 (the result is not in the range 0...255).
- Overflow Flag (OF) (Bit 11) Overflow. Indicates a signed overflow with a 1. E.g.: bytes 100 + 50 (result is not in the range of 128...127).

Adding and subtracting within the set of INTEGERS brings us a new problem to the
Time to do it on a PC: How to represent negative numbers? For this
We analyze 4 points:

- Sign and Magnitude

• 1's Complement • 2's Complement • Shifted Binary Sign and magnitude:

uses the most significant bit (MSB) to represent the sign and the remainder for the modulo. So, working with 4 bits, we have:

- 00102 is equal to 2. 10102
- is equal to 2.

Disadvantages of this method:

• 2 values are consumed to represent zero 0000b=1000b (0=0) • The same HARDWARE that adds positive numbers cannot be used to add negative numbers. 2+(1)=1 0010b+1001b=1011b (3)

• To perform an addition, first determine if the two numbers have the same value. sign. ÿ

Same sign: add significant part ÿ Different sign: subtract the greater from the smaller and assign the sign of the greater.

1's complement: To represent a negative number, each bit is inverted by its complement (1 to 0 and vice versa).

01112=710 1's complement: 1000b = 0111b = 7d (Same problem with “0”).

If a carry occurs at the end of the addition/subtraction, add it to the result obtained (end-around carry) e.g.:

The result is in one's complement, so if the answer is negative, the bits are inverted. When adding like signs, if the answer is opposite, then I went off scale (the last two cases).

Two's complement: To represent a negative number, each bit is inverted by its complement and 1 is added (C2=C1+1). 0111b=7d 0000b=0d now being negative the complement is done (taking into account C2 = C1 + 1) 1111b=0000b+1=1d 1000b=0111b+1=8d

No adjustment is required when adding two numbers with different signs. (The carry is not added, having

take into account again the sum of signs that corresponds to the previous case).

Shifted Binary: The absolute value of half the range minus 1 is added to the signed value. (I add half the signed absolute value to the value I want to represent; this absolute value is 2 (in binary) raised to the number of bits to be represented, and I subtract 1 from this value.) The result of the calculation is the value (which I have to convert to binary) that represents the number in shifted binary. E.g.:

Floating point: Represents any real number.

Examples

- **Example Wikipedia:**

Let's encode the decimal number -118,625 using the IEEE 754 system.

We need to get the sign, exponent and fraction.

Since it is a negative number, the sign bit is "1".

First, we write the number (unsigned) using binary notation. See the binary number system . to see how to do this. The result is 1110110.101.

Now, let's move the decimal point to the left, leaving only a 1 to its left.

1110110.101=1.110110101·26 This is a normalized floating point number.

The significant digit is the part to the right of the decimal point, padded with zeros until we have all 23 bits. That is, 11011010100000000000000.

The exponent is 6, but we need to convert it to binary and shift it (so that the most negative exponent is 0, and all exponents are just non-negative binary numbers). For IEEE format

754 of 32 bits, the offset is 127, so 6 + 127 = 133. In binary, this is written as 10000101.

Putting it all together:

```
 1          8                          23                    <-- size in bits
+-+--------+-----------------------+
|S| Exp |          |          Significant
|1|10000101|11011010100000000000000|
+-+--------+-----------------------+
31 30          23 22                                  0     <-- bit index (0 to the right)
      displaced +127
```

- **Example Utenians:**

Express the following numbers in Base 10 given in Single Precision Floating Point form.

And it gives you the following number: 35C1F.

Single Precision Floating Point, it is 4 bytes.

So the number is actually 35C1F000

35C1F000 converted to binary:

00110101111000001111110000000000

And now we divide into: 1 bit of sign, 8 of exponent, 23 of mantissa

0|01101011|10000011111000000000000

Sign: 0 -> +1
Exponent: 01101011 -> 107 -> we subtract the excess -> 107 - 127 = -20 -> as they are powers out of 2 we are left with -> 2^-20
Mantissa: 10000011111000000000000 -> 2^-1+2^-7+2^-8+2^-9+2^-10+2^-11 = 0.515136719 -> 1 is added -> 1.515136719

We multiply everything:

+1 * 2^-20 * 1.515136719 = 1.44494697 E -6

Express in single-precision floating point the following numbers expressed in base 10.

And it gives us this number: 165,625.

We have 165,625

We look for its sign: +1 -> 0

We transform the number without the sign to binary

165,625 -> 10100101,101

Exponent:
We move the comma until there is only a 1 on the left.

1.0100101101 -> 7 places -> Mantissa: 0100101101
Exponent: 7 -> Shifted by 127 -> 127+7 = 134 -> In binary: 10000110

We put together the number

0|10000110|0100101101 -> we fill with zeros up to 4 bytes

01000011001001011010000000000000 -> In hexa: 4325A000 -> 4325A

#define NAME value

enum: A data type or variable similar to #define. This is done by counting.

enum boolean {NO, YES }; // NOT=0 , YES =1.

enum months {January = 1 , February , March , April , May , June , July , August , September , October , November , December }; // February =2 , March = 3 , . . .

enum escapes {BELL = '\a' , TAB = '\t' , NVLIN = '\n'};

if inline: a = (i<0) ? 0 : 100;