



---

## Sistemas Operativos (81.510) Actividad Introductoria

---

**Dr. Adrián Sánchez-Carmona i Dr. Mariano Cabezas-Grebol**

Grau d'Enginyeria de Tecnologies i Serveis de Telecomunicació

Estudis d'Informàtica, Multimèdia i Telecomunicació

Universitat Oberta de Catalunya

Curso 2025-2026

---

# Introducción

## Presentación de la asignatura

La asignatura *Sistemas Operativos* (11.510) del Grado de Ingeniería de Tecnologías y Servicios de Telecomunicación es una asignatura obligatoria que se centra en los siguientes objetivos formativos:

- Conocer qué es un sistema operativo y los servicios que ofrece,
- Adquirir los conocimientos necesarios para acceder a los servicios del sistema operativo desde el intérprete de comandos,
- Conocer el concepto de proceso y los aspectos relacionados con la manipulación de procesos,
- Aprender los conceptos básicos de la gestión de la memoria,
- Conocer el concepto de dispositivo y los aspectos relacionados con la manipulación de estos,
- Ver el sistema de archivos y los aspectos relacionados con el uso de estos,
- Conocer los principios de comunicación y sincronización de procesos.

Para trabajar los conocimientos prácticos vinculados a la asignatura se utilizará el sistema operativo GNU/Linux y el lenguaje de programación C, que permitirá hacer llamadas a las bibliotecas del sistema para crear y gestionar procesos, memoria y archivos, entre otros.

Teniendo en cuenta estos objetivos y metodología docente, la asignatura está organizada en 4 PEC (Pruebas de Evaluación Continua) vinculadas a los diferentes contenidos teóricos y prácticos. Concretamente:

- Actividad previa: Presentación de la asignatura (esta actividad)
- PEC 1: El sistema operativo como máquina virtual, gestión y planificación de procesos
- PEC 2: Gestión de la memoria virtual, conceptos estructurales y funcionales de los sistemas operativos
- PEC 3: Entrada y salida, el sistema de archivos
- PEC 4: Gestión de procesos, comunicación y sincronización

El calendario de inicio y fin de cada una de estas actividades, así como su peso respecto a la EC, se muestran a continuación:

	<b>Act. Intro.</b>	<b>PEC1</b>	<b>PEC2</b>	<b>PEC3</b>	<b>PEC4</b>
<b>Inicio</b>	25/09	08/10	29/10	19/11	10/12
<b>Fin</b>	07/10	28/10	18/11	09/12	05/01
<b>Solución</b>	13/10	03/11	24/11	15/12	07/01
<b>Nota</b>	18/10	08/11	29/11	20/12	10/01
<b>Peso</b>	0%	25%	25%	25%	25%

## Modelo de evaluación de la asignatura

El modelo de evaluación de la asignatura Sistemas Operativos es EC+PS. Por tanto, la asignatura se supera a través de la realización de las actividades de evaluación continua (EC), que tienen un peso del 60% sobre la nota final de la asignatura, y la realización de una prueba de síntesis (PS), que tiene un peso del 40% sobre la nota final de la asignatura. De cara a la evaluación también hay que tener en cuenta:

- Las actividades de evaluación continua (EC) constan de cuatro PEC.
- Para determinar la nota de EC se lleva a cabo la siguiente ponderación de las actividades:  $EC = PEC1 [25\%] + PEC2 [25\%] + PEC3 [25\%] + PEC4 [25\%]$ .
- Para poder superar la asignatura es necesario entregar todas las actividades de EC y obtener una nota mínima de 4 en la PS.

De cara a la realización de las actividades de evaluación continua, se debe tener en cuenta que la entrega fuera de plazo sin justificación siempre que sea antes de la fecha de publicación de las notas, implicará una penalización de medio punto (0.5) por cada día de retraso. En el caso de que el retraso esté debidamente justificado y se pueda prever (por ejemplo, enfermedad), se deberá contactar con el profesor colaborador responsable de la actividad afectada y entregar un documento justificativo para evitar la penalización. No se aceptará la entrega de ninguna actividad una vez pasada la fecha de publicación de la solución correspondiente.

Por último, la falta de originalidad en la autoría o el mal uso de las condiciones en las que se realiza la evaluación de la asignatura supondrá una calificación de suspenso (D/0) de la actividad evaluable afectada. En caso de que la infracción se realice de manera repetida, se procederá al suspenso directo de la asignatura y se reportará el caso a las autoridades competentes.

## Presentación de la actividad

El objetivo de esta actividad es instalar el sistema operativo GNU/Linux y las herramientas para poder compilar y ejecutar un programa escrito en *C* a partir de su código fuente.

Para ello comenzaremos por descargar una imagen del sistema operativo GNU/Linux e instalarla en una máquina virtual. A continuación, instalaremos `git`, un sistema de control de versiones ampliamente utilizado y que nos permitirá trabajar con el código fuente, y `gcc`, el compilador que se encarga de generar el archivo binario que se ejecuta en el procesador a partir del código fuente. Finalmente, detallaremos el proceso de compilación de un programa con `gcc`. Al final de este documento planteamos un conjunto de problemas para practicar.

## Criterios de valoración de la actividad

Si bien el seguimiento de esta actividad no es obligatorio, el objetivo no es otro que garantizar que los estudiantes tienen un sistema listo para trabajar y tienen los conocimientos mínimos de programación para afrontar con éxito la parte práctica de la asignatura. El tiempo estimado para la realización de esta actividad es de 15 horas, por lo que en caso de encontrar dificultad en el seguimiento se recomienda contactar con el profesor para detectar la fuente del problema y buscar una solución.

## Formato y fecha de entrega de la actividad

De manera opcional se puede entregar un documento PDF con la respuesta a las preguntas planteadas. En caso de que sea necesario se deberán entregar **todos** los archivos fuente programados. En este caso, la entrega se hará en un único archivo ZIP que contendrá el documento PDF y los archivos con código fuente. Es necesario que todas las páginas estén numeradas y que en todas aparezca el nombre del estudiante. En ningún caso se aceptarán archivos con otro formato a los indicados en este punto.

La fecha límite de entrega a través del REC (Registro de Evaluación Continua) es el **07 de octubre de 2025 a las 23:59h**. En ningún caso se aceptarán entregas fuera de plazo (si no hay un motivo debidamente justificado) ni entregas a través del correo electrónico, ni entregas de informes que no sean en formato PDF y con portada.

---

# Instalación del sistema operativo

## GNU/Linux

Como se ha introducido anteriormente, en esta asignatura utilizaremos el sistema operativo GNU/Linux (distribución Ubuntu 20.04 LTS) como herramienta de trabajo. Así pues, el primer paso es instalarlo.

Para ello existen diferentes alternativas en función del sistema/entorno de trabajo que tengáis. A continuación se explican dos métodos para instalar el sistema operativo GNU/Linux en un ordenador con sistema operativo Windows 10.

### Instalación GNU/Linux en el entorno WSL (Windows Subsystem for Linux)

Para los usuarios de Windows 10 (64 bits) existe la posibilidad de instalar el sistema operativo GNU/Linux bajo el entorno Windows Subsystem for Linux (WSL), que es una capa de compatibilidad para poder ejecutar binarios de Linux de forma nativa.

Podemos habilitar el entorno WSL desde el entorno de configuración del sistema. Una forma fácil es buscar **Activar o desactivar las características de Windows** para que aparezca la ventana de **Características de Windows**, donde podréis habilitar la opción correspondiente, tal como se muestra en la Figura 1. En cuanto se hayan instalado los ficheros, deberá reiniciarse el sistema para que se habilite la opción.

Otra forma de hacerlo es a través del entorno PowerShell ejecutando el siguiente comando como administrador, tal como se muestra en la Figura 2.

```
1 dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all  
  ↪ /norestart
```

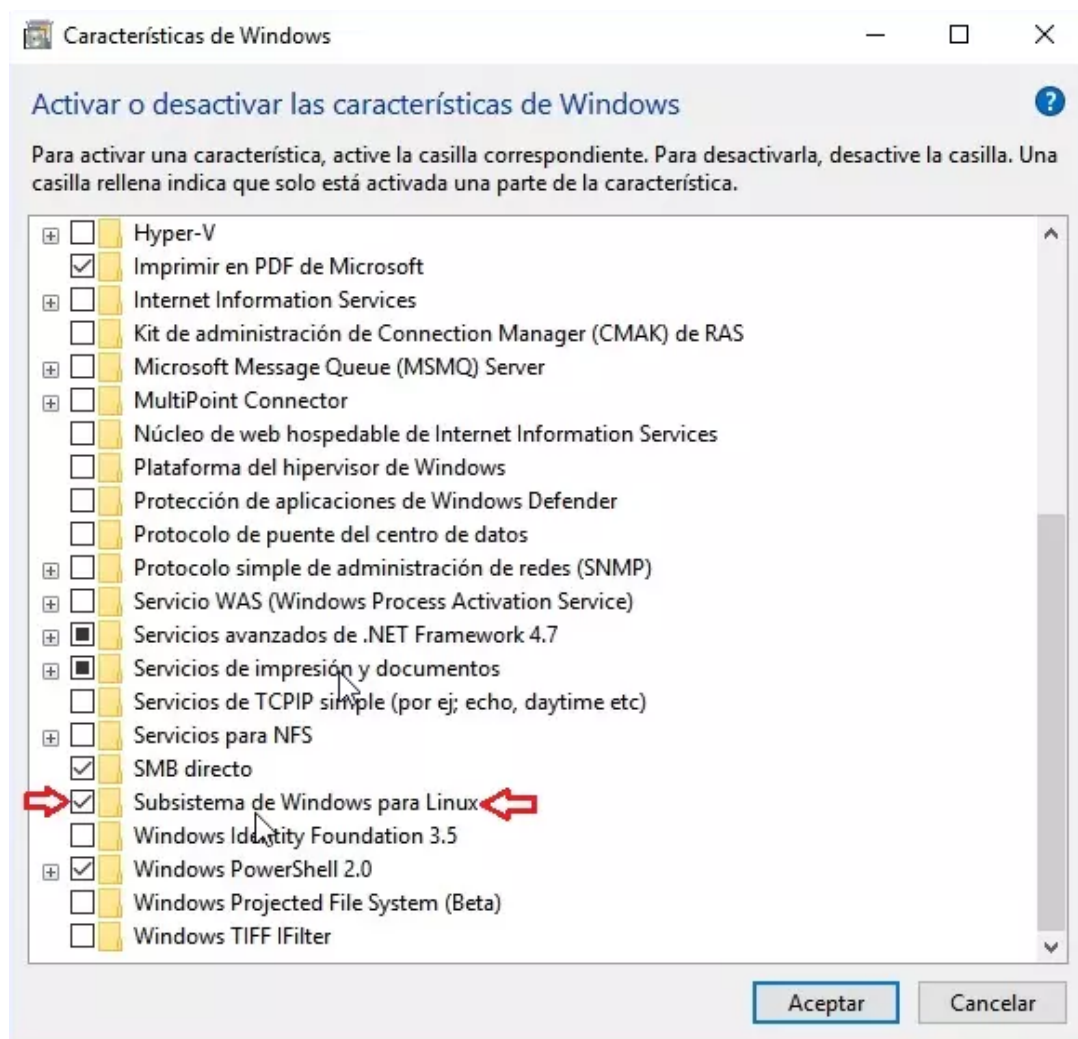


Figura 1: Activación WSL.

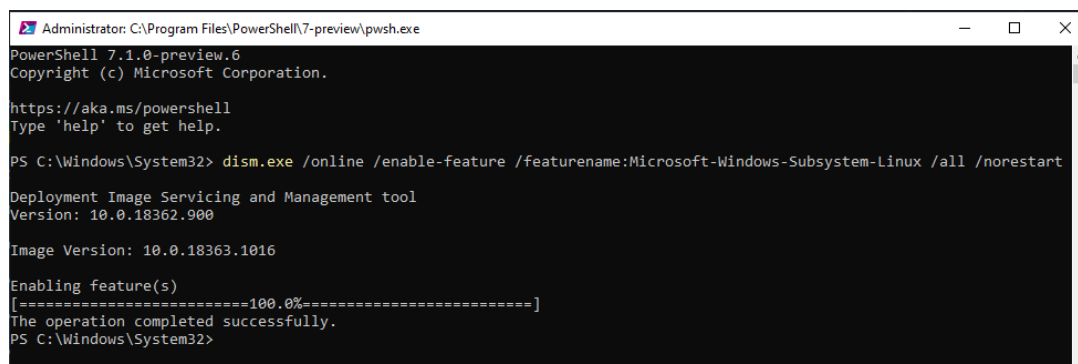
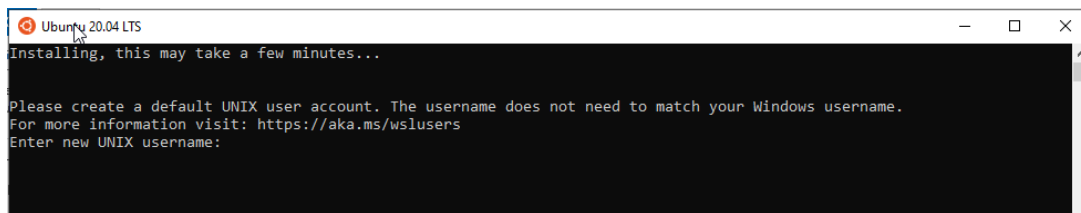


Figura 2: Activación WSL a través de PowerShell como administrador.

En cuanto se hayan instalado los ficheros, deberá reiniciarse el sistema para que se habilite la opción. Una vez reiniciado el sistema debéis ir a la *Microsoft Store* y buscar e instalar la distribución de GNU/Linux Ubuntu 20.04 LTS. Alternativamente, podéis instalarlo directamente haciendo *click* al siguiente [enlace](#). Después de la instalación, que dura unos minutos, os pedirá un nombre de usuario y contraseña, como podéis ver en la Figura 3.



**Figura 3:** Activación de vuestro usuario en el terminal GNU/Linux.

Una vez instalado WSL (versión 1) es posible actualizar a WSL (versión 2). A pesar de que es un paso opcional, se recomienda hacerlo pues esta nueva versión ofrece más prestaciones y mejor rendimiento. Para ello debemos asegurarnos que disponemos de Windows 10 actualizado a versión 1903 o superior, Build 18362 o superior para sistemas x64. Podéis verificar la versión de vuestro sistema desde un entorno PowerShell ejecutando el siguiente comando:

```
1 winver
```

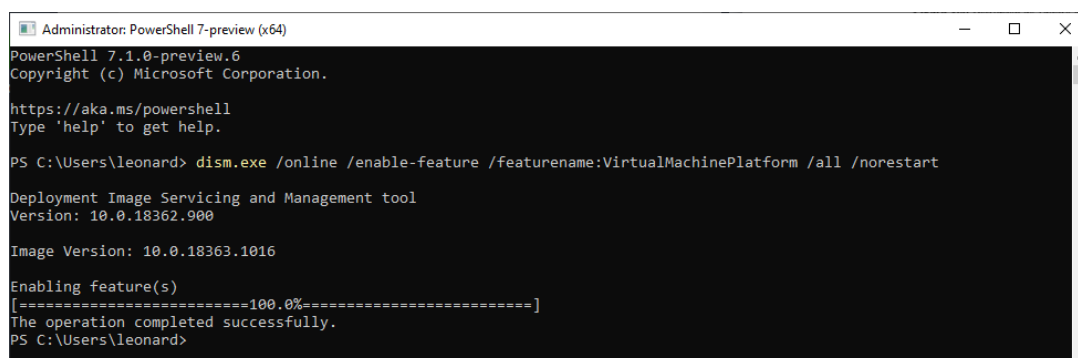
Si cumplís los requisitos, a continuación debéis habilitar el componente opcional de *Virtual Machine Platform* ejecutando el siguiente comando, tal como podéis ver en la Figura 4.

```
1 dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

A continuación deberéis reiniciar vuestro sistema. Para terminar debéis configurar WSL2 como opción por defecto. Para ello deberéis abrir un entorno PowerShell y ejecutar el siguiente comando:

```
1 wsl --set-default-version 2
```





```

Administrator: PowerShell 7-preview (x64)
PowerShell 7.1.0-preview.6
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.

PS C:\Users\leonard> dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart

Deployment Image Servicing and Management tool
Version: 10.0.18362.900

Image Version: 10.0.18363.1016

Enabling feature(s)
[=====100.0%=====]
The operation completed successfully.
PS C:\Users\leonard>
    
```

Figura 4: Activación WSL2.

## Instalación GNU/Linux en el entorno de virtualización

Como alternativa al uso del entorno WSL, se puede instalar el sistema operativo GNU/Linux dentro Windows a través de un entorno de virtualización, como VMWare o VirtualBox.

En este caso el primer paso es descargar el entorno de virtualización. En el caso de VMWare hay que ir a la [web](#)<sup>1</sup> y descargar la última versión (15.5.6 en las capturas, 16.0 última) de VMware Player. En el caso de VirtualBox hay que ir a la [web](#)<sup>2</sup> y descargar la última versión (6.1.14 en las capturas, 6.1.38 última) del programa. En ambos casos el proceso de instalación es muy intuitivo y no debería presentar complicación alguna, por lo que el proceso no se detalla en esta guía.

A continuación hay que descargar la imagen del sistema operativo GNU/Linux. En este caso utilizaremos la última versión de la distribución Ubuntu (20.04.1 LTS Focal Fossa), que tiene soporte para 5 años y se puede descargar de la siguiente [web](#)<sup>3</sup> (atención, hay que descargar la versión *Desktop image*!).

Una vez instalado el entorno de virtualización y descargada la imagen del sistema operativo GNU/Linux procederemos a su instalación. El entorno recomendado para realizar el proceso es VMWare Player, por lo que el resto del documento describe el proceso de creación de la máquina virtual para este entorno.

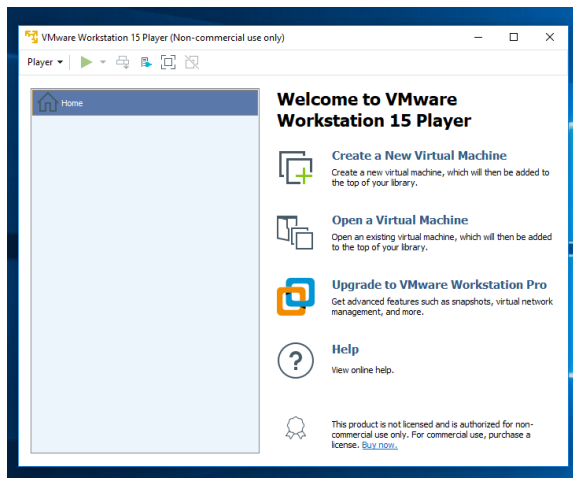
El primer paso es abrir VMWare Player y crear una máquina virtual nueva. Para ello hay que escoger la opción "*Create a New Virtual Machine*", tal como se muestra en la Figura 5a. A continuación hay que elegir la opción "*Typical (recommended)*" (Figura 5b) y elegir el fichero `.iso` que contiene la imagen del sistema operativo (Figura 5c).

---

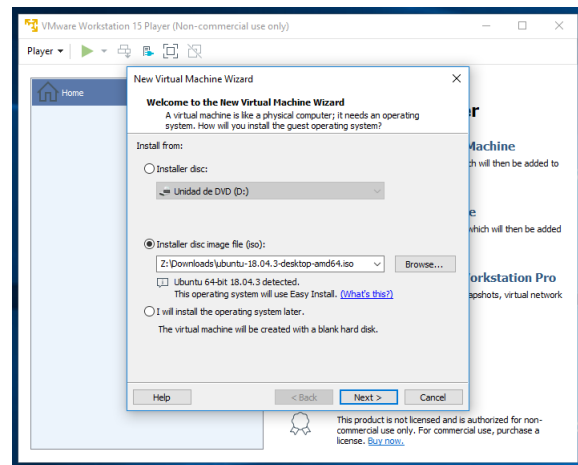
<sup>1</sup><https://www.vmware.com/go/downloadworkstationplayer>

<sup>2</sup><https://www.virtualbox.org/wiki/Downloads>

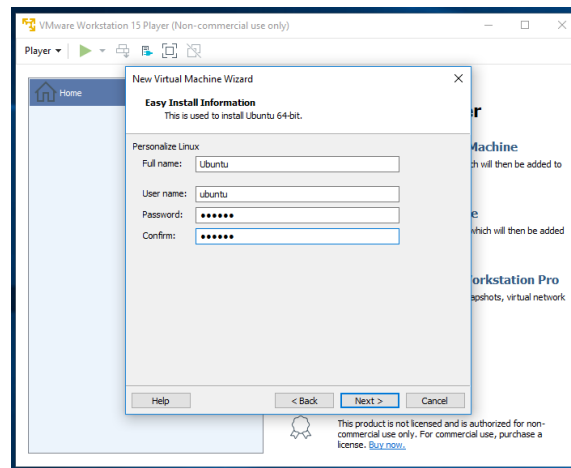
<sup>3</sup><http://releases.ubuntu.com/20.04/>



(a)



(b)

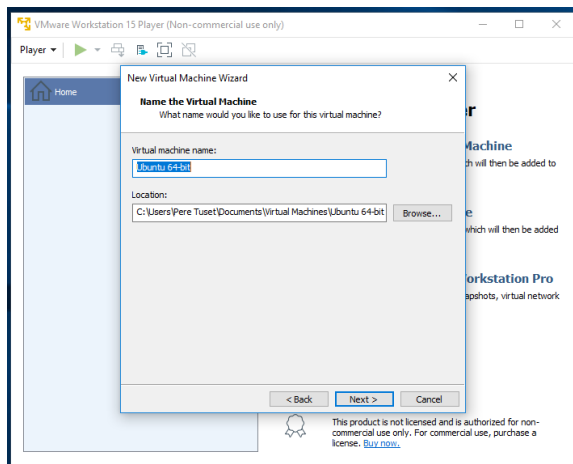


(c)

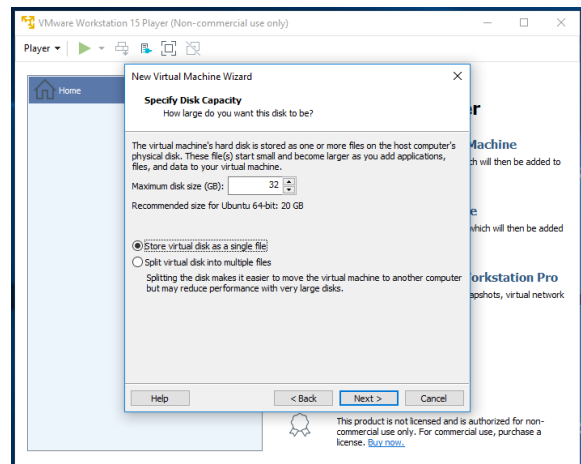
**Figura 5:** Creación de la máquina virtual Ubuntu 20.04 LTS con VMWare Player.

Una vez seleccionada la imagen del sistema operativo procederemos a escoger un nombre de usuario y contraseña (en este caso: `ubuntu/ubuntu`) y el nombre de la máquina virtual (Ubuntu 64-bit), tal como se muestra en la Figura 6a. Finalmente, escogeremos el nombre de la máquina virtual (Ubuntu 64-bit) y el tamaño del disco virtual (32 GBytes), tal como se muestra en la Figura 6b y Figura 6c, respectivamente.

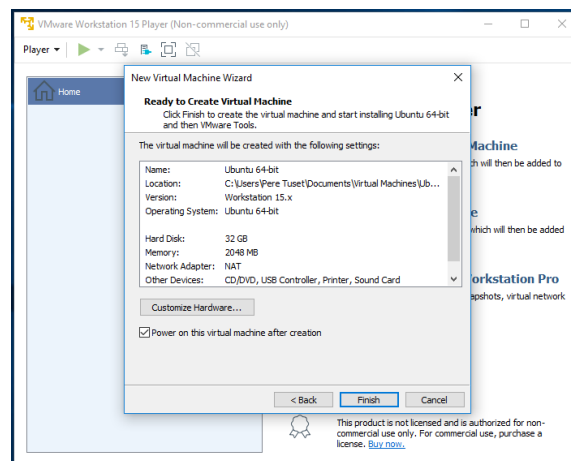
VMWare dispone de un modo de instalación desatendida, por lo que una vez se arranca la máquina virtual por primera vez se procede a la instalación automática del sistema operativo. Una vez completado el proceso de instalación, la máquina virtual se reiniciará y se nos presentará con la pantalla de inicio de sesión, tal como se muestra en la Figura 7a, donde deberemos utilizar el usuario y la contraseña elegidos anteriormente.



(a)



(b)



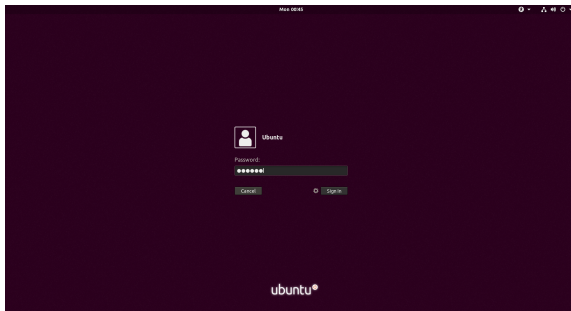
(c)

**Figura 6:** Creación de la máquina virtual Ubuntu 20.04 LTS con VMWare Player.

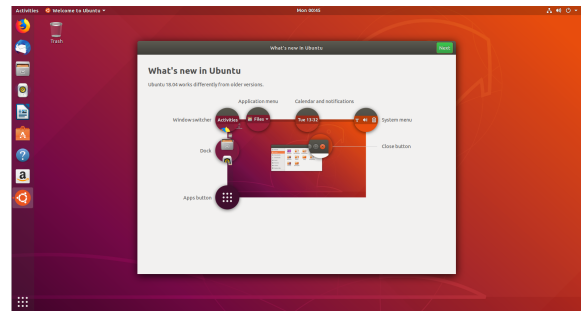
Una vez iniciada la sesión nos encontraremos el escritorio, donde podremos empezar a trabajar, tal como se muestra en la Figura 7b.

El primer paso para configurar el sistema será cambiar la localización del teclado, pues por defecto puede estar en inglés y presentará problemas a la hora de introducir caracteres especiales. Para ello pulsaremos la combinación de teclas **CTRL+ALT+T** para abrir un terminal de comandos y a continuación ejecutaremos los siguientes comandos (un comando por línea):

```
1 gsettings set org.gnome.desktop.input-sources sources "[('xkb', 'es')]"
2 gsettings set org.gnome.desktop.input-sources current 1
```



(a)

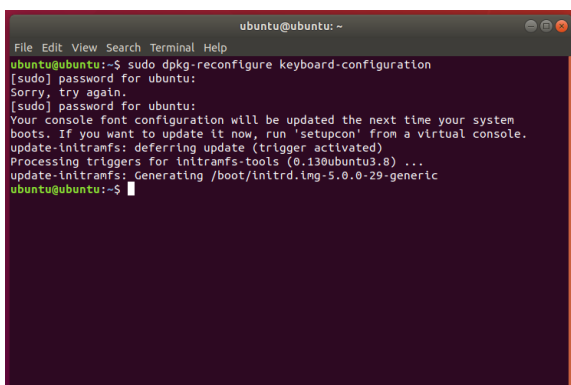


(b)

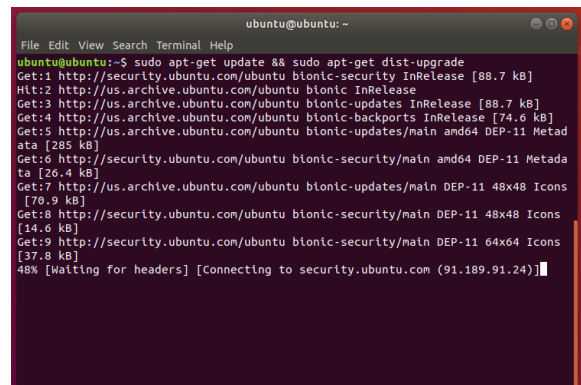
**Figura 7:** Inicio de sesión de Ubuntu 20.04 LTS con VMWare Player.

Una vez modificada la localización del teclado, actualizaremos el sistema operativo ejecutando los siguientes comandos, tal como se muestra en la Figura 8b. El sistema nos pedirá la contraseña de *root* (*ubuntu*) y procederá a actualizarse.

```
1 sudo apt-get update && sudo apt-get dist-upgrade
```



(a)



(b)

**Figura 8:** Cambio de la localización del teclado y actualización del sistema.

Por último, deberemos instalar las *OpenVM Tools*, un conjunto de herramientas que aseguran el correcto funcionamiento entre el sistema operativo *host* (Windows 10) y *guest* (GNU/Linux) que se ejecutan en nuestra máquina. Para ello tendremos que ejecutar el siguiente comando desde un terminal:

```
1 sudo apt-get install open-vm-tools
```

Una vez finalizado el proceso de actualización deberemos reiniciar el sistema operativo

y ya estaremos listos para proceder a la instalación de los programas para descargar y compilar el código fuente, tal como se explica en el siguiente capítulo del documento.

---

# Instalación del entorno de trabajo y compilación del primer programa

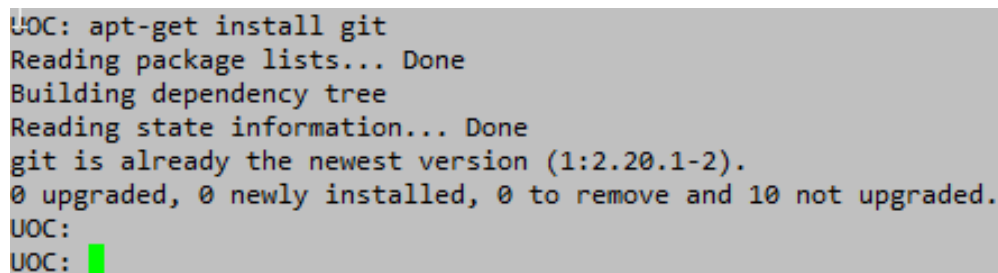
## Instalación de git y gcc

Una vez instalado el sistema operativo vamos a instalar `git`, un sistema de control de versiones ampliamente utilizado y que nos permitirá trabajar con el código fuente, y `gcc`, el compilador que se encarga de generar el fichero binario que se ejecuta en el procesador a partir del código fuente.

Para asegurar que `git` se encuentra instalado en el sistema hay que abrir un terminal de comandos y ejecutar el siguiente comando:

```
1 sudo apt-get install git
```

Después de introducir la contraseña se procederá a la instalación, o se nos indicará que `git` ya se encuentra instalado en el sistema.



```
UOC: apt-get install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
git is already the newest version (1:2.20.1-2).
0 upgraded, 0 newly installed, 0 to remove and 10 not upgraded.
UOC:
UOC: █
```

**Figura 9:** Instalación de `git`, un sistema de control de versiones.

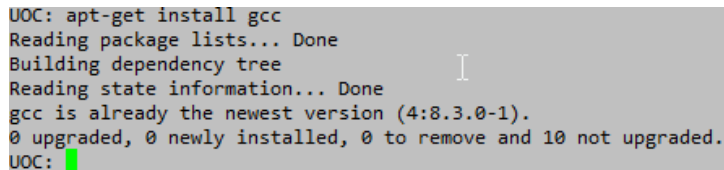
Por su parte, para asegurar que el compilador `gcc` se encuentra instalado en el sistema hay que abrir un terminal y ejecutar el comando<sup>4</sup>.

---

<sup>4</sup>Recordad que podéis utilizar `sudo` delante del comando en caso de no disponer de permisos de *root*.

```
1 apt-get install gcc
```

Después de introducir la contraseña se procederá a la instalación o se nos indicará que `gcc` ya se encuentra instalado en el sistema.



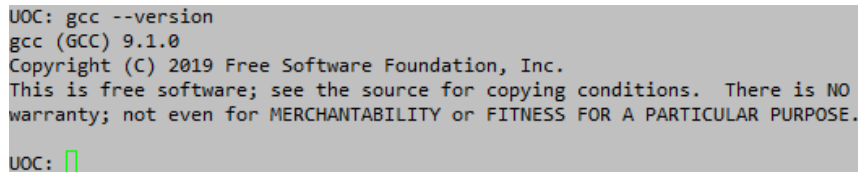
```
UOC: apt-get install gcc
Reading package lists... Done
Building dependency tree
Reading state information... Done
gcc is already the newest version (4:8.3.0-1).
0 upgraded, 0 newly installed, 0 to remove and 10 not upgraded.
UOC: █
```

**Figura 10:** Instalación de `gcc`.

Una vez instalado `gcc` podemos ver con qué versión trabajamos en nuestro sistema abriendo un terminal y ejecutando el comando:

```
1 gcc --version
```

Como vemos, en el caso del ejemplo la versión de `gcc` instalada en nuestro entorno de trabajo es la 9.1.0. La respuesta dependerá de la versión instalada. La última es la 12.2.0.



```
UOC: gcc --version
gcc (GCC) 9.1.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
UOC: █
```

**Figura 11:** Versión de `gcc`

## Obtención del código fuente


Una vez instalado `git` y `gcc` en nuestro sistema procederemos a obtener el repositorio con el código fuente de ejemplos de la asignatura, que se encuentra en [GitHub](#).

Antes de nada crearemos la carpeta `UOC` donde guardaremos los diferentes ficheros de trabajo. Para ello hay que abrir un terminal y ejecutar el comando <sup>5</sup>

---

<sup>5</sup>Asumimos que el usuario creado durante la instalación del sistema es `ubuntu` y, por tanto, su directorio de trabajo `home` se encuentra en `/home/ubuntu`.

```
1 mkdir /home/ubuntu/UOC
```



```
UOC: mkdir /home/UOC
UOC:
UOC: cd //home/UOC
UOC: pwd
//home/UOC
UOC: 
```

**Figura 12:** Creando directorio de referencia.

A continuación clonaremos el repositorio de GitHub con el comando

```
1 git clone https://bitbucket.org/uoc_xx510/extra
```

Aquí tendremos los ejemplos del curso.

A continuación clonaremos el repositorio de GitHub con el comando

```
1 git clone https://bitbucket.org/uoc_xx510/pac0_2022
```

desde la carpeta donde nos interese clonar el repositorio. Al tratarse de un repositorio Git público no se nos pedirán credenciales de acceso, pero si fuese un repositorio privado habría que introducir el nombre de usuario y la contraseña.

Una vez clonado el repositorio podremos entrar en el directorio con el comando

```
1 cd pac0
```

y ver la estructura de directorios con en comando

```
1 ls -la
```

La carpeta contiene diferentes ficheros que luego comentaremos. Finalmente podemos utilizar el comando

```
1 ls -laF|awk '{print $1,$9}'
```

para imprimir los permisos del fichero y su nombre (sin ninguna información adicional).



## Compilación del programa *Hello world!*

Una vez disponemos del código fuente podemos empezar a utilizar `gcc` con el ejemplo `Hello World!`, que estará grabado en un fichero de texto llamado `hello1.c`. Como vemos en el Código 1, el programa imprime por pantalla el mensaje `Hello World !!` y finaliza su ejecución de manera ordenada (devolviendo 0).

```
1  /*
2   * Filename: hello1.c
3   */
4
5  #include <stdio.h>
6
7  int main(int argc, char* argv[])
8  {
9      printf("Hello world !!\n");
10
11     return 0;
12 }
```

**Código 1:** Código fuente del fichero `hello1.c`

Para compilar el fichero podemos utilizar el comando

```
1 gcc hello1.c
```

y si el resultado es exitoso (no hay problemas en el proceso de compilación) obtendremos un fichero ejecutable llamado `a.out`. Es importante tener en cuenta que en sistemas operativos GNU/Linux la extensión del fichero (en este caso `.out`) es irrelevante para determinar si se trata de un programa u otro tipo de fichero. De hecho, la capacidad de ejecución de un fichero se basa en las propiedades del mismo, de modo que la mayoría de programas en un sistema operativo GNU/Linux no tienen extensión. En cambio, en sistemas operativos Windows la mayoría de programas tienen extensión (`.exe`).

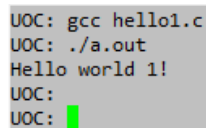
Recordemos también que los parámetros de entrada del programa se definen con las variables `argv` y `argc` de la función `main()`, tal como se muestra en la línea 7. Por un lado, `argc` será un valor entero con el número de argumentos pasados en la ejecución del programa. Por otro lado, `argv` será un vector de punteros a las cadenas de caracteres (*string*) que representan los diferentes parámetros pasados en la ejecución del programa. De este modo el parámetro `argv[0]` será siempre el nombre del programa tal y como lo

hemos introducido en la línea de comandos, mientras que de `argv[1]` a `argv[argc-1]` tendremos el resto de parámetros.

Una vez compilado satisfactoriamente, podemos ejecutar el programa escribiendo el comando.

```
1 ./a.out
```

Como vemos en la Figura 13, la salida del programa es **Hello World 1!**. Es importante tener en cuenta que se ha incluido el prefijo `./` al nombre del programa para indicar que este se encuentra en el propio directorio. Para evitarlo se podría incluir el directorio actual en la variable `PATH` de la consola, pero esto podría llevar a problemas en la búsqueda de programas del sistema y no es recomendable. Por tanto, es un buen hábito indicar siempre que vamos a ejecutar el fichero del directorio actual utilizando el prefijo `./` antes del nombre del programa.



```
UOC: gcc hello1.c
UOC: ./a.out
Hello world 1!
UOC:
UOC: █
```

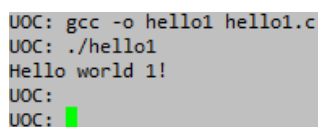
**Figura 13:** Ejecución del programa de ejemplo `a.out`.

Si bien el nombre por defecto de un programa compilado con `gcc` es `a.out`, podríamos definir su nombre en el momento del compilado (es lo habitual) utilizando el comando

```
1 gcc -o hello1 hello1.c
```

donde el parámetro `-o` indica el nombre del fichero de salida. Como vemos en la Figura 14, el resultado es que nuestro programa ahora se llama `hello1`, y podemos ejecutarlo con el comando

```
1 ./hello1
```



```
UOC: gcc -o hello1 hello1.c
UOC: ./hello1
Hello world 1!
UOC:
UOC: █
```

**Figura 14:** Ejecución del programa de ejemplo `hello1.c` compilado con nombre `hello1`.

El proceso de desarrollo de un programa utilizando el lenguaje de programación C es complejo y plagado de errores propios de la condición humana. Por ejemplo, escribir mal el nombre de una variable o olvidarse un `;` al final de una línea. A la vista de esto, es un buen hábito compilar el programa de manera frecuente (para ir detectando posibles errores de manera incremental) y también activar las opciones de alerta (*warning*) del compilador. Esto último se puede hacer activando los parámetros `-Wall` o `-Werror`. El primero activa todos los mensajes de aviso de compilación, de modo que facilita la detección de errores cometidos por el programador. El segundo convierte los mensajes de aviso de compilación en errores, de modo que garantiza que el programador los subsane antes de generar el programa. A pesar de la molestia que puede suponer, el uso de `-Werror` maximiza la probabilidad que el programa funcione correctamente una vez se encuentre en ejecución, por lo que se recomienda su utilización.

Para probar el funcionamiento de estos parámetros vamos a añadir la declaración de una variable de tipo entero que no vamos a utilizar en nuestro programa, tal como se muestra en el Código 2. Como vemos en la Figura 15, con la opción `-Wall` activada el compilador nos avisa que la variable `i` se encuentra declarada en la línea 9 pero no se utiliza a lo largo del programa.

```
1  /*
2   * Filename: hello2.c
3   */
4
5  #include <stdio.h>
6
7  int main(int argc, char* argv[])
8  {
9      int i;
10
11     printf("Hello world 2!\n");
12
13     return 0;
14 }
```

**Código 2:** Código fuente del fichero `hello2.c`

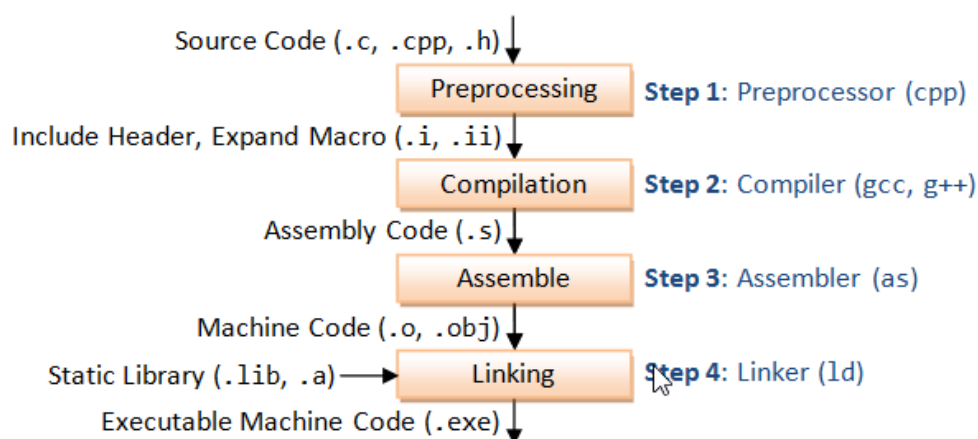
```
UOC: gcc -Wall -Werror -o hello2 hello2.c
hello2.c: In function 'main':
hello2.c:9:9: error: unused variable 'i' [-Werror=unused-variable]
    9 |     int i;
      |         ^
cc1: all warnings being treated as errors
UOC:
UOC: █
```

**Figura 15:** Compilación del programa de ejemplo `hello2.c` con `-Wall`.

---

## Detalle del proceso de compilación de un programa con gcc

Ahora ya sabemos cómo compilar un programa escrito en el lenguaje de programación C utilizando el compilador gcc en un sistema GNU/Linux. Pero la mayoría de programas no se componen de un único fichero, sino que están formados por diferentes módulos (unidades de compilación) que son compilados de manera independiente y que son unidas (linkadas) para formar el programa ejecutable final. Además, la mayoría de programas escritos en C cuentan con inclusiones (directivas `include`) y definiciones (directivas `define`) que deben ser resueltas antes del proceso de compilación de cada unidad de compilación. Así pues, el proceso desde que tenemos nuestro código fuente hasta que lo podemos ejecutar lo podríamos descomponer en las fases que se muestran en la Figura 16 y que son descritas a continuación.




**Figura 16:** Proceso de compilación de un programa en C.

## Paso 1: El preprocesador

El procesador se encarga de incorporar los ficheros de cabecera a través de la directiva `include` (ficheros con extensión `.h`), de expandir las posibles macros de nuestro código a través de la directiva `define` (por ejemplo valores constantes) y de eliminar los comentarios del usuario (ya que éstos no son interpretados por el compilador). Podemos generar el fichero intermedio generado por el preprocesador con el siguiente comando, donde `hello2.i` será el fichero de salida.

```
1 cpp hello2.c hello2.i
```



```
UOC: cpp hello2.c hello2.i
UOC:
UOC: █
```

**Figura 17:** Proceso de preprocesado de un programa en C.

Podemos ver el contenido del fichero expandido `hello2.i` con el siguiente comando:

```
1 cat hello2.i
```

Para mostrar solo las primeras 25 líneas de la salida (para facilitar la lectura) podemos utilizar el siguiente comando:

```
1 head -25 hello2.i
```

También podemos mostrar las últimas 25 líneas del mismo fichero con el siguiente comando:

```
1 tail -25 hello2.i
```

Cómo vemos, la directiva `#include <stdio.h>` ha incorporado un gran número de líneas entre las que se incluyen rutas a las librerías y definición de funciones.

## Paso 2: El compilador

A partir de la salida del preprocesador, el compilador se encarga de generar un código en ensamblador para la arquitectura con la que vayamos a trabajar. Para generar el código

```
UOC: head -25 hello2.i
# 1 "hello2.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "hello2.c"

# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 442 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 443 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 444 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
UOC:
UOC: █
```

**Figura 18:** Primeras 25 líneas del fichero resultado del preprocesado.

```
UOC: tail -25 hello2.i
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 864 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 879 "/usr/include/stdio.h" 3 4

# 6 "hello2.c" 2

# 7 "hello2.c"
int main(int argc, char* argv[])
{
    int i;

    printf("Hello world 2!\n");

    return 0;
}
UOC: █
```

**Figura 19:** Últimas 25 líneas del fichero resultado del preprocesado.

en ensamblador a partir de la salida del preprocesador utilizamos el siguiente comando:

```
1  .file "hello2.c"
2  .text
3  .section .rodata
4  .LC0:
5  .string "Hello world 2!"
6  .text
7  .globl main
8  .type main, @function
9  main:
10 .LFB0:
11 .cfi_startproc
12 pushq %rbp
13 .cfi_def_cfa_offset 16
14 .cfi_offset 6, -16
15 movq %rsp, %rbp
16 .cfi_def_cfa_register 6
17 subq $16, %rsp
18 movl %edi, -4(%rbp)
19 movq %rsi, -16(%rbp)
20 movl $.LC0, %edi
21 call puts
22 movl $0, %eax
23 leave
24 .cfi_def_cfa 7, 8
25 ret
26 .cfi_endproc
27 .LFE0:
28 .size main, .-main
29 .ident "GCC: (GNU) 9.1.0"
30 .section .note.GNU-stack,"",@progbits
```

**Código 3:** Código ensamblador del fichero `hello2.s`.

```
1 gcc -S hello2.i
```

Como la salida del preprocesador no es demasiado útil para el usuario, podemos realizar el proceso de compilación directamente a partir del fichero original utilizando el siguiente comando:

```
1 gcc -S hello2.c
```

En este caso, el compilador se encargará de llamar al preprocesador de manera transparente.

El resultado del proceso de compilación es un código en ensamblador tal como se



```
UOC: gcc -S hello2.i
UOC: gcc -S hello2.c
UOC:
UOC: █
```

**Figura 20:** Resultado en código ensamblador de nuestro ejemplo.

muestra en el Código 3. Como vemos, el código de alto nivel se ha convertido en un conjunto de instrucciones en ensamblador para una arquitectura genérica.

```
UOC: hexdump hello2.o | head -10
00000000 457f 464c 0102 0001 0000 0000 0000 0000
00000010 0001 003e 0001 0000 0000 0000 0000 0000
00000020 0000 0000 0000 0000 0290 0000 0000 0000
00000030 0000 0000 0040 0000 0000 0040 000d 000c
00000040 4855 e589 8348 10ec 7d89 48fc 7589 bff0
00000050 0000 0000 00e8 0000 b800 0000 0000 c3c9
00000060 6548 6c6c 206f 6f77 6c72 2064 2132 0000
00000070 4347 3a43 2820 4e47 2955 3920 312e 302e
00000080 0000 0000 0000 0000 0014 0000 0000 0000
00000090 7a01 0052 7801 0110 0c1b 0807 0190 0000
UOC: hexdump hello2.o | tail -10
0000540 0008 0000 0000 0000 0018 0000 0000 0000
0000550 0009 0000 0003 0000 0000 0000 0000 0000
0000560 0000 0000 0000 0000 01c8 0000 0000 0000
0000570 0014 0000 0000 0000 0000 0000 0000 0000
0000580 0001 0000 0000 0000 0000 0000 0000 0000
0000590 0011 0000 0003 0000 0000 0000 0000 0000
00005a0 0000 0000 0000 0000 0228 0000 0000 0000
00005b0 0061 0000 0000 0000 0000 0000 0000 0000
00005c0 0001 0000 0000 0000 0000 0000 0000 0000
00005d0
UOC: █
```

**Figura 21:** Resultado del ensamblado de nuestro ejemplo. Primeras y últimas 25 líneas.

## Paso 3: El ensamblador

El ensamblador se encarga de convertir un código en ensamblador en lenguaje máquina de la arquitectura de nuestro procesador. Es decir, convierte el conjunto secuencial de instrucciones (y sus respectivos parámetros) en instrucciones que pueden ser ejecutadas por el procesador.

La salida del ensamblador es un fichero que ya no es editable con un editor de texto normal, pues puede contener información no imprimible por pantalla (códigos ASCII por debajo del 0x20). Por tanto, el fichero `hello2.o` debe visualizarse con un editor hexadecimal como `hexdump` como se muestra en la Figura 21. Para ello utilizamos el siguiente comando:

```
1 hexdump hello2.o
```

En caso de que el comando no se encuentre en el sistema, se puede instalar utilizando el siguiente comando:

```
1 apt-get install bsdmainutils
```

Igual que en el caso de preprocesador, la salida intermedia no es demasiado útil para las personas, de modo que también podemos utilizar la herramienta de ensamblar. Para ello utilizamos el siguiente comando:

```
1 as hello2.s -o hello2.o
```

## Paso 4: El linkador

Por último, el linkador toma el código en lenguaje máquina junto con las librerías para producir el fichero ejecutable por el sistema. Para ello utilizamos el siguiente comando:

```
1 ld -o hello2 hello2.o -lc --entry main
```

Para no tener errores que nos impiden generar el fichero de salida tenemos que linkar nuestro programa con la librería estándar del lenguaje de programación C (parámetro `-lc`) e indicar el punto de entrada en ejecución del programa (parámetro `-entry main`).

En caso contrario, el linkador nos indicará que no encuentra las referencias a las funciones utilizadas o bien que no encuentra el punto de entrada en ejecución, tal y como se muestra en la Figura 22.

Una vez linkado el programa podemos ver las librerías que se han utilizado para el linkado de nuestro ejecutable utilizando el siguiente comando:

```
1 ldd hello2
```

Cómo vemos en la Figura 23, en nuestro caso se ha linkado el programa con la librería `linux-vdso.so.1`, `libc.so.6` y `ld64.so.1`. La primera es una librería (dinámica) que se encarga de mapear el espacio de direcciones del programa de manera automática para

```
UOC: ld -o hello2 hello2.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000000401000
ld: hello2.o: in function `main':
hello2.c:(.text+0x15): undefined reference to `puts'
UOC: ld -o hello2 hello2.o -lc
ld: warning: cannot find entry symbol _start; defaulting to 0000000000401020
UOC: ld -o hello2 hello2.o --entry main
ld: hello2.o: in function `main':
hello2.c:(.text+0x15): undefined reference to `puts'
UOC: ld -o hello2 hello2.o -lc --entry main
UOC: █
```

**Figura 22:** Linkado de nuestro ejemplo.

```
UOC: ldd hello2
linux-vdso.so.1 (0x00007ffe35d8b000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2202b6f000)
/lib/ld64.so.1 => /lib64/ld-linux-x86-64.so.2 (0x0000564802533000)
UOC:
UOC: █
```

**Figura 23:** Listado de las librerías en nuestro ejemplo.

aumentar el rendimiento de las llamadas al sistema. La segunda es la librería estándar del lenguaje de programación C para el sistema operativo utilizado, en nuestro caso GNU/Linux. La tercera es la librería que se encarga de realizar el linkaje dinámico requerido por el programa en el momento de la ejecución del mismo, de cargarlo en memoria y de ejecutarlo.

Lógicamente, podemos realizar todos los cuatro pasos (preprocesado, compilado, ensamblado y linkado) de manera encadenada y transparente para el usuario utilizando el siguiente comando:

```
1 gcc -o hello2 hello2.c
```

Como en el caso anterior, esto generará un fichero ejecutable con el nombre **hello2**.

Una vez compilado el programa podemos utilizar herramientas del sistema operativo para analizar el tipo de los diferentes ficheros generados, como vemos en la Figura 24. Ejecutando el comando **file** para cada fichero podemos ver que **hello2.i** es un fichero de texto (ASCII) que contiene código fuente en C y **hello2.s** es un fichero de texto (ASCII) que contiene código fuente en ensamblador. Por su parte, **hello2.o** es un fichero objeto pendiente de linkado y **hello2** es un fichero ejecutable para la arquitectura **x86-64**.

Además, también podemos utilizar la utilidad **nm** para tener la tabla de símbolos referenciados en nuestro programa. En el caso del fichero **hello2.o**, tal y como se muestra en la Figura 25 vemos cómo los únicos símbolos son **main** y **puts**.

En cambio, para el fichero ejecutable **hello2** como vemos en la Figura 26 la lista de

```
UOC: file hello2.c
hello2.c: C source, ASCII text
UOC: file hello2.i
hello2.i: C source, ASCII text
UOC: file hello2.s
hello2.s: assembler source, ASCII text
UOC: file hello2.o
hello2.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
UOC: file hello2
hello2: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib/ld64.so.1, not stripped
UOC:
UOC: █
```

**Figura 24:** Información sobre los diferentes ficheros del proceso de compilado de nuestro ejemplo.

```
UOC: nm hello2.o
0000000000000000 T main
                 U puts
UOC: █
```

**Figura 25:** Símbolos referenciados en `hello2.o`.

símbolos y funciones se incrementa debido al proceso de linkado.

```
UOC: nm hello2
0000000000403eb0 d _DYNAMIC
0000000000404000 d _GLOBAL_OFFSET_TABLE_
0000000000404020 D __bss_start
0000000000404020 D _edata
0000000000404020 D _end
0000000000401020 T main
                 U puts@@GLIBC_2.2.5
UOC: █
```

**Figura 26:** Símbolos referenciados en `hello2`.

---

# Ejercicios a realizar

A continuación se presentan dos ejercicios de uso de línea de comandos en Linux y cuatro ejercicios de programación que se deben resolver utilizando el lenguaje de programación C y el sistema operativo GNU/Linux. La duración estimada para la realización de estos ejercicios es de 15 horas.

Para cada ejercicio/programa, se debe entregar:

- Como archivo adjunto: el código fuente del programa.
- Como parte del informe en PDF:
  - Las explicaciones relevantes del código. No solo se pide incorporar el código (o las partes más relevantes de este) al informe, con los comentarios que pueda tener el código incorporados, sino también una explicación redactada.
  - Las indicaciones de cómo generar el archivo ejecutable (añadiendo una captura de pantalla de cómo lo habéis ejecutado en vuestro sistema, donde se pueda ver la correcta generación del mismo),
  - Las indicaciones de cómo ejecutar el programa, y capturas de pantalla de algunas ejecuciones donde se pueda constatar el correcto funcionamiento de este, para los casos de uso indicados en el enunciado.

## Ejercicios de línea de comandos (40%)

*Esta Prueba de Evaluación Continua vale 0 puntos en la evaluación final de la asignatura.*

### Ejercicio 1

Usando solo los comandos `mkdir` y `cd`, crear la siguiente estructura de directorios:

- SO
  - Actividad Introdutoria
    - \* Bash
    - \* Programación
      - Ex1
      - Ex2
      - Ex3
      - Ex4
    - \* Informe
  - PEC1
  - PEC2
  - PEC3
  - PEC4

Guarden este documento (el enunciado) en `SO/Actividad Introdutoria/`. Cree un archivo de texto cualquiera y guárdelo en `SO/Actividad Introdutoria/Bash/`.

Usando solo los comandos `cd` y `ls`, comenzando con la línea de comandos situada en `SO/`, muestre que ambos archivos están donde deben estar y consulte sus permisos.

## Ejercicio 2

Sustituyan el contenido del archivo de texto en `SO/Actividad Introdutoria/Bash/` por el siguiente:

```
1  #!/bin/bash
2
3  # Crear un número aleatorio de directorios
4  num_directorios=$((2 + RANDOM % 5))
5
6  for ((i=1; i<=num_directorios; i++))
7  do
8      directorio="directory_$i"
9      mkdir $directorio
10     cd $directorio
```

```
11
12  # Crear un número aleatorio de archivos dentro de cada directorio
13  num_archivos=$((2 + RANDOM % 2))
14
15  for ((j=1; j<=num_archivos; j++))
16  do
17      archivo="file_$(j).txt"
18      content=$((RANDOM % 2))
19      if [ "$content" -ne "1" ]
20      then
21          echo "KEEP" > $archivo
22      else
23          echo "DELETE" > $archivo
24      fi
25  done
26  cd ..
27 done
28
```

A continuación, usando solo instrucciones de línea de comandos:

1. Usar `chmod` para poner los permisos de este archivo a 755.
2. Ejecute el archivo.
3. Renombre todos los directorios que se hayan creado que tengan un número impar para que mantengan el mismo nombre, pero que su número sea par.
4. Elimine los archivos que se hayan creado que tengan como contenido "DELETE".

## Ejercicios de programación (60%)

*Esta Prueba de Evaluación Continua vale 0 puntos en la evaluación final de la asignatura.*

### Contexto

En un sistema de *lootboxes*, cada vez que el usuario gana un premio, el sistema debe elegir qué tipo de premio ha obtenido. Estos premios pueden pertenecer a colecciones, y algunos elementos de las colecciones son más difíciles de obtener que otros.

En esta actividad, programaremos una versión virtual de un sistema de *lootboxes*.



**Figura 27:** Ejemplo de sistema de *lootbox* de cartas coleccionables.

## Ejercicio 1

Escribir un programa `ex1` que genere 1000 números enteros aleatorios siguiendo una distribución de probabilidad uniforme. Todos los números generados deben estar entre el 1 y el 100 (ambos incluidos).

El programa debe contar cuántas veces se genera cada número y mostrar los resultados de este conteo por pantalla cuando termina su ejecución.

Debido a las características de la distribución de probabilidad uniforme, en todas las ejecuciones del programa deberían salir unos resultados similares, en los que todos los números hayan salido aproximadamente las mismas veces.

## Ejercicio 2

Escribir un programa `ex2` que simule un sistema de *lootboxes*, a la que hay 4 tipos diferentes de premios, en función de su rareza. Y a la que la probabilidad que tiene cada bola de contener uno de estos premios se define de la forma siguiente:

- Épico: la probabilidad de obtener un premio épico es del 1%.
- Raro: la probabilidad de obtener un premio raro es del 5%.
- Infrecuente: la probabilidad de obtener un premio infrecuente es del 15%.
- Común: la probabilidad de obtener un premio común es del 79%.

El programa debe simular la obtención de 1000 premios, contar cuántos premios de cada tipo se obtienen, y mostrar los resultados de este conteo por pantalla.

Dado que esta distribución de probabilidad es diferente de la del ejercicio anterior, los resultados también deberían mostrar un patrón diferente.



## Ejercicio 3

Modificar el programa anterior para hacer un nuevo programa `ex3` en el que en el sistema de *lootboxes* haya premios de 4 colecciones diferentes:

- Cromos de fútbol: con una probabilidad del 10%.
- Personajes de un juego de rol: con una probabilidad del 20%.
- Armas de un juego cooperativo: con una probabilidad del 30%.
- Monstruos de bolsillo: con una probabilidad del 40%.

Igual que antes, el programa simulará la obtención de 1000 premios, y para cada premio se elegirá a qué colección pertenece, y de qué rareza es. Entonces el programa hará un conteo de cuántos premios de cada rareza y de cada colección se han obtenido, y mostrará los resultados.

## Ejercicio 4

Modificar el programa anterior para hacer un nuevo programa `ex4` que reciba 8 parámetros, por lo tanto, la llamada del programa debe ser `./ex4 EPIC RARE UNCOMMON COMMON CARDS CHARACTERS WEAPONS POKEMONS`.

Para empezar, el programa debe comprobar que el número de parámetros introducidos sea correcto, comprobar que los dos grupos de probabilidades (EPIC, RARE, UNCOMMON y COMMON por un lado, y CARDS, CHARACTERS, WEAPONS y POKEMONS por el otro) sumen 100 y mostrar un mensaje de error y finalizar la ejecución en el caso de que no sea así.

A continuación, el programa debe simular la obtención de 1000 premios haciendo uso de las probabilidades de rareza y las probabilidades de colección suministradas por el usuario, hacer el conteo y mostrar los resultados.