

---

# T1-Introducción

# Índice

---

- El papel del S.O.
- Servicios que ofrece el S.O.
- Formas de acceder al kernel (Tema 8 EC)
  - Modos de ejecución
  - Interrupciones, excepciones y llamadas a sistema
- Llamadas a sistema (Tema 8 EC)
  - Generación de ejecutables
  - Requerimientos de las llamadas a sistema
  - Librerías

---

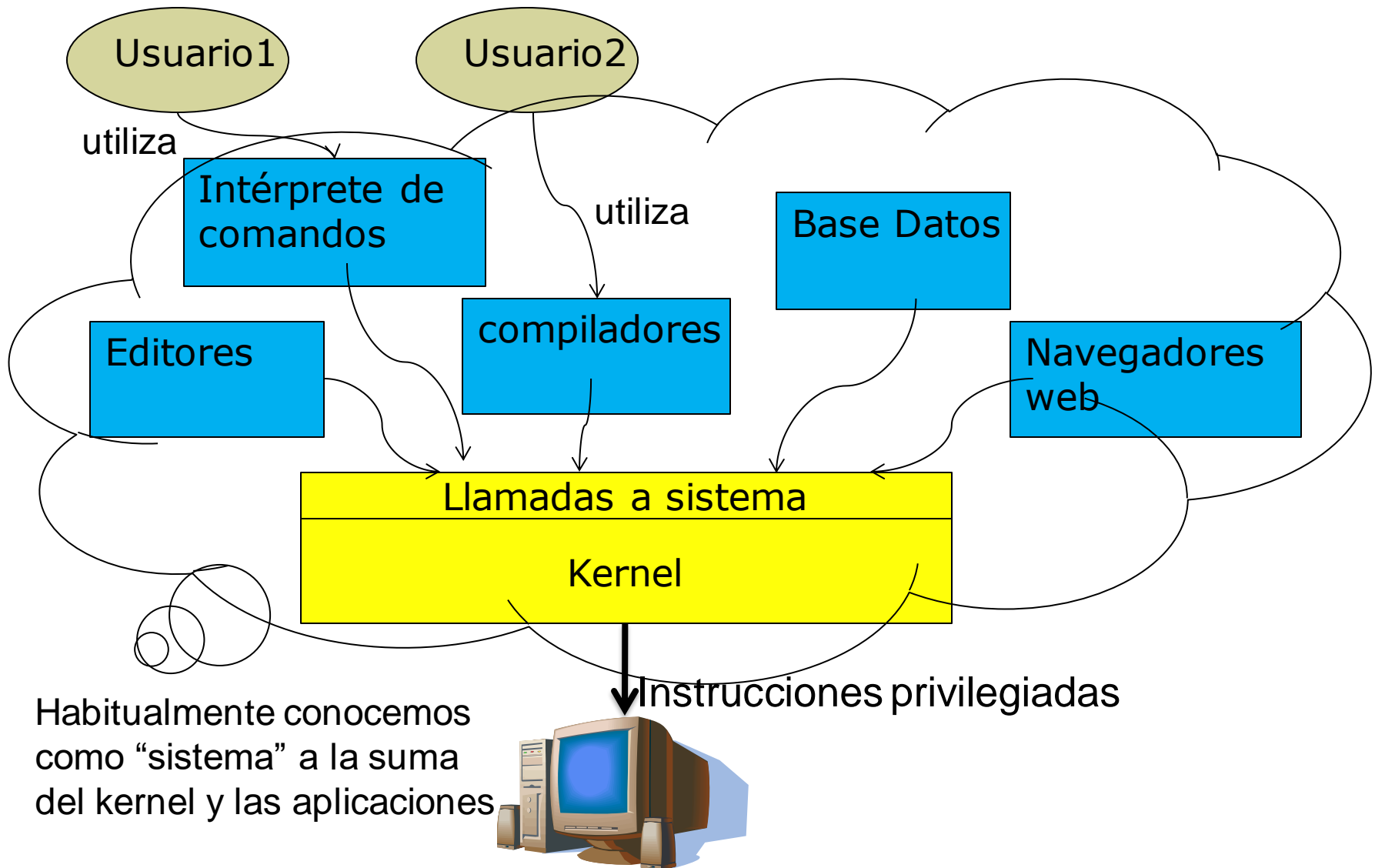
# **EL PAPEL DEL S.O.**

# ¿Qué es un SO?

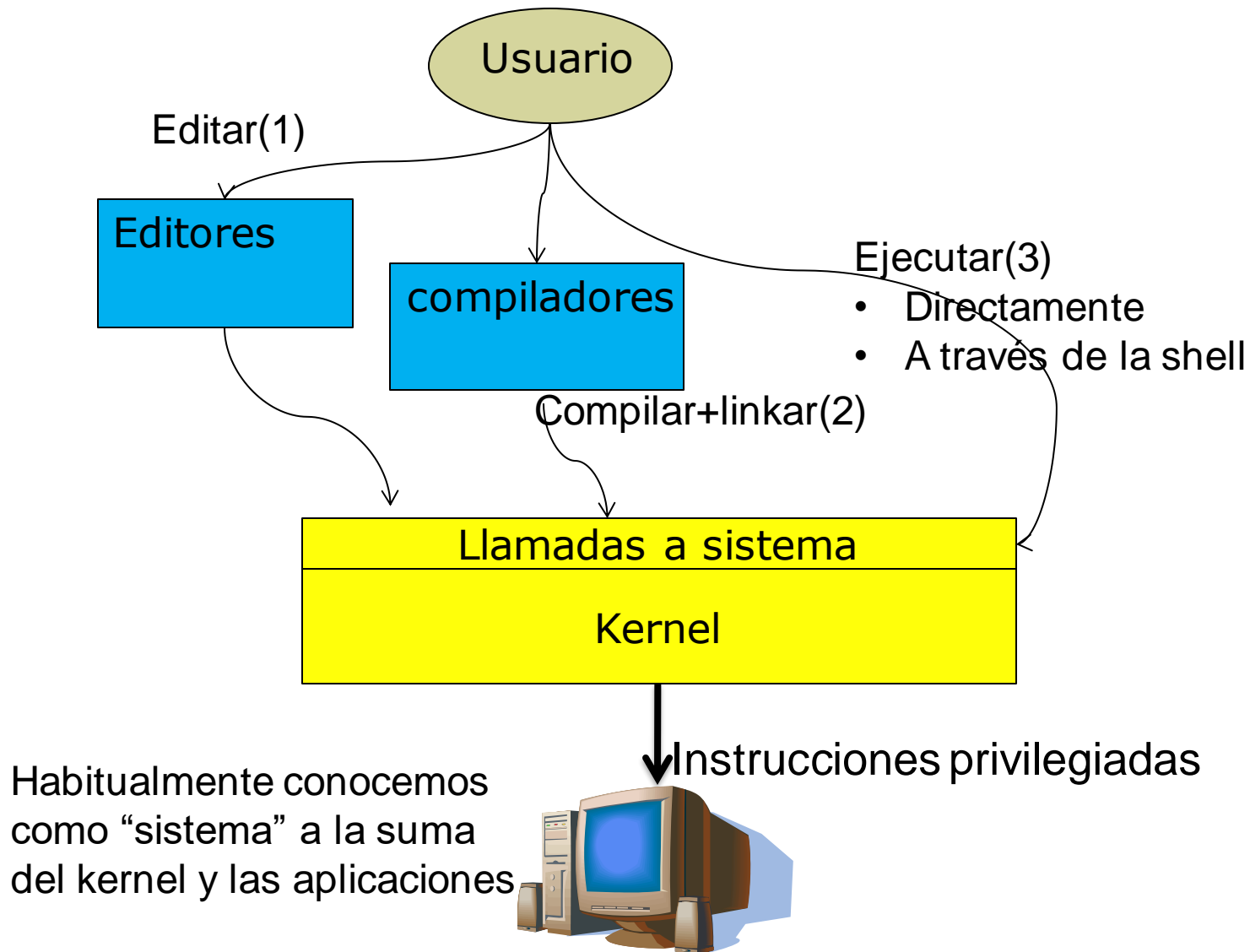
---

- El Sistema Operativo es un software que controla los recursos disponibles del sistema hardware que queremos utilizar y que actúa de intermediario entre las aplicaciones y el hardware
  - **Internamente** define estructuras de datos para gestionar el HW y algoritmos para decidir como utilizarlo
  - **Externamente** ofrece un conjunto de funciones para acceder a su funcionalidad (o servicios) de gestión de recursos

# Componentes del sistema



# Utilización del sistema: Desarrollo y ejecución de programas



# Utilización del sistema



```
# gedit p1.c  
# gcc -o p1 p1.c  
# p1
```

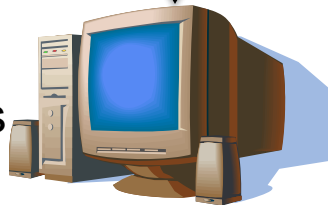
Normalmente utilizamos un **entorno de trabajo**, le llamaremos shell o intérprete de comandos.

Llamadas a sistema

Kernel

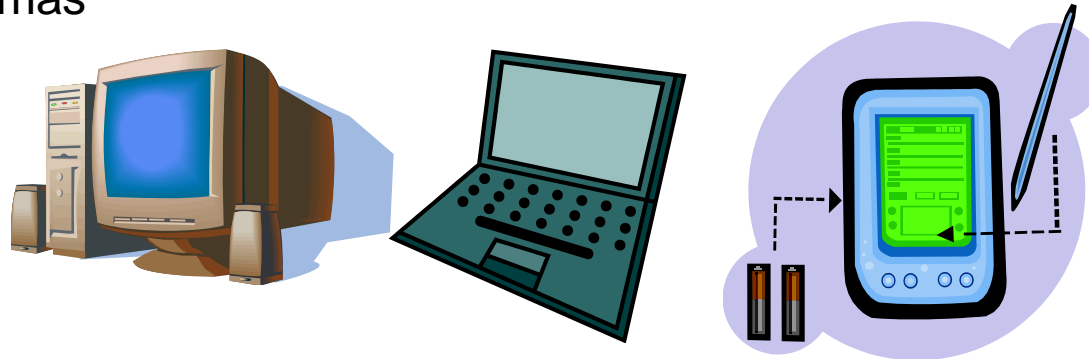
↓ Instrucciones privilegiadas

Habitualmente conocemos como “sistema” a la suma del kernel y las aplicaciones



# ¿Qué esperamos del S.O?

- ❑ Ofrece un **entorno “usable”**
  - ❑ Abstrae a los usuarios de las diferencias que hay entre los diferentes sistemas



- ❑ Ofrece un **entorno seguro**
  - ❑ Protege el HW de accesos incorrectos y a unos usuarios de otros
- ❑ Ofrece un **entorno eficiente**
  - ❑ Proporciona un uso eficiente de los recursos del sistema
  - ❑ Múltiples usuarios acceden a un mismo sistema y tienen una sensación de acceso en “exclusiva”



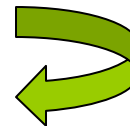
# ¿Que ofrece el SO?



# Arranque del sistema

---

- El hardware carga el SO ( o una parte) al arrancar. Se conoce como *boot*
  - **El sistema puede tener más de un SO instalado (en el disco) pero sólo uno ejecutándose**
  - El SO se copia en memoria (todo o parte de él)
  - Inicializa las estructuras de datos necesarias (hardware/software) para controlar la máquina i ofrecer los servicios
  - Las interrupciones hardware son capturadas por el SO
  - Al final el sistema pone en marcha un programa que permite acceder al sistema
    - ▶ Login (username/password)
    - ▶ Shell
    - ▶ Entorno gráfico



# Entorno de desarrollo

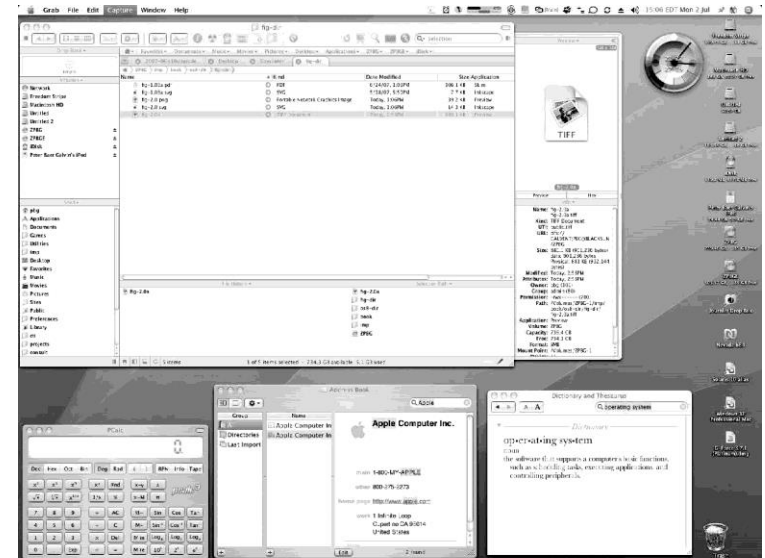
- ❑ El SO ofrece, como parte de sus servicios, un entorno de trabajo **interactivo**. Fundamentalmente puede ser de dos tipos: intérprete de comandos o entorno gráfico
- ❑ Dependiendo del sistema, este entorno puede formar parte del *kernel* o puede ser un programa aparte. En cualquier caso lo encontramos como parte del sistema para poder utilizarlo.

```
Terminal
File Edit View Terminal Tabs Help

fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
sd0 0.0 0.2 0.0 0.2 0.0 0.0 0.0 0.4 0 0
sd1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0

extended device statistics
device r/s w/s kr/s kw/s wait actv svc_t %w %b
fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
sd0 0.6 0.0 38.4 0.0 0.0 0.0 8.2 0 0
sd1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0

(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
~/var/tmp/system-contents/scripts# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
~/var/tmp/system-contents/scripts# uptime
12:53am up 9 min(s), 3 users, load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
~/var/tmp/system-contents/scripts# w
4:07pm up 17 day(s), 15:24, 3 users, load average: 0.09, 0.11, 8.66
User tty login@ idle JCPU PCPU what
root console 15Jun0718days 1 /usr/bin/ssh-agent -- /usr/bi
n/d
root pts/3 15Jun07 18 4 w
root pts/4 15Jun0718days w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
~/var/tmp/system-contents/scripts#
```



# Entorno “usable”, seguro y eficiente

---

- Durante el funcionamiento del sistema, estos son sus tres objetivos básicos
  - Usabilidad
  - Seguridad
  - Eficiencia
- También deberán serlo de vuestros programas
  - Usables:
    - ▶ Si el usuario no lo ejecuta correctamente, habrá que dar un mensaje indicándolo que sea claro y útil
    - ▶ Si alguna función de usuario o llamada a sistema falla, el programa deberá dar un mensaje claro y útil
  - Seguro:
    - ▶ El programa no puede (o no debería) “fallar” si el usuario lo utiliza de forma incorrecta, debe controlar todas las fuentes de error.
    - ▶ Mucho menos hacer fallar al sistema
  - Eficiente:
    - ▶ Hay que plantearse la mejor forma de aprovechar los recursos de la máquina, pensando que los recursos que tiene el usuario son limitados (ejecución en entorno compartido)

---

(Explicado en EC)

# FORMAS DE ACCEDER AL KERNEL

## Como ofrecer un entorno seguro: “Modos” de ejecución

---

- El SO necesita una forma de garantizar su seguridad, la del hardware y la del resto de procesos
- Necesitamos instrucciones de lenguaje máquina privilegiadas que sólo puede ejecutar el kernel
- El HW conoce cuando se está ejecutando el kernel y cuando una aplicación de usuario. Hay una instrucción de LM para pasar de un modo a otro.
- **El SO se ejecuta en un modo de ejecución privilegiado.**
  - **Mínimo 2 modos (pueden haber más)**
    - ▶ **Modo de ejecución NO-privilegiado , *user mode***
    - ▶ **Modo de ejecución privilegiado, *kernel mode***
  - **Hay partes de la memoria sólo accesibles en modo privilegiado y determinadas instrucciones de lenguaje máquina sólo se pueden ejecutar en modo privilegiado**
- **Objetivo: Entender que son los modos de ejecución y porqué los necesitamos**

# ¿Cuándo se ejecuta código de kernel?

---

- Cuando una aplicación ejecuta una llamada a sistema
- Cuando una aplicación provoca una excepción
- Cuando un dispositivo provoca una interrupción
  
- Estos eventos podrían no tener lugar, y el SO no se ejecutaría → El SO perdería el control del sistema.
- **El SO configura periódicamente la interrupción de reloj para evitar perder el control y que un usuario pueda acaparar todos los recursos**
  - **Cada 10 ms por ejemplo**
  - **Típicamente se ejecuta la planificación del sistema**

# Acceso al código del kernel

---

- El kernel es un código guiado por eventos
    - Interrupción del flujo actual de usuario para realizar una tarea del SO
  - Tres formas de acceder al código del SO (Visto en EC):
    - **Interrupciones** generadas por el hardware ( teclado, reloj, DMA, ... )
      - ▶ asíncronas (entre 2 dos instrucciones de lenguaje máquina)
    - Los errores de software generan **excepciones** (división por cero, fallo de página, ... )
      - ▶ síncronas
      - ▶ provocadas por la ejecución de una instrucción de lenguaje máquina
      - ▶ se resuelven (si se puede) dentro de la instrucción
- Peticiones de servicio de programas: **Llamada a sistema**
    - ▶ síncronas
    - ▶ provocados por una instrucción explícitamente (de lenguaje máquina)
    - ▶ para pedir un servicio al SO (llamada al sistema)
  - Desde el punto de vista hardware son muy parecidas (casi iguales)

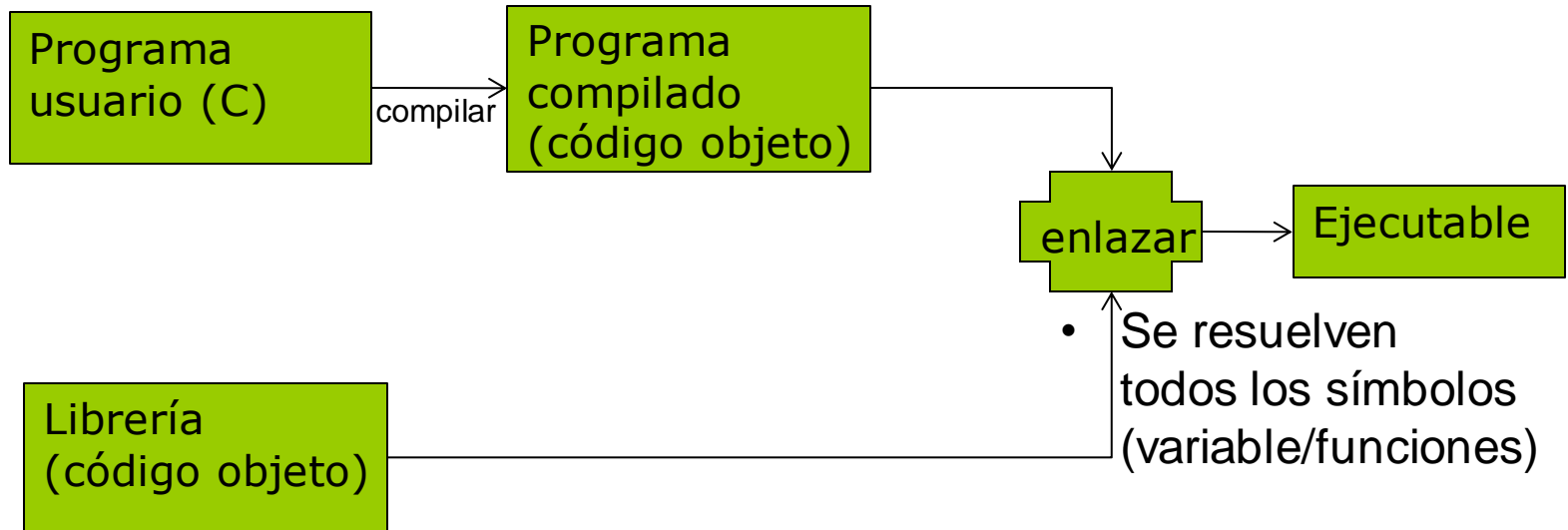


- 
- Generación de ejecutables
  - Requerimientos de las llamadas a sistema
  - Librerías

# LLAMADAS A SISTEMA

# Generación ejecutables

- Se comprueba la sintaxis, tipos de datos, etc
- Se hace una primera generación de código



- Librerías: Rutinas/Funciones ya compiladas (es código objeto), que se “enlazan” con el programa y que el programador sólo necesita llamar
- Pueden ser a nivel de lenguaje (libc) o de sistema (libso)

# Llamadas a Sistema

---

- Conjunto de **FUNCIONES** que ofrece el kernel para acceder a sus servicios
- Desde el punto de vista del programador es igual al interfaz de cualquier librería del lenguaje (C,C++, etc)
- Normalmente, los lenguajes ofrecen un API de más alto nivel que es más cómoda de utilizar y ofrece más funcionalidades
  - Ejemplo: Librería de C: printf en lugar de write
    - ▶ Nota: La librería se ejecuta en **modo usuario** y no puede acceder al dispositivo directamente
- Ejemplos
  - Win32 API para Windows
  - POSIX API para sistemas POSIX (UNIX, Linux, Mac OS X)
  - Java API para la Java Virtual Machine

# Requerimientos llamadas a sistema

---

- Requerimientos
  - Desde el punto de vista del programador
    - ▶ Tiene que ser tan sencillo como una llamada a función
      - `<tipo> nombre_función(<tipo1> arg1, <tipo2> argc2..);`
    - ▶ No se puede modificar su contexto (igual que en una llamada a función)
      - Se deben salvar/restaurar los registros modificados
  - Desde el punto de vista del kernel necesita:
    - ▶ Ejecución en modo privilegiado → soporte HW
    - ▶ Paso de parámetros y retorno de resultados entre modos de ejecución diferentes → depende HW
    - ▶ Las direcciones que ocupan las llamadas a sistema tienen que poder ser variables para soportar diferentes versiones de kernel y diferentes S.O. → por portabilidad

# Solución: Librería “de sistema” con soporte HW

---

- La librería de sistema se encarga de traducir de la función que ve el usuario a la petición de servicio explícito al sistema
  - Pasa parámetros al kernel
  - Invoca al kernel → TRAP
  - Recoge resultados del kernel
  - Homogeneiza resultados (todas las llamadas a sistema en linux devuelven -1 en caso de error)
- ¿Como conseguimos la portabilidad entre diferentes versiones del SO?
  - La llamada a sistema no se identifica con una dirección, sino con un identificador (un número), que usamos para indexar una tabla de llamadas a sistema ( que se debe conservar constante entre versiones)

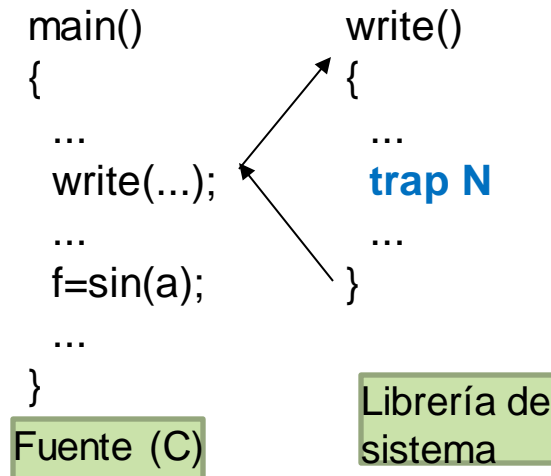
# Llamadas a sistema

---

- La librería de sistema aísla al usuario de los detalles de la arquitectura
- El modo de ejecución privilegiado ofrece seguridad: solo el kernel se ejecuta en modo privilegiado
- La tabla de llamadas a sistema ofrece compatibilidad entre versiones del SO
- Tenemos una solución que solo depende de la arquitectura : lo cual es inevitable ya que es un binario con un lenguaje máquina concreto!!

# Librerías “de sistema”

- El SO ofrece librerías del sistema para aislar a los programas de usuario de todos los pasos que hay que hacer para
  1. Pasar los parámetros
  2. Invocar el código del kernel
  3. Recoger y procesar los resultados



- Se llaman librería de sistema pero se ejecuta en **modo usuario**, **solamente sirven para facilitar la invocación de la llamada a sistema**

# Código genérico al entrar/salir del kernel

- Hay pasos comunes a interrupciones, excepciones y llamadas a sistema
- En el caso de interrupciones y excepciones, no se invoca explícitamente ya que genera la invocación la realiza la CPU, el resto de pasos si se aplican.

	Función “normal”	Función kernel
Pasamos los parámetros	Push parámetros	<b>DEPENDE</b>
<b>Para invocarla</b>	call	sysenter, int o similar
Al inicio	Salvar los registros que vamos a usar (push)	Salvamos todos los registros (push)
<b>Acceso a parámetros</b>	A través de la pila: Ej: mov 8(ebp),eax	<b>DEPENDE</b>
Antes de volver	Recuperar los registros salvados al entrar (pop)	Recuperar los registros salvados (todos) al entrar (pop)
<b>Retorno resultados</b>	eax ( o registro equivalente)	<b>DEPENDE</b>
<b>Para volver al código que la invocó</b>	ret	sysexit, iret o similar