TEMA 1. Introducción

Tema 1. Introducción

1.1 Fundamentos de los Sistemas Operativos

- Kernel
- Llamadas al sistema
- Procesos
- Memoria Virtual
- Planificadores
- Sistemas de Ficheros
- Controladores de Dispositivos

1.2 La línea de comandos shell

- Jerarquía. Ruta absoluta y relativa
- Estructura de directorio
- Enlaces

1.3 Llamadas al sistema y entorno de desarrollo

- CLI
- Llamadas al sistema

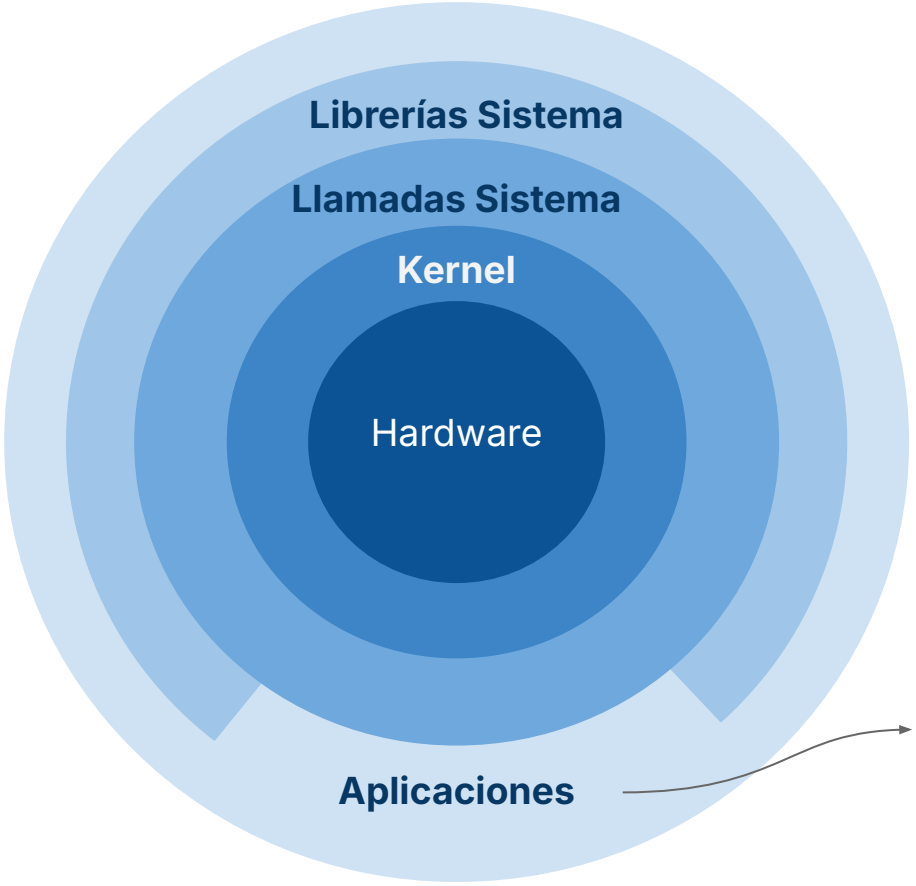
TEMA 1.1 Fundamentos de los SO

Fundamentos de los Sistemas Operativos

- *Kernel*
- Llamadas al sistema
- Procesos
- Memoria Virtual
- Planificadores
- Sistemas de Ficheros
- Controladores de Dispositivos

Kernel. Núcleo del Sistema Operativo

- El **kernel** es el programa que gestiona todo el sistema: dispositivos hardware, memoria, planificación de la CPU... (dependiendo del modelo)
- Modelo de un **kernel monolítico**



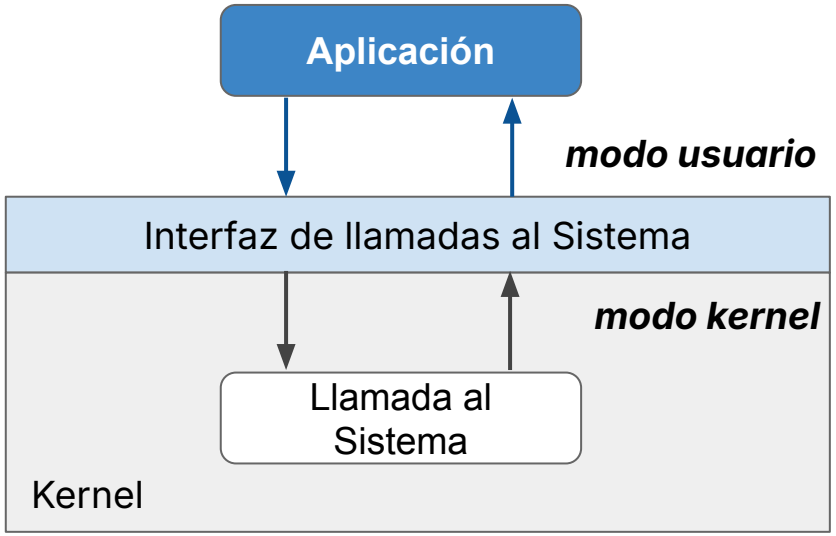
Ej. Golang tiene su propia capa de llamadas al sistema que no requiere la librería de sistema (libc)

Notas del modelo en Linux

- Procesado de paquetes de red en el espacio de usuario mediante librerías y dispositivos especiales (DPDK)
- Ejecución de aplicaciones en el kernel con un API de llamadas propias (eBPF)

Llamadas al Sistema (I)

- El kernel se ejecuta en el *modo kernel o privilegiado* de la CPU; con acceso total a los recursos del sistema (i.e. cualquier instrucción del repertorio, acceso a región de memoria o operación de E/S).
- El resto de las aplicaciones se ejecuta en el *modo usuario* de la CPU con acceso restringido al sistema.



Llamadas al sistema

- El mecanismo que permite a las aplicaciones ejecutar acciones privilegiadas.
- El sistema incluye interfaces *sencillos* para acceder a esta funcionalidad (p.ej. libc).
- Ejemplos: write(2), read(2), open(2), stat(2), connect(2)*...
- `strace(1)` permite ver que llamadas al sistema hace un programa.

*(2) hace referencia a la sección del manual; la 2 es llamadas al sistema, 1 es la comandos y utilidades

Llamadas al Sistema(II)

```
#include <iostream>

int main()
{
    std::cout << "HOLA MUNDO!\n";
    exit(0);
}
```

Aplicación

Las llamadas al sistema se generan por una **interrupción software** con una instrucción especial. En x86_64 la instrucción es `syscall`

`libstdc++.so` **Librería de sistema**

```
...
; ssize_t write(int fd, const void *buf, size_t count)
mov     rax, 1      ; rax registro con la llamada 1 = write
mov     rdi, 1      ; fd: 1 salida estándar
mov     rsi, msg    ; bf: puntero al buffer con la cadena
mov     rdx, msglen ; count: longitud de la cadena
syscall             ; llamada al sistema (al kernel)
...
```

Algunas de las llamadas al sistema del programa:

```
$ strace ./hola_mundo
...
write(1, "HOLA MUNDO!\n", 12)           = 12
exit_group(0)                           = ?
+++ exited with 0 +++
```

Más información en:

- syscall(2)
- syscalls(2)

Procesos

- Un proceso (~1960 Multics) es el entorno de ejecución de un programa de usuario:
  - Espacio de direccionamiento de memoria
  - Descriptores de ficheros en uso
  - Registros de la CPU
  - Pila de memoria (*por thread*[*]*)*
  - Estado incluye todos los atributos que usa el kernel para su gestión
- El desarrollo del concepto de proceso permite al SO:
  - Mejorar la eficiencia del uso de los recursos (***multiprogramación***). Ej. permite que un proceso use la CPU mientras otro hace un operación de E/S.
  - Uso en ***tiempo-compartido (multitarea)*** de los recursos del sistema para soportar la ejecución simultánea de múltiples programas (y de múltiples usuarios)
  - Desarrollo de sistemas de ***tiempo real*** para asegurar la ejecución determinista (planificación) y aislada de aplicaciones.

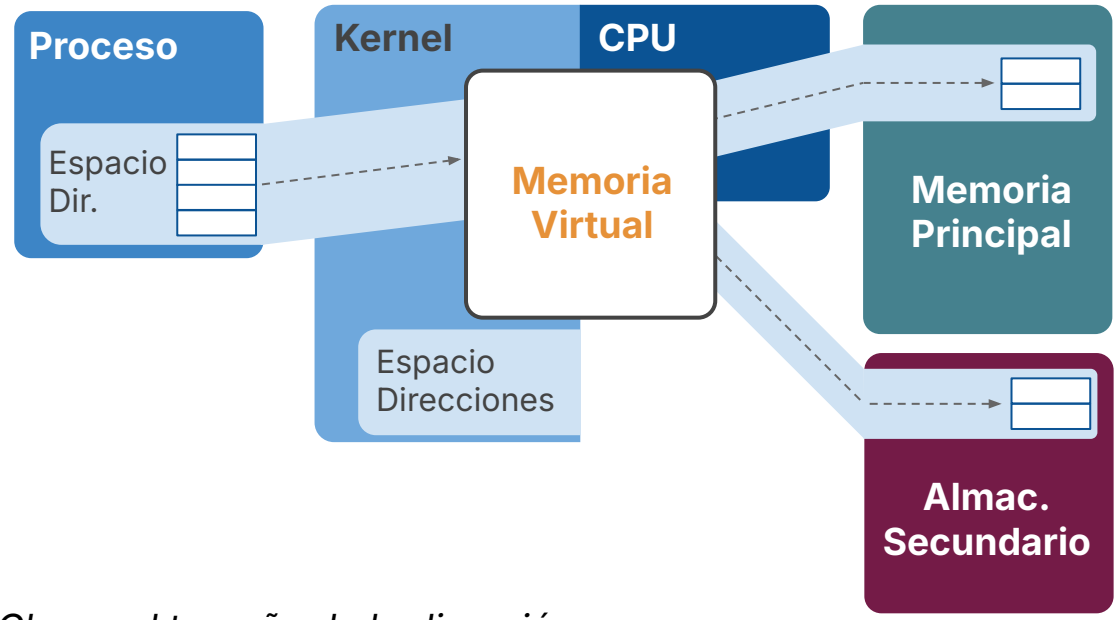
[*]En Linux *procesos y threads* se representan con la misma estructura *"task"*. Los threads de un proceso comparten el espacio de direcciones y descriptores; y permiten a un proceso usar múltiples CPUs.

Memoria Virtual

- La memoria virtual es una abstracción que permite que los procesos (y el kernel) tengan la ilusión de tener una memoria principal ilimitada* y exclusiva.
- El sistema de memoria virtual se implementa por la CPU y el sistema operativo.
- La memoria virtual permite:
  - **Multitarea**, ya que los procesos trabajan con su espacio de direcciones privado sin conflictos.
  - **Sobresuscripción** (*over-provisioning*) de la memoria principal usando almacenamiento secundario (discos).

Esquemas de Implementación

- *Process swapping*. Se mueven los procesos completamente entre la memoria y disco
- *Paginación*. Mover pequeños segmentos de memoria (páginas, p.ej. 4K).



*En arquitecturas de 32bits el límite es de 4Gb por el tamaño de la dirección

Planificación

- En un sistema de ***tiempo-compartido***, el planificador es el responsable de asignar los procesos (*threads)* a las CPUs del sistema.
- Objetivos del planificador
  - Dividir los *ciclos* de CPU entre los procesos activos.
  - Establecer *prioridad* (dinámica) de unos procesos sobre otros. Los threads del kernel tienen más prioridad que los de usuario.
- Caracterización de los procesos:
  - **Intensivo CPU**, principalmente usan operaciones de cálculo.
  - **Intensivo en E/S**, principalmente realizan operaciones de E/S (p.ej. red or disco).

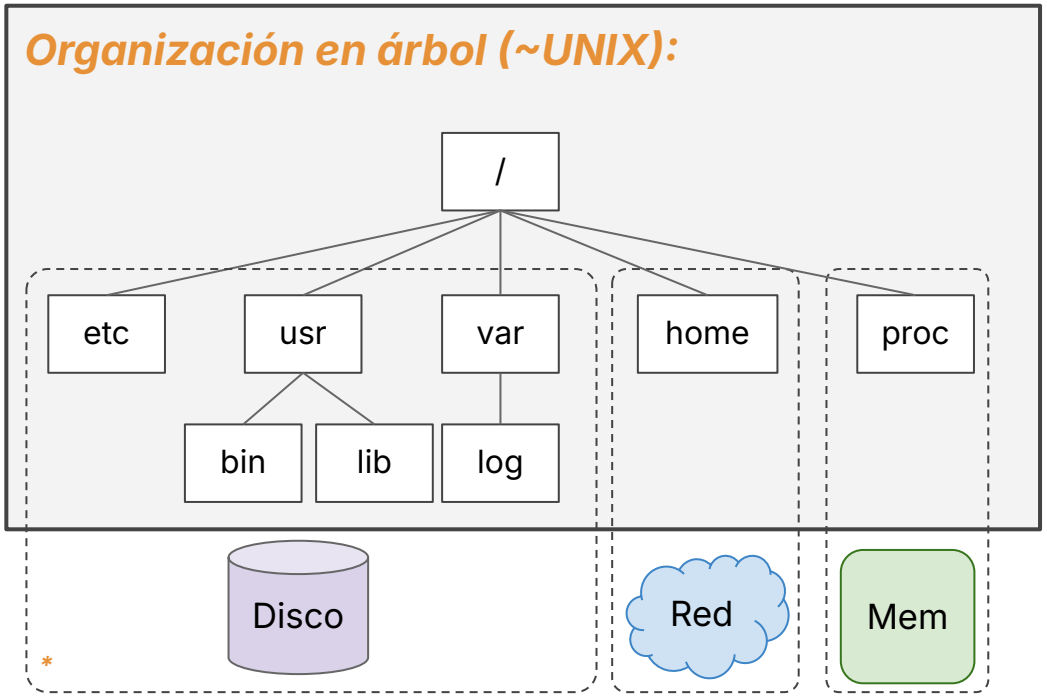
Ejemplo (UNIX): El planificador aumenta la prioridad de las aplicaciones intensivas E/S para intentar reducir su *latencia*. Para cada proceso

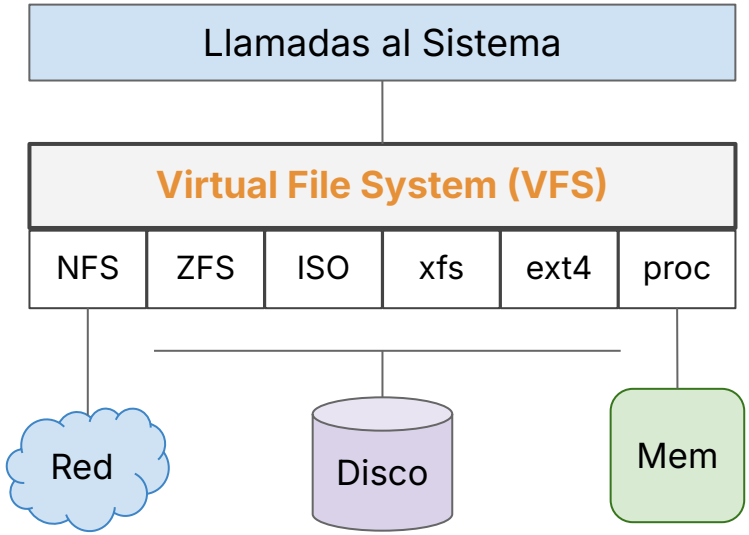
Tasa CPU = tiempo CPU / tiempo real,

reduciendo la prioridad de los procesos con una Tasa CPU alta.

Sistemas de Ficheros

- Un sistema de ficheros permite organizar datos en ficheros y carpetas.
- El sistema de ficheros ofrece un interfaz (***POSIX***) sobre estos ficheros
  - Abrir, cerrar, leer, escribir..
- El kernel soporta distintos sistemas de ficheros accesibles desde un interfaz común independiente del tipo.

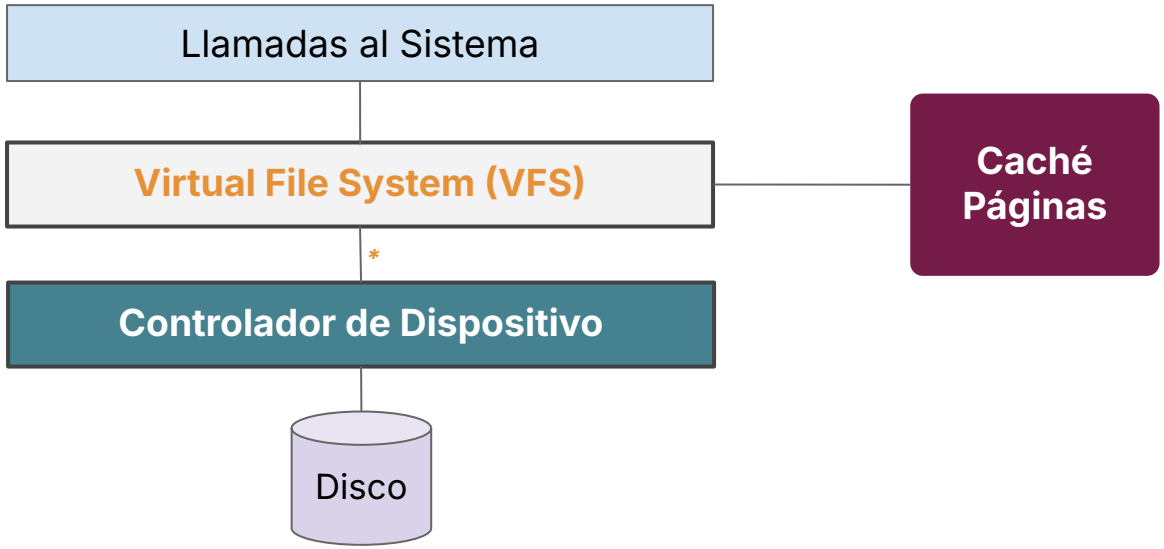




* Un sistema de ficheros accesible en el árbol está *montado*

Controladores de Dispositivos

- El kernel interactúa (gestión y operaciones E/S) con los dispositivos conectados al sistema mediante controladores (*drivers)*.
- Normalmente permiten la carga dinámica (sin reinicio)
- Tipos de dispositivo
  - **Modo caracter (*raw*).** Acceso secuencial sin *buffer* de cualquier tamaño (hasta byte) según el dispositivo, p.ej. teclados, impresoras...
  - **Modo bloque.** Acceso aleatorio (*offset)* en bloques (p.ej. 512 bytes). UNIX originalmente ofrecía caché de los buffers de dispositivo para mejorar el rendimiento*

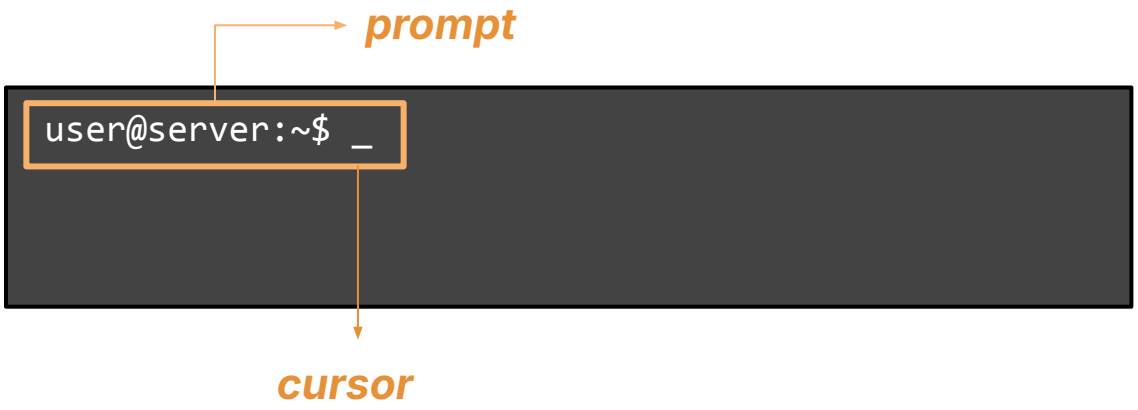


* *En Linux el VFS no usa directamente el controlador de disco, lo hace a través de un gestor de E/S de bloques y el planificador de E/S*

TEMA 1.2 Intérprete de Comandos (*Shell*)

Interfaz de Línea de Comandos.

- La línea de comandos (CLI, *command line interface*) es esencial para interactuar con los objetos (procesos, ficheros, dispositivos...) del sistema operativo.
- El elemento principal de la CLI, es la *shell*. La *shell* es un programa que:
  - Recibe, interpreta y ejecuta los comandos (externos o *built-in,* implementados por la *shell*)
  - Ofrece un lenguaje de programación (*scripts*)
  - Establece el entorno de ejecución de los programas (p.ej. `PATH`, `CWD`...)
  - Servicios para la gestión del flujo de datos (tuberías y redirecciones)
  - Ejemplo: **bash** *(Bourne Again Shell)*, **zsh** *(Z Shell)*, sh, fish, tcsh, ksh,...



Información de los Comandos

- Las llamadas al sistema y las funciones de biblioteca están documentadas en las páginas de manual (ver `man man`):
  - **Sección 1: Comandos y aplicaciones**
  - **Sección 2: Llamadas al sistema**
  - **Sección 3: Funciones de biblioteca**
  - Sección 4: Dispositivos y ficheros especiales
  - Sección 5: Formatos de ficheros y convenciones
  - Sección 6: Demostraciones y juegos
  - **Sección 7: Miscelánea (convenciones, protocolos, señales...)**
  - Sección 8: Comandos de administración del sistema (superusuario)
  - **Sección 9: Documentación del núcleo y desarrollo de *drivers***
- El formato general de consulta es: `man [`*`sección`*`] `*`página`*
- La sección del manual se especifica seguida de la página, en la forma: `open(2)`
- El uso de `-k` *`keyword`* es útil para buscar páginas específicas.

Variables de Entorno (I)

- Todos los procesos (y por tanto la *shell*) tiene asociado un ***entorno***, definido por una lista de pares nombre valor (**variable de entorno**). Ejemplos:
  - USER, el nombre de usuario
  - HOME, el directorio del usuario
  - SHELL, la shell en uso
- El valor de una variable se puede acceder con el operador **$** (`$USER`)
- El valor de una variable se fija con el operador **=** (`MI_VARIABLE="Hola"`)

El entrecomillado en la shell

- Comillas dobles “, definen cadenas con resolución de variables

  `“El usuario es $USER”`
- Comillas sencillas ‘ definen cadenas sin resolución de variables

  `‘El usuario es $USER’`
- Comillas de ejecución ` sustituye el valor por la ejecución del comando

  `“El usuario es ` whoami ` ”` **Nota**: `$(whoami)` es equivalente

Variables de Entorno (II)

- Los comandos son ficheros **ejecutables (+x)**
- La shell busca los comandos en la lista de directorios definidos por `PATH`
- Un comando se puede ejecutar siempre (aunque no esté en el `PATH`) usando la ruta
  - **relativa** `./programa1`
  - **absoluta** `/home/user/so/programa1`
- `which` es un comando que permite determinar el fichero que se usará para un comando

Algunos Comandos Importantes

- cat
- wc
- head
- tail
- tr
- sed
- sort
- cd
- ls
- mkdir
- rmdir
- cp
- find

> "*Make each program do one thing well. To do a new job, build a fresh program rather than complicate old programs by adding new features.*"
>
> — Doug McIlroy, co-autor de Unix en Bell Labs

Comandos del sistema de ficheros.

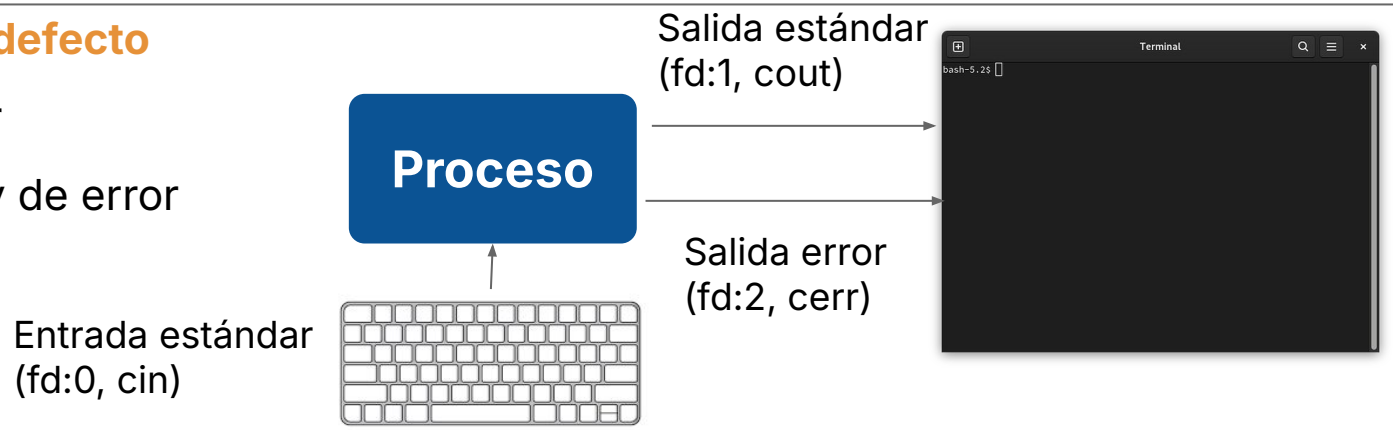
- La *shell* realiza la expansión (*globbing*) de caracteres comodín, **antes de la ejecución del comando.**
- **Patrones soportados**
  - * 0 o más caracteres
  - ? 1 sólo caracter
  - [abc] Un caracter del conjunto
  - [a-z] Un caracter en el rango
  - [!0-9] Un caracter que no esté en el conjunto

Redirecciones y Tuberías

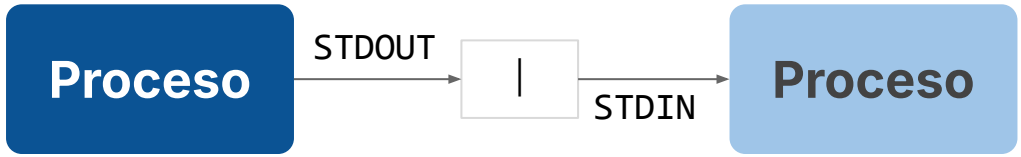
"Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently structured binary input. Don't insist on interactive input."

Flujos (streams) por defecto

- Entrada estándar
- Salida estándar y de error

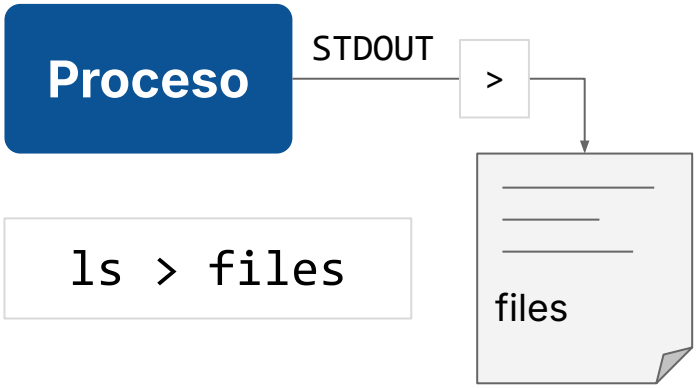


Tuberías




```
ls -1 | sort
```

Redirecciones




```
ls > files
```

Expresiones Regulares

- Una expresión regular define un patrón que se busca en un flujo o archivo.
- El comando principal para buscar patrones es `grep`

Caracteres y grupos	a, b, [aA], [0-9], [A-Za-z], [:blank:], [:alnum:]
Posicionamiento (anclas)	^ principio de línea $ final de línea \< ppio de palabra \> final de palabra
wildcards	. cualquier carácter * el patrón anterior 0 o más veces + el patrón anterior 1 o más veces
Repeticiones	{N} {N,} {N,M} el patrón se repite N veces, N veces o más, N veces y no más de M

Ejemplo. Determinar el propósito de `"^[0-9]{1,2}$"`

Scripts Shell

```
#!/bin/bash
# --- Variables ---
DIR=$1
EXT=$2
# Comprobación de argumentos
echo "Buscando en el directorio: $DIR"
echo "Archivos con extensión: .$EXT"
if [[ ! -d "$DIR" ]]; then
    echo "Error: El $DIR no existe."
    exit 1
fi
# Usar find para la búsqueda
find "$DIR" -type f -name "*.${EXT}"
exit 0
```

Los scripts comienzan definiendo el intérprete (*shebang #!)*. El kernel es el responsable de la carga (i.e. común para otros lenguajes python...)

Las variables se definen con =, **sin espacios**. $1,$2 son los argumentos del programa

Las *shell* cuenta con las estructuras básicas `if-else`, `for`, `case`, `arrays`, funciones...

Los comandos se pueden ejecutar directamente, su salida puede almacenarse en variables o comprobarse con `$?` o ejecución condicional `||` y `&&`

Nota: Debe observarse la sintaxis del intérprete usado (bash)

TEMA 1.3 Llamadas al Sistema y Entorno de Desarrollo

El lenguaje C

Características

- Amplio uso en programación de sistemas
  - Gestión manual de memoria (`malloc`, `free`, `mmap`, `void*`, `struct`/`union`)
  - Control “preciso” de las operaciones realizadas sin sobrecarga
  - Posibilidad de interactuar con el hardware directamente
- Portabilidad y estabilidad del interfaz binario (**ABI***)
- Interoperabilidad (uso en otros lenguajes)
- Amplio ecosistema (perfilador, depurador, compiladores, analizadores...)

***Incluye: Convención para la llamada a funciones, uso de registros, disposición de la pila, tipos de datos y tamaños.

El lenguaje C

Compilación

- En este curso, GNU C compiler (`gcc`)
  - **`-g`** (incluye símbolos de depuración),
  - **`-o`** ejecutable de salida*
  - **`-l<nombre>`** enlazado con librerías (p.ej -l**m**, l**pthread**)
  - Ejemplo: `gcc -g -o ejercicio_1 ejercicio_1.c -lpthread`
- Compilación de proyectos complejos: `make`, `cmake`, `scons`

Depuración

- GNU C debugger (`gdb`)
  - Ejecutable, `core dump` o proceso (`-p`)
  - Ejemplo: `gdb ./ejercicio1`
- Comandos importantes:
  - `list`, `break`, `run [args]`, `bt` (backtrace), `print`, `frame`, `next`, `step`, `cont`

El lenguaje C

Funciones de Librería Importantes

- printf(3): Impresión con formato en *flujos* de salida (fichero, stdout, cadena)
- `scanf(3)`: Manejo de flujos de entrada
- string(3): Manejo de cadenas.

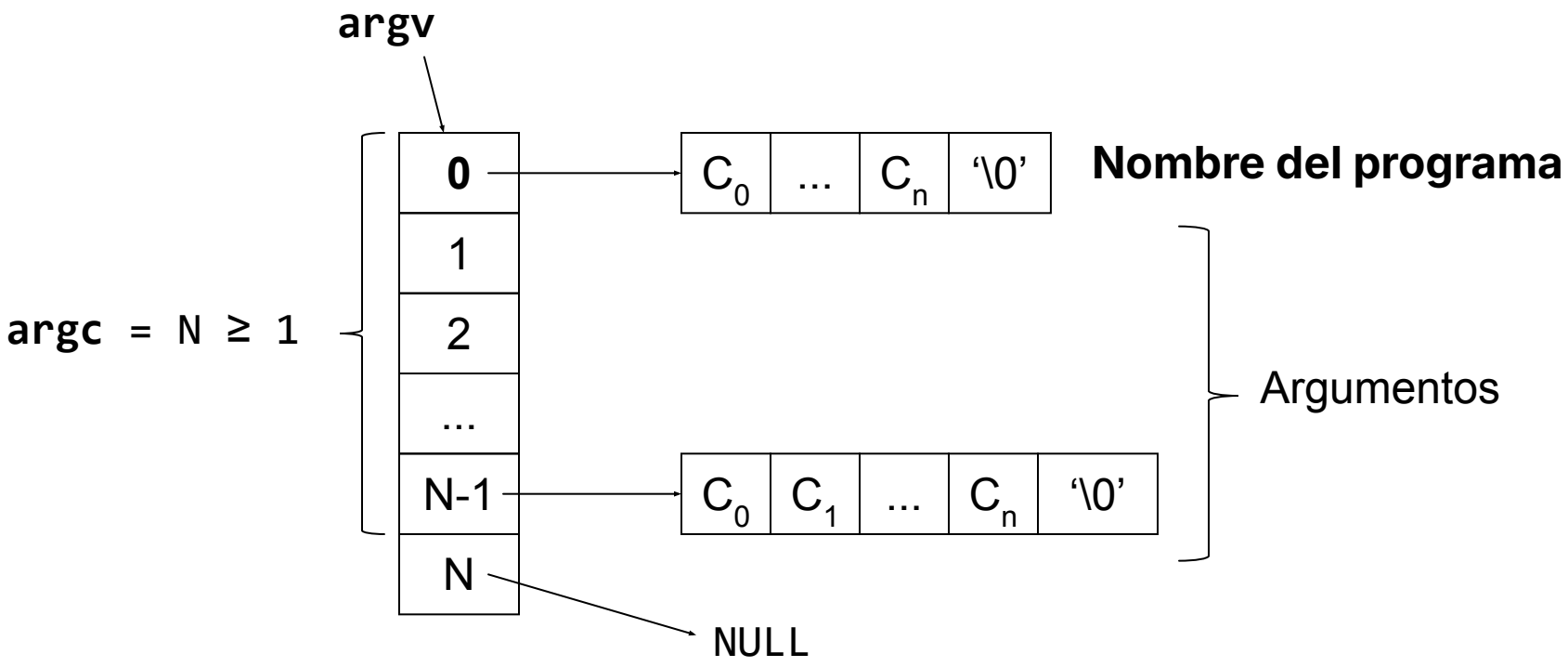
	Llamadas al Sistema	Función de Librería
Sección de manual	2	3
Área de ejecución	Usuario/Núcleo	Usuario
Espacio de parámetros	No se reserva	Dinámico/Estático
Código de error	-1 + `errno`	`NULL` + no `errno`

El lenguaje C

Argumentos

- Definición del **programa principal**:

```
int main(int argc, char **argv);
int main(int argc, char *argv[]);
```



Gestión de Errores

```
<stdio.h>
<errno.h>
<string.h>
```

- Imprimir un mensaje de error:

```
void perror(const char *s);
```

  - Imprime por la salida de error (stderr) un mensaje que describe el **último** error encontrado en una llamada al sistema o función de biblioteca
  - El formato de salida es:

  - Incluir el nombre de la función que produjo el error

- El código de error se obtiene de la variable errno, que se fija cuando se produce un error, pero no se borra cuando la llamada tiene éxito:

```
int errno;
```

- Por convenio, las llamadas al sistema* devuelven -1 cuando se produce un error
- Devolver una cadena que describe el número de error:

```
char *strerror(int errnum)
```

* Algunas funciones de librería también devuelven -1 en caso de error

Llamada Sistema. Ejemplo (I)

- Obtener información sobre el sistema operativo:

`<sys/utsname.h>`

```
int uname(struct utsname *buf);

struct utsname {
    char sysname[];    /* Nombre del SO (ej. "Linux") */
    char nodename[];   /* Nombre del host */
    char release[];    /* Nº versión del SO (ej. "3.10.0") */
    char version[];    /* Fecha versión del SO */
    char machine[];    /* Hardware (ej. "x86_64") */
#ifdef _GNU_SOURCE
    char domainname[];/* Nombre de dominio (con -D)*/
#endif
}
```

  - Almacena la información en la estructura apuntada por `buf`
  - Devuelve 0 en caso de éxito y -1 si error (**`EFAULT:`** `buf` no es válido)

- `uname(1)` proporciona acceso a esta funcionalidad y parte de la información puede obtenerse por medio de `sysctl(1)` y en los ficheros `/proc/sys/kernel/{ostype,hostname,osrelease,version,domainname}`

Llamada Sistema. Ejemplo (II)

```
<unistd.h>
<sys/types.h>
```

- Obtener los identificadores de usuario:

```
uid_t getuid(void);
```

- Los procesos disponen de un identificador de usuario (**UID**) que corresponden a los identificadores del propietario del proceso.

Llamada de Librería. Ejemplo (III)

```
<pwd.h>
<sys/types.h>
```

- Obtener **información de un usuario** de la base de datos de contraseñas:

```
struct passwd *getpwuid(uid_t uid);

struct passwd {
    char  *pw_name;   /* Nombre de usuario */
    char  *pw_passwd; /* Contraseña */
    uid_t  pw_uid;    /* Identificador de usuario */
    gid_t  pw_gid;    /* Identificador de grupo */
    char  *pw_gecos;  /* Descripción del usuario */
    char  *pw_dir;    /* Directorio "home" */
    char  *pw_shell;  /* Shell */
}
```

  - Devuelven un puntero a una estructura asignada estáticamente que puede sobreescribirse (hay versiones **reentrantes**); y `NULL` si no se encuentra el usuario o se produce un error (establece `errno`).

- Las contraseñas se almacenan encriptadas en `/etc/shadow` (sólo legible por `root` para evitar ataques) y es necesario utilizar `getspnam(3)`