Programación en C

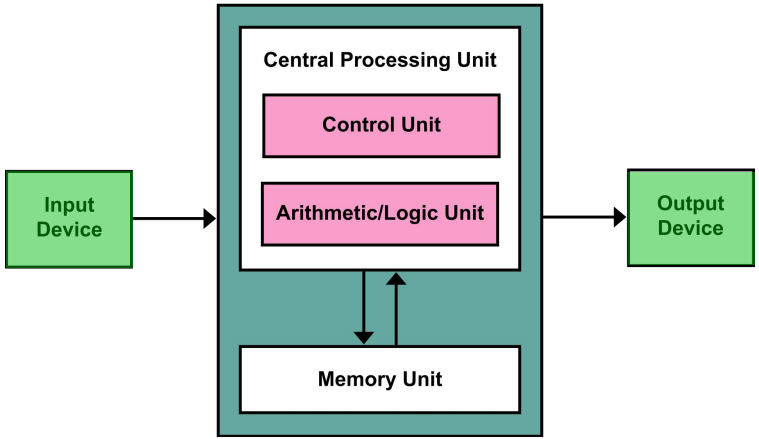
Input/Output

Arquitectura Von Neumann

Ya cubrimos el funcionamiento de la memoria de un proceso, ahora nos queda ver cómo podemos utilizar los dispositivos de input y output para que el proceso realice un cambio en el exterior.

Esto incluye dos aspectos:

- Hablar con el Sistema Operativo (system calls)
- Interactuar con el dispositivo (drivers)



Everything is a file

Unix introdujo la idea de que “todo es un archivo”. Linux y MacOS que están basados en Unix siguen esta idea bastante cerca.

Además de tener todos nuestros documentos, videos, imágenes, etc, en el filesystem, también tenemos acceso a los dispositivos conectados a la computadora. Estos se encuentran en el directorio `/dev/`.

Estos dispositivos no sólo aparecen en nuestro filesystem, sino que también podemos interactuar con muchos de ellos de la misma manera que lo hacemos con un archivo.

System calls

Además de alocar memoria, el Sistema Operativo también provee un set de system calls para poder interactuar con archivos. Cómo *todo es un archivo*, estas mismas system calls son las utilizadas para interactuar con cualquier dispositivo.

Las principales son:

- `open`: abre un archivo.
- `close`: cierra un archivo.
- `read`: lee una cierta cantidad de bytes del archivo.
- `write`: escribe una cierta cantidad de bytes en el archivo.

open

```
int open(const char *pathname, int flags);
```

open abre el archivo que se encuentre bajo el `pathname` dado.

“Abrir” un archivo quiere decir que reservamos los recursos necesarios para interactuar con él.

El parámetro `flags` es un int que representa las opciones elegidas para abrir el archivo, por ejemplo permisos para leer o escribir.

Retorna un **file descriptor** o -1 para indicar error.

File descriptors

Los file descriptors son ids que identifican los archivos abiertos por un proceso. Cada file descriptor es una referencia a una entrada en la tabla de archivos abiertos que es mantenida por el Sistema Operativo.

Dentro del mismo proceso, podemos utilizar un file descriptor para realizar system calls consecutivas sobre el mismo archivo.

El número máximo de file descriptors que pueden ser obtenidos por un proceso al mismo tiempo es 1024.

close

```
int close(int fd);
```

`close` cierra el archivo referenciado por el file descriptor `fd` dado.

Si `open` es el equivalente a `malloc` para archivos, `close` es `free`. Al “cerrar” un archivo liberamos todos los recursos utilizados para interactuar con ese archivo.

Retorna 0 si la acción fue exitosa, -1 si hubo un error.

Más leaks

Al igual que con el manejo de memoria, es posible leakear file descriptors. Si no cerramos un archivo después de utilizarlo vamos a consumir esos recursos hasta que el programa termine.

Esto incluye:

- Memoria extra consumida por el Sistema Operativo para la tabla de archivos abiertos.
- La cantidad de file descriptors que el proceso puede tener.

read

```
ssize_t read(int fd, void *buf, size_t count);
```

`read` intenta leer un máximo de `count` bytes del file descriptor `fd` y guardarlos en el buffer `buf`.

En caso de éxito, retorna la cantidad de bytes leídos, en caso de error retorna -1.

write

```
ssize_t write(int fd, const void *buf, size_t count);
```

`write` intenta escribir un máximo de `count` bytes del buffer `buf` en el file descriptor `fd`.

En caso de éxito, retorna la cantidad de bytes escritos, en caso de error retorna -1.

Seeking

Ciertos archivos soportan **seeking** (seekable), esto quiere decir que podemos movernos al offset de bytes que queramos en el contenido del archivo.

Generalmente, estos son los archivos que tienen un contenido persistente, como por ejemplo un documento o una imagen.

Cómo todo es un archivo, también podríamos estar leyendo datos de una descarga de internet, en cuyo caso no podemos hacer seeking.

Seeking

Los archivos que son seekable tienen relacionados con su file descriptors un file offset, que indica la posición actual en el archivo como la cantidad de bytes desde el inicio del archivo.

Al ejecutar `read` o `write` este offset es modificado según la cantidad de bytes que fueron respectivamente leídos o escritos.

Además, estos archivos brindan la posibilidad de utilizar una nueva system call: `lseek`.

Ex 3.1, 3.2

lseek

```
off_t lseek(int fd, off_t offset, int whence);
```

`lseek` reposiciona el file offset relacionado con el `fd` dado al argumento `offset`, según las directivas especificadas por `whence`:

- `SEEK_SET`: setea el file offset a `offset`.
- `SEEK_CUR`: suma `offset` al file offset actual.
- `SEEK_END`: setea el file offset al tamaño del archivo + `offset`.

Retorna el file offset resultado o -1 en caso de error.

Standard files

Más allá de los archivos que podemos abrir durante la ejecución del programa, todo proceso está conectado a tres archivos desde que arranca.

Standard in, standard out y standard error. Resumidos como stdin, stdoud y stderr respectivamente. Podemos acceder a sus file descriptors mediante las directivas `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO`.

Por defecto, estos archivos van a ser asignados a la consola que inicia el proceso.

`errno`

Las librerías estándar suelen seguir la convención de retornar -1 en caso de error. ¿Cómo distinguimos los distintos errores que pueden suceder?

Además de indicar error con -1, las funciones setean una variable global llamada `errno` con el código de error correspondiente al problema.

También se provee la función `perror` que printea un mensaje de error según el código de error guardado en `errno` y `strerror` que devuelve un string explicando el código.

Ex 6.1, 6.2, 6.3

stdio

stdio

A la hora de distribuir nuestro programa a distintos sistemas, no deberíamos depender directamente de las system calls. Aunque en Linux y MacOS son las mismas, Windows ofrece system calls diferentes.

Para cubrir esa diferencia tenemos que usar las funciones provistas por `stdio.h`.

Son muy similares a las system calls que ya vimos, pero en vez de utilizar file descriptors, utilizan **`FILE*`**.

Además de mayor portabilidad, ofrecen buffered I/O.

Buffered I/O

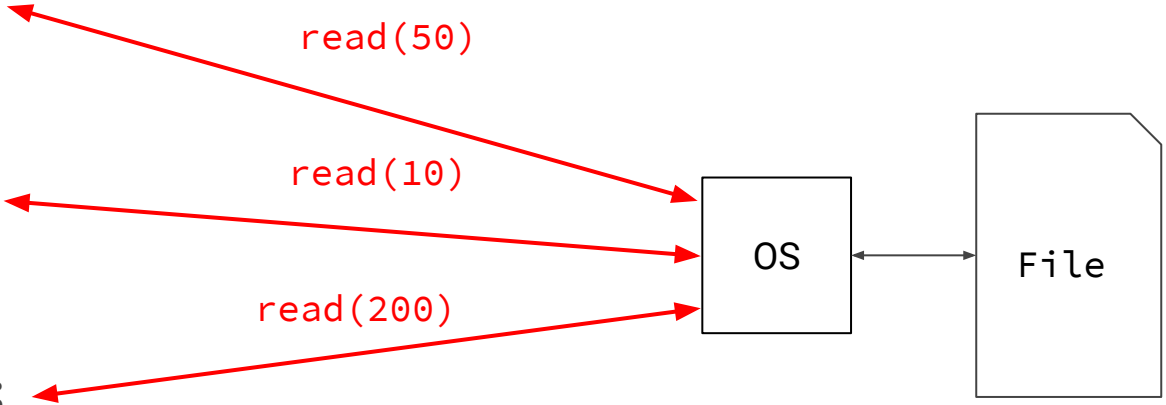
Siempre va a ser necesario, tal vez detrás de escenas, realizar una system call para poder realizar acciones de input/output. Sin embargo, como vimos antes, cada system call introduce un overhead significante.

En vez de realizar una system call cada vez que deseamos leer o escribir algo en un archivo, `stdio` va a utilizar un buffer interno para disminuir la cantidad de system calls.

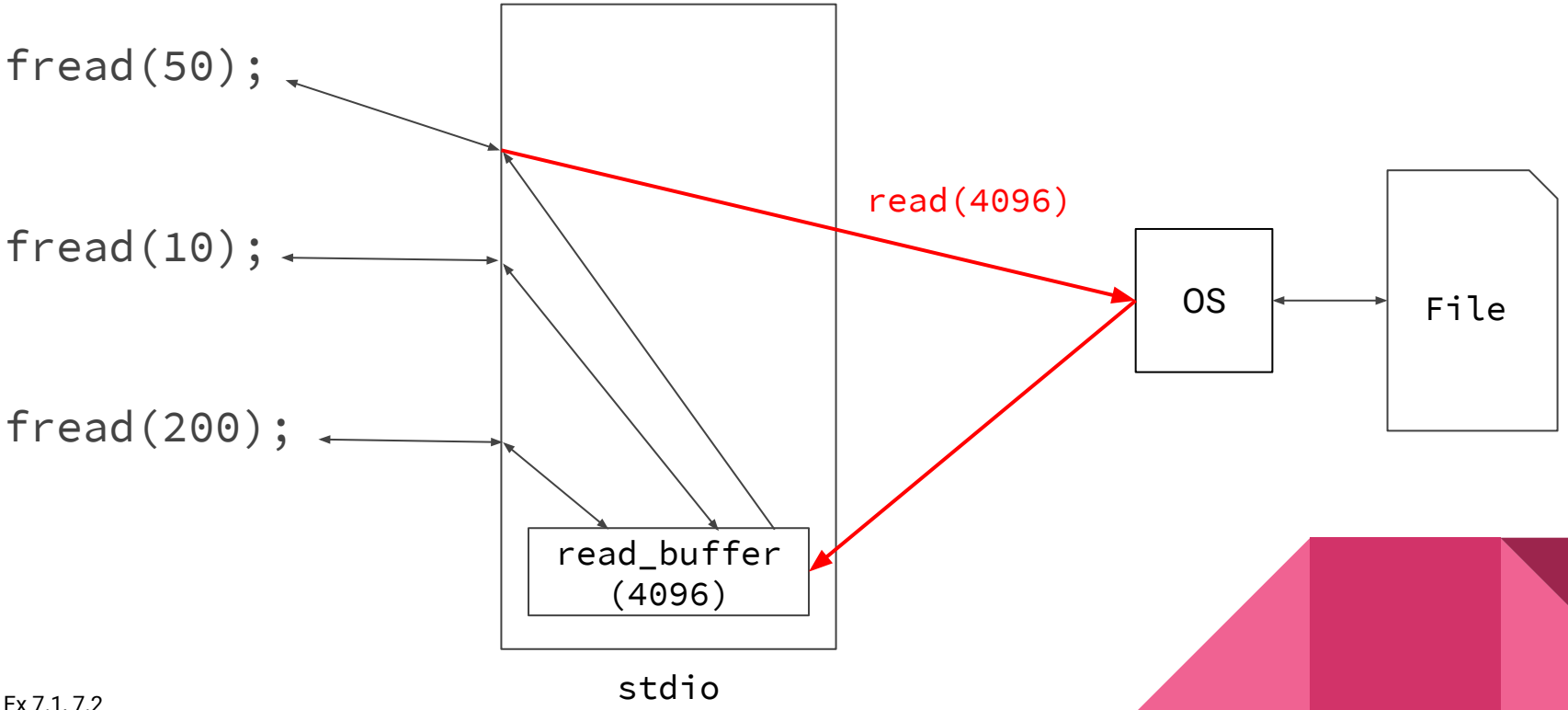
- Para leer, lee más de lo necesario y lo guarda en el buffer.
- Para escribir, espera a llenar el buffer.

read

```
read(50);
read(10);
read(200);
```



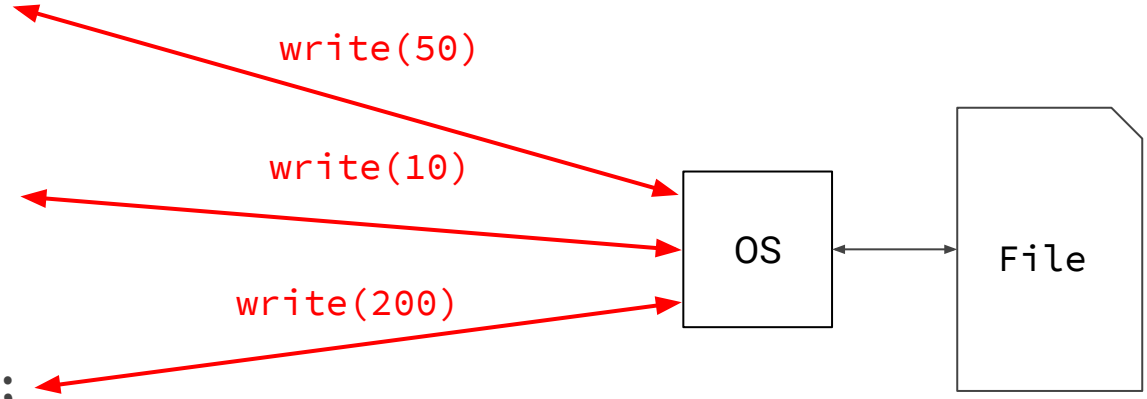
Buffered read con `fread`



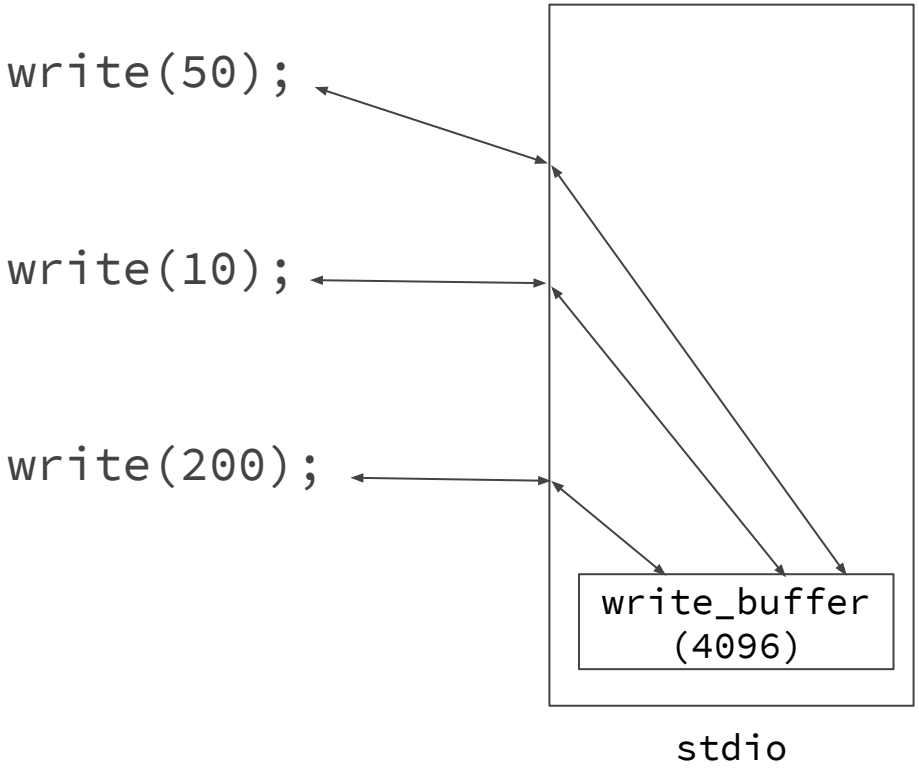
Ex 7.1, 7.2

write

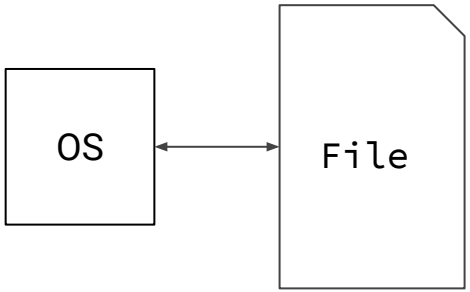
```
write(50);
write(10);
write(200);
```



Buffered write con `fwrite`



???

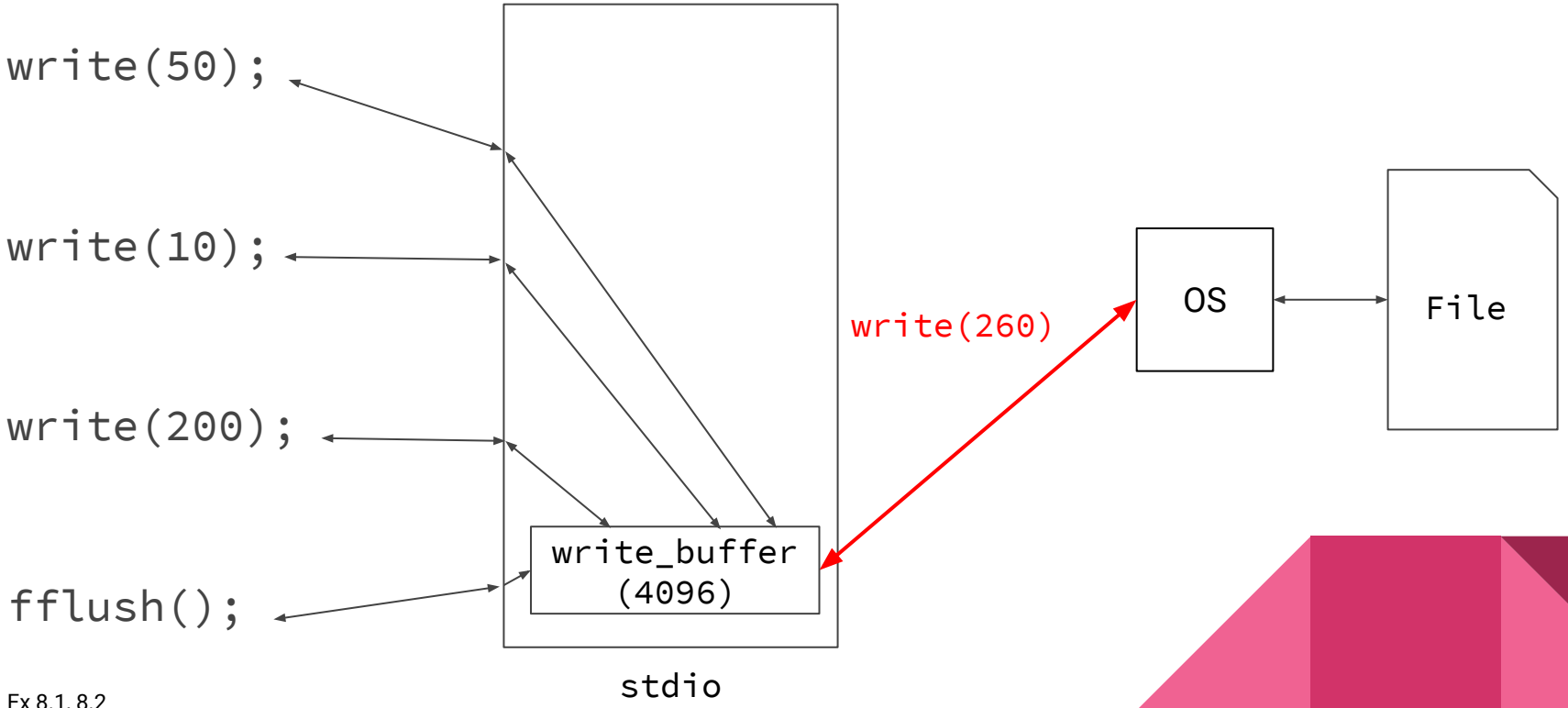


Buffered write con `fwrite`

Para escribir `stdio`, espera a **llenar el buffer**. Cómo no podemos depender del tamaño del buffer interno de `stdio`, necesitamos una forma de indicar que deseamos escribir al archivo. Este es el objetivo de la función `fflush`.

Al llamar a `fflush` forzamos la escritura al archivo, sin importar cuanto hemos escrito en el buffer.

Buffered write con `fwrite`



Ex 8.1, 8.2

Buffered write con `fwrite`

Si no llamamos a `fflush`, todo el contenido restante del buffer será escrito al llamar a `fclose`.

En los casos donde no llamamos a ninguna de las dos funciones, `stdio` va a escribir el archivo al terminar el programa **normalmente**.

Si el programa crashea y quedan datos por escribir en el buffer, estos no serán escritos en el archivo.

stdin, stdout y stderr

Al usar system calls podemos usar las directivas de `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO` para hacer input/output a través de estos archivos.

Cuando usamos `stdio` tenemos acceso a `FILE *` con los nombres `stdin`, `stdout` y `stderr` para hacer lo mismo.

Drivers

Drivers

Cuando el Sistema Operativo recibe una system call para realizar una operación de input/output, debe redirigir esta solicitud al dispositivo indicado (según el file descriptor dado).

Para hablar con un dispositivo de hardware, el Sistema Operativo necesita una interface fija, que no dependa del hardware con el que está hablando. Esto le permite adaptarse a cambios físicos en la computadora sin necesidad de actualizaciones.

Para lograr esto cada dispositivo debe proveer un **driver**.

Drivers

Un driver es una interface de software que permite controlar un dispositivo de hardware u otro tipo de servicio de bajo nivel.

Cada Sistema Operativo determina la interface necesaria que debe implementar un driver para lograr que su dispositivo pueda ser utilizado.

Al tener una interface fija para todos los drivers, el Sistema Operativo puede interactuar directamente con el driver sin importar las diferencias en el hardware subyacente. Existen distintos tipos de drivers, pero es un número mucho menor al de posibles dispositivos.

Los drivers del mismo tipo son **polimórficos**.

Character drivers

Cómo ejemplo vamos a ver character drivers. Este tipo de drivers controlan character devices, que realizan operaciones de input/output en sequencias de bytes. Esto los diferencia de block drivers, que realizan sus operaciones en bloques.

Algunos dispositivos que utilizan estos drivers son: placas de video, de audio, cámaras, impresoras, y más. La mayoría de los dispositivos que no son de almacenamiento o de red, son character devices.

Character drivers

Una particularidad de los character drivers es que tienen un mapeo directo a las system calls, incluyendo las 4 que vimos:

- `open`
- `read`
- `write`
- `release (close)`

Polimorfismo en C

Cómo vimos en la primera clase, el núcleo de los Sistemas Operativos, así como los drivers, están implementados en C.

¿Cómo podemos lograr polimorfismo en C?

Ya vimos que podemos pasar diferentes structs tomando ventaja de sus estructuras en memoria, pero para los drivers necesitamos polimorfismo con todas las letras. Necesitamos la posibilidad de llamar a una función diferente dependiendo el driver con el que estamos interactuando.

Function pointers

C nos permite pedir una referencia (pointer) a una función. Este pointer luego podemos guardarlo en una variable con el tipo “pointer a función”, que especifica la signature de la función a la que apunta el pointer.

Si tenemos una función:

```
void function() { ... }
```

Podemos tomar su referencia y guardarla en una variable así:

```
void (*f_ptr)() = &function;
```

Ex 10.1, 10.2, 10.3, 10.4

Polimorfismo en C

Combinando function pointers con las técnicas que aprendimos de structs:

```
struct file_operations {
    ssize_t (*read)(int fd, void *buf, size_t count);
    ssize_t (*write)(int fd, const void *buf, size_t count);
    int (*open)(const char *pathname, int flags);
    int (*close)(int fd);
};
```

Ex 11.1

Polimorfismo en C

Los character drivers usan un struct muy similar donde definen todas sus funciones.

Cuando el Sistema Operativo arranca, registra los struct de cada uno de los drivers y cuando el usuario realiza una system call redirige la solicitud al struct correspondiente según el file descriptor / pathname provisto por el usuario.

Polimorfismo en C

A diferencia de lenguajes orientación a objetos, este tipo de diseño en C es la **excepción y no la regla**.

La flexibilidad y extensibilidad obtenida se paga en términos de performance y mantenibilidad del boilerplate necesario para permitir esta funcionalidad. Desde un principio, si estamos usando C nuestro objetivo debería ser otro.

Un lenguaje se comporta de la mejor manera cuando es utilizado para el propósito que es diseñado. Esta es la razón por la que existen tantos lenguajes de programación y es importante saber elegir cuál usar.