# Persistencia

Ticiano Ian Morvan

Octubre 2024

## 1 Dispositivos de entrada/salida

Introducimos el concepto de **dispositivo de entrada/salida** (ó I/O, del inglés *input/output*), que es todo aquel que reciba un valor de entrada y devuelva uno de salida. La existencia de ambos es crucial, puesto que sin un valor de entrada, la salida sería la misma siempre; mientras que sin un valor de salida, no habría sentido para la ejecución del dispositivo.

### 1.1 Arquitectura del sistema

Teniendo en cuenta que mientras más rápido sea un *bus* de datos, más corto debe ser, planteamos una jerarquía en la que los dispositivos se encuentran más lejos del CPU a medida de que su velocidad disminuye. Es entonces que, en sistemas modernos, vemos que la memoria principal está casi a la par del procesador, siguiéndole otras conexiones como un *bus* general de I/O, como el **PCI**, gráficos y finalmente un *bus* de periféricos, como **SCSI**, **SATA** o **USB** que abarcan **discos**, el ***mouse*** y el **teclado**.

### 1.2 Dispositivos canónicos

Un dispositivo canónico (es decir, no uno real) nos permitirá entender los elementos requeridos para hacer una interacción eficiente con éste. Tendremos, en particular, dos partes esenciales: una **interfaz** y una **estructura interna**. La interfaz del dispositivo se presenta al resto del sistema, permitiéndole al *software* de éste controlar sus operaciones, por lo que todos los dispositivos especifican una interfaz y un protocolo para interactuar con ellos. Por otro lado, la estructura interna es específica de su implementación y es responsable de la abstracción del dispositivo que se presenta al sistema. Puede ir desde un par de *chips* hasta una pequeña computadora integrada, con procesador, memoria y otros *chips* específicos.

### 1.3 Protocolo canónico

La interfaz de este dispositivo tiene, necesariamente, tres registros: un registro de **estado**, que comunica el estado actual del dispositivo; un registro de **comandos**, que permite comunicarle una instrucción al dispositivo y un registro de **datos**, para enviar u obtener información del dispositivo. Típicamente, se siguen los siguientes pasos:

1. El sistema operativo espera hasta que el dispositivo pueda recibir un comando al repetidamente consultar el registro de estado. Llamamos a esto hacer ***polling*** del dispositivo.
2. El sistema operativo envía información al registro de datos. Si el CPU principal está involucrado en este envío, nos referimos a esto ***I/O programado (PIO)***.
3. El sistema operativo escribe un comando en el registro de comandos, implícitamente asegurando que la información ya está presente y que debe empezar a trabajar en el comando.
4. Por último, el sistema operativo espera que el dispositivo termina de ejecutar el comando haciendo *polling* de éste en un bucle, esperando que termine (pudiendo recibir un código que indique éxito o fracaso de la operación).

### 1.4 Optimización del protocolo

Consultar repetidamente el estado del dispositivo puede no ser la alternativa más eficiente, por lo que presentamos un concepto ya conocido: las **interrupciones**. En vez de congelar el hilo de ejecución mientras se espera por un dispositivo I/O, podemos suspender el proceso y hacer un cambio de contexto a otra tarea. Luego, cuando el dispositivo termine su operación, lanzar una interrupción de *hardware* que será administrada por una ***interrupt service routine*** *(ISR)* o simplemente un manejador de interrupciones. Éste último es una pieza de código del sistema operativo que deberá ejecutarse para terminar la operación y, en función del resultado, despertar al proceso y continuar con su ejecución. Los beneficios del cambio de contexto mientras se espera por un dispositivo I/O ya los hemos recorrido anteriormente, pero un punto importante a tomar en cuenta es que, si el dispositivo fuese lo suficientemente rápido para que este cambio fuese *menos eficiente* que simplemente consultar por su estado. Es por eso que a veces, en función de la rapidez del dispositivo, se puede usar una u otra forma de administrarlo, incluso una forma **híbrida** que logre lo mejor de ambas.

### 1.5 Transferencias más eficientes con DMA

El motor **DMA** *(Direct Memory Access)* es un dispositivo muy específico de un sistema que se encarga de realizar transferencias entre la memoria del sistema y los dispositivos sin mucha intervención del CPU. Esto permite que, cuando tenemos procesos previos a la operación propia de I/O, podamos seguir utilizando el procesador para ejecutar otros procesos. El sistema operativo es el encargado de dictar las reglas a seguir por el DMA, de donde obtener la información en memoria, cuanta información a copiar y a que dispositivo. Una vez estipuladas, el sistema sigue ejecutando otros trabajos hasta que la DMA termina y lanza una interrupción, lo que avisa al sistema de la finalización del proceso.

### 1.6 Métodos de interacción

El sistema operativo puede tomar dos estrategias para comunicarse con los dispositivos: tener **instrucciones I/O** explícitas que permitan enviar y recibir datos de los registros del dispositivo, siendo estas instrucciones **privilegiadas** del sistema; y por otro lado tenemos **I/O *mapeada* en memoria**, donde el *hardware* posibilita acceder a los registros del dispositivos como si fueran direcciones de memoria, existiendo una estrecha relación entre el sistema que abre una solicitud de lectura o escritura y el *hardware* es quien redirecciona correctamente la dirección al dispositivo solicitado.

### 1.7 Driver del dispositivo

Un ***device driver*** es una pieza de software que define específicamente el funcionamiento de un dispositivo y permite su correcta intercomunicación con las demás partes del sistema. En diversos casos, como Linux, gran parte del código de su *kernel* está compuesto por *drivers*, que en muchos casos no se terminan utilizando y, aún peor, muchas veces son escritos por programadores *amateurs* y por lo tanto representan un gran porcentaje de los problemas del sistema. Siguiendo el caso de Linux, su sistema de archivos presenta pasos intermedios entre las aplicaciones de usuario y el propio dispositivo. Encontramos, desde la mayor abstracción hacia abajo, la **API POSIX** que define las operaciones válidas de I/O, una **intefaz de bloque genérica** que permite leer y escribir sobre bloques y por último una **interfaz de bloque específica** que define las operaciones de lectura y escritura específicas a un protocolo (como SCSI, SATA o USB, por ejemplo).

## 2 Discos duros

Los discos duros han sido la forma principal de almacenamiento persistente de información desde hace décadas y gran parte del desarrollo de sistemas de archivos está basado en su comportamiento.

### 2.1 La interfaz

Un disco duro consiste de un gran número de sectores (bloques de 512-*bytes*) que pueden ser escritos o leídos, numerados normalmente de 0 a $n-1$ donde $n$ es el número de sectores. Estos sectores son el **espacio de direccionamiento** del dispositivo. Si bien se pueden ejecutar operaciones sobre múltiples sectores al mismo tiempo, generalmente leer o escribir al menos 4KB, la única garantía es que la escritura sobre un bloque de 512-*bytes* es **atómica**, es decir que se ejecutará completamente o no lo hará, por lo

que si hay una falla eléctrica, solo una porción de la transacción se completará. Hay algunas cosas que los clientes del disco duro asumen, como que operar sobre dos bloques cercanos es más rápido que hacerlo con dos que están lejos y que acceder a bloques contiguos (es decir, una operación secuencial) es la forma más rápida de acceder, usualmente mejor que cualquier patrón aleatorio de acceso.