Una de las funciones principales de un s.o. es la de controlar todos los dispositivos de entrada/salida del ordenador proporcionando una interface entre éstos y el resto del sistema.

Esta interface ha de procurar independizar el máximo posible al usuario de las particularidades de cada dispositivo. Una manera de conseguirlo es tal como se ha visto en la práctica de Unix (asignatura ISO), donde todas las E/S se realizan con un subconjunto de operaciones (llamadas al sistema) y sobre dispositivos virtuales.

Lo que pretendemos es dar una visión del tema de entrada/salida basada principalmente en el sistema operativo Onion. El objetivo principal es complementar la bibiografía básica en aquellos puntos en los que el enfoque dado en clase no quede totalmente reflejado.

ENTRADA/SALIDA

íNDICE

1. CARACTERÍSTICAS DIFERENCIALES DE LOS PERIFÉRICOS DE E/S.

Normalmente, los dispositivos de E/S que configuran un sistema son muy diferentes, haciendo que las generalizaciones puedan resultar difíciles, por no decir imposibles, de conseguir.
Una determinada configuración puede incluir periféricos que difieran notablemente en lo que respecta a sus características y modo de operación.
A continuación se enumeran algunos de los aspectos que pueden diferir entre los diferentes dispositivos.

- Velocidad de transferencia
Existe una gran diferencia en la velocidad de transmisión de los diferentes periféricos. Un disco magnético puede transferir 10^6 caracteres por segundo, mientras que el teclado de un terminal sólo puede transferir unos pocos caracteres por segundo.

- Unidad de transferencia
La información puede transferirse en unidades de carácter, palabra, bloque, ó registro dependiendo del periférico empleado.

- Representación de los datos
Un elemento de información puede codificarse de diferentes maneras, en función del soporte de entrada/salida. Más aún, dentro de un mismo soporte pueden utilizarse diferentes codificaciones: ascii, 7 bits, 8 bits, ...

- Operaciones permitidas
Los diferentes periféricos difieren en los tipos de operaciones que pueden llevar a cabo. Por ejemplo, la impresora sólo puede ser utilizada para escritura, mientras que el terminal es usado para lectura y escritura.

- Modalidades de trabajo
Cada dispositivo puede trabajar con diferentes modalidades. Por ejemplo, un terminal puede trabajar haciendo el *echo* de los caracteres y una impresora puede trabajar en modo *spool.* No todas las modalidades tienen sentido en todos los dispositivos.

- Códigos de error
Es difícil conseguir un tratamiento uniforme de los errores producidos por la gran diversidad de dispositivos que hemos visto en las lineas anteriores.

Un ejemplo de todo ésto se puede ver en el enunciado de la práctica de Onion.

2. OBJETIVOS DEL DISEÑO DE LAS E/S.

2.1 Eficiencia.

Ya que las operaciones de E/S acostumbran a ser el cuello de botella de un s.o., es deseable que sean lo más eficientes posible. Con esta idea se incorporan las técnicas de *buffering* y *spooling* (apdo. 4.3.3).

2.2 Seguridad y protección.

El s.o. ha de controlar la protección de los dispositivos (derechos de acceso, ...). También ha de controlar la seguridad del sistema. Por ejemplo, los errores producidos por uno de los usuarios no han de afectar al buen funcionamiento del resto del sistema.

2.3 Independencia de los dispositivos.

Un concepto importante en el diseño de las entradas/salidas de un s.o.es la independencia de los dispositivos. Es decir, que un mismo programa pueda trabajar con diferentes dispositivos sin tener que modificar su código.
Por ejemplo en Unix el comando: *%sort <input >output* nos permite redireccionar su entrada y su salida a cualquier dispositivo.
Llevado a un extremo, la independencia de los dispositivos permite que un programa se pueda transportar entre máquinas que utilizan el mismo s.o. (e incluso con s.o. diferentes). Ésto se conoce con el nombre de **portabilidad**.

Esta independencia la podemos ver a diferentes niveles: juego de caracteres, tipos de periféricos, y unidad de transferencia.

- Independencia del juego de caracteres
La representación interna de los datos ha de ser transparente al programador.

- Independencia del periférico utilizado
La podemos ver a dos niveles:
- Independencia entre diferentes tipos de la misma clase de dispositivo: Por ejemplo, un programa ha de funcionar igual si la impresora es láser, de agujas, ó de margarita.
- Independencia entre dispositivos de diferentes clases: Por ejemplo, un programa ha de poder recibir datos tanto de un disco como de un terminal.

- Independencia de la unidad de transferencia
El número de caracteres con que el programa hace la entrada/salida ha de ser independiente de si éste lee de un terminal que transfiere los caracteres de uno en uno, ó de si lee de un disco con una unidad de transferencia de 512 bytes.
En definitiva, un programa ha de ser independiente del dispositivo que está utilizando e incluso, ha de ser independiente de las características própias de éste.

No obstante, esta independencia no se puede conseguir al 100% debido a las grandes diferencias que existen entre los dispositivos.

3. PRINCIPIOS DE DISEÑO DE LAS E/S.

Para conseguir la independencia de dispositivos deseada, nos basamos en tres principios básicos al diseñar las entradas/salidas de un s.o. : *uniformidad de las operaciones*, *dispositivos virtuales*, y *redireccionamiento*.

3.1 Uniformidad de las operaciones.

En interés de la simplicidad, es deseable que se puedan gestionar todos los periféricos de manera uniforme. La idea es proporcionar un conjunto reducido de operaciones (llamadas al sistema), lo suficientemente generales para que sirvan a todos los tipos de dispositivo.
Un buen ejemplo lo tenemos en *Unix*. Las llamadas *open*, *close*, *read*, *write*, ... sirven para todos los dispositivos. No obstante, es difícil generalizar totalmente, así las llamadas *fcntl* ó *ioctl* permiten realizar operaciones específicas de ciertos dispositivos. Un caso similar ocurre en Onion.

3.2 Dispositivos virtuales.

- Concepto
Los programas trabajan con dispositivos virtuales en lugar de hacerlo directamente con los dispositivos físicos. El s.o. asocia un dispositivo virtual a uno físico (herencia, *open*, *exec*). Por tanto, los programas trabajan con *stdin*, *stdout*, y *stderror* a los que previamente se les habrá asociado un dispositivo.

- Operaciones

El s.o. ha de proporcionar la forma de poder asociar un dispositivo virtual a uno físico, y de poder deshacer esta asociación. Hay diferentes posibilidades:

asignar (físico, virtual)
abrir (físico, virtual, modo)
crear_proceso (ejecutable, input, output, error, ...)

Las operaciones que permiten deshacer esta asociación serían:

desasignar (virtual)
cerrar (virtual)
fin_programa()

La operación de asociar la puede realizar tanto el propio proceso como algún proceso que tenga relación de parentesco con él (herencias).

- Nombres

El nombre del dispositivo virtual lo puede dar el sistema ó el usuario.
Son *nombres lógicos* cuando los proporciona el usuario y el sistema los mantiene. Normalmente son strings.
Son *canales* cuando los proporciona el sistema. Normalmente son números.

3.3 Redireccionamiento.

Diremos que un s.o. permite redireccionar los dispositivos virtuales de un proceso cuando estos dispositivos virtuales pueden ser asignados antes de iniciar la ejecución de dicho proceso, y éste los encuentra ya abiertos.
Este mecanismo permite cambiar de una ejecución a otra los dispositivos físicos con los que trabaja un proceso, sin tener que modificar su código.
La asignación de estos dispositivos se puede hacer, según el s.o. con el que trabajemos, desde el proceso padre (creador) y/ó en la llamada al sistema de crear proceso.
Ejemplos de redireccionamiento desde el intérprete de comandos:

VMS:

```
$ assign sys$output sal.dat
$ dir
$ deassign sys$output
$ type sal.dat
   ...
```

UNIX:

```
$ ls > sal.dat
$ cat sal.dat
   ...
```

ONION:

```
$ dir > sal.dat
$ copy sal.dat
   ...
```

4. IMPLEMENTACIÓN DE LAS E/S.

En este apartado discutiremos la implementación de las entradas/salidas del sistema aprovechando el marco que ofrece el s.o. ONION, el cual utiliza canales como dispositivos virtuales.

4.1 Tabla de traducción de dispositivos virtuales a físicos.

La implementación de los dispositivos virtuales se basa en una tabla de traducción que llamamos *Tabla de Canales* (Onion trabaja con *canales virtuales*).
El s.o. mantiene para cada proceso una tabla con información de los dispositivos físicos que utiliza. El proceso accede a los dispositivos mediante el índice a esta tabla. Este índice es el que conocemos como canal. Cada entrada a esta tabla contiene información referente al dispositivo físico que tiene asociado.
Generalmente, el s.o. es el encargado de adjudicar los números de canal en función de las entradas libres a la tabla de canales en el momento de la asociación del dispositivo físico al virtual.

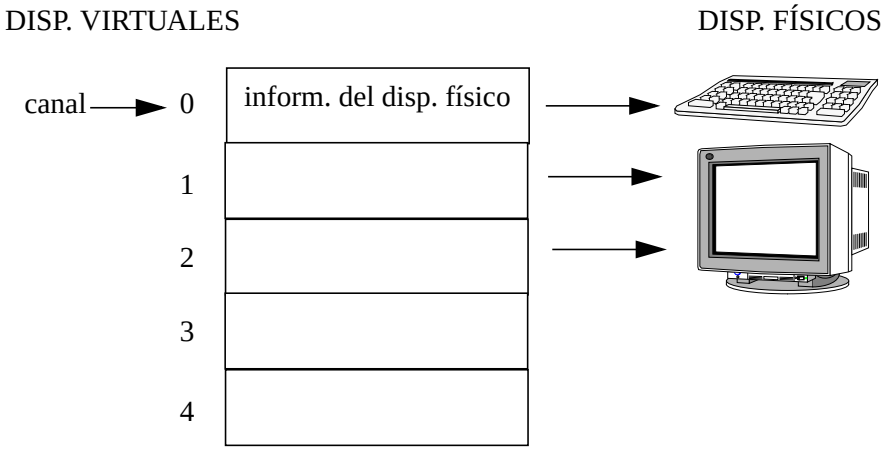

Figura 1: Tabla de Canales

Normalmente hay una tabla de canales por proceso, que acostumbra a formar parte de su PCB. Concretamente, en el s.o. Onion hay un PCB en cada nivel. En este caso, la tabla de canales estará almacenada en el PCB del nivel sistema.
Cuando lo que se tienen son nombres lógicos en lugar de canales virtuales, la implementación es similar. El s.o. mantiene unas tablas de traducción entre los nombres lógicos y los dispositivos físicos. Estas tablas se pueden organizar como una jerarquía de niveles. Por ejemplo, en el s.o. VMS hay una tabla por proceso, una por grupo de procesos, y una general al sistema. La búsqueda de un nombre se hace desde la más interna a la más externa

4.2 Implementación de las operaciones uniformes.

Para poder realizar llamadas uniformes tenemos que proporcionar algún mecanismo que permita pasar de una **llamada genérica** (llamada al sistema que realiza el proceso de usuario), a una **llamada específica** (propia del dispositivo físico asociado al canal).
En la figura 2 se muestra la idea de como hacerlo. Cuando se implementan las llamadas de entrada/salida se acostumbra a ver el sistema (nivel sistema en Onion) dividido en dos partes. Una parte **independiente de los dispositivos**, donde se encuentran las rutinas genéricas (de cada llamada al sistema), y una parte **dependiente de los dispositivos**, donde se encuentran las rutinas propias de cada dispositivo.

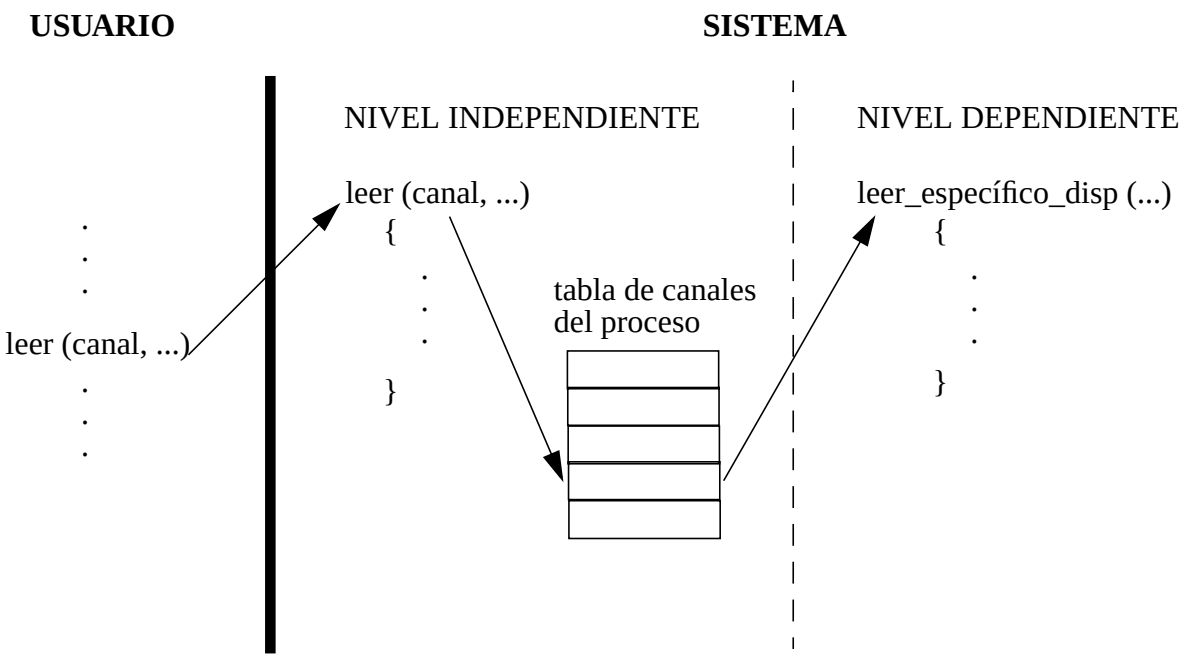

Figura 2: Esquema de la implementación de las operaciones uniformes

El usuario hace una llamada al sistema sobre un determinado canal . La rutina de la parte independiente accede mediante el canal especificado a la tabla de canales para determinar a qué dispositivo físico se quiere acceder. Una vez identificado el dispositivo se ha de ejecutar la rutina específica (dependiente) de este dispositivo. De alguna forma, las rutinas específicas de cada dispositivo han de ser identificables a partir de la tabla de canales. En estos apuntes se presentan dos opciones: (a) **por programa**, con estructuras condicionales que permiten ejecutar las rutinas de servicio específicas una vez se conozca el dispositivo físico ó (b) con estructuras de datos que conocemos como **descriptores de dispositivo**.

4.2.1 Por programa (estructuras condicionales)

La rutina genérica de E/S accede, mediante la información obtenida en la tabla de canales del proceso que ha hecho la llamada, a una estructura de código condicional que es la encargada de hacer la llamada a la rutina específica del dispositivo asociado al canal. La figura 3 muestra un esquema de esta opción.
El principal inconveniente de este método es que cada vez que se reconfigura el sistema se tienen que reescribir todas las rutinas independientes dando así poca flexibilidad.

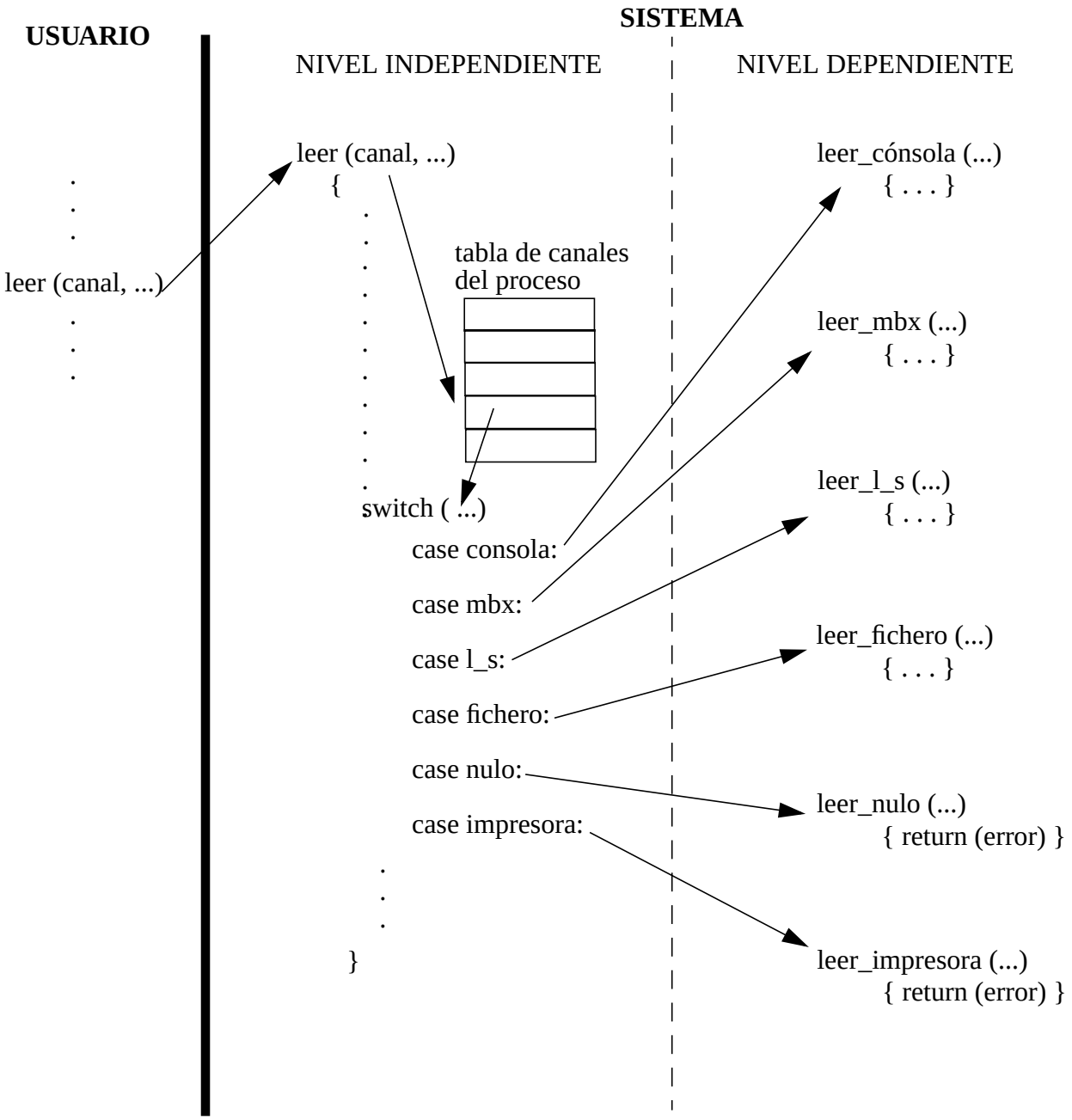

Figura 3: Implementación de las operaciones uniformes con estructuras de código condicionales

4.2.2 Con estructuras de datos (descriptor de dispositivo).

La rutina genérica de E/S después de identificar el dispositivo, llama a la rutina de servicio específica mediante la dirección que contiene una estructura de datos asociada al dispositivo. Esta estructura de datos recibe el nombre de **descriptor de dispositivo** (DD).
En este caso, si añadimos nuevos dispositivos al sistema, sólo hará falta incluir el código de las rutinas dependientes del dispositivo y su descriptor de dispositivo. Este modelo aporta mucha más flexibilidad y simplicidad que el anterior.
La misma estructura se utiliza para almacenar otros campos de información del dispositivo .La figura 4 muestra un esquema de esta opción.

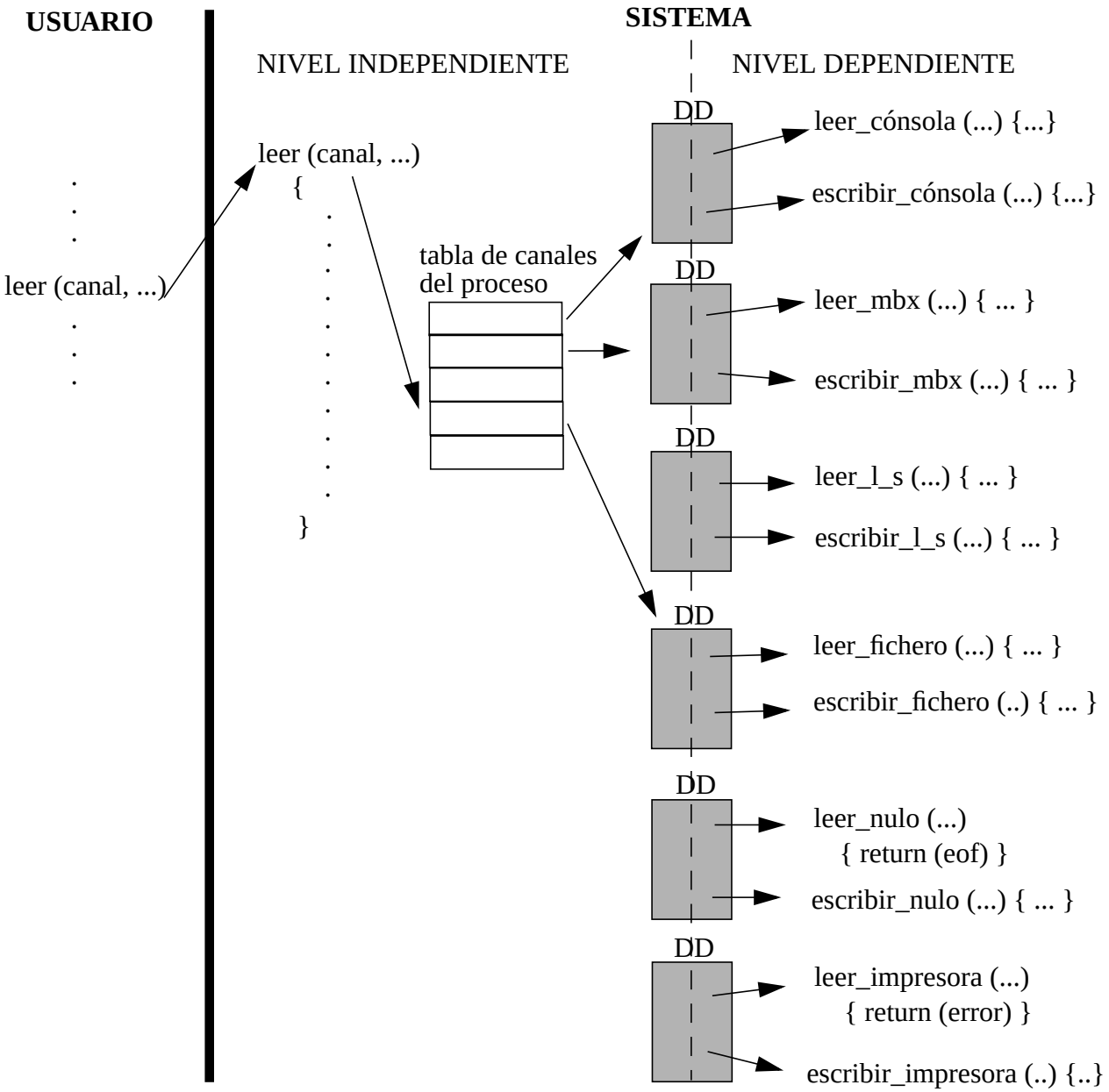

Figura 4: Implementación de las operaciones uniformes con descriptores de dispositivo (DD)

Tenemos un descriptor de dispositivo (DD) por cada dispositivo (en ficheros podemos tener más de uno). La información que se encuentra en el DD la podemos dividir en dos, la que es común a todos ellos y la que es propia de cada dispositivo. La figura 5 muestra un ejemplo de un DD relativo a un fichero y otro a impresora.

```
/* parte común */

char nom[12];
int caract;
int opens;          /* herencias */
int sem_mutex;
int sem_gestor;
int (*p_obrir) ();
int (*p_tancar) ();
int (*p_llegir) ();
int (*p_escriure) ();
int (*p_posicionar) ();
        . . .
```

```
/* parte específica ficheros */

struct cua *q_io_fin;
struct cua *q_iorb;
int modalitat;      /* modo de acceso r/w */
char buff_dd [512];
long pL_E;
int id_fitxer_bfs;
long longitud; /* longitud en bytes del fichero */
```

```
/* parte específica impresora */

struct cua *q_io_fin;
struct cua *q_iorb;
struct cua *q_pid;      /* cola pid modo Nospool*/
struct cua *q_fitx_temp; /* id_fitxer para spooler*/
int propietari;
char prot;
int proc; /*pid primer proceso que abre en modo Nospool*/
```

Figura 5: Descriptores de Dispositivo relativos a fichero e impresora

4.3 Operaciones de entrada/salida.

A continuación se describe como se pueden implementar las operaciones de entrada/salida basándose en las estructuras de datos descritas en el apartado 4.2.2

4.3.1 Secuencia de llamadas (síncronas).

La primera forma de implementar una llamada al sistema de entrada/salida es mediante una secuencia de llamadas, que empiezan en la rutina de servicio al trap, y terminan en la rutina del nivel más bajo del s.o. que pueda atender la petición que pide el usuario (fig. 6). Una vez se ha satisfecho la petición, se desencadena una secuencia de retornos hasta llegar al proceso de usuario.

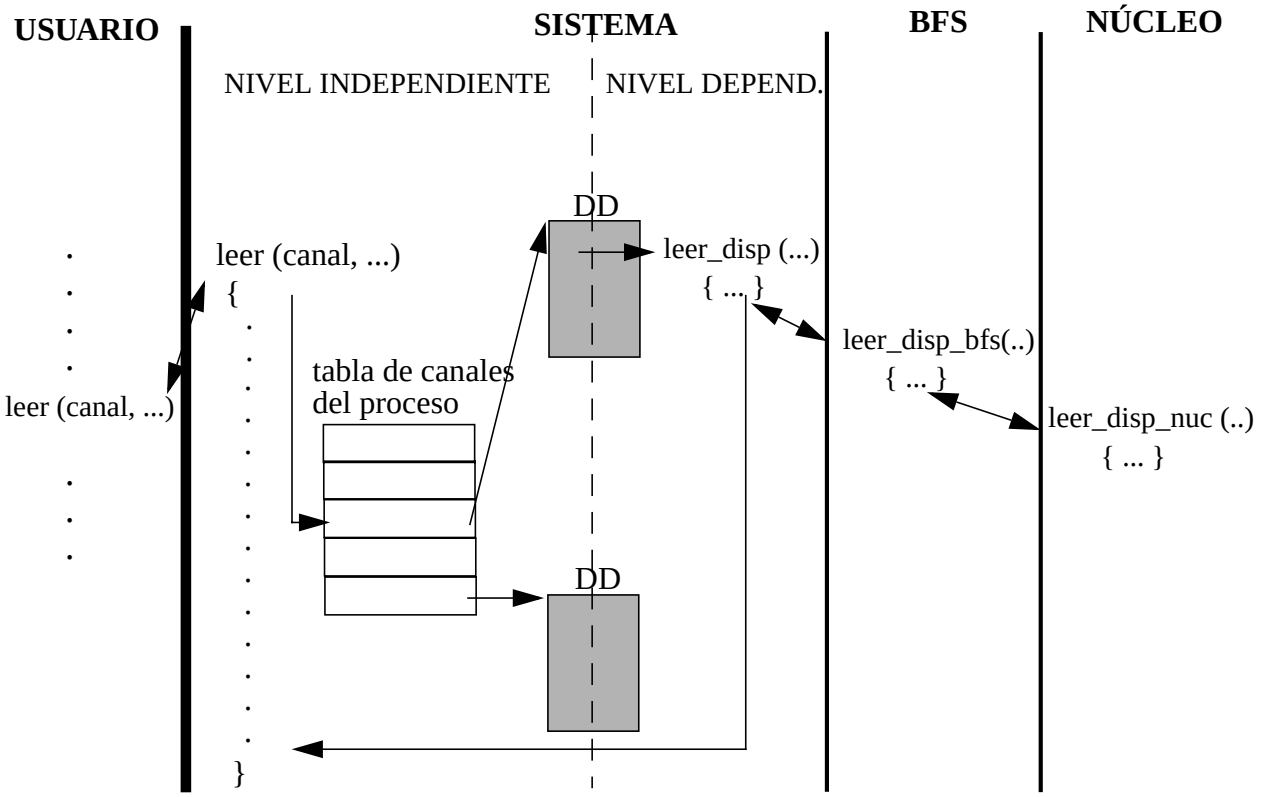

Figura 6: Secuencia de llamadas en el entorno del s.o. Onion

4.3.2 Gestores de dispositivo (device handler).

Una segunda forma de implementar las llamadas al sistema de E/S es añadir al esquema anterior un proceso de sistema encargado de recibir y servir las peticiones. Este proceso lo llamaremos *gestor de dispositivo* (device handler). Este proceso se encuentra entre los usuarios y las rutinas que acceden directamente al dispositivo (*device drivers*).
Los gestores de dispositivo los podemos tener al nivel que nos haga falta (sistema, bfs, núcleo). Cada gestor suele realizar operaciones de un determinado dispositivo. En la figura 7 se muestra un primer esquema de un gestor de dispositivo.

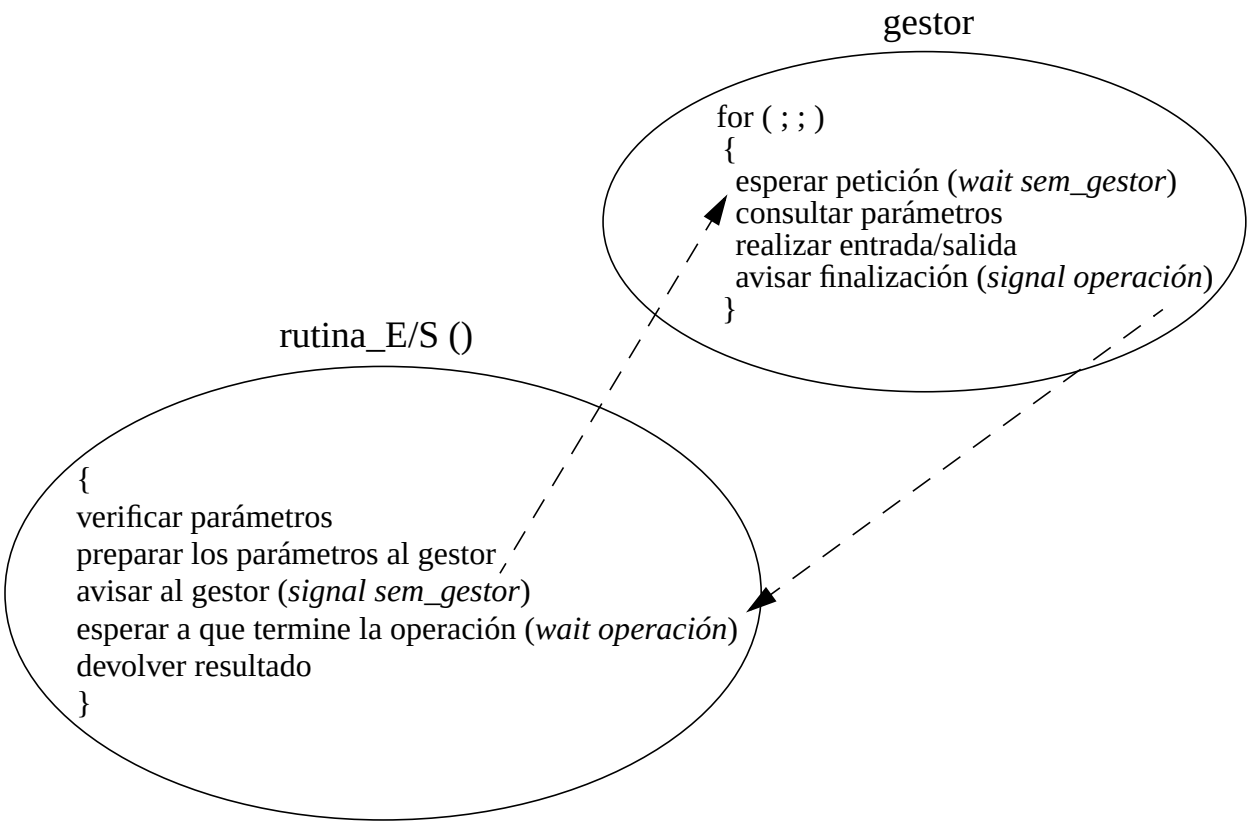

Figura 7: esquema de un gestor de dispositivo

La rutina de E/S prepara los parámetros y notifica al gestor del dispositivo que tiene una petición pendiente. Esta notificación se realiza normalmente mediante un signal a un semáforo. A continuación el proceso que ha hecho la llamada se espera a que el gestor termine la operación. Una vez finalizada la operación, el gestor lo notifica mediante un signal a otro semáforo.

4.3.2.1 Semáforos de sincronización.

- Semáforo de petición pendiente

El semáforo de petición pendiente (*sem_gestor*) sirve para indicar al gestor, que tiene operaciones de entrada/salida por realizar. La rutina de E/S realiza un signal sobre este semáforo cada vez que llega una nueva petición.
Es un semáforo n-ario, inicializado a cero. Cada gestor tendrá el suyo propio, y normalmente estará contenido en el descriptor de dispositivo (DD) que gestiona.

- Semáforo de operación finalizada

Este semáforo se utiliza para sincronizar al proceso que ha hecho la llamada al sistema con la finalización de la operación de E/S. El gestor realiza un signal sobre este semáforo una vez haya terminado la operación.
Es un semáforo binario, inicializado a cero. Si cada proceso sólo puede realizar una E/S, el semáforo puede estar contenido en su PCB (*sem_proc*). Si el proceso puede realizar varias E/S, el semáforo estará asociado a la E/S (*id_io*).

4.3.2.2 IORB (Input Output Request Block).

Generalmente la rutina de E/S, una vez ha comprobado los parámetros, rellena una estructura de datos (IORB) que no es más que una petición de entrada/salida, la cual se encola con el resto de peticiones que existan sobre ese dispositivo.
Una posible estructura IORB es la mostrada en la figura 8:

```
struct IORB
        {
         struct item elem;          /* para encolar los IORB */
         char *buff                 /* buffer para realizar la E/S */
         long *lon;                 /* nº de caracteres a leer ó escribir */
         int operación;             /* lectura ó escritura */
         int id_io;                 /* nº de semáforo asociado a la E/S */
         struct dd *eldd;           /* puntero al DD sobre el que se realiza la E/S */
        }
```

Figura 8: Estructura de un IORB

Esta cola de IORB's está ligada al descriptor de dispositivo (DD) en cuestión, y es atendida por su gestor de dispositivo. El gestor cada vez que haya finalizado con una petición, recogerá otra de la cola para atenderla ó se bloquea en el caso de que la cola se encuentre vacía. El gestor puede utilizar cualquier algoritmo para extraer peticiones de la cola (se profundizará más en este tema).
Desde el punto de vista de implementación en Onion, normalmente hay un pool de IORB's libres ó se utiliza la rutina de *assignar_memoria* para crear uno nuevo. La rutina de E/S busca un IORB libre, lo rellena y lo encola. Una opción es que el gestor devuelva el resultado en el IORB, y sea la propia rutina de E/S la que se encargue de liberarlo. Otra opción es que el gestor devuelva el resultado en otra estructura de datos, por lo que una vez obtenga los parámetros para realizar la entrada/salida, libere el IORB.
La cola de IORB's tiene que ser accedida en exclusión mutua.

4.3.2.3 Implementación síncrona.

En la figura 9 se presenta un esquema más detallado de lo que se ha visto hasta ahora, en el caso que se tenga el gestor a nivel sistema. En este esquema, los procesos se bloquean hasta que finalice la operación de E/S que han solicitado. A esta implementación la llamamos *modelo síncrono*.

4.3.2.4 Implementación asíncrona.

En muchos casos no es necesario que el proceso se espere por la finalización de la E/S para continuar ejecutándose. Por ese motivo, el siguiente paso es la implementación de las E/S de forma asíncrona, y cuando el proceso necesite asegurarse de que la operación de lectura/escritura ha terminado se puede sincronizar con el gestor mediante la llamada al sistema *esperar.*
La implementación del modelo asíncrono es similar al del síncrono con la diferencia que la rutina de leer/escribir no hace el *wait (sem_proc)*. En la llamada al sistema *esperar* es cuando se ejecutará el *wait (id_io)*.
Es importante observar que la sincronización es opcional, y a la hora de implementarla se ha de preveer que a veces no se realiza el wait. También se ha de tener en cuenta la devolución de resultados y códigos de error. La figura 10 muestra el esquema de este modelo.

4.3.2.5 Finalidad de un gestor de dispositivo.

La idea del gestor es aplicable a cualquier esquema de cliente/servidor, ya sea dentro ó fuera del s.o. Las razones que justifican su utilización en la implementación de las E/S son básicamente, la de que sean asíncronas y la de permitir una ordenación de las peticiones.

- E/S asíncrona
El esquema de “secuencia de llamadas” del apartado 4.3.1 no permite realizar operaciones asíncronas. La única forma de hacerlo es mediante un gestor de dispositivo.

- Ordenación de las peticiones
La utilización de un gestor facilita la aplicación de una política de ordenación de las peticiones que mejore el rendimiento del dispositivo. Esta ordenación también se podría realizar con una “secuencia de llamadas”, pero no sería tarea sencilla. Un ejemplo de ordenación lo encontramos con el gestor de disco que optimiza el movimiento del brazo según la política que utilicemos (FCFS, SSTF, SCAN, C-SCAN, ...). De esta forma se reduce el tiempo de seek (posicionarse en la pista) (ver Peterson cap. 7).
Otro ejemplo de Onion es el de permitir a más de un proceso retardarse a nivel BFS mientras que en núcleo se permite sólo a uno (gestor). Este proceso es el que se encarga de ir despertando a los otros que están en una cola ordenada por incrementos de tiempo.

4.3.3 Técnicas de mejora del rendimiento.

- Buffering
Consiste en disponer de un buffer que almacene temporalmente los caracteres que son enviados/recibidos por el dispositivo que provoca la E/S. De esta forma evitamos dos cosas, por una parte que se puedan perder caracteres cuando exista un pico alto de entrada/salida y el proceso encargado no puede atender toda la información y por otra, evitar en lo posible que dicho proceso se bloquee cuando no disponga de caracteres. Hay que significar que el buffering no elimina ninguno de estos dos problemas pero sí que los puede reducir mucho.

- Spooling
Consiste en realizar la entrada/salida sobre un dispositivo intermedio. Cuando sea posible, los datos pasarán del dispositivo intermedio al dispositivo final. La ventaja que puede reportar el spooling es significativa cuando el dispositivo final no es compartible. El dispositivo intermedio, normalmente el disco, puede ser compartible y además de acceso más rápido.

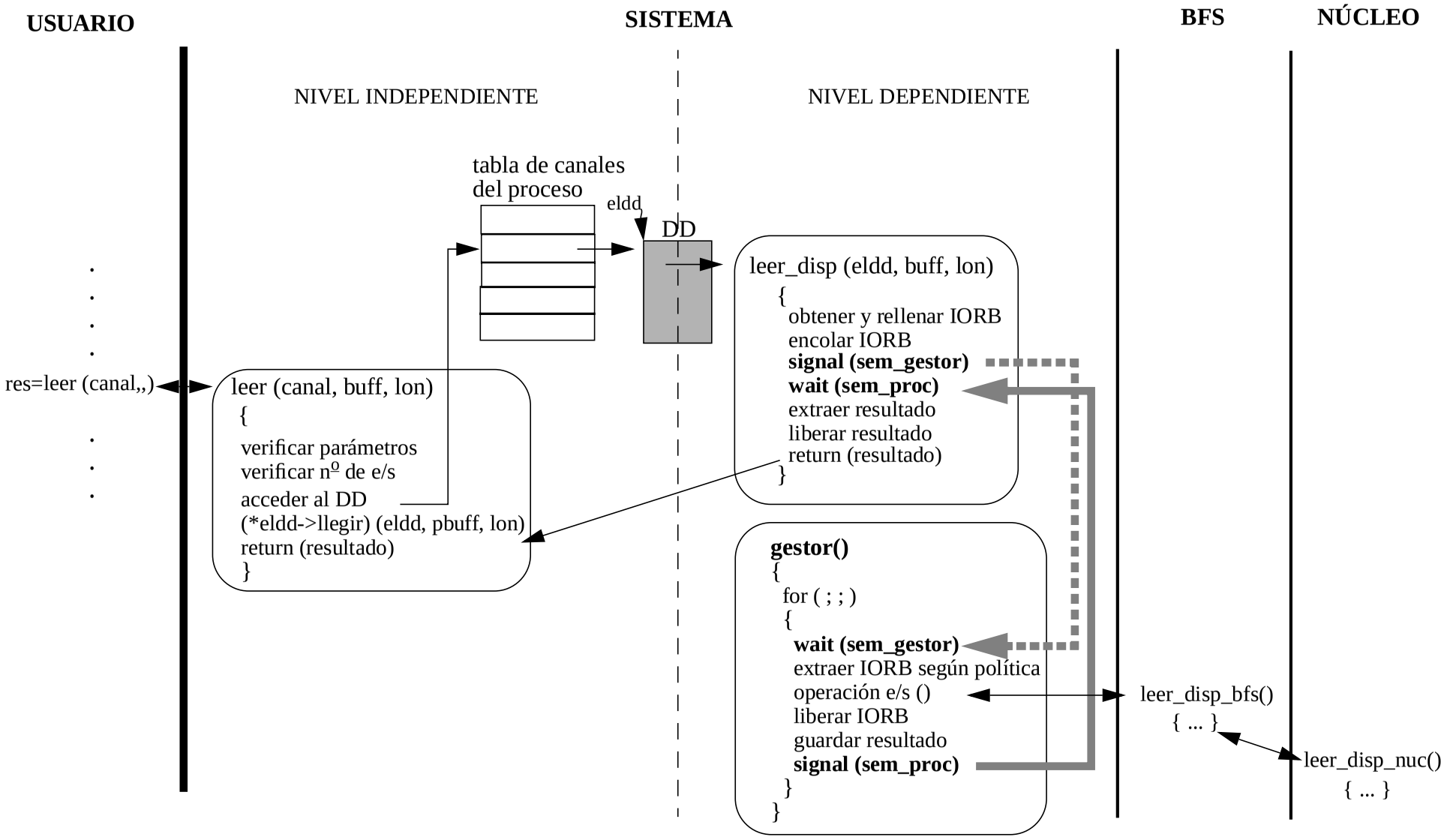

Figura 9: Esquema de modelo síncrono

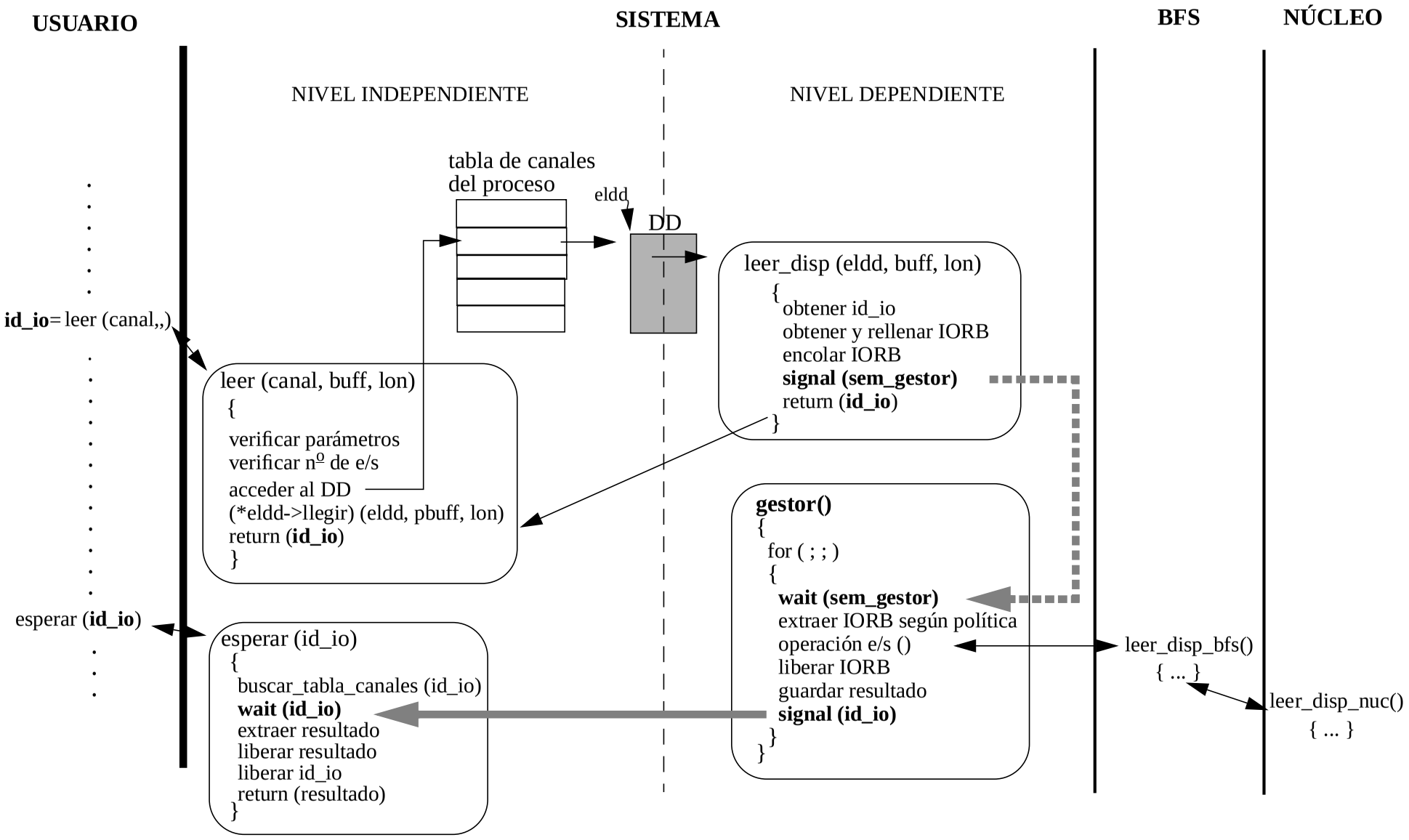

Figura 10: Esquema de modelo asíncrono

Estructures de dades

```
struct entrada_tc
{
   int mode;
   struct dd * pdd;
   struct cua q_io;
};
```

```
struct pcbsys
{
   ...
   int n_es;
   struct entrada_tc t_c[5];
};
```

```
struct iorb
{
   struct item elem;
   char * buff;
   long * lon;
   int operacio;
   int id_io;
   struct dd * eldd;
};
```

```
struct io_fin
{
   struct item elem;
   int id_io;
   int result;
};

struct io
{
   struct item elem;
   int id_io;
};
```