Gestión de Entrada/Salida y Sistema de Ficheros

Yolanda Becerra Fontal
Juan José Costa Prats

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya
BarcelonaTech
2018-2019QT

Índice

- Conceptos básicos
- Estructuras de datos básicas
- Visión de usuario
- Implementación
- Optimizaciones
- Ejemplos
  - Unix, Windows
- Comunicación entre procesos

Conceptos básicos

- Se entiende por E/S la transferencia de información hacia/desde un proceso
- Necesario para intercambiar información
  - Con usuario
  - Otros procesos
- Dispositivos de E/S son los que permiten hacer esta transferencia: teclado, ficheros, red, pantalla, etc

Conceptos básicos

- Dispositivos muy distintos entre si
- Características diferentes:
  - Velocidad de transferencia
  - Unidad de transferencia (bloque o carácter)
  - Operaciones permitidas
  - Modos de trabajo (compartible o no, síncrono o asíncrono)
  - Tipo de acceso (secuencial o aleatorio)
  - Tipos de errores

Conceptos básicos

- Acceso a un dispositivo es:
  - Complejo y muy dependiente del tipo de dispositivo concreto
  - Código de bajo nivel
  - Accesos simultáneos de varios usuarios podrían provocar interferencias

Conceptos básicos

- Objetivo del SO: Gestionar el acceso a los dispositivos
  - **Uniformidad de operaciones:** Ocultar las particularidades de cada dispositivo al usuario
  - Garantizar que no habrá interferencias: instrucciones de acceso son **privilegiadas**
  - **Optimizar** el rendimiento de los dispositivos
  - Facilitar la incorporación de nuevos dispositivos

  minimizando los cambios en el código del SO

Conceptos básicos

- Independencia de dispositivo
  - Conseguir que la mayor parte del código de usuario sea independiente del tipo de dispositivo que accede e incluso del modelo concreto de dispositivo
- Ventajas
  - Facilidad de uso
  - Portabilidad de los programas y fácil (o nula) adaptación a dispositivos diferentes
  - Soporte para la redirección de E/S
    - Sin modificar el código de un programa se puede cambiar el dispositivo al que accede

Conceptos básicos

- Necesitamos definir 3 tipos de dispositivos
  - Dispositivo Físico
  - Dispositivo Lógico
  - Dispositivo Virtual

Conceptos básicos

- Dispositivos Físicos
  - Hw: disco, teclado, ….
  - **No son visibles por el nivel de usuario**
  - Código que accede directamente al dispositivo físico
    - Bajo nivel: dependiente del dispositivo
    - Aislado para que sea fácil de substituir o de añadir
      - Resto del sistema de gestión de E/S independiente
  - Device Driver
    - Proporcionado por el fabricante del dispositivo
    - Implementa el **interfaz** definido por el SO

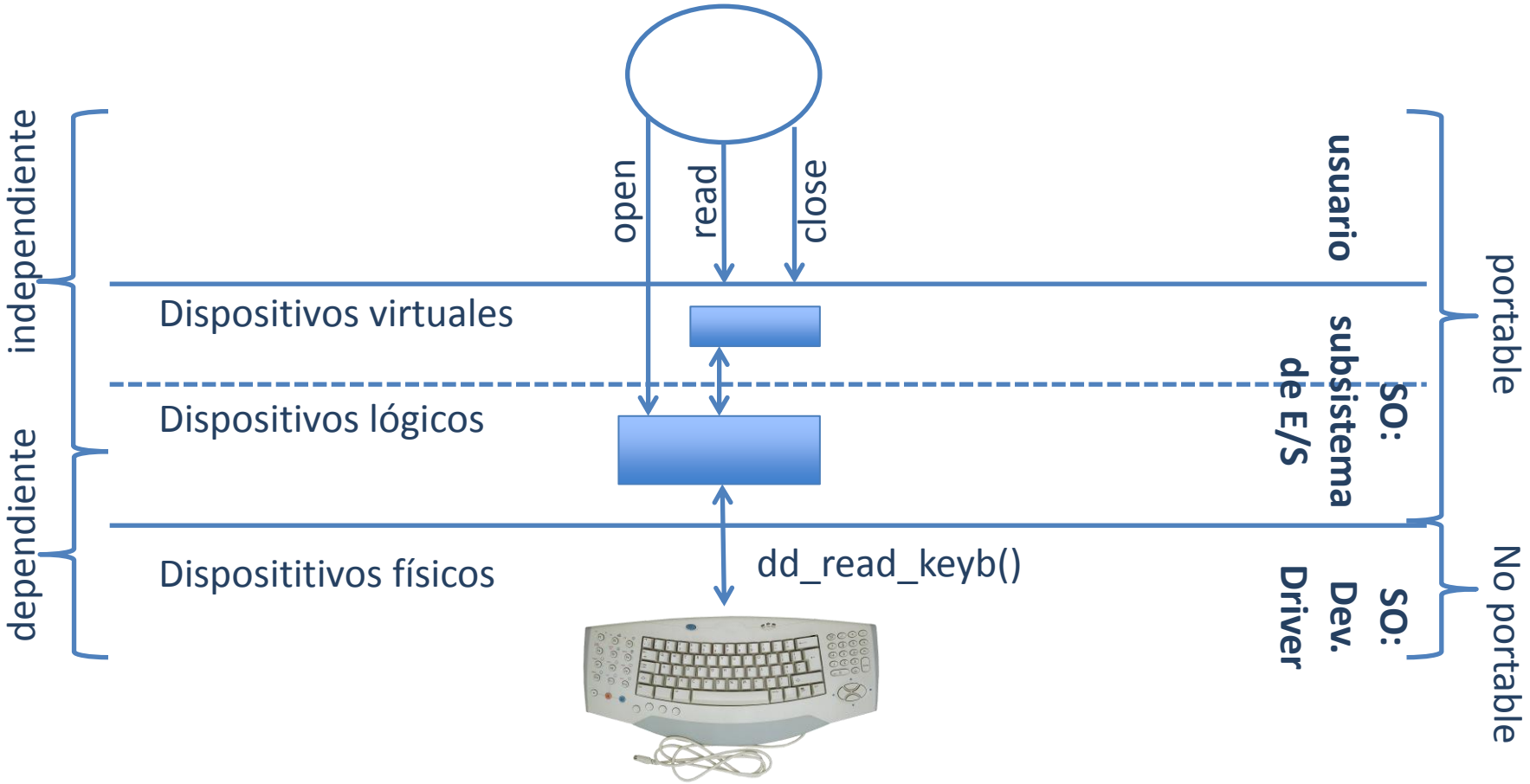
Conceptos básicos

- Dispositivos Lógicos
  - Abstracción implementada por el sistema operativo para representar un dispositivo de entrada salida
  - Pueden tener diferentes asociaciones
    - 1 dispositivo hw (teclado)
    - 2 dispostivos hw (consola: teclado y pantalla)
    - Ningún dispositivo hw (Nul)
  - Puede añadir funcionalidades sobre un dispositivo hw
    - Ficheros
  - Mayor parte del código de gestión del sistema operativo trabaja sobre este tipo de dispositivo: facilita la portabilidad del código de sistema
  - **Visibles desde el nivel de usuario**: el usuario se refiere a un dispositivo lógico para inicializar el uso de un nuevo dispositivo

Conceptos básicos

- Dispositivos virtuales
  - Interfaz que usa el código de un usuario para acceder a un dispositivo
    - Todos los accesos se hacen a través de dispositivos virtuales usando el mismo interfaz
  - El SO ofrece una llamada a sistema para asociar un dispositivo virtual con un dispositivo lógico
    - Única llamada que depende del tipo de dispositivo que se quiere usar

Conceptos básicos



Estructuras de datos básicas

- Para usar un dispositivo:
  - Usuario asocia disp. Lógico <-> disp. Virtual
  - Usuario accede a disp. Virtual con operación genérica (cualquiera de las llamadas a sistema de E/S)
  - Sistema invoca la operación específica del dispositivo Lógico (y hace todas las operaciones de gestión y optimización necesarias)

Estructuras de datos básicas (unix)

- Tabla de Canales o dispositivos virtuales
- Tabla de Ficheros Abiertos
- Tabla de I-nodes
- Directorio

Visión de usuario

- int open (char *nombre, int modo, [int permisos])
- int close (int canal)
- int read(int canal, char *buff, int nbytes)
- int write(int canal, char *buff, int nbytes)
- dup, dup2, lseek
- ioctl, fcntl

Implementación

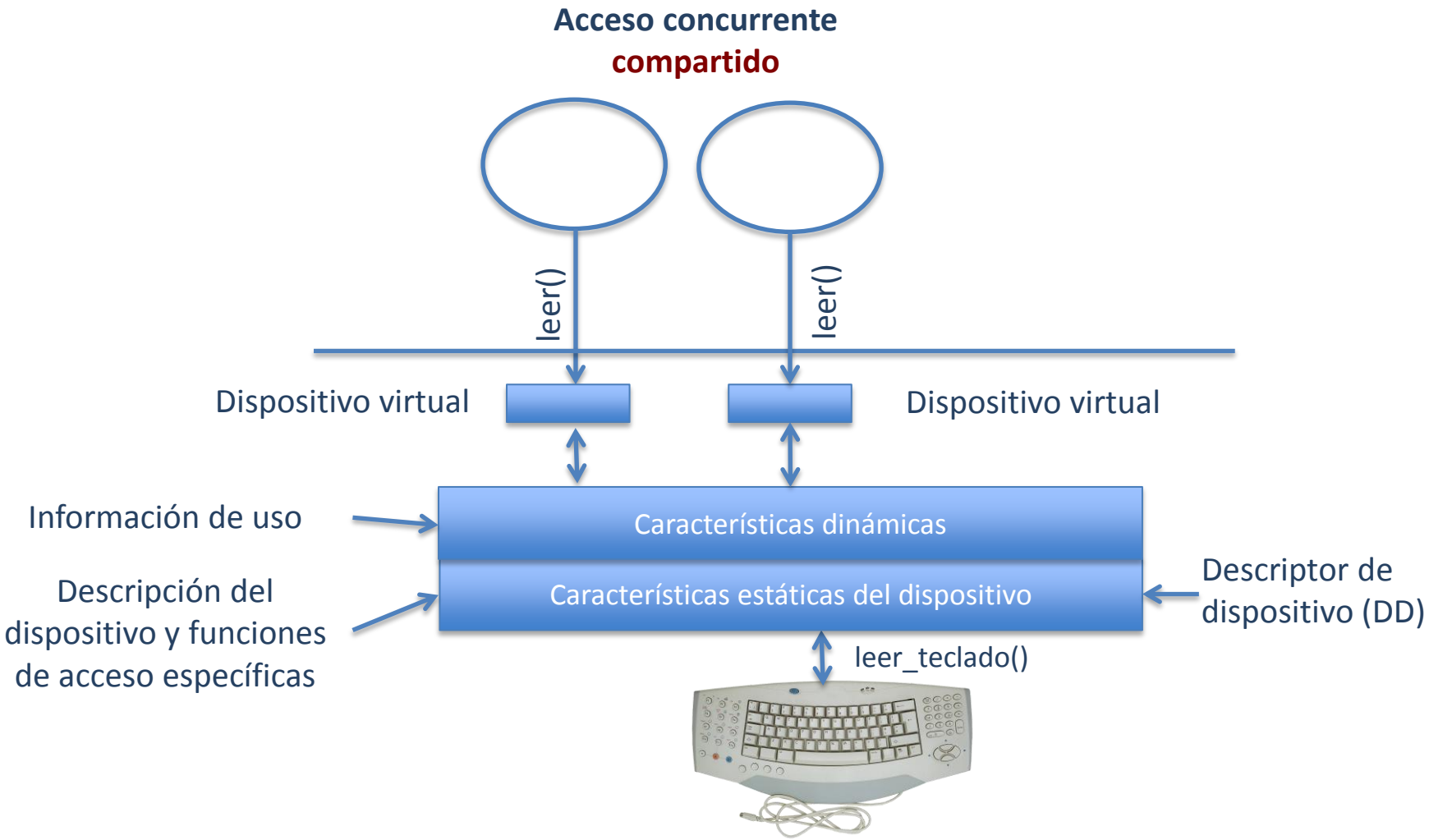
- Visión global
- Estructuras de datos
  - Soporte a la concurrencia
- Mecanismos de acceso al dispositivo
  - E/S Síncrona
  - E/S Asíncrona
  - Implementación
  - Optimizaciones
- Ejemplos
  - Unix
  - Windows

Visión global

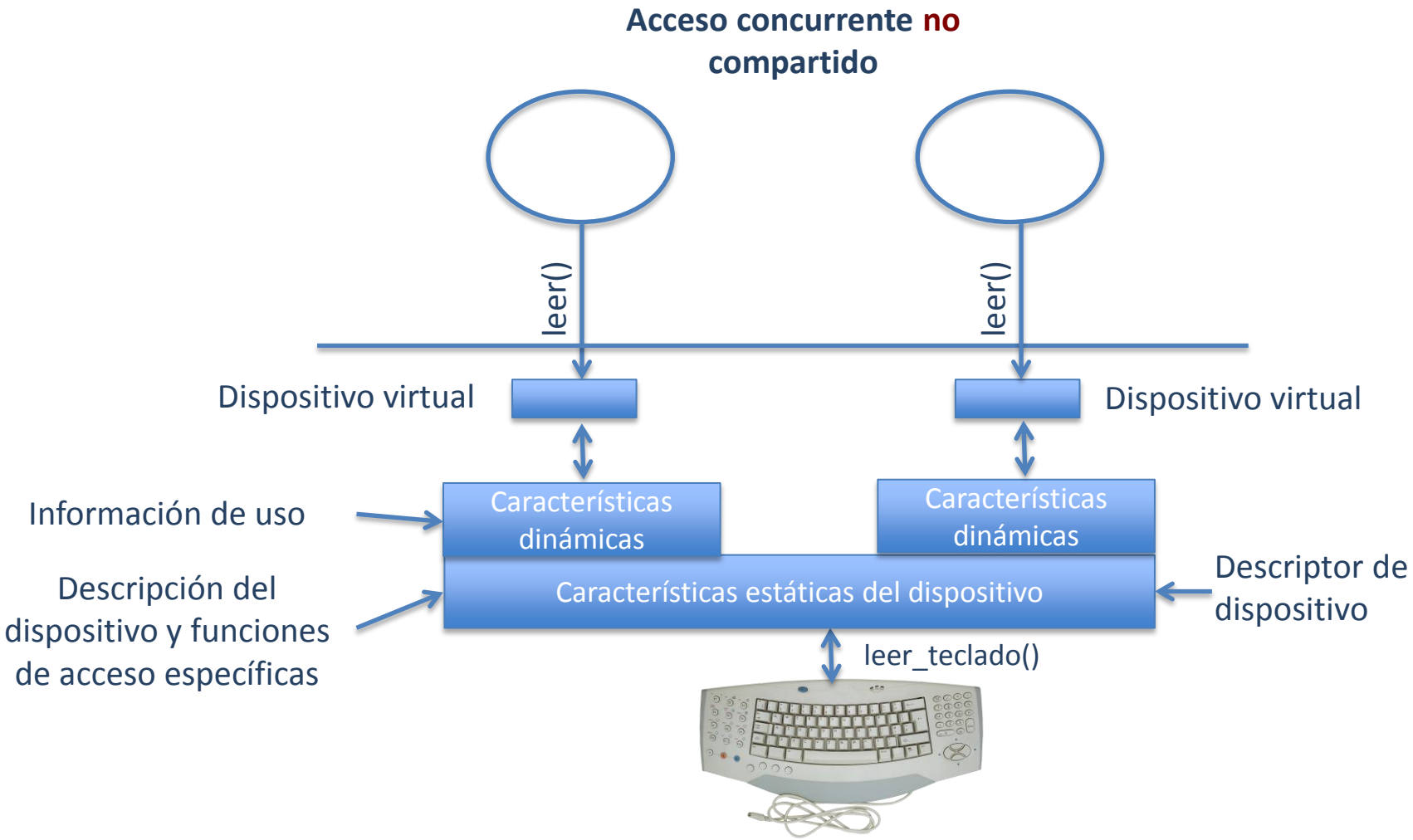
Estructuras de datos

- Características dinámicas
  - Modo de acceso, posición
- Características estáticas: descriptor de dispositivo (DD)

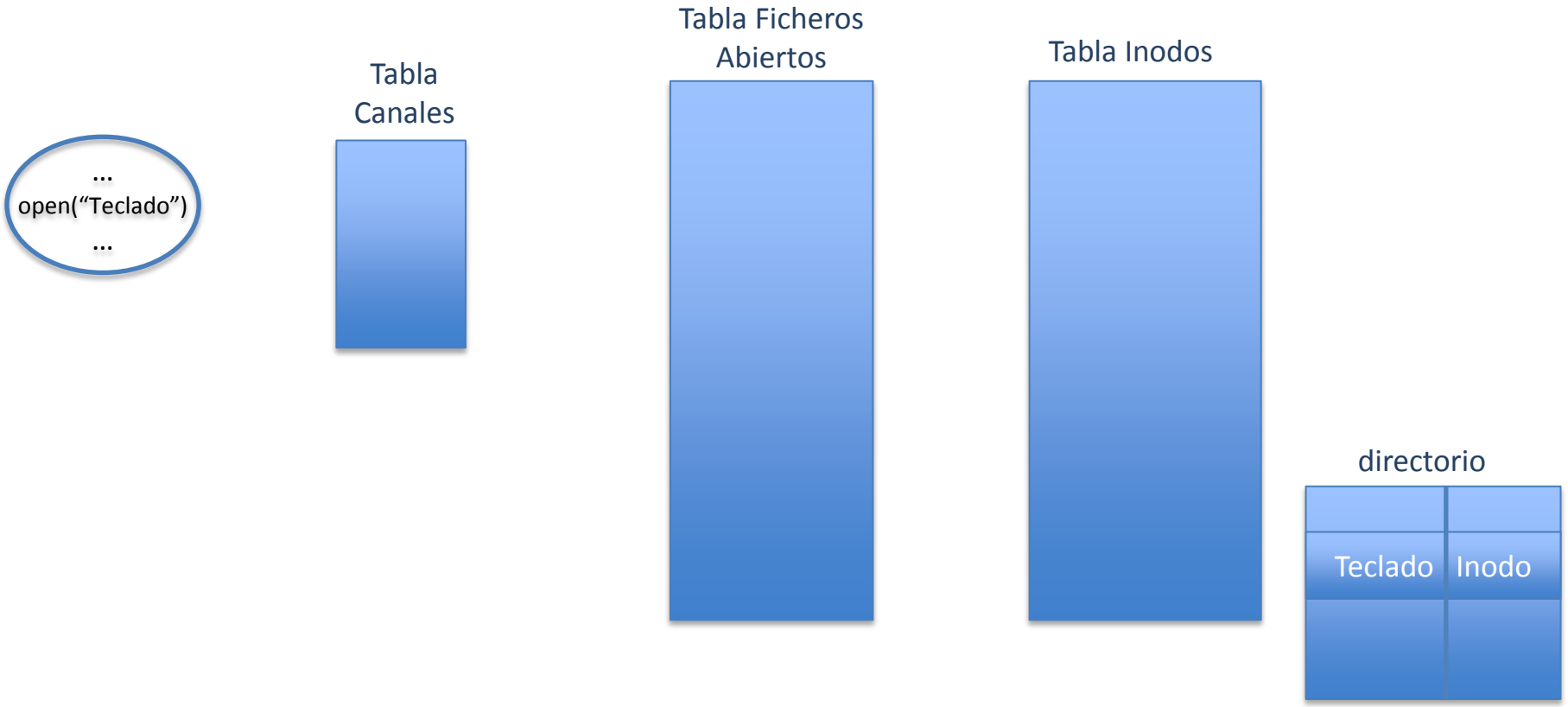
Soporte a la concurrencia



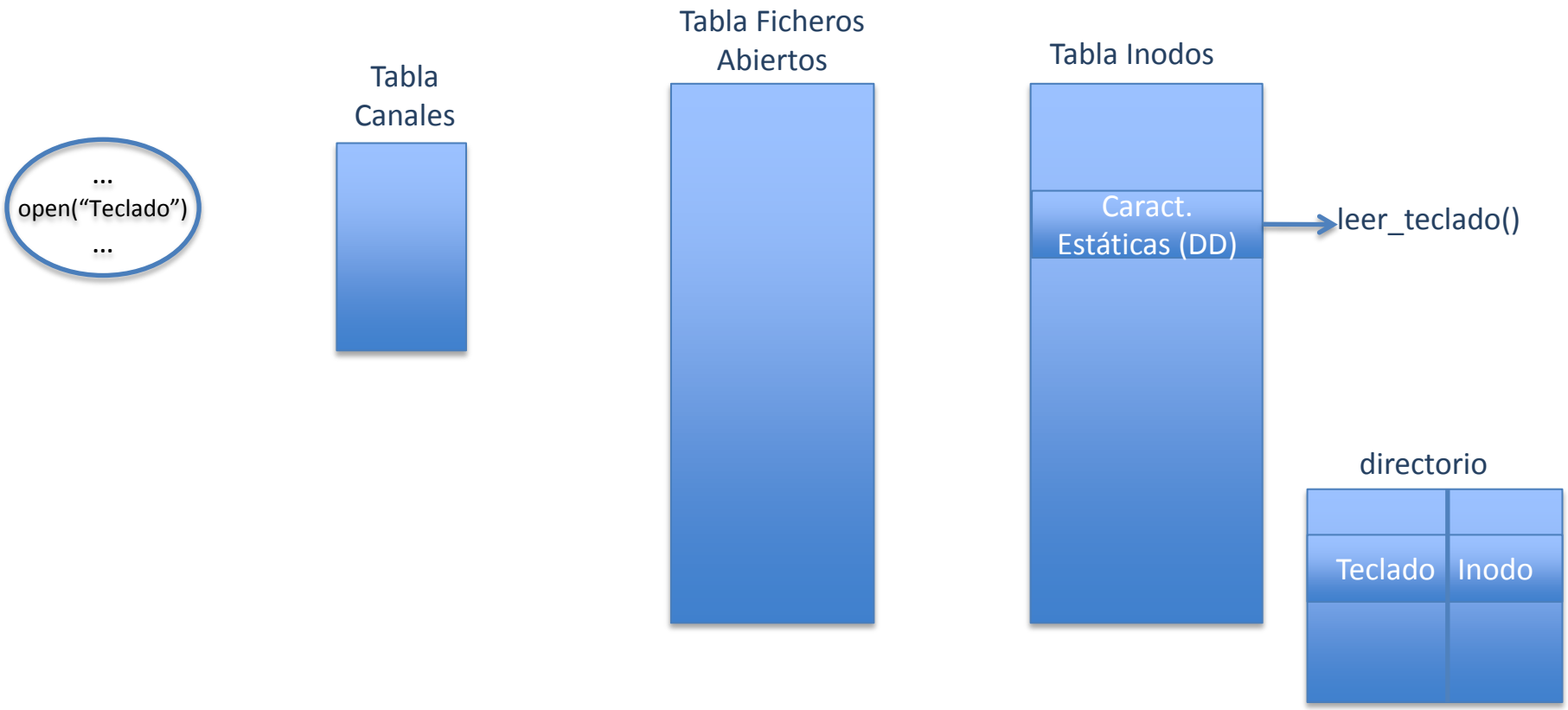
Soporte a la concurrencia



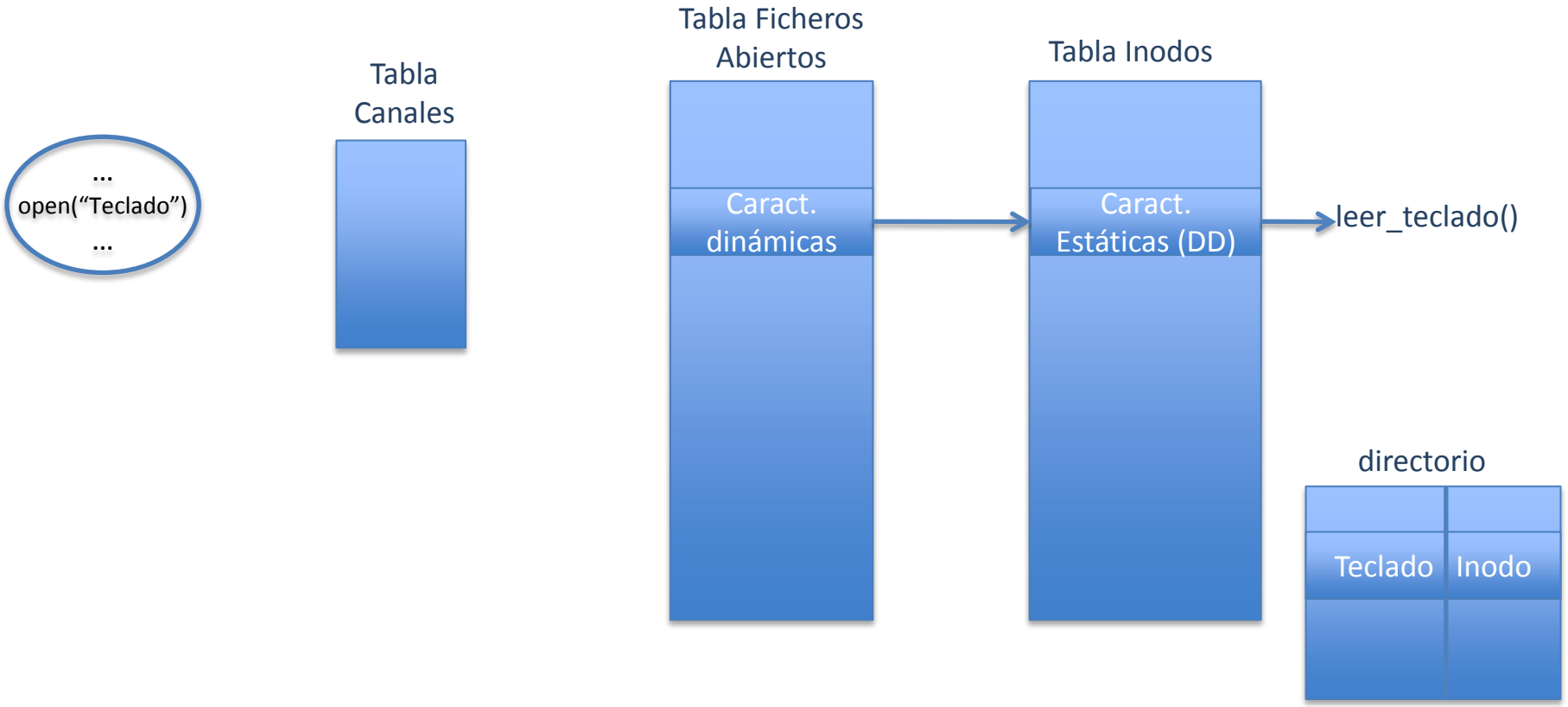
Ejemplo: Unix



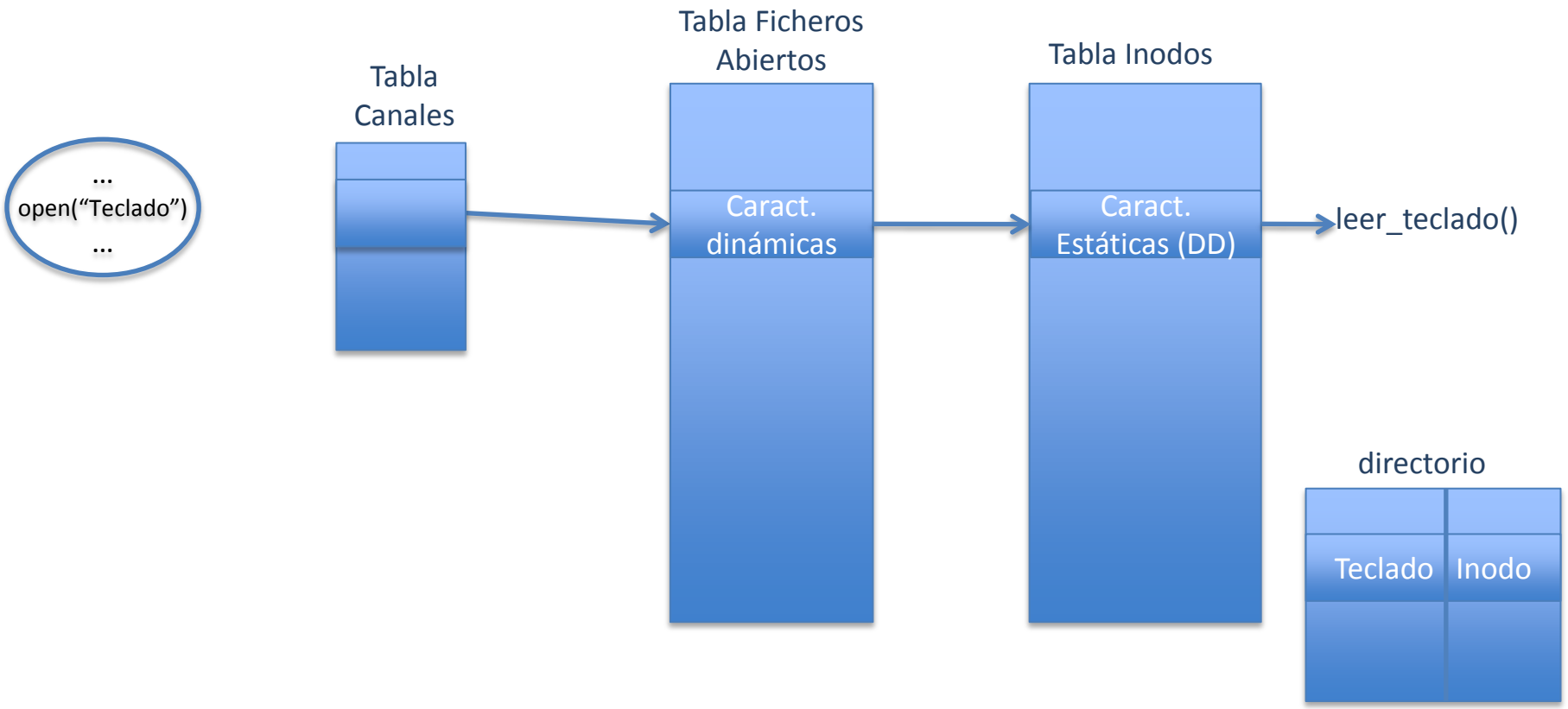
Ejemplo: Unix



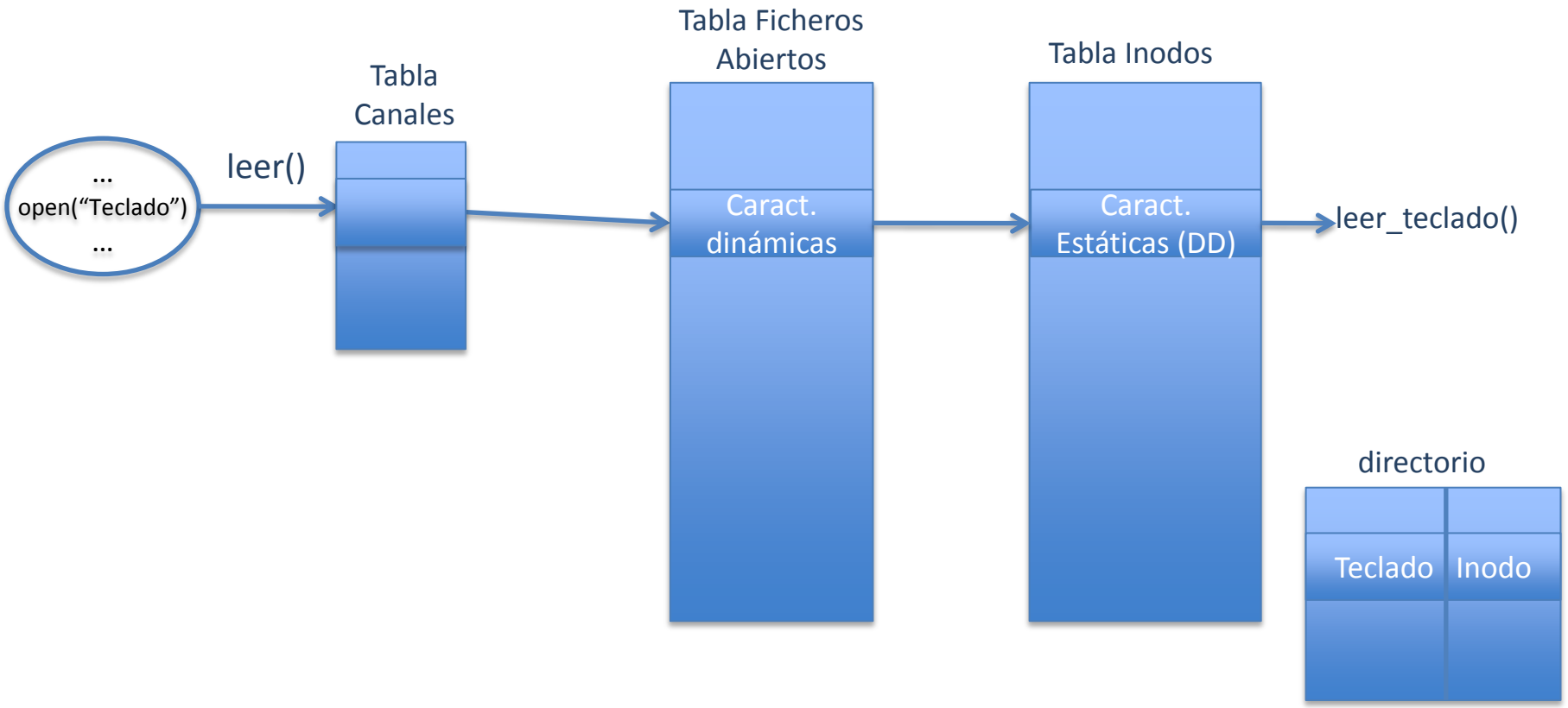
Ejemplo: Unix



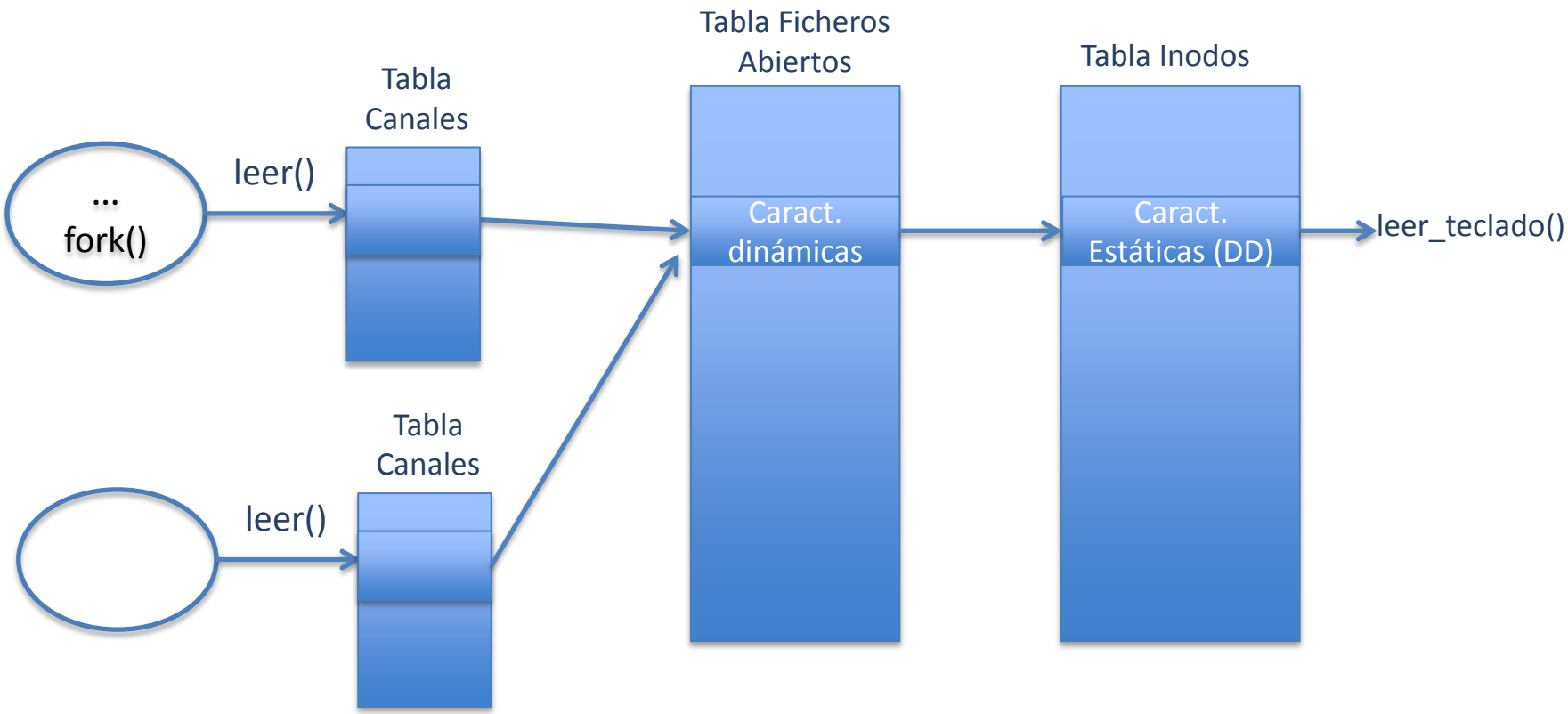
Ejemplo: Unix



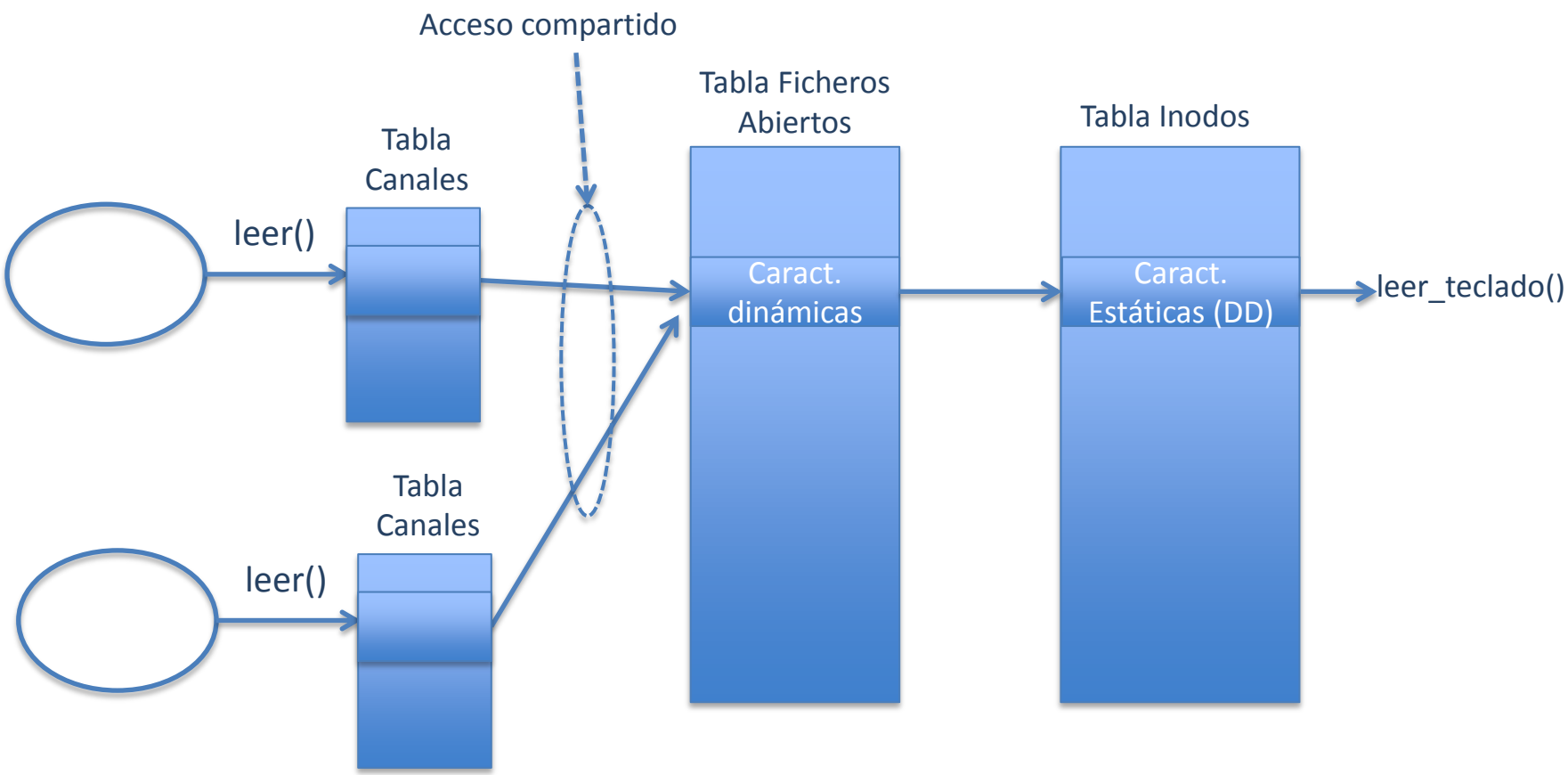
Ejemplo: Unix



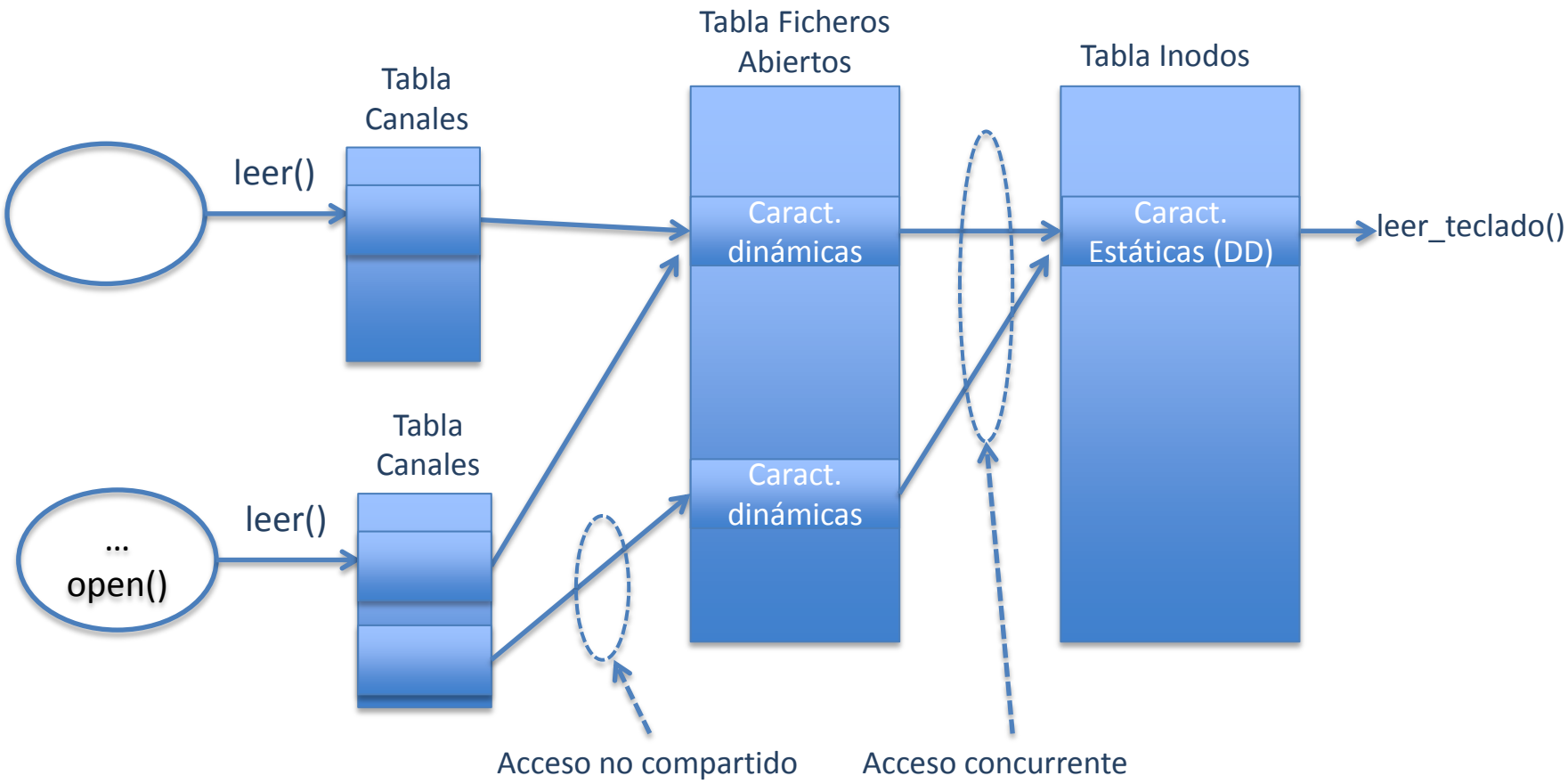
Ejemplo: Unix



Ejemplo: Unix



Ejemplo: Unix



Implementación

- Visión global
- Estructuras de datos
  - Soporte a la concurrencia
- Mecanismos de acceso al dispositivo
  - E/S Síncrona
  - E/S Asíncrona
  - Implementación
  - Optimizaciones
- Ejemplos
  - Unix
  - Windows

Acceso a dispositivo

- Descriptor de dispositivo
  - Características estáticas
  - Punteros a las funciones dependientes específicas del dispositivo
    - Define el interfaz de acceso a cualquier dispositivo
  - Añadir un dispositivo es añadir un descriptor y sus funciones relacionadas → Device Driver
- Device Driver
  - Implementa las funciones específicas del dispositivo (nivel físico)
    - No necesariamente todo el interfaz
  - Software que se comunica directamente con el hardware (*device controller*) a traves de los registros del dispositivo

Acceso a dispositivo

- Para usar un dispositivo Lógico:
  - El dispositivo Lógico tiene asociado un driver
  - Cuando el Sistema invoca una operación específica del dispositivo Lógico → se invoca una función específica del driver
  - La función del driver se comunica con el disp. Físico

- La comunicación entre el driver y el disp. Físico puede ser de dos maneras:
  - por encuesta (polling)
  - por interrupciones

Mecanismos de acceso a dispositivo

- Encuesta
  - La CPU consulta constantemente al dispositivo si la operación ya se ha realizado

```
preparar E/S
while ( consultar_dispositivo != FINALIZADO );
finalizar E/S
```

  - Sencillo
  - **Muy poco eficiente** desde el punto de vista del sistema
    - Pero muy rápido
  - Usar sólo cuando no se pueda hacer de otra manera

Mecanismos de acceso a dispositivo

- Interrupciones
  - El proceso programa la E/S y al finalizar recibe una interrupción
  - El proceso puede **bloquearse** y ceder la CPU hasta que reciba la interrupción
    - mejor uso de la CPU en el sistema
  - Minimizar **siempre** el trabajo que se hace en la rutina que atiende a la interrupción

Tipos de E/S

- Síncrona
  - El proceso de usuario no continua su ejecución hasta que finaliza la operación de E/S
- Asíncrona
  - El proceso de usuario se ejecuta concurrentemente mientras se realiza la E/S
    - El Sistema Operativo notifica al proceso cuando ésta finaliza y/o el proceso dispone de un interfaz para consultar el estado de las operaciones pendientes
      - esperar, cancelar, estado_es
    - Programación más compleja
- En un sistema que sólo proporciona E/S síncrona un proceso de usuario puede conseguir E/S asincrona si usa diversos flujos:
  - Un flujo (o varios) pueden realizar la(s) E/S (posiblemente bloqueándose)
  - Otro(s) flujo(s) pueden realizar los cálculos (sincronizándose cuando sea necesario con los flujos que realizan la E/S)

Implementación

- Gestores
  - **Proceso de sistema** encargado de atender y resolver peticiones de E/S
  - Simplifica el acceso a las estructuras de datos
  - Reduce el uso de exclusiones mutuas
  - Permite planificar las peticiones
  - Facilita la implementación de E/S asíncrona
  - Puede haber 1 o más gestores por dispositivo

Implementación

- Gestor
  - Algoritmo general

```
for ( ; ; ) {
        esperar petición
        recoger parámetros
        realizar E/S
        entregar resultados
        notificar finalización E/S
}
```

  - Sincronización proceso gestor y usuario
  - Paso de parametros
  - Retorno de resultados

Implementación

- Sincronización proceso de usuario <-> gestor
  - Mediante semáforos (operaciones wait / signal)

    wait: esperar_petición
    signal: enviar_petición
  - Notificar una nueva petición de E/S al gestor
    - El gestor espera a recibir notificaciones
      - wait sobre el semáforo del gestor
    - La rutina de E/S avisa al gestor
      - signal sobre el semáforo del gestor
  - Notificar finalización de E/S a usuario
    - La rutina de E/S espera la finalización
      - wait sobre un semáforo
      - Cada operación de E/S tiene un semáforo **propio**
    - El gestor avisa de la finalización de la E/S
      - signal sobre el semáforo

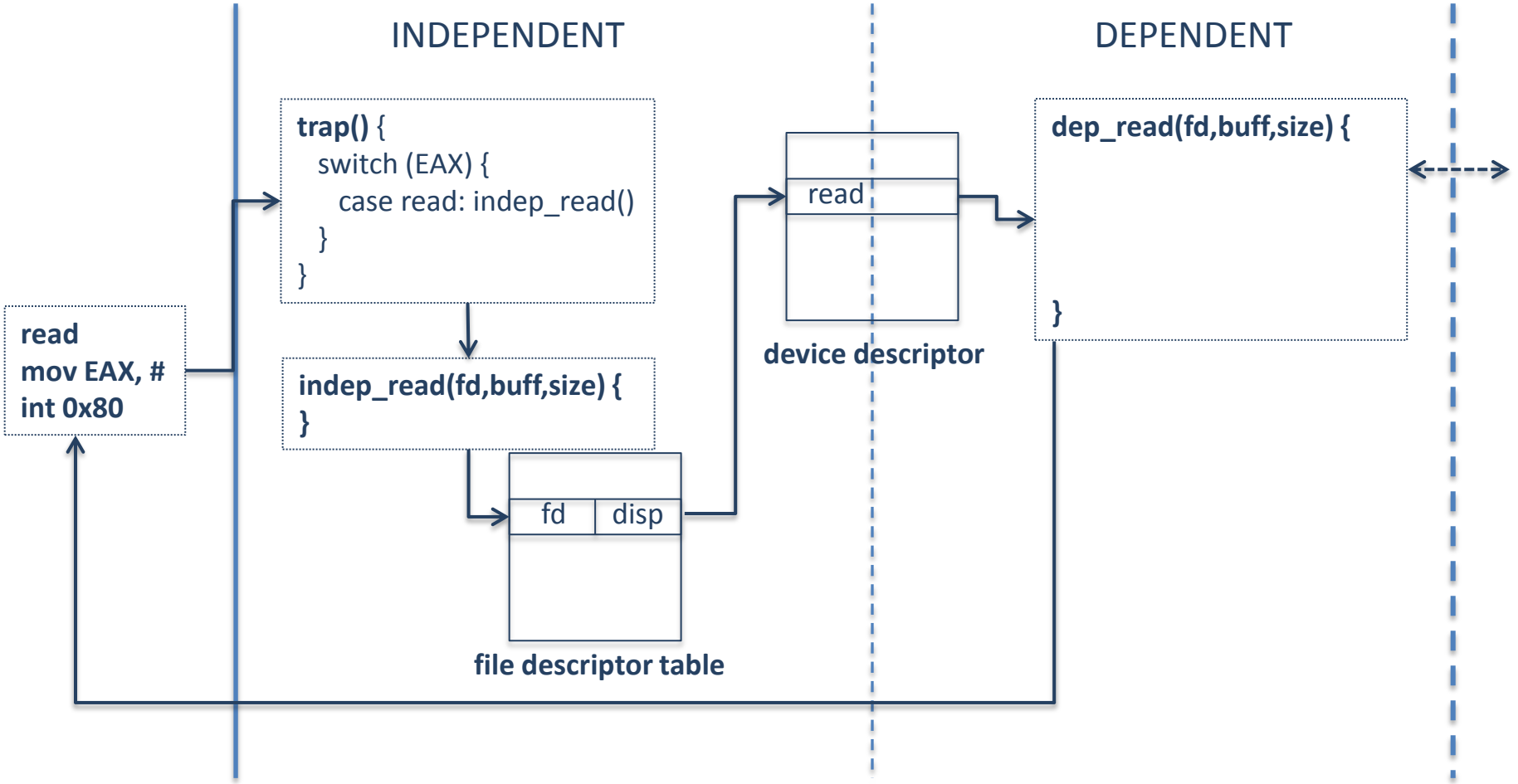
Implementación

- Paso de parámetros usuario →Gestor
  - Mediante la estructura IORB (Input/Output Request Block)
    - El contenido de los IORBs varía según el dispositivo
    - Cada gestor/dispositivo tiene una cola de IORBs con las peticiones pendientes
    - Las rutinas de E/S rellenan y encolan los IORBs
- Retorno de resultado Gestor -> usuario
  - Mediante la estructura io_fin
    - Contiene el identificador de la operación E/S y su resultado
    - Cola de resultados por dispositivo
    - El gestor encola el io_fin y la rutina de E/S lo recoge

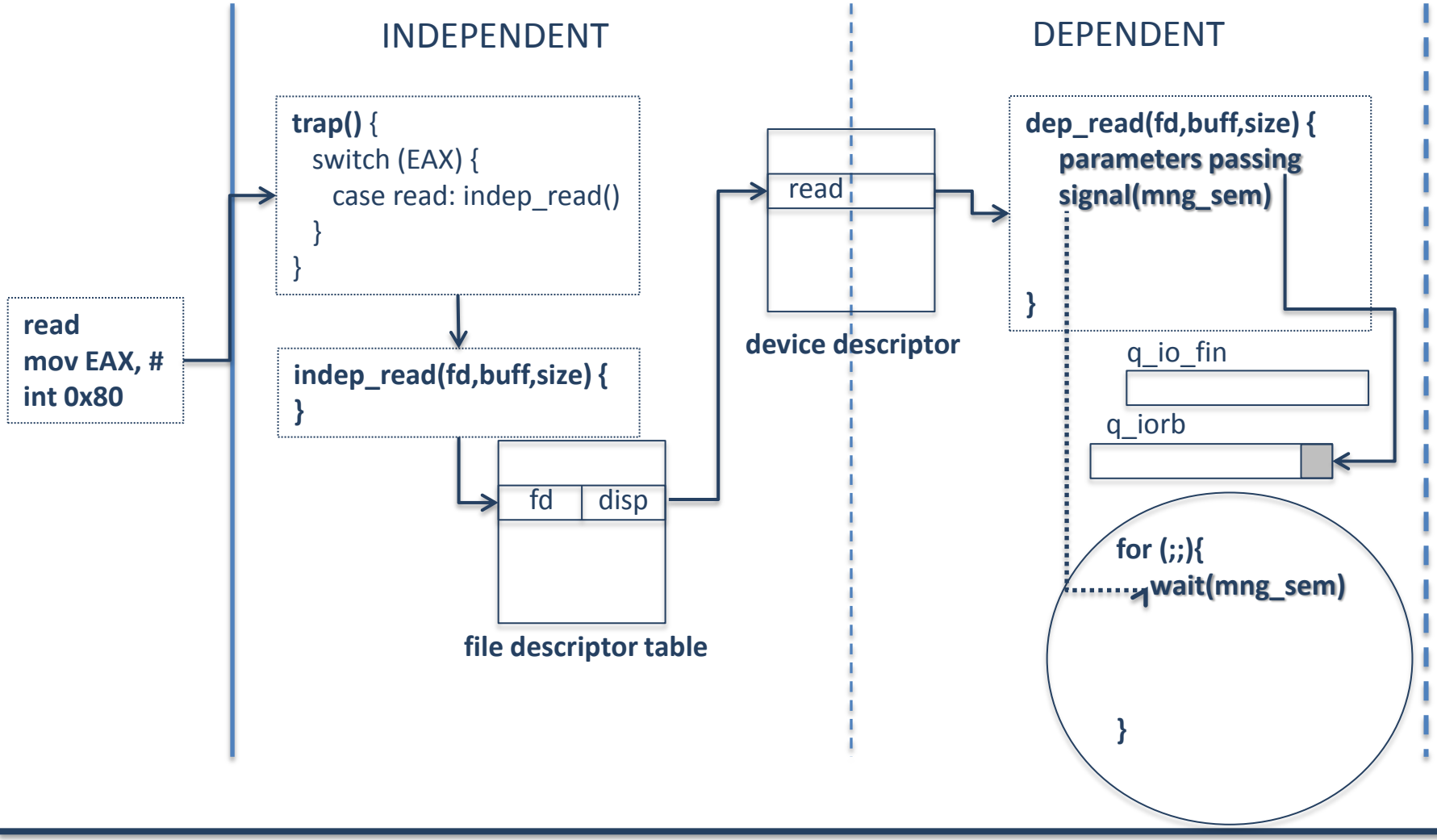
Implementación

- IORB
  - Buffer:
    - buffer de usuario donde están o donde se dejan los datos
  - Tipo de operación:
    - Lectura o escritura?
  - id_io:
    - Identificador de la operación E/S que representa el IORB

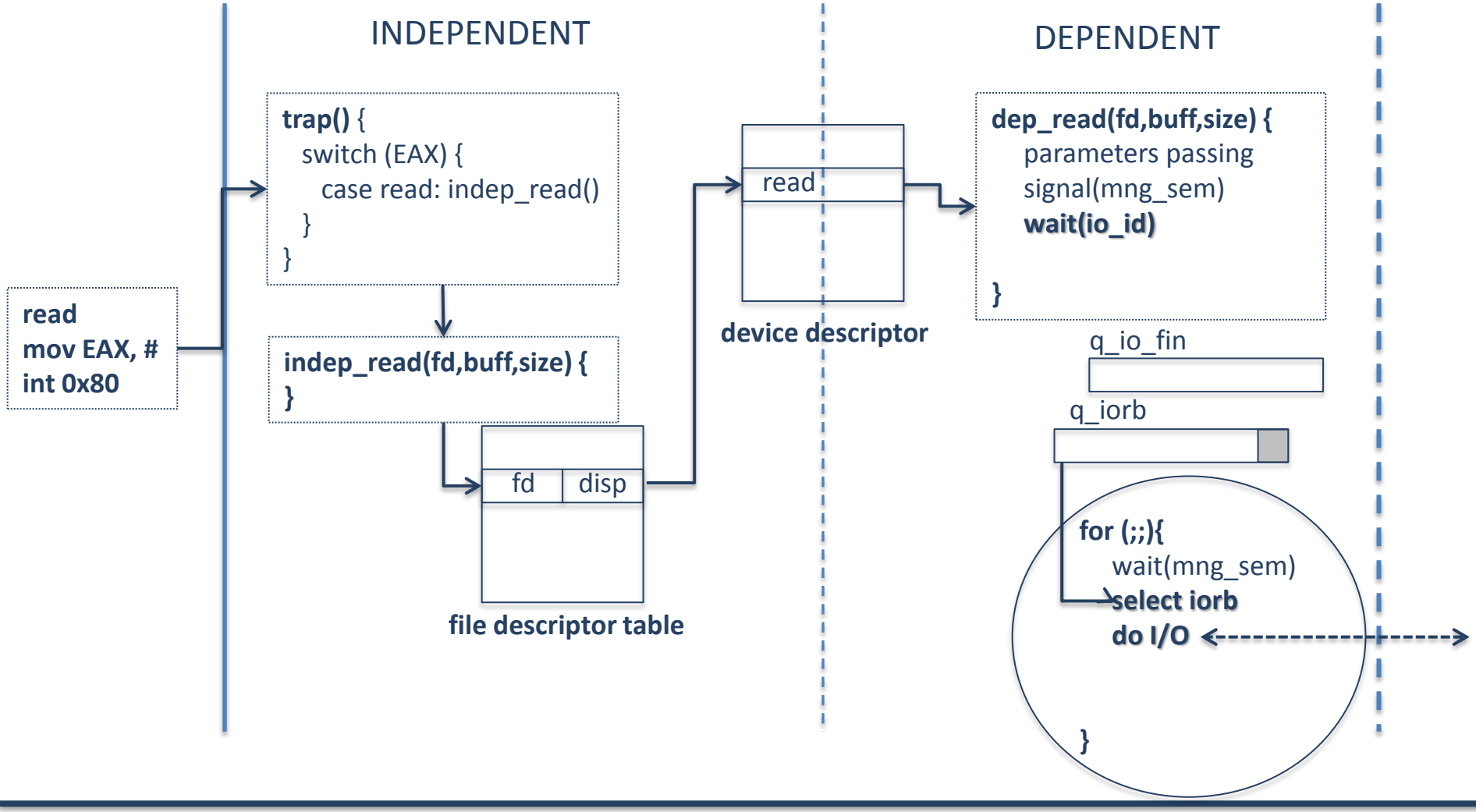
E/S Síncrona


INDEPENDENT
DEPENDENT
trap() {
switch (EAX) {
case read: indep_read()
}
}
read
mov EAX, #
int 0x80
indep_read(fd,buff,size) {
}
fd
disp
file descriptor table
read
device descriptor
dep_read(fd,buff,size) {
}

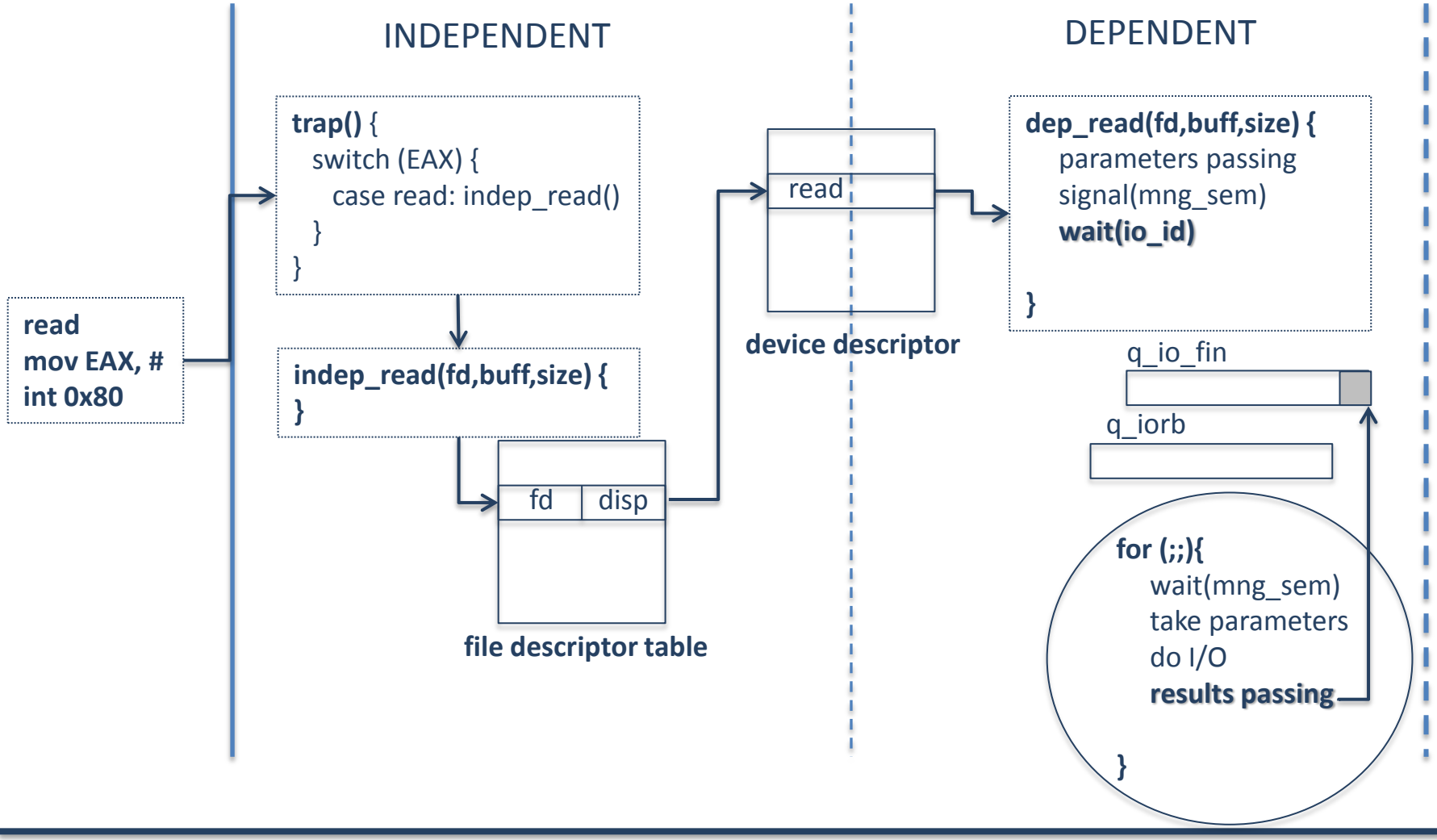
E/S Síncrona con gestor



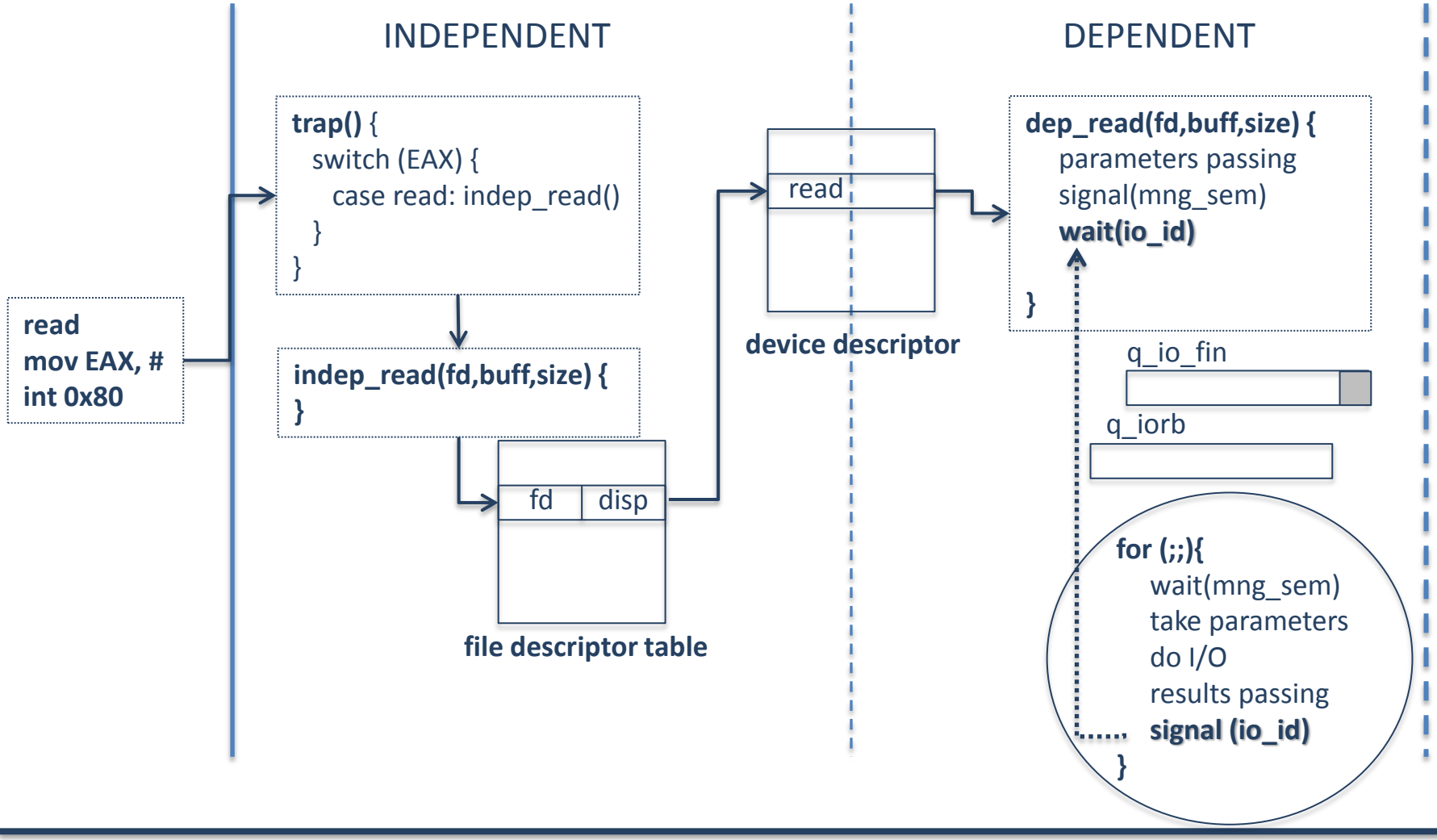
E/S Síncrona con gestor



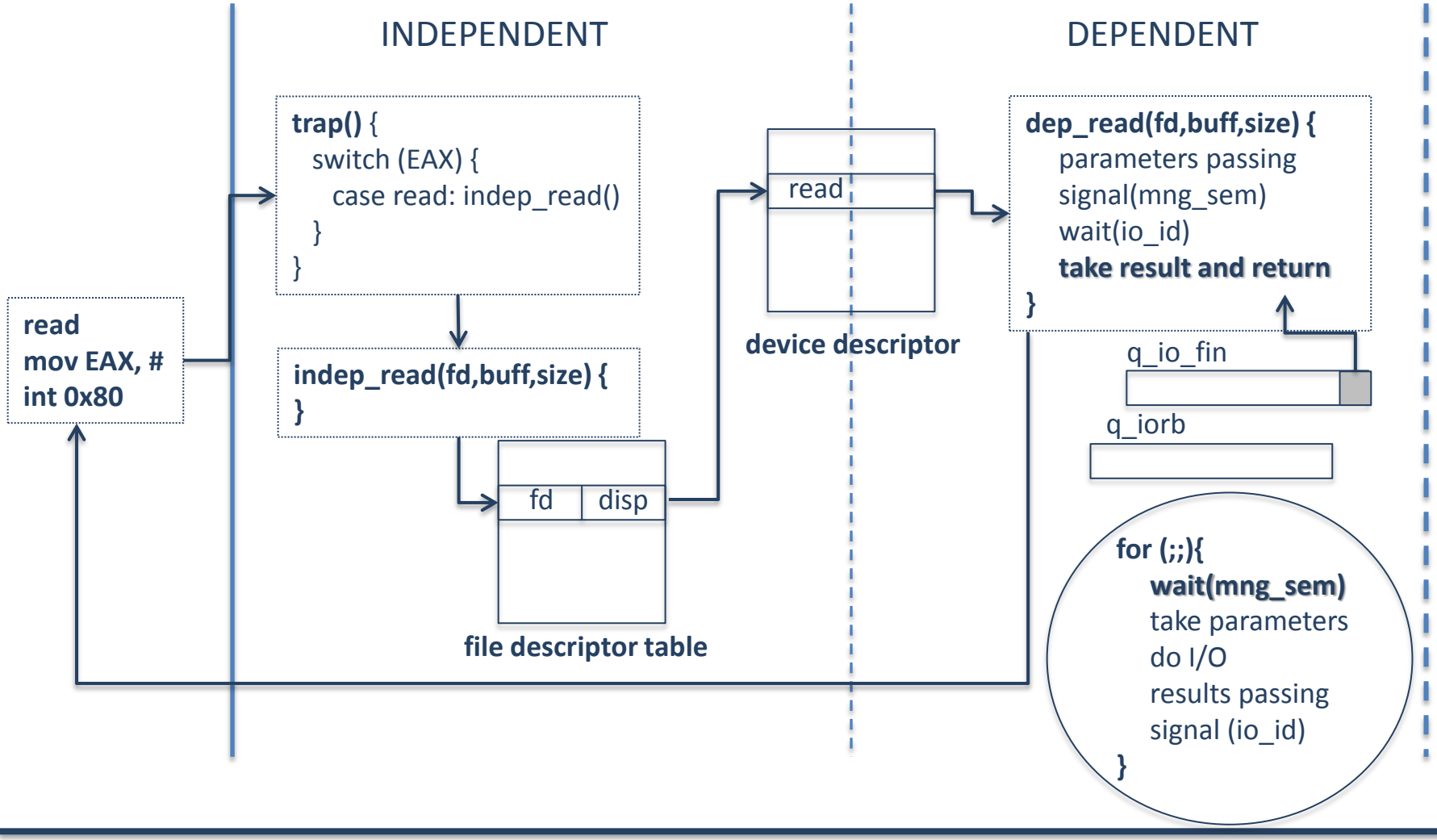
E/S Síncrona con gestor



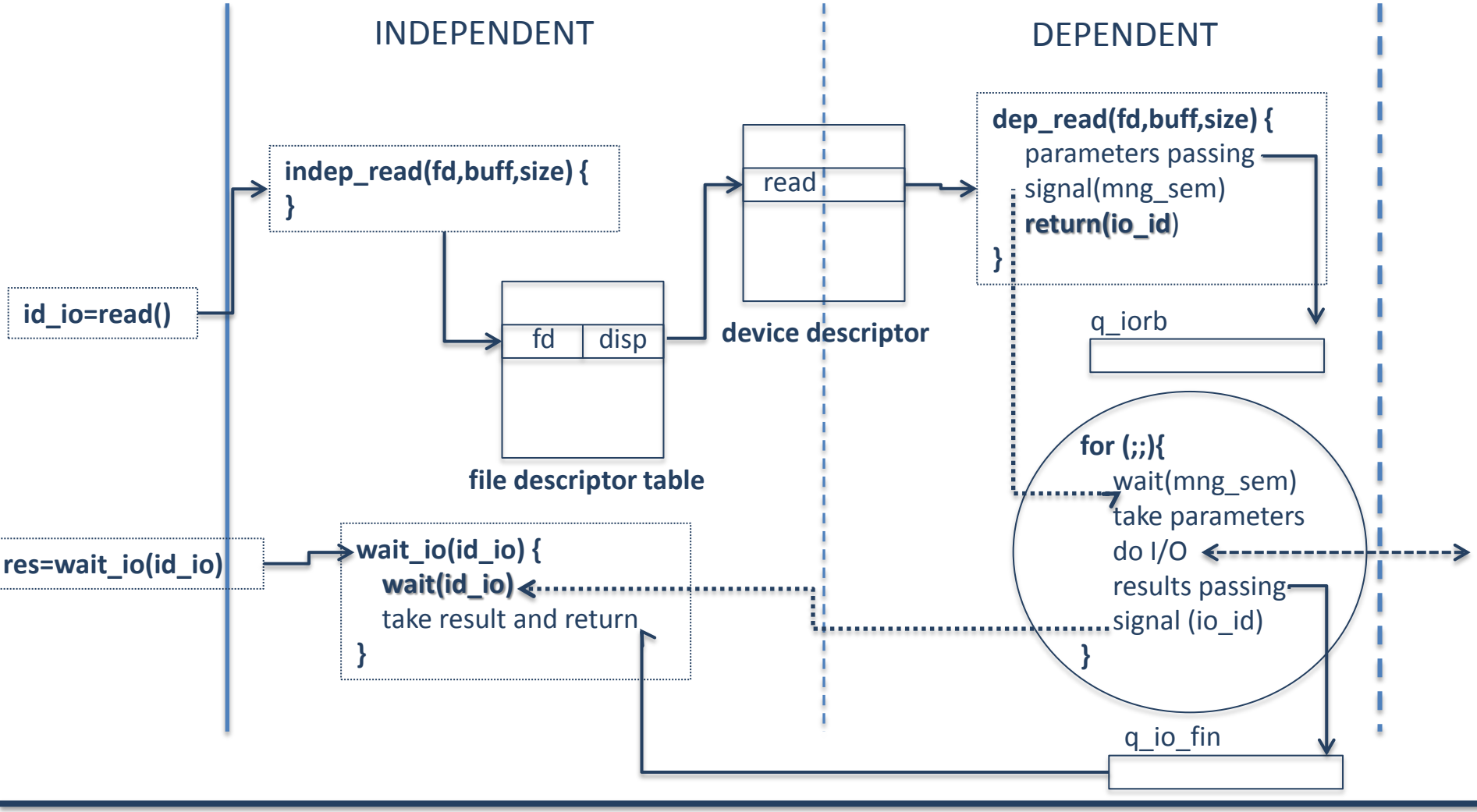
E/S Síncrona con gestor



E/S Síncrona con gestor



E/S Asíncrona con gestor



Optimizaciones

- Buffering
  - El dispositivo dispone de un buffer donde guarda los datos enviados/recibidos
    - El buffer se va llenando/vaciando mientras los procesos trabajan
    - Permite evitar bloqueos
      - evitando picos de E/S
    - Permite evitar la perdida de información
  - Doble buffering
    - Permite que se produzca a la vez movimiento de datos entre usuario – sistema y sistema - dispositivo
  - Buffering circular

ZeOS Read Keyboard

- How does it work?

Optimizaciones

- Spooling
  - La E/S se realiza sobre un dispositivo intermedio
    - El sistema posteriormente la realizará sobre el dispositivo final
    - Permite compartir dispositivos *no compartibles*
    - El dispositivo intermedio suele ser más rápido
  - Ejemplo
    - Impresora: dispositivo no compartible
      - Mientras se esta imprimiendo un documento no se puede imprimir otro
    - Disco: dispositivo compartible
      - Se pueden ir alternando accesos a diferentes ficheros para diferentes procesos
    - Se pueden guardar peticiones de impresión en ficheros temporales. Se usa una cola para gestionar las peticiones. Se imprimen de uno en uno.

Optimizaciones

- Algoritmos eficientes de acceso
  - Reordenar peticiones para mejorar la eficiencia en el acceso
  - Ejemplo: politicas de planificación de acceso a disco
    - Según quién hace la petición
      - FIFO, LIFO, random, prioridades
    - Según el contenido de la petición
      - SSTF (shortest seek time first), SCAN, C-SCAN, N-step-SCAN, FSCAN
- Organización y uso del hardware
  - Ejemplo: RAID
    - Distribución de un fichero en diversos discos: acceso en paralelo
    - Replicación: aumento de la tolerancia a fallos

Implementación

- Visión global
- Estructuras de datos
  - Soporte a la concurrencia
- Mecanismos de acceso al dispositivo
  - E/S Síncrona
  - E/S Asíncrona
  - Implementación
  - Optimizaciones
- Ejemplos
  - Unix
  - Windows

Ejemplos: UNIX/Linux

- Dispositivos lógicos accesibles a través del Sistema de ficheros
  - Ficheros especiales (normalmente situados en /dev)
    - /dev/hda1
    - /dev/audio0
    - /dev/nul
  - Se utilizan con las llamada a sistema normales (open,read,write,...)
- Se crean mediante *mknod*
  - Asigna dos numeros especiales al fichero: *major* y *minor*
    - *Relaciona dispositivo lógico con dispositivo físico*
  - Asigna el tipo de entrada/salida: por bloques o por carácteres

Ejemplos: Unix/Linux

- Device drivers: código de gestión de los dispositivos físicos
- Son ficheros objeto que se pueden enlazar de forma dinámica con el kernel (módulos)
  - Sólo aquellos drivers que se vayan a usar están realmente en memoria
  - insmod, modprobe
- El device driver se registra vinculando sus operaciones con su identificador
  - Hay diferentes funciones de registro según el tipo de driver
    - int devfs_register_chrdev (unsigned int major, const char *name, struct file_operations *fops);
    - int devfs_register_blkdev (unsigned int major, const char *name, struct block_device_operations *bdops);
    - int register_netdevice(struct net_device *dev);
    - int register_filesystem(struct file_system_type *);
    - …

Ejemplos: UNIX/Linux

- El *major* establece la relación entre el fichero y el *driver* de dispositivo a utilizar
  - Los números específicos dependen de cada SO específico.
  - P.ej. en linux:
    - 2 -> pseudo terminales
    - 3 -> primer disco ide
    - 6 -> impresora
- El *minor* permite al driver distinguir entre diferentes dispositivos del mismo tipo
  - /dev/hda1, /dev/hda2, /dev/hda3, ...

Ejemplos: Windows

- HANDLE CreateFile(name, access, sharemode, security, creation, attributes, NULL)
- Función utilizada por el sistema operativo
  - No es independiente del tipo de fichero
  - El usuario ha de saber qué tipo de fichero abrirá

Ejemplos: Windows

- Ejemplo:
  - Fichero normal abierto para leer:
    - CreateFile("\\prueba.txt", FILE_READ_DATA, FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
  - Es equivalente a:
    - open("prueba.txt", O_RDONLY);

Ejemplos: Windows

- Ejemplo:
  - Abrir un dispositivo por su nombre lógico:
    - CreateFile (("\\\\.\\PhysicalDrive0", 0, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
    - Retorna un identificador con el cual se puede escribir físicamente en el disco duro

Comunicación entre procesos

- Métodos de comunicación entre procesos
- Sockets en Linux
  - Nivel de usuario
  - Implementación
- Pipes en Linux (repaso)
  - Nivel de usuario
  - Implementación

Métodos de comunicación entre procesos

- Memoria compartida
  - Entre flujos de un proceso
    - Cualquier variable global
  - Entre flujos de diferentes procesos
    - Zona de memoria definida como compartida
    - shmget, shmat, shmdt,…
- Paso de mensajes
  - Dispositivos para el intercambio de información
    - Sockets
      - Procesos locales o remotos
    - Pipes
      - Procesos locales
      - Pipes sin nombre (sólo procesos relacionados por herencia), pipes con nombre
      - En Linux se implementan mediante sockets
- Signals
  - Notificación de eventos entre procesos del mismo usuario y en la misma máquina

Métodos de comunicación entre procesos

- Clasificación en función de si permiten comunicación dentro de la máquina o entre máquinas

		local	remota
Memoria Compartida	Flujos de un proceso	X	
	Flujos de diferentes procesos	X	
Paso de Mensajes	Pipes	X	
	Sockets	X	X
	Signals	X	

Sockets

- Socket: dispositivo lógico de comunicación bidireccional que se puede usar para comunicar procesos que están en la misma máquina o procesos en diferentes máquinas a través de la red
- Para crear un socket es necesario definir
  - Tipo de comunicación
  - Espacio de nombres
  - Protocolo de comunicación

Sockets

- Tipo de comunicación
  - Orientado a conexión (stream)
    - Se establece un circuito virtual a través del que se enviará la información
    - A la hora de enviar información no hace falta especificar dirección destino
  - No orientado a conexión (datagram)
    - No se establece el circuito: para cada paquete buscar un enlace libre
    - En cada envío se especifica destinatario
    - No garantiza ni la recepción de los paquetes ni el orden de recepción

Sockets

- Espacio de nombres
  - Para especificar dirección fuente y destino
  - Si sockets para comunicar procesos dentro de una máquina: espacio de nombres de ficheros
  - Si sockets para comunicar procesos a través de la red: espacio de nombres para direccionar dentro de internet.
    - Identificar host: dirección IP (32 bits)
    - Identificar socket dentro del host: número de puerto al que se asocia el socket (16 bits). Pueden ser conocidos (asociados a un servicio como ftp, web,…) o registrados dinámicamente

Sockets

- Protocolo de comunicación
  - Reglas para transmitir la información
  - TCP (Transport Control Protocol)
    - Orientado a conexión (stream)
    - 3 fases: establecer conexión/transferir datos/ cerrar conexión
  - UDP (User Datagram Protocol)
    - No orientado a conexión (datagram)
    - La aplicación tiene que implementar la fiabilidad
  - Unix Local comunication
    - Cuando se usan sockets para la transmisión local

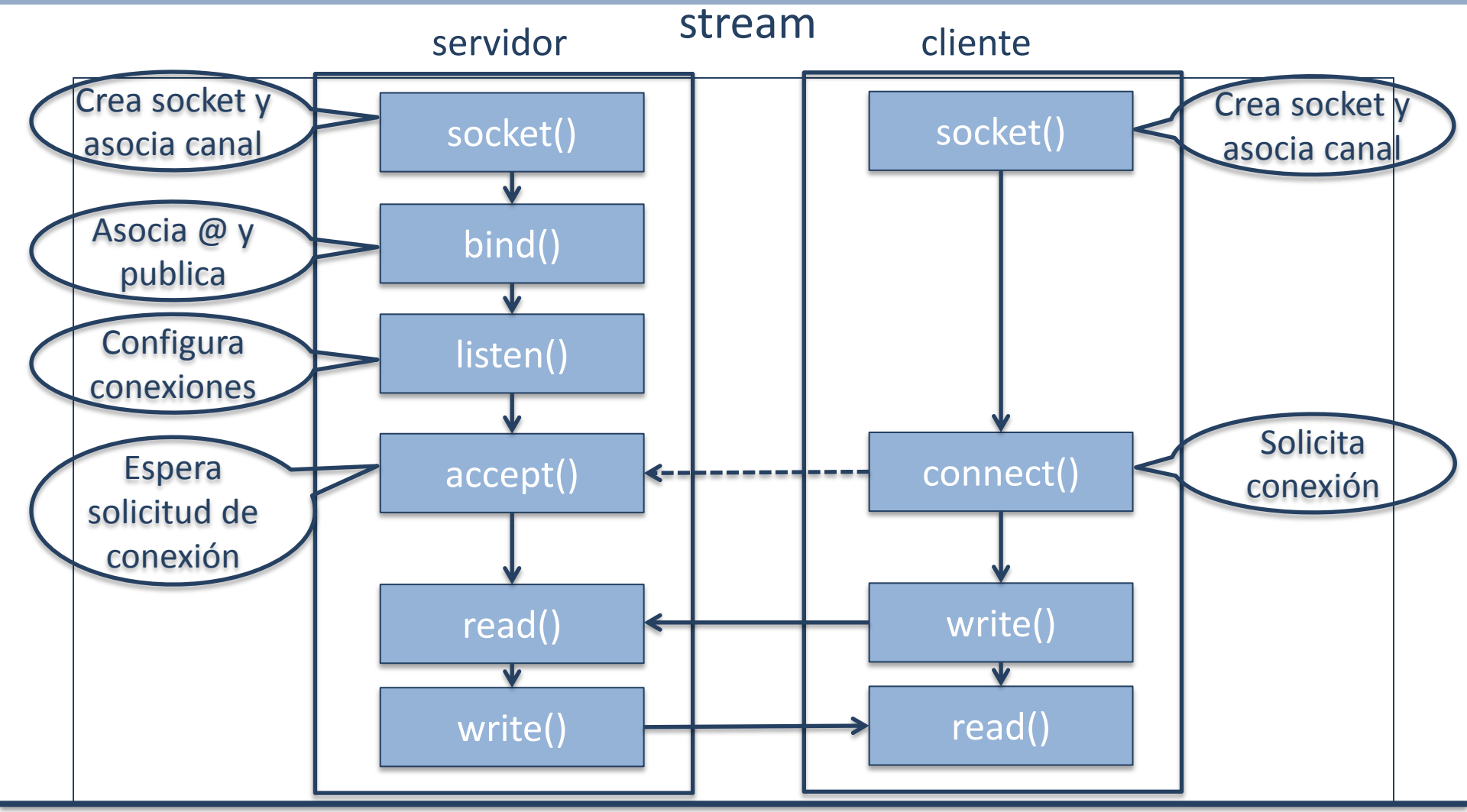
Sockets: nivel de usuario

- Modelo cliente-servidor
  - Servidor: gestiona el acceso a un recurso
    - Secuencial o interactivo:
      - Recoge petición y la sirve
      - Resto de clientes tienen que esperar
    - Concurrente:
      - Varios flujos o procesos sirviendo peticiones
      - Diferentes esquemas de creación de flujos
  - Cliente: peticiones de acceso al recurso

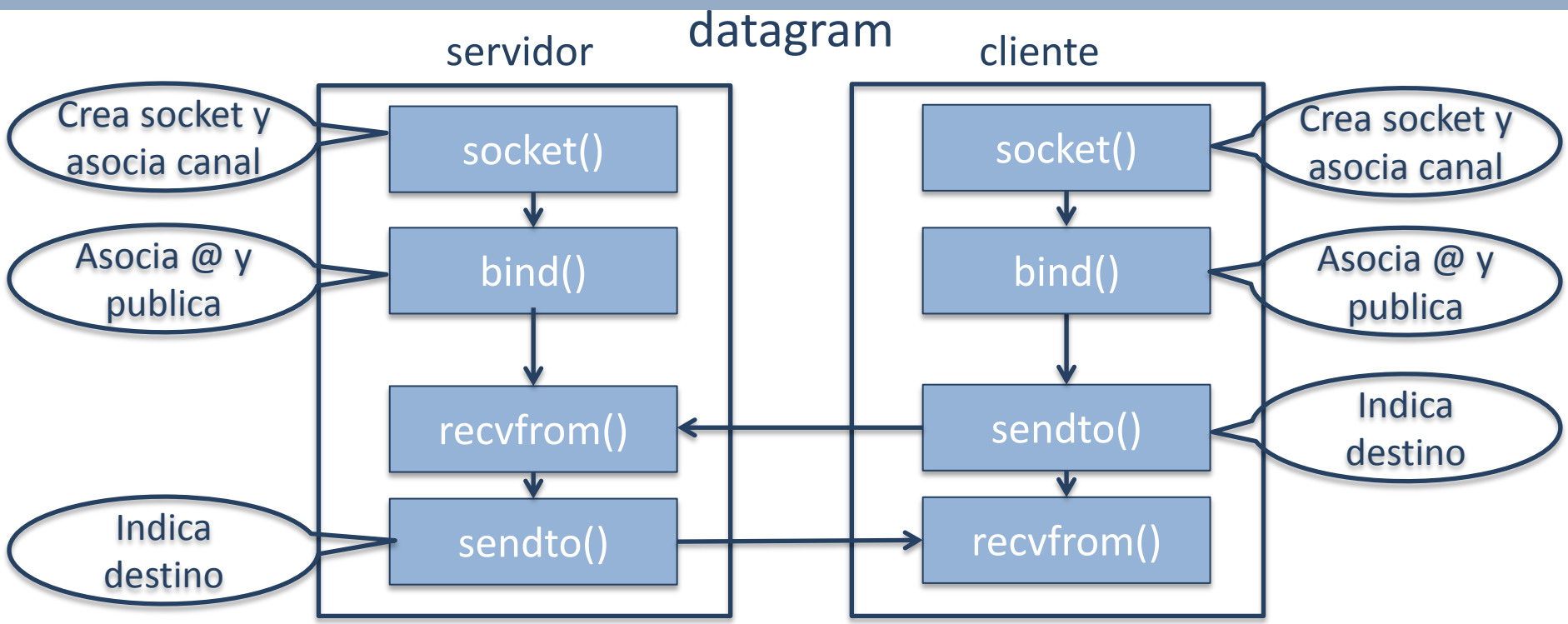
Sockets: nivel de usuario

- Modelo de comunicación
  - Un socket por proceso
    - La comunicación es full-duplex: los dos procesos pueden leer y escribir sin necesidad de añadir sincronización en el acceso al socket
  - Al crear un socket se le asocia directamente un canal
  - Si se quiere que el socket sea accesible por otros procesos se tiene que publicar su dirección
    - Comunicación stream: sólo hace falta que lo publique el servidor
    - Comunicación datagram: tienen que publicarlo tanto los clientes como el servidor
  - Accesos al socket
    - Comunicación stream: read y write
    - Comunicación datagram: sendto y recvfrom

Sockets: nivel de usuario



Sockets: nivel de usuario



Sockets: nivel de usuario

- Crea socket y asocia canal

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int af, int tipo, int protocolo)
```

af: familia (espacio de nombres para la dirección)
PF_UNIX
PF_INET

tipo: tipo de conexión
SOCK_STREAM
SOCK_DGRAM

protocolo: Si se deja el parámetro a 0 el sistema elige el apropiado

Devuelve el canal asociado al socket o -1 si error

Sockets: nivel de usuario

- Asocia dirección y publica

#include <sys/socket.h>
int bind (int canal, struct sockaddr *direccion, int tam_dirección)
canal: el creado con la llamada socket
dirección: dirección del socket
tam_dirección: número de bytes que ocupa la dirección del socket
devuelve 0 si ok y -1 si error

- Tipo de dirección depende de la familia de socket
  - AF_UNIX: nombre de fichero
  - AF_INET: IP+puerto
- Interfaz genérico y al usarla se especifica el tipo que toca

Sockets: nivel de usuario

- Tipo dirección
  - PF_UNIX: nombre de un fichero nuevo

```
#include <sys/un.h>
struct sockaddr_un {
        sa_family_t sun_family;
        char sun_name[UNIX_PATH_MAX]
}
```

  - PF_INET: dirección ip + puerto

```
#include <sys/netinet/in.h>
struct sockaddr_in {
        sa_family_t     sin_family; /* 1 byte*/
        struct in_addr      sin_addr;    /* 4 bytes */
        in_port_t           sin_port;    /* 1 byte */
        char                sin_zero [8]; /* no usado, debe ser 0 */
}
```

sin_family: PF_INET

sin_addr: constante INADDR_ANY representa IP de la máquina donde se ejecuta el código

sin_port : si 0, el sistema asigna uno libre; si no asegurarse de que no está ocupado. Para no interferir con sistema > 5000.

Sockets: nivel de usuario

- Endianismo: orden en el que se almacen un tipo de dato en memoria no es el mismo en todas las máquinas
  - Big Endian: byte de mayor peso en dirección baja
  - Little Endian: byte de mayor peso en dirección alta
- Formato estándar para transmitir enteros a través de la red
  - Funciones que adaptan la representación interna de la máquina a este formato y al revés
    - htons (host to network short)
    - htonl (host to network long)
    - ntohs (network to host short)
    - ntohl (network to host long)

Sockets: nivel de usuario

- Configurar conexión

int listen (int canal, int backlog)

backlog: número de peticiones pendientes que puede tener un servidor. Si se supera, el cliente recibirá un error en su petición de conexión

Devuelve 0 si ok y -1 si error

Sockets: nivel de usuario

- Espera solicitud de conexión

#include <sys/socket.h>

int accept(int canal, struct sockaddr *dirección, int *tam_dirección)

direccion: obtiene la dirección del cliente que solicita la conexión

tam_dirección contiene el tamaño que ocupa la dirección

devuelve canal para usar en la transmisión o -1 si error

Si conexiones pendientes acepta la primera

Si no, se bloquea hasta nueva petición (si se ha activado el flag N_DELAY con fcntl entonces no se bloquea y devuelve error)

Sockets: nivel de usuario

- Solicita conexión

#include <sys/socket.h>

int connect(int canal, struct sockaddr *direccion, int tam_direccion)

Si el servidor no puede antender la petición (se ha superado el parámetro indicado en el listen) devuelve -1. Si no se bloquea hasta que servidor acepta la conexión. Si se ha activado el flag N_DELAY no se bloquea y devuelve -1

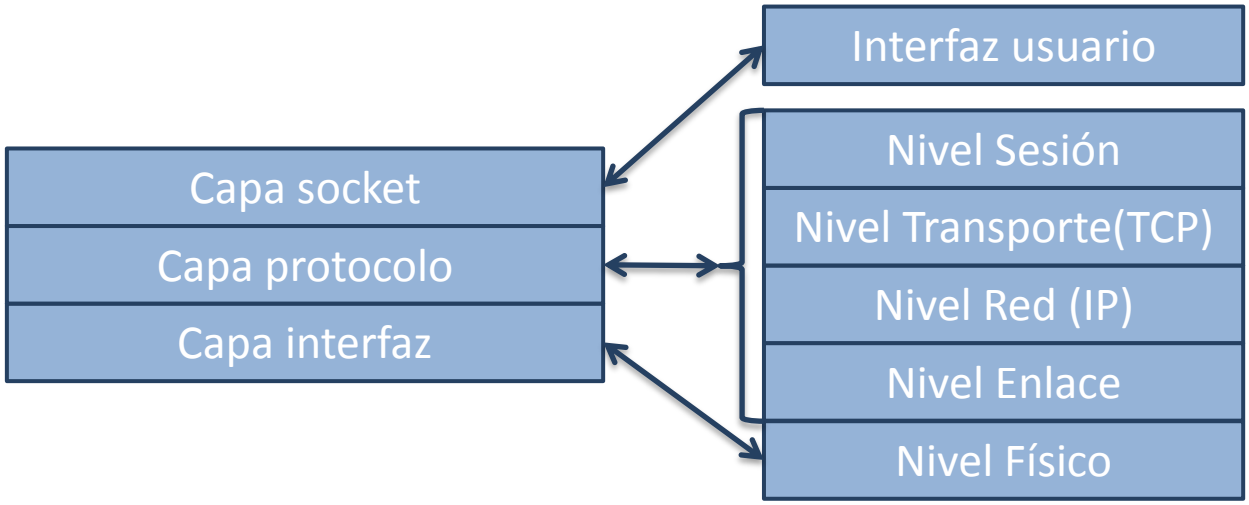
Sockets: nivel de usuario

- Lectura/Escritura
  - read/write
  - recvfrom/sendto

```
#include <sys/socket.h>
int recvfrom(int canal, void *buf, int length, int flags, void *from, int *fromlength)
int sendto (int canal, void *buf, int length, int flags, void *to, int tolength)
```

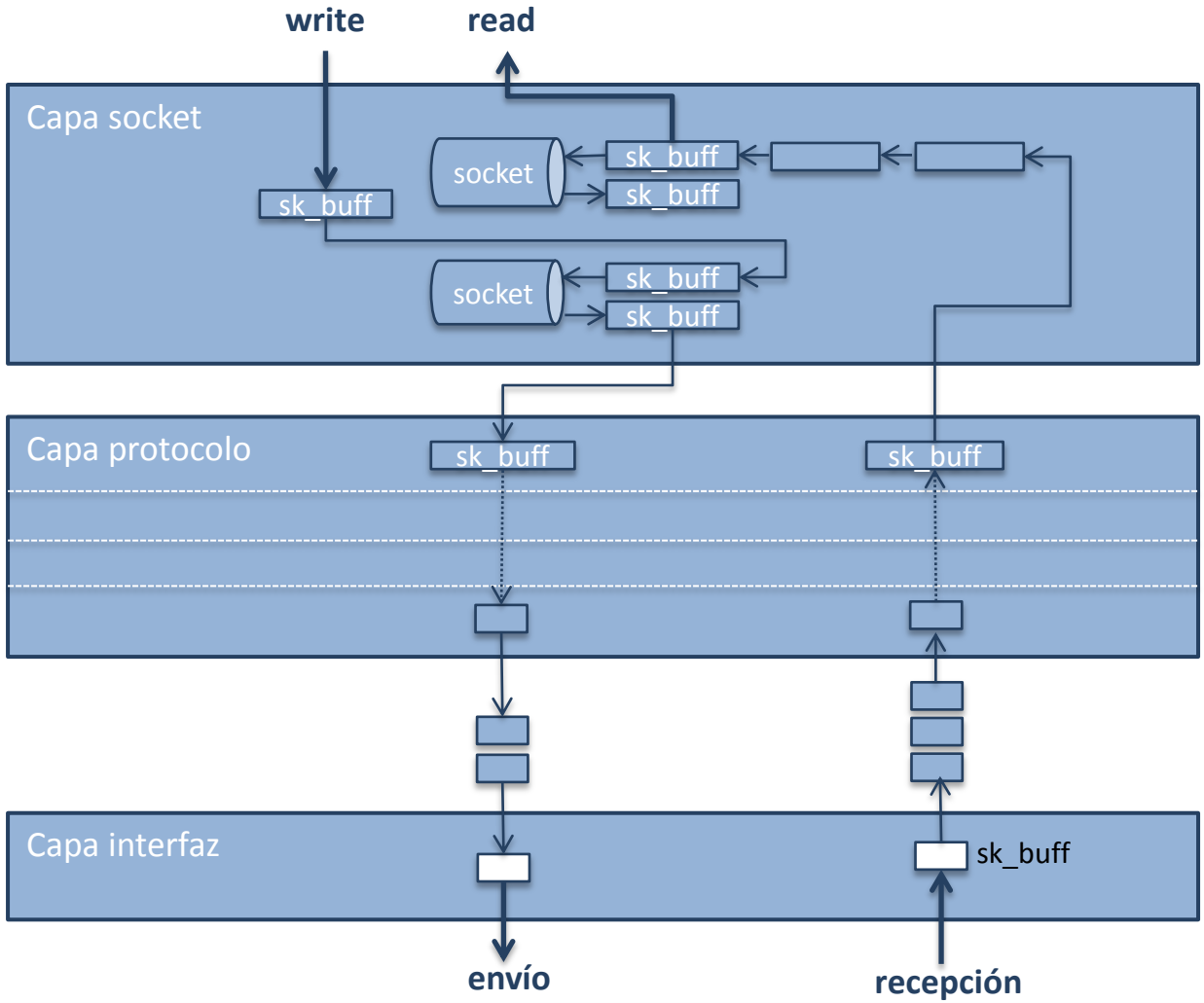
Sockets: nivel de sistema

- Ejemplo: Linux
  - Tres capas que se encargan de implementar las tareas correspondientes a la interfaz con la red, la gestión del protocolo y el interfaz con el usuario



- Comunicación entre las capas: colas de paquetes e interrupciones software
- Estructura de datos donde se almacena el paquete: sk_buff
- Cada socket tiene dos colas de sk_buff: recepción y envío

Sockets: nivel de sistema



Sockets: nivel de sistema

Recepción de un paquete

- Gestión interfaz
  - Gestión de la tarjeta de red
    - gestión interrupción: recepción de paquetes
    - Decodifica mensaje
  - Crea estructura de datos para almacenar el mensaje (sk_buff) recibido
  - Paso de paquetes a la capa superior: cola
  - Notificación al nivel superior mediante interrupción software
- Gestión protocolo
  - Existe una para cada protocolo soportado
  - Implementa las tareas de routing: selecciona paquetes para el host y enruta el resto
  - Para cada nivel del protocolo se añade al mensaje la información necesaria
  - Determina socket destino del mensaje
  - Paso de paquetes a la capa superior: cola para cada socket
  - Notificación al nivel superior mediante interrupción software
- Gestión sockets
  - Cuando se ejecuta una lectura sobre el socket
    - Accede a la cola del socket para seleccionar el mensaje
    - Transfiere a la zona de memoria del usuario la información recibida
  - Si en el momento de recibir el paquete había un proceso bloqueado intentando leer del socket se desbloquea

Sockets: nivel de sistema

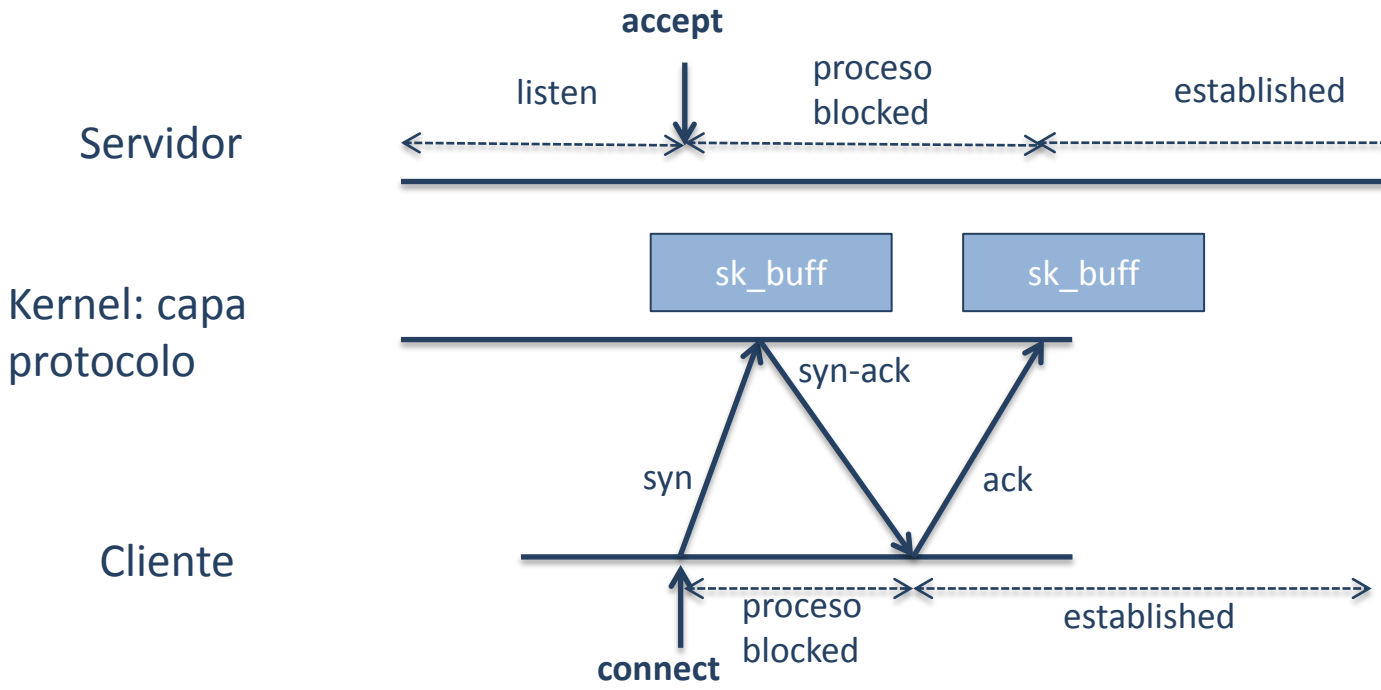
Envío de un paquete

- Gestión sockets
  - Cuando se ejecuta una escritura sobre el socket
    - Crea estructura sk_buff
    - Transfiere de la zona de memoria del usuario al sk_buff la información a transmitir
  - Paso de paquetes a la capa inferior: cola para cada socket
- Gestión protocolo
  - Existe una para cada protocolo soportado
  - Para cada nivel del protocolo se añade al sk_buff la información necesaria
  - Implementa las tareas de routing: determina dirección destino
  - Paso de paquetes de la capa inferior: cola
- Gestión interfaz
  - Gestión de la tarjeta de red
    - Codificación mensaje y programación tarjeta de red

Sockets: nivel de sistema

Establecimiento de conexión

- Recordatorio: 3WHS (3-way handshake)
  - Protocolo para establecer conexión en TCP y relación con llamadas a sistema de sockets



Sockets: nivel de sistema

Establecimiento de conexión en Linux

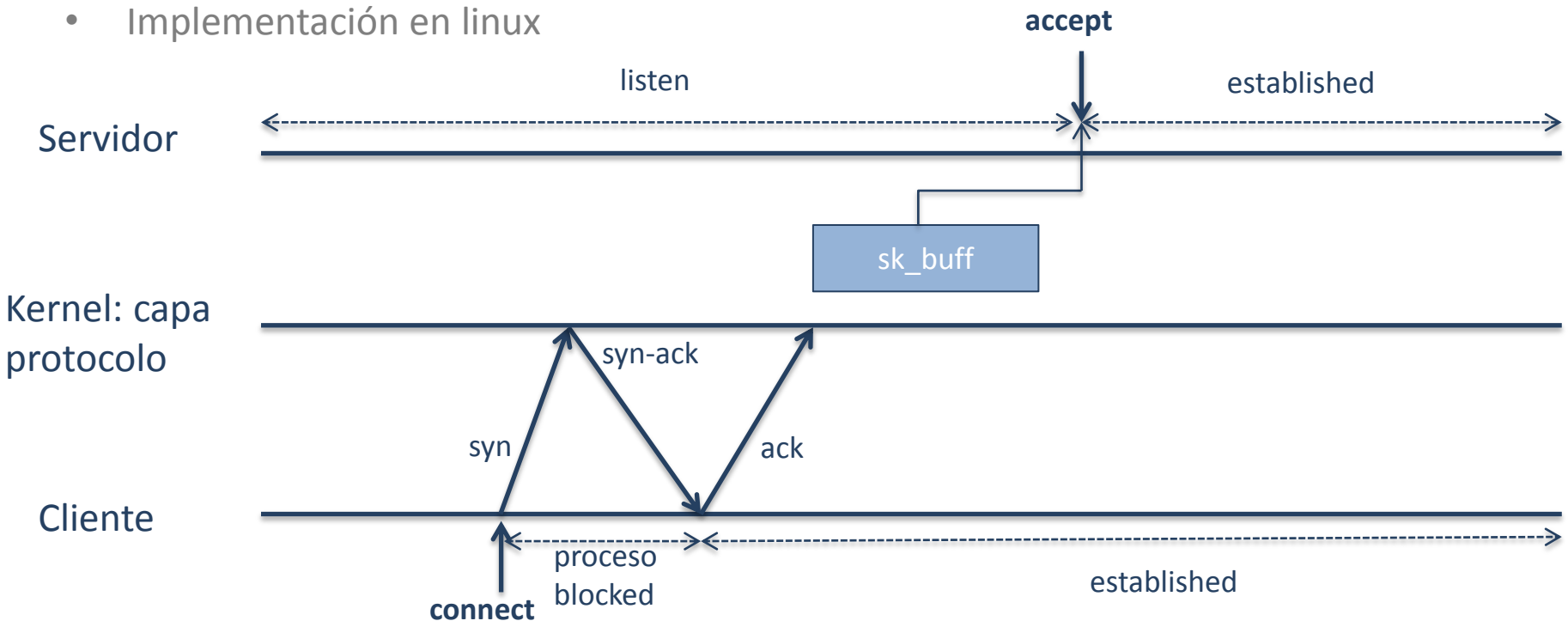
- Gestión protocolo
  - Al recibir un paquete con la solicitud de conexión encola el sk_buff correspondiente en el socket objetivo
    - Si el servidor está esperando en un accept, se le desbloquea para que continúe con la ejecución
    - Si el servidor no está esperando en un accept, pre-acepta conexiones
      - Evitar denegación de servicio
  - Devuelve al cliente un paquete con el syn-ack
  - Cuando el cliente reciba el syn-ack responde con el ack y ya puede empezar a enviar mensajes
  - Listen no tiene efecto en la mayoría de casos
    - Mientras la cola del socket no llegue al límite de su tamaño

- Gestión socket
  - Cuando el servidor ejecuta accept
    - Si hay una petición de conexión encolada se gestiona y el servidor continúa la ejecución
    - Si no hay ninguna petición de conexión el servidor se bloquea a la espera de una petición

Sockets: nivel de sistema

Establecimiento de conexión

- Implementación en linux



Sockets: nivel de sistema

- Principales estructuras de datos
  - Socket (linux/net.h)
    - Características del socket independientes del protocolo y del tipo de socket
    - Contiene apuntador a operaciones específicas y estructura de tipo sock
  - Sock (include/net/sock.h)
    - Representación de socket del nivel de red
    - Contiene dos colas de sk_buff: una para los paquetes a enviar y otra para los paquetes a recibir
  - sk_buff (linux/skbuff.h)
    - Contiene la información de un paquete: datos que quiere transmitir el usuario y datos de gestión añadidos por el protocolo
    - Pensado para optimizar el proceso de añadir/quitar información a medida que se atraviesan las capas del protocolo
      - Estructura formada por punteros a la información y así evitar la copia de los datos

Sockets: nivel de sistema

```
struct sk_buff {
        /* These two members must be first. */
        struct sk_buff          *next;
        struct sk_buff          *prev;

        struct sock             *sk;
        struct skb_timeval      tstamp;
        struct net_device       *dev;
        struct net_device       *input_dev;

        union {
                struct tcphdr   *th;
                struct udphdr   *uh;
                struct icmphdr  *icmph;
                struct igmphdr  *igmph;
                struct iphdr    *ipiph;
                struct ipv6hdr  *ipv6h;
                unsigned char   *raw;
        } h;
```

http://lxr.linux.no/linux+v2.6.14/include/linux/skbuff.h#L211

Pipes: nivel de usuario

- Intercambio de datos entre procesos que se ejecutan en la misma máquina
- Dos tipos
  - Pipes sin nombre
    - No tienen un nombre que las represente
    - Sólo la pueden utilizar procesos relacionados por herencia
  - Pipes con nombre
    - Tienen un nombre en el sistema de ficheros que las representa
    - Al crear el nombre se especifican los permisos de acceso: cualquier proceso con permiso podrá usarla
- Los dos tipos de pipes se acceden de la misma manera
  - Comportamiento FIFO
  - A medida que se leen bytes desaparecen de la pipe
  - Un canal de comunicación común para lecturas y escrituras
    - Están pensadas para ser unidireccionales: un proceso lector y un proceso escritor
      - Si un proceso escribe y a continuación leer recibirá los datos que ha escrito
    - Sincronización en el acceso

Pipes: nivel de usuario

Aceso a una pipe: lecturas

- Llamada a sistema read
  - Si hay suficientes datos en la pipe para servir la lectura se devuelven los datos que se han pedido
  - Si no hay suficientes datos (pero no está vacía) se devuelven los datos que hay
  - Si se intenta leer de una pipe vacía
    - Si no hay ningún canal de escritura asociado a la pipe
      - Devuelve 0
    - Si hay algún canal de escritura asociado a la pipe el proceso se bloquea hasta que alguien escribe algo o hasta que se cierran todos los canales de escritura
    - **Importante cerrar los canales de escritura** en la pipe que no son necesarios

Pipes: nivel de usuario

Aceso a una pipe: escrituras

- Llamada a sistema write
  - Si hay espacio en la pipe se escriben los datos y se acaba
  - Si no hay sufieciente espacio se escriben los datos que quepan
    - Atómicamente si la cantidad a escribir es menor que el tamaño de la pipe.
  - Si se intenta escribir y la pipe está llena, el proceso se bloquea hasta que se pueda escribir
  - Si no hay ningún canal de lectura asociado a la pipe, el proceso recibe SIGPIPE y el write acaba con error

Pipes: nivel de usuario

Creación de dispositivo virtual

- Pipe sin nombre

```
#include <unistd.h>
int pipe(int fd[2])
```

  - Crea dos canales (los dos primeros libres): el primero asociado al extremo de lectura de la pipe y el segundo asociado al extremo de escritura
- Pipe con nombre
  - Es necesario que tenga un nombre en el sistema de ficheros

```
#include <syst/stat>
int mknod(char *path, mode_t mode, dev_t dev )
```

    - Se usa para crear dispositivos lógicos de todo tipo
      - Mode codifica tipo de dispositivo y permisos de acceso
      - Dev contiene major y minor (se ignora en el caso de una pipe)
      - Para crear una pipe:

```
mknod("nombre", S_IFIFO|S_IRUSR|S_IWUSR);
```

  - Se hace open sobre ese nombre como con cualquier otro dispositivo que se quiera usar

```
open("nombre", O_RDONLY);
```

  - Si al hacer el open no hay ningún canal abierto para hacer el acceso complementario el proceso se bloquea

Pipes: nivel de usuario

Configuración de la pipe: evitar bloqueos en el uso

- Asociar flags O_NONBLOCK al dispositivo virtual asociado a la pipe
  - En el open de la pipe con nombre: modo de acceso

```
open("nombre", O_RDONLY|O_NONBLOCK);
```

  - Mediante la llamada a sistema fcntl
    - Modifica el comportamiento de los dispositivos virtuales
    - Las modificaciones dependen del dispositivo lógico asociado al dispositivo virtual

```
#include <fcntl.h>
int fcntl(int fildes, int cmd, ...);
```

    - Para activar flag:

```
fnctl (fd_pipe, F_SETFL, O_NONBLOCK);
```

Pipes: nivel de usuario

- Comportamiento si O_NONBLOCK
  - Open pipe con nombre:
    - Sólo lectura: devuelve canal sin bloquearse
    - Sólo escritura: devuelve error ((errno==ENXIO)
  - Lectura pipe vacía con escritores: devuelve error (errno == EAGAIN)
  - Escritura pipe llena: devuelve error (errno == EAGAIN)

Pipes: nivel sistema

- Tienen asocciado un inode
  - Pipes con nombre: se crea al crear el nombre
  - Pipes sin nombre: se crea al crear la pipe
- Inode
  - Campo pipe_inode_info
    - Contiene buffers de memoria y las operaciones sobre ese buffer
  - Operaciones de acceso a la pipe
    - Acceden al buffer
    - Hay una estructura file_operations para cada modo de acceso y para cada tipo de pipe (con nombre y sin nombre)
  - Semáforo para implementar los bloqueos

Sistema de Ficheros: Índice

- Introducción
- Descripción básica del hardware
- Visión estática
  - Organización del espacio de disco
  - Gestión del espacio de disco
  - Gestión del espacio de nombres
  - Ejemplo: Linux Ext2
- Visión dinámica
  - Arquitectura del Sistema de Ficheros
    - Acceso a diferentes sistemas de ficheros
  - Ejemplo: Linux Ext2

Definiciones: fichero

- ¿Qué es un fichero?
  - Para el usuario
    - Conjunto de información relacionada que tiene un nombre
  - Para el sistema
    - Una secuencia de bytes
    - Dispositivo lógico

Definiciones: sistema de ficheros

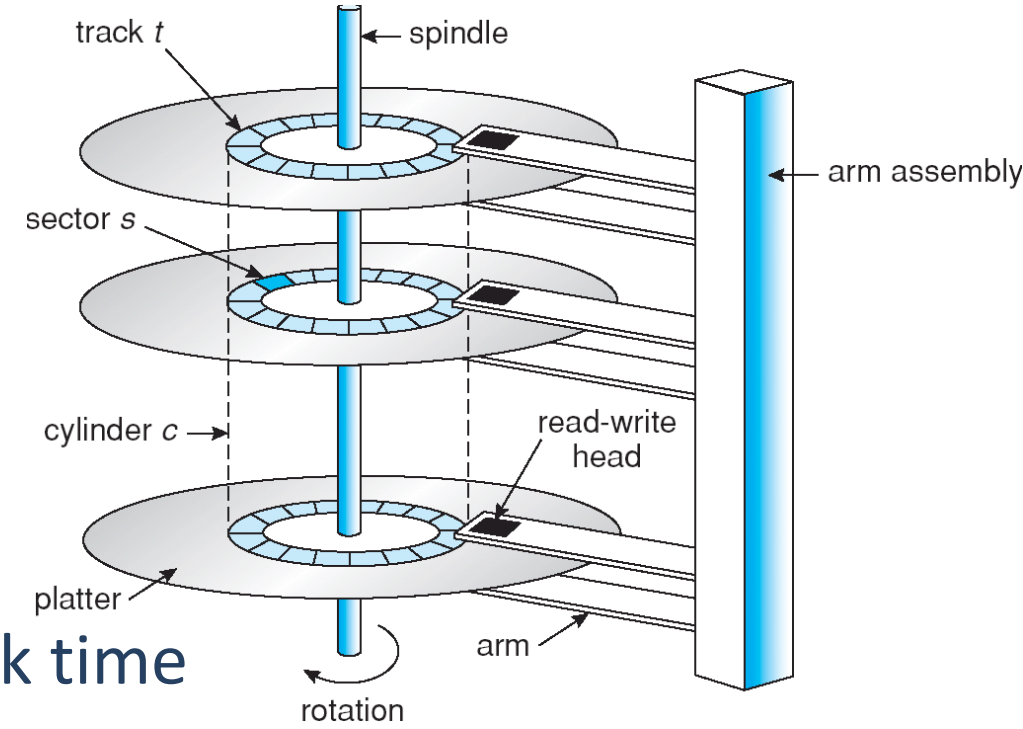
- ¿Qué es el Sistema de Ficheros?
  - Conjunto de estructuras de datos y algoritmos para almacenar, localizar y recuperar información de un dispositivo de almacenamiento persistente (ej. Disco)
- Tareas del sistema de ficheros
  - Gestionar el espacio del almacenamiento
    - Asignar espacio a los ficheros
    - Liberar el espacio de los ficheros eliminados
  - Encontrar/almacenar los datos de los ficheros
  - Organizar los ficheros en el sistema
  - Garantizar las protecciones de los ficheros
  - Gestión del espacio de nombres

Definiciones: sistema de ficheros (II)

- Interfaz de usuario
  - Se accede mediante el interfaz de E/S
    - Acceso a ficheros: open, read, write, close, ..
    - Gestión: link, unlink, chmod, chown,….

Descripción básica del HW

- Discos mecánicos
  - Unidad de trabajo: sector
    - Asignación y transferencia
    - 512 bytes
  - Tiempo de acceso
    - Seek time
      - Posicionamiento en pista
      - Espera sector
    - Transferencia
  - Tiempo de acceso dominado por el seek time



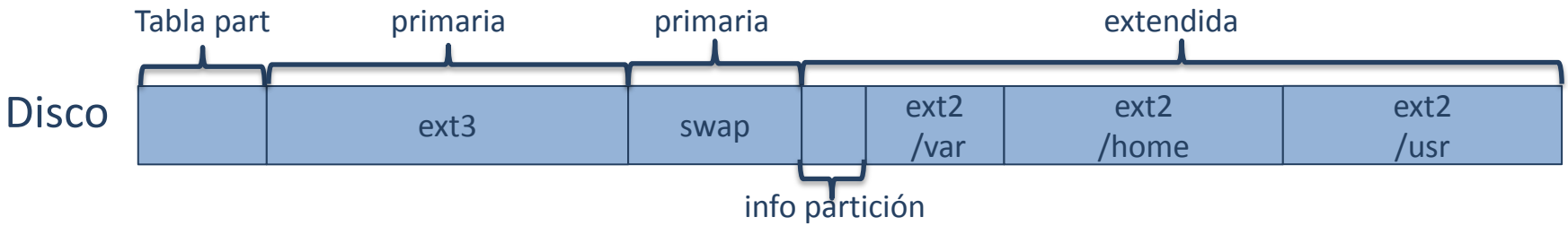
Descripción básica del HW (II)

- Almacenamiento SSD (Solid State **Drive)**
  - Unidad de trabajo: sector
    - Asignación y transferencia
    - 4096 bytes
  - Tiempo de acceso
    - Transferencia
  - No hay tiempo de seek time (acceso directo)
- SF tradicionales no pensados para las características de SSD y su rendimiento degrada con el uso

Organización del espacio de disco

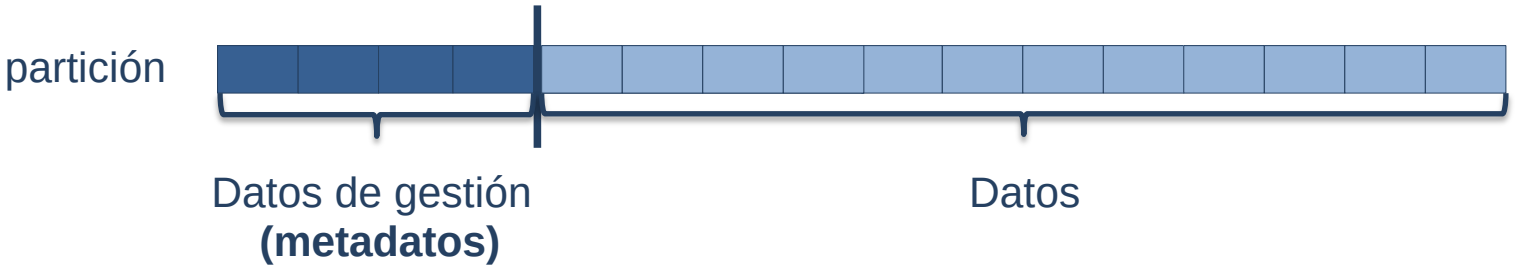
- Particiones de disco
  - Cada disco puede dividirse lógicamente en particiones
  - Cada partición puede soportar un sistema de ficheros diferente
  - Máximo de 4 particiones
- Tipos de particiones
  - Primaria
    - Soporte para un sistema de ficheros
  - Extendida
    - Objetivo: solventar la limitación del número máximo de particiones
    - Soporte para dividir una partición primaria y crear nuevas particiones lógicas

Visión estática



Organización del espacio de disco (II)

- Organización de una partición
  - En una partición tenemos
    - Datos: información guardada por el usuario
    - Metadatos: información necesaria para gestionar los datos y guardada por el sistema de ficheros
  - Ejemplo:



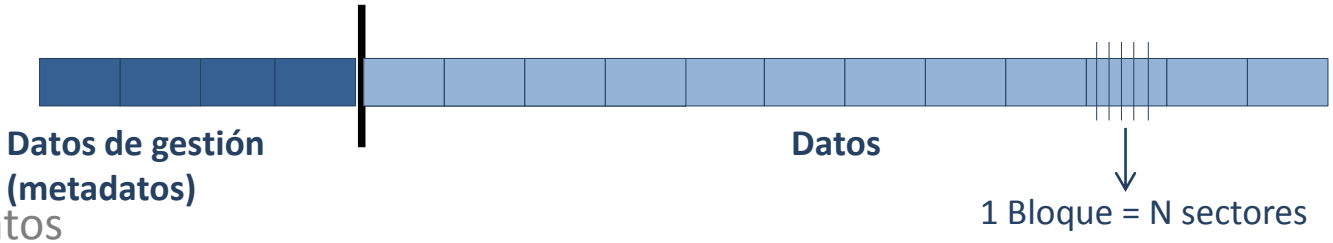
Visión estática

Gestión del espacio de disco

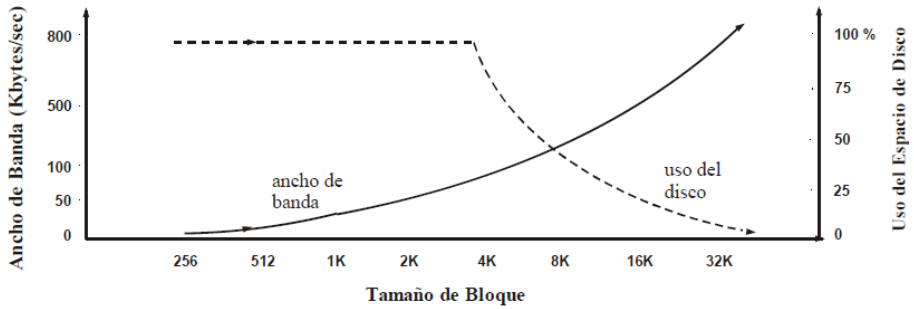
- Bloque
  - Unidad de trabajo del sistema de ficheros
    - Acceso y transferencia
  - 1 Bloque == N sectores
    - Alternativas en la correspondencia bloque <-> sector
      - Fija, variable, N grande, N pequeña, ….
- Gestión del espacio libre
  - Localización de los bloques libres
- Gestión del espacio ocupado
  - Asignación de bloques a ficheros
  - Localización de los bloques de un fichero

Visión estatica

Contenido de una partición: Datos

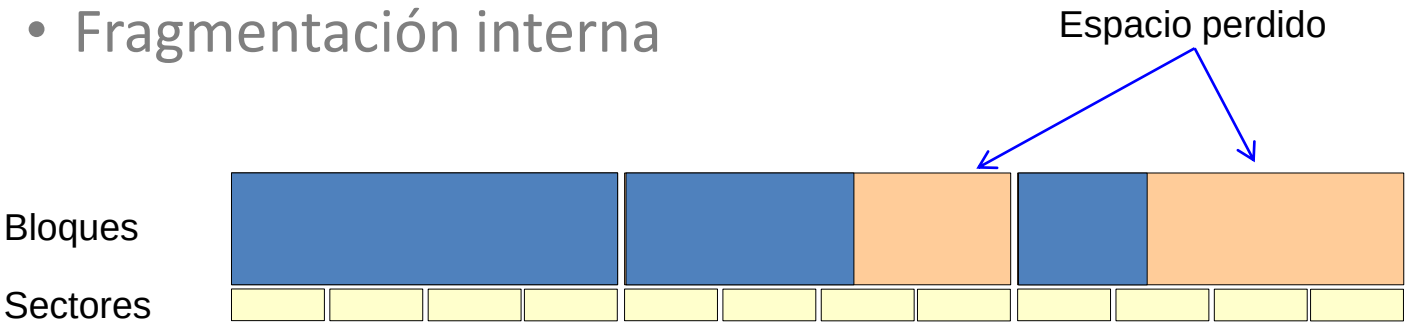


- Datos
  - Información organizada en **bloques**
  - **Sector:** unidad de transferencia (definida por el Hw)
  - **Bloque:** unidad de asignación (definido por el SO)
  - qué tamaño definimos? (Fijo/Variable, Grandes/Pequeños)
    - Bloques Pequeños
      - Aprovecha mejor el espacio, pero hay que hacer muchos accesos
    - Bloques Grandes
      - Aumenta el rendimiento (menos accesos a disco por KB), pero desperdicia espacio



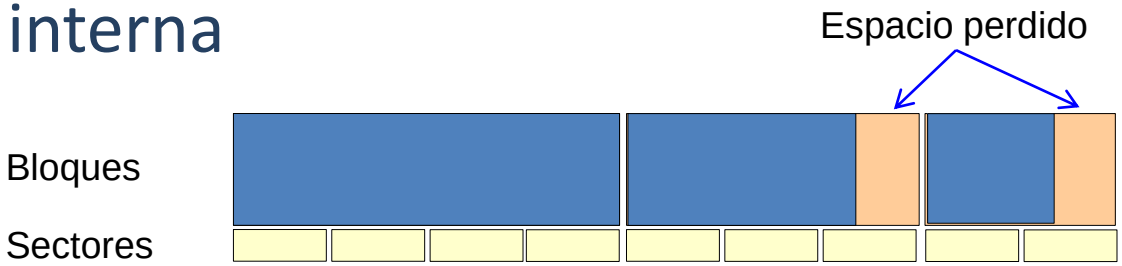
Datos: Bloques de tamaño fijo

- Todos los bloques tienen el mismo tamaño
  - Muy sencillo de implementar
  - Compromiso en el tamaño de bloque
    - Eficiencia
    - Fragmentación interna

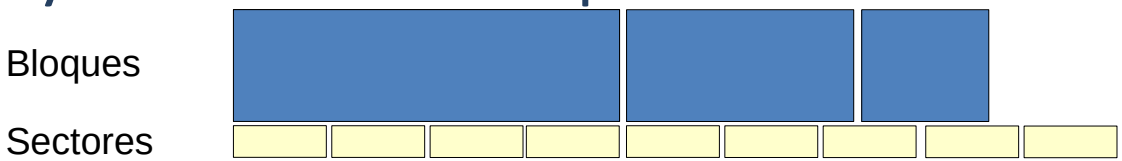


Datos: Bloques de tamaño variable

- Bloques sin compartir sectores
  - Fragmentación interna



- Bloques compartiendo sectores
  - Uso eficiente del espacio
  - Complejidad muy elevada en la implementación



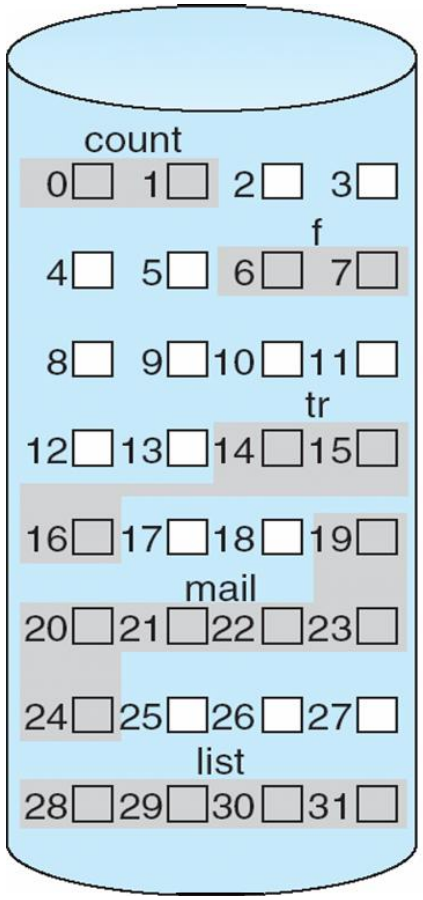
Gestión del espacio ocupado

- Proporcionar espacio de almacenamiento secundario a los archivos
- El SF utiliza una estructura donde guarda la relación entre el archivo y su espacio asignado
  - Normalmente accesible a través del directorio
  - Almacenada en el SF (opcionalmente en memoria)
- El espacio se asigna en forma de bloques contiguos (secciones)…
  - Cuantos bloques consecutivos?
- … o en forma de bloques remotos
- Diversos mecanismos de asignación, pero nos centraremos en:
  - Asignación **contigua**
  - Asignación **enlazada** y **enlazada en tabla** (FAT)
  - Asignación **indexada** e **indexada multinivel**

Asignación contigua

- Todos los bloques del archivo se asignan de manera consecutiva
  - CDROM, DVDs, …
- Localización: para cada archivo se necesita guardar
  - Bloque inicial
  - Longitud del archivo
- Ventajas
  - Acceso eficiente al dispositivo
  - Localización del bloque i-ésimo sencilla
- Desventajas:
  - Se produce fragmentación externa
  - Necesita asignación previa (determinar el tamaño a priori)

Asignación contigua

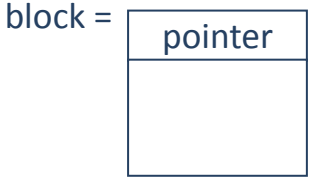


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

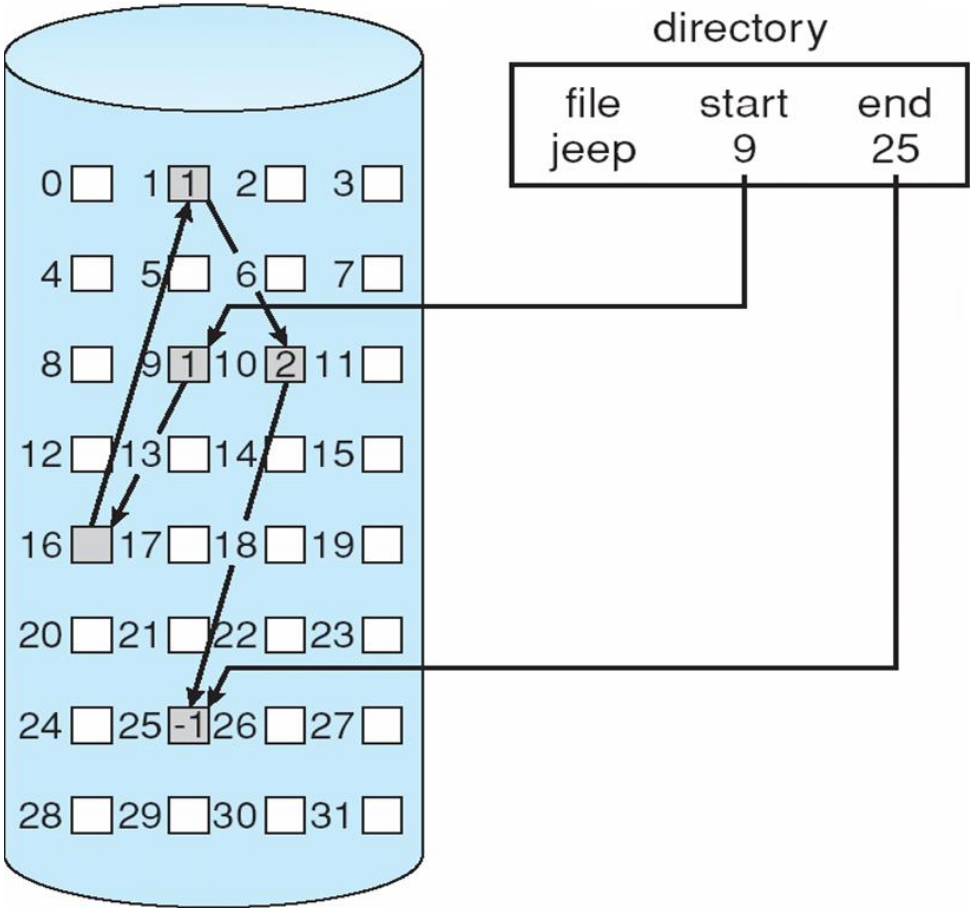
Asignación encadenada

- Cada bloque de datos reserva espacio para un puntero que indica cual es el siguiente bloque del archivo



- Localización: para cada archivo se necesita guardar
  - Bloque inicial
- Ventajas:
  - asignación previa o dinámica
  - no hay fragmentación externa
- Desventajas:
  - para acceder al bloque i-ésimo hay que recorrer los anteriores
    - adecuado para accesos secuenciales
    - terrible para accesos directos
  - Poca fiabilidad→ Si hay un fallo en un bloque es muy crítico

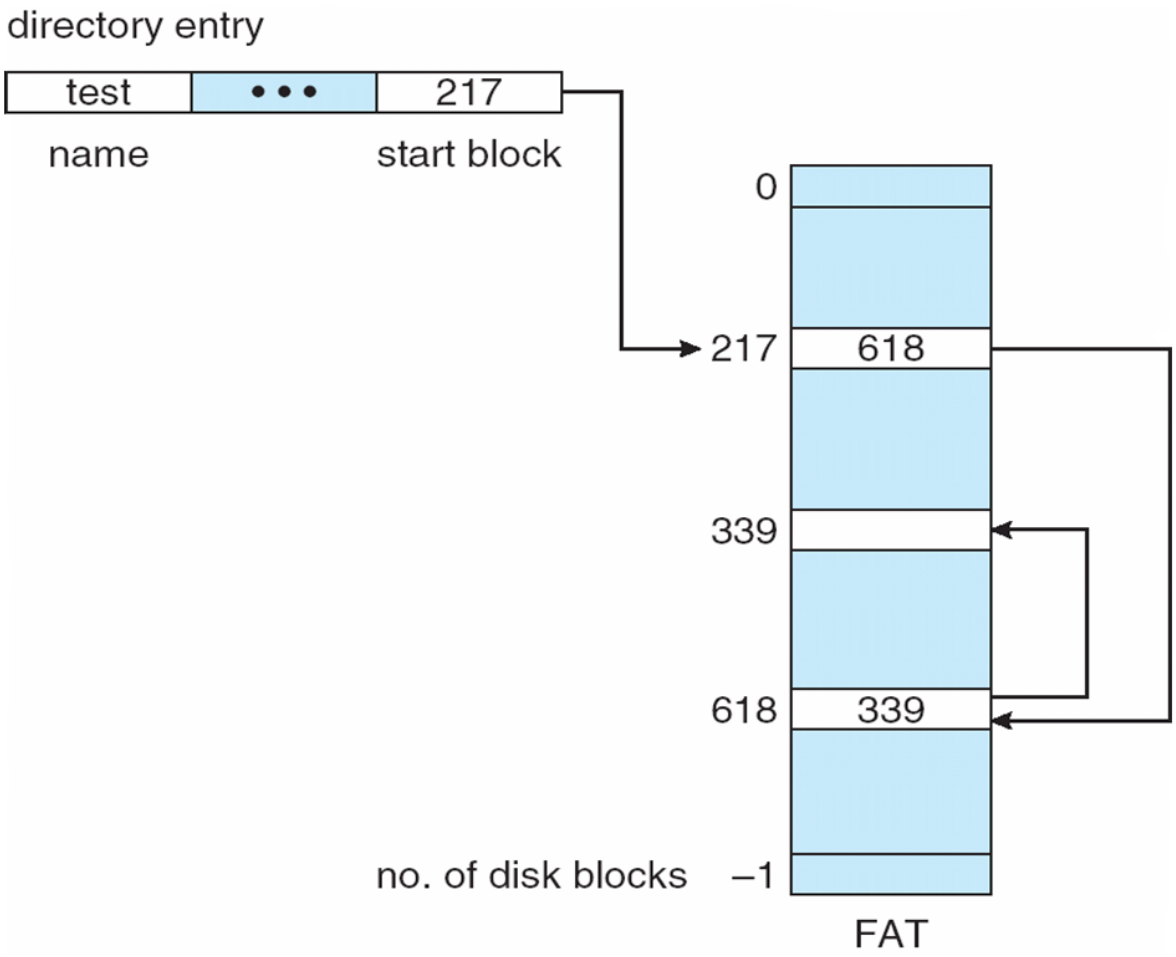
Asignación encadenada



Asignación encadenada en tabla

- Se enlazan los bloques con punteros pero los punteros se guardan en una tabla en lugar de ponerlos en los bloques de datos
- Esta tabla se suele llamar **FAT** (*File Allocation Table*)
- Localización: para cada archivo se necesita guardar
  - Nombre + bloque inicial + tamaño (+ info adicional)
- Características
  - Para acceder al bloque i-ésimo, basta con acceder a la tabla
  - Se puede replicar la tabla para aumentar la fiabilidad
  - Se puede utilizar para gestionar el espacio libre
- Inconvenientes
  - Problemas con discos grandes (tabla grande)

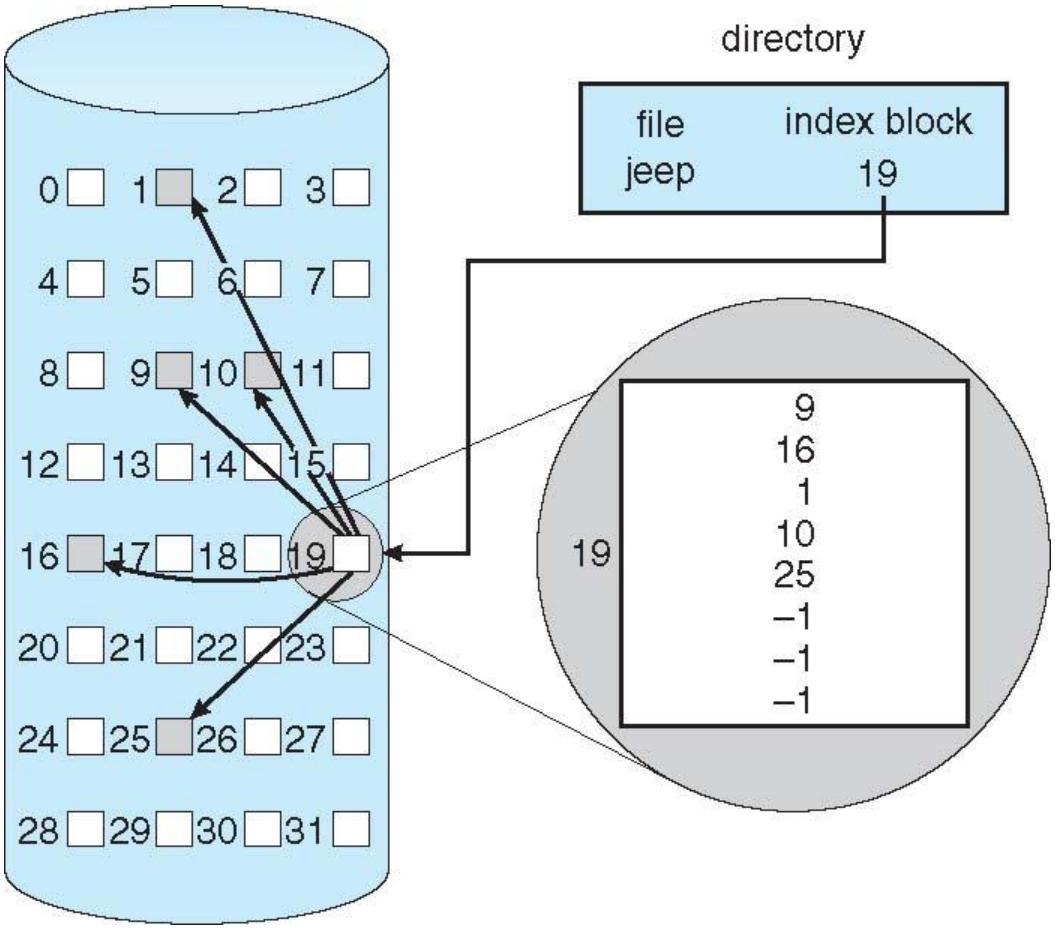
Asignación encadenada en tabla (FAT)



Asignación indexada

- Existe un bloque índice para cada archivo
- Este índice contiene un vector de identificadores de bloques
- Al final hay un puntero al siguiente bloque índice (o a NULL)
- Traducción: en cada entrada referencia a bloque de índices
- Ventajas:
  - Buen acceso secuencial y directo
- Desventajas:
  - Pérdida de espacio (bloques de índices grandes)
  - Muchos accesos en ficheros grandes (bloques de índices pequeños)

Asignación indexada



Asignación indexada multinivel

- En el bloque índice existen algunos apuntadores indirectos
  - apuntan a nuevos bloques índices
- Se crea una estructura jerárquica de bloques índice
- i-nodo: contiene índices directos e índices indirectos
- Ventajas
  - Muy pocos accesos, incluso en ficheros grandes
  - Poca perdida de espacio en ficheros pequeños
- Inconvenientes
  - Añadir o borrar datos que no están al final del fichero

Gestión del espacio libre

- Bitmaps
- Chained free portions
- Indexing
  - Free space as a file
- Free block list

Gestión del espacio de nombres

- El espacio de nombres ofrece al usuario una visión de todos los ficheros contenidos en el sistema de ficheros
- Cada fichero debe tener un nombre simbólico
- Define reglas específicas para crear nombres
  - Ej: En MSDOS nombres de 8 carácteres + 3 para extensión
- Permite traducir los nombres de los ficheros a su ubicación en el sistema de ficheros

Visión estática

G. espacio de nombres: Directorios

- Archivo especial gestionado por el SO
  - Llamadas específicas de acceso y creación
    - No accesible directamente mediante read/write
- Da acceso a la información sobre los archivos
  - Atributos
    - Tipo de archivo
    - Fechas de creación, acceso, modificación, …
    - Propietario
    - Permisos
    - Tamaño
    - …
  - Ubicación en el dispositivo de almacenamiento
- Si la información está dentro del directorio dificulta la creación de links (diferentes nombres para un mismo archivo)
  - Estructura separada y el directorio sólo referencia a ella
- Operaciones gestión
  - Buscar, crear, borrar, enumerar, actualizar entradas

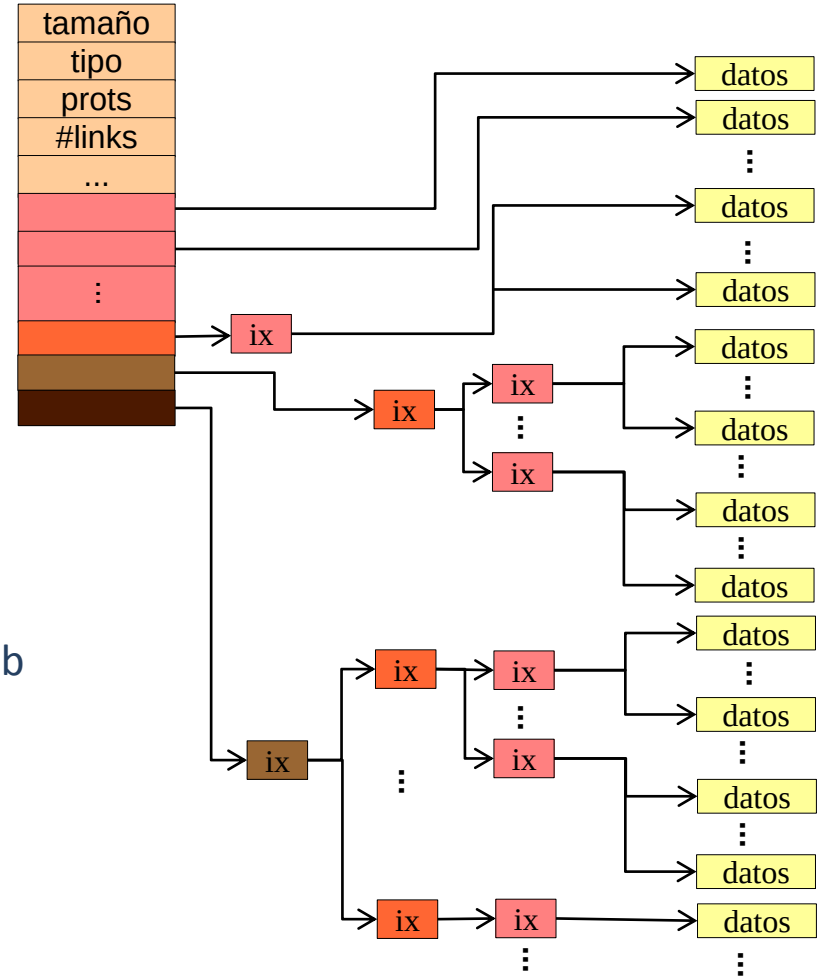
Ejemplo: Unix Ext2

- Metadatos
  - Sector de arranque (Boot)
    - Info básica para arrancar el SO instalado en la partición
  - Superbloque:
    - Formato del SF (tamaño bloque,, #inodes, #inodes libres, #bloques datos ,#bloques libres,…)
    - Gestión espacio libre/ocupado: inodes, cuál es el inode raíz, acceso a bloques libres, acceso a inodes libres
  - Inodos
    - Asignación de bloques indexada multinivel
- Datos
  - Bloques de tamaño fijo
- Directorio
  - Enlaza un nombre de fichero con su inodo
  - Los atributos del fichero se encuentran en el inodo

Visión estática

Ejemplo: Unix Ext2

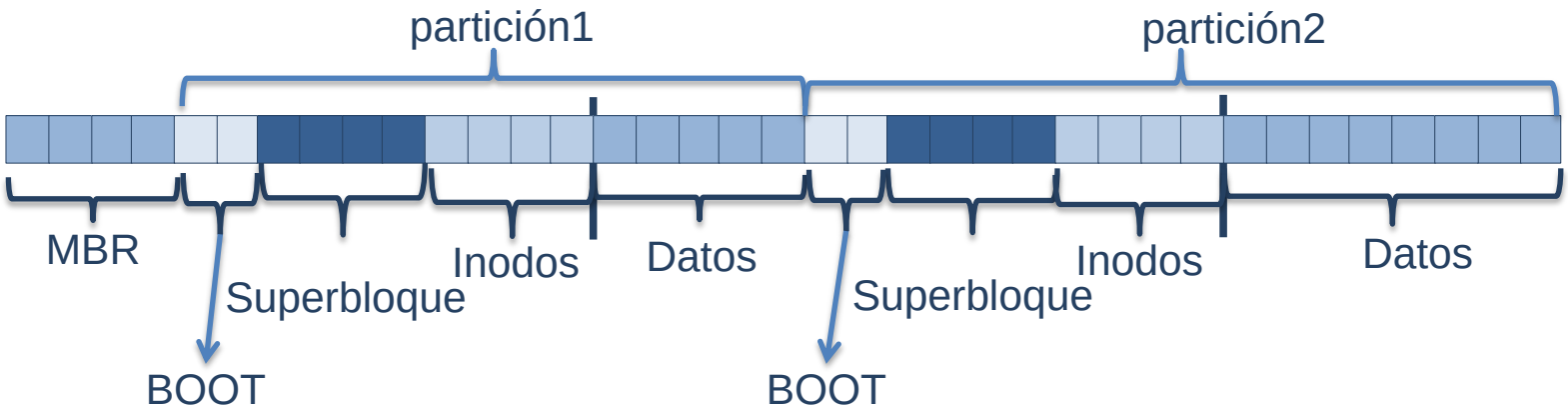
- Inodo
  - Bloque con información del archivo
    - Tamaño, tipo, protecciones, …
  - Indices a bloques de datos (1-4Kb)
    - 10 índices directos
      - 10-40Kb
    - 1 índice indirecto
      - 256-1024 bloques == 256Kb – 4 Mb
    - 1 índice indirecto doble
      - 65K – 1M bloques == 65Mb – 4Gb
    - 1 índice triple indirecto
      - 16M – 1G bloques == 16Gb – 4Tb



Visión estática

Ejemplo: Unix Ext2

- Organización disco con 2 particiones ext2

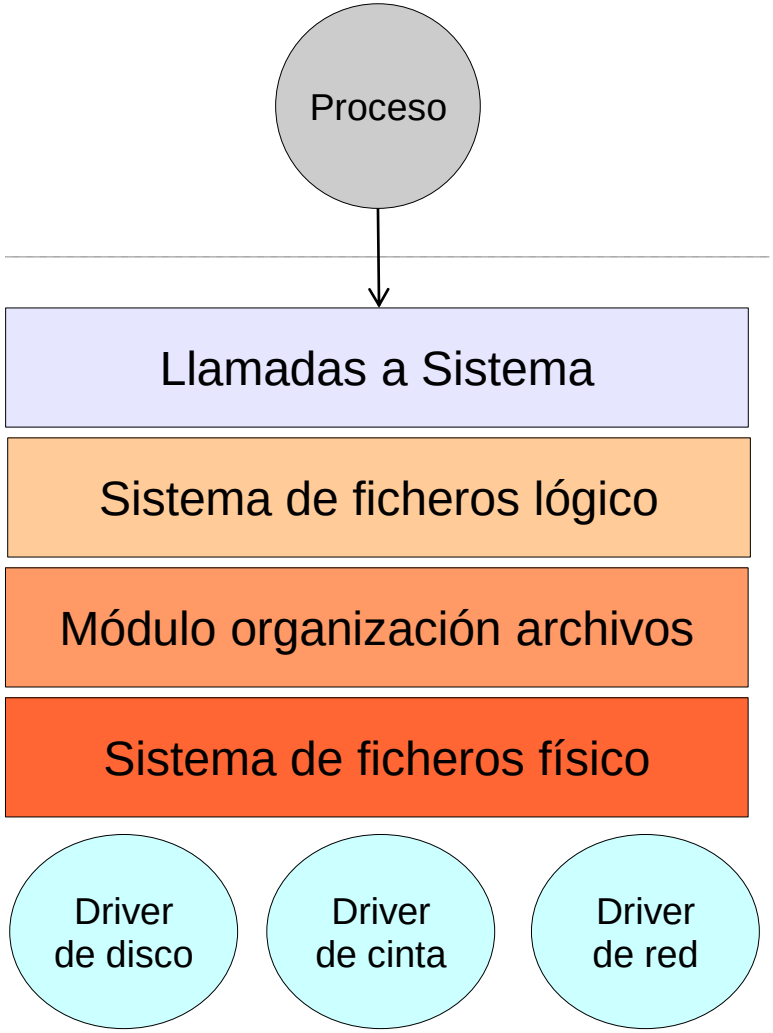


Visión dinámica

- Arquitectura del sistema de ficheros
  - Capas
  - Montaje de sistemas de ficheros
  - VFS
  - Windows
- Ejemplo: Linux
  - Estructuras internas
  - Read y open

Arquitectura del sistema de ficheros

- Sistema de ficheros lógico
  - Proporciona la abstracción fichero para realizar la E/S
  - Gestión del espacio de nombres (directorios)
  - Información sobre fichero para siguiente nivel
- Módulo de organización archivos
  - Correspondencia archivos <-> bloques
  - Gestión espacio libre/ocupado
- Sistema de ficheros físico
  - Emite comandos al driver del dispositivo para leer/escribir bloques



Visión dinámica

Montaje de sistema de ficheros

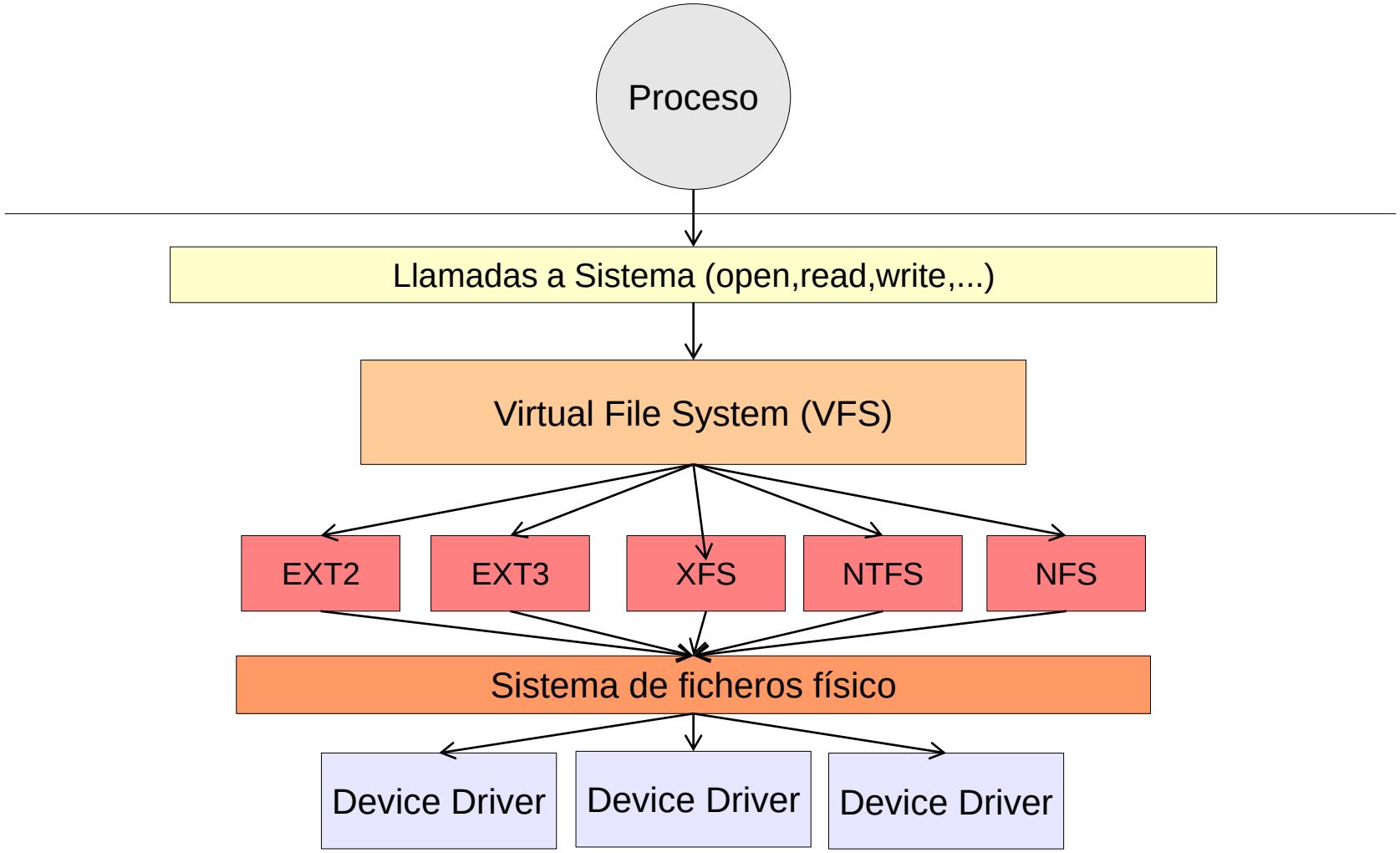
- Para poder acceder al SF de un dispositivo, primero se ha de **montar**
- **Montar** significa incluir el dispositivo (la partición) en el SF que maneja el SO, para que sea accesible a través de un directorio (*punto de montaje*)
- Existe un dispositivo raíz que se monta en el directorio "/" del sistema de ficheros
- Los demás dispositivos se pueden montar en cualquier directorio del SF.

VFS: Virtual File System

- Los SO soportan diferentes sistemas de ficheros:
  - Ext2, ext3, FAT, ISO9660, XFS, ReiserFS, NTFS, …
- Linux utiliza el VFS para acceder a todos de forma uniforme
- VFS proporciona un mecanismo orientado a objetos para acceder a todos estos sistemas de ficheros usando la misma interfaz de llamadas a sistema.
- Estructuras de datos en 2 niveles
  - Independientes del sistema de ficheros
    - Contiene descripciones de los sistemas soportados
    - Son consultadas/modificadas por las llamadas a sistema
      - sys_open(), sys_read(), …
  - Dependientes del sistema de ficheros
    - Estructuras internas para identificar ficheros, gestión espacio disco, …
    - Consultadas/modificadas por las rutinas específicas del VFS
      - sys_open_ext2(), sys_read_ext2(), …

Visión dinámica

VFS: Virtual File System



Visión dinámica

Estructuras Linux

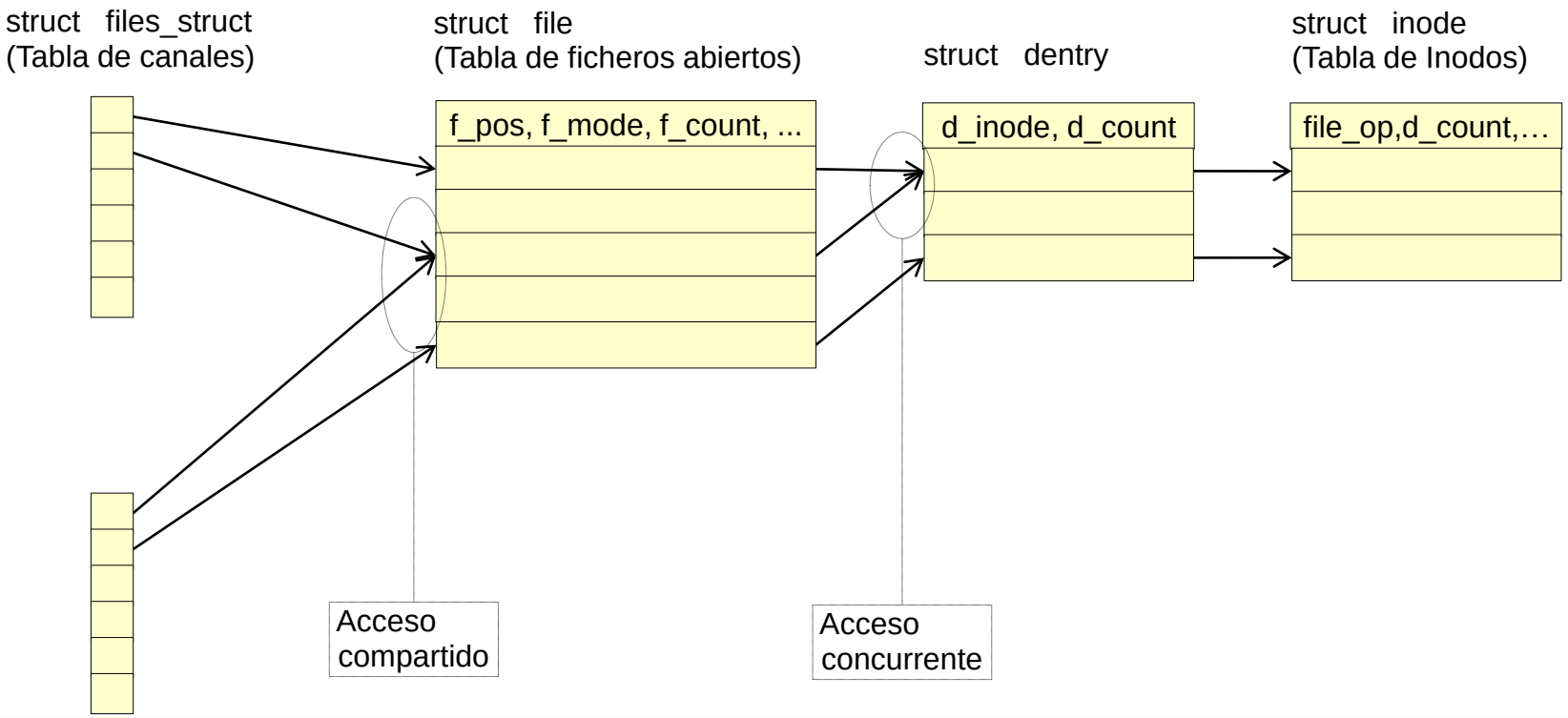
- Procesos (struct task_struct)
  - Cada proceso tiene una tabla de canales
- Tabla de canales (struct files_struct)
  - Cada canal apunta a un fichero abierto
  - Información sobre el dispositivo virtual
- Tabla de ficheros abiertos (struct file)
  - Tabla global a todo el sistema
  - Posición actual, modo acceso, …
  - Cada fichero abierto apunta a su entrada de directorio

Estructuras Linux: Optimización

- Para evitar accesos a disco guarda en memoria los datos/metadatos más usados
  - Superbloque
  - Cache de bloques (Buffer cache)
    - Hay una cache para cada Sistema de Ficheros
  - Cache de directorios (struct dentry)
    - Entradas de directorio usadas
  - Tabla de inodos (struct inode)
    - La estructura inode con sus operaciones

Estructuras Linux

- files_struct: http://lxr.linux.no/linux+v2.6.14/include/linux/file.h#L35
- file: http://lxr.linux.no/linux+v2.6.14/include/linux/fs.h#L576
- dentry: http://lxr.linux.no/linux+v2.6.14/include/linux/dcache.h#L83
- inode: http://lxr.linux.no/linux+v2.6.14/include/linux/fs.h#L422
- file_operations: http://lxr.linux.no/linux+v2.6.14/include/linux/fs.h#L946



Visión dinámica

Estructuras Linux

- Todas estas estructuras estan en C, pero orientadas a objetos
- Tienen datos + código para acceder a ellas
  - file_operations
  - dentry_operations
  - inode_operations

Exemple utilització a Linux

- Cas Read:
  - sys_read (…
    - vfs_read (file, buf, count, &pos)
      - Llama a file -> f_op -> read (file, buf, count, &pos)
- Cas open:
  - sys_open (filename, flags, mode)
    - filp_open (filename, flags, mode)
      - open_namei (filename, flags, mode, &nd) crea dentry (si no existia)
        - » dentry_open (nd.dentry, nd.mnt, flags);
          - Llama a file -> f_op -> open (inode, file)

Exemple utilització a Linux(3.19-4.0)

- Cas Read: http://lxr.linux.no/linux+v2.6.14/fs/read_write.c#L35
  - sys_read (… → SYSCALL_DEFINE3(read, …
    - vfs_read (file, buf, count, &pos)
      - Llama a file -> f_op -> read (file, buf, count, &pos)
- Cas open:
  - sys_open (filename, flags, mode)
    - filp_open (filename, flags, mode)
      - open_namei (filename, flags, mode, &nd) crea dentry (si no existia)
        - » dentry_open (nd.dentry, nd.mnt, flags);
          - Llama a file -> f_op -> open (inode, file)

Visión dinámica