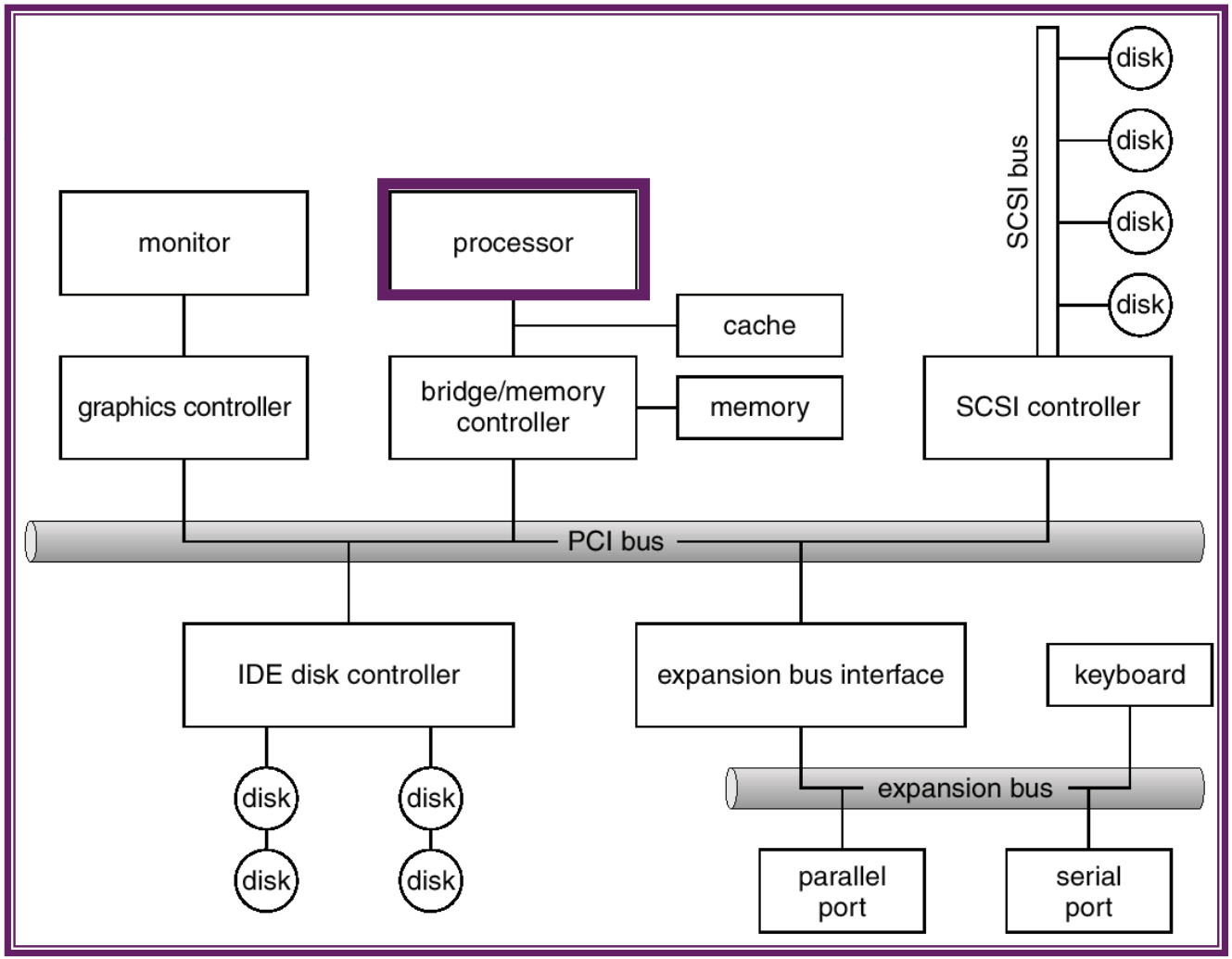
Sistemas operativos.
Entrada Salida

Diciembre de 2012

Contenidos I

Entrada/salida

A Typical PC Bus Structure



Entrada/Salida

- Los dispositivos de e/s permiten a la CPU relacionarse con el mundo exterior: teclados, pantallas, impresoras, discos ...
- La comunicación de la CPU con un elemento externo es similar a la comunicación con la memoria: se leen y escriben datos
- El comportamiento es distinto: los datos no siempre están disponibles, y el dispositivo puede no estar preparado para recibirlos (p.ej. teclado) (p.ej. impresora)
- Al ser distinto el comportamiento los métodos son distintos que para el acceso a memoria

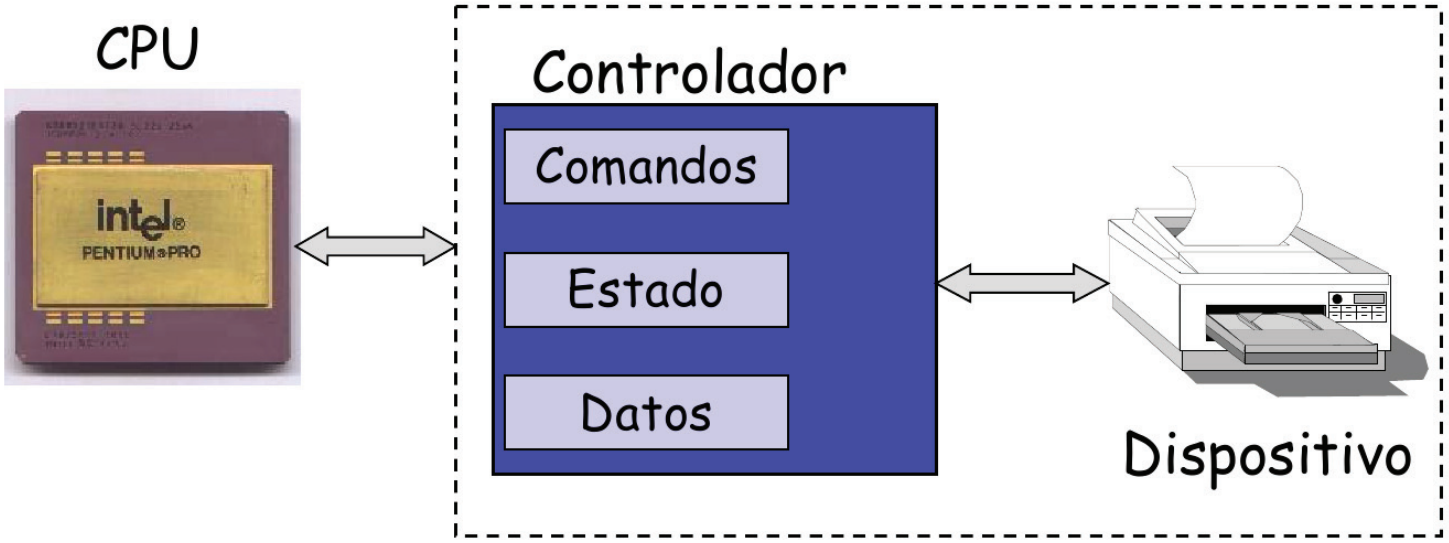
Estructura de un sistema de e/s

- En teoría los dispositivos de e/s se comunicariían con la CPU por los buses del sistema
- Dado que son muy heterogéneos sería muy costoso que la CPU los manejase directamente (P.ej.: S.O. debería conocer "detalles", CPU mucho más rápida que disp. e/s,...)
- Los dispositivos están conectados a una pieza de *hardware* llamada *controlador de dispositivo* (a veces controladora o adaptador)
- El controlador de dispositivo admite comandos *abstractos* de la CPU y se encarga de transmitrlos al dispositivo

  escribir bloque 2534 a disco
- Se libera a la CPU de tareas de muy bajo nivel

  escribir datos en cara X del plato Y, en cilindro Z, sector T

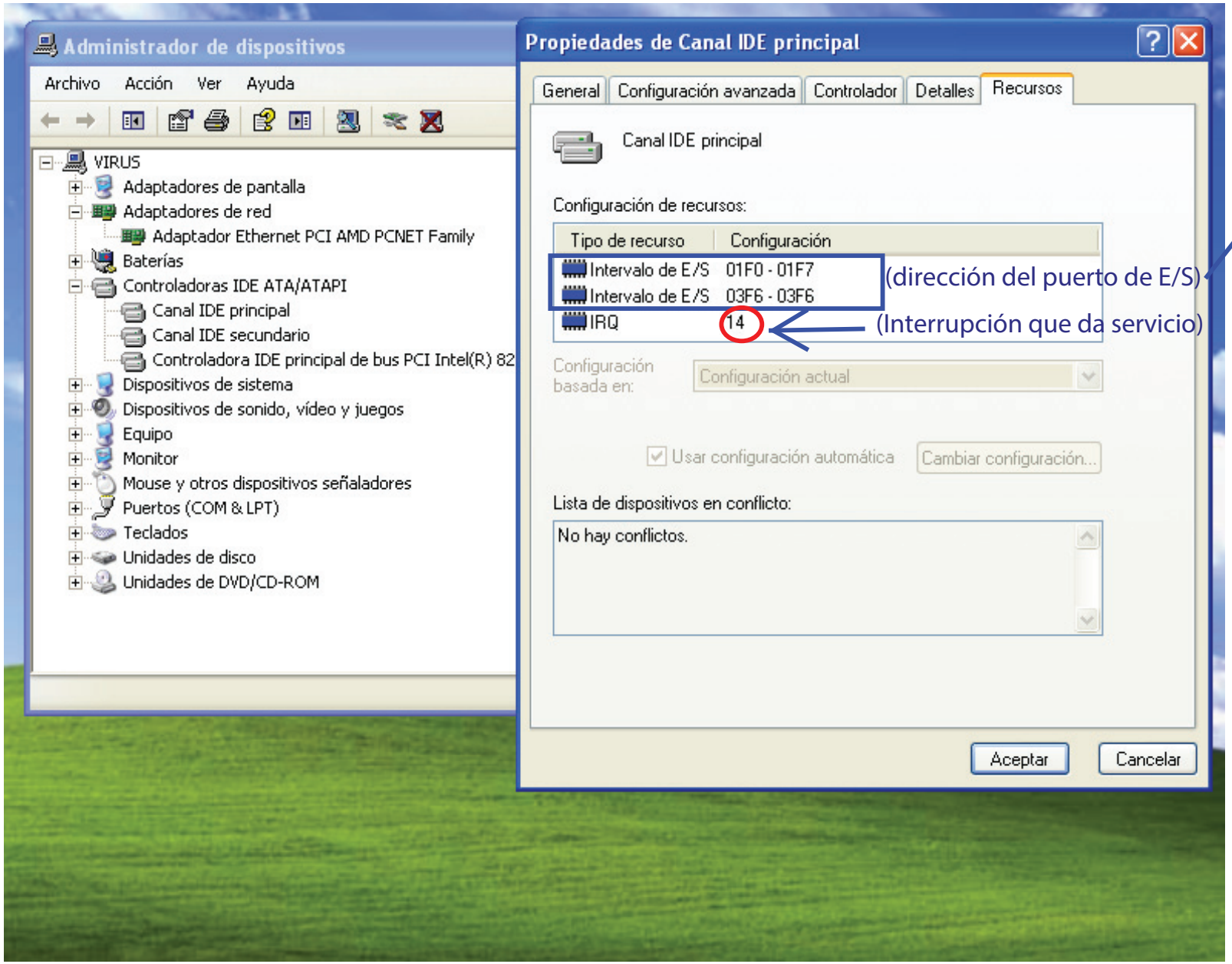
Estructura de un sistema de e/s



Estructura de un sistema de e/s

- El controlador de dispositivo actúa de interfaz entre la CPU y el dispositivo de E/S
- Cada controlador puede ocuparse de uno o varios dispositivos del mismo tipo (p.ej. controlador IDE --> Varios HDD, o un HDD y un CDRW,...)
- Los controladores se comunican con la CPU a través de unos registros o puertos. Generalmente incluyen
  - **Registros de control** Para enviar órdenes al dispositivo
  - **Registros de estado** Para obtener información del estado dispositivo o controlador, disponibilidad de datos ... (comando ya anterior ya completado)
  - **Registros de datos** Pueden ser de entrada, salida o bidireccionales
    - Típicamente son registros de 1 a 8 bytes de tamaño.
    - Algunos controladores tienen chips FIFO que permiten almacenar pequeñas ráfagas de datos (buffers) , mientras el host (CPU) no los puede ir recibir.

Comunicación con un dispositivo e/s



(lo veremos después) aquí se ubican los registros del controlador --> eL acceso a este rango de direcciones permite el acceso al controlador del canal IDE principal

Estructura de un sistema de e/s

- El controlador del dispositivo se encarga de
  - Coordinar el flujo de tráfico entre la CPU o memoria, y el dispositivo periférico
  - Comunicación con la CPU: decodificación de los comandos que vienen de la CPU, intercambio de los datos de E/S con la CPU, reconocimiento de la dirección del dispositivo (debe darse cuenta que los datos que vienen por los buses van dirigidos a este dispositivo y no a otro)
  - Comunicación con el Dispositivo: envío de comandos, intercambio de datos y recepción de la información de estado
  - Almacenamiento temporal de datos (buffer) puesto que las velocidades son muy distintas (CPU y dispositivo)
  - Detección de Errores

P.ej. La CPU es mucho más rápida --> el controlador almacena los datos que le va enviando la CPU y que debe ESCRIBIR en el dispositivo ... y los envía paso a paso al DISPOSITIVO LENTO

Estructura del software de e/s

- ¿**Cómo** puede una aplicación escribir en un HDD sin saber de que clase de disco se trata?

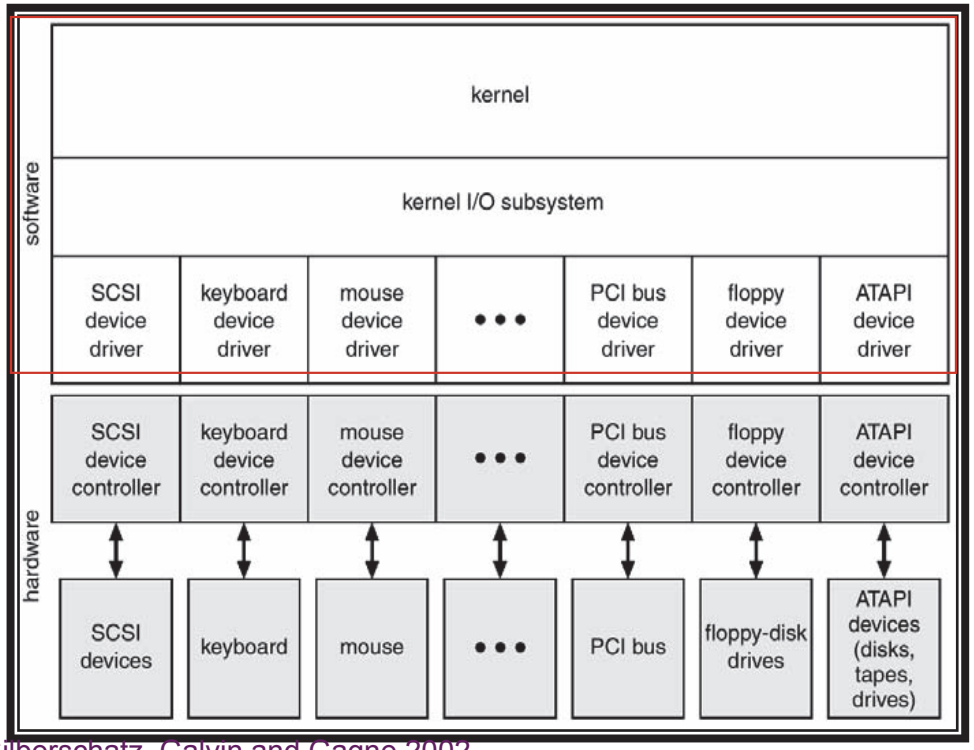
  ¿**Cómo** podemos añadir un nuevo dispositivo sin perturbar el funcionamiento de un computador?

ABSTRACCIÓN + ENCAPSULAMIENTO + CAPAS

- **Abstracción**:
  - Eliminar las diferencias entre los dispositivos al identificar unas pocas **clases generales**
  - Para cada clase se define una **interfaz** con unas **funciones estándar**
- **Encapsulamiento**:
  - Las diferencias reales se encapsulan mediante los *device drivers* que internamente se adaptan a las particularidades del dispositivo, pero que exportan una de las interfaces estándar (sus funciones).
- **Capas**

Estructura del software de e/s

- **Capas:**
  - El software de e/s se estructura en **capas**.



Silberschatz, Galvin and Gagne 2002
(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html)

- Las llamadas al subsistema de e/s observan el comportamiento de los dispositivos en unas cuantas clases genéricas
  → las aplicaciones no perciban las diferencias al operar con un hardware u otro

- Por otro lado, el subsistema de e/s (del S.O.) tiene por debajo la **capa del *device driver*** *(*controlador del dispositivo) que oculta las diferencias entre los distintos controladores de dispositivo

- El disponer de un subsistema de e/s INDEPENDIENTE del hardware,...
  - facilita la vida a los diseñadores del S.O. (no tienen que interactuar con INFINITOS dispositivos) y a los fabricantes de hardware:
    - O crean su hardware compatible con una interfaz (anfitrión/controlador) existente
    - O escriben/crean el *device driver* que establece la interfaz entre el nuevo dispositivo y los S.O's en los que vayan a operar

Estructura del software de e/s

- El software de entrada salida se estructura en capas

Modo usuario

- **Software de nivel de usuario**(user level software): Es el que proporciona la interfaz con el usuario. A este nivel se realizan tareas tales como formateo de datos

Modo kernel

-Bloques/Chars
-Buffers/Caché
-G. dispositivos:
nombrado, protec,
ctrl acceso.
-Planific. e/s

- **Software independiente de dispositivo**(device independent software). En esta capa tienen lugar todas las tareas de asignación de espacio, control de privilegios, uso de cache (si lo hubiere) ...
- **Manejador de dispositivo**(device driver): Es la capa de software que se comunica con el *controlador* de dispositivo y es la única pieza de software en todo el sistema operativo que lo hace. Para que un dispositivo se utilizable por el S.O. basta con disponer del *device driver*. Todo el conocimiento de los detalles del dispositivo está aqui.
- **manejador de interrupciones** (interrupt handler) Gestiona las interrupciones generadas por el dispositivo

Estructura del software de e/s

- Consideremos el siguiente ejemplo, y veamos lo que hace cada una de las capas

  `fprintf(fich,"%1.6f",x);`
- **Software de nivel de usuario**
  - Genera la cadena de caracteres adecuada: un dígito, un punto y 6 decimales a a partir del valor de x. (formateo de datos)
  - A partir del `FILE *` *fich* obtiene el descriptor de fichero (fd) para ser usado en la llamada al sistema *write* correspondiente. (preparación para llamada al sistema)
- **Software independiente de dispositivo**
  - Recibe un descriptor de fichero, una dirección de transferencia de memoria y una cantidad de bytes a transferir write (fd, buffer, N)

Estructura del software de e/s

(fd)

localizar bloque

- A partir del descriptor de fichero accede a la copia en memoria del inodo del fichero y, utilizando el offset de la tabla de ficheros, (t. fich abiertos) determina en qué bloque de disco (y en que *offset*) dentro del bloque) corresponde hacer dicha escritura

- traer bloque a cache
- escribir datos en dito bloque
- asignar espacio si necesario

- Comprueba que dicho bloque está en el cache (si no lo está lo trae) y transfiere los datos de la dirección de memoria de usuario al cache y marca el buffer del cache como modificado. Si la escritura hiciese necesario asignar otro bloque de datos al fichero, es aquí donde se haría

en algún momento: escribir bloque a disco

- Posteriormente (dependiendo del uso del cache) se invocariá al manejador de dispositivo para que escriba ese bloque a disco

Estructura del software de e/s

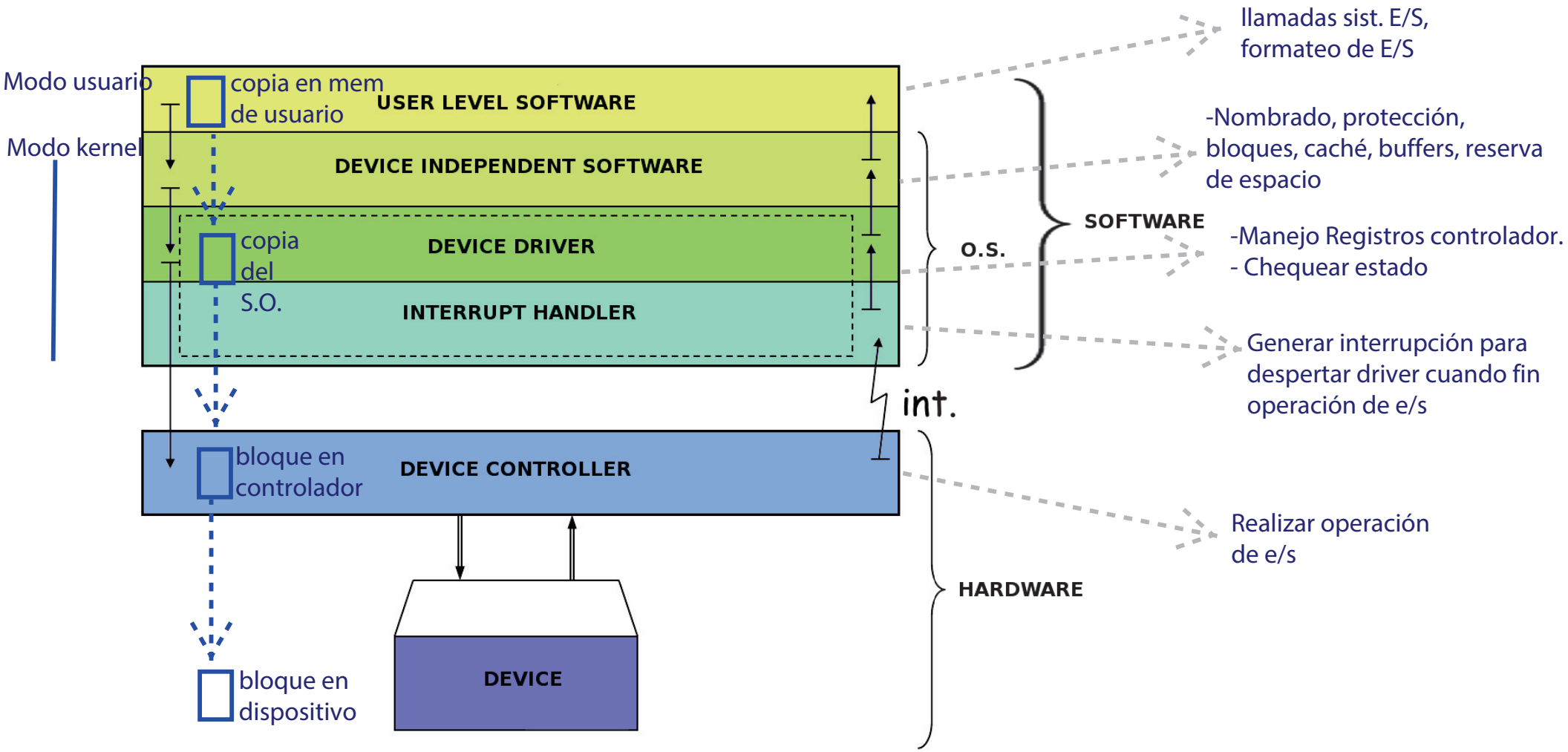
- **Manejador de dispositivo** (device driver)
  - Recibe la orden de escribir un bloque en determinado dispositivo
  - Determina a que dispositivo físico debe acceder.
  - Determina a que sector (o sectores) del disco (asi como sus cooredenadas de cilindro y cara) corresponde dicho bloque
  - Envia los comandos adecuados a los registros correspondientes del *controlador* (para que se realice la transferencia de datos)

un bloque lógico puede estar asociado a K sectores (K* 512bytes)

- **Manejador de interrupciones**
  - Cuando el dispositivo completa la solicitud, genera un interrupción que es gestionada por el manejador de interrupciones
  - Despierta al proceso que habia solicitado la e/s (en caso de que esta fuese síncrona
  - Libera los recursos ocupados por la operación e/s

el manejador mira qué e/s ha terminado (y generó esa interrup)

Estructura del software de e/s



Tipos de entrada salida

- Según el meétodo de comunicación de la CPU con los dispositivos distinguimos
  - e/s explícita
  - e/s mapaeada en memoria
- Según la percepción que tiene un proceso de como se realiza la e/s
  - e/s síncrona
  - e/s asíncrona

Comunicación de la CPU con los dispositivos

- **Dispositivos mapeados en memoria**
  - Los dispositivos aparecen en el espacio de direcciones. Se comparte el espacio de direcciones entre la memoria y los dispositivos
  - No suele plantear mucho problema porque espacio e/s $<<$ espacio de direcciones
  - Para acceder a los registros del controlador se hace con instrucciones de tipo *move* (como cualquier otra lectura/escritura en memoria)
  - motorola 68000 **MOVE, REG, ADDRESS**
- **Espacio e/s propio** **(distinto del espacio de direcciones normal)**
  - Se dispone de un rango de direcciones explicitas e/s. (este sistema se llama e/s explícita)
  - A veces hay instruciones especiales para acceder a este espacio e/s (IN y OUT en intel) o un registro de control que activa este espacio (powerpc)
    IN, REG, ADDRESS
    OUT, REG, ADDRESS
  - En estos sistemas tambien se pueden utilizar los dispositivos mapeados en memoria

E/S síncrona y asíncrona

SINCRONA = El proceso (percibe que) espera hasta que termina la operación de e/s

ASINCRONA = El proceso (percibe que) NO espera hasta que termina la operación de e/s, y ya lo avisarán

- Dado que la CPU es mucho más rápida que los dispositivos de e/s, una vez iniciada la e/s el S.O. asigna la CPU a otra tarea dejando al proceso que inici'n la tarea en espera (pasa a la cola de espera --> cuando termine pasará a la cola de listos)
  - El proceso percibe la e/s como síncrona, aunque es de hecho asíncrona
- Muchos sistemas operativos permiten tambien la e/s asíncrona el proceso inicia la e/s y continua su ejecución. El proceso es avisado por el S.O. cunado la operación de e/s se ha completado

  Avisarlo p.ej: (a) con una interrupción, (b) cambiando el valor de una variable en su espacio de usuario

Métodos de e/s

- Hay tes métodos de realizar la e/s
  - polling
  - interrupciones
  - dma

entrada salida por *polling*

- Es la forma más sencilla de realizar operaciones de E/S
- La sincronización se consigue al preguntarle la CPU (*poll*) al dispositivo si tiene un dato que entregar o, si es el caso, si está listo para recibir uno
- Se pierde tiempo en preguntar la dispositivo (se pierde más cuanto más a menudo se le pregunte) (espera activa)
- Es lento, e incluso podrían perderse datos si no se le pregunta muy a menudo

e/s polling (programada)

```
copy_from_user(buffer, p, count);      /* p is the kernel buffer */
for (i = 0; i < count; i++) {          /* loop on every character */
      while (*printer_status_reg != READY) ;   /* loop until ready */
      *printer_data_register = p[i];           /* output one character */
}
return_to_user( );
```

copia buffer (de usuario) a kernel space

(espera activa)

Figure 5-8. Writing a string to the printer using programmed I/O.
(Fuente:Tanembaum, Modern Operating Systems, 3rd ed.)

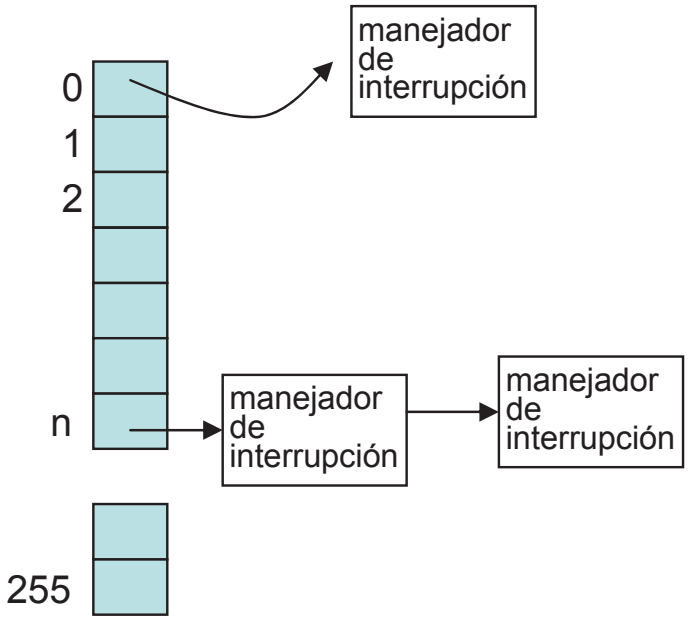
entrada salida por interrupción

- El dispositivo *avisa* a la CPU de que requiere antención mediante una *interrupción*
- Cuando llega la interrupción el S.O.
  - Salva el estado de lo que está haciendo
  - Trasfiere el control a la Rutina de Servicio de dicha interrupción
  - Ejecuta la Rutina de Servicio de dicha interrupción
  - Reanuda la ejecución donde fue interrupida
- Dichas rutinas están en memoria en unas direcciones apuntadas por los *vectores de interrupción*
- Es el S.O., en su proceso de inicialización, el que instala dichas rutinas
  - mira qué dispositivos están presentes,
  - e instala todos las rutinas necesarias (manejadores) en el vector de interrupciones.

Vector de interrupciones y rutinas

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19Ð31	(Intel reserved, do not use)
32Ð255	maskable interrupts

Vector interrupciones en Intel Pentium



Cada entrada del vector de interrupciones apunta al manejador (o cadena de manejadores) que debe usarse para atender a dicha interrupción

manejador = rutina

e/s interrupciones

llamada al sistema

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

copia buffer (de usuario) a kernel space

¿impresora libre?

copia primer dato a impresora (inicio de transferencia)

Poner proceso que quiere imprimir en cola de espera hasta que alguien llame a "unblock_user()"

(a)

manejador interrupciones

```
if (count== 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
     i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

//finalizado

//copia siguiente dato

(b)

Figure 5-9. Writing a string to the printer using interrupt-driven I/O. (a) Code executed at the time the print system call is made. (b) Interrupt service procedure for the printer.

(Fuente:Tanembaum, Modern Operating Systems, 3rd ed.)

entrada salida por dma

- En la e/s por polling y por interrupción es la CPU la que realiza la transferencia de datos cuando estos están listos (bien comprobando el registro de estado, o bien siendo avisada por el dispositivo mediante una interrupción) y atendiendo a dicha interrupción
- Esto presenta los siguientes inconvenientes
  - La velocidad de transferencia está limitada a la velocidad a la que la CPU puede mover los datos entre la CPU <----> Controlador|Dispositivo
  - Mientras dura una transferencia la CPU no puede dedicarse a otra cosa
- Estos probelmas no tienen gran importancia con dispositivos lentos y con pocos datos que tansferir, pero SI con dispositivos con mucho volumen de información, por ejemplo un disco duro
- Solución: Controlador de DMA (Direct Memory Access)

e/s Polling, la CPU...
- Inicia Transferencia de **cada dato** y
- Espera a que cada dato se transmita

e/s Interrupciones, la CPU...
- Inicia Transferencia de **cada dato** y
- Ejecuta Rutina Tratamient Interrupción

e/s DMA, la CPU...
- Inicia e/s (**conjunto de datos**)
- Ejecuta 1 Rutina Trat. Interrup final

e/s DMA

llamada al sistema

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

copia buffer (de usuario) a kernel space

inicializar controlador DMA

Poner proceso que quiere imprimir en cola de espera hasta que alguien llame a "unblock_user()"

(a)

manejador interrupcion

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

//finalizado

(b)

(sólo recibe interrupción al final)

Figure 5-10. Printing a string using DMA. (a) Code executed when the print system call is made. (b) Interrupt service procedure.

(Fuente:Tanembaum, Modern Operating Systems, 3rd ed.)

entrada salida por dma

- El controlador de DMA se encarga de la transferencia directa de datos, sin intervención de la CPU.
- (2) El controlador de DMA suministra las señales de dirección y todas las señales de control del bus (al dispositivo)
- (1) La CPU debe suministrar al controlador de DMA el tipo de operación (lectura o escritura), dirección de transferencia de datos y cantidad de bytes a transferir (dirección de memoria donde están/poner los datos)
- (3) Cuando la transferencia se completa le DMA avisa a la CPU por medio de una interrupción
- El bus debe compartirse entre la CPU y el DMA. Varios modos de compartición
  - DMA bodo bloque (*burst*)
  - DMA por robo de ciclos
  - Bus transparente

entrada salida por dma

SÓLO ESPERA si necesita el bus, pero puede seguir leyendo instrucciones y datos de la memoria caché (L1, L2, L3)

- **DMA bodo bloque (burst)**
  - Una vez que el DMA obtiene el control del bus se transfiere un bloque y la CPU queda en espera hasta que la transferencia termina
  - Es el método más rápido
  - Es el método que se utiliza con los dispositivos de almacenamiento secundario, como discos

ráfaga

PROS:
Como "adquirir acceso al bus" lleva algo de tiempo
--> Burst tiene la ventaja de que con una única "adquisición" se pueden enviar varias palabras (1 bloque completo)
.
CONTRAS:
La CPU podría tener que esperar si necesita acceder a un dato que no está en la caché (ir a memoria)

entrada salida por dma

- **DMA por robo de ciclos**
  - Cada vez que obtiene el control del bus, el DMA transfiere una palabra (p.ej: 4-8 bytes), devolviéndole el control del bus a la CPU
  - La trasnferencia ser realiza con una serie de ciclos de DMA (robados) intercalados con ciclos de CPU
- **Bus transparente**
  - El DMA sólo utiliza el bus cuando la CPU no lo usa (p.e., durante la decodificación de una instrucción o utilizando la ALU)
  - SI hay memoria cache, la CPU puede acceder a código y datos en el cache mientras el DMA realiza las transferencias

Discos

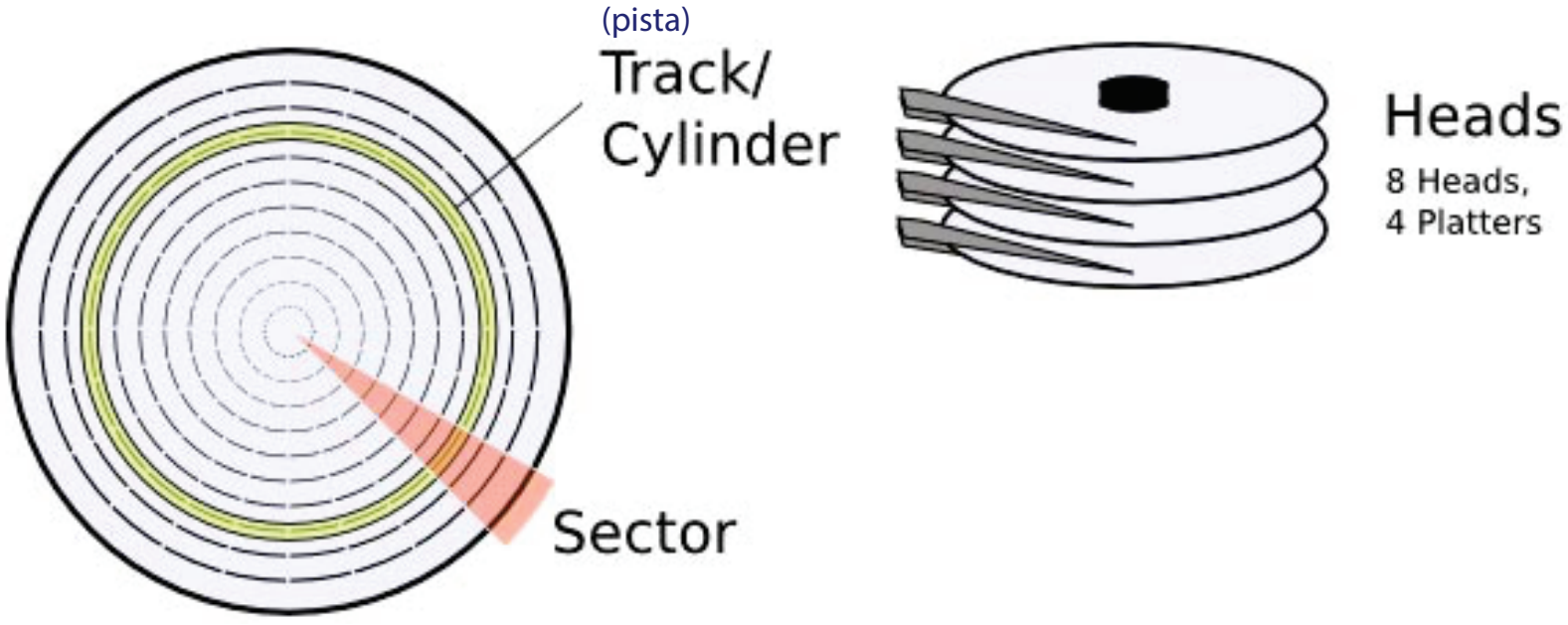
- Un disco esta formado por un conjunto de platos que giran solidariamente.
- Cada uno de las superficies de dichos platos se denomina cara
- Cada cara esta compuesta por una serie de coronas circulares concéntricas denominada pista. Al conjunto formado por la misma pista en distintas cara se le denomina cilindro
- Cada cilindro está formado por una serie de *sectores*
  - El sector es la unidad de e/s elemental de un disco duro.
  - Tí picamente cada sector tiene 512 byte-s

Discos

- Cada sector queda perfectamente especificado por sus tres coordenadas: cilindro cara y sector.(tipicamente la numeración de las caras y cilindros comienza en 0 y la de los sectores en 1)
- Solo el manejador de dispositivo conoce las características físicas del disco. El S.O. trata el disco como una sucesión de bloques lógicos. P.e. un disquete de 1.4 Mbytes tine 80 cilindros, 2 caras y 18 sectores por pista. (2880 sectores de 512 bytes). SI lo formateamos con bloques de 4k el S.O. lo considerará como una sucesion de 360 bloques de 4K (cada bloque) tiene 8 sectores
- La asignación de espacio (y la contabilidad del espacio libre) se hace por bloques, no por sectores

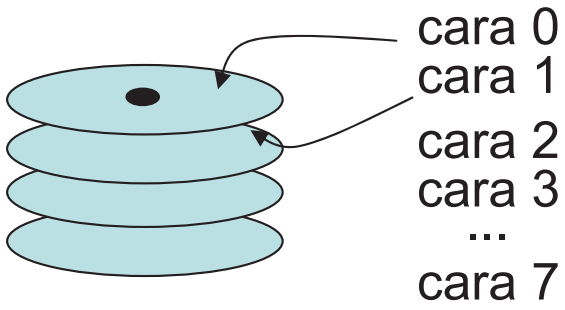
DISCO --> SECTORES (512b) S.O. --> BLOQUES LÓGICOS (512*k bytes)

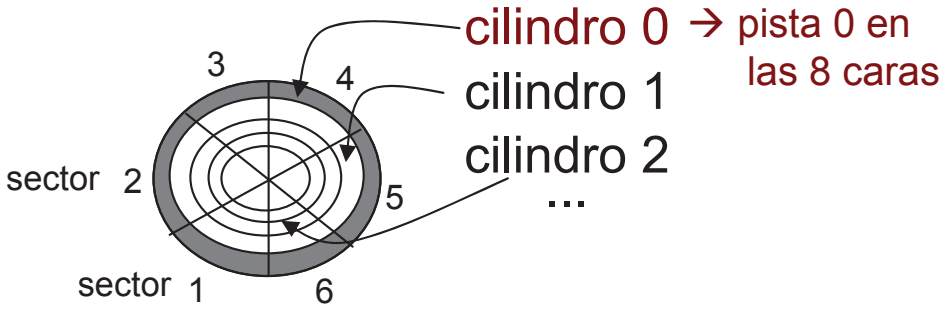
Estructura de un disco



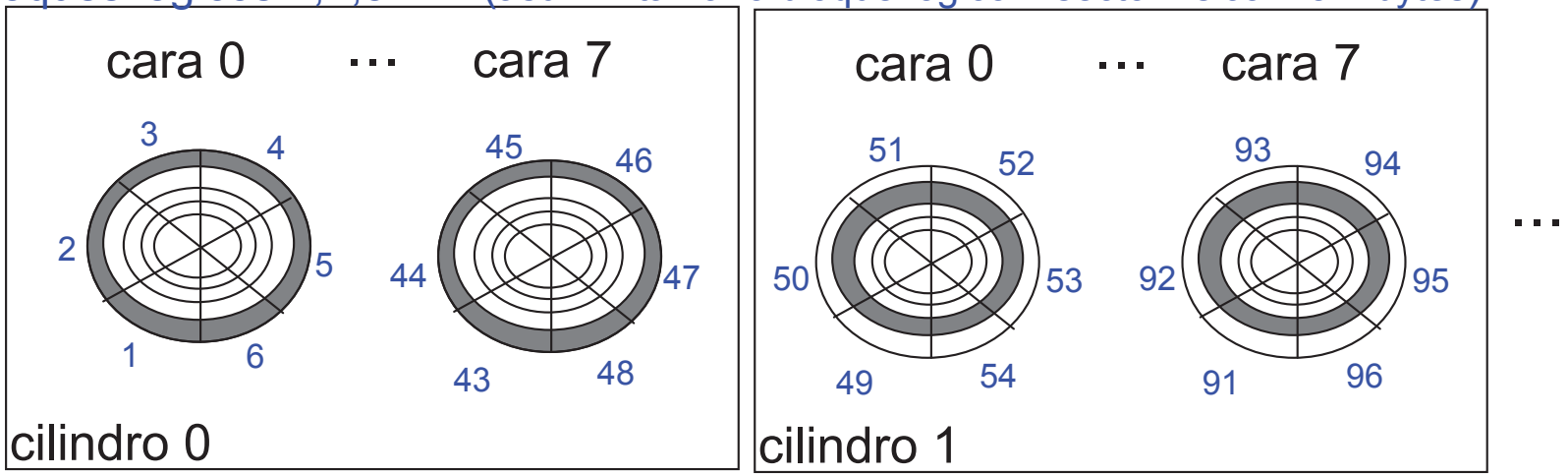
Estructura de un disco

- Bloques lógicos 1,2,3..N (512bytes)→ correspondencia a sectores físicos (cilindro, cara, sector).





Bloques lógicos 1,2,3..N (asumir: tamaño bloque lógico = sector físico = 512bytes)



Discos

- En los discos de más capacidad que un disquete, cada disco se divide en zonas que se pueden tratar como sistemas deficheros independientes.
  P.ej. particiones
- En el primer sector del disco (cara 0, cilindro 0, sector 1) está una tabla que nos indica qué *zonas* existen en el disco.
- El nombre y el formato de dicha tabla varía de unos sistemas a otros: *tabla de particiones*, con cuatro entradas en sistemas windows y linux; *disklabel* con 8 o 16 enytradas en sistemas BSD ...

Planificación de discos

- En un sistema donde se generan multiples solicitudes de entrada salida a discos, estas pueden planificarse. En general se intenta mejorar los tiempos de búsqueda. T= tiempo búsqueda + tiempo rotación + tiempo transferencia
  mover cabeza a cilindro
- Lo algoritmos de planificación de búsquedas
  - **FCFS** o FIFO: Es igual que no tener planificación. Las solicitudes se atienden en el mismo orden que llegan
  - **SSTF** Shortest Seek Time First. Se antiende primero a las solicitudes con menor tiempo de búsqueda desde la posición actual. Mejora el tiempo de bśqueda promedio, pero no produce tiempos homogeéneos ni predecibles. Riesgo de inanición

Planificación de discos

- **SCAN** (Tambien denominado algoritmo del ascensor: se atiende a la solicitud mas proxíma pero solo en un sentido. Como las solicitudes de la zona central del disco reciben más atención que las de lo extremos:
  - **C-SCAN** Igual que el SCAN pero al llegar a un extremo del disco se vuelve al principio sin atender ningua solicitud (como un ascensor que sólo para en las subidas) (El tiempo promedio de acceso a cada cilindro es el mismo)
- **SCAN N pasos o C-SCAN N pasos**. Como el scan, (o c-scan) salvo que en cada *viaje* se atienden como mámimo N solicitudes en cada recorrido

C-LOOK (variante de C-SCAN, pero que (si no es necesario) no llega a los extremos al subir, pues se queda en el último cilindro "pedido", ni al iniciar la subida, pues comienza NO desde el cilindro "0" SINO desde el primero "pendiente".)

Planificación de discos: ejemplos

- Algoritmos de planificación:

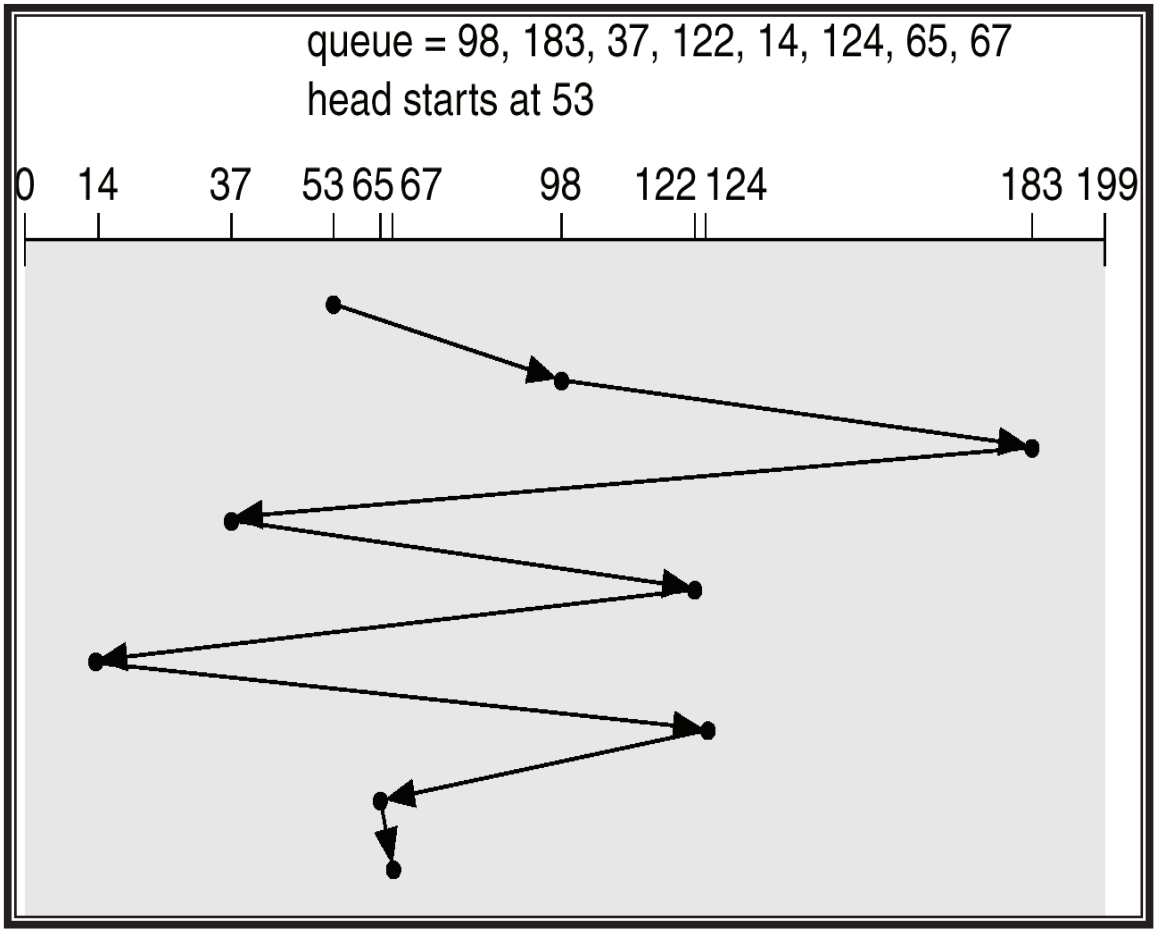
  FCFS, SSTF, [c]SCAN[n], C-LOOK

  (tratan de mejorar del *seek-time* +-= *distancia movimiento cabeza*):

- Asúmase que:
  - hay sólo 200 cilindros [0..199]
  - la cabeza está situada sobre el cilindro 53, y
  - que en la cola de acceso a disco se presenta la siguiente secuencia de peticiones de acceso a bloques que están en los cilindros:

    < 98, 183, 37, 122, 14, 124, 65, 67 >

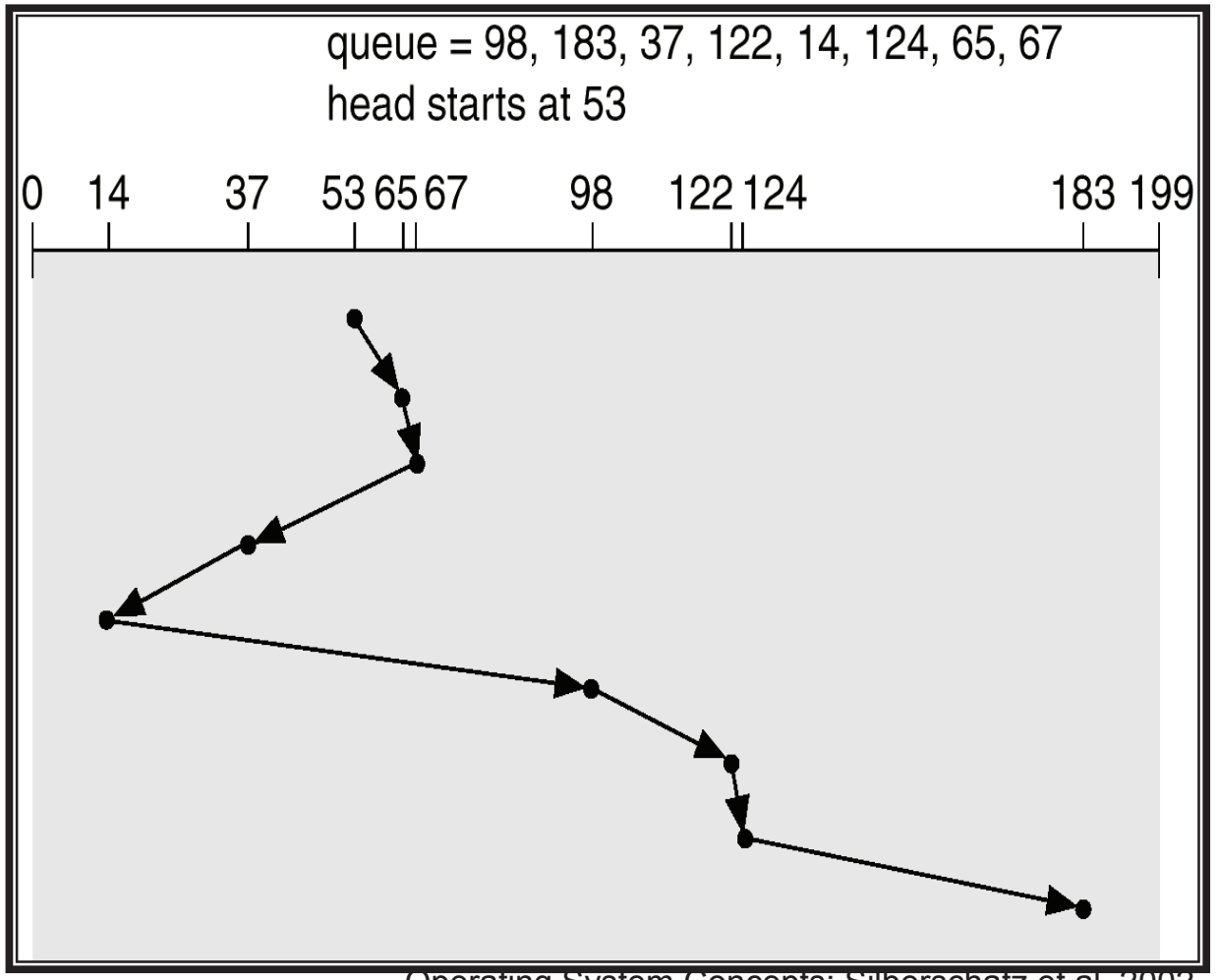
FCFS (FIFO)



Operating System Concepts: Silberschatz et al, 2002
(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html)

Número de movimientos: 640 cilindros

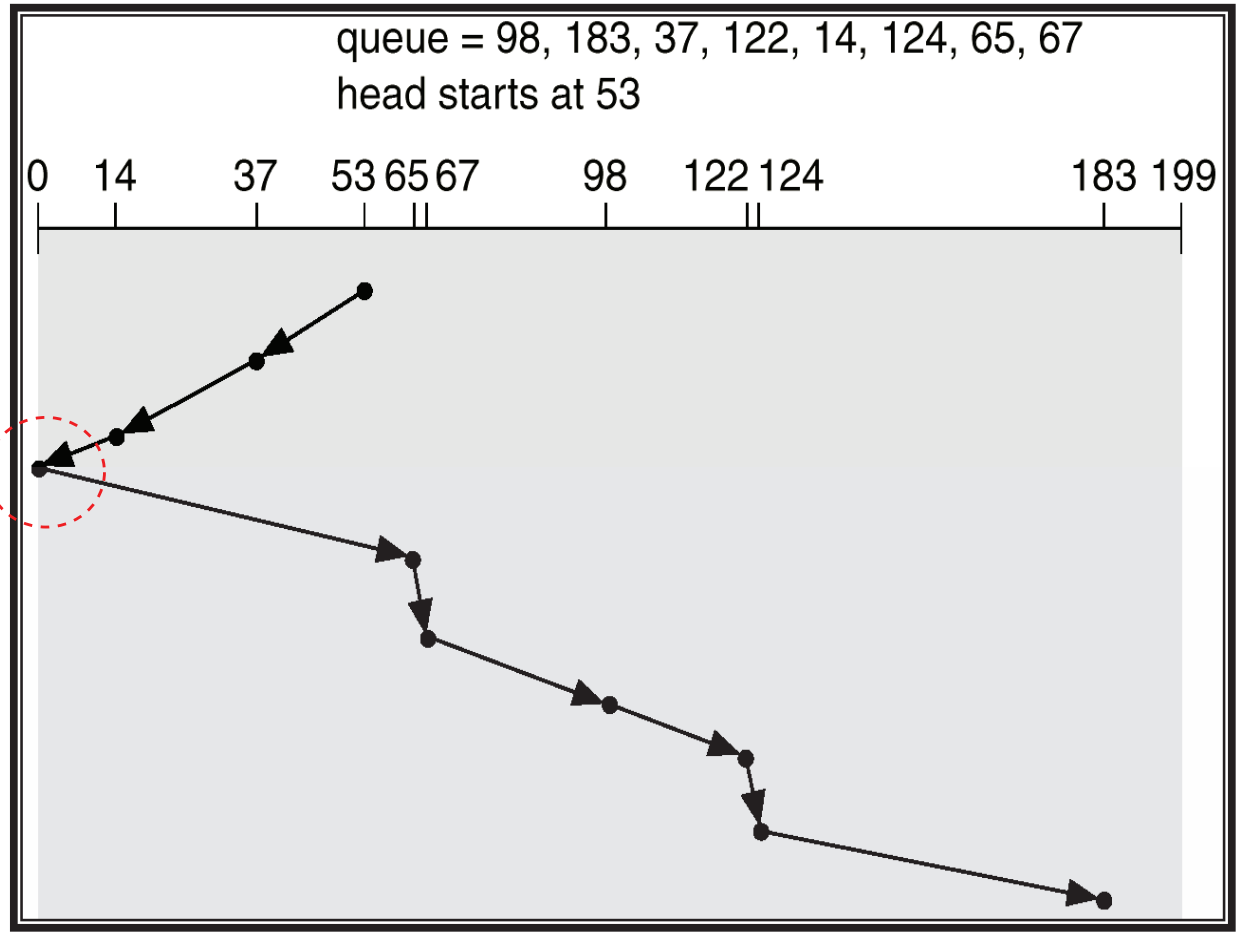
SSTF



Operating System Concepts: Silberschatz et al, 2002
(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html)

Número de movimientos: 236 cilindros, pero podría crear “inanición"!!!

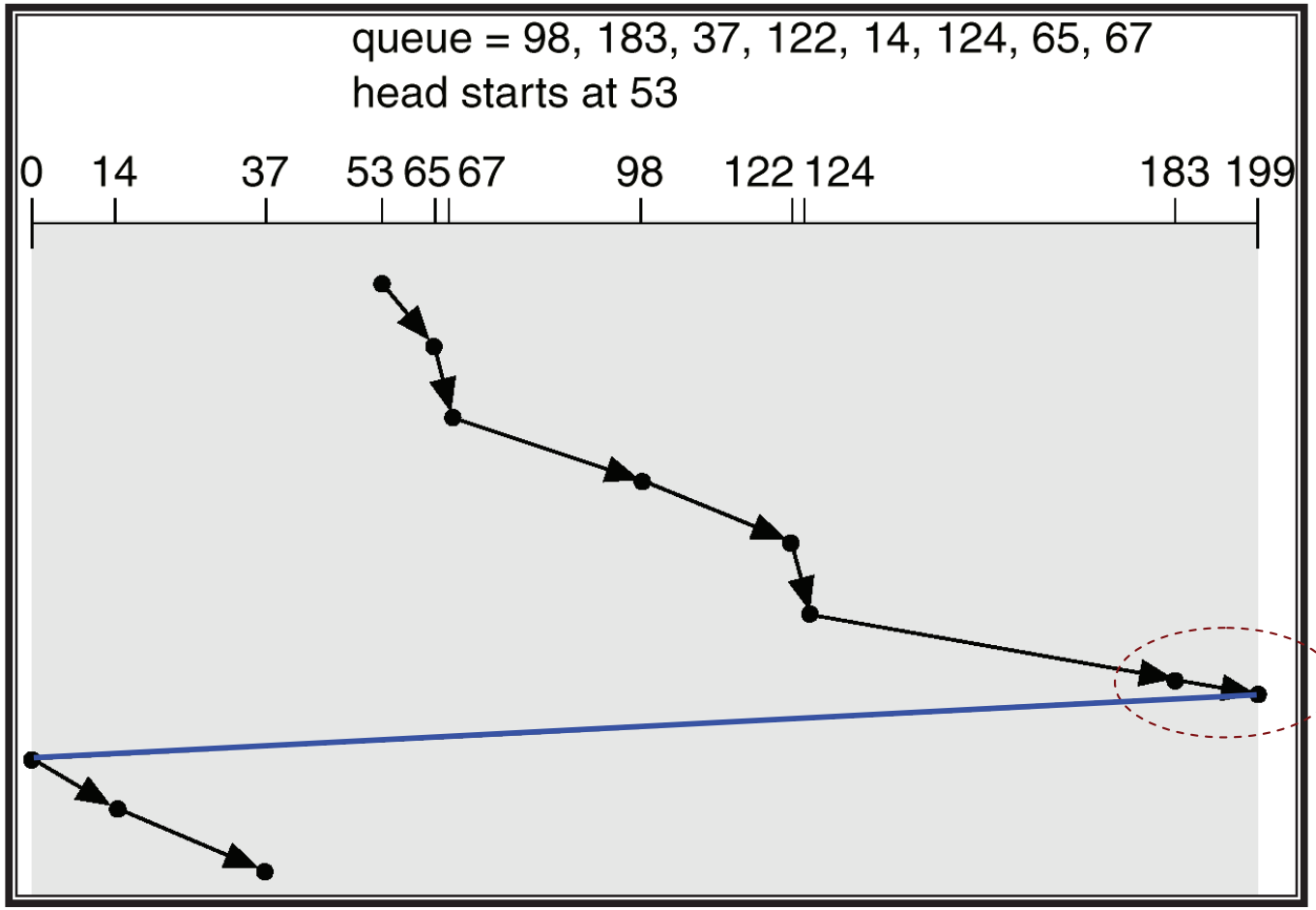
SCAN (algoritmo del ascensor)



Operating System Concepts: Silberschatz et al, 2002
(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html)

Inicialmente bajando (53→0) → Número de movimientos : 208 cilindros

C-SCAN

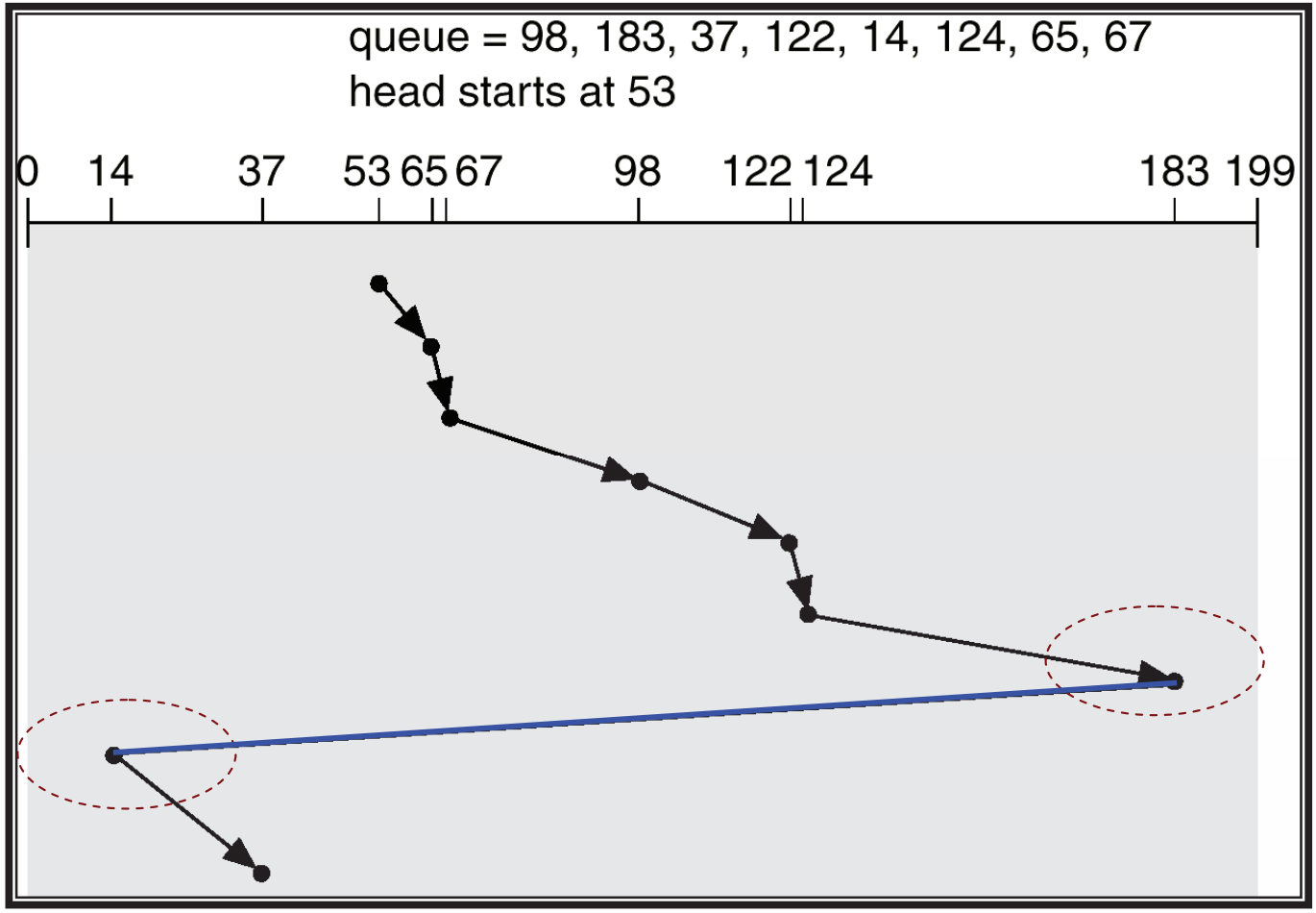


Operating System Concepts: Silberschatz et al, 2002
(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html)

Inicialmente subiendo (53+) → Número de movimientos : 199-53 + **200** + 37 cilindros

C-LOOK (versión de C-SCAN)

Versión de C-scan que no llega al final en cada subida sino a la última petición. Y que comienza la siguiente "subida" NO desde el cilindro 0, sino desde el primero "pendiente"



Operating System Concepts: Silberschatz et al, 2002
(http://www.wiley.com//college/silberschatz6e/0471417432/slides/slides.html)

Inicialmente subiendo (53+) → Número de movimientos: 183-53 + **(183-14)** + (37-14) cilindros

Planificación de discos

- **CFQ** Complete Fair Queuing. Asigna distintas colas para las solicitudes de acceso a disco de los distintos procesos, y a cada cola le asigna *quantos* de tiempo. La longitud del *quanto* y el número de solicitudes que pueden procesarse de una cola dependen de la *prioridad i/o* del proceso. En linux, el comando `ionice` nos permite modificar la *prioridad i/o* de un proceso.

```
ionice(1)                                                    ionice(1)

NAME
       ionice - get/set program io scheduling class and priority

SYNOPSIS
       ionice [[-c class] [-n classdata] [-t]] -p PID [PID]...
       ionice [-c class] [-n classdata] [-t] COMMAND [ARG]...
```

- Tambien pueden intentar mejorarse las latencias (planificación rotacional), aunque en sistemas modernos, es normal que la controladora lea pistas enteras y solamente se transfieran los sectores necesarios

ionice - set or get process I/O scheduling class and priority

SYNOPSIS

ionice [-c class] [-n level] [-t] -p PID...

ionice [-c class] [-n level] [-t] command [argument...]

DESCRIPTION

- This program sets or gets the I/O scheduling class and priority for a program. If no arguments or just -p is given, ionice will query the current I/O scheduling class and priority for that process.

- When command is given, ionice will run this command with the given arguments. If no class is specified, then command will be executed with the "best-effort" scheduling class. The default priority level is 4.

  -c 2 -n 2

- As of this writing, a process can be in one of three scheduling classes:

  – **Idle (-c 3)**: A program running with idle I/O priority will only get disk time when no other program has asked for disk I/O for a defined grace period.
  The impact of an idle I/O process on normal system activity should be zero. This scheduling class does not take a priority argument. Presently, this scheduling class is permitted for an ordinary user (since kernel 2.6.25).

  – **Best-effort (-c 2)**: This is the effective scheduling class for any process that has not asked for a specific I/O priority. This class takes a priority argument from 0-7, with a lower number being higher priority. Programs running at the same best-effort priority are served in a round-robin fashion.

    Note that before kernel 2.6.26 a process that has not asked for an I/O priority formally uses "none" as scheduling class, but the I/O scheduler will treat such processes as if it were in the best-effort class. The priority within the best-effort class will be dynamically derived from the CPU nice level of the process: io_priority = (cpu_nice + 20) / 5.

    For kernels after 2.6.26 with the CFQ I/O scheduler, a process that has not asked for an I/O priority inherits its CPU scheduling class. The I/O priority is derived from the CPU nice level of the process (same as before kernel 2.6.26).

  – **Realtime (-c 1)**: The RT scheduling class is given first access to the disk, regardless of what else is going on in the system. Thus the RT class needs to be used with some care, as it can starve other processes. As with the best-effort class, 8 priority levels are defined denoting how big a time slice a given process will receive on each scheduling window. This scheduling class is not permitted for an ordinary (i.e., non-root) user.

OPTIONS

- -c, --class class : Specify the name or number of the scheduling class to use:
  0 for none, 1 for realtime, 2 for best-effort, 3 for idle.

- -n, --classdata level: Specify the scheduling class data. This only has an effect if the class accepts an argument. For realtime and best-effort, 0-7 are valid data (priority levels).

- -p, --pid PID... : Specify the process IDs of running processes for which to get or set the scheduling parameters.

- -t, --ignore :Ignore failure to set the requested priority. If command was specified, run it even in case it was not possible to set the desired scheduling priority, which can happen due to insufficient privileges or an old kernel version.

EXAMPLES

- # ionice -c 3 -p 89
  Sets process with PID 89 as an idle I/O process.

- # ionice -c 2 -n 0 bash
  Runs 'bash' as a best-effort program with highest priority.

- # ionice -p 89 91
  Prints the class and priority of the processes with PID 89 and 91.

Entrada/salida en UNIX

Dispositivos e/s en UNIX

- En UNIX los dispositivos aparecen como un fichero (tipicamente en el directorio /dev). En agunos sistemas (p.e. solaris) este fichero es un enlece simbólico a donde está realmente el fichero del dispositivo.
- Tienen asignado un inodo, en donde además de indicar si es un dispositivo de bloque o de carácter, figuran dos números (*major number* y *minor number*) que indican que manejador de despositivo se utiliza para acceder a él y que unidad dentro de las manejadas por dicho manejador

```
# ls -li /dev/sda /dev/sda1 /dev/sda2 /dev/audio
4232 crw-rw---T+ 1 root audio 14, 4 Jan  7 18:51 /dev/audio
3106 brw-rw---T  1 root disk   8, 0 Jan  7 18:51 /dev/sda
3110 brw-rw---T  1 root disk   8, 1 Jan  7 18:51 /dev/sda1
3111 brw-rw---T  1 root disk   8, 2 Jan  7 18:51 /dev/sda2
```

- Para acceder a los dispositivos se pueden utilizar las mismas llamadas que para el acceso a los ficheros (*open*, *read*, *write*) siempre que el proceso que lo haga tenga los provilegios adecuados

ls -li /dev/...

Shows permissions (brw-rw----), owner (root), group (disk), major device number (8), minor device number (0), date (mars 9 - french, no year), hour (07:56) and device name (guess :-).

When accessing a device file, ...

... the major number selects which device driver is being called to perform the input/output operation. This call is being done with the minor number as a parameter and it is entirely up to the driver how the minor number is being interpreted. The driver documentation usually describes how the driver uses minor numbers.

```
~$ ls -li /dev/sd*
  6922 brw-rw---- 1 root disk 8,  0 oct  8 12:12 /dev/sda
  6923 brw-rw---- 1 root disk 8,  1 oct  8 12:12 /dev/sda1
~
  7211 brw-rw---- 1 root disk 8, 16 oct  8 12:12 /dev/sdb
  6924 brw-rw---- 1 root disk 8, 17 oct  8 12:12 /dev/sdb1
  6925 brw-rw---- 1 root disk 8, 18 oct  8 12:12 /dev/sdb2
  6926 brw-rw---- 1 root disk 8, 19 oct  8 12:12 /dev/sdb3
~
  1306 brw-rw---- 1 root disk 8, 32 oct  8 12:12 /dev/sdc
  1307 brw-rw---- 1 root disk 8, 33 oct  8 12:12 /dev/sdc1
  1308 brw-rw---- 1 root disk 8, 34 oct  8 12:12 /dev/sdc2
  1309 brw-rw---- 1 root disk 8, 35 oct  8 12:12 /dev/sdc3
~
  8221 brw-rw---- 1 root cdrom  11,  0 oct  8 12:12 /dev/sr0
  1044 crw-rw---- 1 root tty        4,  1 oct  8 12:12 /dev/tty1
```

Minor number:
16 +0 --> sdb
16 +1 --> sdb1
16 +2 --> sdb2

Dispositivos e/s en UNIX

- Para acceder (leer o escribir) a los dispositivos se pueden utilizar las mismas llamadas que para el acceso a los ficheros (*open*, *read*, *write*...) siempre que el proceso que lo haga tenga los provilegios adecuados
- Tambien se puede acceder a las características (y demas funcionalidades) de los dispositivos mediante la llamada `ioctl`

```
IOCTL(2)                    Linux Programmer's Manual                    IOCTL(2)

NAME
       ioctl - control device

SYNOPSIS
       #include <sys/ioctl.h>

       int ioctl(int d, int request, ...);

DESCRIPTION
       The  ioctl()  function  manipulates the underlying device parameters of
       special files.  In particular, many operating characteristics of  char
       acter  special  files  (e.g., terminals) may be controlled with ioctl()
```

... incluyendo funciones de control que normalmente no están disponibles a través de las funciones de librería:
- expulsar un CDROM, cambiar la velocidad de conexión de un módem (baud),...

Ej: Uso de ioctl para manejar el cdrom

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/cdrom.h>
#include <time.h>

void cerracd(int fd) {
    if (-1 == ioctl(fd, CDROMCLOSETRAY, 0))
        perror("\nioctl set, operation FAILED");
    else printf("\nioctl set, CDROMCLOSETRAY OK!!");
}

void abrecd(int fd){
    if (-1 == ioctl(fd, CDROMEJECT, 0))
        perror("\nioctl set, operation FAILED");
    else  printf("\nioctl set, CDROMEJECT OK!!");
}

int main(int argc, char *argv[])
{   char *fichero_cdrom = "/dev/sr0";
    if (argc ==2) fichero_cdrom = argv[1];
    printf("\n Abriré dispositivo de CDR = %s",fichero_cdrom);
    int i, fd = open(fichero_cdrom, O_RDONLY | O_NONBLOCK);
    if (fd == -1){ perror("ERROR open"); return -1;}
    abrecd(fd);
    fflush(stdout);sleep (10);
    //cerracd(fd);
    return 0;
}
```

```
$ ./ioctl_cd
  Abriré dispositivo de CDR = /dev/sr0
  ioctl set, CDROMEJECT OK!!
  ioctl set, CDROMCLOSETRAY OK!!
```

$ **man ioctl_list** → ver todas IO_calls

→ /usr/include/linux/cdrom.h

```
use open("/dev/cdrom",
            O_RDONLY | O_NONBLOCK);
```

```c
#define CDROMPAUSE
#define CDROMRESUME
#define CDROMPLAYMSF
#define CDROMPLAYTRKIND
#define CDROMREADTOCHDR
#define CDROMREADTOCENTRY
#define CDROMSTOP
#define CDROMSTART
#define CDROMEJECT
#define CDROMVOLCTRL
#define CDROMSUBCHNL
#define CDROMREADMODE2
#define CDROMREADMODE1
#define CDROMREADAUDIO
#define CDROMEJECT_SW
#define CDROMMULTISESSION
#define CDROM_GET_MCN
#define CDROMRESET
#define CDROMVOLREAD
#define CDROMREADRAW

#define CDROMCLOSETRAY
#define CDROM_SET_OPTIONS
#define CDROM_CLEAR_OPTIONS
#define CDROM_SELECT_SPEED
#define CDROM_SELECT_DISC
#define CDROM_MEDIA_CHANGED
#define CDROM_DRIVE_STATUS
#define CDROM_DISC_STATUS
#define CDROM_CHANGER_NSLOTS
#define CDROM_LOCKDOOR
#define CDROM_DEBUG
#define CDROM_GET_CAPABILITY
```

Entrada/salida en UNIX

Llamadas para e/s en UNIX

- En UNIX la e/s es sin formato: se escriben y se leen bytes; las llamadas reciben una direcci'on de memoria y el n'umero de bytes a leere (o escribir)
- **open**

```
int open(const char *path, int flags, mode_t mode)
```

  - Recibe el nombre del fichero, el modo de apertura (*flags*, que puede tomat los valores O_RDONLY, O_WRONLY, O_RDWR . . .), y los permisos (*mode*,en caso de que se cree el fichero)
  - Devuelve un entero, *descriptor de fichero*,(-1 en caso de error) que es el que se utilizará en subsecuentes llamadas a *read y write*

O_APPEND, O_TRUNC, O_CREAT, O_EXCL

0777 ... S_IRUSR, S_IWUSR,...

- **close**
  - `int close(int fd)` cierra un descriptor de fichero obtenido con *open*

Llamadas para e/s en UNIX

- **read y write**

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

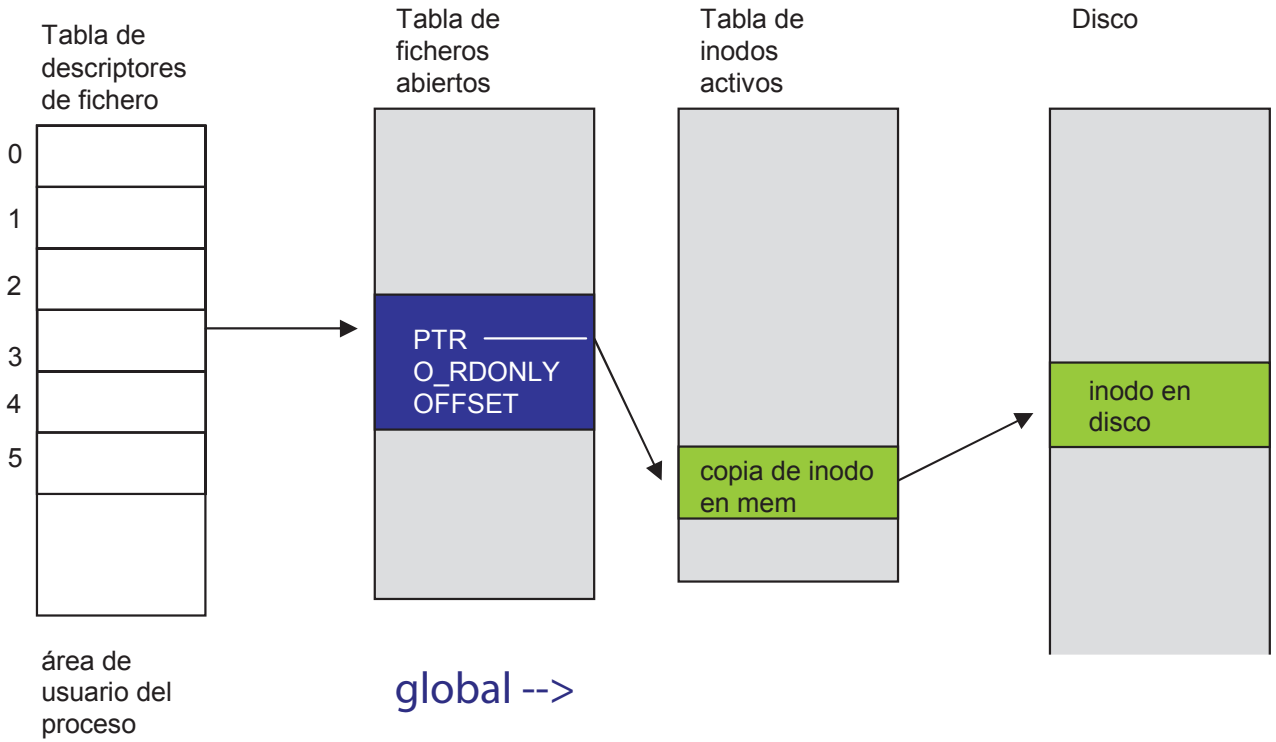
  - Reciben el descriptor de fichero, una dirección de transferencia y el número de bytes para e/s
  - Devuelve en número de bytes que han sido leidos o escritos
  - La lectura (o escritura) es a partir de la posición u *offset* actual del fichero, que es actualizada tras estas llamadas.
  - La posición u *offset* actual del fichero, pude ser modificada mediante la llamada `off_t lseek(int fd, off_t offset, int whence)`

offset bytes
a partir de "whence"

SEEK_SET (inicio)
SEEK_CUR (actual)
SEEK_END (tras final: ocos se write !!)

Ficheros: open, close,read,write,seek

```
int fd;
fd = open(nombre_de_archivo, .....);
       ...
read(fd, .....);
write(fd, ....);
close(fd);
```



Llamadas para e/s en UNIX

- **pread y pwrite**

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset)
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset)
```

  - Análogas a las llamadas *read* y *write* salvo que se suministra la posición de de lectura
  - Estas llamadas no actualizan la posición u *offset* actual del fichero

Llamadas para e/s en UNIX

- **llamadas para e/s a posiciones de memoria no contiguas**

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt)
ssize_t writev(int fd, const struct iovec *iov, int iovcnt)
ssize_t preadv(int fd, const struct iovec *iov, int iovcnt, off_t offset)
ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt, off_t offset)
```

  - Estas llamadas son análogas a las *read*, *write*, *pread* y *pwrite* salvo que en lugar de hacer la e/s a un solo buffer de memoria la realizan a `iovcnt` bufferes descritos en `iov`
  - Cada `struct iovec` describe un buffer para e/s (su dirección y el número de bytes a transferir a dicha dirección)

```
struct iovec {
    void  *iov_base;    /* Starting address */
    size_t iov_len;     /* Number of bytes to transfer */
};
```

readv.c

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/uio.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    if (argc < 2)  {
        printf("\n sintaxis: %s <file-to-read-from>",argv[0]);
        return -1;
    }

    int fd= open(argv[1],O_RDONLY);

    char buffer1[100]; char buffer2[100];
    struct iovec v[2];  int count = 2;

    v[0].iov_base = buffer1;    v[0].iov_len =10;
    v[1].iov_base = buffer2;    v[1].iov_len =7;

    int leidos = readv(fd,v,count);
    if (leidos == (v[0].iov_len + v[1].iov_len)) {
    printf("\n leidos = %d bytes",leidos);
        int i,j;
        for (i=0;i<count;i++) {
            printf("\n [** l %3d **]",i);
            for (j=0;j<v[i].iov_len; j++) {
                printf("%c",((char*) v[i].iov_base)[j] );
            }
        }
        printf("\n");
    }
}
```

Exemplo readv()

```
/* ************************** */
/* CODIGO                     */

    char buffer1[100];
    char buffer2[100];

    struct iovec v[2];

    v[0].iov_base = buffer1;
    v[0].iov_len  = 10;
    v[1].iov_base = buffer2;
    v[1].iov_len  = 7;

    int leidos = readv(fd,v,count);
    ...

/* ************************** */
/* EJECUCIÓN                  */

user@laptop:~$ cat a.txt

  1234567890abcdefghijk

user@laptop:~$ ./a.out a.txt

  leidos = 17 bytes

  [** 1   0 **]1234567890    //leu os 10 primeiros bytes e cárgaos no buffer1
  [** 1   1 **]abcdefg       //leu os  7 seguintes bytes e cárgaos no buffer2
```

Entrada/salida en UNIX

Entrada/salida asíncrona

- En unix tenemos la posibilidad de que los procesos hagan una e/s asíncrona
  - Inician la e/s y son avisados cuando se completa
- Las funciones corresponden al estándar *POSIX real time* (debe *linkarse* con `-lrt`)

```
int aio_read(struct aiocb *aiocbp)
```
Queues request: returns 0 on success, -1 on error

```
int aio_write(struct aiocb *aiocbp)
```
Queues request: returns 0 on success, -1 on error

```
ssize_t aio_return(struct aiocb *aiocbp)
```
gets return status of asynchronous I/O op (ex. same as synchronous read,write)

```
int aio_error(const struct aiocb *aiocbp)
```
shows the error status of a previous async request

```
int aio_cancel(int fd, struct aiocb *aiocbp)
```
Tries to cancel a queued async I/O request

Entrada/salida asíncrona

+ info: man aio

- La estructura aiocb contiene los siguientes miembros (y es posible que otros, dependiendo de la implementación)

```
int              aio_fildes;     /* File descriptor */
off_t            aio_offset;     /* File offset */
volatile void   *aio_buf;        /* Location of buffer */
size_t           aio_nbytes;     /* Length of transfer */
int              aio_reqprio;    /* Request priority */
struct sigevent  aio_sigevent;   /* Notification method */
int              aio_lio_opcode; /* Operation to be performed;
                                    lio_listio() only */
```

man aio -->
aio_sigevent This field is a structure that specifies how the caller is to be notified when the asynâ chronous I/O operation completes. Possible values for aio_sigevent.sigev_notify are SIGEV_NONE, SIGEV_SIGNAL, and SIGEV_THREAD. ***See sigevent(7) for further details.***

CAMPOS: sigevent.sigev_notify = SIGEV_SIGNAL; sigevent.sigev_signo=IO_SIGNAL; sigevent.sigev_value = (valor a pasar al manejador) sigevent.sigev_notify_function; sigevent.sigev_notify_attributes; sigevent.sigev_notify_thread_id

Entrada/salida en UNIX

Redirección de entrada, salida o error

- Los descriptores de fichero 0, 1 y 2, (STDIN_FILENO, STDOUT_FILENO y STDERR_FILENO) corresponden, respecitvamente a la entrada estándar, salida estándar y error estándar de un proceso
- Tanto la entrada como la salida o el error estándar pueden ser redireccionados a un fichero cualquiera. Por ejemplo, desde el `bash`, esto se hace con los símblos > para la salida, < para la entrada y `&>` para el error
- Ejemplo 1: el listado de ficheros del directorio `/usr` va al fichero `lista`

```
$ ls -l >lista
```

- Ejemplo 2: los errores de compilación del programa `p1.c` se guardan en el fichero `err`

```
$ gcc p1.c  &>err
```

Redirección de entrada, salida o error

- Las llamadas al sistema *dup, dup2* y *fcntl* con el comando F_DUPFD permiten hacer la redirección
- *dup* duplica un descriptor de fichero, y utiliza en número mas bajo disponible. EL siguiente trozo de código muestra como redireccionar la salida estándar a un fichero. (No se incluye el contro de errores)

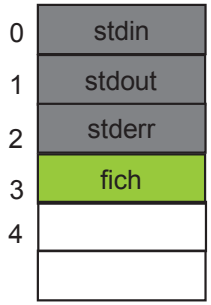
```
df=open("fichero")
close(STDOUT_FILENO);
dup(df);
/*a partir de aqui la salida est\'a redireccionada*/
```

dup / dup2

int newfd = dup(fd): encuentra la primera entrada libre en la tabla de descriptores y pone su apuntador a apuntar al mismo lugar al que apunta fd.
El viejo y el nuevo descriptor de fichero pueden ser usados indistintamente. Se refieren al mismo fichero abierto, y por ello comparten flags de status como, por ejemplo, el offset ➔ si hacemos lseek sobre uno, el offset también se ve modificado para el otro.

int fd;
fd=open(fich, O_WRONLY...);

Tabla de descriptores de fichero

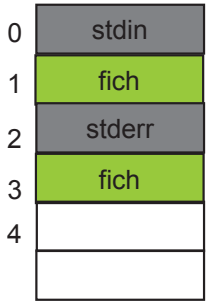


close(STDOUT_FILENO)

Tabla de descriptores de fichero

dup(fd)

Tabla de descriptores de fichero



tanto **1** como **fd** apunta a mi fichero "fich"➔La salida estándar está redireccionada a "fich"

close(fd)

Tabla de descriptores de fichero

no se podrá deshacer la redirección pues no sabemos a dónde apuntaba la entrada "STDOUT_FILENO" inicialmente

esto es similar a **dup2(fd, 1)**

dup2(fd,newfd) cierra newfd antes de duplicar si es necesario;

Es decir, dup2(fd,newfd) es similar a hacer:

close(newfd); newfd = dup(fd);

Redirección de entrada, salida o error

- Si queremos deshacer la redirección, necesitamos guardar un duplicado del descriptor original y luego redireccionar a dicho duplicado

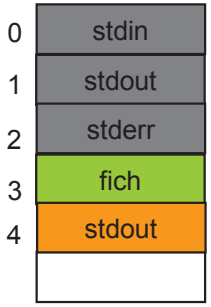
```
copiaEntrada=dup((STDOUT_FILENO);
df=open("fichero")
close(STDOUT_FILENO);
dup(df);
/*a partir de aqui la salida esta redireccionada*/
.....
/*deshacemos la redirecci\'on*/
close(STDOUT_FILENO);
dup(copiaENtrada);
close(df); /*si ya no hacen falta*/
close(copiaEntrada);
```

- Tambien puede hacerse con la llamada *dup2()* o con *fcntl()*

dup / dup2

```
int fd,copia;
fd=open(fich, O_WRONLY...);
copia = dup(STDOUT_FILENO);
```

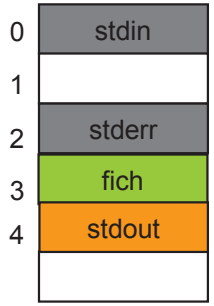
Tabla de descriptores de fichero



Guardamos una copia del descriptor asociado a la entrada “1” antes de cerrarla ➔ podremos deshacer la redirección

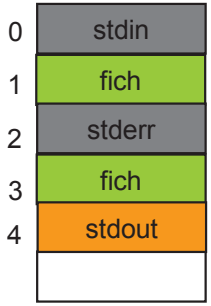
close(STDOUT_FILENO)

Tabla de descriptores de fichero



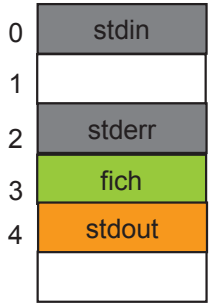
dup(fd)

Tabla de descriptores de fichero



close(STDOUT_FILENO)

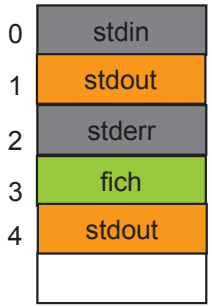
Tabla de descriptores de fichero



deshacemos redirección asociada a entrada “1”

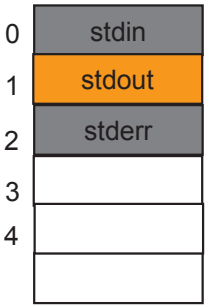
dup(copia)

Tabla de descriptores de fichero



close(copia)
close(fd)

Tabla de descriptores de fichero



Redirección: ejemplo I

- El siguiente código ejecuta el programa (con argumentos) que se le pasa como parámetro con la salida redireccionada

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

void main (int arcc, char *argv[])
{
   int df, copia,estado;
   pid_t pid;
   if (argv[1]==NULL || argv[2]==NULL){
     printf ("uso %s fich prog arg...\n",argv[0]);
     printf ("\t ejecuta prog con su argumentos redireccionando la salida a fich\n"
     exit(0);
   }
   if ((df=open(argv[1], O_CREAT|O_EXCL|O_WRONLY,0777))==-1){
     perror("Imposible abrir fichero redireccion");
     exit(0);
   }
```

Redirección: ejemplo II

```
    copia=dup(STDOUT_FILENO);
    close(STDOUT_FILENO); /*redireccion*/
    dup(df);
    if ((pid=fork())==0){
      execvp(argv[2],argv+2);
      perror ("Imposible ejecutar");
      exit(255);
    }
    waitpid(pid,&estado,0);
    close(STDOUT_FILENO); /*deshacemos redireccion*/
    dup(copia);
    if (WIFEXITED(estado) && WEXITSTATUS(estado)!=255)
      printf("programa %s redireccionado a %s termino normalmente(%d)\n",
                                        argv[2],argv[1],WEXITSTATUS(estado));
    else unlink(argv[1]);
}
```

copia del "stdout"

descriptor "1" apuntará al fichero de salida "argv[1]"

El hijo recibe copia de los descriptores de ficheros
--> donde la salida "ya" está redireccionada