1. El sistema de entrada y salida

La API de Unix uniformiza la E/S para los distintos dispositivos por medio de `open`, `read`, `write`, `close`, `ioctl`. Sin embargo, el tratamiento interno es completamente distinto según el tipo de dispositivo.

Todo parte a nivel del lenguaje de programación cuando un proceso abre un archivo para leerlo o escribirlo:

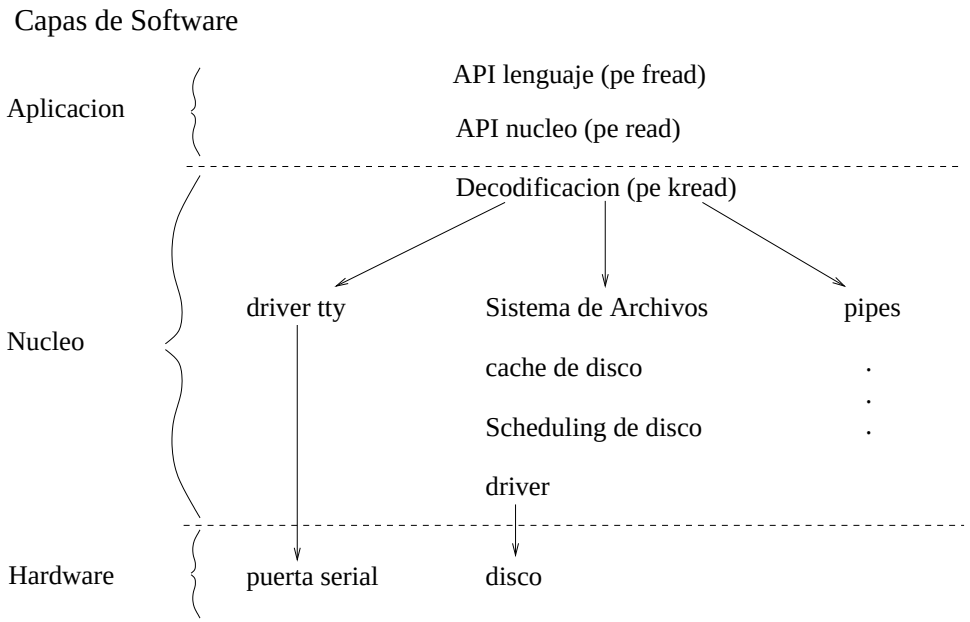
```
FILE *file= fopen("nombre", "r");
int rc= fread(file, buff, count);
...
```

Normalmente el nombre corresponde a un archivo en disco, pero en general puede ser cualquiera entre:

- un archivo: tarea1.c
- una partición de disco: /dev/hda1
- una puerta serial: /dev/tty01
- una puerta paralela, USB, etc.
- la interfaz de red
- etc.

1.1. Diseño en capas del sistema de E/S

El procesamiento de la E/S se hace a través de capas sucesivas de software. Cada una de estas capas cumple una función específica. En un diseño en capas, cada capa se implementa invocando procedimientos de la capa inferior.



A continuación se explica la función que cumple cada una de las capas.

1.1.1. API del lenguaje

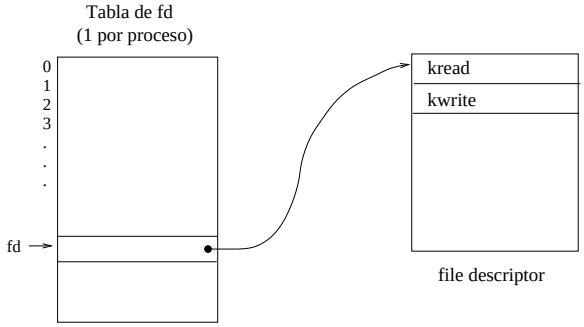
- Suministra una API común para todos los programas escritos en un lenguaje de programación específico, como por ejemplo C, independizándolo así de las variaciones en la API de las diferentes plataformas, como Windows o Unix.
- Implementa una primera barrera de buffering para disminuir las llamadas al sistema.

1.1.2. API del núcleo

- Ofrece un mecanismo estándar para hacer requerimientos de E/S al núcleo del S.O.
- Permite pasar a modo sistema.

```
int fd= open(...);
int rc= read (fd, buff, count);
```

El `fd` (*file descriptor*) es un índice a un arreglo dentro del núcleo que tiene los datos de los archivos abiertos por el proceso.



1.1.3. Decodificación

Esta capa despacha la llamada al subsistema de E/S que corresponda. El número `fd` que suministra el proceso se busca en la tabla de descriptores del proceso, que contiene un puntero a la verdadera estructura de datos asociada al archivo en el núcleo, que a su vez contiene punteros a las funciones que se utilizan para acceder al descriptor.

En C:

```
FileDescr *kfd=tablaFD[fd];
(*kfd->kRead)(kfd, buff, count);
```

En C++:

```
tableFD[fd]->kRead(buff, count);
```

1.1.4. Sistema de archivos

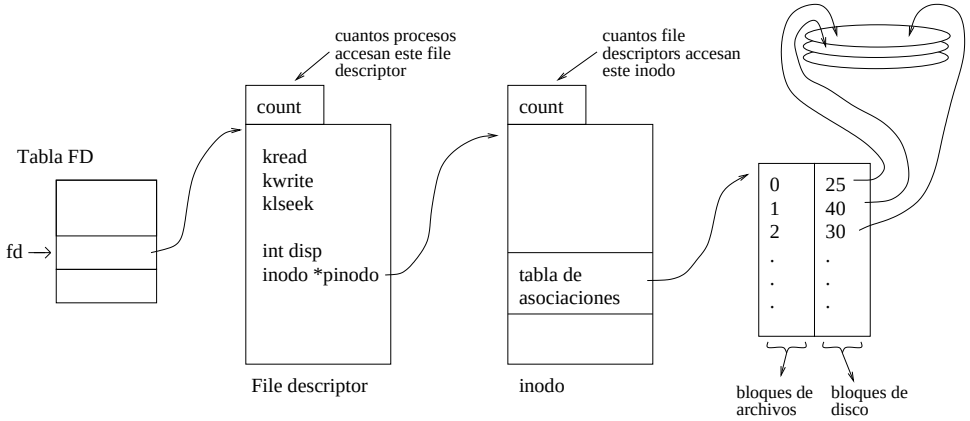
A partir de este nivel, las capas que aparecen dependen de la naturaleza del descriptor suministrado en la capa anterior. La capa sistema de archivos se aplica sólo para el caso en que el descriptor corresponde a un archivo (o directorio). Descriptores asociados a dispositivos de E/S, pipes u otros poseen capas específicas a esos subsistemas de E/S.

De aquí en adelante, consideraremos solo el caso de los descriptores asociados a archivos. En tal caso, el encabezado de los procedimientos que implementan esta capa es del tipo:

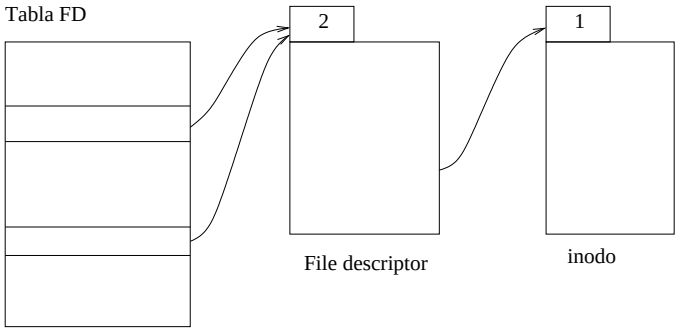
```
int kReadFile(KFD* kfd, char* buff, int count);
```

en donde KFD es el tipo de datos que se usa para representar los descriptores de archivos.

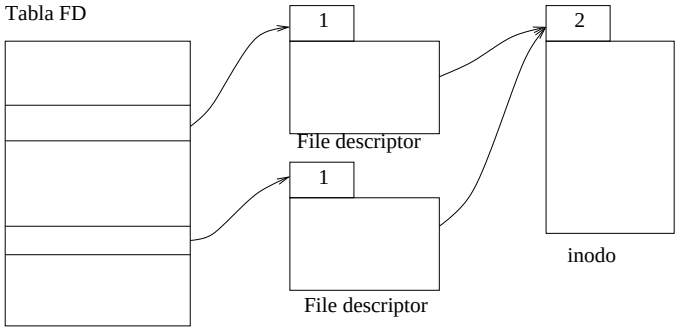
Esta capa determina en qué bloques del disco se encuentran los datos pedidos. Por ejemplo, en el sistema de archivos Unix que se verá más adelante, cada inodo tiene una tabla de asociación que indica qué bloques físicos del disco corresponden a bloques lógicos del archivo, así como un contador de referencia que indica cuántos descriptores de archivo están accediendo el inodo. A su vez, cada descriptor de archivo tiene un contador de referencias que indica cuantos procesos lo están accediendo:



Obs: al invocar `fork` el hijo hereda los descriptores del padre.



Ambos procesos se disputan la lectura del mismo archivo usando el mismo descriptor de archivo. En cambio, cuando dos procesos abren el mismo archivo,



Ambos procesos leen el archivo en forma independiente.

1.1.5. Cache de disco

Esta capa cumple las siguientes funciones:

- Mantiene los bloques recientemente leídos en memoria para no tener que volver a leerlos de disco.
- Prelee los bloques en caso de lectura secuencal (*read-ahead).*
- Escribe los bloques asincrónicamente:

  write() deja normalmente en memoria, escribe cuando falta espacio en memoria

  sync() lleva a disco todos los bloques que hayan sido modificados

Las llamadas al cache de disco son del estilo:

```
kReadCache(kDisk disc, char **psysbuff, int block);
```

El cache decide dónde deja el bloque en la memoria del núcleo y entrega en `psysbuff` un puntero hacia el bloque asignado.

1.1.6. Scheduling de disco

- Resuelve el problema del acceso concurrente al driver (secuencializa los accesos).
- Reordena los requerimientos de modo que disminuya el desplazamiento del cabezal del disco.

El encabezado del servicio entregado por esta capa es del estilo:

```
kReadSched(Disc *disc, char *sysbuff, int block)
```

Aquí `sysbuff` es `char *`, puesto que es el contenido del bloque, y no un puntero a dicho contenido. En la capa cache de disco, el parámetro `sysbuff` era un `char **`.

1.1.7. Drivers (*Pilotos*)

Esta capa es específica a tipo de disco en donde se ha instalado el sistema de archivos. Cumple con las siguientes funciones:

- Implementan la lectura y escritura del disco, independizando la programación del núcleo de los detalles específicos del disco.
- Existe un driver para discos IDE, otro para SCSI, etc.

El acceso se hace típicamente por medio de escrituras y lecturas a a los puertos de la interfaz del disco. Estos puertos corresponden a direcciones de memoria, cuya comportamiento es distinto a la memoria tradicional. Un escritura en el puerto corresponde al envío de un comando hacia el disco, y una lectura es una consulta por el estado del disco, o también para recuperar datos que se había pedido previamente. Ejemplos de accesos a la interfaz del disco son los siguientes:

```
*cmdport= ...;   /* un comando */
...= *data_port; /* lectura de datos */
```

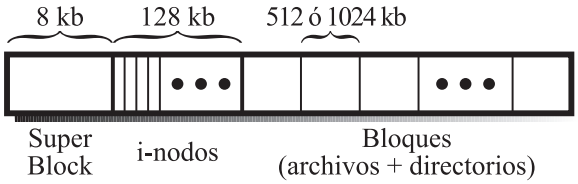
1.1.8. Disco

Esta capa la implementa el *hardware* del disco y su interfaz con el computador. Un programa almacenado en la ROM del disco es ejecutado por un microcontrolador, situado también en el disco, para desplazar el cabezal a la posición adecuada y leer o escribir los datos solicitados.

1.2. El sistema de archivos Unix

El sistema de archivos Unix se sitúa típicamente dentro del núcleo del S.O., pero existen variantes de Unix en donde se implementa en un proceso pesado. Sus funciones son las siguientes:

- Se encarga de darle estructura en archivos a una partición[1] del disco.
- Implementa el sistema jerárquico de nombres y directorios.



1.2.1. Superbloque

Almacena:

- tamaño de los bloques
- número de inodos
- número de bloques
- etc.

Por ejemplo, en Linux (`linux/include/linux/ext2_fs_sb.h`):

```
struct ext2_sb_info {
    unsigned long s_frag_size;
                /* Size of a fragment in bytes */
    unsigned long s_frags_per_block;
                /* Number of fragments per block */
    unsigned long s_inodes_per_block;
                /* Number of inodes per block */
    unsigned long s_frags_per_group;
                /* Number of fragments in a group */
    unsigned long s_blocks_per_group;
                /* Number of blocks in a group */
    unsigned long s_inodes_per_group;
                /* Number of inodes in a group */
    unsigned long s_itb_per_group;
                /* Number of inode table blocks per group */
    unsigned long s_gdb_count;
                /* Number of group descriptor blocks */
    unsigned long s_desc_per_block;
                /* Number of group descriptors per block */
    unsigned long s_groups_count;
                /* Number of groups in the fs */
    struct buffer_head * s_sbh;
                /* Buffer containing the super block */
    struct ext2_super_block * s_es;
                /* Pointer to the super block in the buffer */
```

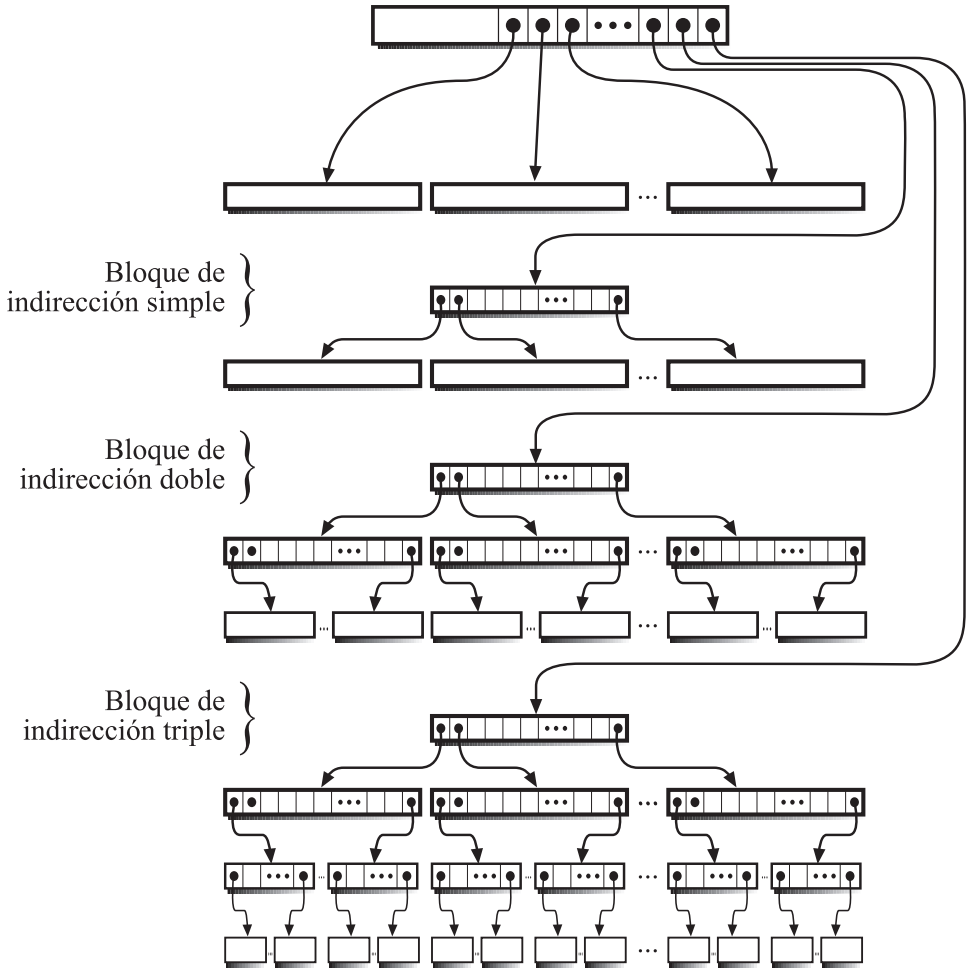
[1] El disco, visto como un conjunto lineal de bloques, se divide en trozos continuos de disco, cada uno de los cuales se denomina *partición*.

```
    struct buffer_head ** s_group_desc;
    unsigned short s_loaded_inode_bitmaps;
    unsigned short s_loaded_block_bitmaps;
    unsigned long s_inode_bitmap_number[EXT2_MAX_GROUP_LOADED];
    struct buffer_head * s_inode_bitmap[EXT2_MAX_GROUP_LOADED];
    unsigned long s_block_bitmap_number[EXT2_MAX_GROUP_LOADED];
    struct buffer_head * s_block_bitmap[EXT2_MAX_GROUP_LOADED];
    unsigned long  s_mount_opt;
    uid_t s_resuid;
    gid_t s_resgid;
    unsigned short s_mount_state;
    unsigned short s_pad;
    int s_addr_per_block_bits;
    int s_desc_per_block_bits;
    int s_inode_size;
    int s_first_ino;
};
```

1.2.2. Inodo

Es la estructura de datos que representa un archivo. Existe un inodo por cada archivo. Almacena:

- el largo del archivo
- permisos (12 bits)
- número de usuario
- número de grupo
- fecha y hora de creación, modificación y consulta (entero de 32 bits que representa segundos desde las 00:00:00 del 1-1-1970)
- número de enlaces duros (*hard links*
- el tipo: archivo normal, directorio, link simbólico, dispositivo
- 12 punteros a bloques en disco que contienen los datos del archivo
- un puntero de indirección simple
- un puntero de indirección doble
- un puntero de indirección triple



Por ejemplo, en Linux (`linux/include/linux/ext2_fs.h`):

```
struct ext2_inode {
        __u16   i_mode;         /* File mode */
        __u16   i_uid;          /* Low 16 bits of Owner Uid */
        __u32   i_size;         /* Size in bytes */
        __u32   i_atime;        /* Access time */
        __u32   i_ctime;        /* Creation time */
        __u32   i_mtime;        /* Modification time */
        __u32   i_dtime;        /* Deletion Time */
        __u16   i_gid;          /* Low 16 bits of Group Id */
        __u16   i_links_count;  /* Links count */
        __u32   i_blocks;       /* Blocks count */
        __u32   i_flags;        /* File flags */
        __u32   i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __u32   i_generation;   /* File version (for NFS) */
        __u32   i_file_acl;     /* File ACL */
        __u32   i_dir_acl;      /* Directory ACL */
        __u32   i_faddr;        /* Fragment address */
        /* Tambien un par de campos dependientes del sistema
           operativo */
};
```

Observaciones:

- Los inodos se enumeran como 0, 1, 2, ...
- El inodo 0 es el directorio raíz de la partición.
- Los directorios vienen en dos sabores: SysVr3 tenía un archivo especial con líneas de 16 bytes, donde los 14 primeros indicaban el nombre y los otros 2 el inodo donde se ubicaba. Las dos primeras entradas indican `.` y `..` respectivamente. BSDv4.2 reserva un primer byte para el largo del nombre, de largo variable hasta 255 caracteres, y luego 2 bytes para el inodo. Las primeras dos entradas siguen siendo `.` y `..`.

¿Cómo se busca un nombre? Por ejemplo `/usr/bin/X11/xfig`:

1. Ubicar partición: Si hay montados/, `/usr`, `/home` y `/tmp`, sabemos que hay que buscar en la partición que tenga `/usr`.
2. Buscamos archivo `bin` en el directorio raíz, representado por el inodo 0.
3. Buscar `X11` en el inodo donde se encuentra `bin`.
4. Buscar `xfig` en el inodo donde se encuentra `X11`.

Esto es terriblemente lento, pero la siguiente vez los datos van a estar en cache de disco y va a ser casi instantáneo.

¿Por qué poner los nombres de archivo en los directorios y no en los inodos? Hay varias razones:

- Las estructuras de los inodos serían de largo variable y eso complica bastante su mantención.
- Permite implementar *hard links*[2].

1.2.3. Gestión de bloques disponibles

Hay varias maneras de mantener información sobre los bloques que están disponibles en una partición:

1. Lista enlazada. Desventaja: produce fragmentación. Además, para leer un bloque hay que leer su contenido, lo cual es ineficiente.
2. Vector de bits: Un vector de bits almacenado en algún lugar de la partición que tiene un bit por bloque que indica si el bloque está ocupado o no. Ventaja: no se fragmenta tanto. Desventaja: necesita espacio adicional (0.025 % de sobrecosto).

[2] `ln /usr/bin/X11/xfig /usr/bin/xdraw` crea un segundo "nombre" para un archivo ya existente, siempre y cuando se esté dentro de la misma partición. Problema: borrar un link duro simplemente decrementa el contador de referencia, y sólo se elimina el espacio de datos cuando ese contador llega a cero.

Se diferencia de un link simbólico en que este último tiene un inodo propio cuyo bloque de datos contiene el nombre de lo apuntado. Por ejemplo, ejecutar `ln -s x /home/y` crea un inodo cuyo bloque de datos contiene el string "x".

Unix no permite que los usuarios creen links duros de directorios, con la excepción del usuario `root`.

1.3. El sistema de archivos FAT

Cada directorio se almacena como un conjunto de filas, que contiene un nombre y un puntero a la FAT (*file allocation table*) (un número entero).

La FAT misma es simplemente una tabla de números de 16 bits que indica para cada bloque cual es el siguiente bloque del archivo, y una marca especial (EOF) si es el último bloque del archivo.

Ventajas:

- simple
- es fácil recuperar un archivo recién borrado

Desventajas:

- ineficiente en acceso directo
- se fragmenta mucho
- como los bloques eran de 512 bytes, estaba limitado a $512 * 2^{16} = 32MB$. Primera solución: bloques de 2, 4, 16 KB, etc. Solución posterior: FAT32 (direccionamiento de 32 bits).

1.4. Scheduling de disco

El principal problema a resolver con el scheduling es que el tiempo de acceso a disco (10 mseg) es órdenes de magnitud mayor que el tiempo de acceso a memoria (60ñseg). Además, el acceso secuencial es mucho más eficiente que el acceso aleatorio: da lo mismo acceder 1 byte que 10 kbytes si están dispuestos secuencialmente en el disco.

El tiempo de acceso al disco se puede dividir en tres partes:

- Tiempo de búsqueda (*seek* en Inglés): es el tiempo que toma desplazar el cabezal hacia la pista del disco en donde se leerán o escribirán los datos. Usualmente éste es el mayor de las 3 componentes.
- Tiempo de latencia: es el tiempo que hay que esperar a que el sector que se debe leer o escribir pase por debajo del cabezal. El tiempo medio de latencia es la mitad del período de rotación del disco.
- Tiempo de transmisión: es el tiempo que hay que esperar a que los datos sean leídos o escritos.

Entonces tenemos que:

$T_{acceso} = T_{seek} + T_{latencia} + T_{transmision}$

Dada una secuencia de requerimientos de acceso a disco pendientes, el problema de *scheduling de disco* consiste en reordenar la secuencia de modo que se minimice el tiempo total de servicio. Las distintas estrategias se basan en minimizar el tiempo de búsqueda.

A continuación mostraremos como se podría resolver este problema en nSystem. Supongamos que internamente se dispone de un procedimiento `DoIODriver` que permite leer o escribir directamente un determinado sector del disco (este procedimiento estaría en la capa driver). El ecabezado de este procedimiento es el siguiente:

```
int DoIODriver(Disk *dsk, int op, int block, char *buff);
```

La implementación de este procedimiento tiene la desventaja de que sólo puede ser invocado por una tarea a la vez. Por lo tanto, la capa de scheduling debe preocuparse de no hacer más de una invocación a la vez.

Problema: Escribir el procedimiento `DoIOSched()`, perteneciente a la capa scheduling, que realiza el scheduling de disco.

Primera solución: Sólo se aborda el problema de garantizar la exclusión mutua (luego se vera cómo reordenar los requerientos pendientes). Para ello hay que modificar la estructura `Disk` de modo que colocar un semáforo:

```
typedef struct {
  nSem sem;   /* se crea con nMakeSem(1) */
  ... /* el resto de la estructura */
} Disk;

int doIOSched(Disk *dsk, int op, int block, char *buff) {
  int rc;
  nWaitSem(dsk->sem);
  rc= doIODriver(dsk, op, block, buff);
  nSignalSem(dsk->sem);
  return rc;
}
```

Segunda solución: Se reordenan los acceso al disco para reducir la suma total de los tiempos de búsqueda. Por ejemplo supongamos que una tarea pide realizar una operación de E/S con el bloque 5. Mientras se procesa este requerimiento llega primero una tarea que solicita una operación con el bloque 10, y luego, otra tarea con el bloque 2000. En el momento en que se termina la operación con el bloque 5 hay entonces 2

requerimientos pendientes. La solución de más arriba procesaría los requerimientos en el mismo orden en que se hicieron: primero el bloque 2000 y más tarde el bloque 10. Sin embargo, dado que el cabezal se encuentra en la pista del bloque 5, es más eficiente atender primero el requerimiento del bloque 10 y luego el del bloque 2000, porque los bloques se enumeran de modo que guarden una correlación lineal con el la pista en donde se encuentran. Por lo tanto, el cabezal está más cerca de la pista en donde se encuentra el bloque 10 que la pista del bloque 2000.

Por lo tanto, el scheduler debe saber en qué posición se encuentra en un instante dado el cabezal y los bloques implicados en cada uno de los requerimientos de E/S pendientes. A partir de esta información el scheduler usa alguna de las siguientes estrategias para decidir qué requerimiento conviene procesar a continuación:

- *Shortest Seek Time First* (SSTF): Se sirve primero el bloque que está más cercano a la posición actual del cabezal. Esta estrategia no se usa porque puede causar hambruna.
- LOOK: Esta estrategia es similar a un ascensor. Se sirve primero los requerimientos pendientes en orden ascendente. Si llegan requerimientos con un número de bloque superior a la posición actual del cabezal, éstos se encolan para ser servidos en esta vuelta del cabezal. Pero si el número del bloque es inferior a la posición actual del cabezal, el requerimiento se encola para ser servido a la vuelta del cabezal. En el momento en que se sirve el último requerimiento ascendente, entonces se comienza a servir los requerimientos ahora en orden descendente, y luego ascendentemente, etc.

  El problema de esta estrategia es que introduce una disparidad no deseada: las pistas centrales son mejor servidas que las pistas exteriores.
- SCAN: Similar a LOOK, pero los requerimientos se atienden siempre en orden ascendente, evitando así cualquier disparidad.

Programa:

```
typedef struct {
  nTask sched;  /* =nEnterTask(DiskSched, disk) */
  ...
} Disk;

typedef struct {
  int op, block;
  char *buff;
  nTask client;
} Req;

int DoIOSched(Disk *dsk, int op, int block, char *buff) {
  Req req;
  req.op= op;
  req.block= block;
  req.buff= buff;
  req.client= nCurrentTask();
  return nSend(dsk->sched, &req);
}

int DiskSched(Disk *dsk) {
  PriQueue *q1= MakePriQueue();
  PriQueue *q2= MakePriQueue();
  int block= 0;
  for (;;) {
    nTask client;
    Req *preq;
```

```
    while((preq= (Req*)nReceive(NULL, 0)!=NULL)
      PriPut(preq->block>=block ? q1 : q2,
             preq, preq->block);
    preq= (Req *)PriGet(q1);
    if (preq==NULL) {
        PriQueue *aux= q2; /* Intercambiar q1 y q2 */
        q2= q1;
        q1= aux;
        preq= (Req *)PriGet(q1);
        if (preq==NULL)
          preq= (Req *)nReceive(NULL, -1);
    }
    block= preq->block;
    rc= DoIODriver(preq->op, block, preq->buff);
    nReply(preq->client, rc);
  }
}
```

Observaciones:

- Esta estrategia sólo es útil en la medida que hayan varios procesos realizando lecturas/escrituras sobre el mismo disco.
- El nuevo estándar Serial ATA II permite encolar en el disco hasta 8 requerimientos de E/S. El microcontralador incluido en el disco implementa una estrategia de scheduling como la de más arriba, liberando así al sistema operativo de esta tarea.

Agradecimientos: Esta segunda parte de los apuntes de sistemas operativos se hizo realidad el semestre primavera 2001 gracias a la cooperación de los alumnos Alvaro Herrera y Jocelyn Simmonds.