PRÁCTICO 8

Objetivos

- Familiarizarse con las funciones de manejo de sistemas de archivo de un sistema operativo.

Ejercicio 1 (medio)

Un sistema operativo maneja sus archivos en disco utilizando las siguientes estructuras:

```
type dir = array [1..max_files_on_disk] of record
      used : boolean
            {entrada usada o no}
      file_name : array [1..8] of char;
      file_ext  : array [1..3] of char;
      informacion : array [1..100] of char;
      comienzo : integer
            {dirección del primer sector asignado a este archivo}
end

type fat = array [1..sectors_in_disk] of integer
```

Cada elemento de la *fat* corresponde a un sector del disco. Cuando ese sector no está utilizado el elemento correspondiente de la *fat* contiene cero. En caso de que el sector este asignado a un archivo pero no sea el último del mismo, contiene la dirección del siguiente sector asignado a dicho archivo. Cuando es el último sector asignado a un archivo, contiene FFFF.

(a) Escribir un algoritmo que encuentre en el directorio un archivo, y lo recorra (es decir, lea en orden todos sus sectores).

(b) Escribir un algoritmo que cree e inicialice con ceros un archivo de *n* bytes (un sector tiene 512 bytes).

(c) Discutir las posibles consecuencias de una caída de la máquina (por ejemplo, por corte de energía) en cuanto a la integridad de las estructuras definidas.

(d) Escribir un algoritmo que chequee la consistencia de las estructuras de un disco y que, de ser posible, corrija sus anomalías.

(e) Escribir un algoritmo que reorganice el disco de manera que todo archivo quede almacenado en sectores contiguos. Explicar la conveniencia de esto último.

Ejercicio 2 (medio)

Considere un archivo, actualmente conteniendo 100 bloques. Indicar cuántas operaciones de entrada / salida son necesarias cuando estamos trabajando con las siguientes estrategias de ubicación: contigua, enlazada e indexada, si un bloque:

(a) Es agregado al comienzo.

(b) Es agregado en el medio.

(c) Es agregado al final.

(d) Es borrado del comienzo.

(e) Es borrado del medio.

(f) Es borrado del final.

Ejercicio 3 (básico)

La asignación contigua lleva a la fragmentación de disco interna o externa?

Ejercicio 4 (básico)

En los sistemas que soportan el acceso secuencial a los archivos siempre existe una operación para volver a posicionarse al comienzo. Esto es necesario para sistemas que soportan acceso aleatorio? (Random access).

Ejercicio 5 (avanzado)

Se considera un disco con *c* cilindros, *t* pistas por cilindro, *s* sectores por pista, siendo los sectores de *sl* bytes. Un archivo *L* con registros de largo fijo *rl* es almacenado en el disco en una única zona (contigua) comenzando en la ubicación (*cl*, *tl*, *sl*).

(a) Escribir fórmulas que permitan calcular la dirección (cilindro, pista, sector) donde comenzar a leer un registro *x* del archivo, para los dos casos siguientes:

- $rl \leq sl$
- $rl > sl$

(b) Determinar la cantidad de operaciones de seek a realizar para leer todo el archivo si éste contiene *m* registros.

Ejercicio 6 (medio)

Se considera un disquete que está formateado con 8 sectores de 512 bytes por pista, y rota a 300 revoluciones por minuto.

(a) ¿Cuánto se demora en leer en orden todos los sectores de una pista, asumiendo que el brazo ya está posicionado y que se demora 1/2 revolución en llegar al primer sector? ¿Cuántos bytes se transfieren efectivamente por segundo?

(b) Contestar las preguntas anteriores suponiendo un factor de entrelazado (interleaving) de 2.

Ejercicio 7 (básico)

Sea un disco con 48 cilindros numerados del 1 al 48. Se considera una cola de solicitudes de acceso para los cilindros 10, 22, 20, 2, 40, 6, y 38, en ese orden. Se supone el tiempo de seek en 6 milisegundos por cilindro. El brazo se encuentra en el cilindro 20. Determinar el tiempo total de seek para las estrategias:

(a) First Come, First Served.

(b) Shortest Seek Time First.

(c) SCAN (atendiendo solicitudes en ambas pasadas, asumiendo que en la primera pasada la dirección es hacia cilindros de número creciente).

(d) C-LOOK .

Ejercicio 8 (avanzado)

Se dispone de un sistema operativo que almacena sus datos en un sistema de archivos tipo XSFAT (eXtended Secure FAT). Este sistema tiene algunas modificaciones con respecto al clásico FAT, ya que:

- Soporta archivos y directorios jerarquizados
- Permite almacenar los datos en múltiples discos duros. Esto es, un archivo puede estar disperso a lo largo de múltiples dispositivos
- Los archivos y directorios del mismo tienen asociados un conjunto de usuarios los cuales podrán operar con los mismos (altas, bajas, lecturas y escrituras)
- Los discos poseen sectores de 256 bytes. Si un archivo utiliza parte de un sector, entonces ese sector es inaccesible para otro archivo.

Se dispone de las siguientes funciones y procedimiento auxiliares:

- **busca_libre(in tabla): integer**
  Dada una tabla (según su definición) retorna un índice en la misma donde exista una entrada libre

- **parse(in string, out string[ ], out largo)**
  Dado un string que representa una ruta, devuelve un arreglo de strings conteniendo los componentes de la ruta, asumiendo que estos componentes están separado por "/".

  Por ejemplo, al invocar **parse("/dir1/dir2/arch1", salida, largo)**, obtendremos salida = { "dir1", "dir2", "arch1" } y largo = 3

- **read y write**
  Puede asumir que existe una función *read* para lectura de datos del disco, que recibe la dirección física donde estos se encuentran y retorna un arreglo de bytes con los datos allí presentes. Análogamente se puede suponer lo mismo para la función *write*

Nota: Puede asumir también que el directorio raíz se encuentra en una posición fija definida por usted.

Se pide:

(a) Definir las estructuras de datos de la XSFAT, de manera tal que los requerimientos anteriores puedan ser cumplidos

(b) Implementar el procedimiento **xs_create** que crea un archivo en disco. Este tiene el siguiente prototipo:

xs_create(in name: string, in data: byte[], in users: string[100])

- name: Nombre completo del archivo, incluyendo la ruta completa
- data: Es un array de bytes con los datos del archivo
- users: Es el array de usuarios que tienen permiso para trabajar con el archivo

Ejercicio 9 (en clase)

Un sistema operativo administra sus archivos en disco utilizando el método de asignación indexada. Para esto, se dispone de las siguientes estructuras:

```
type block = array [0..511] of byte;              // 512 bytes
type dir_entry = Record
                   name : Array [1..12] of char;  // 12 * 8 bits
                   type : (file,dir);             // 1 bit
                   used : Boolean;                // 1 bit
                   inode_num : Integer;           // 16 bits
                   perms : array [1..14] of bit;  // 14 bits
                 End;
type inode = Record
               inode_num : Integer;                // 16 bits
               used      : Boolean;                // 1 bit
               data      : Array [1..5] of Integer;// 5 * 16 bits
               tope      : 0..5;                   // 3 bits
               type      : (file, dir);            // 1 bit
               size      : Integer;                // 16 bits
               reserved  : array[1..11] of bit;    // 11 bits
             End;
type inode_table = Array [0..max_inode_on_disk] of inode;
type disk = Array [0..max_blocks_on_disk] of block;
Var
  TI   : inode_table;
  D    : disk;
```

A su vez, se sabe que el directorio raíz es el inodo número 0, que la tabla de inodos y el disco son globales, y que cada bloque de datos de los directorios tiene 32 entradas de tipo `dir_entry`.

(a) Implementar una función que retorne la cantidad de bytes utilizados por los archivos (`type == file`) del sistema de archivos.

(b) Implementar una función que busque un archivo (`file` o `dir`) dentro de un directorio (no búsqueda recursiva).

```
Procedure searchFile (archivo: Array [1..12] of char; inodo : Integer; Var
nro_inodo: Integer; Var ok: Boolean);
```

Donde `archivo` es el nombre del archivo a buscar, `inodo` es el número de inodo del directorio donde buscar el archivo, `nro_inodo` es para retornar el número de inodo buscado y `ok` es para retornar si la operación se concretó con éxito o no.

Asumir que se dispone de una función que lee del disco el bloque pasado como parámetro: `readBlock(d: disk; block_num: 0..max_blocks_on_disk; Var buff : block);`

(c) Implementar una función que dado un camino absoluto (ej.: `/home/sistoper/archivo.txt`) retorne el número de inodo correspondiente.

```
Procedure getInode(cam: array of char; Var nro_inodo: Integer; Var ok: Boolean);
```

Donde `cam` es el camino absoluto, `nro_inodo` es el número de inodo del archivo referenciado y `ok` es para devolver si la operación se concretó con éxito o no.