Operaciones en Organizaciones de archivos

Explicación de primitivas.

Organización de Datos (75.06)
Cátedra: Saubidet – Vera – Argerich
Facultad de Ingeniería. Universidad de Buenos Aires

Fecha de última edición: Agosto 2004

Índice

1.- Streams.

Se utilizarán las funciones provistas por el lenguaje C. Para mas información consultar un manual de lenguaje de Programación en C.

2.- Organización Secuencial.

Características

Sólo soporta lectura/escritura secuencial.

Crear:

int S_CREATE (const char nombre_fisico);*

Devuelve un código de resultado indicando si pudo crear el archivo (RES_OK / RES_ERROR / RES_EXISTE).

Abrir:

int S_OPEN (const char nfisico, int modo);*

Devuelve el file handler si puede abrir el archivo o RES_NULL si no. Los valores posibles para el parámetro modo son en este caso READ que abre el archivo (previamente creado) y se posiciona en el primer registro (en este modo las llamadas S_WRITE devuelven RES_ERROR), y APPEND que se posiciona al final del archivo (en este modo las llamadas a S_READ devuelven RES_ERROR).

Cerrar:

int S_CLOSE (int handler);

Devuelve un código de resultado(RES_OK / RES_ERROR).

Leer:

int S_READ (int handler, void reg);*

Sólo soporta lectura secuencial. Lee a partir de la posición indicada por el record pointer y avanza dicho puntero a la posición siguiente a la última que lee. Los registros pueden en este caso ser de tamaño variable. Una forma posible de implementar registros de longitud variable es utilizando un carácter separador de fin de registro. Este caracter de fin de registro es interno a la organización y es responsabilidad de las primitivas de lectura y escritura agregar y quitar dicho caracter, así como escapearlo si llegara a aparecer contenido dentro de los datos (por supuesto con otro carácter). El usuario no necesita conocer la existencia de dicho caracter. Por ejemplo si definimos el carácter '\' para escapear y el '*' para indicar fin de registro, entonces cuando guardamos un registro que contiene dentro un '\' lo expandimos a '\\' y cuando encontramos un * lo expandimos a '*'. Otra técnica consiste en anteponer la longitud de cada registro al comienzo del mismo al momento de la escritura. De esta forma, lee primero dicha longitud y extrae la cantidad de bytes a leer hasta el final del registro.

Por ser lectura secuencial y debido al uso de buffering, no todas las veces que se ejecuta S_READ se lee al dispositivo físico.
Devuelve un código de resultado(RES_OK / RES_EOF / RES_ERROR).

Escribir:

int S_WRITE (int handler, const void reg, unsigned long cant);*

Se puede agregar sólo al final del archivo. El parámetro *cant* se utiliza para indicar la longitud de *reg*. Toma los datos de *reg* (tantos bytes como indique *cant)*, si se utiliza un caracter de fin de registro, si dentro de estos datos apareciera un carácter especial (de escapeo o fin de registro) lo expande, agrega un caracter de fin de registro al final de los datos y por último escribe al archivo. Si se utiliza la técnica de almacenar la longitud de registro, escribe al archivo la longitud indicada por *cant* y luego escribe los datos al archivo.
Devuelve un código de resultado(RES_OK / RES_ERROR).

Borrar registros:

No soporta borrado de registros.

Borrar el archivo

int S_DESTROY(const char nombre_fisico);*

Devuelve un código de resultado(RES_OK / RES_ERROR / RES_NO_EXISTE).

3.- Organización Relativa.

Características

- Soporta acceso secuencial y por referencia por número de celda.
- No soporta clave primaria.
- Para hacer búsquedas en archivos relativos, puesto que soportan acceso por referencia por número de celda, si el archivo está ordenado por la clave de búsqueda, no es necesario buscar secuencialmente sino que se pueden utilizar algoritmos más rápidos como búsqueda binaria por ejemplo. Es entonces un error de diseño realizar una búsqueda secuencial en un archivo relativo si el mismo se encuentra ordenado por clave de búsqueda.

Crear:

int R_CREATE (const char nombre_fisico, int tam_registro, int max_reg);*

Inicializa todas las celdas del archivo.
tam_registro: tamaño de celda.
max_reg: cantidad de celdas.
Devuelve un código de resultado(RES_OK / RES_ERROR / RES_EXISTE).

Abrir:

int R_OPEN (const char nombre_fisico, int modo);*

Devuelve el file handler si puede abrir el archivo o NULL si no.

Cerrar:

int R_CLOSE (int handler);

Devuelve un código de resultado(RES_OK/RES_ERROR).

Posicionar:

int R_SEEK (int handler, int nrec);

Posiciona el record pointer en el número de celda indicado en nrec, aún si la celda está vacía. No devuelve ningún registro de datos.
Devuelve un código de resultado indicando si se ejecutó con éxito, si la celda está vacía o si se produjo algún error. (RES_OK / RES_ERROR / RES_NO_EXISTE)

Leer:

int R_READ (int handler, int nrec, void reg);*

Se usa para acceso por referencia. En el área de memoria apuntada por reg se copian los datos del registro que se encuentra en la celda indicada en nrec. No modifica el record pointer.
Devuelve un código de resultado indicando si se ejecutó con éxito, si la celda está vacía o si se produjo algún error. (RES_OK / RES_ERROR / RES_NO_EXISTE)

int R_READNEXT (int handler, void reg);*

Se usa para acceso secuencial. En el área de memoria apuntada por reg se copian los datos del primer registro que no esté vacío a partir de la posición del record pointer inclusive. Avanza el record pointer a la siguiente posición.
Si se ejecutó con éxito, devuelve el número del registro de datos accedido, si no devuelve un valor negativo que indica si se produjo algún error o se llegó al fin de archivo. (número de registro / RES_ERROR / RES_EOF)

Forma correcta de hacer lectura directa: R_READ(fd, nro_de_reg, buffer)

Forma correcta de hacer lectura secuencial (recorre sólo celdas que no estén vacías):

```
if (R_SEEK(fd,nro_de_celda_inicial)==RES_OK){
        status=R_READNEXT(fd,buffer);
        while ((status!=RES_ERROR)&&(status!=RES_EOF)){
                HacerOtrasCosas();
                status=R_READNEXT(fd,buffer);
        }
}
```

El uso de R_READ+R_READNEXT en reemplazo de R_SEEK+R_READNEXT para hacer lectura secuencial es un error en el manejo de la organización relativa. También lo es el uso de R_SEEK+R_READNEXT cuando se necesite hacer un acceso por referencia.

Escribir:

int R_WRITE (int handler, int nrec, const void reg);*

Si la celda indicada por *nrec* está vacía, escribe los datos de *reg* en dicha celda. Si está ocupada, no escribe nada e informa de que no pudo concretar la operación.

Devuelve un código de resultado indicando si se ejecutó con éxito, si la celda está ocupada o si se produjo algún error. (RES_OK / RES_EXISTE / RES_ERROR)

int R_UPDATE (int handler, int nrec, const void reg);*

Si hay datos en la celda indicada por *nrec*, actualiza con los daros en *reg.* Si la celda está vacía, devuelve RES_NO_EXISTE
Devuelve un código de resultado indicando si se ejecutó con éxito, si la celda está vacía o si se produjo algún error. (RES_OK / RES_NO_EXISTE / RES_ERROR)

Borrar registros:

int R_DELETE (int handler, int nrec);

Marca como borrado el contenido de la celda indicada por *nrec* y devuelve RES_OK. Si está vacía devuelve un código de error: RES_NO_EXISTE.

Borrar el archivo

int R_DESTROY(const char nombre_fisico);*

Devuelve un código de resultado(RES_OK / RES_ERROR / RES_NO_EXISTE).

Obtener cant. Max de registros

int R_GETMAXREGS(int handler);

Devuelve la cantidad máxima de registros que puede guardar el archivo o si el handler no es válido devuelve RES_ERROR.

4.- Organización Directa.

Características:

- Sólo soporta acceso directo.
- Al aplicar la función de hashing a la clave, se obtiene el número de registro. La función a aplicar es siempre la misma para todos los registros de un archivo dado.
- Según la función de hashing utilizada, habrá una cantidad mayor o menor de colisiones. Las mismas deberán ser resueltas de la misma forma por D_READ, D_WRITE, D_UPDATE y D_DELETE.
- Si se debe recorrer todo el archivo, es necesario hashear todo el espacio de claves posibles, puesto que no cuenta con lectura secuencial.
- Soporta clave primaria

Crear:

int D_CREATE (const char nombre_fisico, const campo *reg, const campo *clave, int max_reg);*

Inicializa todo los registros del archivo.

reg y clave: Permiten inicializar la estructura del archivo según los datos que deba contener (ver mas abajo).
max_reg: cantidad de registros que tendrá el archivo directo.

Devuelve un código de resultado(RES_OK/RES_ERROR/ RES_EXISTE).

Abrir

int D_OPEN (const char nombre_fisico, int modo);*

Devuelve el file handler si puede abrir el archivo o RES_NULL si no.

Cerrar

int D_CLOSE (int handler);

Devuelve un código de error.

Leer

int D_READ (int handler, void reg);*

Dado el registro de datos *reg*, extrae del mismo la clave (*C*) y aplica sobre la misma la función de hashing para obtener el número de registro (*N*).Luego compara la clave del registro número *N* con la clave *C*, si son iguales, ese es el registro que busca. Si no, aplica el método de resolución de colisiones. Si no logra encontrar la clave buscada devuelve un código de error.
Devuelve: (RES_OK / RES_NO_EXISTE / RES_ERROR).

Escribir

int D_WRITE (int handler, const void reg);*

Dado el registro de datos *reg*, extrae del mismo la clave (*C*) y aplica sobre la misma la función de hashing para obtener el número de registro (*N*) donde debe escribir el registro. Si la posición *N* está ocupada, compara la clave de dicha posición con *C*. Si son iguales, devuelve un código de resultado indicando que el registro ya existe. Si son distintas aplica el método de resolución de colisiones para obtener una ubicación donde grabar el registro.
Si la posición está borrada, debe aplicar el mecanismo de resolución de colisiones de todas formas para asegurarse que el registro no se grabó anteriormente en otra posición.
Si no puede obtener un lugar en el archivo, devuelve un código de resultado indicando que el archivo está lleno..
Devuelve: (RES_OK / RES_EXISTE / RES_ARCHIVO_LLENO / RES_ERROR)

int D_UPDATE (int handler, const void reg);*

Dado el registro de datos *reg*, extrae del mismo la clave (*C*) y aplica sobre la misma la función de hashing para obtener el número de registro (*N*) donde debe escribir el registro. Si la posición *N* esta marcada como borrada, aplica el mecanismo de resolución de colisiones (porque podría ocurrir que el registro se grabó originalmente el otro lugar). Si no encuentra el registro, devuelve un error indicando que no existe. Si está ocupada, compara la clave de la posición *N* con la clave *C* y si son iguales, actualiza el registro. En cambio, si son distintas, aplica el método de resolución de colisiones para buscar el registro que debe actualizar. Si lo encuentra hace la actualización, si no, devuelve un código de resultado indicando que el registro que se intenta actualizar no existe.
Si es necesario modificar la clave de un registro, debe darse de baja el mismo y volver a insertarlo con la nueva clave (D_DELETE+D_WRITE).
Devuelve: (RES_OK / RES_NO_EXISTE / RES_ERROR)

Borrar registros

int D_DELETE (int handler, const void reg);*

reg: Es un registro en el cuál sólo es necesario completar la clave (el resto de los campos no necesitan tener información válida).
Dado el registro de datos *reg*, extrae del mismo la clave (*C*) y aplica sobre la misma la función de hashing para obtener el número de registro (*N*) donde debería encontrar la clave. Si la posición *N* esta marcada como borrada, aplica el mecanismo de resolución de colisiones (porque podría ocurrir que el registro se grabó originalmente en otro lugar). Si no encuentra el registro devuelve un error indicando que no existe. Si está ocupada, compara la clave de la posición *N* con la clave *C* y si son iguales, marca el registro como borrado. En cambio, si son distintas, aplica el método de resolución de colisiones para buscar el registro que debe borrar. Si lo encuentra lo borra, sino , devuelve un código de resultado indicando que el registro que se intenta eliminar no existe.
Devuelve: (RES_OK / RES_NO_EXISTE / RES_ERROR)

Borrar el archivo

int D_DESTROY(const char nombre_fisico);*

Borra el archivo de datos y todos los archivos propios de la organización relacionados con el mismo (por ejemplo, si el área de overflow estuviera en un archivo aparte, borra también el archivo de overflow).
Devuelve un código de resultado(RES_OK / RES_ERROR / RES_NO_EXISTE).

5.- Organización Indexada.

Características:

- Soporta acceso secuencial, dinámico y por referencia.
- Soporta clave primaria y secundarias. El contenido de la clave primaria no puede repetirse en diferentes registros. Por otro lado las claves secundarias pueden tener duplicados.

Crear:

int I_CREATE (const char nfisico, const campo *reg, const campo *cl_prim);*

reg y cl_prim: Permiten inicializar la estructura del archivo según los datos que deba contener (ver mas abajo). La clave primaria es obligatoria en esta organización y al ejecutar I_CREATE, además de crear el archivo, se creará el índice por la clave primaria.
Devuelve un código de resultado(RES_OK / RES_ERROR/ RES_EXISTE)

Abrir:

int I_OPEN (const char nombre_fisico, int modo);*

modo: Lectura/Escritura.
Devuelve el file handler si puede abrir el archivo o RES_NULL si no.

Cerrar:

int I_CLOSE (int handler);

Devuelve un código de resultado(RES_OK / RES_ERROR).

Agregar índice:

*int I_ADD_INDEX(int handler, const campo *cl);*

cl : Describe los campos por los cuales se creará el índice en cuestión (Ver mas abajo).
Devuelve el ID del índice creado o un código de error. El ID 0 se reserva para la clave primaria.

Esta primitiva permite agregar los índices secundarios que sean necesarios. Por ser secundarios, es importante tener en cuenta que estos índices podrán tener valores repetidos.
Devuelve : (IndexId / RES_ERROR / RES_EXISTE_INDICE)

Quitar índice:

int I_DROP_INDEX(int handler, int indexId);

indexId: Id devuelto por int I_ADD_INDEX. No puede eliminarse el IndexId 0 pues es la clave primaria.
Devuelve un código de error. (RES_OK / RES_NO_EXISTE_INDICE / RES_ES_PRIM / RES_ERROR). Devuelve RES_ES_PRIM si se intenta eliminar el índice principal del archivo.

Verificar existencia de un índice

*int I_IS_INDEX (int handler, campo *clave)*

Verifica si existe un índice creado según la clave indicada en cl. Siempre antes de usar un índice que no sea el principal debe utilizarse I_IS_INDEX para averiguar su Id.
Devuelve: (Index Id / RES_NO_EXISTE_INDICE / RES_ERROR)

Posicionar:

int I_START(int handler, int indexId, char operador, const void* val_ref);*

indexId: Id del índice devuelto por I_ADD_INDEX o 0 para la clave primaria. Permite identificar el índice que se desea inicializar.
operador: puede ser cualquier string representando un operador relacional: “=”, “>”, “<”, “>=”, “<=”, “!=”.
val_ref: es un puntero a un registro con valores validos solo en los campos que definen al índice que referencia el parámetro indexId".
Devuelve un código de resultado(RES_OK / RES_ERROR / RES_NO_EXISTE_INDICE. / RES_NO_EXISTE). Devuelve RES_NO_EXISTE si no hay ningún registro que cumpla la condición deseada.

Esta primitiva inicializa el índice indicado por IndexId para hacer lectura secuencial. Posiciona el record pointer del índice en el primer registro que cumple la condición al comparar el valor de su clave (la correspondiente al índice que se está inicializando, no necesariamente la clave primaria) con *val_ref*, usando *operador.*

I_START no devuelve ningún registro de datos (no hace ninguna lectura), sólo posiciona el record pointer. Se debe usar I_READNEXT para leer el registro apuntado por el record pointer y es un error intentar utilizar I_READ a tal fin.

La utilización de I_START se considera como un acceso de tipo dinámico. Al realizar I_START se posiciona un puntero en un índice (primario o secundario) quedando disponible para luego recorrer el archivo mediante accesos de tipo secuencial (con el I_READNEXT, explicado a continuación).

A continuación se detalla la forma en que se realizan las búsquedas de registros en el índice.

Operador	**Búsqueda**
=	I_START posiciona el record pointer en el primer registro del índice cuya clave de búsqueda es **mayor o igual** a la indicada como parámetro. I_READNEXT lee este y los subsiguientes registros en cada nueva llamada hasta que la condición de igualdad deje de cumplirse devolviendo en este ultimo caso un RES_EOF.
>	I_START posiciona el record pointer en el primer registro del índice cuya clave de búsqueda es **mayor** que la indicada como parámetro. I_READNEXT lee este y los subsiguientes registros en cada nueva llamada hasta que se haya llegado al final del índice devolviendo en este último caso RES_EOF.
> =	Ídem al anterior salvo que se posiciona en el primer registro **mayor o igual**.
<	I_START posiciona el record pointer en el **primer registro del índice**. I_READNEXT lee este y los subsiguientes registros en cada nueva llamada hasta que la condición deje de cumplirse en cuyo caso devuelve RES_EOF.
< =	Ídem al anterior salvo que la condición en este caso es **menor o igual**.
! =	I_START posiciona el record pointer en el **primer registro del índice**. También calcula la posición del primer registro del índice cuya clave es **mayor o igual** a la indicada, que luego utilizará I_READNEXT. I_READNEXT lee el primer registro del índice y los subsiguientes en cada llamada mientras la condición de desigualdad se cumpla. Cuando se encuentre por primera vez un registro con la misma clave, I_READNEXT utilizará la posición calculada por I_START del primer registro con clave **mayor o igual** al indicado para mover el record pointer y de esta forma saltear a todos los registros con la misma clave. Luego continúa leyendo los siguientes registros hasta que se haya llegado al final del índice en cuyo caso devuelve RES_EOF.

Leer:

Lectura por referencia

int I_READ (int handler, void reg);*

reg: Al llamar a la primitva, los campos correspondientes a la clave primaria deben contener el valor de la clave que se desea leer (si contiene datos válidos en otros campos, los ignora). La primitiva devuelve el registro completo.

I_READ se utiliza para el acceso por referencia al archivo indexado (sólo por clave primaria). Extrae de *reg* el valor de la clave primaria y busca ese valor en el archivo usando el índice principal. Si lo encuentra, devuelve en *reg* el registro con todos los datos, si no, devuelve un código de error. El acceso por referencia sólo es posible por la clave primaria por ser esta la única que garantiza la ausencia de claves repetidas.
Devuelve: (RES_OK / RES_ERROR / RES_NO_EXISTE).

Lectura secuencial

int I_READNEXT(int handler, int indexId, void reg);*

indexId: Id del índice a utilizar
I_ READNEXT se utiliza en conjunto con I_START para lectura secuencial. Una vez que se posiciona el record pointer del índice usando I_START (Acceso Dinámico), se utiliza I_ READNEXT para leer el registro al que quedó apuntando. Además, avanza el record pointer al siguiente registro (según el índice utilizado) de forma que si se ejecuta nuevamente I_ READNEXT, se leerá el siguiente registro según el orden de dicho índice (ver tabla de búsqueda).
Devuelve: (RES_OK / RES_ERROR / RES_EOF)
En la sección 7 se puede encontrar un ejemplo que muestra la forma correcta de hacer lectura secuencial:

El uso de I_READ+I_READNEXT en reemplazo de I_START+I_READNEXT para hacer lectura secuencial es un error en el manejo de la organización indexada. También lo es el uso de I_START+I_READNEXT cuando se necesite hacer un acceso por referencia.

Escribir:

int I_WRITE (int handler, const void reg);*

reg: Es un registro de datos completo (todos los campos deben contener datos a menos que alguno sea opcional).

Se utiliza para insertar registros nuevos. Extrae de *reg* el valor de la clave primaria y busca ese valor en el archivo usando el índice principal. Si lo encuentra, devuelve un código de resultado indicando que se trató de insertar una clave duplicada. Si no lo encuentra, inserta el registro en el archivo y actualiza todos los índices.
Devuelve: (RES_OK / RES_ERROR / RES_EXISTE)

int I_UPDATE (int handler, const void reg);*

reg: Es un registro de datos completo (todos los campos deben contener datos a menos que alguno sea opcional).

Se utiliza para modificar registros ya existentes. Extrae de *reg* el valor de la clave primaria y busca ese valor en el archivo usando el índice principal. Si no lo encuentra, devuelve un código de resultado indicando que se trató de modificar un registro que no existe. Si lo encuentra, actualiza el contenido del archivo con el contenido de *reg*. No es posible utilizar I_UPDATE para modificar el valor de la clave, para eso debe usarse I_DELETE+I_WRITE
Devuelve: (RES_OK / RES_ERROR / RES_NO_EXISTE)

Borrar registros:

int I_DELETE (int handler, const void reg);*

reg: Al llamar a la primitiva, los campos correspondientes a la clave primaria deben contener el valor de la clave que se desea eliminar.

Extrae de *reg* el valor de la clave primaria y busca ese valor en el archivo usando el índice principal. Si no lo encuentra, devuelve un código de resultado indicando que se trató de eliminar un registro que no existe. Si lo encuentra, marca el registro como borrado.
Devuelve: (RES_OK / RES_ERROR / RES_NO_EXISTE)

Borrar el archivo

int I_DESTROY(char nombre_fisico);*

Borra el archivo de datos y todos los archivos propios de la organización relacionados con el mismo.
Devuelve un código de resultado(RES_OK / RES_ERROR / RES_NO_EXISTE).

6.- Tipos de datos de los parámetros

Tipo campo.

La estructura de datos a utilizar para los parámetros de tipo registro y tipo clave dependerá de la aplicación. Si se desarrollan las primitivas para ser utilizadas con archivos cuya estructura de registro se conoce es posible utilizar estructuras fijas, por ejemplo

```
struct registro{
int     campo1;
char    campo2[20];
int     campo3;
};
```

Esto tiene la ventaja de requerir menos programación a la hora de utilizar y de desarrollar las primitivas, pero estas quedan "atadas" a la estructura de registro del archivo. Para manejar varios archivos de la misma organización con distinta estructura de registro será necesario programar las primitivas para cada estructura en particular.
Un caso más real es desarrollar las primitivas para que puedan ser usadas con cualquier estructura de registro; En tal caso, se debe definir una forma de indicarle a las primitivas no sólo el contenido de un registro dado sino también su estructura. Esto se logra por medio de estructuras genéricas como las siguientes:

```
typedef struct {
        const char* nombre;
        int     tipo;
        int     longitud;
}campo;
```

Entonces un registro o una clave se especifica como una sucesión de campos, que contienen nombre de campo, tipo de campo y longitud del mismo, más un ultimo campo con nombre 0 (**null pointer**).

```
/* valores con signo. */
#define CHAR            1
#define INT             2
#define LONG            3
#define FLOAT           4
#define DOUBLE          5
#define UNSIGNED        0x80 /* para indicar valores sin signo se utiliza el operador |      (OR a nivel de bits) por ejemplo, para indicar un campo de entero sin signo se utiliza UNSIGNED | INT */
```

Problemas con la alineación de registros

Otro problema a la hora de manejar las primitivas se debe a la forma en que los compiladores almacenan las estructuras (**structs**) en memoria. Cuando declaramos una **struct** de C como la siguiente:

```
struct {
  int campo1;
  char campo2[20];
  long campo3;
  double campo4;
  ....
}registro;
```

esperaríamos que la siguiente condición se cumpliera:

sizeof(registro) = sizeof(campo1) + sizeof(campo2) + sizeof(campo3) + sizeof(campo4) +....

Pero en general la condición no se cumple, en realidad suele pasar que:

sizeof(registro) >= sizeof(campo1) + sizeof(campo2) + sizeof(campo3) + sizeof(campo4) +....

Esto se debe a que los compiladores tratan de acomodar los registros de una forma particular en la memoria para que luego puedan ser accedidos de forma eficiente por el procesador, a este proceso se lo llama comúnmente **alineación**. Por desgracia la forma en que el compilador realiza la alineación cambia de versión en versión y se ajusta al tipo de procesador para el cual se realiza el código final. Existen comandos para indicarle al compilador que no realice alineación de registros pero esto baja la performance del programa final así que no es recomendable utilizarlos.

Todo esto repercute en el manejo de las primitivas de la siguiente manera. Supongamos que creamos un programa que utiliza la estructura *registro* como esquema de un archivo de la organización relativa y compilamos el programa en una versión X para una plataforma Y. Luego ejecutamos el programas y genera los archivos relativos. Mas adelante compilamos el programa pero para otra versión Z para un plataforma W. Es probable que a la hora de ejecutar este nuevo programa, los datos extraídos del archivo generado con la otra versión no correspondan a los que habían sido guardados previamente debido a que las estructuras que utilizamos para almacenarlos en memoria hayan cambiado en tamaño y en alineación.

Como se puede notar, el problema no esta en la definición de las primitivas de las organizaciones sino en la forma en que pasamos los parámetros. Si hubiéramos utilizado en ambos casos la misma alineación no hubiese pasado nada grave. Por lo tanto para acceder a las primitivas necesitamos un mecanismo que sea independiente de la alineación de las estructuras que nos provee el lenguaje.

Para solucionar este inconveniente supondremos que las primitivas de las organizaciones asumirán que los campos de los registros se encuentran en forma continua (uno a continuación del otro sin dejar huecos de memoria) y que el tamaño del registro es igual a la suma de los tamaños de cada campo.

A continuación definimos algunas primitivas que resultarán útiles para el manejo de los registros en memoria:

*int REG_SIZEOF(const campo *esquema);*

Devuelve el tamaño en bytes que ocuparía un registro con el esquema pasado como parámetro.

*int REG_SET(void *buf, const campo *esq, const char *desc, …);*

Esta función recibe en “buf” un buffer de memoria en donde se guardaran los valores de los registros con esquema “esq”. En “desc” recibe un string con los nombres de los campos separados por “,” cuyos valores queremos asignar y a continuación (parámetro variable) pasamos los valores correspondientes. Devuelve RES_OK si pudo realizar la asignación, en otro caso devuelve RES_ERROR.

*int REG_GET(const void *buf, const campo *esq, const char *desc, …);*

Es similar a la función anterior, en este caso se utiliza para obtener los valores de los campos de un registro y en el parámetro variable debemos pasarle punteros a variables del tipo de campo que corresponda con el definido en el esquema “esq”. Devuelve RES_OK si pudo extraer los valores, en otro caso devuelve RES_ERROR.

Ejemplo de utilización

Este fragmento de código recorrerá un archivo indexado mostrando el campo promedio de los registros cuyo campo DNI tenga un valor mayor a 29000000. En este caso el registro contiene los campos PADRON, DNI y PROMEDIO y la clave del índice primario es PADRON. Para simplificar el código no se codifica ningún tipo de respuesta ante posibles errores al utilizar las primitivas, cosa que si debe hacerse en un ejercicio de parcial o final (ejemplo, si I_START devuelve NO_EXISTE, imprimir en pantalla “No existe ningún registro cuyo dni sea mayor a 29000000”)

```
int MostrarPromedios(){
        // Variables a utilizar
        void *mi_registro, *reg_datos;
        int fd, indice,status,size;
        unsigned int dni;
        float prom;
        campo esquema[]={ {“PADRON”, UNSIGNED | INT, 1},{“DNI”, UNSIGNED | INT, 1 }
                          {“PROMEDIO”, FLOAT, 1}, {0,0,0} };
        campo clavesec[] ={ {“DNI” , UNSIGNED | INT , 1} , {0,0,0} };

        // Asigna memoria para los registros.
        size = REG_SIZEOF(esquema);
        mi_registro = malloc(size);
        reg_datos   = malloc(size);
        // limpiamos los buffers.
        memset(mi_registro,0,size);
        memset(reg_datos,0,size);
        //Abre el archivo como lectura
        fd = I_OPEN(“/home/datos/alumnos.idx”, READ);
        indice=I_IS_INDEX(fd, clavesec);
        if (indice== NO_EXISTE_INDICE)
                indice=I_ADD_INDEX(fd, clavesec);
        REG_SET(mi_registro,esquema,”DNI”,29000000);
        if (I_START(fd,indice,”>”,mi_registro)==RES_OK){
                        status=I_READNEXT(fd,indice, reg_datos);
                        while ( status == RES_OK ){
                                REG_GET(reg_datos,esquema,”DNI,PROMEDIO”,&dni,&prom);
                                printf(“DNI: %u – Promedio: %f\n”, dni , prom);
                                status=I_READNEXT(fd,indice, reg_datos);
                        }
                }
                I_CLOSE(fd);
        }
```

```
        free(mi_registro);
        free(reg_datos);
        return 0;
    }
```

7.-Definiciones de las primitivas

Además de las definiciones previas, también se definen los siguientes valores para la utilización de las primitivas:

Modos de apertura de archivos

```
#define READ            1
#define WRITE           2
#define READ_WRITE      3 /* READ | WRITE */
#define APPEND          4
```

Valores devueltos por las primitivas

```
#define     RES_OK                  0   /*La primitiva se ejecutó con
                                          éxito*/
#define     RES_NULL               -1   /*No se pudo abrir el archivo*/
#define     RES_ERROR              -2   /*Se produjo un error*/
#define     RES_EOF                -3   /*Fin de Archivo*/
#define     RES_EXISTE             -4   /*El registro/archivo  ya existe*/
#define     RES_NO_EXISTE          -5   /*El registro no existe*/
#define     RES_EXISTE_INDICE      -6   /*El índice ya existe*/
#define     RES_NO_ EXISTE_INDICE  -7   /*El índice no existe*/
#define     RES_ARCHIVO_LLENO      -8   /*El archivo directo está lleno*/
#define     RES_ES_PRIM            -9   /*El índice que se intenta eliminar
                                          es el principal*/
```

Apéndice – Alineación

```
/* devuelve el tamanio de un campo en bytes */
int sizeofField(int tipo, int cant)
{
      tipo &= ~UNSIGNED;
      switch(tipo)
      {
      case CHAR: return sizeof(char) * cant; // cant es valido solo para char.
      case INT : return sizeof(int);
      case LONG: return sizeof(long);
      case FLOAT:return sizeof(float);
      case DOUBLE: return sizeof(double);
      default : return 0;
      }
      return 0;
}
// devuelve el tamanio en bytes de un registro con esquema esq.
int REG_SIZEOF(const campo *esq)
{
      int size;

      if( !esq ) return 0;

       // suma todos los tamanios.
       for(size=0; esq->nombre;size += sizeofField(esq->tipo, esq->longitud ),esq++  );
```

```
       return size;
}

// rellena el buffer con los valores de los campos especificados.
int REG_SET(void *buf, const campo *esq, const char *desc, ... )
{
      va_list marker;
      int    offset,i,tipo,len;
      const char *token,*strfield;
      char str[1024];


      if( !buf || !esq || !desc )
            return RES_ERROR;

      strcpy(str,desc);
      va_start(marker,desc);
      token = strtok(str,", "); // delimitadores ',' y ' '.
      while( token )
      {
            for(offset=0,i=0;esq[i].nombre && strcmp(esq[i].nombre,token); i++,
                  offset+=sizeofField(esq[i].tipo,esq[i].longitud));
            if( esq[i].nombre )
            {
                  tipo = esq[i].tipo & ~UNSIGNED;
                  switch(tipo)
                  {
                  case INT:    *(int   *)((char*)buf+offset) = va_arg(marker, int );break;
                  case LONG:   *(long  *)((char*)buf+offset) = va_arg(marker, long);break;
                  case FLOAT:  *(float *)((char*)buf+offset) = (float)va_arg(marker,
            double);break;
                  case DOUBLE: *(double*)((char*)buf+offset) = va_arg(marker,
            double);break;
                  case CHAR:
                        strfield = va_arg(marker,char*);
                        len      = strlen(strfield);
                        if( len > esq[i].longitud ) len = esq[i].longitud;
                        memcpy((char*)buf+offset,strfield,len*sizeof(char));
                        memset((char*)buf+offset+len,0,esq[i].longitud-len);
                        break;
                  default: return RES_ERROR;
                  }
            }
            token = strtok(0,", ");
      }
      va_end(marker);

      return RES_OK;
}
// obtiene los valores del buffer y los guarda en las variables especificadas.
int REG_GET(const void *buf, const campo *esq, const char *desc, ... )
{
      va_list marker;
      int    offset,i;
      const char *token;
      char str[1024];

      if( !buf || !esq || !desc ) return RES_ERROR;

      strcpy(str,desc);
      va_start(marker,desc);
      token = strtok(str,", "); // delimitadores ',' y ' '.
      while( token )
      {
            for(offset=0,i=0;esq[i].nombre && strcmp(esq[i].nombre,token); i++,
                  offset+=sizeofField(esq[i].tipo,esq[i].longitud));
            if( esq[i].nombre )
            memcpy(va_arg(marker,void *),(const
            char*)buf+offset,sizeofField(esq[i].tipo,esq[i].longitud));
       token = strtok(0,", ");
      }
      va_end(marker);

      return RES_OK;
}
```