CM

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITECNICO NACIONAL

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

SECCIÓN DE COMPUTACIÓN

"LENGUAJE C_PERSISTENTE"

Tesis que presenta FELIX FRANCISCO RAMOS CORCHADO para obtener el grado de MAESTRO EN CIENCIAS en la especialidad de INGENIERIA ELECTRICA con opción de COMPUTACION

México D.F. Julio de 1989

<dedidicación> ⟶ <padres> 'y' <hermanos>

<padres> ⟶ Félix Ramos Saavedra

| Hermila C. de Ramos

<hermanos> ⟶ <nombres><apellidos>

<nombres> ⟶ Miguel Angel | Mario Ernesto

| Ma de Lourdes | Ma del Pilar

| Patricia | Ma Luisa

| Guillermo Pablo | Pedro Javier

| Jorge | Marco Antonio

| Juan Carlos

<apellidos> ⟶ Ramos Corchado

<agradecimientos> ⟶ <asesores> / <jurado>

/ <institución> / amigos

<asesores> ⟶ Renato Barrera Rivera

/ Juan Carlos Pérez Castañeda

<jurado> ⟶ Juan Carlos Pérez Castañeda

/ Josef Kolar Sabor

/ Oscar Olmedo Aguirre

<institución> ⟶ CFE / COFAT / CIMAT

LENGUAJE C_PERSISTENTE

TABLA DE CONTENIDO

CAPITULO IV

PREPROCESADOR

CAPITULO V

MANEJADOR DE OBJETOS PERSISTENTES.

CAPITULO I

INTRODUCCION

El propósito del presente trabajo es presentar un enfoque en la programación que tiende a facilitar el desarrollo de sistemas, basándose en la unificación de las abstracciones de memoria primaria y memoria secundaria.

I.1 ANTECEDENTES

En lenguajes tradicionales existe la facilidad de crear datos en memoria primaria; sin embargo, si el usuario desea conservarlos tendrá que crear alguna función que entere al sistema manejador de archivos ("file system") para que éstos sean almacenados en memoria secundaria, otra opción disponible para conservar información que nos es útil es el empleo de otras herramientas, como lo son los manejadores de bases de datos o los manejadores de archivos.

De lo anterior podemos decir que el usuario interesado en conservar cierta información, tiene que trabajar con dos tipos de datos diferentes que son:

i.- Los que desaparecen junto con el programa que los creó.

ii.- Los que desea conservar aún después de que el programa que los creó termine.

El primer tipo de dato es manejado normalmente con las capacidades mismas del lenguaje, mientras que el segundo lo es mediante el sistema manejador de archivos en el caso de lenguajes proceduales, por funciones especificas para ese fin en el caso del manejador de bases de datos, o mediante las facilidades implantadas en el manejador de archivos.

Haciendo un análisis de las facilidades para crear información y almacenarla, nos encontramos que crear información con las facilidades propias de un lenguaje tradicional, es la más general, ya que podemos establecer casí cualquier relación entre entidades, así por ejemplo; podemos generar una estructura de datos arborecente en la cual cada uno de sus nodos represente un registro de un trabajador o una capa de un sistema jerarquico, o bien podemos crear una red que represente la fase análisis léxico de un compilador, etc.. Esta versatildad de generar información de los lenguajes no puede ser obtenida con las facilidades de un manejador de bases de datos, sin embargo, en el proceso necesario para guardar la información deseada mediante las facilidades disponibles en un lenguaje procedual tenemos que el mapeo necesario entre los dos tipos de memoria, primaria y secundaria, para conservar los datos es efectuado en dos partes:

i.- Código generado por el usuario para este fin.

ii.- Funciones del manejador de archivos.

Lo anterior es necesario debido a que los lenguajes y el manejador de archivos organizan de una manera diferente los datos, ya que administran diferentes tipos de memoria .

Las diferencias entre los dos tipos de datos, tanto respecto a su longitud de vida como a su dominio, traen consigo problemas al usuario, por ejemplo:

- Entender el manejo del mapeo entre los dos tipos de datos para así generar el código necesario.
- Pérdida de la posible protección ofrecida por el lenguaje a la información.

I.2 DEFINICION DEL PROBLEMA Y SOLUCION

Teniendo los antecedentes anteriores, podemos definir el problema como la existencia de dos tipos de datos y dos tipos de organización, cada uno manejado por partes diferentes (lenguaje y manejador de archivos respectivamente). lo cual dificulta al usuario a hacer uso armonioso de las facilidades, qué para generar información y conservarla proporciona un lenguaje.

Existen como es el caso común diversas soluciones a un problema, y en nuestro caso tenemos las siguientes:

1ª.- implantar un lenguaje que sustraiga el problema detectado.[1]

2ª.- implantar funciones que ayuden de alguna manera a solucionar el problema en las herramientas disponibles ya sean manejadores de bases de datos o lenguajes proceduales.

En nuestro caso despues de hacer un estudio de las herramientas disponibles y de las corrientes que dirigen la solución a este problema, seleccionamos la segunda opción para el caso de lenguajes proceduales y establecemos como solución la siguiente:

Unificar el manejo de los dos tipos de datos detectados (bajo el punto de vista del usuario) en el lenguaje C, mediante la implantación de funciones que permitan al lenguaje hacer el

[1] implantaciones de este tipo inician con el desarrollo de SmallTalk de Xerox.

mismo nivel de abstracción para ambos tipos datos. La concepción del sistema es escencialmente dejar el mayor trabajo posible al compilador del lenguaje e implantar dos módulos, el primero deberá de extraer información de los objetos persistentes, y el segundo puede conceptulizarse como una biblioteca la cual tiene el objetivo de indicar al compilador el manejo de la información que se desea conservar o consultar. El propósito del presente trabajo describir la implantación de dichas funciones.

El presente trabajo está organizado en seis capítulos, incluyendo el presente, de los cuales damos una lista a continuación.

El primer capítulo contiene: La introducción, los antecedentes, la definición del problema y la solución implantada. En el capítulo II se muestra la concepción del sistema, sus características y describe las partes que lo constituyen. En el capítulo III se describen los términos usados en la implantación. Los capítulos IV y V describen las implantaciones de los módulos que componen el sistema. El capítulo VI contiene los resultados obtenidos de las pruebas y los tiempos de ejecución de las operaciones básicas efectuadas sobre objetos persistentes. El capítulo VII contiene conclusiones y perspectivas de la implantación. Finalmente el apéndice contiene la documentación de las funciones que componen el sistema.

CAPITULO II

EL SISTEMA

II.1 CONCEPCION DEL SISTEMA

El sistema presentado es un programa compuesto de dos módulos que permiten hacer programación persistente, ésto es, escribir programas que tienen el mismo nivel de abstraccción para memoria primaria y memoria secundaria. De está manera con las capacidades del lenguaje, el usuario puede crear, eliminar o modificar información, mandarla a memoria secundaria y solicitarla posteriormente para emplearla con el mismo procedimiento o con otro si tiene el permiso de acceso apropiado.

Este nivel de abstracción permite al usuario no preocuparse de la forma en que están guardados los datos.

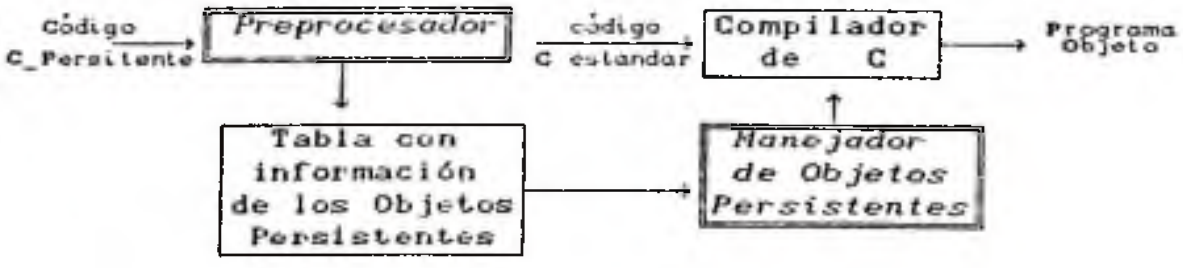


Fig. II.1 Diagrama de concepción del sistema

II.1 CARACTERISTICAS DEL C-PERSISTENTE

i).- El universo de discurso del lenguaje "C_persistente" es el mismo que el del lenguaje "C" más los objetos persistentes, salvo restricciones discutidas posteriormente. Con esto aseguramos un cambio mínimo al lenguaje.

ii).- Creación de objetos persistentes. Su identificación será a partir de su declaración.

II.2 DESCRIPCION DEL SISTEMA

Los módulos que componen el sistema están implantados en lenguaje "C" y son para proporcionar persistencia al lenguaje "C", el cual fue escogido por las siguientes razones:

- Posee buena portabilidad, esto es, es el código generado en la mayoría de los casos corre sin ningún cambio en diferentes computadoras.

- Es un lenguaje que provee la facilidad de incluir funciones en bibliotecas, las cuales pueden ser empleadas facilmente.

El sistema está dividido en dos módulos con funciones específicas. Estos módulos y sus funciones son las siguientes:

i).- Preprocesador.

Función que desempeña:

Genera a partir del programa escrito en lenguaje "C-persistente", un programa equivalente en lenguaje "C".

ii).- Manejador de Objetos Persistentes (MOP)

Función que desempeña:

Mapea de memoria primaria (RAM) a memoria secundaria (disco) y viceversa.

La forma como el sistema opera es la siguiente:

i.- El procesador recibe como entrada un archivo escrito en "C-persistente". Con este pograma, genera una lista de identificadores de objetos persistentes, identifica las llamadas a las funciones del MOP para efectuar las expansiones de las funciones necesarias, e identifica las instrucciones que manejan indirecciones de objetos persistentes para hacer la llamada a la función manejadora de indirecciones.

ii.- El programa resultante en lenguaje "C" puede pasar ya por un compilador estandar que detectará los errores cometidos y ligara las funciones del manejador de objetos.

CAPITULO III

CONCEPTOS

Este capítulo describe algunos de los conceptos empleados a lo largo de la tesis. Los conceptos descritos son: persistencia, programación persistente, objeto persistente, índice de objeto base, clase e índice de clase.

III.1 ESTRUCTURA DE DATOS

Este término es empleado para referir datos que se manejan como una unidad y cuya característica principal es la representación de ciertas estructuras, por ejemplo: listas, pilas, arboles, etc. .

III.2 ESTRUCTURA

Por simplicidad, cuando se mencione el término estructura, se estará hablando de una estructura del lenguaje "C".

III.5 OBJETO PERSISTENTE

Un término que manejamos frecuentemente es el de Objeto Persistente ("OP" u "op"). Este término lo aplicamos a objetos que tienen las propiedades siguientes:

i).- Un objeto persistente es un elemento del discurso del lenguaje que soporta la persistencia.

ii).- Un objeto persistente, si se desea, puede existir más tiempo que el programa que lo activa.

iii).- Un objeto persistente puede ser accesado para efectuar con él cualquiera de las operaciones básicas (inserción, modificación, borrado), por cualquier procedimiento que lo requiera, si tiene el permiso necesario y sin importar que éste no lo haya creado.

III.4 PERSISTENCIA

El término persistencia se aplica a un objeto (véase II.5) y está relacionado con su período de vida, esto es, con el tiempo durante el cual la información (objeto) es accesible para su empleo.

En lenguajes tradicionales los datos normalmente tienen un período de vida igual o menor al del programa que los activa; si se desea que los datos sobrevivan se necesitará del sistema de archivos, o de un manejador de base de datos y de código del usuario.

En lenguajes con la facilidad de persistencia, el programador no tiene que salirse de su tarea para almacenar su información. El método de acceso a los datos es independiente de si el dato es de vida corta o larga.

III.5 PROGRAMACION PERSISTENTE

Por programación persistente entendemos un método de programación donde el usuario no tiene que preocuparse por el almacenamiento de los datos que sean útiles para

procesamientos posteriores, ya sea con el procedimiento que los creó o con otro que los requiera.

III.6 OBJETO BASE

Un objeto base es una estructura de tipo "C" que funciona como encabezado para "colgar" objetos persistentes .

III.7 INDICE

Estructura de datos por la que se pueden accesar los objetos presistentes y clases "colgados" de un objeto base.

III.8 CLASE

Una clase es un conjunto de objetos que comparten una declaración. La declaración de una clase es obtenida de la declaración de una estructura tipo "C". Su identificador dentro del índice es obtenido del identificador de la estructura.

III.9 INDICE DE CLASE

Estructura de datos por medio de la cual se pueden accesar los objetos de una clase y que pertencen a un objeto base.

CAPITULO IV

PREPROCESADOR

Este capítulo se refiere al trabajo realizado por el preprocesador, uno de los dos módulos que componen el sistema.

En este capitulo se justifica la necesidad del preprocesador, se explica la forma en que son realizadas las tareas que efectúa y se describe la información generada, con la cual el compilador estandar C, puede manejar conjuntamente con el MOP los OPs.

IV.1 NECESIDAD DEL PREPROCESADOR

En el sistema presentado el preprocesador de C_persistente es un procedimiento que toma como entrada un programa escrito en lenguaje C_persistente, y entrega como salida un programa que efectúa la(s) misma(s) tarea(s) que el que tomó como entrada, con la diferencia de que el programa resultante podrá ser compilado con un compilador de C "estandar", incluyendo el preprocesador propio del compilador.

Este módulo es necesario por las siguientes razones:

i.- El método adoptado de incrementar la potencia del lenguaje base (en nuestro caso el lenguaje "C"), para que con éste se pueda hacer programación persistente, trae consigo que el compilador ya no sea omnipotente en cuanto a los tipos de datos que pueden ser manejados en un programa, esto implica, que el sistema propuesto debe de obtener información necesaria y comunicarla al compilador, para que éste pueda efectuar su tarea normalmente.

ii.- Los usuarios serán los que utilizarán las funciones del MOP por lo que éstas deben de ser: poderosas y en lo posible transparentes.

IV.2 TAREA Y SU IMPLANTACION

La tarea encargada al preprocesador de obtener información de los objetos persistentes que maneja un programa, se resuelve efectuando un análisis al programa escrito en C_pesistente, en este análisis se identifican:

i.- Declaraciones de OPs.

ii.- Ambito de las declaraciones de OPs.

iii.- Instrucciones que involucran identificadores de OPs.

iv.- Instrucciones que contienen llamadas a funciones del manejador de OPs.

A continuación se describe la información que se obtiene de cada uno de los incisos anteriores, y se explica porqué es necesaria.

IV.2.1 DECLARACIONES DE OPs.

El procesamiento de las declaraciones de OP es necesario por las siguientes razones:

i.- El MOP necesita saber la descripción de los **OPs** para tomar decisiones con respecto a su administración, ya que de la descripción se puede determinar:

- Tipo del objeto (carácter, entero, estructura, etc.).
- Tamaño en bytes que ocupa.
- Alcance de los identificadores de los **OPs**.

ii.- Los **OPs** son entidades que pueden ser referenciadas únicamente por medio de sus identificadores únicos, los cuales les son asignados al momento de crearse y que denominamos **PIDs** (véase V.2.1). Como el usuario no sabe los **PIDs**, para hacer referencia a algún objeto de tipo estructura lo hace por medio de variables peristentes declaradas de tipo apuntador (únicos apuntadores válidos), estos apuntadores tienen que ser identificados dentro de las declaraciones de estructuras y transformados a **PIDs**.

Ejemplo 1:

Supongamos la siguiente declaración:

```
persistent struct perro{
        char nombre[20];
        int edad;
        struct perro *hijo;
        struct perro *padre;
 };
persistent struct perro *pperro1,*pperro;
```

El preprocesador transforma este conjunto de declaraciones en:

```
struct perro{
        char nombre[20];
        int edad;
        long hijo;
        long padre;
};
struct perro *pperro1,*pperro;
```

iii.- La posición ocupada por una declaración dentro del código nos sirve para:

- Generar una tabla que nos indique el alcance de los identificadores.
- Generar el apuntador apropiado, a partir de las instrucciones que hacen uso de los identificadores.

Debido a que las declaraciones de **OPs** son diferentes de las declaraciones del C estandar, es necesario un procedimiento que verifique la sintaxis de las declaraciones. En esta implantación este procedimiento a la vez que verifica la sintaxis obtiene la información que requiere el **MOP.**

Los **OPs** que son reconocidos por el sistema son "char", "int", estructuras y arreglos de éstos. A continuación se muestran, a manera de gráfos, los autómatas que reconocen su sintaxis.

IV.2.1.1 Autómata que Reconoce OPs de Tipo Básico.

Un **OP** de tipo básico puede ser cualquier elemento del discurso del C estandar de tipo entero, carácter, o arreglo de estos. Debido a

que los OPs son entidades que únicamente pueden ser referenciadas por medio PIDs, deben de ser distinguibles los OPs de las variables de tipo básico de C. La a manera en que se diferencian éstos en la presente implantación es dando una sintaxis diferente

a los OPs de tipo básico.

A continuación se muestra a manera de gráfo dirigido el autómata que reconoce la sintaxis de declaraciones de tipo entero, carácter y arreglos de estos .

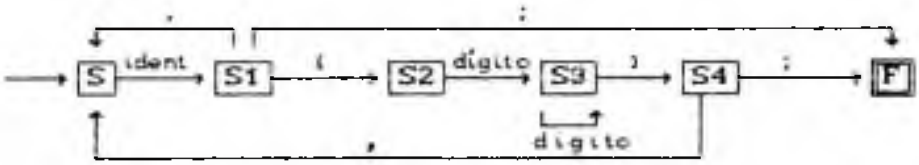


Fig IV.1 Autómata para reconocer tipos entero, caracter y arreglos de estos.

IV.2.1.2 Autómata Que Reconoce Tipos Estructura.

El autómata representado en forma de gráfo en la figura IV.2 y IV.2a muestran la sintaxis de declaraciones de OP de tipo estructura, la figura IV.2 muestra la estructura general de la sitaxis de una estructura y IV.2a la sintaxis de los miembros de la estructura.

Las diferencias entre esta sintaxis y la del lenguaje "C" para el tipo estructura son:

i.- En lenguaje "C_persistente" una declaración de un tipo estructura no puede contener otra declaración de tipo estructura.

ii.- En lenguaje "C_persistente" las estructuras pueden hacer referencia a otras estructuras de tipo

persistente a través de apuntadores.

El preprocesador verifica la sintaxis de las declaraciones, si no son correctas, genera un mensaje de error, en caso contrario modifica la declaración, sustituyendo los campos de tipo apuntador por campos de enteros largos que contendrán los **PIDs** de los objetos persistentes referenciados.

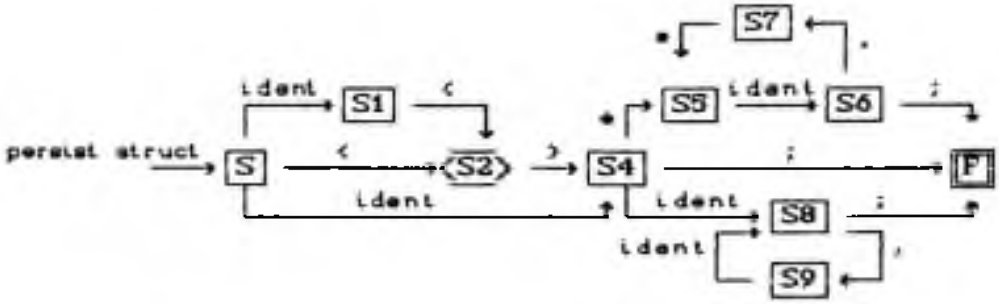


Fig IV.2 Autómata para verificar la sintaxis de estructura de OPs

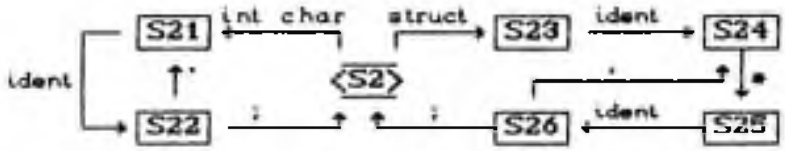


Fig. IV.2a Autómata para verificar la sintaxis de los miembros de una estructura. La salida de este autómata es mostrada en la fig. IV.2.

IV.2.3 SINTAXIS DE DECLARACIONES PARA OBJETOS PERSISTENTES

En esta sección se describe la sintaxis de la gramática de **OPs**, empleando la notación Bakus Naur Form (BNF) extendida y modificada. Los términos empleados fueron tomados del apéndice de la referencia 18.

Sintaxis de declaraciones de OP:

```
type-specifier : :=
                  integer-type-specifier
                | structure-type-specifier

integer-type-specifier :: =
                  signed-type-specifier
                | character-type-specifier

signed-type-specifier ::= int

character-type-specifier ::= char

struct-type-specifier ::=
                  structure-type-definition
                | structure-type-reference

structure-type-definition ::=
                  'struct' (structure-tag?) '{' field-list '}'

structure-type-reference ::= struct-structure-tag

structu-tag ::= identifier

field-list ::=   { component-declaration ';' }+

component-declaration ::=
                  type-specifier component-declaration-list ';'

component-declarator-list ::= (component declarator# ';' }+

component-declarator ::= simple-component

simple-component ::= declarator

declarator ::=
                 simple-declarator
               | array-declarator
               | pointer-declarator

simple-declarator ::= identifier

pointer-declarator ::= '*'declarator

array-declarator ::= declarator'['(constant-expresion)?']'

constant ::= expresión
```

IV.2.2 Ambito de las declaraciones de OP

Dado que el objetivo del preprocesador es generar un código comprensible por un compilador de C estandar, el sistema necesita saber a partir del análisis del código C_persistente, el alcance de los identificadores de los OPs, para generar el apuntador apropiado.

La forma como se determina el alcance de los identificadores, es igual a la manejada por los lenguajes estructurados de tipo bloque, conjuntamente con la técnica de anidamiento más cercano. Esta técnica es empleada por lenguajes como "pascal" y "C", etc..

IV.2.8 INSTRUCCIONES QUE INVOLUCRAN IDENTIFICADORES DE OP

Como consecuencia de la propiedad de los OPs de poder existir aún después de que el programa que los creo haya terminado, las instrucciones que hacen referencia a los OPs pueden hacer referencia a OPs que hubierán sido creados con anterioridad, por lo que tenemos dos situaciones diferentes. La primera ocurre cuando las instrucciones hacen referencia a OPs contenidos en memoria primaria (caso en el cual los OPs han sido creados durante la transacción en curso o bién que hayan sido requeridos y por lo tanto leidos). La segunda situación se presentará cuando los OPs referenciados no estén en memoria primaria.

Las situaciones descritas anteriormente fuerzan a dar un preprocesamiento a las instrucciones mencionadas. La manera como se hace el preprocesamiento en la presente implantación, es mediante una expansión de la instrucción que hace referencia a OPs. La expansión la efectúa un procedimiento de expansión, el cual intercala llamadas a la función locind (véase a continuación del ejemplo 3 de esta sección). Esta función opera con los PIDs de los OPs para, sin importar su localización, ver si es posible referenciar los OPs requeridos. En caso de que no sea posible

referenciar un OP se genera un mensaje de error.

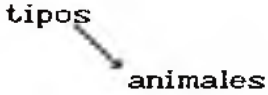
Los parámetros de la función son generados a través de la tabla de identificadores de OPs generada durante el análisis sintáctico. El tipo del valor que regresa la función es apuntador a carácter, por lo que es necesario un "cast" para forzar al tipo requerido.

Ejemplo2:

Supongamos la siguiente declaración, que es la única de tipo persistente en el archivo:

```
persistent struct animales{
        struct mamiferos  *pmam;
        struct aves       *pave;
        struct pez        *ppez;
        struct reptil     *prep;
        } *pani;
```

A partir de ésta , el preprocesador genera la siguiente información apuntada por la variable tipos que es una variable de de tipo global:



clase	campo	tipo
mamiferos	pmam	pest
avez	pavez	pest
pez	ppez	pest
reptil	prep	pest

pest declara el tipo del campo como apuntador a estructura.

Supongamos que el preprocesador encuentra la siguiente instrucción:

```
....
.....
pani->pave->patos->nombre ...
.....
...
```

Entonces genera la información siguiente:

campo	tipo	clase
pani	pest	animales
pave	pest	aves
patos	pest	pato
nombre	char	----

y con ayuda de ésta se efectúa la siguiente expansión:

```
(struct patos *)locind(locind(pani->pave,ptipos[i],obase)->patos,
                                                ptipos[j],obase)->nombre
```

donde ptipos[i] y ptipos[j] son los indices del arreglo ptipos generado por el preprocesador y que identifica el la clase a la que petenece el OP al que se hace referencia.

Ejemplo 3:

Teniendo la declaración dada en ejemplo 1, sí encontramos la siguiente instrucción:

```
pperro1->edad=25;
```

El preprocesador la expande a :

```
(struct perro *)locind(pperro1,ptipos[i],obase)->edad = 25;
```

y la instrucción:

```
pperro1->hijo->edad = 25;
```

es expandida a:

```
(struct perro *)
locind(
        locind(pperro1,ptipos[i],obase)->hijo,
        ptipos[j],obase)->edad
        = 25;
```

La función locind es una función del MOP que regresa un apuntador al objeto referenciado, en su desempeño puede encontrarse con dos posibilidades, la primera es que el OP referenciado se encuentre en memoria primaria , en cuyo caso regresará la dirección donde se encuentra. El segundo caso será cuando el OP no se encuentre en memoria primaria, en este caso la función efectúa la lectura y asignación de memoria, regresando ésta última (como un apuntador).

A la vez que el preprocesador va analizando la expresión para verificar que sea correcta, se va generando la tabla de información referente a los OPs. La forma de verificar si una instrucción es correcta, es por medio del sigiente autómata.

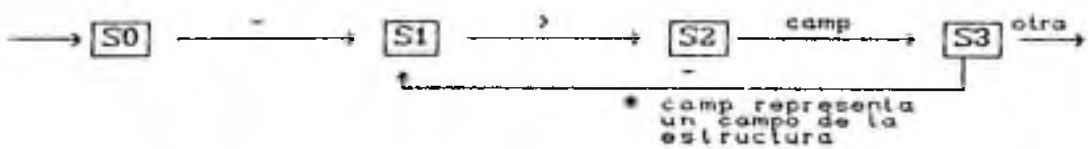


Fig IV.3 Autómata que verifica la sintaxis de una instrucción con idirecciones.

IV.2.4 INSTRUCCIONES QUE CONTIENEN LLAMADAS A FUNCIONES DEL MOP

Las instrucciones que manejan llamadas a funciones del MOP necesitan procesamiento debido principalmente a la transparencia que se desea brindar al usuario. Las operaciones efectuadas con las instrucciones que contienen llamadas a funciones del MOP son:

- Obtener información del medio ambiente en el cual se encuentra la llamada, para obtener la información necesaria y poder generar los parámetros que le hacen falta a la función.

- Hacer el "cast" al valor que regresa la función, tomando el tipo de la variable de tipo apuntador que recibe el valor.

Por ejemplo:

supongamos que tememos la siguiente declaración:

```
persistent struct vampiros{
        char clase [20];
        char origen[30];
        struct familia *apfam;
};
persistent struct vampiros *apvamp;
```

y el siguiente fragmento de código:

```
.
obasep = abrebase("mamiferos","felix","xilef",MODOLE);
.
apvamp=buasca("derringue",obasep);
.
.
```

El preprocesador expande la llamada a la función busca de la siguiente manera:

Verifica a que clase pertenece la variable que resibe el

valor que la función regresa, y con éste hace el "cast" al apuntador que regresa la función, además inidica el tipo a que pertenece el apuntador y pasa como parámetro su descripción. de esta manera tenemos para la llamada a busca finalmente:

```
.
.
apvamp=(struct vampiros *)buasca(ptipos[j],
                                 "derringue",obasep);
.
```

No todas las funciones del Manejador de Objetos son expandidas, una descripción detallada de su implantación se da en el capítulo V, y una descripción de su uso en el manual de usuario, el cual está disponible en la sección de computación, de este centro.

CAPITULO V

MANEJADOR DE OBJETOS PERSISTENTES (MOP)

En las secciones de este capítulo se describen las tareas efectuadas por el MOP, uno de los módulos que componen el sistema que se presenta en esta tesis. La primera sección lista los objetivos que debe cumplir el MOP. La segunda sección explica el mecanismo de correspondencia entre los dos tipos de memoria. La tercera sección describe el mecanismo de validación de la información. La cuarta sección describe el mecanismo de transacción. Y en la última sección de este capítulo se describe con seudocódigo el mecanismo de transacción.

V.1 MANEJADOR DE OBJETOS PERSISTENTES

En el sistema que se describe, el MOP es el módulo que se encarga de diversas tareas referentes al manejo de los OPs. Para que éstas sean efectuadas correctamente, el MOP debe de cumplir con ciertos objetivos, los cuales se listan a continuación:

i. Efectuar las operaciones para hacer la correspondencia entre la memoria primaria y la memoria secundaria.

ii. Conservar la integridad de la información, confrontando las declaraciones obtenidadas del programa de aplicación, con las declaraciones residentes en memoria secundaria para objetos persistentes.

iii. Proporcionar un mecanismo que asegure eficiencia e integridad de la información contenida en memoria primaria.

V.2 MECANISMO DE CORRESPONDENCIA ENTRE LA MEMORIA PRIMARIA Y LA MEMORIA SECUNDARIA

Los OPs son unidades que ocupan un espacio en memoria secundaria, en nuestro caso nos referimos a éste por medio de direcciones (lógicas), así también, cuando un OP reside en memoria primaria emplea un espacio en la memoria principal. Ya que el sistema permite el uso de la información contenida en los OPs sin importar que se encuentren en memoria primaria o secundaria es necesaro un mecanismo de correspondencia entre ambas memorias.

El mecanismo que efectúa la correspondencia entre la memoria primaria y la memoria secundaria toma información de dos fuentes. La primera es el preprocesador, primer módulo que compone el sistema y que se explicó en el capítulo IV. La segunda es la información generada por el propio MOP en procesamientos anteriores en caso de que hayan existido. La información obtenida de ambas fuentes es almacendada en estructuras de datos que guarda información útil para la administración de los objetos base, y el contenido propio de los OPs. A continuación se explican algunos de los componentes importantes de las estructuras, y su utilidad en el mecanismo de transacción. La declaración completa se puede encontrar en el archivo nombrado "thead.h". Las estructuras descritas son:

i. Identificador de Objeto Persistente PID.

ii. Estructura de un Objeto Persistente.

iii. Directorio de Usuarios.

iv. Directorio de Información.

v. Indice de Objetos Base.

vi. Formato de OP en Memoria Primaria.

vii. Formato de OP en Memoria Secundaria.

viii. Indice de Clase.

V.2.1 Identificador de Objeto Persistente (PID)

Cada OP recibe al ser creado un identificador único nombrado PID, que conservará durante toda su vida. Dicho identificador es un número que está realcionado con la dirección (lógica) del OP en memoria secundaria.

El PID es un número de tipo "long" en C, que se forma de la siguiente manera:

La unidad mínima de memoria secundaria que puede accesar es un sector y un conjunto de sectores forman un bloque. Un pid se forma considerando el bloque en el que se escribe el OP y su localización dentro del mismo bloque. Esta información nos sirve para localizar un objeto cuando no está en memoria primaria.

Fig V.1 Significado de un PID

V.2.2 Estructura de un Objeto Persistente

Los objetos persistentes tienen dos formatos: uno cuando están en memoria primaria y otro cuando están en memoria secundaria. La

diferencia básica entre estos dos formatos es que en memoria primaria se tienen ambos: verdaderos apuntadores del lenguaje "C" y direcciones lógicas PIDs mientras que el formato en memoria secundaria sólo contiene los segundos.

Un objeto persistente es cualquier variable declarada con alguno de los tipos validos en "C" y que haya sido declarada con propiedad persistente (para enterar al manejador de objetos persistentes MOP), por lo que estos pueden ser:

i).- Cualquier tipo básico (enteros, reales, caracter).

ii).- Vectores de cualquiera de los tipos básicos.

iii).- Estructuras que contengan cualquier número de campos de los tipos i y ii.

Ya que el MOP es el encargado de manejar los OP este necesita conocer la composición e información de todos los OPs. La primera es generada por el preprocesador descrito en el capítulo IV, y la segunda es proporcionada por el usuario. Si vemos a un OP como una unidad dividida en dos partes, a la primera le llamamos encabezado (contiene datos útiles al MOP), y a la segunda le llamamos cuerpo (contiene la información del usuario), tenemos que:

La estructura del encabezado es la siguiente:

PID	IDEN. DE TIPO	BANDERA DE OPCION	TAMANO EN BYTES	NUMERO DE APUNTADORES	APUNTA-DORES
1	2	3	4	5	6

Fig V.2 Estructura del Encabezado

Significado y utilización de los campos:

1o. PID (entero de tipo long). Sirve al manejador de objetos persistentes para identificar unívocamente un

objeto, y con éste efectuar las operaciones indicadas por la transacción al objeto.

2o. Identificador de tipo, su valor es de acuerdo a la convención adoptada en la implantación. Sirve al MOP para identificar el tipo del OP (entero, caracter etc.) La convencion adoptada es:

100 entero.
110 arreglo de enteros.
200 caracter.
210 arreglo de caracteres.
300 estructura.
310 arreglo de estructuras.

3o. Bandera de opción, su valor es de acuerdo a la siguiente convención y lo utiliza el MOP para efectuar el proceso de compromiso. La convención tomada en la implantación es la siguiente:

- creado _____ 1.
- modificado ____ 2.
- borrado _____ 3.
- leido _____ 4.

4o. Tamaño en Bytes, su información es el tamaño del objeto, éste es utilizado por el MOP cuando solicita memoria para el OP, tanto para memoria primaria como para memoria secundaria.

5o. Número de apuntadores, su significado es el número de objetos, que se pueden referir desde este OP.

6o. Apuntadores, este campo es en realidad un arreglo de enteros que contiene los PID's de los objetos descendientes desde este OP.

V.2.3. Directorio de Usuarios

El directorio de usuarios está contenido en una estructura arborecente (árbol binario). Esta estructura contiene los identificadores de los usuarios, las claves por las cuales se puede hacer accesible su información, y un apuntador al directorio de objetos base creados por el usuario. Esta estructura está ordenada por el identificador del usuario.

IDENTIFICADOR DEL USUARIO	CLAVE	APUNTADOR AL DIRECTORIO DE INFORMACION

Fig V.3 Estructura del Directorio de Usuarios

V.2.4. Directorio de Informacion

Cada usuario registrado tiene derecho a crear, modificar y consultar información. Tal información puede ser accesada mediante una estructura arborecente (árbol binario) que contiene el directorio de identificadores de los objetos base creados Esta estructura es ordenada por el identificador que el usuario de al objeto base.

IDENTIFICADOR DEL OBJETO BASE	APUNTADOR AL INDICE DE OBJETOS	APUNTADOR A LISTA DE OBJETOS RESIDENTES EN MEMORIA PRIMARIA

Fig V.4 Estructura del Directorio de Objetos Base

V.2.5 Indice del Objeto Base

En los sistemas de cómputo es deseable un tiempo de respuesta corto por lo que la información debe estar organizada para optimizar el tiempo de acceso.

De las diversas formas de organización que existen (índice, función de hash, etc.) en el presente trabajo se adoptó la de crear un índice. Si el tipo del objeto es carácter, entero o arreglo de estos (básicos), se oredena por la llave proporcionada por el usuario. De otra manera, se ordena por el identificador de la clase. El índice creado tiene una estructura de árbol de BAYER[1] con el formato que se muestra a continuación:

Fig V.5 Estructura del Indice del Objeto base

V.2.6 Formato de los OPs Residentes en Memoria Primaria

Los objetos que han sido creados leídos o modificados durante la transacción tienen que estar en memoria primaria para evitar accesos no necesarios a memoria secundaria.

Los OPs se guardan en listas ligadas independientes para cada objeto base y ordenadas por el PID del OP. Esta estructura es implantada teniendo en cuenta:

i).- Que el acceso a memoria primaria es más rápido que el acceso a memoria secudaria.

ii).- Todo objeto persistente creado, modificado o leído, debe poder ser accesado sin tener que referirse a memoria secundaria más de una ocasión.

[1] Véase la referencia 21.

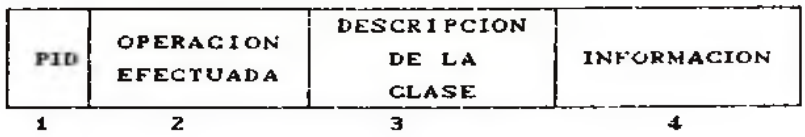


Fig V.6 Estructura de Objetos Persistentes en Memoria Primaria

Los contenidos de los campos de la figura V.6 son.

1o.- El identificador del OP (PID).

2o.- Indicador de la operación que se efectuó con el OP (leído, creado o modificado).

3o.- Un apuntador a una estructura que contiene la descripción cualitativa del OP.

4o.- Apuntador a la información que contiene el OP.

V.2.7 Formato de los OPs Residentes en Memoria Secundaria

Los OPs residentes en memoria secundaria no son otra sino cadenas de caracteres, sin embargo, pueden ser consebidos como un conjunto de caracteres atados a una estructura la cual contiene su descripción, como se muestra en la siguiente figura V.7.

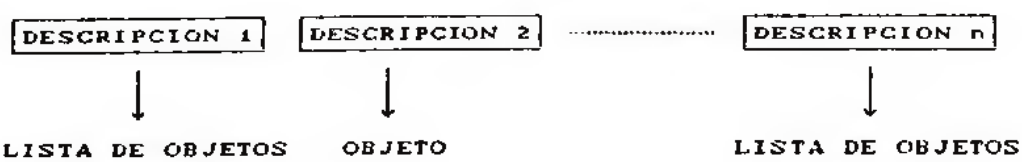
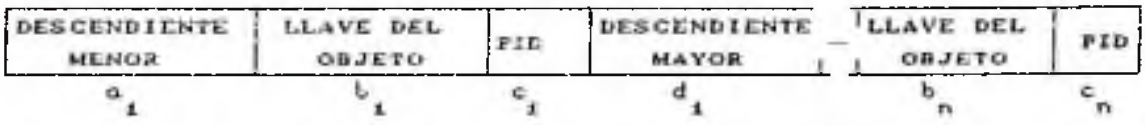


Fig V.7 Concepción de OP Residentes en Memoria Secundaria

V.2.8 Indice de Clases

Así como la información de tipo básico puede ser accesada por medio de un índice, la información de una clase es accesible a través de un índice de clase. Este índice está implantado como un árbol de BAYER, y su contenido es el siguiente:



CADA NODO DE LA ESTRUCTURA CONTIENE
n ELEMENTOS, SIENDO n EL ORDEN DEL ARBOL

Fig V.8 Estructura del Indice de Clases

La información contenida y el significado de los campos de la fugura **V.8** son los siguientes:

a_i.- Apuntador al descendiente menor (con respecto a la llave por la que se ordena la estructura).

b_i.- LLave por la que está ordenada la estructura.

c_i.- PID del objeto identificado por la llave contenida en el campo número dos del elemento del nodo.

d_i.- Apuntador al descendiente mayor (con respecto a la llave por la que se ordena la estructura).

La figura V.9 muestra como administra la información que el usuario desea conservar por medio de las estructuras de datos que crea el manejador de objetos persistentes. El manejador de OPs modifica y crea esta información para efectuar su objetivo.

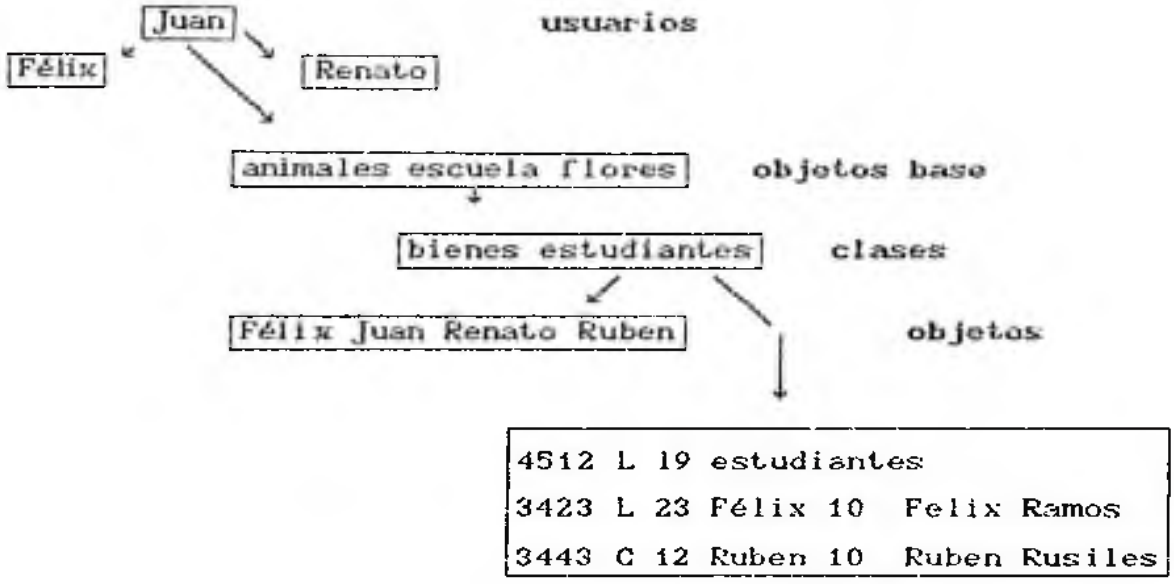


Fig V.9 Ejemplo de Relación entre Estructuras de Datos creadas por el manejador de OPs y la información del usuario MOP

V.3 ALMACENAMIENTO PERSISTENTE

El objetivo de este inciso es mostrar las ideas seguidas para implantar la propiedad de persistencia al lenguaje "C".

La forma en que los lenguajes organizan la memoria principal es compleja y la forma de implantación es material de un tema que conviene tratar aparte. En el mismo caso, y con cierta complejidad, se encuentra la oganización de memoria secundaria. Si consideramaos como punto de partida que el lenguaje debe sufrir pocos cambios, y que la adición de persistencia es por medio de funciones que se encargan de hacer transparente el manejo de la memoria secundaria, una forma de alcanzar nuestro propósito es dejar sin modificar las funciones de entrada salida del lenguaje, y abocarnos a la tarea de dar una organización a la información para obtener los resultados deseados.

V.3.1 Información que se Necesita Conservar

Algunos lenguajes dan la facilidad de hacer referencia a la información contenida en la memoria primaria por medio de:

i.- Nombres de variables declaradas de algún tipo que maneja el lenguaje.

ii.- Variables de tipo apuntador, que pueden hacer referencia a las localidades asignadas a los identificadores de las variables o bién a espacios de memoria definidos mediante una asignación.

Estas últimas hacen referencia a las localidades de memoria que contienen los datos. Dichos datos se encuentran en una área denominada "heap" [ref 12]. Esta área es administrada por el lenguaje, y en ella tenemos la información que el proceso necesita y la memoria de que dispone. Si pudieramos distinguir

la información que deseamos conservar de la que no se desea conservar, sería fácil seleccionar y guardar la información contenida en este "heap". Esto sólo es posible si modificamos la estructura del "heap" (el lenguaje organiza el "heap"), lo cual no está contemplado por el principio de hacer pocos cambios al lenguaje. Por lo que identificaremos la información persistente de la no peristente, por medio de estructuras generadas por el MOP y el preprocesador y, que hacen referencia a la información que se desea conservar (OPs) (toda contenida en el "heap").

V.3.2 Consistencia en los datos

La memoria persistente trae consigo problemas para verificar la consistencia en el tipo de información. Los lenguajes consideran que cualquier variable encontrada en un proceso pertence a un tipo declarado en el lenguaje, o a algún tipo definido en el proceso; esto no sucede cuando tenemos la propiedad de almacenamiento persistente, puesto que los datos pueden ser emmpleados por procesos diferentes a los que los crearon. Por esta razón es necesaria la implantación de un mecanismo de consistencia en los datos.

V.3.3. Método de Consistencia

Puesto que el compilador únicamente puede verificar la consistencia de los datos que le son declarados, es necesario conservar información de los tipos definidos con propiedad persistente, para confrontar con ésta la consistencia de los tipos en diferentes procesos. La implantación presente maneja todos los tipos del lenguaje "C" excepto el de "union", por tanto, la cantidad de posibles tipos es infinita. En un proceso dado, una variable declarada con propiedad persistente está definida por la declaración dada dentro del proceso; pero si ya existia, su definición debe de coincidir con la definición anterior.

Esto trae consigo una concepción de un OP diferente de la de un objeto que no posee la propiedad de persistencia. Un objeto persistente tiene una declaración generada cuando es creado por primera vez, y está se conserva durante su tiempo de vida.

La concepción de un OP dada en la sección **V.2** permite verificar la consistencia en declaraciones de tipo persistente, además de proporcionar información útil en la implantación de procedimientos que manejan este tipo de datos, ya que la información no será duplicada, sino que se trabajará con una sóla información.

V.4 MECANISMO DE TRANSACCION

La función de este procedimiento es reflejar en memoria secundaria las modificaciones que hubiesen sido efectuadas durante la transacción al Objeto Base u Objetos Bases involucrados. El mecanismo implantado asegura a las transacciones:

i).- Eficiencia.

ii).- Atomicidad

La descripción de estos puntos es la siguiente:

V.4.1. Eficiencia en las Transacciones

El término eficiencia en las transacciones está en función del costo de una transacción y de las limitantes físicas del sistema en el que se trabaja; el análisis es el siguiente:

V.4.1.1. Costo de las Transacciones

En transacciones que modifican un cierto número "n" de registros, donde "n" es mucho menor que el número total de registros

contenidos en el objeto base, el precio de lectura y escritura de todo el objeto base puede ser muy alto en comparación con la tarea efectuada, dependiendo del mecanismo de transacción implantado.

V.4.1.2. Limitantes Físicas del Sistema

Los sistemas reales tienen limitantes físicas de memoria primaria, las cuales deben de ser tomadas en cuenta para el desarrollo de cualquier sistema. En sistemas que requieren manejo de mucha información se hace más evidente la necesidad de establecer un mecanismo que tome en cuenta estas limitantes, esto es, se deben mantener en memoria primaria únicamente los datos necesarios, ya que es posible la existencia de una base de datos que sea más grande que la memoria disponible.

Del análisis anterior es evidente la necesidad de un mecanismo de transacción que efectúe únicamente la lectura y escritura de los OPs que se indiquen.

V.4.2 Transacciones Atómicas

Un punto que requiere consideración es el de conservar la base de datos consistente cuando se efectúa alguna transacción.

Al efectuarse una transacción sobre la base de datos de un objeto base, éste puede pasar por una serie de estados inconsistentes; pero al final debe conservar su consistencia y estos estados deben ser transparentes al usuario, el cual deberá ver la transacción simplemente, como efectuada o no efectuada, esto es lo que se conoce como transacción átomica.

Un objeto base puede pasar de un estado consistente a otro inconsistente por dos causas principales que son:

i.- Falla del sistema.

ii.- Falla del programa.

En el presente trabajo el mecanismo de transacción es el que sigue:

Existe una estructura que contiene los identificadores de los objetos base que han sido modificados durante la transacción. A partir de esta estructura se visitan los OPs persistentes residentes en memoria primaria y que hayan sido modificados o creados. OPs son alcanzables independientemente para cada objeto base modificado. De estos OPs residentes en memoria primaria, son escritos en un archivo temporal únicamente los que fueron creados y modificados. Una vez efectuada esta tarea, se escriben los índices de las clases modificadas, y por último se escribe en elarchivo que contiene los objetos base modificados, el contenido de la lista de Objetos Base modificados. A continuación, se modifican los Objetos Base que contiene la lista de objetos base modificados, se cambian los índices de temporales a definitivos y se borra el identificador del objeto base de la lista.

Las tareas anteriores se efectuan para cada Objeto Base que esté contenido en la lista de Objetos Base modificados.

V.4.2.1 Condiciones de Falla

Este punto es importante porque existen dos posibles condiciones de fracaso durante la transacción, ellas son:

i.- No poder crear un archivo temporal.

ii.- Falla del sistema debido a causas externas.

La primera provoca que no se refleje la transacción en memoria secundaria y es causa de frustración en la transacción. La falla del sistema debida a causas externas tiene dos casos interesantes que son:

i.- Cuando se está escribiendo en el archivo temporal.

ii.- Cuando se está actualizando el objeto base en memoria

secundaria.

La primera es causa de frustración y el archivo original no se afecta.

La segunda deja el objeto base en un estado inconsistente y recuperar la consistencia es tarea de algún procedimiento.

El seudocódigo siguiente muestra el mecanismo implantado y seguido por todas las transacciones que se efectúan.

```
Transacción()
        begin
          inicializa()
          mientras exista condición de transación
          begin
            casos del MOP
            abrebase(): si abrebase() falla
                        begin
                          genera mensaje "falla en abrebase"
                          genera condición de falla
                        end
            inserta() : si inserta() falla
                        begin
                          genera mensaje "falla en inserta"
                          genera condición de falla
                        end
            busca()   : si busca() falla
                        begin
                          genera mensaje "falla en busca"
                          genera condición de falla
                        end
        compromiso()  : si compromiso() falla
                        begin
                          genera mensaje "falla en compromiso"
                          genera condición de falla
                        end
            end casos del MOP
            regresa
        end transacción
```

Del seudocódigo anterior las funciones indicadas con negritas son funciones del **MOP**, y serán descritas en la siguiente sección de este capítulo. Adelantándonos un poco, diremos que estas pertenecen a las funciones disponibles para el usuario y con ellas son manejados los **OPs**. De estas funciones, la que se encarga de la atomicidad de la transacción es la llamada compromiso; misma que a continuación se describe a manera de seudocódigo.

```
compromiso()
begin
  si existen objetos base modificados o creados
  begin
    para cada objeto base creado o modificado
    begin
      si tiene el objeto base OP en memoria primaria
      begin
        crea un temporal y guarda en éste los OP creados
          o modificados
        crea un temporal para el índice del objeto base
        crea un temporal para cada índice de clase accesado
        guarda los identificadores de los índices de clase
          modificados.
      end
      genera temporales para los objetos base modificados
    end
    guarda la lista de objetos base modificados
    actualiza físicamente la base de datos del objeto base
    cambiar temporales a definitivos
    cambiar temporales de directorios a definitivos
    eliminar los objetos base borrados
    se borra la lista de actualización
    regresa exito
  end
end compromiso
```

V.5 FUNCIONES IMPLANTADAS

El objetivo de las funciones que constituyen el manejador de OPs es, como se dijo anteriormente, hacer transparente al usuario el manejo de memoria secundaria. Estas funciones consultan y modifican el contenido de las estructuras definidas anteriormente, y son la única adición al lenguaje "C" para poder hacer con él programación persistente.

Los objetivos de las funciones del manejador de objetos son:

i.- Hacer transparente el mapeo de memoria primaria a memoria secundaria (memoria secundaria a memoria primaria).

ii.- Garantizar lo que en base de datos se llama atomicidad de transacción.

iii.- Proporcionar entrada a funciones generales para que el usuario pueda efectuar cualquiera de las operaciones básicas (inserción, busqueda, etc.) con los OP.

Las funciones implantadas pueden ser divididas de acuerdo al tipo de OPs con que trabajan, y la tarea que desempeñan en funciones para:

i.- Manejo de Objetos Base.

ii.- Manejo de Objetos.

iii.- Para efectuar el compromiso de la transacción.

A continuación se describen los procedimientos que componen cada uno de estos tres tipos de funciones que se mencionaron. Para cada procedimiento se indica(n):

i.- El Nombre de la función.

ii.- La sintaxis de la llamada a la función.

iii.- La forma en que se obtienen los parámetros.

iv.- Un seudocódigo que describe la forma en que está implantada la función.

v.- Las estructuras con que interactúa.

V.5.1 Funciones para el manejo de Objetos Base

En este primer conjunto de funciones, se describen los procedimientos por medio de los cuales, el usuario puede manipular, crear, accesar, eliminar o modificar los Objetos Base. Las funciones que componen este primer conjunto son:

- abrebase();
- bbase();
- cierrabase();

V.5.1.1. *abrebase*

La sintaxis de la función es la siguiente:

```
OBJETOB
*abrebase( NombreBase, Contraseña, Modo)
char *NombreBase;     /* identificador del objeto base a   */
                      /* abrir                             */
char *Contraseña      /* llave del objeto que se intenta   */
                      /* abrir                             */
int Modo;             /* modo en el que se abre o se       */
                      /*  crea el objeto objeto base       */
```

Forma de obtener los parámetros:

NombreBase: parámetro proporcionado explícitamente por el usuario en el programa de aplicación escrito en lenguaje C_persistente.

Contraseña: parámetro proporcionado explícitamente por el

usuario en el programa de aplicación, escrito en lenguaje C_persistente.

Modo: parámetro proporcionado explícitamente por el usuario en el programa de aplicación, escrito en lenguaje C_persistente.

El objetivo de esta función es abrir o crear el Objeto Base nombrado en el primer parámetro y con los permisos indicados en el parámetro Modo. El siguiente seusdocódigo muestra las deciciones y acciones que toma la función:

```
abrebase
begin
  Si el modo en que se quiere abrir el objeto base es lectura
  lectura escritura y ya existe otro objeto base abierto
  en ese modo
  empieza
     genera mensaje que indique el hecho
     regresa un apuntador nullo
  end
  Si existe la base de datos
   si está abierta para escritura
      regresa apuntador nulo
    si la Contraseña es incorrecta
      regresa apuntador nulo
  si no existe la base de datos
    manda mensaje base no existe.
  si no se pudo crear
    regresa apuntador nulo
  regresa apuntador útil.
end de Abrebase.
```

La apertura de un objeto base es efectuada con los parámetros que le son proporcionados de la siguiente manera:

i.- Si el modo en el que se quiere abrir el objeto base es

escritura o lectura_escritura, se verifica que no exista otro Objeto Base abierto en esos modos, esto es, sólo podrá existir un Objeto Base abierto en uno de estos dos modos.

ii.- Verifica si existe el Objeto Base, si es el caso y se puede accesar se regresa el apuntador al Objeto Base.

iii.- Si no existe el Objeto Base se intenta crear y si es exitosa la creación, se regresa el apuntador al Objeto Base creado.

Estructuras que intervienen en abrebase

i. Directorio de usuarios.
ii. Directorio de objetos base del usuario.

V.5.1.2. *bbase*

La sintaxis de la función es la siguiente:

```
bbase( Usuario, Nombrebase, Contraseña)
char *Usuario;        /* identificador del usuario que     */
                      /* quiere borrar el Objeto Base      */
char *Nombase;        /* identificador del objeto base     */
char Contraseña       /* contraseña del usuario            */
```

Forma en que se obtiene la información:

Usuario: Parámetro proprorcionado explícitamente para identificar al usuario que intenta borrar el objeto base, éste debe ser el propietario.

Nombase: Parámetro proporcionado por el usuario explícitamente. Identifica el objeto base a ser eliminado.

Contraseña: Clave de acceso del usuario a la información.

Función primitiva del MOP, sus objetivos son:

i. Eliminar un objeto base del directorio de Objetos Bases del usuario.

ii. Liberar la memoria útlizada por el objeto base. Para efectuar esta tarea se necesita saber el identificador del objeto base.

El seudocódigo siguiente muestra la forma en que la función efectúa su objetivo:

```
bbase()
empieza
  busca el identificador del objeto base en el directorio de
     objetos base del usuario.
  si el objeto fue encontrado
  empieza
    elimina el identificador del directorio de objeto bases del
      usuario
    copia el identifcador del objeto base en la lista de objetos
      base borrados
  termina
  si el objeto no fue encontrado
    genera mensaje "objeto base no existe"
  regresa
termina bbase
```

Las estructuras con que interactúa la función son:

i.- Directorio de objetos base del usuario.

ii.- Lista de identificadores de objetos base borrados durante la transacción.

La función elimina el identificador del objeto base en la estructura residente en memoria primaria; pero el objeto base es eliminado de memoria secundaria sólo cuando se efectúe la función de compromiso.

V.5.2 Funciones para el Manejo de OP

En este inciso se describen las funciones con las cuales el usuario puede manejar los objetos persistentes.

V.5.2.1 *busca*

La sintaxis de la función busca es la siguiente:

```
char
*busca( Ptipo, Llave, Objetobase )
TIPOS *Ptipo;          /* apuntador a la descripción del      */
                       /* objeto a insertar                   */
char *Llave;           /* llave por la que se busca el objeto */
OBJETOB *Objetobase    /* apuntador al objeto donde se busca  */
```

Forma de obtener los parámetros

Ptipo: este parámetro es generado por el preprocesador a partir del identificador que recibe el valor que la función regresa y la tabla que genera el preprocesador.

Llave: parámetro proporcionado explícitamente por el usuario en el código del programa escrito en lenguaje C_ persistente.

Objetobase: parámetro proporcionado por el usuario explícitamente, en el código del programa de aplicación escritó en lenguaje C_persistente, es obtenido de la función abrebase.

Esta función es de tipo apuntador. Su objetivo es buscar en el objeto base especificado en el parámetro Objetobase, el objeto que tenga llave igual a la indicada en el parámetro LLave. La función regresa un apuntador al objeto, si éste existe, y un apuntador nulo en caso contrario. El siguiente seudocódigo describe la forma en que opera la función.

```
busca
begin
  Si objeto base no está abierto
    genera mensaje
    regresa apuntador nulo
  si existen objetos en memoria primaria
    si esta el objeto en memoria primaria
      regresa apuntador al objeto
  si existe la llave en el índice
  begin
    lee el objeto de memoria secundaria
    insertalo en la lista de objetos en memoria primaria
    regresa apuntador al objeto
  end
  regresa apuntador nulo
end de busca
```

La función realiza su objetivo en base a los parámetros que le son suministrados de la siguiente manera:

i.- Verifica si el parámetro *objeto base*, de tipo apuntador es diferente de nulo.

ii.- Inspeccionando la información referenciada por el parámetro *ptipo* identifica el tipo del **OP** buscar (básico o estructura), y decide el paso a seguir.

iii.- Se efectúa la busqueda del Op en memoria primaria. Si se localiza el objeto se regresa su apuntador.

iv.- Si el **OP** no fue localizado en memoria primaria busca en memoria secundaria; si existe, se inserta en la lista de objetos residentes en memoria primaria y se regresa el apuntador a su localización.

v.- Si el **OP** no es localizado se regresa un apuntador nulo.

Estructuras que intervienen en busca

i.- Estructura generada por el preprocesador.

ii.- Lista de OPs residentes en memoria primaria.

iii.- Objetos Base.

iv.- Indice de objetos base.

Observaciones

i. El usuario no tiene que hacer declaración de esta función.

ii. El preprocesador debe de hacer el "cast" al tipo que regresa la función para no tener problemas de incompatibilidad de tipos.

V.5.2.2 *Inserta*

La sintaxis de la función inserta es la siguiente:

```
inserta( Llave, Objetobase, Pinform, Tinform)
char *Llave;                /* llave por la que se identifica*/
                            /* el objeto                     */
OBJETOB * Objetobase;       /* objetobjeto base en el que se
                                        inserta               */
TIPOS * Tinfor;             /* apuntador a la descripción    */
                            /* del objeto a insertar         */
char *Pinform;              /* apuntador a la información a  */
                            /* insertar.                     */
```

Forma de obtener los parámetros

Llave: este parámetro es proporcionado por el usuario en forma de apuntador a cadena de caracteres

Objetobase: parámetro de tipo apuntador que es obtenido del valor que regresa la función abrebase, del conjunto de

primitivas del manejador de objetos, este parámetro es proporcionado explícitamente por el usuario.

Tinfor: parámetro de tipo apuntador que contiene la dirección para accesar información, este parámetro es generado por el preporcesador tomando la lista de identificadores de objetos persisistentes generada (véase capítulo IV).

Pinform: parámetro de tipo apuntador a caracter que contiene la información a insertar. Este parámetro se convierte al tipo caracter para proporcionar generalidad a la función.

Esta función es de tipo entero. Su objetivo es, insertar un **OP** en el objeto base especificado por el parámetro *Objetobase*. El objeto es insertado con la llave dada en el parámetro LLave.

La estructura de los elementos generados por el preprocesador tiene el siguiente formato:

IDENTIFICADOR	TAMANO DEL OBJETO	NUMERO DE CAMPOS	TIPO DE LOS CAMPOS DEL OBJETO	IDENTIFICADORES DE LOS CAMPOS

Fig. V.9 Estructura generada por el preprocesador

Teniendo la información generada por el preprocesador la función es dividida en dos partes:

1).- Identificar.
2).- Insertar.

Identificar

Este proceso verifica el tipo del objeto para saber si pertenece a una clase, si es el caso, se verifica que sean compatibles, si no lo son, se genera un mensaje y se aborta la transacción. En el caso de declaraciones que pertenecen a una clase, se dice que son declaraciones equivalentes si el identificador de

la clase es el mismo y todos los campos tienen el mismo identificador y éstos son del mismo tipo. Esta comparación se hace entre la clase que existe en memoria secundaria y la declarada en la transacción, sólo en el caso que exista la clase.

Insertar

Este proceso inserta el OP en la lista de OPs residentes en memoria primaria .

La inserción en el archivo físico (memoria secundaria) se efectuará al llamar explícitamente la función de compromiso (commit).

Existe una lista de OPs modificados y creados para cada objeto base, esto es, para cada objeto base que durante la transacción haya sido abierto en algún modo que modifique la información, existe una lista que contiene únicamente los OPs que hayan sido creados y/o modificados. En el caso que no exista la clase en la que desamos insertar el OP, se efectúa el siguiente procedimiento:

1).- Insertar el identificador de la clase en el índice del objeto base.

2).- Pedir a través del encabezado del objeto base un bloque inicial e iniciarlo.

3).- Crear un archivo que contendrá el índice de la clase creada.

4).- Pedir un identificador (PID) para el encabezado del bloque. Insertar el objeto en la lista que contiene los objetos modificados, creados o borrados del Objeto Base.

El seudocódigo siguiente describe las decisiones que se toman en la función:

```
inserta
  si objeto base no está abierto para escritura
    no se efectúa ninguna tarea y se genera un mensaje
  si la clase a la que el OP a insertar existe
    si los tipos no son compatibles
      no se efectúa ninguna tarea y se genera un mensaje
    si la llave por la que se inserta existe en la clase
      obten el PID
      modifica información.
      bandera de operación = modificación
    bandera de operación = creación.
    inserta el OP en la lista de OPs residentes en memoria
    primaria
  si no existe la clase
    crea la clase (con toda la información administrativa
                                                  necesaria)
    bandera de operación = creación
  inserta OP en la lista de residentes en memoria primaria
                                    del objeto base.
fin de inserta.
```

Para saber si el OP que deseamos insertar está en memoria primaria, se busca con la llave en la lista de OPs modificados y creados del objeto base, en caso contrario se verificara si la clase existe para buscar el OP en los elementos que ésta contiene.

Para saber si la clase existe, se hace la busqueda del identificador de la clase en la estructura del índice, la cual está apuntada desde la base abierta para escritura y por el parámetro pinobjeto base.

Para saber si un tipo es compatible con otro que ya existe, se hace la comparación de la información contenida en la lista generada por el preprocesador y la contenida en el nodo del identificador de la clase, estó se tiene que hacer una sola vez y antes de realizar la inserción.

Para saber si la llave por la que se quiere insertar ya fue empleada, lo cual indica que el OP ya existia, se la busca en el índice de OPs del objeto base.

Cuando se inserta un objeto cuya clase no exista, ésta se crea y se dice que pertenece al objeto base donde se inserta el OP.

Estructuras que intervienen en inserta.

1).- Lista generada por el preprocesador.
2).- Lista de objetos residentes en memoria primaria.
3).- Apuntador al objeto base.

V.5.2.3 *borraob*

La sintaxis de la función es la siguiente:

```
char
*borraob( Llave,Clase,Objetobase)
char * Llave;          /* llave que tiene el objeto a borrar */
TIPOS *Clase;          /* apuntador a la descripción del     */
                       /* objeto a borrar.                   */
OBJETOB *Objetobase;
                       /* apuntador al objetobjeto del que se
                       /* borra el objeto.                   */
```

Forma de obtener los parámetros

Llave: parámetro proporcionado explícitamente por el *Usuario*, identifica el objeto a borrar.

Clase: parámetro generado por el preprocesador a partir del tipo del objeto que recibe el valor de la función.

Objetobase: apuntador al objeto base del que se quiere eliminar el

Observaciones:

i. El parámetro clase es generado por el preprocesador a partir del tipo de la variable que recibe el valor, la cual debe ser de tipo peristente.

ii. Puesto que la función regresa un apuntador a caracter, el preprocesador, genera la conversión al tipo de la variable que recibe el valor de la función.

Función del conjunto de primitivas del MOP, esta función regresa un apuntador a caracter, su objetivo es eliminar de un objeto base, el objeto identificado por la llave dada en el parámetro llave.

V.5.3 Funciones Para Efectuar El Compromiso

V.5.3.1 *Compromiso*

La sintaxis de esta función es:

```
int commit();
```

ya que en el lenguaje C todas las funciones son de tipo entero por default, esta declaración puede ser omitida

Los objetivos de esta función son:

i.- Reflejar en memoria secundaria el estado de la información en el momento de su llamado.

ii.- Asegurar la atomicidad de la transacción.

Lo anterior puede ser mejor entendido con el siguiente seudocódigo.

```
compromiso()
empieza
  si  no existe base abierta para escritura
    regresa
  si  no tiene elementos creados o modificados
    regresa
  si  crea  temporal  para  elementos  creados  y/o  modificados
                falla
    genera mensaje
    regresa
  inserta elementos en el archivo creado
  si  crea  temporal de indice  falla
    genera mensaje
    regresa
  guarda información del indice en el archivo creado
  si  genera temporales para las clases falla
    genera mensaje
    regresa
  si crea archivo que contenga las clases modificadas  falla
    genera mensaje
    regresa
  si  falla  en  la  escritura  al  archivo  generado  de  las  clases
      modificadas
    genera mensaje
    regresa
  prende bandera a archivo.
  actualiza el archivo con lista residente en memoria.
  mover temporales a definitivos
  apagar la bandera al archivo.
  regresa exito
termina compromiso.
```

Estructuras con que interactúa la función:

1.-Lista de objetos base abiertos para escritura o escritura lectura durante la transación.

2.-Lista de OPs residentes en memoria primaria de cada objeto

base que cumpla con i.

La información que emplea esta función está contenida en variables de tipo global, que hacen referencia a estructuras de datos que contienen:

i.- Lista de objetos base modificados.

ii.- Listas de objetos persistentes modificados.

CAPITULO VI

PRUEBAS Y EJEMPLOS

En este capítulo se muestran las pruebas y algunos de los ejemplos que se desarrollaron empleando el sistema.

Los resultados que se muestran fueron obtenidos de las pruebas efectuadas al sistema en una computadora Gould con sistema operativo "ultrix" versión 3.02.

VI.1 EJEMPLOS DE RESULTADOS DEL PREPROCESADOR.

En esta sección se muestra el funcionamiento del preprocesador, primer módulo del sistema implantado, mostrando el archivo que resulta de procesar un programa escrito en "C_persistente". Para cada ejemplo se muestra primero el archivo de entrada al preprocesador, y en segundo lugar el archivo resultante. Por conveniencia los archivos son sencillos, y en lo posible ilustrativos del trabajo que realiza el preprocesador.

El programa uno realiza la inserción de un número deseado de OPs en un Objeto Base determinado. En éste se muestran los pasos a seguir, los cuales son: abrir el Objeto Base en el cual se desean insertar los OPs y efectuar la llamada a la función

inserta, la cual se encarga de verificar si es posible la inserción, si no lo es el sistema generará un mensaje.

```
/*  pi.p
      programa de prueba para el sistema efectúa las
      siguientes tareas:
      1. abre un Objeto Base de un usuario específico
      2. inserta un número de objetos de tipo perro en el
      obase
      3. hace llamadas a la función leellaves para obtener
      las llaves que identifican a los OPs insertados.
*/

#include "thead.h"
#include  "hextern.h"
persist  struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      struct perros *hijo;
      };
persist struct perros *pperro;
persist struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{

  int i=1,j,k=0;
  char obase[9];
  char apar[500][5];

  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
  printf("objeto base a abrir ?");
  scanf("%s",obase);
  if((pobase=abrebase("felix",obase,"xilef",2))==NULL){
    printf("problemas en la apertura del obase\n");
    exit();
  }
  while(i){
    leellaves(apar);
    printf("cuantos caninos inserto ?");
    scanf("%d",&j);
    empconteo();
    for(;k<j;k++){
      pperro=(struct perros*)malloc(sizeof(struct perros));
      strcpy(pperro->nombre,apar[k]);
      if(inserta(pperro->nombre,pobase,pperro)==FALLA)
        exit();
```

```
    }
    paraconteo(stderr,"insercion");
    comunit();
    printf("\n \t\totra medicion ?");
    scanf("%d",&i);
  }
} /* de main */
```

Prog 1.a. Progrmama en C_persistente muestra la inserción OPs de clase perro en el Objeto Base deseado

El siguiente código es el que entrega el preprocesador cuando se le da como entrada el prog. 1.a. Las diferencias entre ambos archivos se indica en negritas, y como puede observarse son la inclusión de una llamada a una función con un único parámetro que es un archivo que contiene los identificadores de OPs con que trabaja la función, y la expansión de la llamada al procedimiento inserta.

```
#include "thead.h"
#include  "hextern.h"
struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      struct perro *hijo;
      };
struct perros *pperro;
struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{
  int i=1,j,k=0;
  char obase[9];
  char apar[500][5];

  inicializa("idpi.p");
  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
  printf("objeto base a abrir ?");
  scanf("%s",obase);
  if((pobase=abrebase("felix",obase,"xilef",2))==NULL){
    printf("problemas en la apertura del obase\n");
    exit();
  }
```

```
   while(i){
     leellaves(apar);
     printf("cuantos caninos inserto ?");
     scanf("%d",&j);
     empconteo();
     for(;k<j;k++){
       pperro=(struct perros*)malloc(sizeof(struct perros));
       strcpy(pperro->nombre,apar[k]);
       if(inserta(pperro->nombre,pobase,(char *)pperro,
                  &ptipos[0])==FALLA)
         exit();
     }
     paraconteo(stderr,"insercion");
     commit();
     printf("\n \t\totra medicion ?");
     scanf("%d",&i);
   }
} /* de main */
```

Prog 1.b. Programa que entrega el preprocesador del sistema cuando se le da como entrada el código del prog. 1.a

El código del programa dos muestra como pueden ser accesados los atributos de un **OP** a través de una indirección, esto es, uno puede localizar un **OP** y con éste obtener información de los **OPs** a que éste haga referencia, si es que la hace. En el programa 2 se obtiene información de un **OP** de la misma clase, sin embargo en otro caso pueden ser de diferente clase.

```
/*
     programa de prueba para el sistema C_persistente. El
    objetivo de éste es localizar indirectamente un OP si es que
     existe.
*/
#include "thead.h"
#include  "hextern.h"
persist  struct perros{
     char nombre[45];
     char raza[32];
     int edad;
     struct perros *hijo;
     };
persist struct perros *pperro;
persist struct perros *pperro1;
```

```
OBJETOB *pobase = NULL;
main()
{
  char per[45],objbase[9];
  char hija[30];

  if((raizu=emptrans())==NULL)
    exit();
  printf("objeto base a abrir ?");
  scanf("%s",objbase);
  if((pobase=abrebase("felix",objbase,"xilef",2))==NULL){
   printf("problemas para abrir el objeto base\n");
   exit();
  }
  printf("perro a buscar ?");
  scanf("%s",per);
  if((perro2 =busca(per,pobase))==NULL)
    printf("perro no registrado\n");
  else
   if(pperro2->hijo != NULL)
     printf("el hijo de (%s) es %s\n",per,per->hijo->nombre);
   else
     printf("no tiene hijo\n");
}
```

Prog 2.a Programa en C_persistente cuyo objetivo es loclaizar a través de una indirección un objeto. En este caso ambos son de la misma clase

El programa 2.b muestra el código que entrega el sistema cuando se le proporciona el programa 2.a. Los objetivos de este código son obtener información de un OP por medio de una indirección, esto es, un OP puede hacer referencia a otros OPs, por lo que se puede obtener la información contenida en estos a partir del OP localizado. Las diferencias entre los códigos de los programas 2.a y 2.b están en la instrucción que maneja la indirección, en ésta se efectúa una expanción, que indica al MOP que debe de accesar un OP, idependientemente de si este se encuentra en memoria primaria o secundaria, y la inclusión de la llamada a la función *inicializa* con un parámetro que es el nombre del archivo que contiene los identificadores del OP con que trabaja el procedimiento.

```
#include "thead.h"
```

```
#include  "hextern.h"
struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      long hijo;
      };
struct perros *pperro;
struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{
  char per[45],objbase[9];
  char hija[30];

  inicializa(idp2);
  if((raizu=emptrans())==NULL)
    exit();
  printf("objeto base a abrir ?");
  scanf("%s",objbase);
  if((pobase=abrebase("felix",objbase,"xilef",2))==NULL){
   printf("problemas para abrir el objeto base\n");
   exit();
  }
  printf("perro a buscar ?");
  scanf("%s",per);
  if((perro2 = (struct perro *)busca(&ptipos[0],per,
                                          pobase))==NULL)

    printf("perro no registrado\n");
  else
   if(pperro2->hijo != NULL)
     printf("el hijo de <%s> es %s\n",per,(struct perro *)
        locind(pperro2->hijo,&ptipos[0],pobase)->nombre);
   else
     printf("no tiene hijo\n");
}
```

prog. 2.b Código C resultante al proporcionar como entrada el programa 2.a.

El programa 3a muestra la forma en que dos OPs insertados previamente pueden ser relacionados.

```
/*
```

programa de prueba para el sistema C_persistente. El objetivo de éste es establecer una relacion entre dos OPs

```
    que existan
*/

#include "thead.h"
#include  "hextern.h"
persist  struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      struct perros *hijo;
      };
persist struct perros *pperro;
persist struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{
  char per[45],objbase[9];
  char hija[30];

  if((raizu=emptrans())==NULL)
    exit();
  printf("objeto base a abrir ?");
  scanf("%s",objbase);
  if((pobase=abrebase("felix",objbase,"xilef",2))==NULL){
   printf("problemas para abrir el objeto base\n");
   exit();
  }
  printf("perro a buscar ?");
  scanf("%s",per);
  if((perro2 =busca(per,pobase))==NULL)
    printf("perro no registrado\n");
  else{
   printf("Cual es su hija ?");
   scanf("%s",hija);
   if((perro1 =busca(hija,pobase))==NULL)
     printf("no esite la hija\n");
   else
     perro2->hijo = perro1;
  }
  commit();
}
```

Prog 3.a Programa en C_persistente. En este ejemplo se muestra como puede ser establecida una relación entre OPs. En este caso, los OPs son de la misma clase

El código del programa 3.b es el programa que entrega como salida el sistema cuando se proporciona como entrada el programa 3.a. Las diferencias entre los programas 3.a y 3.b están en las

instrucciones que contienen identificadores de OPs, las cuales son expandidas para indicar al MOP que tarea debe de efectuar. En el código presente estas diferencias son marcadas con negritas. la tarea de la función locpid es localizar el **pid** del objeto, ya que éste es necesario cuando se está haciendo referencia a los campos de tipo apuntador. como se explicó en el capítulo V.

```
#include "thead.h"
#include  "hextern.h"
struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      long hijo;
      };
struct perros *pperro;
struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{
  char per[45],objbase[9];
  char hija[30];

  inicializa(idpro2);
  if((raizu=emptrans())==NULL)
    exit();
  printf("objeto base a abrir ?");
  scanf("%s",objbase);
  if((pobase=abrebase("felix",objbase,"xilef",2))==NULL){
   printf("problemas para abrir el objeto base\n");
   exit();
  }
  printf("perro a buscar ?");
  scanf("%s",per);
  if((perro2 = (struct perro *) busca(&ptipos[0],per,pobase))==NULL)
    printf("perro no registrado\n");
  else{
   printf("Cual es su hija ?");
   scanf("%s",hija);
   if((perro1 = (struct perro *)busca(&ptipos[0],hija,pobase))==NULL)
     printf("no esite la hija\n");
   else
     perro2->hijo = locpid((char *)perro1,pobase);
  }
  commit();
}
```

Prog 3.b Programa C obtenido del sistema cuando se da como entrada el código de la figura 3.a. Este programa muestra la manera de establecer la relación entre dos OPs

El programa 4.a muestra la forma como puede ser utlizada la función *buscaob*, con la cual puede ser localizado un OP en un Objeto Base, así mismo se muestran los pasos que deben seguirse, los cuales son: abrir un Objeto Base y efectuar llamadas a la función *busca*. El programa efectúa una llamada a la función *leellaves* la cual inicializa un arreglo de cadenas de carácteres, las cuales son utilizadas como llaves de los OPs que se buscan.

```
/*
      programa de prueba para el sistema. el objetivo de este
      programa es buscar objetos persistentes. Las busquedas son
      efectuadas en grupos. Las llaves de los OPs que se buscan
      son tomadas del arreglo apar, el cual es inicializado por la
      funcion leellaves.
*/


#include "thead.h"
#include  "hextern.h"
persist  struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      struct perro *hijo;
      };
persist struct perros *pperro;
persist struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{

  int i=1,j,k=0;
  char obase[9];
  char apar[500][5];

  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
  printf("objeto base a abrir ?");
  scanf("%s",obase);
  if((pobase=abrebase("felix",obase,"xilef",2))==NULL){
    printf("problemas en la apertura del obase\n");
    exit();
  }
  leellaves(apar);
```

```
   while(i){
     printf("cuantos caninos busco   ?");
     scanf("%d",&j);
     empconteo();
     for(;k<j;k++){
       if((pperro1=busca(apar[k],pobase))==NULL){
         printf("pperro1->nombre =%s\n"pperro1->nombre);
         printf("pperro1->raza = %s\n",pperro1->raza);
         printf("pperro1->edad = %d\n",pperro1->edad);
       else
         printf("no existe el perro\n");
     paraconteo(stderr,"insercion");
     printf("\n \t\totra medicion ?");
     scanf("%d",&i);
   }
} /* de main */
```

Prog 4.a Programa en C_persistente. El objetivo de esta función es efectuar busquedas de OPs en un Objeto Base, todos los OPs son de la misma clase en este ejemplo.

El código del programa 4.b es el resultado que entrega el sistema cuando se le da como entrada el programa 4.a. las diferencias entre ambos códigos se marcan en 4.b con negritas. Como puede observarse las únicas dos diferencias son: la inclusión de la llamada a la función *inicializa* con una cadena como parámetro, la cual es el nombre del archivo que contiene los identificadores de OPs con que trabaja el procedimiento, y la expanción de la llamada a la función *buscaob*.

```
#include "thead.h"
#include  "hextern.h"
struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      long  hijo;
      };
struct perros *pperro;
struct perros *pperro1;
OBJETOB *pobase = NULL;

main()
{
```

```
  int i=1,j,k=0;
  char obase[9];
  char apar[500][5];

  inicializa(idbus);
  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
  printf("objeto base a abrir ?");
  scanf("%s",obase);
  if((pobase=abrebase("felix",obase,"xilef",2))==NULL){
    printf("problemas en la apertura del obase\n");
    exit();
  }
  leellaves(apar);
  while(i){
    printf("cuantos caninos busco   ?");
    scanf("%d",&j);
    empconteo();
    for(;k<j;k++){
      if((pperro1=(struct perro *)busca(&ptipos[0],apar[k],
                                        pobase))==NULL){
        printf("pperro1->nombre =%s\n"pperro1->nombre);
        printf("pperro1->raza = %s\n".pperro1->raza);
        printf("pperro1->edad = %d\n",pperro1->edad);
      else
        printf("no existe el perro\n");
    paraconteo(stderr,"busqueda");
    printf("\n \t\totra medicion ?");
    scanf("%d",&i);
  }
} /* de main */
```

prog 4.b código que entrega el sistema cuando se le da como entrada el programa 4.a.

El programa 5 muestra la forma como puede ser utlizada la función *borraob* para eliminar OPs de una Objeto Base perteneciente a un usuario determinado. El procedimiento elimina objetos con llaves contenidas en el arreglo llamado apar.

```
/*
      programa de prueba par el sistema. el objetivo de este codigo
      es eliminar OPs de un Objeto Base. Las llaves de los OPs que
      se eliminan son tomadas del arreglo apar, el cual es
      inicializado por la funcion leellaves.
*/
#include "thead.h"
#include  "hextern.h"
persist  struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      struct perro *hijo;
      };
persist struct perros *pperro;
persist struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{

  int i=1,j,k=0;
  char obase[9];
  char apar[500][5];

  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
  printf("objeto base a abrir ?");
  scanf("%s",obase);
  if((pobase=abrebase("felix",obase,"xilef",2))==NULL){
    printf("problemas en la apertura del obase\n");
    exit();
  }
  leellaves(apar);
  while(i){
    printf("cuantos caninos elimino ?");
    scanf("%d",&j);
    empconteo();
    for(;k<j;k++)
      pperro1=borraob(apar[k],pobase)
    paraconteo(stderr,"borrado ");
    printf("\n \t\totra medicion ?");
    scanf("%d",&i);
  }
  exit();
} /* de main */
```

Prog 5.a Programa en C_persistente cuyo objetivo es eliminar OPs de un Objeto Base, todos los OPs son de la misma clase.

El programa 5.b es el código resultante del sistema cuando se da como entrada el programa 5.a. Las diferencias son: la llamada a la función *borraob* y la inclusión de la llamada a la función *inicializa*, con un parámetro que es el nombre del archivo que contiene los identificadores de OPs con que trabaja el procedimiento.

```
#include "thead.h"
#include  "bextern.h"
struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      long  hijo;
      };
struct perros *pperro;
struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{
  int i=1,j,k=0;
  char obase[9];
  char apar[500][5];

  inicializa(idborra);
  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
  printf("objeto base a abrir ?");
  scanf("%s",obase);
  if((pobase=abrebase("felix",obase,"xilef",2))==NULL){
    printf("problemas en la apertura del obase\n");
    exit();
  }
  leellaves(apar);
  while(i){
    printf("cuantos caninos elimino ?");
    scanf("%d",&j);
    empconteo();
    for(;k<j;k++)
      pperro1=(struct perros *)borraob(&ptipos[0],apar[k],pobase)
    paraconteo(stderr,"borrado ");
    printf("\n \t\totra medicion ?");
    scanf("%d",&i);
```

```
  }
  exit();
} /* de main */
```

Prog 5.b Programa resultante del preprocesador cuando se le da como entrada el código del programa 5.a. El código de esta función resultante está en C estandar.

El programa 6 muestra la utilización de la función abrebase la cual creará los Objetos Base deseados si estos no existen, en caso de que existán, únicamente serán abiertos en la manera que se indica.

```
/*
     programa de prueba para el sistema C_persistente. el
     objetivo de este procedimiento es crear el numero posible de
     objetos base, esto es el numero permitido para una
     transaccion.
*/

#include "thead.h"
#include  "hextern.h"
persist  struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      struct *hijo;
      };
persist struct perros *pperro;
persist struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{
  int i=1,j,k=0;
  char obase[9];
  char apar[500][5];

  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
  leellaves(apar);
  while(i){
    printf("cuantos Objetos Base creo ?");
    scanf("%d",&j);
    empconteo();
    for(;k<j;k++)
```

```
      if((abrebase("felix",apar[k],"xilef",MODOL)==NULL){
        printf("problemas en la apertura creacion del %d esimo
               Objeto Base\n");
        exit();
      }
    commit();
    paraconteo(stderr,"borrado ");
    printf("\n \t\totra medicion ?");
    scanf("%d",&i);
  }
  exit();
} /* de main */
```

Prog. 6.a Programa en C_persistente cuyo objetivo es abrir si es que existen los Objetos Base, o crearlos en caso contrario. El número de Objetos Base abiertos, dependerá de la configuración que tenga el sistema que se use

El programa 6.b es el resultado que da el sistema cuando le es proporcionado el programa 6.a. La diferencia de este programa con el 6.a es únicamente la inclusión de la llamada a la función inicializa(), con un parámetro que indica el nombre del archivo que contiene los identificadores de las clases de OPs que maneja el procedimiento.

```
#include "thead.h"
#include  "hextern.h"
struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      long  hijo;
      };
struct perros *pperro;
struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
{
  int i=1,j,k=0;
  char obase[9];
  char apar[500][5];

  inicializa("idcrea.p");
  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
```

```
    leellaves(apar);
    while(i){
      printf("cuantos Objetos Base creo ?");
      scanf("%d",&j);
      empconteo();
      for(;k<j;k++)
        if((abrebase("felix",apar[k],"xilef",MODOL)==NULL){
          printf("problemas en la apertura creacion del %d esimo
                  Objeto Base\n");
          exit();
        }
      commit();
      paraconteo(stderr,"borrado ");
      printf("\n \t\totra medicion ?");
      scanf("%d",&i);
    }
    exit();
} /* de main */
```

Prog 6.b Programa que resulta cuando se da el código del prog. 6.a. Este código en C efectúa la apertura o creación de Objetos Base pertenecientes a un usuario particular.

El programa 7 muestra la forma como puede ser utilizada la función *bobase* la cual tiene el objetivo de eliminar Objetos Base de un usuario

```
/*
      programa de prueba para el sistema. El objetivo de este
      procedimiento es eliminar objetos base pertenecientes a un
      usuario.
*/
#include "thead.h"
#include  "hextern.h"
persist  struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      struct perro *hijo;
      };
persist struct perros *pperro;
persist struct perros *pperro1;
OBJETOB *pobase = NULL;

main()
{
  int i=1,j,k=0;
  char obase[9];
```

```
   char apar[500][5];

   if((raizu=emptrans())==NULL){
     printf("no existe registro de usuarios\n");
     exit();
   }
   leellaves(apar);
   while(i){
     printf("cuantos Objetos Base elimino ?");
     scanf("%d",&j);
     empconteo();
     for(k=j;k++)
       if(bobase("felix",apar[k],"xilef")==FALLA){
         printf("problemas en la eliminacion del %desimo
                 Objeto Base\n");
         exit();
       }
     commit();
     paraconteo(stderr,"borrado ");
     printf("\n \t\totra medicion ?");
     scanf("%d",&i);
   }
   exit();
} /* de main */
```

prog 7.a Programa en C_persistente,cuyo objetivo es eliminar Objetos Base de un usuario

El programa 7.b es el resultado de proporcionar como entrada el programa 7.a al sistema implantado. Los objetivos de este procedimiento son eliminar Objetos Base pertenecientes a un ususario particular. La diferencia está en la inclusión de una llamada a la función *inicializa* con un parámetro que contiene los identificadores de las clases con que trabaja la función.

```
#include "thead.h"
#include  "hextern.h"
persist  struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      long  hijo;
      };
struct perros *pperro;
struct perros *pperro1;
OBJETOB *pobase = NULL;
main()
```

```
{
  int i=1,j,k=0;
  char obase[9];
  char apar[300][5];

  inicializa("idpbobo.p");
  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }

 leellaves(apar);
  while(i){
    printf("cuantos Objetos Base elimino ?");
    scanf("%d",&j);
    empconteo();
    for(k<j;k++)
      if(bobase("felix",apar[k],"xilef")==FALLA){
        printf("problemas en la eliminacion del %desimo
                Objeto Base\n");
        exit();
      }
    commit();
    paraconteo(stderr,"borrado ");
    printf("\n \t\totra medicion ?");
    scanf("%d",&i);
  }
  exit();
} /* de main */
```

Prog 7.b Programa en C que resulta cuando se da al sistema el código mostrado en el prog. 7.a. El objetivo de este procedimiento es eliminar Objetos Base de un usuario.

El progrma 8 es un ejemplo del uso del sistema con el cual se puede examinar o modificar un Objeto Base de un usuario, el programa principal efectúa llamadas a las funciones básicas para OPs.. Este ejemplo es típico de un sistema de consulta.

```
/*
     programa de aplicacion del sistema C_persitente. el
     objetivo de este procedimiento es el clasico de altas bajas
     y cambios que se efectua con una base de datos.
*/

/
#include "thead.h"
#include  "hextern.h"
```

```
persist  struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      struct perros *hijo;
      };
persist struct perros *pperro;
persist struct perros *pperro1;
OBJETOB *pobase = NULL;

main()
{
  int trans = 1;
  char obase[9];

  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
  printf("objeto base a abrir ?");
  scanf("%s",obase);
  if((pobase = abrebase("felix",obase,"xilef",2))==NULL){
    printf(problemas en la apertura \n");
    exit();
  }
  while(trans){
    printf("\n\n");
    printf("\t\t\t   salir   ----------0\n");
    printf("\t\t\t insertar -----------1\n");
    printf("\t\t\t buscar -------------2\n");
    printf("\t\t\t borrar -------------3\n");
    printf("\n\n");
    scanf("%d",&trans);
    if(trans==0)
      ;
    else if(trans ==1)
          alta();
    else if(trans ==2)
          localiza();
    else if(trans ==3)
          borra();
    else printf("opcion invalida");
  }/* de while */
  commit();
} /*de main */

alta()
{
  int i=1;

  while(i){
    pperro = (struct perros *)malloc(sizeof(struct perros));
    printf("\n\t\t perro->nombre? ");
```

```
      scanf("%s",perro->nombre);
      printf("\n\t\t perro->raza? ");
      scanf("%s",perro->raza);
      printf("\n\t\t perro->edad? " );
      scanf("%d",perro->edad);
      pperro->hijo = NULL;
      inserta(pperro->nombre,pobase,pperro);
      printf("\n\n");
      printf("otra insercion (0 1)? ");
      scanf("%d",i);
    }
} /* de alta *\
localiza()
{
   int i=1;
   char llave[30];

   while(i){
     printf("llave del objeto a buscar ? ");
     scanf("%s",llave);
     if((pperro1=busca(llave,pobase))!=NULL){
       printf("pperro1->nombre = %s\n",pperro1->nombre);
       printf("pperro1->raza = %s\n",pperro1->raza);
       printf("pperro1->edad = %d\n",pperro1->edad);
     }
     else
       printf("no existe el objeto\n");
     printf("otra busqueda (0 1) ?");
     scanf("%d",&i);
   }
}
borra()
{
   int i=1;
   char llave[30];

   while(i){
     printf("llave del objeto a borrar ? ");
     scanf("%s",llave);
     if((pperro1 = borraob(llave,pobase))!=NULL){
       printf("pperro1->nombre = %s\n",pperro1->nombre);
       printf("pperro1->raza   = %s\n",pperro1->raza);
       printf("pperro1->edad   = %d\n",pperro1->edad);
     }
     else
       printf("no existe el objeto\n");
     printf("otra exclucion ? ");
     scanf("%d",i);
   }
}
```

prog 8.a Programa en C_persistente. Este procedimiento muestra las operaciones básicas que se pueden realizar con los OPs , en una aplicación de altas bajas y busquedas.

El programa 8.b muestra el código C que resulta cuando se da como entrada al sistema el programa 8.a. Las diferencias entre los códigos son marcadas con negritas y son, básicamente, la expanción de las funciones con que se manejan los OPs, las cuales han sido aumentadas con los parámetros que el preprocesador obtubo, así como la inclusión de la llamada a la función *inicializa*.

```
#include "thead.h"
#include  "hextern.h"
struct perros{
      char nombre[45];
      char raza[32];
      int edad;
      long  hijo;
      };
struct perros *pperro;
struct perros *pperro1;
OBJETOB *pobase = NULL;

main()
{
  int trans = 1;
  char obase[9];

  if((raizu=emptrans())==NULL){
    printf("no existe registro de usuarios\n");
    exit();
  }
  printf("objeto base a abrir ?");
  scanf("%s",obase);
  if((pobase = abrebase("felix",obase,"xilef",2))==NULL){
    printf(problemas en la apertura \n");
    exit();
  }
  while(trans){
    printf("\n\n);
    printf("\t\t\t   salir   ----------0\n");
    printf("\t\t\t insertar -----------1\n");
    printf("\t\t\t buscar -------------2\n");
    printf("\t\t\t borrar -------------3\n");
    printf("\n\n);
    scanf("%d",&trans);
    if(trans==0)
      ;
    else if(trans ==1)
```

```
            alta();
        else if(trans ==2)
            localiza();
        else if(trans ==3)
            borra();
        else printf("opcion invalida");
    }/* de while */
    commit();
} /*de main */

alta()
{
    int i=1;

    while(i){
        pperro = (struct perros *)malloc(sizeof(struct perros));
        printf("\n\t\t perro->nombre? ");
        scanf("%s",perro->nombre);
        printf("\n\t\t perro->raza? ");
        scanf("%s",perro->raza);
        printf("\n\t\t perro->edad? " );
        scanf("%d",perro->edad);
        pperro->hijo = NULL;
        inserta(pperro->nombre,pobase,(char *)pperro,&ptipos[0]);
        printf("\n\n");
        printf("otra insercion (0 1)? ");
        scanf("%d",i);
    }
} /* de alta *\

localiza()
{
    int i=1;
    char llave[30];

    while(i){
        printf("llave del objeto a buscar ? ");
        scanf("%s",llave);
        if((pperro1=(struct perros *)busca(&ptipos[0],llave,
                                                pobase))!=NULL){
            printf("pperro1->nombre = %s\n",pperro1->nombre);
            printf("pperro1->raza = %s\n",pperro1->raza);
            printf("pperro1->edad = %d\n",pperro1->edad);
        }
        else
            printf("no existe el objeto\n");
        printf("otra busqueda (0 1) ?");
        scanf("%d",&i);
    }
}

borra()
```

```
{
  int i=1;
  char llave[30];

  while(i){
    printf("llave del objeto a borrar ? ");
    scanf("%s",llave);
    if((pperro1 = (struct perros *)borraob(&ptipos[0],
                                          llave,pobase))!=NULL){
      printf("pperro1->nombre = %s\n",pperro1->nombre);
      printf("pperro1->raza   = %s\n",pperro1->raza);
      printf("pperro1->edad   = %d\n",pperro1->edad);
    }
    else
      printf("no existe el objeto\n");
    printf("otra exclucion ? ");
    scanf("%d",i);
  }
}
```

VI.2 TIEMPO DE EJECUCION DE OPERACIONES BASICAS DEL MOP

En esta sección se muestran los tiempos de ejecución de las operaciones básicas de que se disponen para manipular **OPs** y Objetos Base. La función que se empleó para efectuar las mediciones es *times*, la cual forma parte de la biblioteca de procedimientos del sistema operativo UNIX. Esta función tiene un parámetro, de tipo apuntador a estructura. en este parámetro se reflejan las mediciones efectuadas. En los programas de ejemplos mostrados, las funciones empconteo y paraconteo hacen uso de la función *times* y reportan el tiempo que esta contabiliza. Las gráficas que se reportan son el tiempo que emplea el CPU en efectuar las operaciones, el tiempo empleado en efectuar el trabajo en el area del usuario es similar en este caso ya que el sistema se probó cuando no existian otros usuarios.

Las gráficas de tiempos de respuesta del sistema implantado son obtenidas de la ejecución de los programas mostrados como ejemplos. El objetivo de los ejemplos es obtener un tiempo de respuesta del sistema, por lo que se trata de eliminar en lo posible la interacción del sistema con el usuario.

Para algunas funciones no es posible mostrar un tiempo de respuesta único debido a la naturaleza misma de la función.

En cada caso se indica la función que se utilizo, y se explica la gráfica resultante.

Insercion Conservando OPs en Memoria Primaria

Sin Compromiso

En la gráfica (no x,y) mostrada en la figura 6.1 se muestra los resultados obtenidos al ejecutar el programa mostrado en el programa 1.b, exeptuando en éste la inclusión de la llamada a la

función *commit*. En ésta se gráfica se tienen los tiempos de respuesta del sistema cuando se insertan de 1 a 500 OPs. En este caso la inserción es incremental, esto es, se inserta un objeto, posteriormente se pide insertar otro objeto conservando los objetos previamente insertados en memoria primaria, hasta insertar un total de 500 OPs. De la gráfica podemos observar que el tiempo de respuesta tiende a ser lineal.

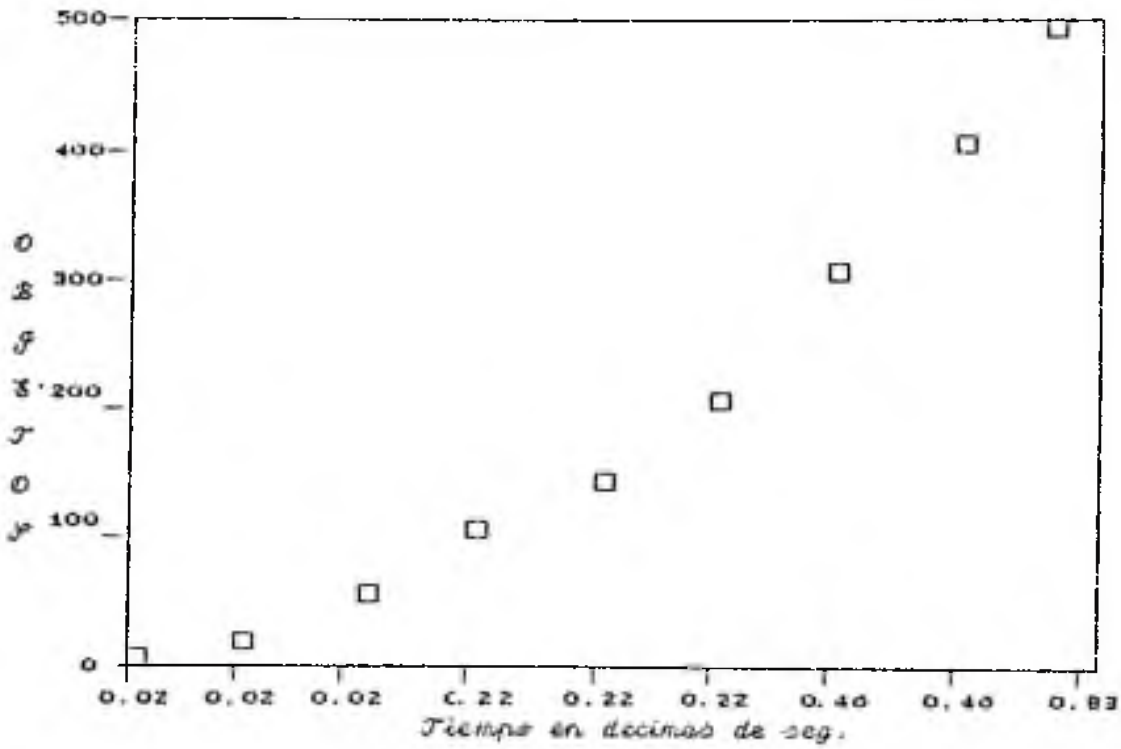


fig. 6.1 Tiempos de Respuesta del Sistema a Inserción.
de OPs conservandolos en Memoria Primaria sin Compromiso

Insercion sin Conservar OPs en Memoria Primaria

Sin Compromiso

La gráfica 6.2 fue obtenida de la ejecución del programa 1.b, pero sin la ejecución de la llamada a la función de compromiso. En

dicha figura se muestra la respuesta del sistema para la función inserción. En este caso los objetos fueron insertados por grupos sin conservarlos en memoria, esto es, por ejemplo se insertaron 10 OPs, se reinicializo el sistema, y nuevamente se insertó otro grupo de más elementos.

De las gráficas podemos observar el comportamiento lineal del sistema.

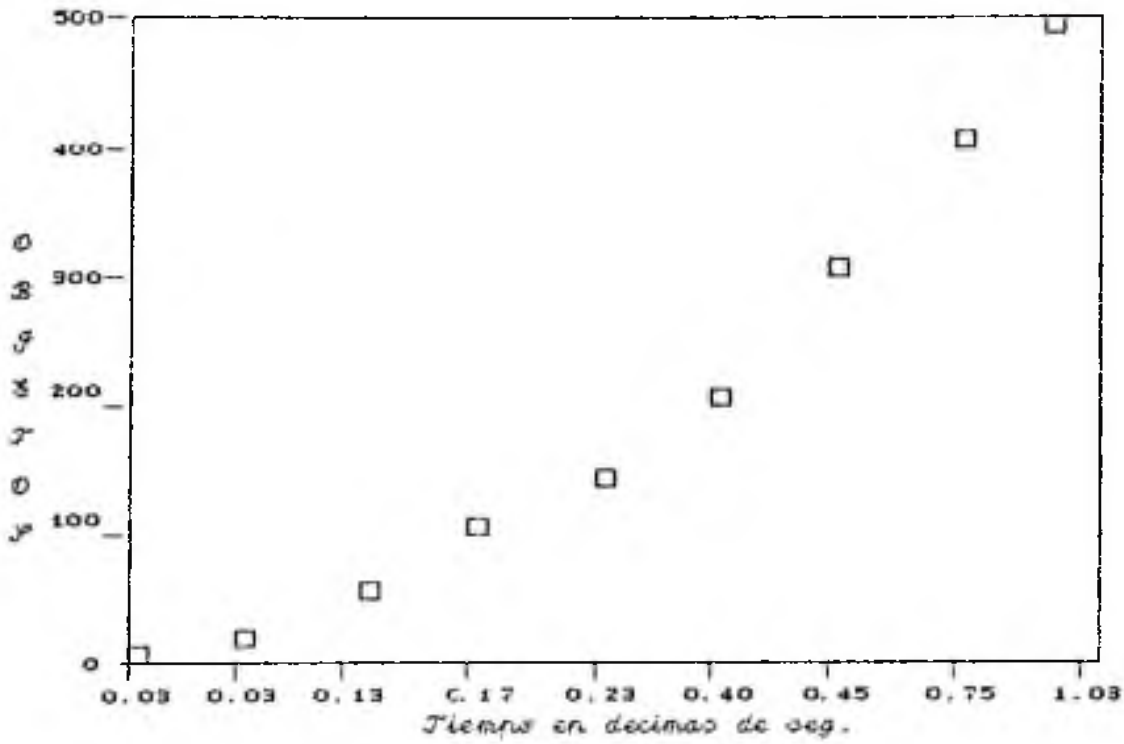


fig. 6.2 Tiempos de Respuesta del Sistema a Inserción.
de OPs sin conservarlos en Memoria Primaria y sin Compromiso

Insercion Conservando OPs en Memoria Primaria

Efectuando Compromiso

La gráfica 6.3 muestra los tiempos obtenidos al ejecutar el programa 1.b. Esta gráfica muestra el comportamiento del sistema cuando se insertan OPs en un Objeto Base. Para cada inserción se

conservan los OPs ya insertados en memoria primaria y se siguen efectuando inserciones, al final de las inserciones se efectúa el compromiso de la transacción.

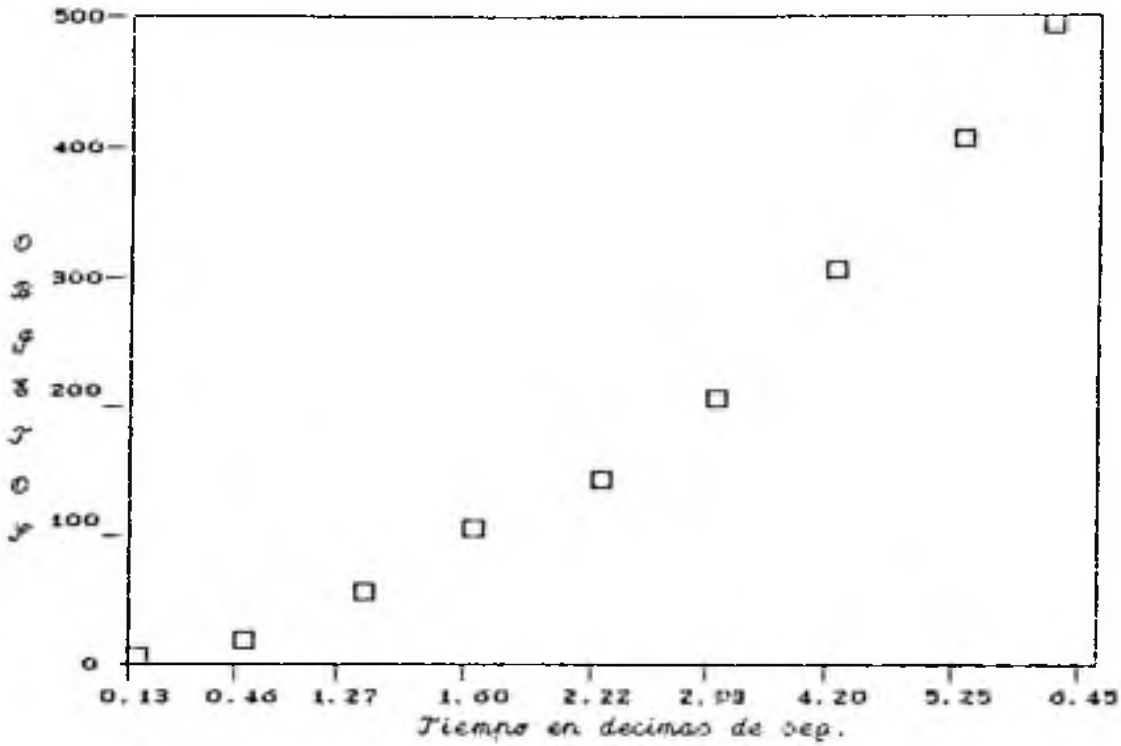


fig. 6.3 Tiempos de Respuesta del Sistema a Inserción. de OPs conservandolos en Memoria Primaria con Compromiso

Insercion sin Conservar OPs en Memoria Primaria

Efectuando Compromiso

La figura 6.4 muestra los tiempos de respuesta del sistema para la operación de inserción sin conservar objetos en memoria primaria, esto es, se ejecutaron varias veces el programa 1.b, insertando cada vez un número diferente de objetos, además se realizó al final de cada transacción la función de compromiso.

Nuevamente de la información de la gráfica (no x,y) podemos observar que el sistema tiende a tener una respuesta lineal.

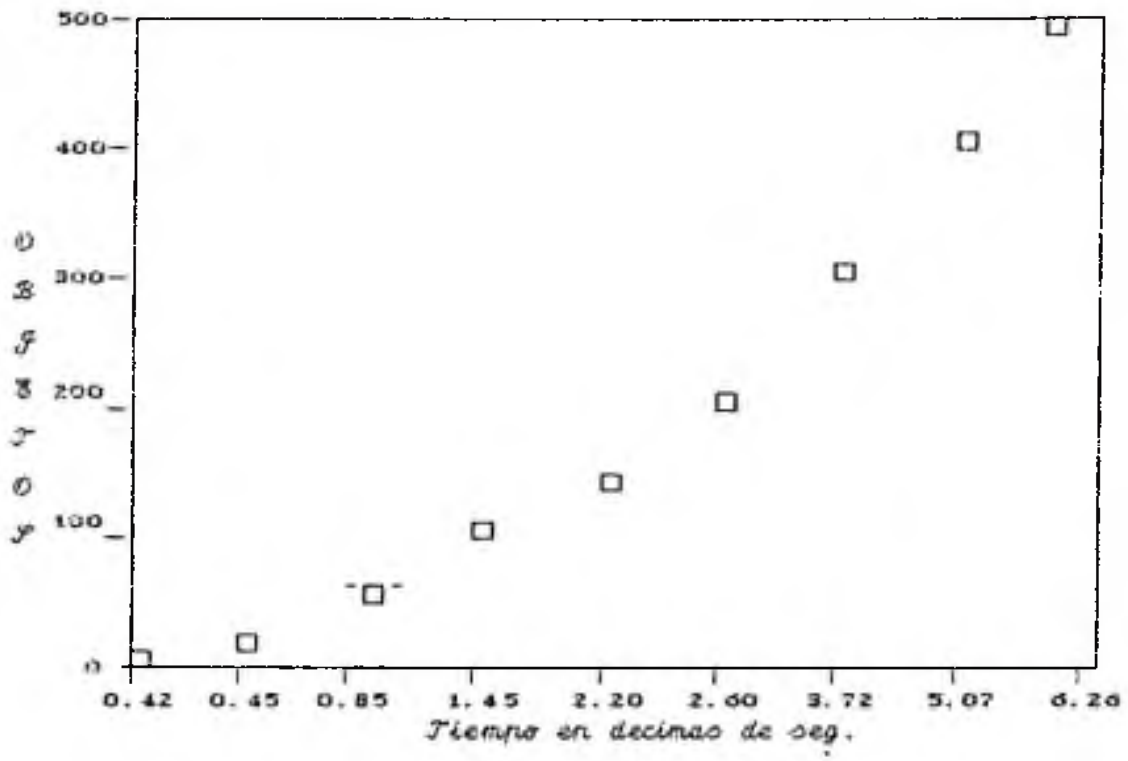


fig. 6.4 Tiempos de Respuesta del Sistema a Inserción.
de **OPs** sin conservarlos en Memoria Primaria con Compromiso

Busqueda de **OPs** Teniendo **OPs** en Memoria Primaria

La figura 6.5 muestra los tiempos de respuesta del sistema para la operación de busqueda teniendo **OPs** en memoria primaria, esto es, el procedimiento realiza busquedas de **OPs**, de los cuales es posible que algúnos se encuentren presentes en memoria primaria. Los resultados que se muestran en la gráfica fueron obtenidos al ejecutar el programa 4. La manera en que se efectuaron las mediciones fue tomando como llaves de los **OPs** a localizar las contenidas en el arreglo *apar*. Inicialmente ningún **OP** se encuentra en memoria primaria, sin embargo conforme se localizan, permanecen en memoria primaria, por lo que en la siguiente

busqeueda se encuentran ya algunos de éstos en memoria primaria.

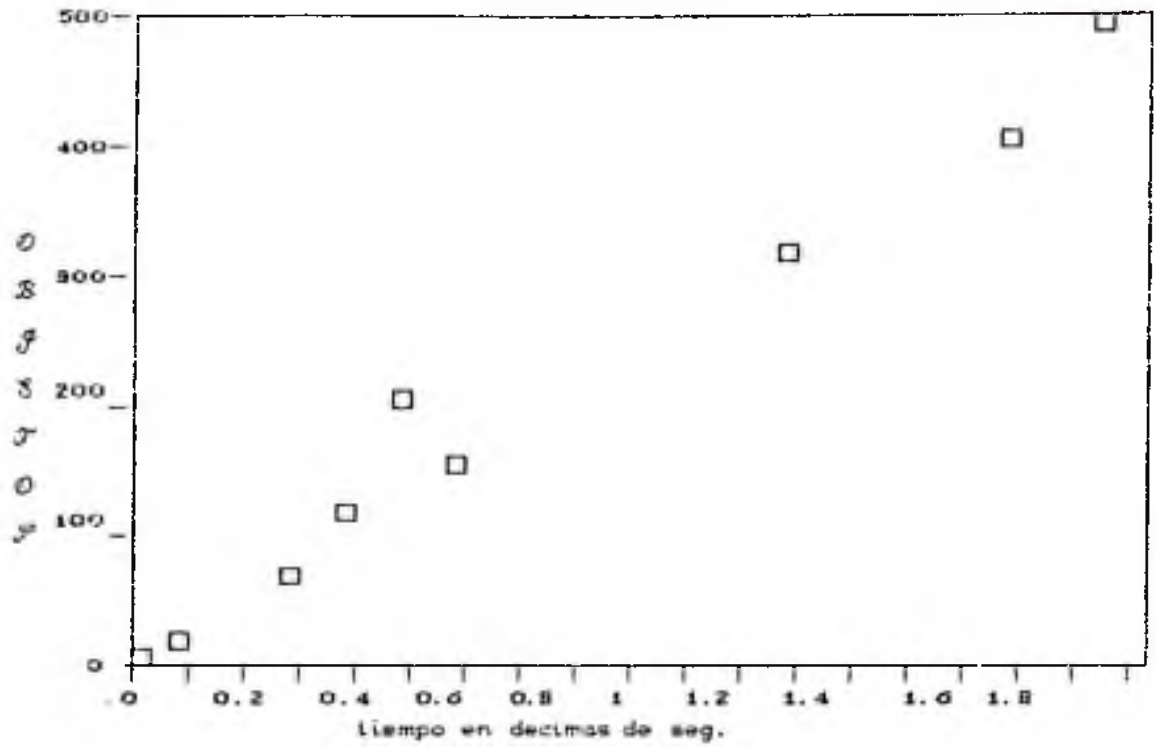


fig.6.5 Tiempos de CPU empleados por el sistema para efectuar busquedas de OPs conservando los que se encuentran en memoria primaria.

Busqueda de OPs en Sin Mantenerlos en Memoria Primaria

La figura 6.6 muestra los resultados que se obtuvieron al ejecutar el programa 4. La forma en que se efectuaron las mediciones fue ejecutando varias veces el programa, de tal manera que no pudieran existir OPs en memoria primaria. Las llaves por las cuales se localizan los OPs son tomadas del arreglo *apar*.

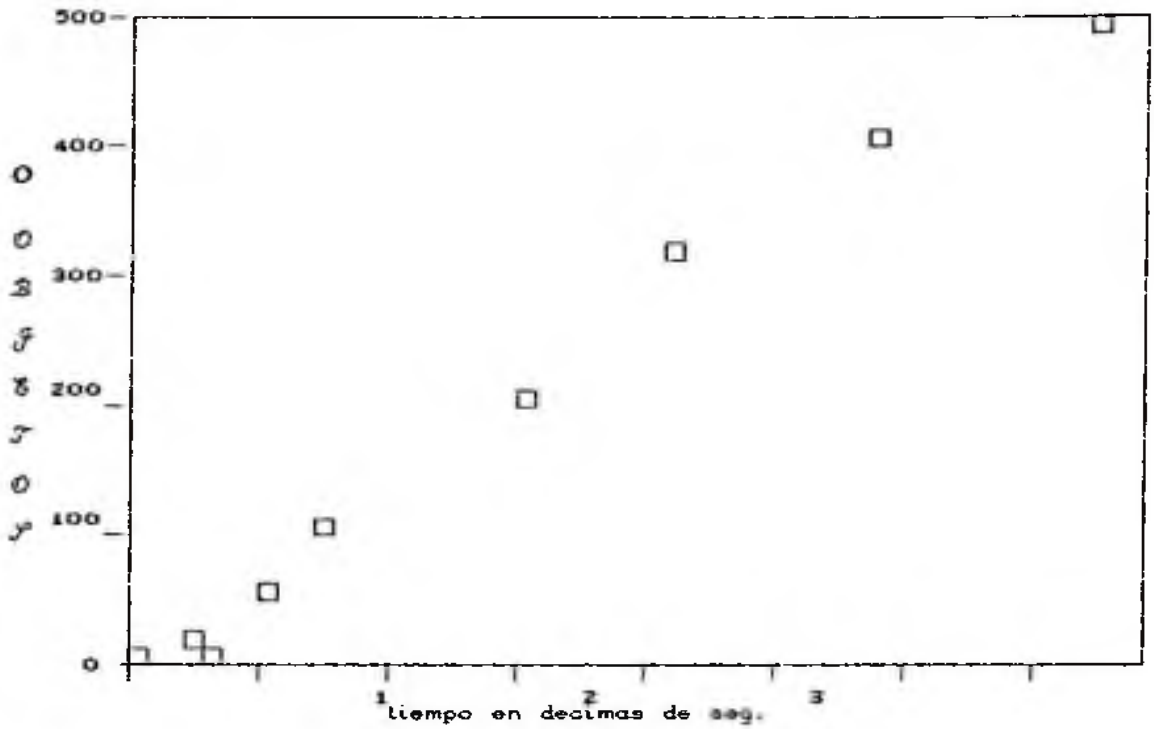


fig.6.6 Tiempos de CPU empleados por el sistema para efectuar busquedas de **OPs** sin conservar los que se encuentran en memoria primaria.

Eliminación de **OPs** de Un Objeto Base

La gráfica 6.7 muestra los resultados que se obtuvieron al ejecutar el programa 5. La primera gráfica es obtenida cuando se eliminan grupos de **OPs** pero sin efectuar la función de compromiso, mientras que los resultados gráficados en la ultima, los datos fueron obtenidos de la misma forma, sólo que en esta ocasión si se efectúa la función de compromiso. En ambos casos cuando se eliminan los **OPs**, estos no se encuentran en memoria primaria.

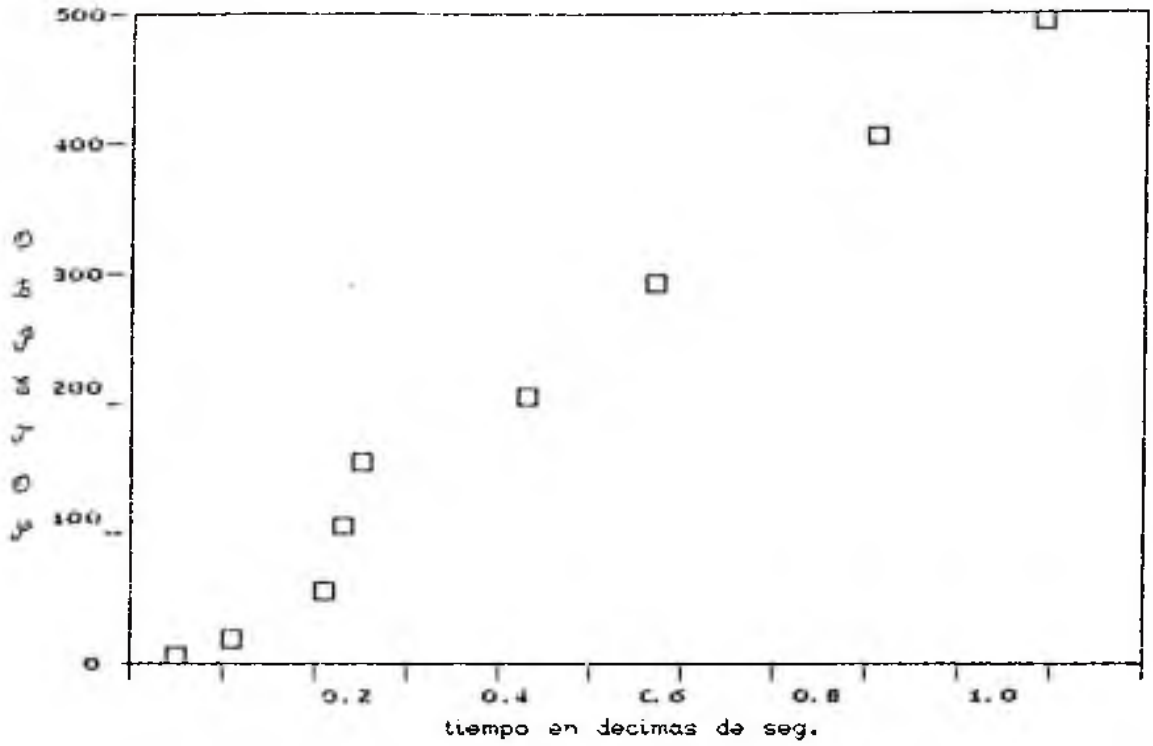


fig.6.7a Tiempos de CPU empleados por el sistema para borrar OPs sin efectuar la función de compromiso

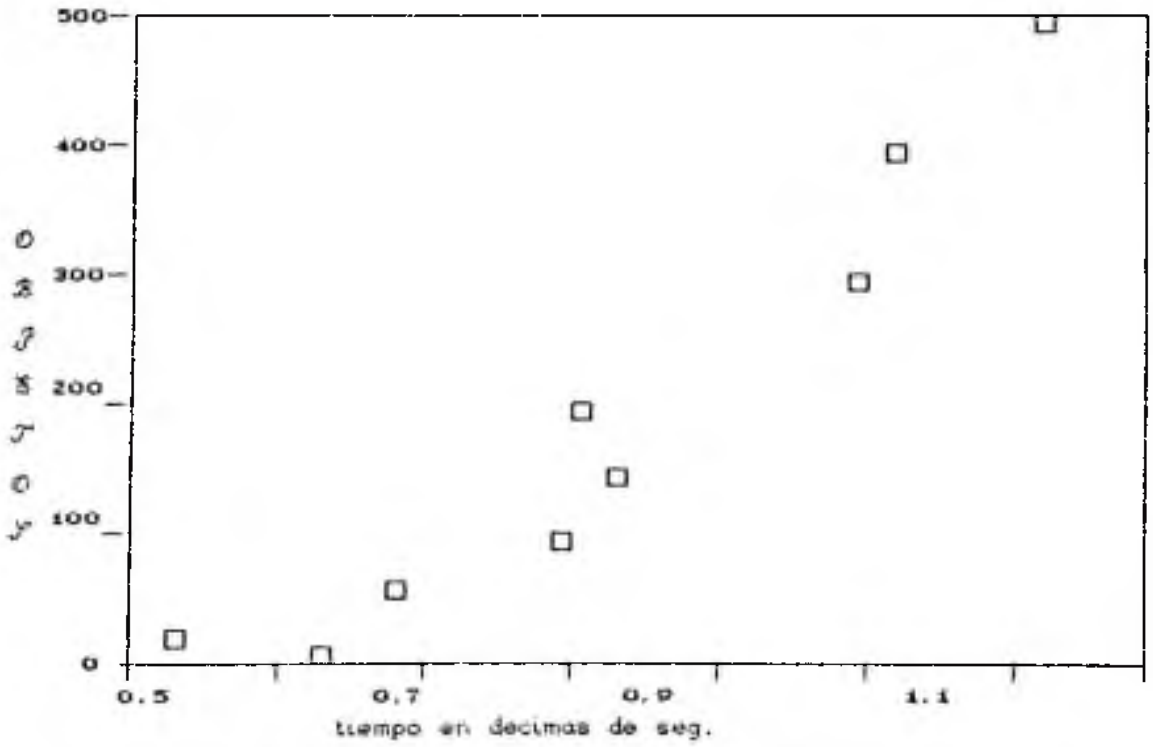


fig.6.7c Tiempos de CPU empleados por el sistema para borrar OPs efectuando la función de compromiso

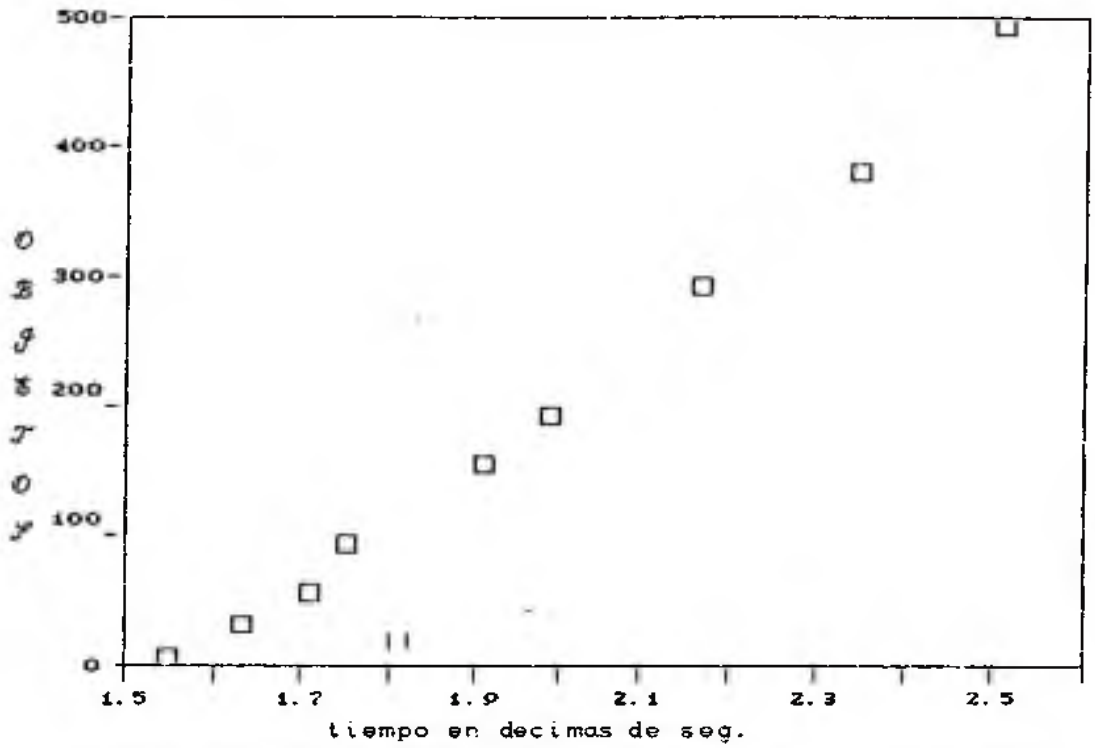


fig 6.7d Tiempos de Usuario reportados cuando se eliminan **OPs** de un Objeto Base efectuando la función de compromiso

Creación y/o Apertura de Objetos Base

La gráfica 6.8 muestra los resultados obtenidos cuando se ejecuta el progrma 6.b cuyo, objetivo es abrir o crear Objetos Base. La razón por la que indicamos creación o apertura es porque en el caso que el Objeto Base que se desea abrir no exista se "creará", en caso que ya existiera sólo se abrirá. Las gráficas muestran los tiempos medidos cuando se llama la función primitiva *abrebase* y ninguno de los Objetos Base que se abren existía en memoria secundaria y/o primaria, esto es, los Objetos Base son creados. El número de Objetos Base credos en el ejemplo son 28, debido a que es el máximo que permite la configuración del sistema en el que se ejecutó el programa.

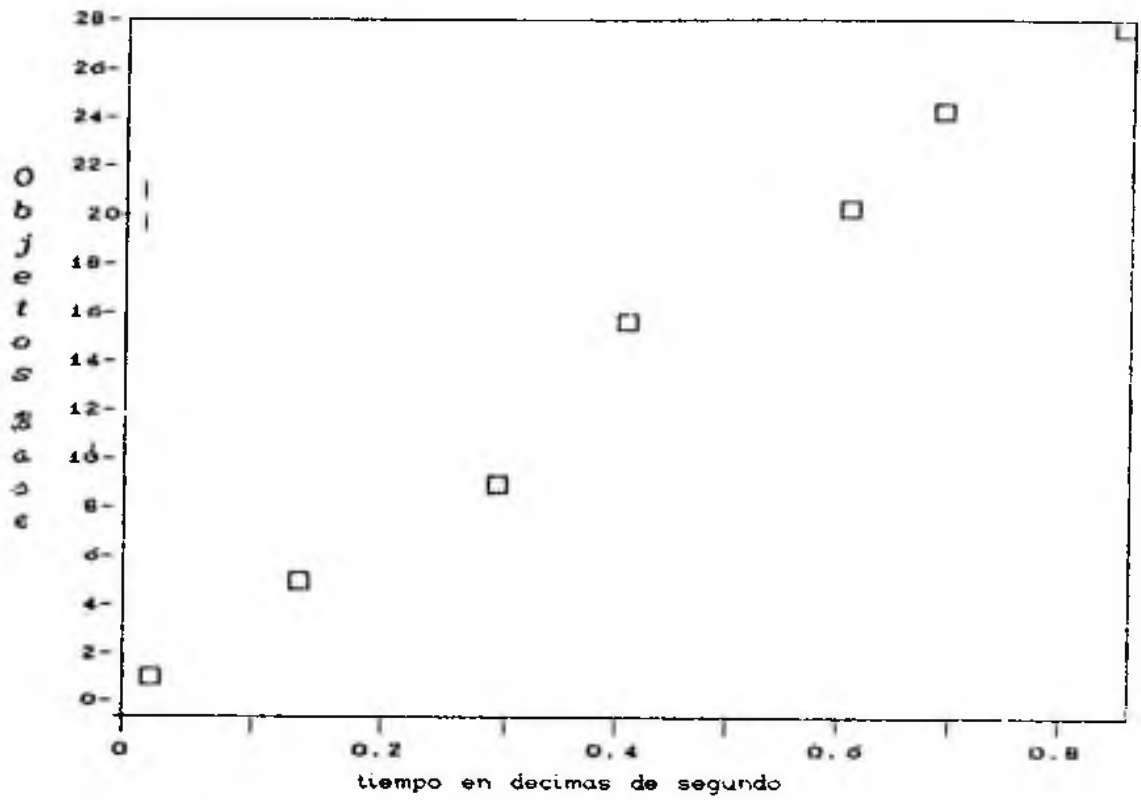


fig.6.8 Tiempos de CPU empleados por el sistema para crear Objetos Base

Eliminar Objetos Base

Eliminar un Objeto Base es una de las tareas que efectúa una de las funciones básicas del MOP, sin embargo la cuantización del tiempo que requiere para efectuar su trabajo no es una característica del sistema, ya que como se explica en los capítulos 3 y 4, un Objeto Base puede contener varias clases, las cuales utilizan espacio y pueden ser variadas en número, por lo que para eliminar un Objeto Base, es necesario recuperar el espacio empleado por cada un de las clases que contiene, esto trae con sigo que eliminar un Objeto base que contenga pocas clases sea más rapido que eliminar otro Objeto Base que contenga más clases.

VI.3 PROYECCION DE LA IMPLANTACION

En esta sección se discuten algunas sugerencias para la proyección del sistema.

El sistema orientado hacia el manejo de datos.

La implantación realizada como tal tiene una orientación similar a la de un Manejador de Bases de Datos, sin embargo , no existe en ella un lenguaje que permita hacer un tipo de interrogación para obtener información más concreta. Por lo que se presenta la opción de implantar un lenguaje de interrogación, por ejemplo alguno basado en el algebra relacional el cual haciendo uso de las funciones básicas del MOP, o directamente de las estructuras generadas por el sistema, permita obtener información más concreta.

El sistema orientado hacia una programación orientada a objetos.

Como resultado de la forma en que se dividio el sistema, compuesta de dos módulos un preprocesador y un manejador de objetos; haciendo unos cambios al preprocesador se puede obtener información de las funciones que le serián permitidas efectuar a cada clase de objetos, y obtener referencias a objetos para con estas determinar funciones que le sean permitidas efectuar por herencia.

El sistema con capacidades gráficas.

Del punto anterior, pudieramos tener la concepción de objetos de tipo gráfica, estos objetos deberan de poder efectuar

unicamente ciertas operaciones, como son desplegar, imprimir, agrandarse en tamaño, disminuirse en tamaño, etc..

Las sugerencias anteriores no son faciles de implantar. sin embargo el sistema que se presenta es una base que puede tomarse para abordar las tareas mencionadas con "más" facilidad.

CAPITULO VII

CONCLUSIONES

En este capítulo se describe el trabajo presentado, tomando como base, los objetivos fijados como métas, las experiencias en su desarrollo, uso del sistema, utilización de otras herramientas y con otras herramientas, y un análisis de las proyecciones del sistema.

1.- Durante el estudio preliminar a la implantación del sistema se consideraron diferentes aspectos, referentes a las maneras en que podría ser implantada una solución a los problemas que crean los lenguajes proceduales al tratar de diferentes maneras la memoria primaria y la memoria secundaria de una computadora, obteniendose como conclusión que el sistema debería de tener las dos características siguientes:

Adoptar un lenguaje base y sobre éste montar un conjunto de funciones que proporcionaran un sólo nivel de abstracción para memoria primaria y para memoria secundaria.

El lenguaje base seleccionado debería ser poco modificado.

El usuario debe notar una mínima diferencia al programar en el lenguaje modificado y el lenguaje base. Esto implica que debe ser transparente en lo posible el sistema implantado.

El sistema cumple con los puntos anteriores, como puede ser observado en los ejemplos del capítulo 6.

2.- Durante la implantación del los módulos que componen el sistema se tuvieron problemas de equipo, lo cual obligo a migrarlo por diferentes sistemas (todos UNIX), lo cual trajo consigo, una verificación de la portabilidad del sistema.

De estas expriencias tenemos:

i.- El sistema es portable en los sistemas con el mismo sistema operativo (UNIX), salvo diferencias propias de las versiones (Berkley y ATT).

ii.- Como resultado de la portabilidad del lenguaje base seleccionado ("C"), el sistema deberá funcionar en sistemas diferentes al UNIX, efectuando ciertas modificaciones a la función de compromiso. Estas modificaciones serán básicamente sustituciones de nombres de funciones. Ya que esta función hace uso de comandos disponibles en la mayoria de los sistemas operativos.

3.- Como resultado del uso del sistema tememos los siguientes puntos que cabe resaltar:

i.- Debido al poco uso del sistema implantado, no es posible garantizar en su totalidad el sistema, y será únicamente como consecuencia de su utilización que se obtenga la prueba final de su confiabilidad. Sin embargo las fallas que se llegaran a detectar serían consecuencia de una implantación deficiente, no así del concepto desarrollado.

ii.- Cuando se presenta la necesidad de conservar información, lo cual es una característica de los sistemas de información, se tienen diferentes alternativas, de las cuales podemos resaltar: el uso de las funciones de entrada y salida propias del lenguaje que estemos utilizando, el uso de un manejador de bases datos, o el uso del sistema implantado. Para las dos primeras opciones podemos mencionar:

Si utilizamos las funciones de entrada y salida propias de un lenguaje tradicional, tendremos que entender la manera como se almacenan los datos, implantar una forma eficiente de almacenar y leer los datos, implantar procedimientos de lectura y escritura a memoria secundaria.

Si empleamos un manejador de bases de datos tenemos que aprender otro lenguaje, y verificar si la forma en que se organizan los datos nos es útil para la aplicación en la cual necesitamos conservar la información.

Cuando en memoria primaria tenemos una relación entre entidades con un significado específico, tenemos por ejemplo un gráfo que represente el analisador léxico de un lenguaje, es dificil con cualquiera de las dos alternativas anteriores conservar la relacion entre las entidades que la forman.

4.- Del uso del sistema tenemos:

Los problemas que se presentan usando el sistema resultan cuando algún procedimiento hace uso del tipo "union" de "C",esto es causado porque la implantación no maneja este tipo.

5.- Comparando el sistema implantado con otras

herramientas y haciendo un análisis de su posible utilización con otras herramientas tenemos:

i.- En comparación con el uso de manejadores de bases de datos el sistema tiene una respuesta aceptable, como se puede observar en las gráficas de tiempos de respuesta, las cuales tienden a ser lineales en las operaciones de busqueda e inserción, lo cual comparativamente con algunos de los sistemas manejadores de bases de datos es mejor, ya que estos tienen tiempos de respuesta que tienden a ser exponenciales.

ii.- Para usar el sistema con otras herramientas como son otros compiladores, el sistema necesitará modificaciones únicamente al módulo del preprocesador, el cual deberá análizar el código proporcionado para obtener la información que necesita el MOP, y efectuar las expansiones necesarias en la forma que se requiera.

6.- Del análisis de las proyecciones del sistema tenemos:

Las proyecciones del sistema son muchas como resultado de no modificar el lenguaje y sólo incrementar sus capacidades por medio de funciones adicionales, por lo que no sólo se tiene un sistema sino un lenguaje con unas capacidades incrementadas.

REFERENCIAS

1.- M.P Atkinson, P. J. Bailey, K. J. Chisolm and R. Morrison, "PS-algol reference manual", Tech. report PPR-4-83, University of Edimburg, January 1983.

2.- R. Morrison, "S-algol languaje reference manual", Tech. report CS/79/1, University of St. Andrews,1979.

3.- M. P. Atkinson, K. J. Chisholm, P. J. Bailey, W. P. Cockshott and R. Morrison, "An approach to persistent programming", The computer Journal, 26, (1983).

4.- R. Morrison, "S-algol languaje reference manual", Tech. report CS/79/1, University of St. Andrews,1979.

5.- R. Morrison, "The string as a simple data type", ACM SIGPLAN Notices, 17, (3), (1982).

6.- R. Morrison, "low cost computer graphics for microcomputer", Software Practice and Expirience, 12, 767-776 (1982).

7.- B. Bayer and A. McCreight, "Organization and maintenance of large ordered indexes", Acta informatica, 1,173-189 (1972).

8.- M. P. Atkinson, "Progamming languajes and databases", Tech. report CSR-26-78, Computer Science Department, Univerity of Edimburg, 1978. Also in Proceedings of the fourth International Conference on Very Large Databases, 1978.

9.- M. P. Atkinson, K. J. Chisholm and W. P. Cockshott, "Nepal the new Edimburg persistent algorithmic language", in Database, Pergamon Infotech State of the Art Report, Pergamon Infotech, 1982.

10.- M. P. Atkinson, P. Baley, W. P. Cockshott, K. J. Chisholm and R. Morrison, "Progress whith persisntent programming", in P. Stoker et al. (eds), Database-Role and Structure, Cambridge University Press,1984.

11.- M. P. Atkinson, K. J. Chisoholm and W. P. Cockshott, "PS algol: an algol with a persistent heap", ACM Sigplan Notices 17(7), 24-31 (1982).

12.- R. Morrison, "S-algol reference manual", Tech. report cs 79/1, St. Andrews University, Department of Computer Science, 1979.

13.- M. P. Atkinson, K. J. Chisholm, W. P. Cockshott and R. Marshall, "Algorithms for a persistent heap", Software Practice and Experience, 13, 259-271 (1984).

14.- M. P. Atkinson, K. J. Chisholm, W. P. Cockshott,"CMS a chunck management system", Software Practice and Experience, 13, 273-285 (1983).

15.- M. P. Atkinson, K. J. Chisholm, W. P. Cockshott,and R. Morrison, "An approach to persistent programming", The Computer Journal, 26, (4), 360-365 (1983).

16.- C. J. Date, An Introduction to Database Systems vol 2, Fourth Edition, Addison Wesley.

17.- Brian W. Kernighan, Densis M. Ritchie, de la primera edición en inglés, Prentice Hall, El Lenguaje de Programación C.

18.- Samuel P. Harbison, Guy L. Steele Jr.C A Reference Manual, Tartan Laboratories, Prentice Hall.

19.- Brian W. Kernighan, First Edition, The Unix Programing Environment, Prentice Hall.

20.- Aho, A. V., J. E. Hopcroft, and J. D. Ullman, "Data structures and Algorithms", Addison Wesley.

21.- Wirth N. Algoritms+Data Structures=Programs, 'Prentice Hall 1976

22.- Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers principles, Techniques, and Tools, Prentice Hall 1984.

23.- Jeffrey D. Ullman, Principle of Database Systems, Computer Science 1983.

24.- Renato Barrera Rivera, Félix Fco. Ramos Corchado, C_persistenteMemoria s del IIISimp o sium de Informatica, Tecnológico de Toluca Mex. Mex

25.- Félix Fco. Ramos Manual de usuario del lenguaje C_persistente, CIEA.

El jurado designado por la Sección de Computación del Departamento de Ingeniería Eléctrica del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional. Aprobó esta tesis el día 3 de Julio de 1989.

M. en C. J. Carlos Pérez Castañeda

Dr. Josef Kolar Sabor

M. en C. Oscar Olmedo Aguirre

AUTOR RAMOS CORCHADO, F.F.

TITULO LENGUAJE C PERSISTENTE

CLASIF. XM 89.6 RGTRO. 81 11.221

NOMBRE DEL LECTOR	FECHA PREST.	FECHA DEVOL.
Luis Ramos Corchado R.	14-2-91	15 MAR 91