ESCUELA SUPERIOR POLITECNICA DEL LITORAL

FACULTAD DE INGENIERIA ELECTRICA

TOPICO DE GRADUACION
PREVIO A LA OBTENCION DEL TITULO DE

INGENIERO EN COMPUTACION

TEMA:

ADMINISTRACION DE MEMORIA Y PROGRAMACION EN AMBIENTE WINDOWS CON MICROSOFT C/C++ v7.00

PRESENTADO POR:

HARRY MARCELINO CHIQUITO CHOEZ

DIRECTOR: Ing. SIXTO GARCIA

GUAYAQUIL - ECUADOR

1992

AGRADECIMIENTOS

Agradezco a mis padres y hermanos, quienes con esfuerzo, dedicación, y estima me han brindado todo.
A mis profesores, por sus fecundas enseñanzas, en especial al Ing. Jaime Puente y al Ing. Sixto García por sus encomiables conceptos y doctrinas.
A mis amigos y compañeros, en particular a Víctor, y Mauricio.

DEDICATORIA

A mis padres... Que en todo momento supieron darme su apoyo y comprensión.

DECLARACION EXPRESA

" La responsabilidad por los hechos, ideas, doctrinas, expuestas en este documento, me corresponde exclusivamente; y el patrimonio intelectual de la misma a la Escuela Superior Politécnica del Litoral "

(Reglamento de Exámenes y Títulos Profesionales de la ESPOL)

_____

Harry Chiquito Chóez

II. EL MANEJO DE MEMORIA EN AMBIENTE WINDOWS

CAPITULO TRES

III. PROGRAMACION EN AMBIENTE WINDOWS

CAPITULO CUATRO

CAPITULO CINCO

CAPITULO SEIS

INTRODUCCION

Al cumplirse una década, después de la aparición del computador personal, y con los avances de la tecnología electrónica, el desarrollo y crecimiento de ésta herramienta de ayuda ha pasado por extraordinarios cambios, tanto a nivel de hardware, firmware, como al nivel de software.

Al nivel del software el computador personal tuvo sus raíces en el procesador 8080, y con el transcurso del tiempo pasa a través de los procesadores 8085, 8086, hasta llegar al primer computador personal comercial con un procesador 8088, el denominado XT.

Esta industria marcha de una manera acelerada, tanto así que el 80186 pasa casi desapercibido, y pronto nos encontramos con un computador personal que internamente tiene un procesador 80286 con aspectos muchos más avanzados que su antecesor.

En los últimos años aparecen los procesadores: 80386SX, 80386DX, 80486SX, 80486DX, 80486DX2, con amplias características (procesamiento, perfomance, dimensiones, y costo), que apenas hace 10 años atrás no estábamos en capacidad de siquiera imaginar.

El software nos brinda la alternativa de aprovechar las bondades ofrecidas por este tipo de herramientas, e inicialmente se desarrollan los denominados sistemas operativos, que para el PC se le da el nombre de sistema operativo del disco DOS, pero a medida que avanza la tecnología, se hace muy necesario actualizarlos.

Con la llegada del 80286, aparecen nuevas herramientas para tomar ventaja de sus características particulares y se originan aplicaciones con gráficas, ventanas, menús, etc.

Por el año de 1987 aparece un software por parte de Microsft Corporation, que permite aprovechar directamente las características de mejor direccionamiento de las localidades de almacenamiento de datos, tanto permanentes como virtuales, esto es a disco como a memoria respectivamente, este paquete se denomina Microsoft Windows.

Este Software toma ventaja de las bondades de los procesadores 80286 (y mayores), y su uso se extiende ampliamente, tanto a nivel de usuario como a programadores de los computadores personales, a tal punto que actualmente es considerado un sistema operativo.

Para luego encontrar herramientas, y aplicaciones especificas la mismas que necesariamente requieren que un ambiente Windows esté previamente ejecutandose para así poder correr sólo dentro de dicho ambiente.

Para desarrollar este tipo de aplicaciones se hizo necesario un proceso de madurez de diseño, análisis, y desarrollo de aplicaciones; pasando así por etapas definidas en el tipo o modelo de diseño que dependían del tipo de lenguajes de programación con que se contaba. Entre otros podemos citar estos tipos o modelo de diseño:

descomposición algorítmica, programación top-dows, programación estructurada, diseño modular, programación orientada a objetos, etc.

El C++ es un lenguaje que soporta el tipo de programación orientada a objetos, y Windows trata a los segmentos de memoria como objetos, y considerando que la memoria es de suma importancia en el desarrollo de una aplicación, he decidido hacer un breve análisis de las características de la programación orientada a objetos, y el manejo mediante programación de memoria dentro de un ambiente gráfico y amistosos al programador, como lo es MicroSoft Windows.

Además debido a que personalmente considero que en un corto periodo de tiempo este tipo de ambiente predominará en los computadores personales, por su facilidad y simplicidad que brinda en todo momento a cualquier tipo de usuario, sea este temporal o permanente.

El capítulo uno da antecedentes de lo que es un objeto y como enfocarlo de esta forma y como llegar a entenderlo para que nos ayude a simplificar la tarea de tratar la complejidad de un problema.

En el capítulo dos, se da conceptos generales para afirmar nuestro entendimiento de lo que es un objeto, y como conducirnos en la programación, diseño, análisis de este tipo. Además se mencionan los elementos que llegan a conformar un objeto como un ente particular.

A continuación en el capítulo tres, revisamos la administración interna de la memoria dentro de este ambiente, todo lo cual lo hace de una manera transparente, y se tratan los tipos de especificaciones que se le puede dar, de acuerdo a la configuración de hardware que tengamos a mano, y como optimar su perfomance.

El capitulo cuatro muestra como crear una pequeña aplicación que corra dentro de un ambiente Windows.

Finalmente los dos capítulos restantes tratan de como manejar la memoria mediante programación usando sentencias de lenguaje de programación C, y funciones internas que viene con el paquete de Microsoft C/C++ V7.00 u en el Microsoft Software Development Kit (SDK).

Hay que acotar que el cometido o alcance de este documento es el de dar una mejor manera de como llegar a administrar los recursos de memoria dentro del ambiente Windows, tanto al usuario que hace uso de aplicaciones finales, como a aquel que desee programar sus propias aplicaciones y hacer un mejor uso de las declaraciones de tipos, y atributos de segmentos de memoria que usa en su programa. No se pretende alcanzar un alto grado de programación de aplicaciones sino de entender como hacer funcionar la misma optimando la utilización de recursos de memoria.

I LA PROGRAMACIÓN ORIENTADA A OBJETO

CAPITULO UNO

CONCEPTOS

El problema de la complejidad en los sistemas ha merecido a través del tiempo especial atención, por lo que se han buscado incesantemente mecanismos y formas para conllevarla de una mejor manera. La Programación Orientada a Objetos es un método que nos ayuda a administrar la complejidad inherente al software y las maneras como esta se manifiesta.

1.1. ANTECEDENTES

Así como en el mundo físico tenemos objetos que son completamente complejos, por ejemplo, el proceso de la fotosíntesis, la estructura de la materia, la precipitación de los glóbulos blancos para atacar un virus, etc., el Software también tiene elementos de gran complejidad; sin embargo, la complejidad que encontramos aquí es fundamentalmente de diferente clase.

Por ejemplo, nos percatamos que ciertas aplicaciones en si no son complejas, sino que se transforman en tales por su concepción, y que son totalmente olvidadas con el tiempo, desde un cierto punto de vista, este tipo de aplicaciones son especificadas, construidas, mantenidas y usadas por una sola persona - en general los programadores principiantes o los programadores profesionales trabajan de una manera aislada.

Esto no quiere decir que tales sistemas sean malos o que se trate de empequeñecer a sus creadores, sino que dichos sistemas tienden a tener un propósito limitado o un periodo de vida muy corto. Podemos permitirnos salir de ellos y reemplazarlos totalmente con un nuevo software en lugar de intentar volver a usarlos, repararlos, y/o extender su funcionalidad. Este tipo de sistemas son generalmente mas tediosos que dificultosos de desarrollar.

En otros tipos de aplicaciones como son los sistemas que mantienen la integridad de miles de registros de información mientras se permite la actualización concurrente y consultas de acceso; o sistemas que comandan o controlan entidades del mundo real, como el tráfico aéreo o ferroviario, tienden a tener un largo periodo de vida, y durante el tiempo, muchos usuarios vienen a depender de su adecuada funcionalidad.

La característica que distingue a tales software, es que los del segundo tipo son intensamente dificultosos, sino imposible para un programador individual comprender todos las sutilezas de su diseño. Expresado de otra manera, la complejidad de tales sistemas excede la capacidad intelectual humana.

Ciertamente, siempre habrá un genio entre nosotros, gente de extraordinaria capacidad que pueden hacer el trabajo de un grupo de programadores, el ingeniero de software equivalente a Frank Lloyd Wright o Leonardo da Vince. Sin embargo como Peter señala: "El mundo está solamente poblado con escasos genios. No hay razón para creer que la comunidad de ingeniería de software tiene

una gran proporción casi exagerada de ellos.". Si bien hay una pizca de genio en todos, en el reino de la industria del software no siempre podemos confiarnos de esta inspiración divina para realizarnos.

<< Por lo tanto debemos considerar una manera mas disciplinada para administrar la complejidad. >>

Inherentemente la complejidad se deriva de cuatro elementos:

La complejidad del dominio del problema : los usuarios generalmente encuentran que es muy duro dar la expresión precisa a sus necesidades de tal forma que el programador la pueda entender. Los usuarios y programadores tienen diferentes perspectivas sobre la naturaleza del problema y pueden hacer suposiciones respecto a la naturaleza de la solución.

La dificultad de manejar el proceso de desarrollo : la cantidad de trabajo que demanda la creación de un sistema hace necesario el uso de un equipo de programadores, e idealmente usar el grupo tan pequeño como sea posible, sin embargo siempre habrá un significante desafío asociado con el equipo de programadores. Entre más programadores existan, más compleja se hace la comunicación, y así más dificultosa la coordinación, particularmente si el equipo de desarrollo está geográficamente alejado, como con frecuencia ocurre con los proyectos muy grandes.

La flexibilidad a través del software : el software ofrece una flexibilidad esencial, de tal forma que es posible al programador expresar casi cualquier clase de abstracción. Lo cual obliga al programador a construir bloques de acuerdo a las primitivas sobre las cuales estas abstracciones de alto nivel permanecen. Mientras que en otras industrias como la de construcción tienen códigos y estándares para su cometido, en la industria del software pocos estándares existen.

El problema de caracterizar el comportamiento de sistemas discretos : en los sistemas reales rigen las leyes físicas por las cuales podemos predecir el comportamiento de ciertos sistemas, en los sistemas de software no podemos predecir lo que ocurrirá si un programador realiza una mala suposición de ciertos parámetros o variables en un módulo, y como repercutirá esto en el resto de módulos del sistema. Ciertamente esto es la principal motivación de vigorosas pruebas a nuestros sistemas, pero para todos, excepto para los mas triviales, las exhaustivas pruebas son casi , sino, imposibles.

1.2 EL PAPEL DE LA DESCOMPOSICIÓN

Debido a lo anteriormente expuesto se hace necesario encontrar alternativas de soluciones para la administración y ejecución de sistemas complejos. Como sugirió Dijkstra "La técnica de administrar ha sido conocida desde antaño: divide e impera (divide y gobierna)". Cuando diseñamos un sistema complejo de software, es esencial descomponer esto en partes más y más pequeñas, cada una de las cuales puede ser independientemente refinadas. De esta manera, satisfacemos la restricción que existe sobre el canal de capacidad de entendimiento de conocimiento humano: para entender cualquier nivel del

sistema dado, necesitamos comprender unas pocas partes (en lugar de todas las partes) a la vez.

1.2.1. LA DESCOMPOSICIÓN ALGORITMICA

La mayoría de nosotros hemos sido formalmente guiados a el dogma del diseño estructurado top-down, y lo enfocamos a una forma de secuencias simple de descomposición algoritmica, donde cada módulo del sistema denota algún proceso general.

1.2.2. LA DESCOMPOSICIÓN ORIENTADA A OBJETO

Una manera diferente para descomposición del mismo problema, es la descomposición acorde a una clave de abstracción en el dominio del problema. De tal manera que en lugar de descomponer el problemas en pasos sucesivos, se identifica objetos como un conjunto de agentes autónomos que colaboran para ejecutar algún comportamiento de alto nivel. Cada objeto en nuestra solución enmarca su propio comportamiento único, y cada uno de los cuales modela algún objeto en el mundo real. Desde esta perspectiva , un objeto simplemente es una entidad tangible la cual exhibe un comportamiento bien definido. Debido a que la descomposición sobre objetos y no algoritmos, se la denomina descomposición orientada a objeto.

1.2.3. LA DESCOMPOSICIÓN ALGORITMICA CONTRA LA DESCOMPOSICIÓN ORIENTADA A OBJETO

Cual es la manera correcta para descomponer un problema ? Actualmente, esta es una pregunta engañosa, debido a que la respuesta correcta sería que los dos puntos de vistas son importantes: el punto de vista algorítmico destaca los eventos ordenados, en tanto que el punto de vista orientado a objeto destaca los agentes que causan acciones o son los sujetos sobre los cuales estas operaciones actúan. Sin embargo el hecho relevante es que no podemos construir un sistema complejo en las dos maneras simultáneamente, por que son completamente ortogonales. Podemos iniciar la descomposición de nuestro sistema en cualquiera de las dos maneras y usar la estructura resultante como el armazón para expresar la otra perspectiva.

Es aconsejable dar primero dar un punto de vista orientado a objeto, debido a que este es mejor en ayudarnos a organizar la inherente complejidad de un sistema de software. La descomposición orientada a objeto tiene un número de ventajas significativas sobre la descomposición algorítmica. La descomposición orientada a objeto produce sistemas pequeños a través del uso de mecanismos comunes, proveyendo así una economía importante de expresiones.

Los sistemas orientados a objetos son más moldeable a cambios y capaces de evolucionar con el tiempo, debido a que su diseño está basado sobre formas intermedias estables. Realmente reduce el riesgo en la construcción de sistemas complejos, ya que están diseñados para evolucionar de una manera creciente desde pequeños sistemas de los que ya tenemos confianza y seguridad en su funcionamiento. Además dirige directamente la complejidad del sistema de

software, ayudándonos a tomar decisiones inteligentes considerando la separación de intereses en un gran espacio de estados.

1.3 EL PAPEL DE LA ABSTRACCIÓN

Experimentos psicológicos, tal como el de Miller, sugieren que el máximo número de segmentos de información que un individuo puede simultáneamente comprender está en el orden de los siete, más / menos dos. Lo cual parece ser independiente del contenido de la información. Este canal de capacidad parece estar relacionado a la capacidad de corto periodo de la memoria. Simon adicionalmente anota que la velocidad de procesamiento es un factor condicionante : le toma a la memoria humana alrededor de cinco segundos aceptar un nuevo segmento de información.

Como Miller observa, " La extensión del juicio absoluto y la extensión de la memoria inmediata impone varias limitaciones de la cantidad de información que somos capaces de recibir, procesar y recordar. Organizando los estímulos de entrada simultáneamente en varias dimensiones y sucesivamente en una secuencia de segmentos, somos capaces de romper ... este cuello de botella informativo". El llama a este proceso 're-codificación', lo cual es conocido como la abstracción.

Como Wulf describe " Los humanos hemos desarrollado una técnica excepcionalmente poderosa para tratar con la complejidad. La resumimos. Al no ser capaces de administrar la totalidad de un sistema complejo, ignoramos los detalles innecesarios, tratando a su vez con lo generalizado, y modelamos idealmente los objetos". Todo lo cual lo realizamos a través de la abstracción.

1.4 EL PAPEL DE LA JERARQUÍA

Otra manera de incrementar el contenido semántico de los segmentos individuales de información, es explícitamente reconocer tanto la jerarquía en clase como en objeto dentro de un sistema complejo de software. La estructura de objetos es importante porque ilustra como los diferentes objetos colaboran con algún otro a través de la interacción de patrones a los cuales se les denomina 'mecanismos'. La estructura de las clases a su vez también es importante, debido a que destaca la redundancia dentro de un sistema.

Identificar las jerarquías dentro de un sistema complejo no es sencillo, debido a que esto requiere de la identificación de patrones entre varios objetos, cada de los cuales puede enmarcar un comportamiento tremendamente complicado. Pero una vez que hemos expuesto estas jerarquías, la estructura de un sistema complejo, y nuestro entendimiento del mismo, se ve simplificado en grado sumo.

CAPITULO DOS

UNA REVISIÓN DEL MODELAMIENTO DE OBJETOS

En este capítulo se da un breve resumen de las características del modelamiento de objetos, así como de la programación orientada a objetos - un estilo de programación específico que soporta el C++ -.

2.1. QUE ES LA PROGRAMACION ORIENTADA A OBJETOS ?

Si bien el término Object Oriented Programming (OOP) es ampliamente usado, hay una falta de consenso cuando se lo trata de definir. Muchas personas prefieren libremente pensarla como una nueva manera de modelamiento de software basado en el mundo real, y otras como una forma dada para el uso de términos tales como Tipos Abstractos de Datos (ADT), y decir al respecto como el uso de OOP implica el uso de una colección de ADT's. Tomemos el enfoque menos riguroso y describamos al OOP como una nueva manera de organizar nuestros programas.

OOP es sólo un método de diseño e implementación de software. El uso de OOP en si no imparte alguna cosa al producto de software final que el usuario pueda ver. Sin embargo, quien desarrolla software puede ganar algunas ventajas del uso de los métodos de OOP, especialmente en proyectos de software grandes. Debido a que OOP le permite permanecer casi al nivel conceptual, - el modelo de mas alto nivel del problema del mundo real que deseemos resolver - , podemos manejar la complejidad mucho mejor con este, que con enfoques que obliguen a mapear el problema para fijarlo a las características del lenguaje. Podemos tomar ventaja de la modularidad de objetos e implementar el programa en unidades relativamente independientes que son mantenidos (administrados) separadamente. Podemos compartir código y datos entre objetos a través de la herencia.

LA PROGRAMACION ORIENTADA AL PROCEDIMIENTO

Antes de entrar al OOP, echaremos un rápido vistazo a la programación convencional usando lenguajes tales como C. Por la falta de un mejor término, usaremos el término programación orientada al procedimiento, para describir las técnicas de programación convencionales.

En un enfoque a la programación orientada al procedimiento, vemos al problema como una secuencia de cosas que hacer. Escribimos un número de funciones (procedimientos) que nos permiten completar una secuencia de tareas. El dato es organizado en estructuras, pero su enfoque principal está siempre en las funciones. Una función trasforma un dato de alguna manera, por ejemplo Ud. puede tener una función para adicionar un conjunto de números, otra para calcular la raíz cuadrada, y una para mostrar una cadena. Ud. no tendrá que ir muy lejos para encontrar este tipo de organización - la librería run-time del C esta implementada de esta forma-. Cada función en la librería realiza una operación bien definida en los argumentos de entrada, y retorna el dato transformado como un valor de retorno, a través de un puntero a una localidad de almacenamiento, o directamente a un dispositivo de salida como la pantalla de visualización.

Esto no quiere decir que en un enfoque orientado al procedimiento Ud. no tenga cuidado acerca de la organización de los datos. De hecho, Ud. pone una estrecha atención a los datos para manipularlos mediante una aplicación, y usualmente organiza piezas relacionadas de datos en una unidad usando las estructuras de C. Entonces escribe la función que opera con esta estructura de datos. No hay un lazo estrecho entre los datos y las funciones. OOP es diferente en como los datos y funciones son agrupados juntas.

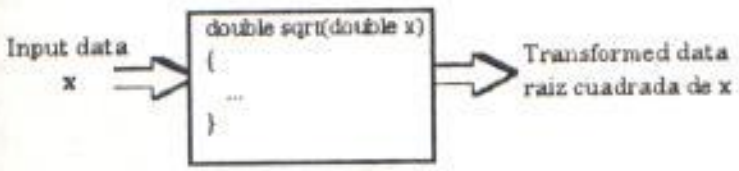

Figura 2-1. Una función transforma datos

2.2 BASES DE LA PROGRAMACION ORIENTADA A OBJETOS

Debido a que el modelamiento de un objeto se deriva de muchas fuentes dispares, este ha sido acompañado desafortunadamente por una terminología confusa. Por ejemplo un programador en Smalltalk usa 'methods', un programador en C++ usa 'virtual member functions', y un programador en CLOS (Common List Object System) usa 'generic functions'. Un programador en Object Pascal habla de 'type coersion', un programador en Ada le denomina a lo mismo 'type conversion'. Para minimizar la confusión definiremos lo que es un object-oriented, y que no lo es.

Anteriormente se definió informalmente a un objeto como una entidad tangible que exhibe un comportamiento bien definido. Stefik y Bobrow definen a los objetos como "entidades que combinan las propiedades de los procedimientos y datos puesto que ellos ejecutan cálculos y salvan un estado local". La definición de objetos como entidades da por sentado algo de lo que se trata de probar, pero el concepto básico es que los objetos sirven para unificar las ideas de los algoritmos y la abstracción de datos. Un objeto sólo puede ser cambiado de estado, actuar, ser manipulado, o mantenerse en relación a otros objetos en una manera apropiada a este objeto. Expuesto de otra manera, existen propiedades invariables que caracterizan a los objetos así como su comportamiento. Por ejemplo un ascensor, está caracterizado por la propiedad invariable de viajar sólo hacia arriba y hacia abajo en un espacio determinado... cualquier simulación de un elevador debe incorporar esta propiedad para que tenga una noción integral de ascensor.

2.2.1 LA PROGRAMACION ORIENTADA A OBJETOS (OOP)

A diferencia de la programación orientada al procedimiento, OOP trata a los datos como entes primarios y a las funciones como entes secundarios. Usted diría que en lugar de los datos sean lo que las funciones usan, las funciones son lo que los datos hacen. En lugar de permanecer aisladas, las funciones son estrechamente asociadas con los datos. Si el lenguaje de programación soporta OOP, usualmente hay una sintaxis específica que Ud. puede usar para indicar esta estrecha relación (por ejemplo la construcción class del C++).

Grady Booch da esta definición :

"La programación orientada a objetos es un método de implementación en el cual los programas están organizados como una colección de objetos cooperativos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son todas miembros de una jerarquía de clase unidas por medio de relaciones de herencias".

Hay tres partes importantes en esta definición: la programación orientada a objetos (1) usa objetos, no algoritmos como sus bloques fundamentales lógicos de construcción; (2) cada objeto es una instancia de alguna clase; y (3) las clases están relacionadas por medio de la relación de herencia. Un programa puede parecer estar orientado a objetos, pero si alguno de estos elementos no es encontrado, no es un programa orientado a objetos. Específicamente, la programación sin herencias es inequívocamente no orientada a objetos, le denominamos a esta programación con tipo abstracto de datos.

De acuerdo a esta definición, algunos lenguajes son orientados a objetos y otros no lo son. El termino 'lenguaje orientado a objetos' significa que el mismo tiene un mecanismo que soporte bien el estilo de programación orientada a objetos. Un lenguaje soporta un estilo de programación si este provee las facilidades para hacer un uso conveniente de dicho estilo. Un lenguaje no soporta una técnica determinada si hay que tomar esfuerzo y destrezas excepcionales para escribir programas con tales técnicas.

Cardelly y Wegner dicen que :

" un lenguaje es orientado a objetos si y sólo si este satisface los siguientes requerimientos:

* este soporta objetos que son abstracciones de datos con una interface de operaciones designadas y un estado local oculto

* los objetos tienen un tipo asociado (clases)

* los tipos pueden heredar atributos de supertipo (superclases)"

Si un lenguaje no provee directamente soporte para la herencia, entonces este no es orientado a objetos. Cardelly y Wegner distinguen tales lenguajes denominándolos object-based (basados en objetos), en lugar de object-oriented (orientado a objetos). Bajo esta definición Smalltalk, C++, Object Pascal, y CLOS son object-oriented, y Ada es object-based.

EL DISEÑO ORIENTADO A OBJETOS (OOD)

El énfasis en los métodos de programación está principalmente sobre las propiedades y el uso efectivo de mecanismos de lenguajes particulares. Por el contrario, los métodos de diseño resalta las propiedades y la estructuración efectiva de un sistema complejo. Esto es object-oriented design ?. Booch sugiere:

"El diseño orientado a objetos es el método de diseño que encuadra los procesos de descomposición object-oriented y una notación para describir tanto los

modelos lógicos como físicos así como los modelos estáticos y dinámicos del sistema bajo diseño".

Hay dos partes importantes en esta definición: el diseño orientado a objetos (1) conduce a una descomposición orientada a objetos y (2) usa diferentes notaciones para expresar diferentes modelos del diseño lógico (estructura de clases y objetos) y físico (arquitectura modular y de procesos) de un sistema.

EL ANALISIS ORIENTADO A OBJETOS (OOA)

El análisis orientado a objetos (object-oriented analysis) resalta la construcción de modelos del mundo real usando un punto de vista orientado a objetos del mundo.

"El análisis orientado a objetos es un método de análisis que examina los requerimientos desde una perspectiva de las clases y objetos encontrados en el vocabulario del dominio del problema"

Como están relacionados OOP, OOD, y OOA?
Básicamente, el producto del análisis orientados a objetos puede servir como el modelo del cual podemos iniciar un diseño orientado a objetos; el producto del diseño orientado a objetos entonces puede ser usado como un plan detallado para implementar completamente un sistema usando métodos de programación orientada a objetos.

ELEMENTOS DEL MODELAMIENTO DE OBJETOS

Bobrow y Stefik definen al estilo de programación como: "una manera de organizar los programas sobre la base de un algún modelo conceptual de programación y un lenguaje apropiado para hacer programas escritos en un estilo claro". Además sugieren que hay cinco clases principales de estilos de programación (listados con la clase de abstracción que emplean):

* orientados al procedimiento	algoritmos
* orientados a objetos	clases y objetos
* orientados a la lógica	metas
* orientados a las reglas	If - then
* orientados a la sujeciones (restricciones)	relaciones invariables

No hay un simple estilo de programación que sea el mejor para todas las clases de aplicaciones. Por ejemplo la programación orientada a las reglas será mejor para el diseño de una base del conocimiento. El estilo de programación orientado a objetos, desde nuestro punto de vista, es lo más apropiado para aplicaciones en las cuales la complejidad es predominante.

Cada uno de estos estilos esta basado sobre su propio armazón conceptual, cada uno requiere un razonamiento diferente, una diferente manera de pensar acerca

del problema. Para todas las cosas orientadas a objetos, el armazón conceptual es el modelamiento de objetos. Hay cuatro grandes elementos en este modelo:

* la abstracción

* el encapsulamiento

* la modularidad

* la jerarquía

por 'grandes', entendemos que un modelo sin alguno de estos elementos no es orientado a objetos.

Hay tres menores elementos en este modelo:

* la representación

* la concurrencia

* la persistencia

por 'menores', entendemos que cada uno de estos elementos son útiles, pero no son parte esencial del modelamiento de objetos.

Sin este armazón conceptual usted puede estar programando en lenguajes tales como Smalltalk, C++, Object Pascal, CLOS, o Ada, pero su diseño es tan complejo y confuso como una aplicación en FORTRAN, Pascal, o C. Usted habrá perdido la confianza puesta, o en otro caso abusado del poder de expresión de lenguaje basado u orientado a objetos que esta usando para implementación. Más aún, es probable que usted no tenga dominada la complejidad con la que esta tratando.

2.2.1. LA ABSTRACCION DE DATOS

Booch define a la abstracción de la siguiente manera :

"Una abstracción denota la características esenciales de un objeto que lo distingue de otras clases de objetos y provee de esta manera límites conceptuales definidos, relativos al punto de vista del observador"

Una abstracción enfoca un punto de vista externo de un objeto, y sirve así para separar el comportamiento esencial del objeto desde su implementación.

Seidewitz y Stark sugieren que "hay un espectro de abstracción, desde objetos que estrictamente modelan entidades en el dominio del problema, hasta objetos los que no tienen razón de ser". Desde los más hasta los menos útiles, estas clases de abstracciones incluye lo siguiente :

* Abstracciones de entidades	Un objeto que representa un modelo útil de una entidad en el dominio del problema

* Abstracción de Acciones	Un objeto que provee un conjunto de operaciones generalizadas, las mismas que ejecutan las mismas clases de funciones
* Abstracción de máquina virtual	Un objeto que agrupa operaciones en común la mismas que son usadas por algún nivel de control superior, u operaciones que usen todas un conjunto de operaciones de algún nivel inferior
* Abstracción coincidencial	Un objeto que empaqueta un conjunto de operaciones que no tienen relaciones entre sí

Todas las abstracciones tienen propiedades estáticas como dinámicas. Por ejemplo un objeto archivo toma un acierta cantidad de espacio en un dispositivo de memoria particular; este tiene un nombre, y tiene contenido. Todas estas son propiedades estáticas. El valor de cada una de estas propiedades son dinámicas, relativo al tiempo de vida del objeto: un objeto archivo puede crecer o reducirse en tamaño, su nombre puede ser cambiado, su contenido puede cambiar.

En un estilo de programación orientado al procedimiento, las actividades que cambian los valores dinámicos de los objetos es la parte central de todos los programas: las cosas suceden cuando los subprogramas son llamados y las rutinas se ejecutan. En el estilo de programación orientada a las reglas : las cosas ocurren cuando nuevos eventos causan que las reglas se activen lo cual puede hacer que otras reglas se disparen, y así sucesivamente. En el estilo de programación orientado a objetos : las cosas suceden cada vez que nosotros operamos sobre un objeto. Así, invocando una operación sobre un objeto obtenemos alguna reacción proveniente del objeto. Que operaciones podemos realizar sobre el objeto, y como reacciona el mismo, constituye el comportamiento de un objeto.

EJEMPLO DE ABSTRACCION

Como un programador de C, Ud. está familiarizado con las rutinas de entrada y salida de archivos en la librería de run-time. Estas rutinas ven al archivo como un flujo de bytes y a su vez le permiten realizar varias operaciones con este flujo. Por ejemplo Ud. puede abrir un archivo (fopen), cerrarlo (fclose), leer de este un caracter (getc), escribirle un caracter (putc), y así sucesivamente. Este modelo abstracto de un archivo es implementado por la definición de un nuevo tipo de dato, FILE, usando el mecanismo typedef del lenguaje C .

Sin embargo, para usar el tipo de dato FILE, Ud. nunca tendrá que ver o preocuparse de la estructura en C que define este tipo. De echo, la estructura de datos FILE pueden variar de un sistema a otro. Aunque, las rutinas de I/O de archivos en C, trabajan de la misma manera en todos los sistema. Esto es conocido como ocultación de información o ocultación de datos.

La abstracción de datos es la combinación de la definición de tipos de datos y la ocultación de datos. Así, el tipo de dato FILE es un ejemplo de abstracción de datos.

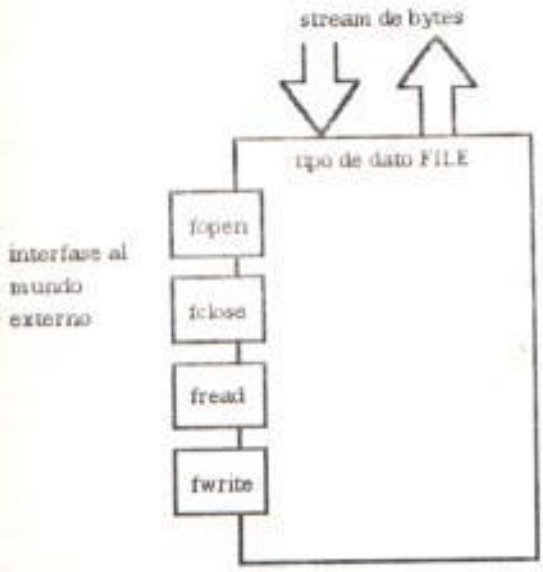

Figura 2-2. El tipo de dato FILE en C es un ejemplo de Abstracción de datos

LOS OBJETOS Y LA ABSTRACCION DE DATOS

Usted puede usar la idea de abstracción de datos para crear un objeto por la definición de un bloque de datos junto con las funciones necesarias para operar sobre estos datos. El dato representa la información contenida en el objeto y las funciones modelan el comportamiento del objeto - definen las operaciones que pueden ser ejecutadas en el objeto. El dato no es accesible al mundo externo. La única manera de hacer algo al objeto es llamando una de las funciones que implementa el comportamiento del objeto.

Así, el tipo de dato FILE en C junto con las rutinas de I/O pueden ser pensadas como la definición de un objeto. Ud. crea un objeto FILE por medio del llamado a la función fopen, la cual retorna un puntero a una estructura FILE. Ud. accede a este objeto FILE por medio de este puntero. Cada vez que usted llame a fopen, crea un nuevo objeto file. Ud. puede pensar del tipo de dato FILE como un template que es usado para generar nuevas instancias de FILEs cuando ellos llamen a fopen. El término objeto es referido a la instancia.

CLASES Y METODOS

En la terminología OOP, el template que define un tipo de dato objeto, usualmente es denominado class - el término puede diferir de un lenguaje OOP a otro. Así, cada objeto es una instancia de la clase.

Las funciones que operan sobre un objeto tienen un nombre especial - son conocidos como métodos 'methods' debido a que fue el nombre usado en el lenguaje orientado a objetos Smalltalk. El método define el comportamiento de un objeto. En C++, los métodos son llamados funciones miembros (member function) de la clase.

Smalltalk también da otro concepto a OOP - que es el de remitente de mensajes a un objeto. Esto se refiere al acto de instruir a un objeto para que realice una

operación invocando uno de los métodos. En C++ esto es hecho por el llamado de la función miembro apropiada del objeto.

2.2. ENCAPSULAMIENTO

La abstracción de un objeto precede las decisiones alrededor de su implementación. Una vez que una implementación es seleccionada, esta debería ser tratada como un secreto y ocultarla para la mayoría de clientes. Como Ingalls sugiere, "Ninguna parte de un sistema complejo debería depender en los detalles internos de otra parte". Puesto que la abstracción "ayuda a las personas a pensar acerca de lo que ellos están haciendo", el encapsulamiento "permite los cambios a un programa sean confiablemente hechos , con mínimo esfuerzo".

La abstracción y el encapsulamiento son conceptos complementarios : la abstracción está enfocada sobre el punto de vista externo de un objeto, y el encapsulamiento - también conocida como ocultación de de información - impide a los clientes ver desde un punto de vista interno, donde el comportamiento de la abstracción esta implementado. De esta manera, el encapsulamiento provee una barrera explícita entre las diferentes abstracciones.

Considerando la estructura de una planta, podemos entender como trabaja la fotosíntesis en un nivel alto de abstracción, podemos ignorar detalles tales como la función de las raíces y la mitocondria en el interior de las células de una planta.

Liskov sugiere que "para que las abstracción trabaje, la implementación debe estar encapsulada", en la práctica, esto significa que cada clase debe tener dos partes : una interface y un implementación. La interfase de una clase captura solo su punto de vista externo, abarcando el comportamiento común de todas las instancias de la clase. La implementación de un clase comprende la representación de la abstracción, así como los mecanismos llevan a cabo el comportamiento deseado. Esta división explícita de interface / implementación, representa una clara separación de inquietudes: la interface de una clase el lugar único donde declaramos todas las suposiciones que el cliente haría acerca de cualquier instancia de la clase; la implementación encapsula detalles de los cuales ningún cliente puede tomar o hacer suposiciones.

Booch define el encapsulamiento como :

"El encapsulamiento es el proceso de ocultar todos los detalles de un objeto, que no contribuyen a sus características esenciales".

2.3. MODULARIDAD

Como Myer observa, "el acto de seccionar un programa en componentes individuales puede reducir su complejidad en algún grado... . Si bien la partición de un programa es útil por esta razón, una justificación más poderosa para seccionar un programa es que éste crea un número de regiones bien definidas y documentadas dentro del programa. Estas regiones, o interfases, son invalorables en la comprensión de un programa".

En algunos lenguajes, tales como Smalltalk, no hay el concepto de módulo, y así la clase forma sólo la unidad física de descomposición. En muchos otros, incluyendo Object Pascal, C++, CLOS, y Ada, el módulo es un constructor de lenguaje separado, y por lo tanto garantiza un conjunto separado de decisiones de diseño. En estos lenguajes, las clases y los objetos forman la estructura lógica de un sistema; nosotros situamos estas abstracciones en 'módulos separados para producir la arquitectura del sistema físico. Especialmente para aplicaciones grandes, en las cuales tenemos cientos de clases, el uso de módulos es esencial para ayudar a administrar la complejidad.

Liskov formula que "la modularización consiste de la división de un programa en módulos los cuales pueden ser compilados separadamente, pero los mismos tienen conexiones con otros módulos". Parma define : "las conexiones entre módulos son las suposiciones que hacen los módulos acerca de cada uno de los otros".

La mayoría de lenguajes que soportan el módulo como un concepto separado también distingue entre la interface de un módulo y su implementación. Así, es admisible decir que la modularidad y el encapsulamiento van a la par. Como en el caso del encapsulamiento lenguajes particulares soportan la modularidad de diferentes maneras. Por ejemplo, módulos en C++ no son más que archivos compilados separadamente. La práctica tradicional de la comunidad del C / C++ es situar los módulos de interfases en archivos nombrados con el sufijo '.h'; los mismos que son denominados ' header files '. Los módulos de implementación son situados en archivos con nombres con un sufijo en '.c'. Las dependencias entre archivos entonces pueden ser declaradas usando la macro '#include'. Esta enfoque es totalmente una convención; no es un requisito ni una imposición del lenguaje mismo.

Decidir sobre el correcto conjunto de módulos para un problema dado es tan difícil como el problema de decidir sobre su correcto conjunto de abstracciones. Los módulos es como un container en el cual declaramos las clases y objetos de nuestro diseño físico. Esto no es diferente a la situación encontrada por un ingeniero eléctrico diseñando una tarjeta de computador. Puertas NAND, NOR, o NOT, deberían ser usadas para construir la lógica necesaria, pero estas puertas deben estar físicamente empaquetadas en un circuito integrado estándar, tal como el 7400, 7402, o 7404. A falta de alguna parte de software estándar, el ingeniero de software tiene un grado de libertad considerablemente mayor - como si el ingeniero eléctrico tuviera su propia fundición (fabrica) de silicio a su disposición.

En el diseño estructurado tradicional, la modularización es principalmente ocupada con la significativa agrupación de subprogramas, usando el criterio de acoplamiento y cohesión. En el diseño orientado a objetos, es problema es similmente diferente: la tarea es decidir donde empaquetar físicamente las clases y los objetos del diseño lógico estructurado, los cuales son inequívocamente diferentes de los subprogramas.

Hay varias técnicas útiles, así como guías no técnicas, que pueden ayudarnos a alcanzar una modularización inteligente de clases y objetos. Como Britton y Parnas han observado, "La meta global de la descomposición de módulos es la reducción del costo de software, permitiendo que los módulos sean revisados y diseñados independientemente... cada una de las estructuras de los módulos

serían lo suficientemente simples tal que puedan ser entendidas completamente; sería posible cambiar la implementación de ciertos módulos sin el conocimiento de la implementación de los otros módulos y sin afectar su comportamiento; lo fácil de hacer un cambio en el diseño apoyaría una razonable relación para la probabilidad de que los cambios sean necesarios".

Hay un margen pragmático en esta norma. En la práctica el costo de re-compilar un módulo es relativamente pequeño : sólo la unidad necesita ser re-compilada y la aplicación re-enlazada. Sin embargo el costo de recompilar la interface de un módulo es relativamente alto. Especialmente con los lenguajes **totalmente simbolizados**, uno debe recompilar la interface modular, el módulo, todos los módulos que dependen sobre esta interfase, todos los módulos que dependen sobre este módulo, y así sucesivamente. Así, para un programa sumamente grande, un cambio en un simple módulo puede resultar en muchas horas de compilación. Obviamente un administrador no puede permitirse esto. Por esta razón un interface modular debe ser tan minuciosa como sea posible, y satisfacer todas las necesidades de los módulos usados. Nuestro estilo es ocultar tanto como podamos en la implementación de un módulo.

El programador por lo tanto debe balancear dos asuntos técnicos importantes : el deseo de encapsular las abstracciones, y la necesidad de hacer ciertas abstracciones visibles a otros módulos.

Booch define :

"La modularidad es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos coherentes y libremente acoplados".

Así, los principios de abstracción, encapsulamiento, y modularidad son "sinergísticos", es decir que trabajan de manera conjunta para obtener un resultado global mucho mejor del que se obtendría si actuaran de manera independiente. Un objeto provee una frágil región alrededor de una abstracción simple, y tanto el encapsulamiento como la modularidad proveen una barrera alrededor de la abstracción.

Dos observaciones técnicas adicionales pueden afectar la decisión de modularización. Primero, puesto que los módulos usualmente sirven como las unidades elementales e indivisibles del software estos pueden ser usados enteramente a través de las aplicaciones, para programador es posible empaquetar las clases y objetos en módulos de tal manera que hagan su uso conveniente. Segundo, muchos compiladores generan código objeto en segmentos, uno para cada módulo. Por lo tanto, habrá límites prácticos en el tamaño de módulos individuales. Con atención a las llamadas dinámicas de subprogramas, la localización de las declaraciones dentro de los módulos pueden afectar en mayor forma las localidades de referencia y así el comportamiento de un sistema de memoria virtual.

JERARQUIA

Un objeto del mundo real a menudo es una extensión de un objeto existente. Por ejemplo, a menudo describimos algo usando sentencias tales como: Y es tal como X excepto que Y tiene ... y Y hace ... . Cuando hacemos esto, estamos definiendo

un nuevo objeto indicando como las características y comportamiento del nuevo objeto difieren de uno anterior.

EL SIGNIFICADO DE LA JERARQUIA

La abstracción es provechosa, pero en todas las aplicaciones excepto en las más triviales, encontraremos muchos más abstracciones de las que podamos comprender en un momento. La encapsulación ayuda a manejar la complejidad ocultando un punto de vista interno de nuestra abstracción. La modularidad también nos ayuda, dándonos una manera para agrupar lógicamente todas nuestras abstracciones relacionadas. Al parecer esto no es suficiente. Un conjunto de abstracciones a menudo forma una jerarquía, e identificando esta jerarquías en nuestro diseño, simplificamos enormemente nuestro entendimiento del problema.

Booch define a la jerarquía de la siguiente manera :

"La jerarquía es una distinción u ordenamiento de las abstracciones".

Las dos más importantes jerarquía en un sistema complejo son su estructura de clase (la ' clase de ' jerarquía), y su estructura de objeto (la ' parte de ' jerarquía). OOP soporta esta noción de definiciones de un nuevos objeto a partir de uno anterior. El termino herencia es usado para este concepto debido a que Ud. puede pensar de un objeto heredando propiedades de otro, o, mas correctamente, una clase hereda el comportamiento de otra clase. La herencia impone una relación jerárquica de padre a hijo. La clase padre es a menudo llamada super clase o clase base.

HERENCIA

La herencia es la más importante ' clase de ' jerarquía, como se anotó anteriormente, este es un elemento esencial de sistemas orientados a objetos. Básicamente la herencia define una relación entre clases, en la cual una clase comparte la estructura o comportamiento definido en una o más clases (denominadas 'herencia simple' y 'herencia múltiple' respectivamente). La herencia representa una jerarquía de abstracciones, en la cual una subclase hereda de una o mas superclases. Típicamente una subclase aumenta o re-define la estructura existente y el comportamiento de su superclase.

En el mundo real, un objeto puede tener comportamiento que son atribuidos a mas de un objeto anterior. Por ejemplo, sobre la base de los hábitos de comida, un animal puede ser clasificado como carnívoro, aunque otra manera de clasificación lo sitúe en una familia especifica, tal como la familia de osos. En el mundo de programación, un editor de texto full-screen tiene dos facetas prominentes: este puede almacenar un arreglo de texto el cual puede ser manipulado (insertar un caracter, borrar un caracter) y este puede mostrar un bloque de texto en la pantalla. Así, un editor de texto puede ser considerado como el heredero del comportamiento de una clase text buffer así como de una clase text screen (el cual maneja un área de pantalla de 80 caracteres por 25 lineas).

Este ejemplo ilustra el concepto de herencia múltiple de una clase que hereda de mas de una super clase. Muchos lenguajes de programación orientado a objetos, no permiten herencia múltiple.

2.2.5. TYPING

El concepto de **'type'** se deriva principalmente de la teoría de tipo abstracto de datos (ADT). Como Deutsch sugiere "un tipo es una caracterización precisa de la estructura o comportamiento de propiedades que toda una colección de entidades comparten". Si bien el concepto de tipo y clase son similares, se incluye el tipo como un elemento separado de el modelamiento de objetos debido a que el concepto de un tipo sitúa un énfasis muy diferente sobre el significado de la abstracción. Específicamente Grady Booch formula lo siguiente :

"**Typing** es la coacción de la clase de un objeto, tal que objetos de diferentes tipos no pueden ser intercambiados, o en su mayoría, ellos serían intercambiados sólo en maneras muy limitadas"

Typing nos permite expresar nuestras abstracciones de modo que el lenguaje de programación en el cual las implementamos pueda ser utilizado para reforzar las decisiones del diseño. Consideramos esto como un elemento menor, sin embargo, debido a que un lenguaje de programación dado puede que enfatize el uso de tipos (strongly-tped), que no lo enfatize (weakly-typed), o aún que no utilize tipos (untyped), estos pueden, sin embargo, ser llamados object-based u object-oriented.

Un lenguaje que enfatiza el uso de tipos es aquel en que todas las expresiones están garantizadas para ser type-consistent.

Hay un número de importantes beneficios que se deriva del uso de lenguajes que enfatiza el uso de tipos:

- Sin el chequeo del tipo (type checking), un programa en la mayoría de lenguajes pueden 'fallar' en forma misteriosa en tiempo de ejecución.
- En la mayoría de sistemas, el ciclo de editar-compilar-depurar es tan tedioso que la detección de error al principio es indispensable.
- La declaración de tipos ayuda a la documentación de programas.
- La mayoría de compiladores pueden generar más eficientemente código objeto si los tipos son declarados.

Lenguajes que no enfatiza el uso de tipos a menudo ofrecen flexibilidad, pero aún en estos lenguajes, como Borning e Ingalls observan, "En casi todos los casos, el programador de hecho debe conocer que clase de objetos son esperados como argumentos de un mensaje, y que clase de objetos serán retornados".

2.2.6. CONCURRENCIA

Para ciertas clases de problemas, un sistema automatizado tendría que manipular diferentes clase de objetos de forma simultánea. Otros problemas involucraría tantos cálculos que excedería la capacidad de un simple procesador. En cada uno de estos casos, es natural considerar el uso de un conjunto distribuido de computadoras para la implementación requerida, o el uso procesadores capaces de realizar multitareas (multitasking).

Un proceso simple - también conocido como el 'hilo del control' - es la raíz desde la cual acciones dinámicamente independiente ocurren dentro de un sistema. Cada programa tiene al menos un hilo de control, pero un sistema que involucra concurrencia podría tener muchos de estos hilos: algunos que son transitorios y otros que persisten durante toda la ejecución del sistema. Sistemas ejecutándose en múltiples CPU permiten la verdadera concurrencia de hilos de control, a diferencia que sistemas ejecutándose en un simple CPU sólo pueden alcanzar la ilusión de concurrencia de hilos de control, usualmente por medio de un algoritmo time-slicing.

Si bien la construcción de una gran pieza de software es bastante complicado; el diseñar una que abarque múltiples hilos de control es mucho mas complicado debido a que uno debe preocuparse acerca de los problemas de interbloqueo, starvation, exclusión mutua, y otras condiciones.

Considerando que la programación orientada a objetos es enfocada sobre la abstracción de datos, el encapsulamiento, y la herencia, la concurrencia es enfoca sobre los procesos de abstracción y sincronización. El objeto es un concepto que unifica estos dos puntos de vista diferentes: cada objeto (desde una abstracción del mundo real) podría representar un hilo de control separado (una abstracción de proceso). Tales objetos son denominados activos. En un sistema basado en un diseño orientado a objetos, podemos conceptualizar al mundo como un conjunto de objetos cooperativos, algunos de los cuales son activos y sirven así como el centro de actividades independientes. De acuerdo a esta concepción Booch define:

"La concurrencia es la propiedad que distingue un objeto activo de uno que no se encuentra activo".

PERSISTENCIA

Un objeto en software toma alguna cantidad de espacio y existe para una cantidad de tiempo particular. Atkison y otros sugieren que hay una existencia de objetos continuos, fluctuando desde objetos transitorios que aparecen dentro de la evaluación de una expresión, hasta objetos en una base de datos que sobreviven a la ejecución de un simple programa. Este espectrum de persistencia de objetos abarca lo siguiente:

- Resultados momentáneos en evaluación de expresiones
- Variables locales en la activación de procesos
- Variables propias, variables globales, e items cuya extensión es diferente a su alcance
- Datos que existen entre la ejecución de un programa
- Datos que existen entre varias versiones de un programa
- Datos que sobreviven a la ejecución de un programa

Los lenguajes de programación usualmente direccionan sólo las tres primeras clases de persistencia de objetos; la persistencia de las tres últimas clases es

típicamente dominio de la tecnología de la base de datos. Esto lleva a un encuentro (choque) de culturas que resultan algunas veces en diseños muy desconocidos : los programadores terminan empleando esquemas 'ad hoc' para almacenar objetos cuyo estado debe ser preservado entre las ejecuciones de programas, y el diseñador de la base de datos hace un mal uso de esta tecnología para hacer frente a los objetos transientes.

Unificando el concepto de concurrencia y objetos se da origen a lenguajes concurrentes para programación orientada a objetos. De una manera similar, introduciendo el concepto de persistencia al modelamiento de objetos se da origen a las base de datos orientadas a objetos. En la práctica, tales base de datos construida sobre tecnología probada, al mismo tiempo ofrecen al programador la abstracción de una interfase orientada a objetos, a través de la cual las consultas a las bases de datos y otras operaciones son complementadas en términos de objetos cuyo tiempo de vida trasciende el tiempo de vida de un programa individual. Esta unificación simplifica en grado sumo el desarrollo de ciertas aplicaciones. El particular, nos permite emplear el mismo método de diseño en una aplicación, tanto para los segmentos de la base de datos, así como para los que no corresponden a la base de datos, .

La persistencia no sólo trata con el tiempo de vida de los datos. En las bases de datos orientadas a objeto, no sólo es 'estado' de un objeto persiste, sino que su 'clase' debe también trascender a cualquier programa individual de modo que cada programa interprete este estado de la misma manera. Esto claramente trata con el desafío de la integridad de la base de datos a medida que esta crece, particularmente si debemos cambiar la clase de un objeto.

En la mayoría de sistemas una vez que un objeto es creado, consume la misma memoria física hasta que cesa de existir, sin embargo para sistemas que se ejecutan sobre un conjunto de procesadores distribuidos, algunas veces debemos estar interesados con la persistencia a través del espacio. En tales sistemas es útil pensar en objetos que pueden ser pasados de máquinas en máquinas, y aún que pudiesen tener diferentes representaciones en diferentes máquinas.

Para resumir, Booch nos da la siguiente definición:

"La persistencia es la propiedad de un objeto a través de la cual su existencia trasciende en el tiempo (es decir un objeto continua existiendo después de que su creador cesa de existir) y/o espacio (es decir que la localización de un objeto se mueve desde el espacio de dirección sobre el cual este fue creado)"

CAPITULO TRES

EL ADMINISTRADOR DE MEMORIA DE WINDOWS

Este capítulo contiene información acerca de como Microsoft Windows 3.1 administra e interactua con la memoria.

ACERCA DE LA MEMORIA.

La Memoria de Acceso Aleatoria (RAM) del computador es un medio volátil donde las aplicaciones y datos son almacenados mientras nos encontramos trabajando con ellos. Cuando finalizamos de trabajar en las aplicaciones, la información es transferida de vuelta a un almacenamiento permanente (disco duro o floppy disk).

Las aplicaciones Windows tales como Microsoft Word y Microsoft Excel tienen que ser cargadas en memoria antes de poder correr. Generalmente cuando Ud. corre Windows, cuanto más memoria del sistema este disponible, más aplicaciones podrán ser corridas simultáneamente, y más rápido se ejecutarán.

Esta sección describe los tipos de memoria en un PC y provee un resumen técnico de la Memoria Expandida. Otras secciones en este capítulo presenta resultados específicos para usar memoria en Windows standard mode y 386 enhanced mode.

TIPOS DE MEMORIA: UN RESUMEN.

Su sistema de computador puede tener tres clase diferentes de memoria: memoria convencional, memoria expandida, y memoria extendida. Windows también crea un cuarto tipo de memoria, la memoria virtual.

- La memoria Convencional consiste de los primeros 640k de memoria disponible en su máquina. La mayoría de PC's tienen al menos 256k de memoria convencional, su sistema debe tener al menos 640k de memoria convencional para correr Windows.

Cuando Ud. enciende su máquina, MS-DOS corre los utilitarios y aplicaciones listadas en los archivos CONFIG.SYS y AUTOEXEC.BAT Estos archivos a menudo usan memoria convencional para funcionar. La memoria restante está disponible para correr otras aplicaciones tales como Windows.

- La memoria extendida es esencialmente una extensión inconsútil más allá del un megabyte de espacio de direccionamiento original de memoria disponible en máquinas 286 y 386. La memoria extendida siempre inicia exactamente en los 1024k, donde el área de memoria superior (upper memory area UMA) termina. Los primeros 64k de la memoria extendida es referida como el área de memoria alta (high memory area HMA).

- La memoria expandida puede ser instalada como una carta de memoria expandida o, en una máquina 386, emulada por un manejador de memoria expandida (EMM). El software EMM mapea páginas de memoria expandida dentro del upper memory area del sistema (desde 640k y 1024k). Las aplicaciones deben de estar diseñadas para interactuar con el software EMM para tomar ventaja de la memoria expandida.

La memoria expandida es más lenta e incomoda de usar que la memoria extendida, porque el manejador de memoria extendida da acceso a las aplicaciones a sólo una limitada cantidad de memoria expandida a la vez.

Un procesador 80286 puede direccionar 16 megabytes de memoria total, y un procesador 80386 puede direccionar 4 gigabytes de memoria. Las máquinas 8086 y 8088 tienen limitaciones de hardware que excluye el uso de memoria extendida. Para tales máquinas, la memoria expandida es la única opción para memoria extra.

Trabajar con memoria convencional y extendida en Windows 3.1 es un proceso serio que rara vez demanda su atención, porque Windows maneja esto automáticamente a través de HIMEN.SYS. Si Ud. desea correr aplicaciones no-Windows que requieren memoria expandida, Ud. necesita un profundo entendimiento de como está se encuentra estructurada y como accesarla. Para más detalle ver "Memoria Expandida: una discusión técnica" posterior a esta sección.

La fig. 3.1 muestra la dirección relativa de la memoria convencional, el upper memory area (usada por la memoria expandida), y la memoria extendida.

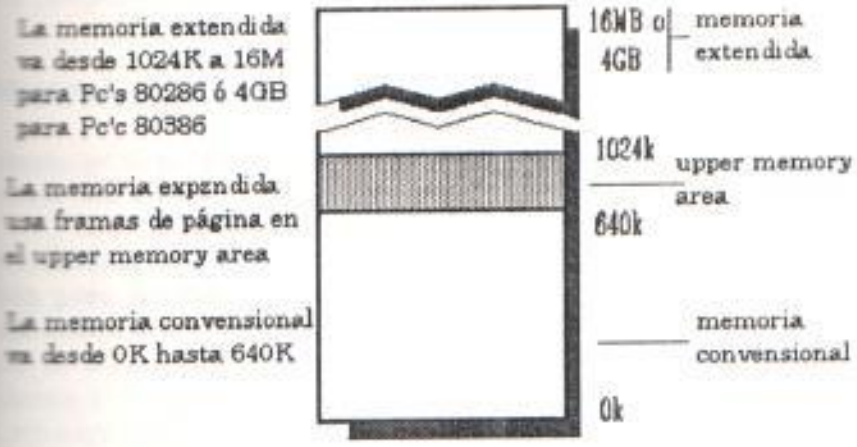

Figura 5-1. Direccionamiento relativo de memoria convensional

Para ver **la clase y cantidad de memoria en su sistema:**

- Digite **msd** en el command prompt para correr el utilitario Microsoft diagnostic, instalado con la versión 3.1 . Este utilitario prepara un extenso reporte de la memoria y drives instalado en su sistema.

- O si MS-DOS 5.0 está instalado en su sistema, digite **mem** en el command prompt para ver un breve resumen de su memoria del sistema.

Ud. puede usar parámetros con el comando **mem** del DOS 5.0 (mem /c y mem /p)

LOS DEVICE DRIVERS DE MEMORIA EN WINDOWS 3.1

Tanto Windows 3.1 como MS-DOS 5.0 provee los siguientes device drivers para administrar memoria:

- HIMEM.SYS provee acceso a la memoria extendida y HMA. La memoria extendida está provista de acuerdo a XMS 3.0.

- EMM386.EXE toma la XMS 3.0 provista por HIMEM.SYS y la usa para emular memoria expandida o para proveer UMB's (upper memory block's), o ambos. EMM386.EXE es un DPMI-servicial y emula memoria expandida de acuerdo a LIM 3.2 o LIM 4.0.

- RAMDRIVE.SYS crea un RAM disk.

- SMARTDRV.EXE es un utilitario de disk caching.

Además, Windows 3.1 tiene dos administradores de memoria internos.

- Swapdisk, el cual administra el proceso de disk swap para caching el código del aplicaciones Windows en los modo estándar y 386 enhanced.

- Virtual Memory Manager (VMM) el cual administra el intercambio a un swap file temporal o permanente en el modo 386 enhanced.

MEMORIA EXPANDIDA: UNA DISCUSION TECNICA.

El diseño original del IBM-PC, basado en el CPU del Intel 8086/8088, limita la memoria útil a 640k.

Una colaboración entre LOTUS/INTEL/MICROSOFT (LIM), desarrollaron una técnica para adicionar memoria a los sistemas PC. El LIM Expanded Memory Specification -EMS- (especificaciones de la memoria expandida) pasa por alto el límite de memoria soportado por las tarjetas de memoria (memory card) que contienen 16k paginas (o bancos) de RAM que están habilitadas o deshabilitadas por software, pero no pueden ser direccionadas por el CPU normalmente. En su lugar cada página es mapeada dentro del espacio de direcciones del procesador.

Las aplicaciones pueden usar memoria expandida para pasar por alto la limitaciones del procesador 8086/8088 usando un área especial del memoria denominada upper memory area.

Esta upper memory area esta localizada siempre en el mismo lugar del espacio de direcciones del computador: desde 640k hasta 1024k (A000 a FFFF hex). El upper memory area (algunas veces referida como "adapter segment" porque esta parte de memoria es usada por adaptadores del hardware tales como los display

adapters) esta también donde el ROM BIOS (segmento de memoria de sólo lectura) está localizado.

Una aplicación estará especificamente escrita para tomar ventaja de la memoria expandida. Muchas aplicaciones no-Windows usan memoria expandida:

- para ganar más perfomance efectiva de aplicaciones no-Windows grandes, tales como Hojas electrónicas y programas CAD.

- para correr programas residentes en memoria o aplicaciones que usan datos compartidos.

NOTA: el procesador 386 (y mayores) puede emular hardware EMS para usar memoria extendida con administradores de memoria y software especiales tal como el EMM386.EXE.

LAS ESPECIFICACIONES DE MEMORIA EXPANDIDA.

Las dos clases de memoria expandidas son diferenciadas una de otra por el número de versión del la especificaciones de memoria expandida LIM:

- LIM 3.2: esta especificación de memoria expandida mueve datos en bloques de 64k, cada uno compuesto de cuatro páginas contiguas de 16k, para formar una frama de página de 64k. La LIM 3.2 estándar trabaja bien para almacenar datos de hojas electrónicas en la memoria expandida, pero es insuficiente para multitasking (multitarea).

- LIM 4.0: esta especificación de memoria expandida permite a los datos ser almacenados en bloques de 1 a 16 páginas (superando las limitaciones en tamaño y flexibilidad del LIM 3.2). LIM 4.0 también quita la restricción de que las páginas deben de estar contiguas, abandonando esencialmente el requerimiento de frama de páginas.

BANCO CONMUTABLE EN MEMORIA EXPANDIDA.

Los Programas de aplicación toman ventaja de la memoria expandida haciendo llamadas al EMM para solicitar bloques de memoria expandida. Un Banco conmutable (de intercambio) es el proceso de mapear o temporalmente asignar memoria desde un reservorio de memoria expandida a un espacio vacio de direcciones en el upper memory area.

Por ejemplo, si una aplicación necesita más memoria para datos, esta contacta al EMM, el cual asigna memoria expandida de 16k a la vez. La aplicación escribe hasta 64k de datos a las direcciones asignadas por el EMM, entonces solicita otros 64k de asignación. Este requerimiento es el bank switch request. El EMM asigna otros 64k de memoria a diferentes direcciones mientras rastrea donde fueron situados los primeros 64k de datos.

El EMM continua esta actividad de intercambio de bancos, 16k a la vez, hasta que la aplicación no realiza más requerimientos de espacio de memoria. Con el banco de intercambio, el EMM puede manipular varios megabytes de datos a través de 64k de espacios individuales.

BACKFILLING EN MEMORIA EXPANDIDA

Con LIM 4.0 el administrador de memoria expandida (EMM) puede volver a llenar (backfilling) las direcciones de memoria convencional desde los 640k hacia abajo hasta aproximadamente los 256k, usando memoria que es generalmente reservada para MS-DOS, utilitarios y TSR's. Con el backfilling el sistema puede manipular grandes cantidades de datos, lo cual es importante para multitasking. Un reservorio alternante de backfilling memory permite a las aplicaciones completas y datos considerables sean cargados a memoria a la vez, aumentando la velocidad de operaciones.

Un artificio para maximizar el beneficio de backfilling en máquinas 286 es deshabilitar tanta memoria del motherboard como sea posible (bajo el nivel de 256k), y permitir que la tarjeta de memoria expandida proporcione esta memoria. Debido a que la tarjeta de memoria expandida esta proporcionando la memoria, esta puede intercambiar el banco libremente y hacer mejor uso de la memoria disponible.

Para máquinas 386, Ud. puede convertir la memoria expandida en memoria extendida, mediante el uso de un administrador de memoria extendida tal como el EMM386.EXE.

Si Ud. está usando una tarjeta de memoria expandida en su 386, lea su manual cuidadosamente antes de intentar un 'backfill' (vuelta a llenar). No todas las tarjetas de memoria tienen el registro de soporte para proporcionar más de cuatro páginas de 16k. También, debido a que Windows 3.1 soporta la memoria extendida directamente, volver a llenar la memoria expandida no le da a Ud. ventaja alguna para corren Windows.

DIFICULTADES CON LA MEMORIA EXPANDIDA

La memoria expandida usa el upper memory area, la cual esta compartida con adaptadores de hardware tales como tarjeta de video SuperVGA, tarjetas de red, tarjetas de emulación 3270, y controladores de disco ESDI. Varias dificultades potenciales pueden presentarse cuando la memoria expandida disputa los espacio de direcciones con los adaptadores:

- **Deficiencia de espacio libre**. El problema principal para el EMM es encontrar al menos 64k de espacio libre contiguos en el cual situar la frama de página. Aun cuando LIM 4.0 técnicamente no requiere la frama de página de 64k, la mayoría de aplicaciones de memoria expandida no usan la memoria expandida a memos que una frama de página de 64k este presente. Frecuentemente, las áreas de direcciones de varias tarjetas de adaptadores necesitan estar mezcladas para liberar suficiente espacio para una frama de página de 64k contigua. Este proceso trae complicaciones con boards tales como el emulador IBM 3270, el cual tiene una dirección fija en la mayoría de las máquinas.

- **Conflicto de mapeo**. La mayoría de administradores de memoria extendida tales como el EMM386 usa un algoritmo de búsqueda para encontrar memoria no usada entre las direcciones hexadecimales C000 hasta DFFF. El EMM entonces usa esta memoria como frama de página. Algunas tarjetas no reservan su espacio de direcciones hasta que la tarjeta es accesada, así el

EMM puede inadvertidamente mapear páginas de memoria expandida en el tope de la tarjeta, causando coaliciones y operaciones entrecortadas. Este problema es bastante raro porque la rutina de búsqueda de página puede localizar siempre todos los adaptadores populares.

Si ocurren problemas inicie deshabilitando la memoria expandida para ver si un conflicto de página esta causando el problema. Si el problema se mantiene, entonces el EMM necesita ser notificado para excluir al adaptador de direcciones de consideraciones como una localidad de página. El adaptador debe ser también ser movido, haga esto en diferentes maneras, con diferentes EMMs. Para información de como excluir un rango de direcciones, consulte la documentación de su administrador de memoria expandida.

WINDOWS STANDARD MODE Y LA MEMORIA

El modo estándar es el modo normal de operación para Windows en máquinas 80286, proveyendo acceso directo a memoria extendida.

La memoria convencional no toma consideraciones especiales para Windows en modo estándar. Windows corriendo en modo estándar trata al total libre de memoria convencional y memoria extendida como un bloque de memoria contigua. Esta sección describe como el modo estándar de Windows toma ventaja de la memoria extendida.

El modo estándar no usa la memoria expandida para operaciones Window, pero puede trabajar con memoria expandida para aplicaciones no-Windows corriendo bajo Windows.

MEMORIA EXTENDIDA Y WINDOWS STANDARD MODE

El modo estándar de Windows accesa la memoria extendida directamente a través de HIMEM.SYS (o a través de una tercera parte del XMS driver), proveyendo el total de memoria convencional y extendida par el uso de aplicaciones Windows. Cualquier aplicación Windows que use memoria extendida puede correr bajo modo estándar en Windows.

El modo estándar de Windows toma ventaja de memoria extendida por:

* Ejecutando código caching con aplicaciones Windows.

* Intercambiando (swapping) aplicaciones no-Windows a memoria extendida.

USANDO MEMORIA EXTENDIDA PARA CODIGO CACHING.

El modo estándar de Windows puede acelerar sus operaciones por **caching code** en la memoria extendida a través de HIMEM.SYS. Para ejecutar caching code, Windows toma ventaja de ciertos atributos de las aplicaciones Windows. El código para cada aplicación Windows esta dividida en segmentos con atributos específicos, incluyendo:

* El código de segmento movible, lo cual significa que el código puede ser movido alrededor de la memoria.

- El código de segmento descartable, lo cual significa que el código puede ser sobre escrito y entonces vuelto a cargar a disco cuando sea necesario

- El código de segmento intercambiable, el cual puede ser intercambiado a disco duro.

Cada aplicación Window mantiene una mínima cantidad de código (este es "swapsize") cargado en memoria. Si Windows corre fuera de memoria cuando una nueva aplicación es ejecutada, Windows descarta parte de una anterior aplicación de la memoria activa y sobreescribe ésta con el nuevo código del nuevo archivo ejecutable.

USANDO MEMORIA EXTENDIDA CON APLICACIONES NO-WINDOWS

Aplicaciones no-Windows que usa memoria extendida puede correr bajo Windows en modo estándar. La cantidad de memoria extendida que la aplicación requiere estaría especificada en el archivo de información del programa (PIF) de la aplicación.

Por ejemplo, su sistema podría tener 2048k de memoria extendida, y Ud. podría especificar en el PIF que la aplicación no-Windows requiere 1024k de memoria extendida. Si el modo estándar de Windows esta ya usando toda la memoria extendida cuando Ud. corre la aplicación no-Windows, entonces la información en el primer megabyte de la memoria extendida es intercambiada a disco, y la aplicación no-Windows accesa a la memoria extendida nuevamente libre. Cuando Ud. intercambia otra vez a Windows de la aplicación no-Windows, los datos originalmente almacenados en los 1024k son vueltos a cargar desde el disco.

Debido a que este proceso de intercambio de archivo puede ser lento, no solicite más memoria extendida en el archivo PIF que la absolutamente necesaria para correr la aplicación no-Windows.

Algunas aplicaciones no-Windows usan MS-DOS extender technology tales como VCPI O DPMI para correr en modo protegido. Si Ud. esta corriendo tales aplicaciones bajo modo estándar en Windows, Ud. puede asignar memoria extendida en el PIF de la aplicación.

3.2.2. MEMORIA EXPANDIDA Y WINDOWS STANDARD MODE

Windows corriendo en modo estándar no usa toda la memoria expandida para sus operaciones. Pero aplicaciones no-Windows corriendo bajo modo estándar en Windows puede accesar memoria expandida si el sistema tiene una tarjeta EMS tal como AST RAMPage! o el Intel Above Board.

El administrador de memoria expandida para la tarjeta EMS usa upper memory block's (UMB's) en el upper memory area. Si Ud. sospecha que un conflicto de UMB esta causando un problema en su sistema, saque el administrador de memoria expandida para ver si esto soluciona su problema.

Debido a que las aplicaciones no-Windows corriendo bajo modo estándar puede solo usar memoria expandida con una tarjeta física EMS, un manejador de

memoria extendida externo como el EMM386.EXE no puede proveer el soporte requerido de memoria expandida en Windows. Sin embargo, un administrador externo 386 puede proveer soporte de memoria expandida para aplicaciones no-Windows, cuando Ud. no esta corriendo en Windows.

Si Ud. quiere usar memoria en el upper memory area, o si Ud. tiene una aplicación que requiere memoria expandida pero no tiene una tarjeta física EMS, Ud. debe correr Windows en modo 386 extendido.

WINDOWS 386 ENHANCED MODE Y LA MEMORIA

Windows 386 enhanced mode es el modo normal de operación para sistemas con 80386 y procesadores mayores.

Cuando Windows corre en 386 enhanced mode, este suma la memoria convencional y extendida y la trata la cantidad total como un bloque disponible de memoria, casi de la misma manera como el modo estándar. Las entradas en la sección **[386enh]** del SYSTEM.INI que controla como Windows asigna la memoria convencional son:

PerformBackfill=,
ReservePageFrame=,
WindowsKBRequired=,
y **WindowMemSize=.**

Si bien 386 enhanced mode no usa memoria expandida para operaciones Windows, este puede simular memoria expandida para uso de aplicaciones no-Windows como se describe en esta sección.

Ud. puede crear también un swap file (archivo de intercambio) tal que Windows 386 enhanced mode pueda tomar ventaja de la capacidades de memoria virtual del procesador 80386 o superiores.

WINA20.386 Y 386 ENHANCED MODE

El programa Setup de MS-DOS 5.0 instala un device drivervirtual que resuelve conflictos entre Windows 3.0 y MS-DOS 5.0 cuando ambos intentan accesar el HMA. Este driver es un archivo de solo lectura denominado WIN20.386, el cual es automáticamente instalado en el directorio raíz. Windows 3.0 no puede correr en 386 enhanced mode sin este archivo.

- Si Ud. actualiza a Windows 3.1, o si Ud. nunca corre 386 enhanced mode con Windows 3.0, Ud. puede remover el archivo WINA20.386 cambiando su atributo de solo lectura, entonces bórrelo de la manera usual

- Si Ud. está corriendo Windows 3.0 y mueve el archivo WINA20.386 a un diferente directorio, Ud. debe asegurarse que Windows pueda encontrar este archivo adicionando un comando **switches=/w** al CONFIG.SYS y también adicionando una entrada **device=**[new path]**\wina20.386** a la sección **[386enh]** del SYSTEM.INI.

2.2. MEMORIA EXTENDIDA Y 386 ENHANCED MODE

Windows 386 enhanced mode usa HIMEM.SYS, el device driver de memoria extendida, para cargarse a si mismo y sus driver dentro de la memoria extendida. Como en el modo estándar, Windows 386 enhanced mode provee el total de memoria libre (convencional y extendida) para el uso de directo de aplicaciones Windows. Por consiguiente code caching para aplicaciones Windows pueden también ser ejecutadas en 386 enhanced mode.

Windows 386 enhanced mode también permite a las aplicaciones no-Windows correr en modo protegido si la aplicación usa las especificaciones de DOS Proteted Mode Interface (DPMI). Lotus 1-2-3 versión 3.1 es un ejemplo de una aplicación DPMI.

Este modo de operación provee acceso a memoria extendida para las aplicaciones no-Windows creando máquinas virtuales hasta 640k en tamaño, o el tamaño definido por la entrada **ComandEnvSize=** en la sección **[NonWindowsApp]** del SYSTEM.INI.

Cada máquina virtual hereda el ambiente presentado antes que Ud. inicie la sesión Windows. Esto significa que cada driver y los programas terminate-and-stay-residend (TSR) cargados antes de correr windows consumen memoria en cada subsecuente máquina virtual. La memoria disponible dentro de cada máquina virtual bajo el 386 enhanced mode es ligeramente menos lenta que la memoria libre disponible en el command prompt antes de que Ud. inicie Windows, dependiendo de la configuración de su sistema.

Cuando creamos máquinas virtuales para aplicaciones no-Windows, Windows 386 enhanced mode usa el upper memory area para dos propósitos:

- Para asignar buffers de traslación para llamadas API modo-protegido para MS-DOS y la red
- Para situar la frama de página para memoria expandida (si es requerida)

Frecuentemente todas las páginas libres en el upper memory area son usadas por el 386 enhanced mode

2.3. MEMORIA EXPANDIDA Y 386 ENHANCED MODE

Windows 386 enhanced mode no usa memoria expandida por si mismo, y las aplicaciones Windows no necesitan memoria expandida, porque ellas corren en modo protegido y pueden accesar la memoria extendida directamente. Sin embargo, Windows 386 enhanced mode puede crear memoria expandida para el uso de aplicaciones no-Windows tales como Lotus 1-2-3 que requiere o puede tomar ventaja de la memoria expandida.

Windows 386 enhanced mode provee automáticamente memoria expandida para aplicaciones no-Windows que requieren de esta cuando Ud. corre estas aplicaciones bajo Windows. Este modo no puede proveer este tipo de memoria si Ud. carga el EMM386.EXE con el switch **noems**. Usé el **ram** switch cuando cargue EMM386.EXE en el CONFIG.SYS, o utilice el parámetro **x=mmmmm-**

nnnn para asignar suficiente espacio en el upper memory area para Windows para crear una frama de página EMM.

CONFLICTOS DE FRAMA DE PAGINA EN EL 386 ENHANCED MODE

Windows 386 enhanced mode provee framas de páginas adicionales para memoria expandida LIM 4.0 en todas las máquinas virtuales. Pero la mayoría de aplicaciones no-Windows usan solo la misma frama de página de 64k, no la páginas adicionales de memoria convencional que LIM 4.0 proporciona. Así para usar memoria expandida para correr aplicaciones no-Windows, Ud. necesita tener una frama de página de 64k contigua, hechas de hasta cuatro páginas de 16k contiguas en el upper memory area. El problema clave, por lo tanto, para memoria expandida bajo Windows es conflicto de frama de página.

Los adaptadores instalados en un sistema pueden esparcir el área libre en el upper memory area de tal forma de no encontrar un área de 64k de memoria contigua para situar la frama de página, y por consiguiente no existir memoria expandida libre para correr aplicaciones no-Windows. Si este problema ocurre, Ud. tendrá que re-ordenar las localizaciones de adaptadores de memoria.

Esto es sumamente fácil de hacer en una máquina micro canal como el IBM Personal System/2, el cual permite cambiar las localizaciones por booteo de los adaptadores de memoria con el PS/2 reference Disk y seleccionando Change Configuration. Un procedimiento similar esta disponible en la mayoría de las barra de máquina (Bus machine) de Extended Industry Standard Architecture (EISA).

Para las barra de máquina de Industry Standard Architecture (ISA), tales como el IBM AT y Compaq 386, Ud. tendrá que abrir el case y variar los DIP switches en la tarjeta para cambiar las direcciones de memoria. Refiérase a su manual de hardware.

Usted también puede deshabilitar la memoria expandida totalmente (y el soporte de frama de página de 64k) en el Windows 386 enhanced mode estableciendo **NoEMMDriver=yes** en la sección **[386enh]** del SYSTEM.INI.

Otras entradas relacionadas en la sección **[386enh]** del SYSTEM.INI son:
EMMSize=, y
IgnoreInstalledEMM=.

SITUANDO BUFFERS DE TRASLACION EN EL UPPER MEMORY AREA

En 386 enhanced mode, Windows asigna buffers en el upper memory area para trasladar las llamadas API de MS-DOS y red desde el modo protegido de Windows a modo real de MS-DOS (debido a que el upper memory area esta dentro del primer megabyte del espacio de direccionamiento, este puede ser accesado por MS-DOS en modo real en un procesador 386 o superior).

Idealmente habrá suficiente espacio libre en el upper memory area para situar tanto los buffers de traslación y cualquier frama de página de memoria expandida requerida. Pero en la mayoría de sistemas no hay suficiente espacio, y Ud. debe elegir entre eliminar la frama de página de memoria expandida o

signar el buffer de traslación en memoria convencional (en lugar del upper memory area)

Si los buffer de traslación están asignados en memoria convencional, ellos tomarán memoria en cada máquina virtual que Windows crea, dejando menos espacio en la máquina virtual para correr las aplicaciones no-Windows. Para arreglar este problema, los buffer de traslación pueden ser asignados en el upper memory area o en la memoria convencional, pero nunca mitad y mitad.

Para especificar una preferencia, fije el valor en la sección **[386enh]** para la entrada **ReservePageFrame=**. Si **ReservePageFrame=true** (por defecto), entonces Windows asigna la frama de página primero y los buffer de traslación segundo. Usualmente en máquinas con UMB's, los buffers de traslación son forzados a situarse dentro de memoria convencional, pero le permite usar memoria expandida para aplicaciones no-Windows.

Si **ReservePageFrame=false**, el buffer de traslación son asignados en el UMB's primero, seguido por la frama de página si hay espacio aún. Esta asignación le da más memoria libre en las máquinas virtuales, pero podria no tener suficiente espacio para las aplicaciones no-Windows.

CONTROLANDO EL MAPEO DE UMB EN 386 ENHANCED MODE

Usted puede controlar la colocación de las framas de páginas de memoria expandida y el mapeo de los buffers de traslación con las entradas **EMMExclude=** o **ReserveHighArea=** en la sección **[386enh]** del SYSTEM.INI. Para excluir explícitamente un área del upper memory area del mapeo de Windows 386 enhanced mode, establezca un valor para **EMMExclude=**, por ejemplo **EMMExclude=E000-EFFF**.

Debido a que no hay un estándar para las implementaciones de hardware para el área E000-EFFF, frecuentemente es necesario excluir este rango, de tal forma que 386 enhanced mode pueda funcionar adecuadamente. Windows 386 enhanced mode detecta y excluye el área de la mayoría de tarjetas adaptadoras automáticamente.

Cualquier valor fijado con el switch **x=** en la linea que carga EMM386.EXE en el CONFIG.SYS anulará el valor fijado por **EMMExclude** en el SYSTEM.INI. La entrada **ReserveHighArea=** provee el mismo soporte, pero para un rango de de 8k en lugar de los 16k.

Otras entradas relacionadas al mapeo de UMB en la sección **[386enh]** son:

EMMInclude=,
EMMPageFrame=,
EMMSize, y
UseableHighArea=.

Debido a que Windows usará todas las páginas libres en el upper memory area automáticamente, hay poco uso para **EMMInclude=**, **EMMPageFrame=**, y **UseableHighArea=**.

Windows/386 2.x no utiliza el área E000-EFFF del adaptador de segmento a menos que sea específicamente instruido para aquello. Windows 386 enhanced mode usa este segmento a no ser que la máquina se identifique a si misma como un PS/2. Como una nota suplementaria, la mayoria de las entradas en la sección **[386enh]** del SYSTEM.INI que inician con las letras "EMM" controla la ubicación de la frama de página de memoria expandida y el mapeo de los buffers de traslación.

Las letras "EMM" son utilizadas solo para mantener compatibilidad; excepto para la entrada **EMMPageFrame=**, la cual ya no se aplica solo a la frama de página de memoria expandida. El parámetro **LastEMMSeg=** usado en Windows 2.x ha sido eliminado para Windows 3.x.

ACERCA DEL 386 EXPANDED MEMORY MANAGER

Un administrador de memoria expandida tal como el EMM386.EXE puede proveer memoria expandida para la aplicaciones no-Windows en máquinas 386 y superiores sin una tarjeta fisica EMS cuando Ud. no esta corriendo windows.

Algunos administradores de memoria expandida tales como el EMM386.EXE y CEMM.EXE ser interrumpidos cuando Windows esta corriendo. CEMM.EXE requiere que la memoria expandida no este siendo usada cuando inicie Windows (esto es **NoEMMDriver=yes** en la sección **[386enh]** del SYSTEM.INI).

Tanto Windows 3.1 como MS-DOS 5.0 soportan los mecanismos definidos por LIM 3.2 y LIM 4.0, de tal forma que los administradores de memoria expandida tales como el EMM386.EXE, 386MAX.SYS, y QEMM.SYS puedan cargar network driver y otros dispositivos de software el upper memory area y se mantengan corriendo con Windows 386 enhanced mode.

MEMORIA VIRTUAL Y 386 ENHANCED MODE

La memoria virtual ha sido usada por años en mainframes, pero llega primero a los PC's con la introducción del sistema operativo IBM/MICROSOFT OS/2. Windows 386 enhanced mode va más alla de OS/2 para ofrecer memoria virtual usando las capacidades especiales de demanda de página del procesador Intel 80386.

Cuando la memoria virtual es usada con Windows 386 enhanced mode, algunos de los códigos de programas y datos son mantenidos en memoria fisica mientras el resto es intercambiado (swapping) al disco duro en un swap file.

Cada vez que una referencia es hecha a una dirección de memoria, esta puede ser usada sin interrupciones si la información está actualmente en memoria fisica. Si la información no se encuentra en memoria fisica, una falla de página ocurrirá y el administrador de memoria virtual de Windows (VMM) toma el control, trayendo la información de vuelta a memoria fisica, y si es necesario, intercambiando otra información a disco. Toda esta actividad es transparente al usuario, quien solo ve alguna actividad en el disco duro.

Las aplicaciones Windows pueden usar la memória virtual sin estar especialmente escrita para tomar ventaja de esta, debido a que Windows

manipula al administrador de memoria, asignando sin embargo mucha memoria a la aplicación requerida. Con Windows administrando la memoria virtual, Ud. vera mucha más memoria disponible que la instalada en su máquina cuando selecciona About Program Manager o About File Manager de el menú del help.

El mayor beneficio de usar memoria virtual es el que Ud. puede correr más programas simultáneamente del que la memoria física del sistema permitiría. Los drawback son los espacios de disco requeridos para el archivo de intercambio (swap file) de memoria virtual y la disminución de la velocidad de ejecución cuando un intercambio de página es requerida. Sin embargo, generalmente es mejor correr un programa lentamente en memoria virtual que no poder correrlo.

CREANDO UN SWAP FILE PARA MEMORIA VIRTUAL

En Windows 3.1 Ud. puede crear un swap file para memoria virtual durante el Setup, o puede seleccionar el icono 386 enhanced en el Control Panel para cambiar el swap file en cualquier momento.

Usted puede crear un swap file temporal o permanente. Un swap file permanente mejora la velocidad del sistema de memoria virtual Windows debido a que el archivo se encuentra en localidades contiguas, de tal manera que al acceso requiere menor overhead que el archivo creado por MS-DOS para un swap file temporal.

- Un swap file temporal denominado WIN386.SWP es creado en el disco duro mientras Windows esta corriendo, y lo borra automáticamente cuando salimos del Windows. Este swap file no es un archivo oculto o del sistema, y puede ser reducido o ampliado de tamaño como sea necesario. La entrada para **PagingFile=** en la sección **[386enh]** del SYSTEM.INI define el nombre del archivo y el path para el swap file temporal. Se necesita alrededor de 1.5MB de espacio libre en el disco duro en el 'paging drive' para un swap file temporal.

- Un swap file permanente es un archivo oculto denominado 386PART.PAR, el cual tiene un atributo del sistema y es siempre creado en el directorio raíz del drive especificado. Windows también crea un archivo de solo lectura SPART.PAR en el directorio Windows para decir a Windows donde y cuan grande es el swap file permanente. Debido a que el swap file debe ser contiguo, no puede crear un swap file permanente mayor al segmento contiguo más grande del disco duro. No es posible crear un swap file permanente si Staker esta corriendo en el sistema.

Ud. puede especificar el tipo y el tamaño de un swap file y el drive donde será localizado en el Virtual Memory dialog box. Un swap file es siempre creado en el directorio raíz. Pero puede especificar un subdirectorio como la localidad de un archivo temporal con **PagingFile=** en la sección **[386enh]** del SYSTEM.INI.

Cuando Ud. instala Windows, Setup chequea si su controladora de disco duro es compatible con acceso a disco de 32-bit. Si es así, el 32-Bit Disk Access check box aparece. Selecciones este check box si su sistema tiene solo una pequeña cantidad de memoria libre y Ud. desea incrementar la perfomance para el MS-DOS prompt. Cuando esta opción es fijada, Ud. puede correr más instancias de MS-DOS prompt e intercambiar entre ellas rápidamente. Si Ud. tiene múltiples

instancias de MS-DOS prompt corriendo y las aplicaciones accesando todos los disk drives, el tiempo de acceso es mejor con 32-bit disk access.

El utilitario de memoria virtual que crea swap file soporta solo discos duros que usan sectores de 512-bytes. Si los sectores de 512-bytes no están siendo usados, esto indica una configuración noestandar, tal como un third-party driver. Nunca cree un swap file permanente en un drive que use un driver particionado, con la excepción del utilitario Compaq ENHDISK.SYS.

Antes que Ud. pueda crear un swap file permanente, debe compactar su disco duro con un utilitario disk compacting. Si un mensaje de error reporta que su swap file está dañado, borre el actual swap file y cree uno nuevo.

Las entradas relacionadas para swap files en la sección **[386enh]** del SYSTEM.INI son:

MaxPagingFileSize=,
MinUserDiskSpace=,
PagingDrive=, y
PagingFile.

Nota: no intente crear un swap file en RAM disk. Un swap file creado en un RAM drive es self-defeating, debido a que sacrifica memoria física por memoria virtual.

DEMANDA DE PAGINA Y MANEJO DE MEMORIA VIRTUAL

Windows 386 enhanced mode administra la memoria virtual como un sistema de demanda de página que usa un administrador de memoria virtual (VMM) y un dispositivo de página de intercambio (el cual esta construido dentro del archivo WIN386.EXE). Esto significa que las páginas de datos son llevadas a memoria física cuando son referenciadas, y el sistema no intenta predecir cual página será requerida en el futuro.

El Windows VMM mantiene la tabla de página de memoria virtual que lista las páginas actualmente en memoria física y aquellas intercambiadas a disco. Debido a que 386 enhanced mode es un ambiente multitasking, la tabla de página VMM también lista cual página de memoria pertenece a cual proceso. Cuando VMM necesita una página que no está actualmente en memoria física, este llama a un dispositivo de intercambio de páginas, el cual asigna memoria virtual y mapea páginas dentro y fuera de la memoria física.

Algunos sistemas de memoria virtual depende del programa de segmentación para realizar su trabajo. Si bien el código para las aplicaciones Windows esta segmentada como se describió anteriormente, el administrador de memoria virtual Windows no está relacionada para esta segmentación. Toda la memoria virtual y física está dividida en páginas de 4k. La memoria asignada a una aplicación está hecha de memoria virtual, y en cualquier momento una página puede estar en memoria física o intercambiada al disco duro.

Las entradas en la sección **[386enh]** del SYSTEM.INI que controla la paginación de memoria virtual son:

LocalLoadHigh=	**Paging=**	**SysVMEMSLimit=**
MaxBPs=	**PagingDrive=**	**SysVMEMSLocked=**

[illegible]Page=	PagingFile=	SysVMV86Locked=
[illegible]LockMem=	PageOverCommit=	SysVMXMSLimit=

[illegible]ALGORITMO LRU PARA ADMINISTRADORES DE MEMORIA VIRTUAL

[illegible]intercambiador de páginas Windows VMM utiliza el algoritmo de reemplazo de página LRU (Least Recently Used). Esto significa que las páginas que no han sido accesadas por un largo periodo de tiempo son las primeras en ser intercambiadas a disco.

La tabla de página VMM contiene banderas para los atributos "accesado" y "sucio" de cada página. "Accesado" significa que un proceso tiene una referencia a la página desde que fue originalmente cargada. "Sucio" significa que una 'escritura' ha sido realizada a la página desde que está fue cargada. Debido a que 'escritura' califica como un acceso, el atributo "sucio" implica el atributo accesado.

Si espacio de memoria física no puede ser encontrado cuando un proceso requiere memoria adicional, Windows usa el algoritmo LRU para decidir que páginas intercambiar al disco duro para cumplir con el requerimiento. Esta decisión es un proceso de tres pasos:

1. Windows busca la tabla de página VMM para ver las páginas sin los atributos "accesado" o "sucio". Durante el proceso de búsqueda, windows limpia el atributo "accesado de todas las páginas.

2. Si Windows encuentra suficientes páginas que cumplen con el requerimiento No accesado / No sucio, éste intercambia estas páginas al disco duro y da la memoria libre resultante al proceso.

3. Si Windows no puede encontrar suficientes páginas inicialmente, repite la búsqueda. Teóricamente más páginas cumplirán el requerimiento debido a que el atributo accesado fue limpiado en la primera búsqueda. Si la segunda búsqueda no encuentra las páginas requeridas, entonces Windows intercambia páginas a el disco duro indiferente de sus atributos.

Las entradas relacionadas en la sección del SYSTEM.INI son:

[illegible]LowRateMult=,
[illegible]RateChngTime=,
[illegible]SweepFreq=,
[illegible]SweepLen=,
[illegible]SweepLowWater=, y
[illegible]SweepReset=.

[illegible]TERCAMBIO DE PAGINAS A UN DRIVE DE RED

Se recomienda que no intercambie páginas a un drive de red. La paginación a un drive de red es posible, pero es extremadamente lento. Si Ud. página a un drive de red, use un swap file permanente. También el directorio no debe tener un atributo de MS-DOS, y Ud. tendrá que crear y escribir acceso al directorio. No fije el valor para **PagingDrive=** o **PagingFile=** en el SYSTEM.INI a un Novell

Network drive, debido a que Novell network no es compatible con el MS-Net Redirector.

OTROS ADMINISTRADORES DE MEMORIA

En esta sección se provee una breve información sobre varios administradores de memoria:

* Especificaciones sobre DPMI y VCPI

* MS-DOS y Windows 3.1

* Requerimientos de memoria y puesta en marcha

* Recursos del sistema y memoria

ESPECIFICACIONES SOBRE DPMI Y VCPI

La Interface del Modo Protegido del DOS DPMI (DOS Protected Mode Interface) fue desarrollado por un grupo de industrias líderes. Varios miembros del comité DPMI también ayudaron a crear la Interface de Control de Programas Virtuales VCPI (Virtual Control Program Interface). DPMI es inicialmente una creación de Microsoft, y VCPI fué inicialmente propuesta por Phar Lab Systems. DPMI y VCPI resuelven dos problemas diferentes.

Las aplicaciones que usa MS-DOS Extender puede ejecutar código en el modo protegido del procesador 80286 o 80386. DPMI provee el método estándar para que tales aplicaciones switchee al procesador 80268 a modo protegido y para asignar memoria extendida. Cientos de aplicaciones usan varios tipos de MS-DOS Extender, y aquellas que ya no soportan DPMI requieren pequeños cambios para hacerlo.

VCPI provee una interface que permite que las aplicaciones usando MS-DOS Extender en máquinas 386 corran simultáneamente con administradores de memoria expandida 386. Por ejemplo QEMM.EXE 386MAX.EXE, y CEMM.EXE soportan las especificaciones VCPI. Windows 3.1 soporta tanto en modo estándar como en modo extendido. Windows 3.0 no soporta VCPI.

MS-DOS 5.0 Y WINDOWS 3.1

Muchos de los cambios en MS-DOS 5.0 hace a esta una plataforma más robusta para Microsoft Windows, ampliando sus capacidades de hacer un mejor uso de sistema de memoria. Para una mejor perfomance de Windows 3.1 (y otras aplicaciones), se sugiere que actualice su sistema a MS-DOS 5.0 si aún no lo ha hecho.

En PC's 80386 y 80486, con MS-DOS 5.0 Ud. puede cargar programas en memoria residente tales como device driver, TSR's, y software de red el upper memory area, por lo tanto liberando memoria convencional, par esto debe cargar HIMEM.SYS y EMM386.EXE, y entonces cargar los programas residentes de

memoria, utilizando el comando **devicehigh** en el CONFIG.SYS y el comando **loadhigh** en el AUTOEXEC.BAT.

CORRIENDO WINDOWS STANDARD MODE CON MS-DOS 5.0

- Utilice Windows Task List para intercambiar aplicaciones (o presione ALT+TAB). Usar el MS-DOS 5.0 Task Swapper mientras corre Windows es redundante, incurriendo a un overhead de memoria convencional innecesario.

- Cargue MS-DOS 5.0 en HMA, indistintamente de si su PC es un 80286, 80386, o 80486. Esto le proveerá de más memoria convencional para aplicaciones no-Windows corriendo bajo el modo estándar de Windows.

CORRIENDO WINDOWS 386 ENHANCED MODE CON MS-DOS 5.0

- Cargue MS-DOS 5.0 en HMA, y cargue todo la que entre en el upper memory area. La memoria convencional liberada al cargar MS-DOS en el HMA, es también liberada en cada máquina virtual, dándole más memoria a cada aplicación no-Windows que Ud. corra en Windows 386 enhanced mode.

- Use el Task List en Windows 3.1 para intercambiar entre aplicaciones (o presione ALT+TAB). Windows 386 enhanced mode también permite multitasking, de tal manera que múltiples aplicaciones puedan correr en el fondo. El MS-DOS 5.0 Task Swapper puede solamente intercambiar tareas (task-switch), lo cual significa que la aplicación intercambiada es suspendida.

USO MEMORIA Y LOS REQUERIMIENTOS DEL WINDOWS STARTUP

Windows automáticamente inicia en el modo de operación apropiativo de Windows, dependiendo de la configuración de su sistema. Sin embargo, Ud. puede iniciar Windows en un modo de operación particular usando uno de los siguientes switches de comando de línea:

- win /s para correr en el Standard mode

- win /3 para correr en el 386 enhanced mode

REQUERIMIENTOS DE PUESTA EN MARCHA EN EL STANDARD MODE

Para que Windows automáticamente inicie en el modo estándar, el sistema deberá tener:

- Procesador 80286 o superiores

- 256K de memoria convencional libre

- 192K de memoria extendida libre

- Un XMS driver tal como el HIMEM.SYS ya cargado

Los requerimientos de memoria convencional y extendida son mutuamente dependientes para el modo estándar y no son fijos.

REQUERIMIENTOS DE PUESTA EN MARCHA EN EL 386 ENHANCED MODE

Para que Windows inicie en el modo 386 extendido, el sistema deberá tener:

- Procesador 80386 o superiores
- 256K de memoria convencional libre
- 1024K de memoria extendida libre
- Un XMS driver tal como el HIMEM.SYS ya cargado

Windows 386 enhanced mode requiere una combinación de memoria convencional y extendida entre 580K y 624K para correr. Una instalación típica requiere un mínimo de 192K de memoria convencional libre en el command prompt más la suficiente memoria extendida para correr en 386 enhanced mode. Windows puede iniciar bajo 386 enhanced mode en memoria baja, debido a que este modo provee soporte de memoria virtual, pero puede ser extremadamente lento debido a que un swapping extra a disco debe efectuarse.

Todos estos números son aproximados y pueden variar ampliamente dependiendo de el Windows device driver presente, la versión MS-DOS, el display adapter, y otros factores.

Por ejemplo en las máquinas Compaq 386, 128K son recuperados de la shadow ROM. Los requerimientos de memoria toma en cuenta la memoria que puede ser liberada del SMARTDRIVE, bajando al mínimo el tamaño del cache especificado.

Windows chequea para un mínimo de 1MB de memoria extendida libre antes de iniciar automáticamente en el 386 enhanced mode. Si encuentra menos, trata de correr en standard mode. En un 386 con sistema de memoria de 2MB o menos, si Windows no encuentra suficiente espacio de memoria extendida libre, correrá en standard mode. Para liberar más memoria extendida de tal forma que pueda correr en 386 enhanced mode, debe intentar reducir de memoria extendida que SMARTDRIVE usa estableciendo su parámetro MinCacheSize a 0.

La entrada en la sección **[386enh]** en el SYSTEM.INI que controla como Windows asigna memoria convencional para arrancar y para 386 enhanced mode son:

SysVMEMSRequired=,
SysVMXMSRequired=,
WindowsKBRequired=, y
WindowsMemSize=.

LA MEMORIA Y LOS RECURSOS DEL SISTEMA DE WINDOWS

El Help About dialog box en el Program Manager y el File Manager (y en cualquier otra aplicación Windows) Muestra el porcentaje y la cantidad de memoria libre. El porcentaje de recursos del sistema refleja la memoria usada por el núcleo de la estructura interna de Windows.

Los tres archivos fundamentales que integran la parte de Windows para correr aplicaciones son:

- El archivo núcleo (KRNL286.EXE o KRNL386.EXE) carga y ejecuta las aplicaciones Windows y manipula su administración de memoria.
- GDI.EXE administra gráficos e impresoras.
- USER.EXE controla las entradas y salidas de usuario, incluyendo el teclado, mouse, driver de sonido, timer, puertos de comunicación, y administrador Windows.

En Windows 3.x estos archivos se encuentran en el subdirectorio SYSTEM (en Windows 2.x estos módulos fueron enlazados en los archivos WIN200.BIN y WIN200.OVL)

Tanto GDI como User tienen áreas de almacenamiento denominadas heap, cada una con un espacio de almacenamiento de 64K. Los recursos de sistema disponibles reflejan el porcentaje remanente libre después de combinar el GDI local heap y el user y menú heap en User. Esto incrementa a 64K la cantidad de heap space que se encuentra disponible en Windows 3.0. En Windows 3.1, los items del Program Manager son tratados de forma separada y no usa el User heap space. Para ver la cantidad de recursos que una aplicación particular usa, observe la cantidad de recursos del sistema disponible antes y después que la aplicación corra (seleccione About Program Manager del menú del Help y compare la cantidad de recursos listados en el dialog box).

Si bien Windows 3.1 le permite correr varias aplicaciones simultáneamente que cualquier otra versión anterior, podría llegar a tener un mensaje de fuera de memoria que indica que su sistema está escaso de recursos. Esto se debe a que cada ventana y subventana creada requiere de User y GDI local heap space. Los recursos del sistema pueden agotarse si muchos objetos son creados por las aplicaciones Windows.

Un elemento importante del administrador de memoria para las aplicaciones Windows que no está incluido en el porcentaje de recursos del sistema es el número de selectores. Un selector es un puntero de memoria que es consumido con cada asignación de memoria realizadas por las aplicaciones Windows. Si una aplicación asigna muchos objetos de datos, es posible agotar los selectores, resultando en un mensaje de fuera de memoria (out-of-memory).

La eficiencia con la cual las aplicaciones Windows manipulan sus objetos de datos pueden ayudar en esta situación. Si Ud. experimenta un problema crónico con una de sus aplicaciones, mientras unas pocas están cargadas, contacte con su vendedor de aplicaciones para que la compañía se entere del problema y lo solucione.

SMARTDRIVE 4.0 : UNA DISCUSION TECNICA

Esta sección describe el nuevo MS-DOS disk cache, SMARTDrive 4.0, el cual reemplaza la versión de SMARTdrive 3.x. Esta sección está dirigida a usuarios técnicos que deseen profundizar en el entendimiento del utilitario.

3.1. ACERCA DE SMARTDRIVE 4.0

SMARTDrive 3.x es un rastreador de cache de solo lectura que mantiene en cache solo las rutas básicas y las operaciones de lectura. La estructura interna de datos son enlazadas a la geometria logica del disco. Mantiene cache en el nivel de la interrupción 13 del ROM BIOS y usa el BIOS especificado en la geometría del disco para decidir el tamaño de su caching element. Esto conduce a problemas, de forma tal que Microsoft elige implementar un nuevo cache en SMARTDrive 4.0.

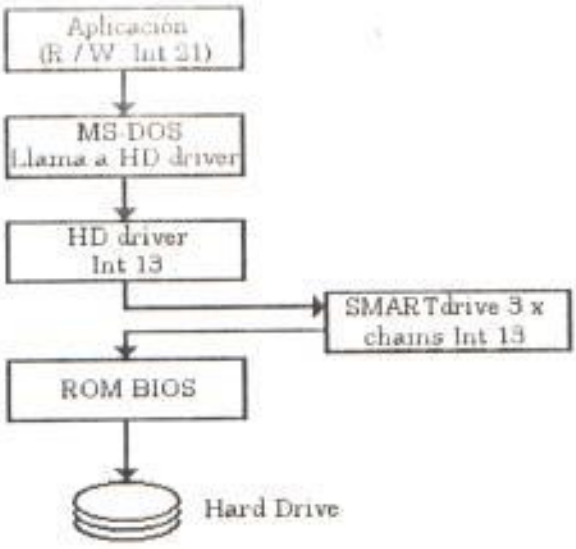

Figura 3-2. SMARTDrive 3.x

SMARTDrive 4.0 está diseñado como un cache de bloque orientado a disco. Este se acopla en el sistema en el nivel del dispositivo del driver de MS-DOS en lugar del nivel de la interrupción 13 del ROM BIOS. Cada bloque de device drive en el MS-DOS device drive encadenado es "adelantado" por un módulo SMARTDrive 4.0 que provee el caching actual. Este cumple con los siguientes beneficios:

- **Independencia de la interface INT 13.** Muchos device drivers no usan la interface INT 13. Esto significa que SMARTDrive 4.0 puede mantener en cache estos dispositivos donde SMARTDrive 3.x no lo hacia. Ejemplo son Bernoully, algunas hard card, y muchos SCSI y WORM drives.

- Independencia de la geometría del disco. Algunos administradores y controladores de disco usan un esquema de mapeo de la geometria del disco que causa una geometria lógica para ser diferenciada de la geometria fisica. Ejemplos son muchos sistemas PS/2, Ontrack Disk manager, y varias controladoras de disco. Caches basados en la INT 13 son sensibles a esto y a menudo causan problemas. Por ejemplo Ontrack Disk Manager actualmente cambia la geometria del disco especificada en el ROM BIOS en su ejecución confundiendo así al SMARTDrive 3.x.

La API determina la geometria real, pero requiere la detección del disk-manager y generalmente complica el asunto. A menudo tracks lógicos cruzan la región de tracks fisicos, lo cual causa desvío de cache para incurrir a recargo de performance. También para eludir la limitación del cilindro 1024 del ROM BIOS, administradores de disco y controladoras "plegaran" múltiples tracks en un sólo

track lógico. Esto reditúa el problema anterior, así como obliga a los track de cache tener un gran track buffer. En algunos casos esto es tan grande como 32.5K y puede residir en memoria baja. El diseño de SMARTDrive 4.0 elimina el problema de la mala unión de la geometría.

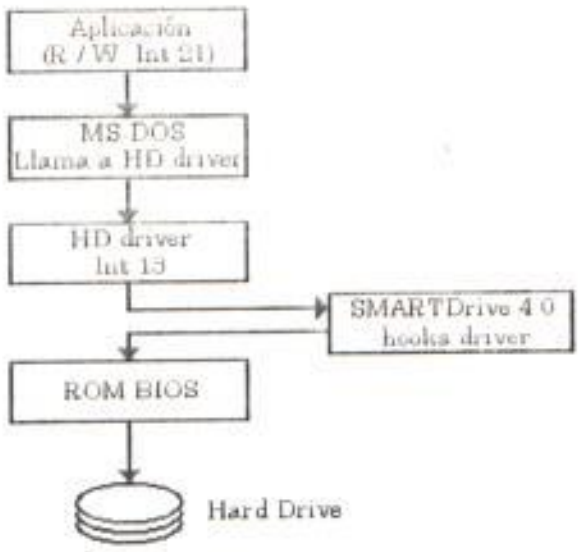

Figura 3-2. SMARTDrive 4.0

SMARTDrive 4.0 es también un write-behind cache. Este adiciona una mejora de perfomance significativa donde los archivos están siendo escritos e implementando un esquema de bits "validos" para prevenir trashing del disco en caso de acceso I/O aleatorio.

SMARTDrive actualmente usa un algoritmo de reemplazo de página FIFO e implementa un algoritmo de contracción que libera memoria para Windows de una manera similar a SMARTDrive 3.x. La diferencia es que SMARTDrive 4.0 espera para el esparcimiento de la inicialización de Windows mientras que SMARTDrive 3.x provee una interface IOCTL. El efecto neto es idéntico, pero SMARTDrive 4.0 es mucho más simple. Cuando Windows finaliza la sesión , el proceso es revertido y la memoria es re adquirida por SMARTdrive 4.0.

CAPITULO CUATRO

COMO CREAR UNA APLICACION

Este capitulo explica como crear un aplicación simple para Microsoft Windows

UNA APLICACION ESTANDAR DE WINDOWS : GÉNERIC

Una aplicación estandar a Windows es cualquier aplicación que especificcamente esta escrita para correr con Windows, y que usa la interface de aplicaciones de Windows (aplication program interface API), para producir sus tareas. Cada aplicación Windows tiene una ventana principal (denominada WinMain) y un procedimiento de ventanas.

Generic es una aplicación estandar, tiene una función WinMain y un procedimiento de ventana, tiene un borde, un menu de aplicaciones, y botones de maxizar y minimizar. El menu de aplicaciones contiene un menu de Ayuda, con el comando About, el cual cuando es selecionado por el usuario muestra un dialog box.

LA FUNCION WINMAIN

Al igual que la función main en el lenguaje C, la función WinMain es el punto de entrada para la aplicación Windows. Cada aplicación Windows debe tener una función WInMain, ninguna aplicación se ejecutara sin ella. Esta función hace lo siguiente:

- Llama a la función de inicialización que registra la windows classes, crea ventanas, y realiza cualquier otra inicialización necesaria.

- Entra a un lazo de mensajes para procesar mensajes desde la cola de aplicaciones.

- Termina la aplicación cuando el lazo de mensajes recibe un mensaje WM_QUIT.

La función WinMain tiene la siguiente forma:

```
int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;          /* instancia actual          */
HANDLE hPrevInstance;      /* instancia previa          */
LPSTR lpCmdLine;           /* linea de comandos         */
int nCmdShow;              /* tipo show-window (open/icon) */
{
```

Cuando el usuario inicia una aplicación, Windows pasa los siguientes parámetros a la función WinMain:

hInstance	El instance handle de la aplicación
hPrevInstance	El handle de otra instancia de la aplicación si alguna esta corriendo. Si ninguna esta ejecutándose Windows establece este parámetro a NULL.
lpCmdLine	Un puntero largo para un linea de comando null-terminated
nCmdShow	Un entero que especifica si mostrar la ventana de la aplicación como un icono o como una ventana. La aplicación pasa este valor a la función ShowWindows, cuando se llama esta función para mostrar la ventana principal de la aplicación.

TIPO DE DATOS Y ESTRUCTURAS

La función WinMain usa varios tipos de datos especiales para definir sus parámetros. Si bien los tipos de datos de Windows son a menudo equivalentes a los tipos de datos familiares del lenguaje C, ellos están intentados para ser más descriptivos y puedan ayudar a un mejor entendimiento de las variables o parámetros usados en la aplicación. En general las aplicaciones Windows usan muchos más tipos de datos que las aplicaciones encontradas en lenguaje C.

Los tipos de datos Windows están definidas en el archivo WINDOWS.H. Para usar estas definiciones debe incluir el este archivo en cada archivo fuente, usando la siguiente linea en al inicio del archivo.

```
#include <windows.h>
```

Los siguientes son los tipos de datos más comúnmente usados:

Tipo	Significado
WORD	Especifica un entero sin signo de 16-bit
LONG	Especifica un entero sin signo de 32-bit
HANDLE	Identifica un entero sin signo de 16-bit para ser usado como handle
HWND	Identifica un entero sin signo de 16-bit para ser usando como un handle de una ventana
LPSTR	Especifica una dirección de 32-bit de una string de caracteres (o tipo char)
FARPROC	Especifica una dirección de una función de 32-bit

Las siguientes son algunas de las estructuras más comúnmente usadas

MSG	Contiene la información referente a un mensaje de entrada desde la cola de aplicaciones de Windows
WNDCLASS	Define una windows class
PAINTSTRUCT	Define una estructura para pintar un área de cliente de una ventana

RECT Define un rectángulo

HANDLES

Dos de los parámetros de la función WinMain (hInstance y hPrevInstance) son denominados handless. Un handle es un entero único que Windows usa para identificar un objeto creado o usado por la aplicación. Windows usa una gran cantidad de handles, identificando objetos tales como ventanas menús, controles, asignación de memoria, dispositivos de salida, archivos, y dispositivos de interface gráfica (graphics device interface GDI) pens y brushes.

La mayoría de handles están indicados en tablas internas. Windows usa índices de handles para accesar la información almacenadas en estas tablas. Típicamente, una aplicación tiene acceso unicamente handle y no a la información. Cuando la aplicación debe examinar o cambiar la información, esta proporciona el handle y Windows hace el resto.

INSTANCIAS

No sólo puede correr m s de una aplicación en Windows, Ud. puede correr m s de una copia, o instancia, de la misma aplicación a la vez. Para distinguir una instancia de otra Windows proporciona un instance handle único (entero único que identifica la instancia) cada momento que llama a la función WinMain para iniciar la aplicación.

Con algunos sistemas multitareas, ejecutar múltiples instancias de una misma aplicación a la vez, el sistema carga una copia fresca del código y datos de la aplicación en memoria y ejecuta esta copia. Con Windows cuando una nueva instancia de la aplicación es iniciada unicamente el dato para la aplicación es cargado. Windows usa el mismo código para todas las instancias de la aplicación. Esto salva tanto espacio como sea posible para otras aplicaciones y otros datos. Sin embargo este m,todo requiere que el segmento de código de la aplicación permanezca sin cambiar mientras que la aplicación esta corriendo. Esto significa que Ud. no debe almacenar datos en un segmento de código o cambiar el código mientras la aplicación esta ejecutándose.

Para la mayoría de aplicaciones, la primera instancia tiene un papel especial, muchos de lo recursos y aplicaciones creadas, tales como windows classes, están generalmente disponibles a todas las aplicaciones. Consecuentemente sólo la primera instancia de la aplicación crea estos recursos.

Para determinar cual es la primera instancia Windows establece el parámetro hPrevInstance de WinMain a NULL si no hay una instancia previa. El siguiente ejemplo muestra como chequear que una instancia previa no existe:

```
int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;              /* instancia actual                */
HANDLE hPrevInstance;          /* instancia previa                */
LPSTR lpCmdLine;               /* linea de comandos               */
int nCmdShow;                  /* tipo show-window (open/icon)    */
```

```
{
  if (hPrevInstance == NULL)
    ...
}
```

Para prevenir que un usuario inicie más de una instancia de la aplicación, la aplicación podría chequear el parámetro hPrevInstance y retornar a Windows si el parámetro no es nulo, por ejemplo:

```
if (hPrevInstance)
  return NULL;
```

REGISTRANDO UNA CLASE (WINDOWS CLASS)

Antes de crear cualquier ventana Ud. debe tener una window class, que es un template que define los atributos de una ventana, tal como la figura del cursor y el nombre del menú de ventana. La 'clase' de ventana también,n especifica el procedimiento de ventana que procesa mensajes para todas las ventanas en la clase.

Usted debe registrar una window class antes de crear una ventana que pertenece a esta clase. Esto lo hace llenando una estructura WNDCLASS con información acerca de la clase y pasando los parámetros a la función RegisterClass.

LLENANDO LA ESTRUCTURA WNDCLASS

Esta estructura provee información a Windows acerca del nombre, atributos, recursos, y procedimiento de ventana para una clase de ventana. La estructura WNDCLASS contiene los siguientes miembros:

Miembro	Descripción
lpszClassName	Apunta al nombre de la window class. Debe ser única
hInstance	Identifica la instancia de la aplicación que esta registrando la clase
lpfnWndProc	Apunta al procedimiento de ventana para producir trabajo en la ventana
style	Especifica el estilo de la clase, tal como re encuadrar la ventana automáticamente cada vez que esta se mueve o cambia de tamaño
hbrBackground	Identifica el brush usado para pintar el fondo de la ventana
hCursor	Identifica el cursor usado en la ventana
hIcon	Identifica el icono usado para representar la ventana minimizada
lpszMenuName	Apunta al nombre de un recurso del menú
cbClsExtra	Especifica el número de bytes extras para asignar a esta estructura. Son inicializados a cero

clWndExtra Especifica el número de bytes extra para asignar a todas las ventanas creadas con esta clase. Son inicializados a cero

Algunos miembros tales como lpszClassName, hInstance, lpfnWndProc, deben estar asignados a valores, otros miembros pueden ser establecidos a cero. Windows usa un atributo por omisión para la ventana creada usando la clase. El siguiente ejemplo muestra como llenar una estructura de ventana:

```
BOOL InitApplication(hInstance)
HANDLE hInstance;                    /* instancia actual                    */
{
   WNDCLASS  wc;

   /* llenado en la estructura de window class con parámetros que describen  */
   /* la ventana principal (main window)                                     */

   wc.style = NULL;                  /* Class style(s)                      */
   wc.lpfnWndProc = MainWndProc;     /* Función para recuperar mensajes     */
                                     /* para ventanas de esta clase.        */
   wc.cbClsExtra = 0;                /* Ningún dato extra por-class         */
   wc.cbWndExtra = 0;                /* Ningún dato extra por-window.       */
   wc.hInstance = hInstance;      /* Aplicación a la que pertenece la clase.*/
   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor = LoadCursor(NULL, IDC_ARROW);
   wc.hbrBackground = GetStockObject(WHITE_BRUSH);
   wc.lpszMenuName =  "GenericMenu";  /* Nombre del menú en archivo .RC      */
   wc.lpszClassName = "GenericWClass"; /* Nombre usado para llamar a         */
                                     /* CreateWindow.                       */

   /* Registra la window class y retorna código success/failure.             */

   return (RegisterClass(&wc));

}
```

4.4 CREANDO UNA VENTANA

Usted puede crear una ventana usando la función CreateWindow. Esta función llama a Windows para crear una ventana que tiene un estilo especifico y pertenece a una clase especifica. CreateWindows toma varios parámetros:

* nombre de la clase
* titulo de la ventana
* estilo de la ventana
* posición de la ventana
* handle de la ventana-padre
* menú handle
* instance handle
* 32 bits de datos adicionales

el siguiente ejemplo crea un ventana perteneciente a la clase GenericWClass (creada por la estructura WNDCLASS)

```
   /* Crea un main window para esta instancia de la aplicación.  */
```

```
hWnd = CreateWindow(
        "GenericWClass",          /* Ver llamada RegisterClass()          */
        "Aplicación Genérica del Ejemplo", /* Texto para título de ventana */
        WS_OVERLAPPEDWINDOW,      /* Window style                         */
        CW_USEDEFAULT,            /* Posición horizontal por defecto.     */
        CW_USEDEFAULT,            /* Posición vertical por defecto.       */
        CW_USEDEFAULT,            /* Ancho por defecto                    */
        CW_USEDEFAULT,            /* Altura por defecto                   */
        NULL,                     /* Ventana solapada no tiene padre.     */
        NULL,                     /* Usa el window class menu             */
        hInstance,                /* Esta instancia pertenece a esta ventana. */
        NULL          /* Pointer no necesitado                            */
);
```

Este ejemplo crea una ventana solapada que tiene el estilo WS_OVERLAPPEDWINDOW y que pertenece a la clase creada por el código en el ejemplo precedente.

4.6.1. MOSTRANDO Y ACTUALIZANDO UNA VENTANA

Si bien CreateWIndow crea una ventana, esta no es mostrada automáticamente. En su lugar la aplicación debe mostrar la ventana usando la función ShowWindow y debe actualizar el area de ventana de cliente usando la función UpdateWindow.

ShowWindows le dicce a Windows que muetre la nueva ventana. Para la aplicación, WinMain puede llamar a ShowWindow poco después de crear la ventana y pasarle el parámetro nCmdShow a esta. CmdShow indica a la aplicación si mostrar la ventana como una ventana abierta o como un icono. Después de llamar a ShowWindow WinMain debe llamar a la función UpdateWindow. El siguiente ejemplo ilustra como mostrar y actualizar una ventana:

```
ShowWindow(hWnd, nCmdShow);       /* muestra la ventana                   */
UpdateWindow(hWnd);               /* envia un mensaje WM_PAINT            */
```

4.6.2. CREANDO UN LAZO DE MENSAJES

Una vez que la aplicación ha creado y mostrado una ventana, WinMain puede iniciar su primera tarea: leer mensajes de la cola de aplicaciones y despacharla a la ventana apropiada. Window hace esto usando un lazo de mensajes, el cual es un lazo de programa, tipicamente creado usando la sentencia while.

Windows no envia directamente las entradas a la aplicación. En su lugar, este situa todas las entradas de teclado y mouse dentro de una cola de aplicaciones (junto con otros mensajes pasados por Windows y otras aplicaciones). La aplicación debe leer los mensajes de la cola, recuperarlos y despacharlos a la ventana apropiada para que puedan ser procesados.

El lazo de mensajes lo más simple posible consiste de las funciones GetMessge y DispatchMessage. Este lazo tiene la siguiente forma:

```
MSG msg;

while (GetMessage(&msg,          /* estructura mensaje                    */
                  NULL,/* handle de la ventana recibiendo mensaje   */
                  NULL,/* el mensaje más bajo a examinar            */
                  NULL)) {        /* el mensaje más alto a examinar        */

        DispatchMessage(&msg);

}
```

TERMINANDO UNA APLICACION

Su aplicación termina cuando la función WinMain retorna el control a Windows. WinMain puede retornar el control en cualquier momento antes de iniciar el lazo de mensajes. Típicamente una aplicación chequea el lazo de mensajes cada momento para asegurarse que cada clase esta registrada, y cada ventana es creada. Si hay un error, la aplicación puede mostrar un mensaje antes de terminar.

Una vez que WinMain entra al lazo de mensajes, sin embargo la única forma de terminar el lazo es situando un mensaje WM_QUIT en la cola de aplicaciones, usando la función PostQuitMessage. Cuando GetMessage recupera un mensaje WM_QUIT, esta retorna un valor NULL, el cual termina el lazo de mensajes. Generalmente, un procedimiento de ventana situa un mensaje WM_QUIT cuando la ventana principal esta siendo destruida (esto es, cuando el procedimiento de ventrana ha recibido un mensaje WM_DESTROY).

En general, la convensión más facil para código retornado es aquella usada por las aplicaciones estandar del lenguaje C: cero para un ejecución satisfactoria, y diferente de cero para un error. La función PostQuitMessage le permite al procedimiento de ventana especificar el valor retornado. Este valor entonces es copiado al parámetro wParam del mensaje WM_QUIT. Para retornar este valor después de terminar el lazo de mensaje, use al siguiente sentencia:

```
return (msg.wParam);          /* retorna el valor desde PostQuitMessage    */
```

FUNCIONES DE INICIALIZACION

La mayoría de aplicaciones usa dos funciones de inicialización definidas localmente:

* La función de inicialización principal produce trabajo que debe ser realizado sólo una vez para todas las instancias de las aplicaciones

* La función de inicialización de instancia realiza tareas que debeb ser realizadas una vez para cada instancia de la aplicación

Estas funciones de inicialización mantiene la función WinMain simple y legible; esta también organiza las tareas de inicialización tal que ellas puedan ser aisladas en segmentos de código separados y descartados después de su uso

LA FUNCION DE INICIALIZACION PRINCIPAL

La función de inicialización para la aplicación genérica se muestra como sigue:

```
BOOL InitApplication(hInstance)
HANDLE hInstance;                    /* instancia actual                    */
{
   WNDCLASS  wc;

   /* llenado en la estructuta de window class con parametros que describen  */
   /* la ventana principal (main window).                                    */

   wc.style = NULL;                  /* Class style(s).                     */
   wc.lpfnWndProc = MainWndProc;     /* Función para recuperar mensajes     */
                                     /* para ventanas de esta clase.        */
   wc.cbClsExtra = 0;                /* Ningún dato extra por-class         */
   wc.cbWndExtra = 0;                /* Ningún dato extra por-window.       */
   wc.hInstance = hInstance;     /* Aplicación a la que pertenece la clase. */
   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor = LoadCursor(NULL, IDC_ARROW);
   wc.hbrBackground = GetStockObject(WHITE_BRUSH);
   wc.lpszMenuName =  "GenericMenu";  /* Nombre del menu en archivo .RC      */
   wc.lpszClassName = "GenericWClass"; /* Nombre usado para llamar a         */
                                     /* CreateWindow.                       */

   /* Registra la window class y retorna código success/failure.             */

   return (RegisterClass(&wc));

}
```

LA FUNCION DE INICIALIZACION DE INSTANCIAS

La función de inicialización de instancias para la aplicación genérica se muestra como sigue:

```
BOOL InitInstance(hInstance, nCmdShow)
   HANDLE      hInstance;          /* Identificador de la instancia actual.  */
   int         nCmdShow;           /* Parametro para la primera llamada a    */
                                   /* ShowWindow().                          */
{
   HWND        hWnd;               /* Main window handle.                    */

   /* Salva el instance handle en una variable estática, la cual puede ser   */
   /* usada en subsecuentes llamadas desde esta aplicación a to Windows.     */

   hInst = hInstance;

   /* Crea un main window para esta instancia de la aplicación.              */

   hWnd = CreateWindow(
```

```
        "GenericWClass",          /* Ver llamada  RegisterClass()          */
        "Aplicación Generica del Ejemplo", /* Texto para titulo de ventana  */
        WS_OVERLAPPEDWINDOW,      /* Window style.                         */
        CW_USEDEFAULT,            /* Posición horizontal por defecto.      */
        CW_USEDEFAULT,            /* Posición vertical por defecto.        */
        CW_USEDEFAULT,            /* Ancho por defecto.                    */
        CW_USEDEFAULT,            /* Altura por defecto.                   */
        NULL,                /* Ventana solapada no tiene padre.           */
        NULL,                /* Usa el window class menu.                  */
        hInstance,           /* Esta instancia pertenece a esta ventana.   */
        NULL          /* Puntero no necesitado.                            */
    );

/* Si la ventana no puede ser creada, retorna "failure"                     */

if (!hWnd)
        return (FALSE);

/* Hace la ventana visible; actualiza el area de cliente, y retorna "success" */

ShowWindow(hWnd, nCmdShow);      /* Muestra la ventana                    */
UpdateWindow(hWnd);              /* Envia mensaje WM_PAINT                */
return (TRUE);              /* Retorna el valor desde PostQuitMessage     */

}
```

LA VENTANA DE PROCEDIMIENTOS

Una ventana de procedimientos responde a entrada y mensajes de ventanas de Windows. El procedimiento puede ser corto, procesar sólo un mensaje o dos, o puede ser muy complejo procesar muchos tipos de mensajes para una variedad de ventanas de aplicaciones. En cualquiera de los caso, cada ventana debe tener una ventana de procedimientos.

Una ventana de procedimientos tiene la siguiente forma:

```
long FAR PASCAL MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;                       /* window handle                        */
UINT message;                    /* tipo de mensaje                      */
WPARAM wParam;                   /* información adicional                */
LPARAM lParam;              /* información adicional                     */
{
    .
    .
    .
    switch (message) {
    .
    .
    .
    default:                     /* Pasa este adelante si no ha sido procesado */
        return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return (NULL);
}
```

El parámetro mensaje define el tipo de mensaje. Puede usar este parámetro en una sentencia switch para procesar directamente a el case correcto.

4.3. CREANDO UN 'ABOUT DIALOG BOX'

Usted podría incluir un About dialog box con cada aplicación. Un dialox box es una venatana temporal para mostrar información o sugerencias de entrada al usuario. Ud. crea y muestra una ventana de información usando la función DialogBox. Esta toma una template dialox box, una dirección de instancia de procedimiento, y un handle de la una ventana padre, y crea un dialog box a través del cual su aplicación muestra información de salida y sugiere entradas al usuario.

Para mostrar y usar un about dialog box siga los siguiente pasos:

1. Cree un template de un About dialog box y adiciones este a su archivo de definición de recursos.
2. Adicione su About dialog box a su archivo fuente en leguaje C.
3. Exporte el procedimiento de About dialog box en su archivo de definición de módulos.
4. Adicione un menú a su archivo de definición de recursos.
5. Procese el mensaje WM_COMMAND en su código de aplicaciones.

4.4. CREANDO GENERIC

Para crear un aplicación de ejemplo genérica, siga los siguientes pasos:

1. Cree un archivo fuente en lenguaje C (.C).
2. Cree un header file (.H).
3. Cree un archivo de definiciones de recursos (.RC).
4. Cree un archivo de definición de módulos (.DBF).
5. Cree un makefile.
6. Corra el utilitario de mantenimiento de programas (NMAKE) en el archivo a compilar y enlace la aplicación.

4.4.1. CREANDO UN ARCHIVO FUENTE EN LENGUAJE C

El archivo fuente en lenguaje C contiene la función WinMain, la función MainWndProc, el procedimiento de About dialog box, y las funciones de inicialización InitApplication y InitInstance. Nombre el archivo GENERIC.C.

El contenido del archivo se muestra a continuación:

```
/*****************************************************************************

    PROGRAMA: Generic.c

    PROPOSITO: Un patrón Generico para aplicaciones Windows

    FUNCIONES:
```

```
WinMain() - llama a la función de inicialización, procesa lazos de mensajes
InitApplication() - inicializa datos de ventanas y registros de ventanas
InitInstance() - salva el 'instance handle' y crea la ventana principal
MainWndProc() - procesa mensajes
About() - procesa mensaje para el "About" dialog box

COMENTARIOS:

Windows puede tener varias copias de su aplicación ejecutandose al mismo
tiempo. La variable hInst mantiene pista de cual instancia de esta aplicación es
tal que el procesamiento ser para la ventana correcta.

********************************************************************************/

#include "windows.h"        /* requerida para todas las aplicaciones Windows */
#include "generic.h"        /* especifica a este programa                    */

HANDLE hInst;               /* instancia actual                              */

/********************************************************************************

FUNCTION: WinMain(HANDLE, HANDLE, LPSTR, int)

PROPOSITO: llamar a la función de inicialización, procesar lazos de mensajes

COMENTARIOS:

Windows reconoce esta función por nombre como punto de entrada inicial para
el programa . Esta función llama a la rutina de inicialización de la aplicación, si
ninguna otra instancia esta corriendo, y siempre llama la rutina de
inicialización de instancia. Entonces ejecuta un lazo de recuperación y despacho
de mensajes que estan en el nivel-tope de la estructura de control para el resto
de la ejecución. El lazo es terminado cuando un menaje WM_QUIT es recibido,
momento en el cual esta función sale de la instancia de la aplicación retornando
el valor pasado por PostQuitMessage(). Si esta función debe terminar antes de
entrar al lazo de mensaje, este retorna el valor convensional NULL.

********************************************************************************/

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;           /* instancia actual                              */
HANDLE hPrevInstance;       /* instancia previa                              */
LPSTR lpCmdLine;            /* linea de comandos                             */
int nCmdShow;               /* tipo show-window (open/icon)                  */
{
    MSG msg;                /* mensaje                                       */

    if (!hPrevInstance)     /* Otra instancia de la app esta corriendo ?     */
        if (!InitApplication(hInstance))    /* Inicializa things compartidos */
            return (FALSE); /* Sale si no es habilitado para inicializar     */

    /* Realiza las inicializaciones que se aplican a una instancia especifica */

    if (!InitInstance(hInstance, nCmdShow))
        return (FALSE);

    /* Adquiere and despacha mensajes hasta que un mensaje WM_QUIT es        */
    /* recibido.                                                             */
```

```
while (GetMessage(&msg,     /* estructura de mensaje                    */
        NULL,       /* handle de la ventana que recibe el mensaje */
        NULL,       /* mensaje mas bajo a examinar                */
        NULL))      /* mensaje mas alto a examinar                */
{
TranslateMessage(&msg);     /* Traslada código de claves virtuales*/
DispatchMessage(&msg);      /* Despacha mensaje a ventanas        */
}
return (msg.wParam);/* Retorna el valor desde PostQuitMessage     */
}

/*********************************************************************

    FUNCTION: InitApplication(HANDLE)

    PROPOSITO: Inicializa los datos de ventanas y registros de la window class

    COMENTARIOS:

    Esta función es llamada en el momento de inicialización unicamente si ninguna
    otra instancia de la aplicación esta corriendo. Esta funcion realiza las tareas de
    inicialización que pueden ser hechas una vez para cualquier número de
    instancias corriendo.

    En este caso, inicializamos una window class llenando externamente la
    estructura de datos de tipo WNDCLASS y llamando la función de Windows
    RegisterClass(). Puesto que todas las instancias para esta aplicación usa la
    misma window class, unicamente necesitamos hacer esto cuando la primera
    instancia es inicializada.

*********************************************************************/

BOOL InitApplication(hInstance)
HANDLE hInstance;                   /* instancia actual                */
{
    WNDCLASS  wc;

    /* llenado en la estructuta de window class con parametros que describen  */
    /* la ventana principal (main window).                                    */

    wc.style = NULL;                    /* Class style(s).                    */
    wc.lpfnWndProc = MainWndProc;       /* Función para recuperar mensajes    */
                                        /* para ventanas de esta clase. */
    wc.cbClsExtra = 0;                  /* Ningún dato extra por-class.*/
    wc.cbWndExtra = 0;                  /* Ningún dato extra por-window.      */
    wc.hInstance = hInstance;       /* Aplicación a la que pertenece la clase.*/
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName =  "GenericMenu";  /* Nombre del menu en archivo .RC  */
    wc.lpszClassName = "GenericWClass"; /* Nombre usado para llamar a     */
                                        /* CreateWindow.                  */

    /* Registra la window class y retorna código success/failure.         */

    return (RegisterClass(&wc));
```

```
/*****************************************************************************

    FUNCTION: InitInstance(HANDLE, int)

    PROPOSITO: Salva el instance handle y crea main window

    COMENTARIOS:

        Esta función es llamada en el momento de inicialización para cada instancia de
        la aplicación. Esta función realiza las tareas de inicialización que no pueden ser
        compartidas por multiples instancias.

        En este caso, salvamos el instance handle en una variable estática y creamos y
        mostramos el the main program window.

*****************************************************************************/

BOOL InitInstance(hInstance, nCmdShow)
    HANDLE          hInstance;          /* Identificador de la instancia actual.        */
    int             nCmdShow;           /* Parametro para la primera llamada a          */
                                        /* ShowWindow()                                 */
{
    HWND            hWnd;               /* Main window handle.                          */

    /* Salva el instance handle en una variable estática, la cual puede ser             */
    /* usada en subsecuentes llamadas desde esta aplicación a to Windows.               */

    hInst = hInstance;

    /* Crea un main window para esta instancia de la aplicación.  */

    hWnd = CreateWindow(
            "GenericWClass",            /* Ver llamada  RegisterClass()         */
            "Aplicación Generica del Ejemplo", /* Texto para titulo de ventana  */
            WS_OVERLAPPEDWINDOW,        /* Window style.                        */
            CW_USEDEFAULT,              /* Posición horizontal por defecto.     */
            CW_USEDEFAULT,              /* Posición vertical por defecto.       */
            CW_USEDEFAULT,              /* Ancho por defecto.                   */
            CW_USEDEFAULT,              /* Altura por defecto.                  */
            NULL,                       /* Ventana solapada no tiene padre.     */
            NULL,                       /* Usa el window class menu.            */
            hInstance,                  /* Esta instancia pertenece a esta ventana.  */
            NULL        /* Pointer no necesitado.                               */
    );

    /* Si la ventana no puede ser creada, retorna "failure"                     */

    if (!hWnd)
            return (FALSE);

    /* Hace la ventana visible; actualiza el area de cliente; y retorna "success"  */

    ShowWindow(hWnd, nCmdShow);         /* Muestra la ventana                   */
    UpdateWindow(hWnd);                 /* Envia mensaje WM_PAINT               */
    return (TRUE);                      /* Retorna el valor desde PostQuitMessage  */

}
```

```
/*****************************************************************************

  FUNCTION: MainWndProc(HWND, UINT, WPARAM, LPARAM)

  PROPOSITO:  Procesa mensajes

  MENSAJES:

  WM_COMMAND    - menu de aplicación (About dialog box)
  WM_DESTROY    - destruye ventana

  COMENTARIOS:

  Para procesar el mensaje IDM_ABOUT, llama a MakeProcInstance() para traer
  la dirección de la instancia actual de la función About(). Entonces llama al
  Dialog box el cual creará el box acorde a la información en su archivo generic.rc
  y cambia el control otra vez a la función About(). Cuando este retorna, libera la
  dirección de la instancia.

*****************************************************************************/

long FAR PASCAL MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;                      /* window handle                        */
UINT message;                   /* tipo de mensaje                      */
WPARAM wParam;                  /* información adicional                */
LPARAM lParam;                  /* información adicional                */
{
  FARPROC lpProcAbout;          /* pointer a la función "About" */

  switch (message) {

  case WM_COMMAND:/* mensaje: comando del application menu        */
        if (wParam == IDM_ABOUT) {
        lpProcAbout = MakeProcInstance(About, hInst);

        DialogBox(hInst,                /* instancia actual             */
                "AboutBox",             /* recurso para usar            */
                hWnd,                   /* handle del padre             */
                lpProcAbout);           /* dirección de la instancia About()  */

        FreeProcInstance(lpProcAbout);
        break;
        }
        else                            /* Deja que Windows procese este        */
        return (DefWindowProc(hWnd, message, wParam, lParam));

  case WM_DESTROY:              /* mensaje: ventana existente destruida         */
        PostQuitMessage(0);
        break;

  default:                      /* Pasa este adelante si no ha sido procesado   */
        return (DefWindowProc(hWnd, message, wParam, lParam));
  }
  return (NULL);

}

/*****************************************************************************/
```

```
    FUNCTION: About(HWND, unsigned, WORD, LONG)

    PROPOSITO:  Procesa mensajes para "About" dialog box

    MENSAJES:

    WM_INITDIALOG - initializa dialog box
    WM_COMMAND    - Entrada recivida

    COMMENTS:

    Ninguna inicialización es necesitada para este dialog box particular, pero TRUE
    debe ser retornado a Windows.

    Espera para que el usuario presione el botón "Ok", entonces cierra el dialog box.

****************************************************************************/

BOOL FAR PASCAL About(hDlg, message, wParam, lParam)
HWND hDlg;                              /* window handle del dialog box     */
unsigned message;                       /* tipo de mensaje                  */
WORD wParam;                            /* información especifica de mensaje */
LONG lParam;
{
    switch (message) {
    case WM_INITDIALOG:                 /* mensaje: initializa dialog box   */
            return (TRUE);

    case WM_COMMAND:            /* mensaje: receve un comando       */
            if (wParam == IDOK ||       /*  box "OK" seleccionado?          */
                wParam == IDCANCEL) /* System menu close command?       */
            {
            EndDialog(hDlg, TRUE);      /* Sale del dialog box              */
            return (TRUE);
            }
            break;
    }
    return (FALSE);                     /* No puede procesar un mensaje     */
}
```

CREANDO UN HEADER FILE

El header file contiene definiciones y declaraciones requeridas por el archivo fuente en lenguaje C, que son incorporadas al código fuente por la directiva #include. Nombre el archivo GENERIC.H. Este debe verse como sigue:

```
#define IDM_ABOUT 100

int PASCAL WinMain(HANDLE, HANDLE, LPSTR, int);
BOOL InitApplication(HANDLE);
BOOL InitInstance(HANDLE, int);
long FAR PASCAL MainWndProc(HWND, UINT, WPARAM, LPARAM);
BOOL FAR PASCAL About(HWND, unsigned, WORD, LONG);
```

CREANDO UN ARCHIVO DE DEFINICION DE RECURSOS

El archivo de definición de recursos debe contener el menú de ayuda y el template del dialog box. Nombre el archivo GENERIC.RC. Este debería verse como sigue:

```
#include "windows.h"
#include "generic.h"

GenericMenu MENU
BEGIN
  POPUP      "&Ayuda"
  BEGIN
    MENUITEM "&Acerca de Aplicacion Generica...", IDM_ABOUT
  END
END

AboutBox DIALOG 22, 17, 170, 80
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Acerca de Generic"
BEGIN
  CTEXT "Microsoft Windows"    -1,      0, 5, 174, 8
  CTEXT "Aplicacion Generica"  -1,      0, 14, 174, 8
  CTEXT "Topico de Graduacion" -1,      0, 34, 174, 8
  CTEXT "Escuela Superior Politecnica del Litoral" -1,   0, 46, 174, 8
  DEFPUSHBUTTON "Retornar"    IDOK,     72, 59, 35, 14,     WS_GROUP
END
```

CREANDO UN ARCHIVO DE DEFINICION DE MODULO

El archivo de definición de módulos debe contener la definición de módulos para generic. Nombre al archivo GENERIC.DEF. Este debe verse como sigue:

```
; archivo de definición de módulo para generic -- usado por LINK.EXE

NAME          Generic          ; nombre del módulo de aplicaciones

DESCRIPTION  'Ejemplo de Aplicaciones en Microsoft Windows'

EXETYPE      WINDOWS    ; requeridas por todas las Windows applications

STUB         'WINSTUB.EXE' ; Genera mensajes de error si la aplicación
                           ; esta corriendo sin Windows

; CODE puede ser movido en memoria y discarded/reloaded
CODE  PRELOAD MOVEABLE DISCARDABLE

; DATA debe ser MULTIPLE si el programa puede ser invocado mas de una vez
DATA  PRELOAD MOVEABLE MULTIPLE


HEAPSIZE     1024
STACKSIZE    5120          ; mínimo recomendado para Windows applications

; todas las funciones que serán llamadas por cualquier rutina Windows
; DEBE ser exportada.
```

```
EXPORTS
  MainWndProc  @1  ; nombre de la función que procesa ventanas
  About        @2  ; nombre de la función que procesa "About"
```

CREANDO UN MAKEFILE

Una vez que tiene los archivos fuentes, Ud. puede crear un makefile para generic, y entonces compilar y enlazar la aplicación usando NMAKE. Para compilar y enlazar generic, el makefile debe llevar estos pasos,

* Usar CL para compilar el archivo GENERIC.C

* Usar el enlazado ejecutable de segmentos de Microsoft (LINK), para enlazar el archivo GENERIC.OBJ con las librerías de Windows y el archivo de definición de módulos, GENERIC.DEF.

* Usar RC para crear un archivo de recursos binarios y adicionar este al archivo ejecutable de la aplicación.

La siguiente compilará y enlazará adecuadamente los archivos creados por generic:

```
# Actualiza los recursos si es necesario

generic.res: generic.rc generic.h
   rc /r geenric.rc

# Actualiza el archivo objeto si es necesario

generic.obj: geenric.c generic.h
   cl /c /Gsw /Oas /Zpe generic.c

# Actualiza el archivo ejecutable si es necesario. (si no es necesario adiciona
# los recursos

generic.exe: generic.obj generic.def
   link /nod generic, , , slibcew libw, generic.def
   rc generic.red

# Si el archivo .RES es nuevo y el archivo .EXE no lo es, actualiza los
# recursos. Note que el archivo .RC puede ser actualizado sin tener que
# compilar o enlazar el archivo

generic.exe: generic.res
   rc generic.res
```

CORRIENDO EL UTILITARIO DE MANTENIMIENTO DE PROGRAMA

Una vez que ha creado el makefile, Ud. puede compilar y enlazar su aplicación corriendo NMAKE. El siguiente ejemplo ejecuta nmake usando los comandos en el archivo GENERIC:

```
nmake generic
```

CAPITULO CINCO

ADMINISTRACION DE MEMORIA

Todas las aplicaciones deben usar memoria para que ellas puedan ejecutarse. Debido a que Microsoft Windows version 3.1 es multitarea, varias aplicaciones pueden usar memoria simultáneamente. Windows administra la memoria disponible para asegurar que todas las aplicaciones tengan acceso a ella, y hacer el uso de memoria tan eficiente como sea posible.

USANDO LA MEMORIA

En el sistema administrador de memoria del Windows, sus aplicaciones pueden asignar bloques de memoria, denominados objetos de memoria. Usted puede asignar memoria tanto de un global heap como de un local heap.

El global heap es un pool (reservorio) de memoria disponible para todas las aplicaciones. El local heap es un pool de memoria libre disponible solamente para su aplicación. En el sistema administrador de memoria, Windows también maneja el segmento de código como el segmento de datos de su aplicaciones.

En algunos sistemas administradores de memoria, la memoria que usted asigna permanece fija el una localidad de memoria específica hasta que usted libere la misma. En Windows, la memoria asignada puede ser también movible y descartable.

Un objeto de memoria movible no tiene una dirección fija; Windows lo puede mover en cualquier momento a una nueva dirección. Los objetos de memoria movible le permiten a Windows hacer un mejor uso de la memoria libre. Por ejemplo, si un objeto de memoria movible separa dos objetos de memoria libres, Windows puede trasladar el objeto movible para combinar los objetos libres en un objeto contiguo.

La memoria descartable es similar a la memoria movible en que Windows puede moverla, excepto en que Windows también puede re-asignar un objeto descartable a una longitud cero si este debe usar el espacio para satisfacer un requerimiento de asignación. Re-asignar un objeto de memoria a una longitud cero destruye el dato que el objeto contenía, pero una aplicación siempre tiene la opción de volver a cargar el dato descartado cuando este es necesitado.

Cuando asigna un objeto de memoria, usted recibe un handle (manejador), en lugar de un pointer (puntero), para este objeto de memoria. El handle identifica la asignación de un objeto. Puede usarlo para recuperar la dirección actual del objeto cuando necesite accesar a la memoria.

Para accesar un objeto de memoria, usted enclava (bloquea) su respectivo handle. Esto temporalmente fija el objeto de memoria y retorna un puntero a su origen o inicio. Mientras un handle de memoria está enclavado, Windows no puede mover o descartar el objeto. Por lo tanto, después de que haya finalizado de usar el objeto, usted debería de desenclavar el handle tan pronto como sea posible. Mantener un handle de memoria enclavado hace que el administrador

manejador) de memoria del Window pierda eficiencia y pueda causar la falla de subsecuentes asignaciones a requerimientos de memoria.

Windows le permite compactar su memoria. "Exprimiendo" la memoria libre de entre los objetos de memoria asignados. Windows acumula los objetos de memoria libres contiguos lo más grande posible, desde los cuales puede asignar objetos de memoria adicionales. Esta 'exprimición' es un proceso de mover y descartar (si es necesario) objetos de memoria. También puede descastar objetos de memoria individuales si temporalmente no tiene necesidad de ellos.

USANDO EL GLOBAL HEAP

El global heap contiene toda la memoria del sistema. Windows asigna la memoria que este necesita para código y dato desde el global heap cuando este arranca. Cualquier memoria restante libre en el global heap esta disponible para aplicaciones y librerías de Windows.

Las aplicaciones típicamente usan el global heap para asignaciones de memoria grandes (mayores de un kilobyte). Si bien usted puede asignar grandes objetos de memoria desde el global heap de la que puede asignar desde el local heap, hay una convención: debido a que es mas fácil manipular datos locales que datos globales, sus aplicaciones serán mas fácil de escribir si usted sólo usa datos locales.

Usted puede asignar cualquier tamaño de objeto de memoria del global heap. Las aplicaciones típicamente asignan sus objetos grandes desde el global heap; estos objetos pueden exceder de los 64K si la aplicación requiere tanto espacio de memoria contiguo. Windows provee servicios especiales para accesar datos pasado el primer segmento de 64K.

Para asignar objetos de memoria globales, use la función **GlobalAlloc**. Usted especifica el tamaño y el tipo (fijo, movible, o descartable); **GlobalAlloc** retorna un handle del objeto de memoria. Antes de poder usar el objeto de memoria, primero debe de enclavarlo (bloquearlo) usando la función **GlobalLock**, la cual retorna la dirección total de 32-bits del primer byte en el objeto de memoria, entonces puede usar un puntero largo (long pointer) para accesar los bytes en el objeto.

En el siguiente ejemplo, la función **GlobalAlloc** asigna 4096 bytes de memoria movible, y la función **GlobalLock** enclava estos de modo que los primeros 256 byte puedan ser establecidos a la dirección 0xFF

```
HANDLE hMem;
LPSTR lpMem;
int i;

if ( ( hMem = GlobalAlloc( GMEM_MOVEABLE, 4096 ) ) != NULL ) {
    if ( ( lpMem = GlobalLock( hMem ) != ( LPSTR ) NULL ) {
            for ( i = 0; i < 256; i++ )
                    lpMem[ i ] = 0xFF;
            GlobalUnlock( hMem );
    }
}
```

En este ejemplo, la aplicación desbloquea el handle de memoria usando la función **GlobalUnlock** inmediatamente después de acceder el objeto de memoria. Una vez que un objeto de memoria movible o descartable es bloqueado, Windows garantiza que el objeto permanecerá fijo en memoria hasta que el miembro sea desbloqueado. Esto significa que la dirección permanecerá válida tanto tiempo como el objeto permanezca bloqueado, pero esto también impide a Windows hacer un mejor uso de la memoria si otros requerimientos de asignación son hechos.

La función **GlobalAlloc** retorna un valor NULL si un requerimiento de asignación de memoria falla. Debería siempre chequear el valor retornado para asegurarse que este es un handle valido. Si usted lo prefiere, puede determinar la cantidad de memoria disponible el global heap mediante el uso de la función **GlobalCompact**. Esta función retorna el número de bytes del objeto de memoria contigua libre mas grande.

Debería también chequear la dirección retornada por la función **GlobalLock**. Está función retornará un puntero NULL si el handle de memoria no es válido o si el contenido del objeto de memoria ha sido descartado.

Puede liberar cualquier memoria global que ya no necesite usando la función **GlobalFree**. En general debería liberar esta memoria para que otras funciones puedan usar este espacio de memoria. Siempre debería liberar la memoria global antes que su aplicación termine.

10.2 USANDO EL LOCAL HEAP

El local heap contiene memoria libre que puede ser asignada para uso privado de la aplicación. El local heap está localizado en el segmento de datos de la aplicación y por lo tanto sólo es accesible a una instancia especifica de una aplicación. Usted puede asignar memoria del local heap en tamaño de hasta 64K y la memoria puede ser fija, movible, o descartable.

Windows no proporciona automáticamente un local heap para la aplicación. Para solicitar un local heap para su aplicación, utilice la sentencia **HEAPSIZE** en el archivo de definición de módulo de la aplicación. Esta sentencia establece el tamaño inicial, en bytes, del local heap. Si el local heap está en un segmento de datos fijo, usted podría asignar sólo hasta el tamaño del heap especificado. Si el local heap está en un segmento de datos movible, usted podría asignar más alla del tamaño inicial especificado, hasta llegar a los 64K, puesto que Windows automáticamente asignará espacio adicional para el local heap hasta que el segmento de datos se extienda hasta 64K de tamaño. Notará, sin embargo, que si Windows asigna memoria local adicional para satisfacer una asignación local, este puede mover el segmento de datos, invalidando cualquier puntero largo (long pointer) a un objeto en memoria local.

El tamaño máximo de cualquier local heap depende del tamaño del stack de la aplicación, los datos estáticos, y los datos globales. El local heap comparte el segmento de datos con el stack y los datos. Puesto que el local heap no puede ser mayor de 64K, un local heap de una aplicación no puede ser mayor de 64K menos: el tamaño del stack de la aplicación, los datos globales, y los datos estáticos. El tamaño del stack de la aplicación es definido por la sentencia

STACKSIZE en el archivo de definición de módulo. El tamaño de los datos estáticos y globales depende de cuantas cadenas, variable locales y globales son declaradas en la aplicación. Windows obliga el tamaño del stack a 5K, tal que si el archivo de definición de módulo especifica un tamaño menor del stack, Windows automáticamente establece el tamaño del stack a 5K.

Usted puede asignar memoria local usando la función **LocalAlloc**. Está función asigna un objeto de memoria el local heap de la aplicación y retorna un handle de la memoria. Bloquee el objeto de memoria local usando la función **LocalLock**. Esta retorna una dirección near (un offset de 16-bit) a el primer byte en el objeto de memoria. El offset es relativo al inicio del segmento de datos. En el siguiente ejemplo, la función **LocalAlloc** asigna 256 bytes de memoria movibles, y la función **LocalLock** los enclava de modo que los primero 256 bytes puedan ser establecidos a la dirección 0xFF:

```
HANDLE hMem;
LPSTR pMem;
int i;

if ( ( hMem = LocalAlloc( GMEM_MOVEABLE, 256 ) ) != NULL ) {
    if ( ( pMem = LocalLock( hMem ) ) != NULL ) {
            for ( i = 0; i < 256; i++ )
                    pMem[ i ] = 0xFF;
            LocalUnlock( hMem );

    }
}
```

En este ejemplo, la aplicación desbloquea el handle de memoria usando la función **LocalUnlock** inmediatamente después de accesar el objeto de memoria. Una vez que un objeto de memoria movible, o descartable es bloqueado, Windows garantiza que el objeto permanecerá fijo en memoria hasta que este sea desbloqueado. Esto significa que la dirección permanece válida tanto tiempo como el objeto permanezca bloqueado, pero esto también impide a Windows hacer un mejor uso de la memoria si otros requerimientos de asignación son realizados. Si quiere asegurarse que está consiguiendo la mejor perfomance para el local heap de su aplicación, asegúrese de desbloquear la memoria después de usarla.

La función **LocalAlloc** retorna un valor NULL si el requerimiento de asignación falla. Debería siempre chequear el valor retornado para asegurarse que un handle válido existe. Si prefiere, puede determinar la cantidad de memoria disponible en el local heap usando la función **LocalCompact**. Esta función retorna el número de bytes libres en el objeto de memoria contigua libre mas grande en el *local heap*.

Debería también chequear la dirección retornada por la función **LocalLock**. Está función retornará un puntero NULL si el handle de memoria no es válido o *si el contenido del objeto de memoria ha sido descartado.*

TRABAJANDO CON MEMORIA DESCARTABLE

Usted puede crear bloques de memoria descartable, combinando las opciones GMEM_DISCARDABLE y GMEM_MOVEABLE cuando asigne un objeto. El objeto resultante será movido como sea necesario para hacer espacio para otros

requerimientos de asignación, o si no hay suficiente memoria para satisfacer el requerimiento, el objeto podría ser descartado. El siguiente ejemplo asigna un objeto descartable desde la memoria global:

```
hMem = GlobalAlloc(GMEM_MOVEABLE | GMEM_DISCARDABLE, 4096L);
```

Cuando Windows descarta un objeto de memoria, vacía el objeto para volver a asignarlo con cero bytes como su nuevo tamaño. Los contenido del objeto son perdidos, pero el handle de memoria del objeto permanece válido. Sin embargo, cualquier intento de enclavar el handle y accesar el objeto fallará.

Windows determina cual objeto descartar usando el algoritmo del menos-recientemente-usado (LRU). Este continua descartando objetos de memoria hasta que halla suficiente espacio de memoria para satisfacer el requerimiento de asignación de memoria. En general, si no ha usado un objeto de memoria en algún momento, este es candidato para ser descartado. Un objeto bloqueado no puede ser descartado.

Usted puede descartar sus propios objetos de memoria usando la función GlobalDiscard. Esta función vacía el objeto pero preserva el handle de memoria. También puede descartar otros objetos de memoria de la aplicación usando la función GlobalCompact. Esta función mueve y descarta objetos de memoria hasta que la cantidad de memoria especificada o la mas grande posible sea disponible. Una forma para descartar todos los objetos descartables es proporcionando -1 como argumento de la función. Esto es un requerimiento para todos los bytes de memoria. Si bien los requerimiento fallarán, este descartará todos los objetos descartables y deja el objeto de memoria libre lo mas grande posible.

Puesto que un objeto de memoria descartado mantiene su handle válido, usted puede recuperar información acerca del objeto usando la función GlobalFlags. Esto es útil para verificar que el objeto ha sido descartado. GlobalFlags establece el bit GMEM_DISCARDED en su valor retornado cuando el objeto de memoria ha sido descartado. Por lo tanto, si intenta enclavar un objeto descartable y el bloqueo falla, usted puede chequear el estatus del objeto usando GlobalFlags.

Una vez que un objeto descartable ha sido descartado, su contenido es perdido. Si desea usar el objeto nuevamente, debe re-asignar este a su tamaño adecuado y llenarlo con los datos que contenía anteriormente. Puede volver a asignarlo usando la función GlobalReAlloc. El siguiente ejemplo chequea el estatus del objeto, y entonces lo llena de datos si ha sido descartado.

```
lpMem = GlobalLock(hMem);

if (lpMem == NULL ) {
    if (GlobalFlags(hMem) & GMEM_DISCARDED) {
        hMem = GlobalReAlloc(hMem, 4096L,
                GMEM_MOVEABLE | GMEM_DISCARDABLE);
        lpMem = GlobalLock(hMem);

        /* mas líneas de programa      */
        /* llenado de datos            */

        GlobalUnlock(hMem);
```

Usted puede hacer un objeto descartable nodescartable (o viceversa) usando la función GlobalReAlloc y la constante GMEM_MODIFY. El siguiente ejemplo cambia un objeto movible, identificado por el handle de memoria hMem, a un objeto movible descartable:

hMem = GlobalReAlloc(hMem, 0, GMEM_MODIFY | GMEM_DISCARDABLE, 0);

El siguiente ejemplo cambia un objeto descartable a uno no descartable:

hMem = GlobalReAlloc(hMem, 0, GMEM_MODIFY);

Cuando usted especifica GMEM_MODIFY en una llamada a la función GlobalReAlloc, el segundo parámetro es ignorado.

USANDO SEGMENTOS

Una de las principales características de Windows es que permite al usuario correr más de una aplicación a la vez. Debido a que múltiples aplicaciones emplean una mayor demanda de memoria de la que una sola aplicación hace, la habilidad de Windows para correr mas de una aplicación a la vez significativamente afecta como usted escriba aplicaciones. Si bien muchos computadores tienen al menos 640K de memoria, esta memoria rápidamente se transforma en limitada a medida que el usuario corre y carga mas aplicaciones. Con Windows usted debe asegurarse de como sus aplicaciones usan la memoria y estar preparado para minimizar la cantidad de memoria que su aplicación ocupa en algún momento dado.

Para ayudarle a administrar el uso de memoria de su aplicación, Windows usa el mismo sistema administrador de memoria para su código de aplicación y segmento de datos que usa dentro de su aplicación para asignar y administrar memoria global. Cuando usted inicia su aplicación, Windows asigna espacio para los segmentos de código y dato en memoria global, y entonces copia desde al archivo ejecutable a memoria. Estos segmentos pueden ser fijos, movibles, y aún descartables. Usted especifica sus atributos en el archivo de definición de módulo de su aplicación.

Puede reducir el efecto que sus aplicación tiene sobre la memoria usando segmento de código y dato movibles. Usando segmento movibles, usted habilita a Windows a tomar ventaja de la memoria libre a medida que la memoria se transforme disponible.

Usando segmento de código descartable, usted puede reducir el efecto posterior que su aplicación tiene sobre la memoria. Si usted hace un segmento de código descartable, Windows lo descarta, si es necesario, para satisfacer los requerimientos de asignación a memoria global. A diferencia de los objetos de memoria ordinarios que usted puede asignar, los segmentos de código descartable son monitoreados por Windows, el cual automáticamente los vuelve a cargar si su aplicación intenta ejecutar código dentro de ellos. Esto significa que el segmento de código de sus aplicación están en memoria si ellos son necesitados.

La descartación de un segmento destruye su contenido. Windows no salva el actual contenido de un segmento descartado. En su lugar, trata al segmento como si no fuera diferente a cuando fue cargado originalmente, y cargará el segmento directamente del archivo ejecutable cuando este es necesitado.

USANDO SEGMENTOS DE CODIGO

Un segmento de código es una instrucción de máquina de uno o más bytes (pero nunca mayores de 64K).

Importante No debe almacenar datos para escritura en el segmento de código; una escritura a un segmento de código causa una falla de protección general (general-protection GP) cuando su versión 3.1 de Windows está corriendo. Windows, sin embargo, permite almacenar datos de sólo lectura en el segmento de código.

Cada aplicación tiene al menos un segmento de código. También se pueden crear aplicaciones que tengan más de un segmento de código. De hecho, la mayoría de aplicaciones tienen múltiples segmentos de código. Usando múltiples segmentos de códigos , se reduce el tamaño de cualquier segmento de código al número de instrucciones necesitadas para producir alguna tarea. Si usted también hace uso de memoria descartable, minimiza efectivamente los requerimientos de memoria al segmento de código de sus aplicación.

Cuando crea aplicaciones del modelo medio o largo (medium- or large-model), usted esta creando aplicaciones que usan múltiples segmentos de código. Estas aplicaciones tiene uno o mas archivos fuente para cada segmento. Cuando trabajamos con múltiples archivos fuente, compilamos cada archivo separadamente y explícitamente nominamos el segmento al cual el código compilado pertenece.

Para definir un atributo de segmento, use la sentencia SEGMENT en el archivo de definición de módulos. El siguiente ejemplo muestra la definición de los tres segmentos:

```
SEGMENTS
        PAINT_TEXT MOVEABLE DISCARDABLE
        INIT_TEXT MOVEABLE DISCARDABLE
        WNDPROC_TEXT MOVEABLE DISCARDABLE
```

También podría usar la sentencia CODE en el archivo de definición de módulo para definir los atributos por defecto para todos los segmentos de código. La sentencia CODE también define atributos para cualquier segmento que no este explícitamente definido en la sentencia SEGMENTS. La siguiente línea de código muestra como hacer todos los segmentos, no listados en la sentencia SEGMENTS, descartables.

```
CODE MOVEABLE DISCARDABLE
```

Si usted usa segmento de código descartable en su aplicación, debe balancear los segmentos descartables con el número de veces que el segmento podría ser accesado. Por ejemplo, el segmento que contiene su procedimiento de ventana principal probablemente no debería ser descartable, debido a que Windows llama

este procedimiento a menudo. Tipicamente este procedimiento es pequeño aproximadamente 4K). Debido a que un segmento descartable tiene que ser cargado del disco cuando sea necesitado, el ahorro de memoria que podria realizar descartando el procedimiento de ventana podría ser compensado por la perdida de perfomance que viene con el frecuente acceso a disco. Para optimar la performance, deberia asegurarse que sólo lo necesario el segmento que contiene el procedimiento de ventana principal sea llamado frecuentemente por el sistema.

EL SEGMENTO DE DATOS (DATA SEGMENT)

Cada aplicación tiene un segmento de DATA. El segmento de DATA contiene la pila de la aplicación (application stack), el local heap, y los datos estáticos y globales. Como el segmento de codigo, el segmento de DATA no debe ser mayor de 64K.

Un segmento de DATA puede ser fijo o movible, pero no descartable. Si el segmento de DATA es movible, Windows automáticamente enclava el segmento sobre el control pasajero a la aplicación. De otra manera, un segmento de DATA movibles podría moverse si una aplicación asigna memoria global o si la aplicación intenta asignar mas memoria de la que actualmente esta disponible en el local heap. Por esta razón es importante no mantener punteros largos (long pointer) a variables en el segmento de DATA.

Se define los atributos en el segmento de DATA usando la sentencia DATA en el archivo de definición de módulos. Los atributo por omisión son MOVEABLES y MULTIPLES. El atributo MULTIPLE direcciona a Windows a crear una copia del segmento de datos de la aplicación para instancia de la aplicación. Esto significa que el contenido del segmento de DATA son únicos para cada instancia de la aplicación.

Una aplicación de modelo largo (large-model) podría tener segmentos de datos adicionales, pero únicamente un segmento e DATA. Usando un modelo largo con segmentos de datos adicionales no es recomendable. Si su aplicación requiere múltiples segmentos de datos, puede asignarlos usando la función GlobalAlloc durante la inicialización de la aplicación.

APLICACION DE PRUEBA : MEMORIA

La aplicación de prueba muestra como crear una aplicación Windows de modelo medio (medium-model) que usa códigos de segmentos descartables. Para crear la aplicación Memory, partimos de una aplicación genérica escrita en lenguaje C, denominada Generic.
Realice las siguiente modificaciones a la aplicación genérica:

1. Divida el archivo fuente en lenguaje C en cuatro archivos separados.
2. Modifique el header file.
3. Adicione un nuevo segmentos de definiciones al archivo de definiciones de módulos.
4. modifique el makefile.
5. Compile y enlace la aplicación.

DIVIDIENDO EL ARCHIVO FUENTE EN LENGUAJE C

Puesto que las funciones dentro de un archivos son compiladas como segmentos separados, usted debe dividir el archivo fuente en lenguaje C en archivos separados. Para esta aplicación, puede dividir el archivo fuente en cuatro partes:

MEMORY1.C — Contiene la función WinMain. Puesto que Window ejecuta el lazo de mensaje en WinMain frecuentemente, el segmento creado de este archivo fuente no es descartable. Esto previene una situación en la cual el segmento tenga que ser cargado desde disco frecuentemente. Debido a que WinMain de cualquier manera es relativamente pequeño, mantener este segmento en memoria tiene un efecto pequeño sobre la memoria global disponible.

MEMORY2.C — Contiene la función MemoryInit. Puesto que la función MemoryInit es usada únicamente cuando la aplicación arranca, es segmento creado por este archivo fuente puede ser descartable.

MEMORY3.C — Contiene la función MemoryWndProc. Si bien el segmento creado de este archivo fuente puede ser descartable, es probable que la función MemoryWndProc al menos sea llamada tan frecuente como la función WinMain reciba el control. En este caso, el segmento es movible pero no descartable.

MEMORY4.C — Contiene la función About. Puesto que la función About es ocasionalmente llamada (sólo cuando el About dialog box es mostrado), el segmento de código creado desde este archivo fuente puede ser descartable.

Debe incluir los header files WINDOWS.H y MEMORY.H en cada archivo fuente.

MODIFICANDO EL HEADER FILE

Debe mover las declaraciones de la variable hInst dentro del archivo de cabecera MEMORY.H. Esto asegura que la variable es accesible a todos los segmentos. La variable hInst es usada en las funciones WinMain y MemoryWndProc.

ADICIONANDO UN NUEVO SEGMENTO DE DEFINICIONES

Para especificar los atributos de cada segmento de código, debe adicionar segmento de definiciones al archivo de definición de módulos. Esto significa que debe adicionar la sentencia SEGMENTS al archivo y listar cada segmento junto al nombre en la aplicación. Después que ha hecho estos cambios, el archivo de definición de módulos debe mostrarse de la siguiente manera:

```
NAME            Memory
DESCRIPTION 'Aplicación de prueba en Microsoft Windows 3.1'
```

```
EXETYPE WINDOWS

STUB            'WINSTUB.EXE'

SEGMENTS
    MEMORY_MAIN     PRELOAD     MOVEABLE
    MEMORY_INIT     LOADONCALL MOVEABLE DISCARDABLE
    MEMORY_WNDPROC  LOADONCALL MOVEABLE
    MEMORY_ABOUT    LOADONCALL MOVEABLE DISCARDABLE

CODE    MOVEABLE DISCARDABLE

DATA    MOVEABLE MULTIPLE

HEAPSIZE        1024
STACKSIZE       8192

EXPORT
    MainWndProc @1
    About       @2
```

En este archivo de definición de módulo, la sentencia SEGMENTS define los atributos de cada segmento:

- El segmento MEMORY_MAIN contiene WinMain
- El segmento MEMORY_INIT contiene funciones de inicialización
- El segmento MEMORY_WNDPROC contiene el procedimiento de ventana
- El segmento MEMORY_ABOUT contiene el procedimiento del dialog box

Cada uno de los segmentos tiene el atributo MOVEABLE, pero sólo MEMORY_INIT y MEMORY_ABOUT tiene el atributo DESCARDABLE. También sólo el segmento MEMORY_MAIN es cargado cuando la aplicación se inicia. Los restantes segmentos tienen el atributo LOADONCALL, lo cual significa que son cargados cuando son necesitados.

Si bien cada segmento de la aplicación esta explícitamente definido, la sentencia CODE no obstante es especificada. Esta sentencia especifica el atributo de cualquier segmento adicional que el enlazador (linker) podría adicionar a la aplicación, por ejemplo, cualquier sentencia que contenga llamadas a funciones run-time del C en los archivos fuente de la aplicación.

MODIFICANDO EL MAKEFILE

Para compilar los nuevos archivos fuente de manera separada, debe referirse a cada archivo fuente en el makefile. Puesto que la aplicación es de modelo medio, use la opción /AM cuando compile. Para clarificar, usted debería nombrar a cada segmento usando la opción /NT.

También necesitará cambiar la línea de comando del link tal que esta refiere a la librería de modelo medio MLIBCEW.LIB en lugar de la librería de modelo pequeño (small-model) SLIBCEW.LIB.

El makefile para la aplicación Memory debería mostrarse así:

```
memory.res: memory.rc memory.h
    rc /r memory.rc

memory1.obj: memory1.c memory.h
    cl /c /AM /Gsw /Zp /NT MEMORY_MAIN memory1.c

memory2.obj: memory2.c memory.h
    cl /c /AM /Gsw /Zp /NT MEMORY_INIT memory2.c

memory3.obj: memory3.c memory.h
    cl /c /AM /Gsw /Zp /NT MEMORY_WNDPROC memory3.c

memory4.obj: memory4.c memory.h
    cl /c /AM /Gsw /Zp /NT MEMORY_ABOUT memory4.c

memory.exe: memory1.obj memory2.obj memory3.obj memory4.obj \
    memory.def
    link memory1 memory2 memory3 memory4, memory.exe,,mlibcew \
    libw, memory.def
    rc memory.res

memory.exe: memory.res
    rc memory.res
```

COMPILANDO Y ENLAZANDO

Después de compilar y enlazar la aplicación Memory, inicie la sesión Windows. Microsoft Windows Heap Walker (HEAPWALK.EXE), proporcionado con el SDK, y Memoria. Use Heap Walker para ver los segmentos varios de la aplicación Memory.

CAPITULO SEIS

ACERCA DE MANEJO DE MEMORIA

El capítulo anterior proporciona la información básica necesaria para conocer acerca del uso de memoria en aplicaciones en Microsoft Windows. Sin embargo, algunas aplicaciones requieren técnicas más avanzada de manejo de memoria. En el presente capítulo se trata de proveer información más detallada acerca de como Microsoft Windows maneja la memoria y como debería escribir sus aplicaciones para hacer un mejor uso de las características avanzadas de memoria del Windows.

CONFIGURACIÓN DE MEMORIA

Usted debería esperar que su aplicación corra en cualquiera de las dos configuraciones de memoria; más frecuentemente, que la aplicación dependa del tipo de CPU y la cantidad y configuración de memoria. Windows soporta las dos configuraciones de memoria.

- standard mode
- 386 enhanced mode

Si el usuario inicia otro programa antes de empezar Windows, la cantidad de memoria disponible para Windows será menor que la instalada en el sistema.

Debido a que Windows usa diferentes configuraciones de memoria en diferentes sistemas, su aplicación debería ser capaz de ejecutarse con cualquiera de los dos tipos de configuraciones de memoria. La mejor manera de asegurar esto es escribiendo la aplicación siguiendo todas las reglas de manejo de memoria en Windows. Estas reglas son descritas en la sección 5.5.

Donde sea posible, su aplicación no debería contener código que dependa de una configuración de memoria en particular. En algunas instancias, sin embargo, la aplicación debe de ser capaz de determinar la configuración en la que se encuentra corriendo. Para hacer esto la aplicación puede llamar la función GetWinFlags. Esta función retorna un valor de 32-bit que contiene los flags que indica la configuración de memoria en la cual Windows está corriendo y otras informaciones acerca del sistema de usuario.

STANDARD-MODE (EL MODO ESTÁNDAR)

Windows utiliza por defecto la configuración de memoria en el standard-mode para sistemas que cumplen con los siguientes criterios:

- Un sistema basado en un 80286 con al menos 1M de memoria
- Un sistema basado en un 80386 con al menos 1M pero menos de 2M de memoria. En sistemas basados en un 80386 con 2M o más, Windows por defecto usa la configuración de memoria 386 enhanced-mode.

El Windows heap está hecho de al menos 2 objetos de memoria, uno en memoria convencional (MS DOS) y uno en memoria extendida. Objetos de memoria adicional en memoria convencional o extendida pueden estar presentes.

El objeto de memoria que Windows usa para el global heap está en memoria convencional. Esta inicia sobre cualquier programa residente TSR (terminate-and-stay-resident), dispositivos controladores (device driver), MS-DOS, y demás, y se extiende hasta el tope de la memoria convencional. Esta memoria convencional usualmente es de 640kb, pero pueden ser menos en algunos sistemas.

El segundo objeto de memoria requerido para la configuración en Windows standard-mode está en memoria extendida. Windows asigna el objeto en memoria extendida a través del dispositivo controlador de la memoria extendida y entonces tiene acceso en el objeto directamente, sin usar el dispositivo. El tamaño y localización de este dispositivo puede variar, dependiendo de lo que el usuario cargó en memoria extendida antes de iniciar Windows.

Windows enlaza dos o más objetos de memoria para formar el Windows global heap. El inicio(fondo) del objeto de memoria convencional es el inicio (fondo) del global heap, y el final (tope) del objeto de memoria extendida es el final (tope) del global heap.

El siguiente cuadro muestra una configuración de memoria típica en Windows standard-mode

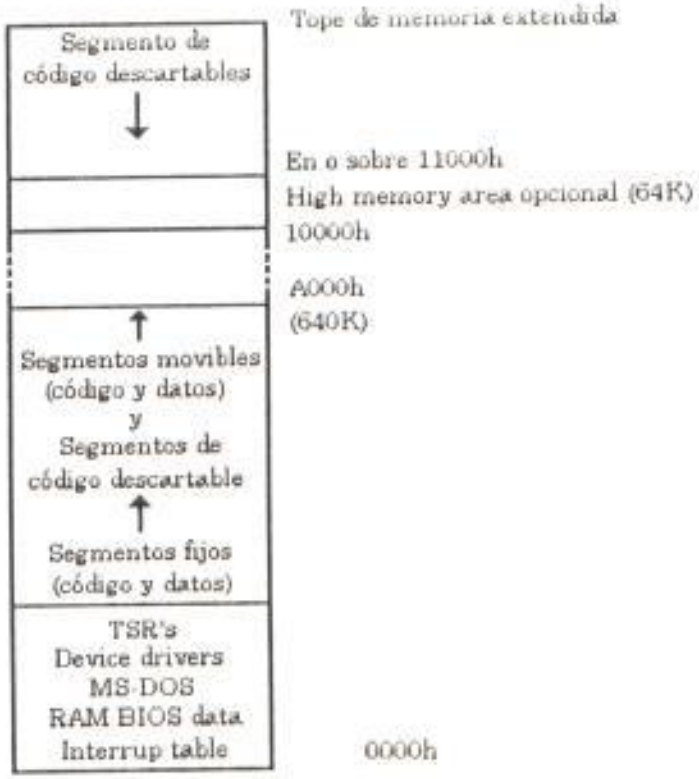

Figura 5-1. Configuración de memoria en el Windows standard-mode

Como sucede con otras configuraciones de memoria, Windows asigna segmentos de código descartable desde el tope del heap, segmentos fijo desde el fondo del heap, y código movible y segmentos de datos sobre los segmentos fijos.

USANDO OBJETOS DE MEMORIA ENORMES (HUGE MEMORY OBJECTS) EN EL MODO ESTÁNDAR

Una dirección far (lejana) está creada de segmentos de direcciones de 16-bit y un offset de 16-bit. La dirección de un segmento es un selector, similar a un Windows handle, que apunta a una entrada en una tabla descriptiva local o global (LDT o GDT). La entrada de la tabla indica si el segmento referido a el selector actualmente reside en memoria. Si el segmento reside en memoria, la entrada a la tabla provee la dirección lineal del segmento.

Si usted asigna un objeto de memoria enorme (más grande que 64K), el compilador de Microsoft C (CL) genera código de huge-pointer que realiza segmentación aritmética para tomar ventaja de un far pointer a través de límites de segmentos de 64K. Sin embargo CL hace esto sólo si el objeto explícitamente está declarado como huge o si el módulo fue compilado con el modelo de memoria sumamente grande (huge memory model).

No cambie directamente la porción de direcciones de segmentos de un far pointer. Intentar incrementar una dirección de segmentos con el intento de avanzar párrafos físicamente sólo resultará en un selector no válido. Cuando el selector no válido es subsecuentemente usado para leer o escribir a una localidad de memoria, Windows reportará una falla de protección general (general-protection GP), o peor aún, el selector no válido puede inadecuadamente apuntar a un dato o código inapropiado.

Si está programando en lenguaje ensamblador, la técnica adecuada para incrementar un far pointer es usando la variable externa __ahincr. Windows fija __ahincr con la constante correcta para incrementar el selector de segmento. Esto es posible debido a que cuando Windows asigna un objeto de memoria enorme (huge memory object), asigna valores de selectores relacionados a los segmentos de memoria relacionados que son de 64K en tamaño.

El siguiente ejemplo ilustra el método adecuado para incrementar un far pointer por 64K (el único incremento provisto):

```
extrn   __ahincr:abs
        .
        .
        .
mov     ax, es          ; es su dirección de segmento que desea incrementar
addax, __ahincr
mov     es, ax
```

El objeto de memoria más grande que Windows puede proporcionar en un procesador 80286 es de 1 megabyte menos 16 bytes. El objeto de memoria más grande en un procesador 80386 es de 16 megabyte menos 64K.

USANDO SELECTORES GLOBALES

Para realizar un mapeo de memoria de entrada y salida, puede usar los siguientes selectores globales en una aplicación en lenguaje ensamblador para acesar la localidad de memoria correspondiente:

*__A000H
*__B000H
*__B800H
*__C000H
*__D000H
*__E000H
*__F000H

El siguiente ejemplo ilustra como usar estos selectores adecuadamente:

```
mov     ax, __A000H
mov     es, ax
```

No use estos selectores excepto para soportar dispositivos de hardware que realicen mapeo de memoria de entrada y salida.

USANDO APELATIVOS (ALIAS) A LOS SEGMENTOS DE CÓDIGO Y DATOS

Usualmente, usted no puede ejecutar código almacenado en el segmento de datos. En el standard-mode, intentar ejecutar código en el segmento de datos resulta en una falla GP. Rara vez, sin embargo, tales ejecuciones podrían ser necesarias, y pueden ser realizadas dándole un alias al segmento de datos en cuestión. Dar un alias (aliasing) involucra copiar un segmento selector y entonces cambiar el campo TYPE de la copia de manera que una operación que normalmente no es permitida pueda ser realizada sobre el segmento.

Windows provee dos funciones que realizan el aliasing de segmentos:

*AllocDStoCSAlias

*ChangeSelector

AllocDStoCSAlias acepta un selector del data-segment y retorna un selector del code-segment. Esto le permite escribir instrucciones de máquina en su pila de datos (data stack), crea un alias de la pila de segmentos (segment stack), y entonces ejecuta el código en el stack.

Esta función asigna un nuevo selector, después de llamar AllocDStoCSAlias, debe llamar a la función FreeSelector cuando el selector no sea más necesitado.

Debe ser cuidadoso de no usar el selector retornado por AllocDStoCSAlias si es posible que el segmento ha sido movido. La única manera de prevenir que un segmento sea movido es llamando la función GlobalFix para fijar en un espacio de dirección lineal después de dar alias a un segmento.

Debe asegurarse también que un segmento no sea movido si su aplicación no accede a otra tarea y no toma acción alguna que pueda resultar en memoria

asignada. Tipicamente, esto requeriría que usted asigne y libere un nuevo selector cada momento que su aplicación cede o asigne memoria.

Para prevenir la asignación y liberación de memoria de manera frecuente, puede usar un selector temporal. ChangeSelector provee un método conveniente para dar 'aliasing' a un selector temporal (generando un selector de código correspondiente a un selector de dato dado, o viceversa). Esta función acepta dos selectores: un selector temporal, y el selector que usted quiera convertir. Para convertir el selector repetidamente, podría realizar los siguientes pasos:

1. Llamar AllocateSelector para crear un selector temporal
2. Llame ChangeSelector, tan frecuente como sea necesario, pasando el selector temporal y el selector que desea convertir. Debido a que ChangeSelector usa un selector previamente asignado, no necesita liberar el selector cada momento que convierte este. En su lugar, usted llama ChangeSelector cada momento que necesite convertirlo después de convertir el segmento que hubiese movido
3. Cuando el selector convertido no sea más necesitado, llame FreeSelector para liberar el selector temporal.

386 ENHANCED MODE

Si el sistema de usuario tiene al menos 2 megabytes de memoria extendida disponible y un microprocesador 80386, entonces Windows y las aplicaciones Windows correrán en el 386 enhanced mode (modo extendido 386). En este modo, al tomar ventaja de ciertas características del procesador 80386, Windows implementa un esquema administrador de memoria virtual usando disk swapping (intercambio de segmentos a disco).

El resultado de este esquema es que la cantidad de memoria disponible para todas las aplicaciones puede ser varias veces la cantidad de memoria extendida en el sistema. En este modo, Windows teóricamente puede direccionar 4 gigabytes de memoria, pero está limitado por la cantidad de RAM y espacio de disco disponible para el swapping.

NOTA: Debido a que el 386 enhanced mode usa las características del modo protegido de un procesador 80386, la restricción de usar memoria en el standard-mode también se aplican al usar memoria en el 386 enhanced mode.

A continuación se describe la configuración de memoria en el 386 enhanced mode:

- El global heap esencialmente es un gran espacio de direcciones virtuales compartidas por todas las aplicaciones
- El tamaño del espacio de direcciones virtuales del global heap no está limitado por la cantidad de memoria extendida. El disco sirve como un medio secundario que extiende el espacio de direcciones virtuales.

INTERCAMBIO DE SEGMENTOS DE CÓDIGO Y DATOS

Los segmentos de código y datos fijos en el 386 enhanced mode están localizados en memoria más baja que los segmentos de código y datos movibles no

descartables, y los segmentos de datos descartables, los cuales están localizados sobre los segmentos de datos y códigos fijos. Los segmentos de código descartables son asignados del tope de memoria.

La configuración de memoria en el 386 enhanced mode es distinta del standard-mode, debido a que Windows intercambia segmentos de código y datos entre memoria física y el disco. En el standard-mode, Windows puede remover datos descartables de memoria, pero no salva el dato a disco de manera que pueda ser vuelto a leer en memoria cuando sea necesitado.

En el 386 enhanced mode, Windows continua asignando memoria física hasta que esta en usada, entonces inicia un swapping de páginas de código y datos de 4K desde memoria física a disco con el fin de hacer memoria física adicional disponible. Windows intercambia objetos (páginas) de 4K, en lugar de segmentos de código y datos de diferente tamaño. Un objeto de 4K intercambiado puede ser sólo una parte de un segmento de datos o código, o este puede cruzar sobre dos o más segmentos de código o dato.

Este intercambio de memoria, o paginamiento, es transparente a las aplicaciones. Si la aplicación intenta accesar un segmento de código o dato, del cual alguna parte ha sido paginada a disco, el microprocesador 80386 emite una interrupción, denominada falla de página, a Windows. Windows entonces intercambia otras páginas de memoria a disco, y restaura las páginas que la aplicación necesita. Windows selecciona las páginas que intercambia a disco basándose en el algoritmo del menos reciente usado (LRU).

Este sistema de memoria virtual provee tanta memoria adicional como el tamaño del swap file (archivo de intercambio) que Windows reserva en el disco de usuario. Windows determina el tamaño del swap file basándose en la cantidad total de memoria física en el sistema y la cantidad de espacio de disco disponible. El usuario puede modificar el tamaño de un swap file cambiando la entrada en el archivo SYSTEM.INI, y puede establecer un swap file permanente usando el comando swapfile.

La demanda-carga de Windows de segmentos de código y dato opera en el tope del esquema de paginamiento de memoria virtual de Windows. Esto es, Windows trata la memoria virtual como si esta fuera memoria convencional para propósitos de determinar cual segmento de código y dato descartar. Windows, sin embargo, remueve segmento de código y datos únicamente cuando la memoria virtual está agotada.

PRESERVANDO MEMORIA QUE ESTA SIENDO PAGINADA A DISCO

Ocasionalmente, es necesario asegurar que cierta memoria este siempre presente en memoria física y que nunca sea paginada a disco. Por ejemplo una función DLL (dinamic-link library) puede ser requerida para responder inmediatamente a una interrupción en lugar de esperar que el sistema genere una falla de página y coloca los datos desde el disco. En tales casos un objeto de memoria puede ser page-locked (bloqueado a paginamiento) para prevenir que este sea paginado a disco.

Para bloquear a paginamiento un objeto de memoria, llame a la función GlobalPageLock, pasándole el selector global del segmento que está para ser

bloqueado. Esta función incrementa (incrementa en uno) un contador de page-lock para el segmento; mientras que el contador para un segmento sea diferente de cero, el segmento se mantendrá en la misma dirección física y no será paginado a disco. Cuando ya no requiera que la memoria este bloqueada, llame a la función GlobalPageUnlock para restar (decremento en uno) el contador page-lock. En el standard-mode esta función no tiene efecto.

NOTA: Usted debería bloquear a paginamiento de memoria únicamente en situaciones críticas. Por ejemplo, habitualmente no bloquee memoria a paginamiento para bloquear la caida de una hoja electrónica (caida del sistema). Bloquear memoria a pagtnamiento afecta inversamente la performance de todas las aplicaciones, incluyendo la suya.

ALMACENAMIENTO DE DATOS

Windows soporta siete tipos de almacenamiento de datos, cada uno es apropiado para diferentes situaciones. La siguiente lista define cada tipo de almacenamiento, y sugiere como decidir cual tipo usar.

Dato estático	Incluyen todas las variables del lenguaje C que el código fuente de la aplicación, implícitamente o explícitamente, declara usando la palabra clave static. Los datos estáticos también incluyen todas la variables del lenguaje C declaradas como externas, ya sea explícitamente (usando la palabra clave extern) o por default (declarando estas fuera de todas las funciones).
Dato automático	Incluyen todas las variables que están declaradas en el stack en el momento que una función es llamada. Las variables incluyen los parámetros de funciones y cualquier variable declarada localmente.
Datos locales dinámicos	Incluyen todos los datos que son asignados usando la función LocalAlloc. El dato local dinámico es asignado fuera del local heap en el segmento de dato automático para el cual un registro DS de la aplicación es establecido. La asignación de un objeto de memoria desde el local heap de una aplicación Windows es similar a la asignación de memoria usando la función malloc de la librería run-time del C en una aplicación no Windows que usa modelos pequeño o mediano (small- or medium-model).
Dato global dinámico	Incluyen todos los datos que son asignados fuera de global heap de Windows usando la función GlobalAlloc. El global heap es un recurso de memoria del sistema (system-wide memory resource). La asignación de objetos de memoria desde el global heap es aproximadamente equivalente a usar la función malloc en una aplicación no Windows que usa modelos de memoria compactos o grandes (compact- or large-

model). La diferencia es que en Windows, su aplicación asigna objetos de memoria fuera de un heap potencialmente compartido por otras aplicaciones, mientras que una aplicación no Windows tiene esencialmente el heap único (whole heap) para si mismo.

Windows extra bytes — Especifica los bytes extra que son asignados en la estructura de datos que Windows mantiene internamente para una ventana creada para su aplicación. Para crear este genero de ventana, registre una clase para la misma (llamando a la función RegisterClase) y solicite que bytes extras sean asignados para cada ventana que es un miembro de esta clase. Solicite bytes extras especificando un valor diferente de cero para el miembro cbWndExtra de la estructura WNDCLASS que usted pasa a RegisterClass. También puede almacenar y recuperarlos datos desde esta área haciendo llamadas a las funciones SetWindowWord, SetWindowLong, GetWindowWord, GetWindowLong.

Class extra bytes — Especifica bytes extras que son asignados al final de la estructura WNDCLASS creada por una window class (clase de Windows). Cuando registra una window class, especifique un valor diferente de cero para el miembro cbClsExtra. Entonces puede almacenar y recuperar datos en esta área haciendo llamadas a las funciones SetClassWord, SetClassLong, GetClassWord, GetClassLong.

Recursos — Especifica una colección no modificables de datos almacenados en la porción de recursos de un archivo ejecutable. Este dato puede ser cargado en la memoria donde sus aplicación pueda usarla convenientemente. Puede definir recursos privados área cualquier especie de dato de sólo lectura que desee almacenar. Usted compila un recurso dentro de su archivo ejecutable (.ECCE) o .DLL usando el compilador de recursos de Microsoft Windows (SRC). En tiempo de ejecución, puede entonces accesar los recursos de datos varias funciones de las librerías de Windows.

5.2.1. MANEJANDO SEGMENTOS DE DATOS AUTOMÁTICOS

Cada aplicación tiene un segmento de datos denominado segmento de dato automático, el cual puede contener hasta 64K. El segmento de dato automático contiene los siguientes géneros de datos:

Task header — Contiene 16 bytes de información que Windows mantiene para cada aplicación. El task header está siempre localizado en los primeros 16 bytes del segmento de dato automático.

Static data — Incluye todas las variables en lenguaje C que son declaradas como static o extern, ya sea explícitamente o por default.

Stack — Almacena dato automático. El stack tiene un tamaño fijo, pero el área activa dentro del stack crece y se contrae a medida de como la función se ejecuta y retorna. Cada momento que una función es llamada, la dirección retornada es introducida (pushed) dentro de la porción activa del stack, junto con los valores de parámetros pasados a la función.

Local heap — Contiene todos los datos locales dinámicos.

La siguiente figura muestra la disposición del segmento de dato automático de una aplicación:

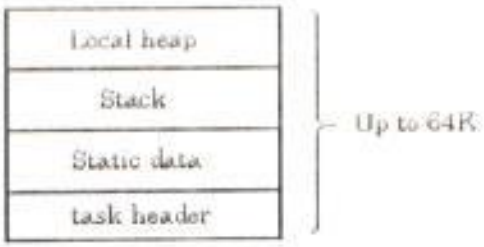

Figura 5-2. Layout del segmento de datos automáticos

El tamaño del stack es siempre fijo para una aplicación dada. Usted especifica el tamaño, en bytes, del stack usando la sentencia STACKSIZE en su archivo de definición de módulos (.DEF). Windows establece un tamaño de stack mínimo de 5K. Debería experimentar con su aplicación para determinar el tamaño optimo del stack, aunque mantenga en mente que el resultado de un desbordamiento del stack es incierto.

El tamaño del local heap es establecido a un valor inicial para la aplicación acorde a la sentencia HEAPSIZE en su archivo .DEF. El local heap crecerá como lo necesario cuando llame a la función LocalAlloc. Para aplicaciones, el tamaño inicial del local heap debe ser lo suficientemente grande para mantener las variables de ambiente actuales; un tamaño de heap de 1K es recomendable.
Si su aplicación requiere memoria del local heap más allá de la que está disponible, el heap puede crecer hasta que el segmento total de datos alcance los 64K. Si algunos objetos del local heap son liberados, sin embargo, el local heap automáticamente no se contrae. Usted puede recobrar esta área llamando la función LocalShrink. Esta función primero compacta el local heap, y entonces trunca el segmento de datos automático a un número especificado de bytes.

Usted puede declarar el segmento de datos automático para ser fijo o movible en el archivo .DEF de la aplicación, lo mismo que cualquier segmento de dato o código. A menos de que tenga una buena razón para hacerlo, declare siempre el segmento de dato como movible y múltiple. El segmento de dato automático

siempre es inicialmente cargado. La siguiente línea de ejemplo muestra como declarar el segmento de dato en el archivo .DEF.

DATA MOVEABLE MULTIPLE

Al declarar el segmento de dato de la aplicación como movible, permita que Windows vuelva a localizar el segmento de datos en memoria así como su cambios de tamaño. Si el segmento de datos automático es fijo, Windows incrementa el tamaño del local heap únicamente si la memoria adyacente resulta disponible. Consecuentemente, si declara el segmento de datos para ser fijo, debería ser cuidadoso de especificar un valor inicial de HEAPSIZE adecuado en el archivo .DEF.

Debería especificar un atributo MULTIPLE para DATA para proveer un segmento de datos automático separado para cada instancia de su aplicación. Sólo las librerías .DLL pueden ser declaradas con el atributo SINGLE para DATA. De hecho, las librerías .DLL deben de ser declaradas de esta manera, puesto que ellas sólo pueden tener una instancia cada una.

6.2.2 MANEJANDO OBJETOS DE DATOS DINÁMICOS LOCALES

En Windows un local heap puede ser establecido en cualquier segmento de datos. El segmento de datos automático de la aplicación, sin embargo, es el sitio más común donde el local heap es usado.

La función LocalInit establece un área especifica dentro de cualquier segmento de datos como un local heap. El llamado a LocalAlloc y otras funciones de memoria local opera sobre el segmento de datos actualmente referenciado por el registro DS. Tantas veces como este segmento de datos sea inicializado por LocalInit, las funciones de memoria local trabajaran.

Si está desarrollando un librería .DLL que requiera un local heap, debería llamar a LocalInit durante la inicialización de la librería. Si está desarrollando una aplicación de Windows, como oposición a la librería .DLL, no debería llamar a LocalInit para el segmento de datos automático de la aplicación.

Basado en la localización de otros datos en el segmento de dato automático (el task header, static data, y stack) y el tamaño del heap especificado en el archivo .DEF, Windows por si mismo llama a LocalInit con los valores correctos para la localización y tamaño para el local heap.

La organización del local heap es similar a la del global heap:

*Los objetos fijos están localizados en el fondo del local heap

*Los objetos no descartables movibles son asignados sobre los segmentos fijos

*Los objetos descartables son asignados desde el tope del local heap

Como Windows adiciona nuevos objetos al local heap de la aplicación, objetos movibles pueden ser movidos a medida que Windows compacte el heap. De igual manera, Windows puede descartar algún objeto para hacer espacio para algunos

nuevos objetos. Windows nunca moverá objetos fijo cuando ellos son asignados en un local heap.

La siguiente figura ilustra la organización del local heap:

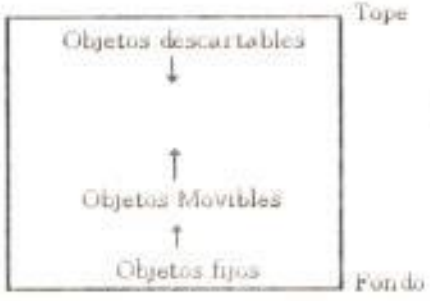

Figura 5-3. Organización del local heap

ASIGNANDO MEMORIA EN UN LOCAL HEAP

Cuando la función LocalAlloc, usted puede permitir un objeto de tamaño específico en un local heap y puede especificar ciertas características del objeto. La más importante característica es si el objeto es movible o descartable, y si este es movible, si este es descartable.

Cuando asigna un objeto en un local heap, otros objetos pueden ser movidos o descartados. En ciertos casos, usted puede no desear que el local heap sea reorganizado a medida que se adicionen nuevos objetos. Puede desear garantizar que punteros previamente establecidos a objetos movibles permanezcan alterados.

Para garantizar que ningún objeto sea descartado desde el local heap cuando llame LocalAlloc, establezca la bandera (flag) LMEM_NODISCARD en el parámetro wFlags. Para garantizar que ningún objeto en el local heap sea movido o descartado, especifique la bandera LMEM_NOCOMPACT.
LocalAlloc retorna un handle al objeto de memoria local asignado. Si no ha memoria disponible en el local heap, LocalAlloc retornará un valor NULL.

BLOQUEANDO Y DESBLOQUEANDO OBJETOS DE MEMORIA LOCAL

Para muchos programadores de C habituados al uso de funciones de la librería run-time malloc, el uso de handles de memoria al inicio le parecerá extraño. Debido a que objetos asignados en el local heap puede moverse por todos lados de memoria a medida que nuevos objetos sean adicionados, no puede esperar que un puntero a un objeto asignado permanezca siempre válido. El propósito de un handle de memoria local es proveer una referencia constante a un objeto movible.

Puesto que un handle de memoria es una referencia indirecta, usted debe omitir la referencia de el handle para obtener la dirección near del objeto local. Para hacer esto, primero llame a la función LocalLock. Esta función temporalmente

fija el objeto en una localidad constante en el local heap. Esto significa que la dirección near retornada por LocalLock permanecerá válida hasta que usted subsecuentemente llame LocalUnlock. Las siguientes líneas de ejemplo muestran como usar LocalLock para omitir la referencia de el handle de un objeto movible.

```
HLOCAL hLocalObject;
char NEAR * pcLocalObject;

/* Near no necesariamente está en modelo small o medium. */

if ( hLocalObject = LocalAlloc( LMEM_MOVEABLE, 32 )) {
    if ( pcLocalObject = LocalLock( hLocalObject )) {

            /*
             * use pcLocalObject como la dirección NEAR del objeto
             * asignado localmente
             */

            .
            .
            .

            LocalUnlock( hLocalObject );
    }

    else {

            /* el lock falló. Proceda adecuadamente */

    }
}

else {

    /* los 32 bytes no pueden ser asignados. Proceda de forma adecuada */

}
```

Si asigna un objeto de memoria local y especifica el atributo LMEM_FIXED, el objeto estará garantizado que no se moverá en memoria. Consecuentemente, no necesitará llamar LocalLock para bloquear el objeto temporalmente como una dirección fija. También, no necesitará omitir la referencia de el handle, como normalmente lo haría usando LocalUnlock, debido a que el handle de 16-bit es simplemente una dirección near de 16-bit de un objeto de memoria local. Las siguientes líneas de ejemplo muestra esto:

```
char NEAR * pcLocalObject;

/* Near no necesariamente está en modelo small o medium. */

if ( pcLocalObject = LocalAlloc( LMEM_FIXED, 32 )) {

    /*
     * use pcLocalObject como la dirección NEAR del objeto asignado
     * localmente. Este no necesariamente bloquea y desbloquea
     * el objeto local fijo.
     */
```

```
    .
    .
}
else {
    /* los 32 bytes no pueden ser asignados. Proceda adecuadamente*/
}
```

Debería prevenir dejar un objeto movible bloqueado si su aplicación necesita asignar otros objetos en el local heap. De otra manera, el administrador de memoria en Windows perderá eficiencia, puesto que Windows tiene que trabajar alrededor de objetos bloqueados mientras que intenta hacer espacio para otros objetos en el área movible del local heap.

CAMBIANDO OBJETOS DE MEMORIA LOCAL

Llame a la función LocalReAlloc para cambiar el tamaño de un objeto de memoria local, pero preservando su contenido. Si usted especifica un tamaño más pequeño, Windows truncará el objeto. Si especifica un tamaño más grande, Windows llenará la nueva área del objeto con ceros si especifica LMEM_ZEROINIT; de otra manera, el contenido de la nueva área es indefinida. Llamar LocalReAlloc puede causar que objetos en el local heap sean descartados o movidos, tal como cuando llama a la función LocalAlloc. Para prevenir que Windows descarte objetos, especifique LMEM_NODISCARD; para prevenir que Windows mueva objetos, especifique LMEM_NOCOMPACT.

También puede llamar a LocalReAlloc para cambiar el atributo del objeto desde LMEM_MOVEABLE a LMEM_DISCARDABLE o viceversa. Para hacer esto primero debe especificar LMEM_MODIFY, como se muestra a continuación:

```
hLocalObject = LocalAlloc( LMEM_MOVEABLE, 32 );
    .
    .
    .

hLocalObject = LocalReAlloc( hLocalObject, 32,
            LMEM_MODIFY | LMEM_DISCARDABLE );
```

Usted no puede usar LMEM_MODIFY con LocalReAlloc para cambiar el atributo de un objeto de memoria local hacia o desde LMEM_FIXED.

LIBERANDO Y DESCARTANDO OBJETOS DE MEMORIA LOCALES

La función de Windows LocalDiscard y LocalFree descartan y liberan objetos, respectivamente.

Hay una diferencia entre liberar un objeto local y descartar el mismo. Cuando usted descarta un objeto local, su contenido es removido desde el local heap, pero su handle permanece válido. Cuando libera un objeto local, no sólo su contenido es removido desde el local heap, sino que su handle es removido desde la tabla de handle de memoria local válidos. Un objeto de memoria puede ser descartado o liberado sólo si no hay condición externa de bloqueo sobre el.

Usted puede querer descartar un objeto en lugar de liberarlo, si desea volver a usar su handle. Para volver a usar un handle, llame la función LocalReAlloc, especificando el handle y un valor diferente de cero. Al volver a usar el handle de esta manera, usted salva el tiempo requerido para que Windows libere un handle antiguo y cree uno nuevo. Volver a usar un handle también le permite determinar cuanta memoria local esta disponible antes de intentar asignar un objeto de memoria local.

RECUPERANDO INFORMACIÓN ACERCA DEL OBJETO DE MEMORIA LOCAL

Las funciones LocalSize y LocalFlags le provee información relacionada a objetos de memoria local. LocalSize retorna el tamaño del objeto. LocalFlags indica si el objeto de memoria es descartable, y si es así, si este ha sido descartado. LocalFlags también reporta la cuenta de bloqueo del objeto de memoria.

6.2.3. MANEJANDO OBJETOS DE MEMORIA GLOBAL

El global heap en el sistema de Windows es un recurso de memoria que es comparado entre aplicaciones. Una aplicación puede requerir que Windows asigne objetos de memoria fuera del global heap llamando a GlobalAlloc, la misma función que Windows llama para asignar internamente objetos de memoria usados. Al usar las funciones de memoria global descritas en esta sección, usted puede tomar ventaja de los mismos mecanismos de administración de memoria que Windows usa para sus propios propósitos. En suma, al usar estas funciones, su aplicación puede competir o cooperar con el sistema mismo, esencialmente con los mismos privilegios. El mal uso de estos privilegios reduce la habilidad de que aplicación coopere con Windows y con otras aplicaciones.

Las siguientes consideraciones pueden ayudarle a determinar si asignar memoria para un objeto de dato dado fuera del global heap o del local heap.

* Usted puede direccionar un objeto de memoria asignado desde el local heap al usar un puntero near (después de que omita la referencia de el handle al usar LocalLock). En la otra manera, debería direccionar un objeto de memoria asignado desde el global heap usando un puntero far (después de que usted omita la referencia de el handle al usar la función GlobalLock)

* Un local heap de una aplicación es un recurso de memoria relativamente raro, puesto que este debe ajustarse en el segmento de datos automático de la aplicación (limitado a 64K bytes) junto con el stack y los datos estáticos; el global heap es mucho más grande.

Si un objeto de memoria está en la tarea actualmente establecida de su aplicación, debería intentar diseñar este como un objeto local para tomar ventaja del direccionamiento near más eficiente. La tarea actualmente establecida es el dato que debe accesar frecuentemente durante una operación bastante extensa. Los objetos que son accesados de una manera menos frecuentemente pertenecen al global heap. Para algunas aplicaciones, traería sentido transferir dato entre el local heap y el global heap de la aplicación en la tarea actualmente establecida.

Cuando diseña la estructura de un objeto de memoria global, a menudo tiene la elección de dividirlo en objetos elementales o consolidarlos en grandes objetos. Al hacer esta elección, debería considerar lo siguiente:

* Cada objeto de memoria global produce un overhead de al menos 20 bytes.

* Los objetos de memoria global son alineados en regiones de 32-byte. Los primeros 16 bytes son reservados para ciertos overhead de información. Para la configuración de memoria tanto en el standard-mode como en el 386 enhanced mode, hay un límite de un sistema extenso de 8192 handles de memoria global, sólo algunos de los cuales son disponibles para cualquier aplicación dada.

En general, debería prevenir la asignación de pequeños objetos de memoria global. Un objeto pequeño (128 bytes o menos) al menos un espacio de overhead del 15 %, más la memoria que es gastada si el tamaño del objeto no es un múltiplo de 32 bytes.

El overhead puede ser justificable en algunos casos, pero usted debería pesar cuidadosamente el overhead involucrado. Debería prevenir la asignación de un gran número (algunos cientos) de pequeños objetos globales si ellos pueden ser consolidados en unos pocos, objetos globales grandes. Esta consolidación no sólo elimina el overhead del espacio sino que también previene el uso innecesario del limitado número de handles de objetos globales.

Con estas consideraciones en mente, como maneje objetos en el global heap es similar a como maneje objetos de memoria en el local heap.

ASIGNANDO MEMORIA EN EL GLOBAL HEAP

Usted llama a la función GlobalAlloc para asignar un objeto de un tamaño especificado en el global heap. Windows administra los objetos en el global heap de acuerdo a la misma clasificación usada en el local heap: fija, movible, y descartable.

Los mismos mecanismos para compactar memoria que son aplicados en el manejo del local heap, también son aplicados en el global heap. Así, puede especificar GMEM_NODISCARD o GMEM_NOCOMPACT cuando llame a la función GlobalAlloc.

GlobalAlloc retorna un handle al objeto de memoria global asignado. Si no hay memoria disponible en el global heap, GlobalAlloc retorna un valor NULL. Siempre es importante chequear el valor retornado de GlobalAlloc, puesto que usted no tiene garantizado que su requerimiento sea satisfecho. Las mayoría de funciones que manejan memoria global requieren este handle para identificar el objeto de memoria.

BLOQUEANDO Y DESBLOQUEANDO UN OBJETO DE MEMORIA GLOBAL

Usted puede omitir la referencia de el handle de un objeto de memoria llamando a la función GlobalLock. Esta función retorna un puntero far que garantiza mantenerlo válido hasta que usted posteriormente llame a la función GlobalUnlock.

GlobalLock debe bloquear el objeto fijándolo en memoria para asegurar que el puntero que este retorne se mantenga válido hasta que llame GlobalUnlock. Debido a que este ha bloqueado el objeto, GlobalLock incrementa una cuenta del lock para el objeto. Esta cuenta ayuda a prevenir que el objeto sea liberado o descartado mientras se mantenga siendo usado.

Windows no necesita fijar el objeto en memoria a menos que este sea descartable. El puntero siempre será válido cuando el objeto se mueva en memoria lineal. Debido a que Windows no bloquea el objeto en memoria, GlobalLock no hace un incremento a la cuenta del lock para un objeto no descartable. GlobalUnlock decrementa la cuenta del lock de un objeto sólo si GlobalLock incrementó ésta para mencionado objeto. Sin embargo, debe llamar a GlobalUnlock cuando no requiera más el puntero retornado por GlobalLock.

En adición a GLoballock y GlobalUnlock, otras funciones afectan la cuenta del lock para un objeto:

Incrementan el lock count	Decrementa el Lock count
GlobalFix	GlobalUnfix
GlobalWire	GlobalUnWire
LockSegment	UnlockSegment

La función GlobalFlags retorna la cuenta del lock como es establecida por estas funciones.

Como anotamos anteriormente, no es necesario llamar a LocalLock para omitir la referencia de un handle local si el objeto es asignado como LMEM_FIXED. No hay capacidad similar para objetos globales fijos. Incluso los objetos globales fijos deben estar bloqueados para omitir la referencia de el handle.

El siguiente ejemplo usa el GlobalLock para omitir la referencia de un objeto global movible:

```
HGLOBAL hGlobalObject;
char FAR* lpGlobalObject;

if ( hGlobalObject = GlobalAlloc( GMEM_MOVEABLE, 1024 )) {
    if ( lpGlobalObject = GlobalLock( hGlobalObject )) {

            /*
             * use lpGlobalObject como la dirección far del
             * objeto globalmente asignado
             */

            .
            .
            .

            GlobalUnlock( hGlobalObject );
    }

    else {

            /* el bloqueo falló. Proceda en forma adecuada. */
```

```
    }
}

else {

    /* los 1024 byte no pueden ser asignados. Proceda adecuadamente. */

}
```

Si usted asigna un objeto cuyo tamaño es 64K o más, debería computar y salvar el puntero retornado por GlobalLock como un huge pointer. El siguiente ejemplo asigna un objeto de memoria global de 128K:

```
HGLOBAL hGlobalObject;
char huge * hpGlobalObject;

if ( hGlobalObject = GlobalAlloc( GMEM_MOVEABLE, 0x20000L ) ) {
    if ( hpGlobalObject =
                    (char huge * ) GlobalLock( hGlobalObject ) ) {
            /*
             * use hpGlobalObject como la dirección far del
             * objeto globalmente asignado
             */
            .
            .
            .
            GlobalUnlock( hGlobalObject );
    }

    else {

            /* el bloqueo falló. Procesa adecuadamente. */

    }
}

else {

    /* los 128K no pueden ser asignados. Proceda en forma adecuada. */

}
```

CAMBIANDO UN OBJETO DE MEMORIA GLOBAL

Usted puede cambiar el tamaño o atributo de un objeto de memoria global mientras preserva su contenido llamando a GlobalReAlloc. Si especifica un tamaño más pequeño, Windows trunca el objeto. Si especifica un tamaño más grande y también especifica GMEM_ZEROINIT, Windows llena la nueva área con ceros. Al especificar GMEM_DISCARD o GMEM_NOCOMPACT, asegura que Windows no descarte o mueva el objeto para satisfacer el requerimiento GlobalReAlloc.

También puede llamar a GlobalReAlloc para cambiar el atributo del objeto de no-descartable a descartable, o viceversa. A diferencia de LocalReAlloc, sin embargo, GlobalReAlloc puede cambiar un objeto GMEM_FIXED a GMEM_MOVEABLE o

la habilidad de Windows para manejar estos otros objetos eficientemente. Para bloquear un objeto de memoria descartable para un periodo prolongado, use la función GlobalWire. Para bloquear un objeto de memoria no descartable para un periodo prolongado, use GlobalLock.

GlobalWire vuelve a localizar el objeto movible a un área baja del global heap reservada para los objetos fijos y entonces bloquea este. Moviendo un objeto bloqueado a memoria baja, Windows puede compactar la memoria alta (upper memory) más eficientemente pero requerirá ciclos adicionales de CPU para mover el objeto. Llame GlobalUnWire para desbloquear un objeto. Después que el objeto es desbloqueado, este puede emigrar fuera de la porción fija del global heap.

NOTIFICANDO CUANDO UN OBJETO DE MEMORIA GLOBAL ESTA PARA SER DESCARTADO

Si desea que su aplicación sea notificada cada vez que Windows este cerca de descartar un objeto de memoria global, llame a la funcion GlobalNotify. Por ejemplo, GlobalNotify es útil si usted está escribiendo un sistema de administración de memoria virtual que intercambie datos desde y hacia el disco.

MODIFICANDO EL MOMENTO CUANDO UN OBJETO DE MEMORIA GLOBAL ES DESCARTADO

Windows emplea un algoritmo LRU para manejar el global heap, este algoritmo determina cual objeto de memoria será descartado cuando la memoria deba ser liberada. Usted puede llamar a la función GlobalLRUOldest para mover un objeto a la posición más antigua en la lista del LRU. Esto significa que este objeto será el más apropiado a ser descartado si Windows posteriormente requiere más memoria. Opuestamente, llamando a la función GlobalLRUNewest, usted asegura que un objeto es el menos probable ser descartado.

Estas funciones son útiles, por ejemplo para descartar el código de inicialización cuando este no sea más necesitado. También podría usar esta función si escribe un sistema administrador de memoria virtual que intercambie datos desde y hacia el disco. Con estas funciones puede influenciar en cuales objetos son los menos o más probables a ser descartados por Windows, minimizando así la cantidad de intercambio a disco.

LIBERANDO MEMORIA GLOBAL EN CONDICIONES DE BAJA MEMORIA

La memoria global es un recurso compartido; la perfomance de todas las aplicaciones dependen de la habilidad de que todas las aplicaciones compartan este recurso. Cuando la memoria del sistema es baja, su aplicación debería estar preparada para liberar la memoria global que esta tiene asignada.

Windows envía el mensaje WM_COMPACTING a todas las ventanas top-level (alto nivel) cuando detecta que más del 15% del tiempo de sistema sobre un intervalo de 30 a 60 segundos está siendo perdido compactando memoria. Esto indica que la memoria del sistema está baja.

Cuando su aplicación recibe este mensaje, debería liberar tanta memoria como sea posible, tomando en cuenta el nivel actual de actividad de la aplicación y el

número total de aplicaciones corriendo en Windows. La aplicación puede llamar a la función GetNumTaskd para determinar la cantidad de aplicaciones corriendo.

12.4 USANDO BYTES EXTRAS EN WINDOWS Y LA ESTRUCTURA DE DATOS CLASS

Usted puede almacenar extras, dato definido por la aplicación usando la estructura de datos que describe el atributo de una ventana o una clase de ventana. Este dato extra es conocido como window extra bytes y class extra bytes.
Este dato privado es localizado en el final de la estructura de datos que Windows mantiene para la ventana. Cuando llama a la funcion RegisterClass, el miembro cbWndExtra de la estructura WNDCLASS especifica el número de bytes extras de información que será manteniendo para cada ventana de esta clase. Los bytes extras son inicializado a cero.

La técnica de usar el área de datos privada de una ventana es particularmente útil en casos donde tenga dos o más ventanas que pertenecen a la misma clase, y quiera asociar datos diferentes con cada ventana.

Sin una facilidad de datos privados, usted tendría que mantener una lista de estructuras de datos privadas para cada ventana. Entonces cada momento que necesite accesar los datos para una ventana en particular, primero tendría que localizar la correspondiente entrada en la lista. Al usar la facilidad de datos privados, sin embargo, directamente puede accesar los datos privados a través del handle de ventana en lugar usar una lista separada.

Una ventaja adicional de usar un área de datos privados de ventanas para almacenar datos es que puede encapsular el dato asociado con cada ventana mejor que si fuese a almacenar tanto dato estático en el mismo módulo como, por ejemplo, el procedimiento de ventana.

Para escribir al área de datos privados ventana, llame las funciones SetWindowWord y SetWindowLong. Esta dos funciones aceptan un offset de byte dentro del área que usted establece aparte para dato privado. Un offset de cero refiere al primer valor de palabra o longitud (word o long) en el área privada. Un offset de 2 (bytes) refiere a al segundo valor de palabra en el área privada. Un offset de 4 (bytes) refiere al tercer valor de palabra en el área privada.

Note que SetWindowWord y SetWindowLong, también aceptan constantes tales como GWW_STYLE y GWL_WNDPROC, las cuales están definidas en WINDOWS.H. Estas constantes son offset negativos dentro de la estructura de ventanas. La longitud de la estructura (menos el área privada) es de este modo adicionada al offset que usted provee en la llamada a SetWindowWord o SetWindowLong para determinar el offset relativo al inicio de la estructura.

Para leer del área de datos privados de ventana, llame a las funciones GetWindowWord y GetWindowLong. El offset que usted especifica trabaja de la misma forma para SetWindowWord y SetWindowLong.

La estructura para una ventana es asignada en el User's local heap (local heap de usuario). Si quiere asociar una gran cantidad de datos (más de 10 bytes) con la

ventana, debería almacenar un handle global en el área privada de ventana en lugar de almacenar el dato actual. El handle apunta al dato. De esta manera, incrementa el tamaño de la estructura de ventana únicamente por los dos bytes necesitados para el handle global, al contrario de un área de datos privados de gran tamaño.

Sólo como puede asociar datos privados con una ventana particular, usted puede también asociar datos privados con la Windows class. Las funciones que hacen esto son SetClassWord, SetClassLong, GetClassWord y GetClassLong. Probablemente hay pocas ocasiones para asociar datos privados con clase de ventanas (Windows class) que con una ventana (window). Usar el área de datos privados para una window class es apropiado para datos que están lógicamente relacionados a la Windows class como una entidad y que es común entre múltiples Windows de la misma clase.

MANEJANDO RECURSOS

Un recurso en un dato almacenado de sólo-lectura de su archivo de aplicación .EXE o su archivo de librería .DLL que Windows lee desde el disco como sea requerido. Ciertos tipos de recursos tienen formatos prescritos que Windows reconoce. Estos incluyen bitmaps, icon, dialox box, y fonts. Usted puede crear estos recursos usando en editor de recurso incluido el en Microsoft Windows 3.1 Software Development Kit (SDK): Microsoft Image Editor (IMAGEDIY.EXE), Microsoft Dialog Editor (DLGEDIT.EXE), y Microsoft Font Editor (FONTEDIT.EXE). Usted enlaza estos recursos en su archivo .EXE o .DLL usando el compilador de recurso de Windows Resource Compiler (RC). Puede tomar ventaja de la habilidad de Windows para trabajar con estos recursos de formatos llamando las funciones asociadas tales como LoadIcon y CreateDialog.

Un recurso dentro de memoria es leído por Windows como un simple segmento de datos. El recurso puede ser declarado en el archivo de definición de recursos para ser fijo, movible, o descartable, tomando en cuenta las mismas consideraciones como para objetos de memoria global.

Si declara objeto usando la opción PRELOAD, Windows carga el recurso en memoria durante el arranque (startup) de su aplicación. De otra manera Windows carga este cuando sea necesitado (la opción LOADONCALL).

En adición al uso de recursos cuyo formato Windows reconoce, también puede desarrollar recursos que sólo su aplicación reconoce. Los datos pueden estar en cualquier formato que diseñe, incluyendo texto ASCII, datos binarios, o una mecla de ellos.

Cuando decida si mantener datos como recursos o como un archivo separado considere lo siguiente:

* Al compilar recursos dentro de su archivo de aplicación .EXE, simplifica el empaquetamiento de su aplicación. Usted y sus usuarios no necesitan preocuparse de la instalación de archivos de datos adicionales junto con el archivo de aplicación .EXE.

* De la otra manera, mantener los datos como un recurso significa que debe recompilar su archivo de aplicación .EXE si usted cambia la data. Si usted planea distribuir datos actualizados a varios usuarios, puede encontrar que es más fácil distribuir un nuevo archivo de datos en lugar de un nuevo archivo .EXE.

LOCALIZANDO UN RECURSO DE CLIENTE

La función FindResource determina la localización de un recurso acorde al nombre especificado en su archivo de definición de recursos. Esta función retorna un handle, el cual puede ser usado en una llamada al archivo LoadResource para cargar el recurso. El handle de recurso retornado por FindResource refiere a la información que describe el tipo de recurso declarado en el archivo de definición de recursos, la posición del mismo en el archivo .EXE o .DLL, y su tamaño.

Por ejemplo, suponga que desea mantener un archivo ASCII como recurso. El archivo texto fuente es denominado MYTEXT.TXT. Nombre el recurso MyText, y arbitrariamente nombre el tipo de recurso TEXT. La sentencia de definición de recursos es:

```
MyText    TEXT  MyText.txt
```

En su aplicación usted recupera el handle de recurso llamando a FindResource:

```
HANDLE  hMyTextResLoc;
    .
    .
    .

    hMyTextResLoc = FindResource( hinst, "MyText", "TEXT");
```

CARGANDO UN RECURSO DE CLIENTE

El llamar a FindResource no carga e recurso desde el archivo .EXE o .DLL dentro de memoria. Al contrario, ésta sólo localiza el recurso y retorna el resultado de la búsqueda como un handle que apunta a la información de la localización de recursos. Para cargar el recurso a memoria, llame a la función LoadResource:

```
HANDLE  hMyTextResLoc;
    HGLOBAL hMyTextRes;
    .
    .
    .

    hMyTextResLoc = FindResource( hinst, "MyText", "TEXT");
    if (!hMyTextRes = LoadResource( hinst, hMyTextResLoc)) {
            /*
             * caso de memoria no disponible para
             * cargar recurso
             */
    }
```

LoadResource por si mismo llama a GlobalAlloc para asignar el objeto de memoria para el recurso de dato, y entonces copiar el dato desde el disco al objeto de memoria.

BLOQUEANDO Y DESBLOQUEANDO OBJETOS DE MEMORIA

Para accesar el recurso de dato residentes en un objeto de memoria global debe llamar a la funcion LockResource para bloquear el recurso y recuperar u far pointer a el dato. Esto es equivalente a usar la funcion GlobalLock para recuperar el far pointer a un objeto de memoria asignado por la función GlobalAlloc. El siguiente ejemplo continua lo anterior:

```
LPSTR     lpstrMyText;

.
.

lpstrMyText = LockResource (hMyTextRes);
```

Una vez que tiene la dirección far al recurso, usted puede leer este como lo haria un objeto de memoria global bloqueado por GlobalLock.

Si ha definido el objeto como descartable, y este ha sido descartado, LockResource primero cargará el recurso desde el disco. Diferente a GlobalLock, LockResource le salva de nuevamente llamar a LoadResource si el recurso ha sido descartado.

Usted deberia llamar a UnlockResource cuando no este en proceso de accesar el recurso de dato. Esta función es equivalente a GlobalUnlock. Si declara el recurso como movible o descartable, este le provee a Windows la flexibilidad de mover o descartar un recurso de memoria como sea necesario para satisfacer otros requerimientos de asignación de memoria.

LIBERANDO UN RECURSO DE CLIENTE

La función FreeResource es similar a a función GlobalFree. Esta descarta la memoria usada por el recurso de datos así como el handle de recurso. Si necesita cargar nuevamente el recurso, puede llamar a LoadResource, usando el handle de localización de recurso retornado por su llamada inicial a FindResource.

USANDO MODELOS DE MEMORIA

Una aplicación Windows es como una aplicación DOS que puede tener uno o más segmentos de código. El modelo de memoria, el cual usted especifica cuando compila los módulos de código fuente, determina si sus compilador genera instrucciones usando direccionamiento far o near. Si usa un modelo de memoria que especifica sólo un segmento de código o dato, el compilador genera instrucciones que emplea direccionamiento near (16-bit) para referencia de código o dato, respectivamente. Si compila usando un modelo de memoria que especifica múltiples segmentos de código o dato, el compilador genera instrucciones que usan direccionamiento para referencia de código o dato.

Hay dos modelos de memoria, large y huge (grandes y enormes), para compilar código que genere direcciones far tanto para referencia de código como de datos. En el modelo de memoria large, el far pointer puede ser incrementado sólo dentro de un rango de offset de 64K de un segmento. En el modelo de memoria enorme (huge memory model), los far pointer pueden ser incrementados a través de regiones de 64K, causando que tanto la dirección de segmentos como la de offset sean incrementadas. También si un módulo de memoria es compilado con el modelo de memoria large, Windows será capaz de cargar sólo una instancia del módulo.

La siguiente figura muestra como el modelo de memoria afecta la manera que una aplicación direcciona código o dato:

		Número de segmentos de código: Uno	Número de segmentos de código: Múltiples
Número de segmentos de datos	Uno	Small memory model	medium memory model
	Múltiples	compact memory model	large memory model

Figura 5-4. Relación entre segmentos

Idealmente, una aplicación Windows usa un modelo de memoria medium, y el tamaño de sus módulos serán de 8K o menos. El módulo que inicializa la aplicación debería ser marcado PRELOAD y DISCARDABLE en el archivo de definiciones .DEF. El módulo que procesa la cola de mensajes debe ser marcada PRELOAD. Todos los otros módulos podrían ser marcados LOADONCALL y DISCARDABLE. Una aplicación que sigue estas guías de línea iniciará más rápido y consumirá menos recursos.

Si está usando CL, compile los módulos en lenguaje C de su aplicación Windows *usando las opciones /AS para el modelo de memoria small o la opción /AM para* el modelo medium.

También puede usar un modelo de memoria combinado. Para un modelos de memoria combinado compile los módulos usando la opción /AS, asignando el mismo nombre de código de segmento para aquellos módulos cuyo segmento de código usted quiera agrupar juntos, y asigne un diferente nombre de segmento de código a aquellos módulos para los cuales usted quiera generar diferente segmento de código. Para asignar un nombre de segmento de código a un módulo, use la opción /NT en CL. Una función que es llamada desde un segmento de código diferente debe ser declarada como una función far en el módulo donde la llamada es realizada., como en el siguiente ejemplo:

```
UINT FAR PASCAL        FuncInAnotherCodeSeg( UINT, LONG );
```

```
UINT uReturn;

.
.
.
uReturn = FuncInAnotherCodeSeg( 0, 0);
```

La ventaja de usar un modelo de memoria combinados es que usted necesita sólo definir las llamadas hechas entre segmentos de código FAR. Funciones que son declaradas FAR incrementa el tamaño de código y requieren más ciclos de máquina para ser llamadas.

Para otra forma de modelos de memoria combinados, puede compilar módulos con la opción /AM, la cual por defecto hace llamadas de funciones FAR. Entonces, en lugar de declarar las funciones FAR, piense como NEAR aquellas funciones que son llamadas sólo dentro del mismo segmento. Las desventaja de este método es que todas las funciones de librería run-time también deben ser funciones FAR.

USANDO DATOS ENORMES (HUGE DATA)

Usted puede declarar datos huge (enormes) en módulos en lenguaje C. CL ejecutará correctamente la aritmética requerida para incrementar el puntero a través de regiones de segmentos. Puede pasar un huge pointer a funciones de librería Windows o a sus propias funciones que esperan far pointer, pero sólo si la función internamente no espera incrementar el far pointer tal que este apunte a un objeto que rebase una región de 64K. Por ejemplo el siguiente código es aceptable, debido a que 16 es un factor de 64K (65536):

```
char huge Record[10000][16];
int i;

TextOut( hDC, x, y, (LPSTR) Record[i], 16);
```

El siguiente ejemplo viola esta limitación, debido a que el puntero pasado a la función TextOut eventualmente apuntará a un objeto que rebasa una región de 64K:

```
char huge Record[10000][15];
int i;

    /* NO HAGA ESTO */

TextOut( hDC, x, y, (LPSTR) Record[i], 15);
```

Puesto que 15 no es un factor de 64K, el segmento sería incrementado a través de un segmento colindante.

TRAMPAS QUE EVITAR CUANDO MANEJAMOS DATOS DE PROGRAMAS

Esta sección está enfocada sobre los errores de programación en Windows comunes cuando maneja datos de programas. Una vez que entienda como Windows maneja la memoria, las siguientes líneas de guía serán más claras

No asuma el nivel de privilegio en el cual su aplicación está ejecutándose.

Futuras versiones de Windows podrían cambiar el nivel de privilegio sustituyendo en el cual la aplicación se ejecuta.

No utilice los servicios de interface de modo protegido del DOS (DPMI) en una aplicación Windows.

Usted puede usar los servicios DPMI sólo en una librería DLL, y únicamente los servicios DPMI no proveídos con Windows. No use los servicios de DPMI para colgar las interrupciones o fallas. La especificación DPMI no provee para descolgar cadena de interrupciones.

Evite los far pointer a datos estáticos en los modelos small y medium.

Suponga que el módulo contenga la siguiente declaración

```
/* NO SIGA EL SIGUIENTE EJEMPLO */

static LPSTR lpstrDlgName = "MyDlg";
    .
    .
    .
    hDlg = CreateDialog( hinst,
                         lpstrDlgName,
                         hWndParent,
                         (DLGPROC) lpDialogProc);
```

El puntero LPSTR (char FAR *) inicialmente establecido por el cargador Windows será inválido si el segmento de datos automáticos que contiene el literal MyDlg mueve se mueve en memoria (a menos que el segmentos de datos automáticos este en un segmento fijo).

La forma adecuada para escribir el código precedente es declarar el string con un puntero near, PSTR (char NEAR *), y vaciar este al tipo de dato LPSTR requerido por la función CreateDialog, como se muestra a continuación.

```
/*  SIGA EL SIGUIENTE EJEMPLO */

static PSTR pstrDlgName = "MyDlg";
    .
    .
    .
    hDlg = CreateDialog( hinst,
                         (LPSTR) pstrDlgName,
                         hWndParent,
                         (DLGPROC) lpDialogProc);
```

Para vaciar dinámicamente a LPSTR introduzca el valor actual del registro DS él lugar del valor del DS en el momento que el módulo fue cargado.

No pase datos a otras aplicaciones por medio de un handle global.

No debería usar un handle global para compartir datos con otra aplicación, debido a que puede asumir que su aplicación y otras aplicaciones tiene espacio de direcciones separadas.

En futuras versiones de Windows, los espacios de direcciones de aplicaciones podrían estar separadas.

El único método soportado por Windows para pasar datos entre aplicaciones son el clipboard y el protocolo Dinamic Data Exchange (DDE). Si pasa un handle global a través de DDE a otra aplicación, el objeto global debió haber sido asignado con el flag GMEM_DDESHARE.

No asuma relación alguna entre un handle y un far pointer en cualquier modo.

Cuando use objetos de memoria global, siempre debe llamar a la función GlobalLock para degradar la referencia de un handle a un far pointer, considerando el modo en el cual Windows está corriendo.

No cargue un registro de segmento con un valor diferente a uno provisto con Windows o DOS.

En Windows, los registros de segmentos son interpretados como selectores, no como direcciones de párrafos físicos. Por lo tanto no debería leer la tabla de interrupciones estableciendo ES y DS a cero, por ejemplo. Use únicamente las funciones de DOS apropiadas para colgar un vector de interrupciones.

No realice segmentación aritmética.

No incremente la dirección de segmentos de un far pointer en un intento de incrementar el puntero. Esta técnica no es soportada por Windows.

No compare direcciones de segmentos.

No compare los valores de selectores que Windows asigna a objetos de memoria para determinar cual objeto está en memoria más baja. Esta técnica no es soportada en Windows.

No lea o escriba después del final de un objeto de memoria.

No lea o escriba después del final de un objeto de memoria bajo cualquier circunstancia. Si bien esto puede resultar en ninguna detección en otras configuraciones, Windows reportará este error como una falla GP.

CONCLUSIONES

Como podemos notar a través de los dos primeros capitulos, la industria del software actualmente tiene una herramienta poderosa para tomar ventaja de las capacidades y bondades de los sofisticados equipos de hardware con que cuenta la industria de los computadores personales en la actualidad.

Algunos computadores personales, de acuerdo a su configuración, no tendrían nada que envidiar a un minicomputador, puesto que poseen una gran capacidad de direccionamiento, y por tanto un extensa habilidad para almacenar datos y fácilmente referenciarlos y recuperarlos.

Con el diseño, análisis, y programación orientada a objetos podemos encontrar fácilmente un enfoque de solución a problemas cuya complejidad no sería muy difícil y complicado de comprender usando los métodos de y herramientas de análisis hasta ahora convencionales.

En cuanto a Microsoft Windows se le vislumbra un espacio predominante en el área de los sistemas operativos para los computadores personales, por su gran capacidad datos de tipo texto y/o binario.

Windows maneja sus datos dentro de una amplia capacidad de memoria virtual, lo cual es transparente, facilitando de esta manera un enfoque menos abstracto al tipo de aplicaciones que el usuario requiere.

De esta manera el usuario puede tomar ventaja de la real capacidad y bondad que ofrezcan determinadas aplicaciones, ya que únicamente requiere entender y aprender a manejar los recursos individuales de cada aplicación, porque Windows se encarga de lo restante.

Para los programadores amantes de los retos y desafíos, Microsoft C/C++ les ofrece una gran alternativa (envidiable para el resto de software de desarrollo), para implementar aplicaciones de lo más sofisticadas.

Debido a que tales aplicaciones a mas de poder ejecutarse dentro de un ambiente como lo es Windows, con todas las ventajas y facilidades que ofrece este ambiente de operación como son: gráficos, textos, sonidos, etc., es decir, un completo ambiente multimedia; También podemos tomar ventaja de sus recursos internos para beneficio de nuestras aplicaciones.

La versión 7.00 de Microsoft C/C++ trae consigo el paquete completo del Software Development Kit (SDK), con el cual podemos administrar la memoria que usa nuestra aplicación particular de una forma más eficiente.

Finalmente hay que resaltar que este documento no trata de enfocar un curso de programación, al contrario su cometido es el de dar al lector una visión y enfoque de entendimiento de como puede programarse aplicaciones, manejando correctamente las funciones de SDK, para tomar ventaja de un ambiente de usuario Windows.

BIBLIOGRAFIA

1. Barkakati, Nabajyoti : The Waite Group's Turbo C++ Bible, The The Waite Group, Inc., SAMS, Carmel, Indiana, 1990.

2. Boock, Grady : Object Oriented Design with Applications, The Benjamin / Cummings Publishing Company, Inc. Reswood City, California, 1991.

3. Jamsa, Kris A.: Microsoft C: Secrets, Shortcuts, and Solutions, Microsoft Press, Redmond, Washington, 1989.

4. Microsoft C/C++, Microsoft Press Microsoft Corporation, Printed in United States of America, 1987-1992.

5. Microsoft Windows 3.1 Operanting System, Microsoft Press Microsoft Corporation, Printed in United States of America, 1990-1992.

6. Microsoft Windows 3.1, Programmer's References Library, Microsoft Press Microsoft Corporation, Printed in United States of America, 1987-1992.

7. Microsoft Windows Resource Kit for Windows 3.1 Operanting System, Microsoft Press Microsoft Corporation, Printed in United States of America, 1992.

8. Mullin, Mark : Object Oriented Program Design with Examples in C ++, Addison - Wesley Publishing Company, Inc., Printed in United States of America, 1990