UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

COLEGIO DE CIENCIAS Y HUMANIDADES
UNIDAD ACADEMICA DE LOS CICLOS
PROFESIONALES Y POSTGRADO
INSTITUTO DE MATEMATICAS APLICADAS Y SISTEMAS
MAESTRIA EN CIENCIAS DE LA COMPUTACION


# TECNOLOGIA PARA DESARROLLAR PROGRAMAS RESIDENTES EN MEMORIA

**T E S I S**

QUE PARA OBTENER EL GRADO DE:

**MAESTRO EN CIENCIAS DE LA COMPUTACION**

P R E S E N T A


JOSE JESUS CASILLAS PELLAT


MEXICO, D. F. 1989


TESIS CON
FALLA DE ORIGEN

**UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso**

**DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL**

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Tabla de contenido.

Agradecimientos. ................................ i

Tabla de contenido. ................................ ii

Introducción. ................................ 1

Capítulo 1.- Creación de un ambiente de desarrollo para programas residentes en memoria. ............... 11

1.1 Antecedentes ................................ 11
1.2 Descripción del desarrollo ................ 13
1.3 Medio ambiente de ejecución de DOS ........... 16
1.4 Acceso al programa residente ................ 20
1.5 Utilización de archivos ...................... 24

Capítulo 2.- Instalación de programas residentes en memoria y manejo de interrupciones. .................. 28

2.1 Terminación y permanencia en memoria ......... 28
2.2 Utilización de funciones de DOS ............. 31
2.3 Manejo de interrupciones ..................... 33
2.4 Despachador de procesos ..................... 42
2.5 Comunicación entre procesos .................. 48

Capítulo 3.- Utilización de Corrutinas. .............. 56

3.1 Conveniencia del uso de corrutinas ........... 56
3.2 Intercambio de control ...................... 61
3.3 Despachador de corrutinas .................... 65
3.4 Consideraciones de programación .............. 68

Capítulo 4.- Remoción de programas residentes en memoria. ........................................ 72

4.1 Condiciones para una remoción segura ......... 72
4.2 Restauración de interrupciones y bloques de memoria ........................................ 78

Capítulo 5.- Aplicaciones. ........................... 81

5.1 Captura de datos experimentales. ............. 81
5.2 Solución de problemas con métodos numéricos. . 82
5.3 Simulaciones, reportes. ..................... 83
5.4 Envio de mensajes en redes locales de computadoras (sin "SERVER"). ................ 83
5.5 Sincronización de procesos en redes locales de computadoras. .............................. 85
5.6 Sistema automático de respaldo de archivos. .. 86
5.7 Sistema de protección anti-virus. ............ 86
5.8 Alertas. .................................... 87
5.9 Alarmas. .................................... 88

Capitulo 6.- Esquema de desarrollo de la tecnologia. . 89

6.1 Desarrollo de programas residentes ........... 89
6.2 Interfaz con "C" ............................ 93
6.3 Manejo de corrutinas ........................ 95
6.4 Comunicación entre programas ................ 97

Capitulo 7.- Conclusiones. ........................ 99

7.1 Discusión ................................... 99
7.2 Degradación de la capacidad de proceso ....... 101
7.3 Relevancia del desarrollo ................... 105

7.4 Posibilidades de extension .................. 106

Apéndice A.- Programas residentes activados via teclado. .................................... 108

Apéndice B.- Ejemplos de rutinas críticas de bajo nivel utilizadas por este tipo de desarrollo. ........ 114

Apéndice C.- Ejemplos de programas en lenguaje "C" y rutinas de alto nivel desarrolladas o usadas con esta tecnología. .................................... 133

Bibliografía .................................... 145

**Tecnología para desarrollo de programas residentes en memoria.**

Introducción.

Un programa residente en memoria es un programa de computadora que se mantiene de manera permanente en la memoria de la máquina. Este tipo de programas son activados normalente por medio de las interrupciones del procesador. Al permanecer en memoria, estos programas tienen la ventaja de poder ser activados desde el interior de otros programas, sin necesidad de interrumpir el trabajo que se está llevando a cabo.

Las utilerías residentes en memoria han cobrado una gran importancia y aceptación dentro del campo de la computación personal, debido al incremento que se ha dado en la capacidad de manejo de memoria de las microcomputadoras, así como a la funcionalidad y conveniencia que muchas de estas utilerías proporcionan.

Entre las más comunes que encontramos están todo tipo de agendas, relojes, calculadoras, alarmas, marcadores

telefónicos, entre las más simples y que permiten realizar alguna operación desde adentro de cualquier otra aplicación sin tener que interrumpir el trabajo en ésta. Existen otras como manejadores de disco, que optimizan el acceso a éste; extensiones al manejo de teclado que permiten diseñar macros de secuencias de teclados, muy útiles cuando es necesario hacer secuencias repetitivas o durante las fases de prueba de los programas, hay otros más sofisticados que por ejemplo toman a ciertos intervalos de tiempo, imágenes instantáneas del contenido de la memoria y lo escriben a disco, siendo de gran utilidad cuando ocurre una caida del sistema, o bien una excelente herramienta para la corrección de programas durante la fase de desarrollo; utilerías para corrección de programas; sistemas de recursos compartidos dentro de ambientes de redes de computadoras tales como archivos, impresoras, etc.

Existen además otro tipo de programas que en el sentido estricto de la palabra, tambien son residentes en memoria, tales como los llamados "device drivers" o manejadores de dispositivos, que por lo general se encargan del control de alguna tarjeta de expansión de la computadora, o de los dispositivos periféricos de ésta, manejadores de acceso a los diferentes mecanismos para expansión de memoria, discos en memoria y por supuesto, la parte residente del sistema operativo.

Un tipo de aplicación muy importante que se puede dar a los programas residentes en memoria, y que ha sido relativamente poco explotada es la de utilizar procesos paralelos en sistemas operativos que no dan esta facilidad, así, mientras un programa residente esta ejecutando un proceso interno ("background"), al nivel del sistema operativo se puede ejecutar cualquier otro proceso externo ("foreground"), simulando así, una estructura multiproceso. Cabe aclarar que no nos estamos refiriendo al caso en que un mismo proceso aprovecha su tiempo de espera para ejecutar operaciones laterales, como algunos programas lo hacen por ejemplo cuando una hoja electrónica de cálculo va recalculando sus valores durante el lapso de espera en que el usuario captura o altera los datos. En el caso de procesos que cooperan entre sí, es preciso proporcionar algún mecanismo de sincronización e intercambio de información entre ellos.

La importancia de éste tipo de aplicaciones radica en la potencialidad que le agregan a un sistema de cómputo que no proporcione multiproceso explícitamente, al permitir que varias aplicaciones se ejecuten simultáneamente y de manera transparente sin necesidad de ningúna adición física de equipo al sistema, aumentando la eficiencia en el uso de los recursos de cómputo. La creación de este tipo de aplicaciones es hacia lo que esta dirigido el trabajo de esta tesis. En el caso de sistemas operativos que sí proporcionan la facilidad de ejecutar programas en modo paralelo ( ej.

UNIX ), la relevancia de la simulación de multiproceso por medio de programas residentes en memoria se ve disminuida, no así la de utilerías residentes en memoria activados via teclado o para expansión de los servicios proporcionados por el mismo sistema operativo.

Diferentes arquitecturas de máquinas implican diferentes tecnologías para el control de los programas residentes en memoria, así pues encontramos que éstos se pueden manejar a través de interrupciones al procesador (PC corriendo bajo el sistema operativo MSDOS), en otros casos en alguna cola de atención de procesos (MacIntosh) ó alguna otra facilidad que permita el equipo y sistema operativo que se esté utilizando.

En muchas ocasiones los programas residentes en memoria presentan conflictos entre si, del tipo de que cuando el programa "X" está corriendo desactiva al programa "Y", ó su funcionamiento depende del orden en que son cargados en memoria, etc. Esto sucede especialmente cuando ambos programas utilizan el mismo medio de acceso al procesador, y uno de ellos no se hace cargo de restablecer el ambiente de proceso apropiadamente o de ceder el control correctamente a la rutina que a su vez le cedió el control.

Entre los requerimientos mínimos que debe satisfacer un programa residente en memoria, encontramos que su operación

debe ser transparente para el usuario de los procesos externos, salvo en ciertos casos excepcionales en los que ocurra alguna condición de error ó un evento especial. Por esto quiero decir que el programa residente en memoria no debe tener interferencia no deseada con los demás procesos que se estén llevando simultáneamente a cabo en la máquina.

Un programa residente en memoria debe contar además con un mecanismo para desactivarse y de ser posible removerse de memoria (esta facilidad, como se mostrará más adelante no siempre es posible llevar a cabo de una manera limpia), dejando libre la memoria que ocupa para que algún otro proceso la pueda utilizar, de otra manera, si se cargan muchas de estas utilerias residentes en memoria, eventualmente la máquina se puede quedar sin la memoria suficiente para correr las aplicaciones que el usuario pudiera necesitar. Por esta misma razón, los programas residentes en memoria deben diseñarse para tener el tamaño más pequeño posible.

Las anteriores consideraciones hacen que en la mayoría de los casos, este tipo de utilerías estén desarrolladas completamente en lenguaje ensamblador. Sin embargo, para sistemas con un alto nivel de complejidad, el desarrollo de estos por medio de exclusivamente lenguaje ensamblador tendría como consecuencia, por un lado que el proceso de desarrollo fuera más lento y propenso a errores, y por otro

que el código de esta manera escrito quedaría atado a una arquitectura de máquina en particular perdiéndose así la posibilidad de creación de código portable que pueda utilizarse en diferentes arquitecturas.

Una de las caracteristicas mas sobresalientes de este proyecto es que al proveer una interfaz con un lenguaje de alto nivel (en este caso el lenguaje de programación "C"), permite el desarrollo sencillo y portable de aplicaciones para diferentes arquitecturas de máquinas, en las cuales solo la interfaz de bajo nivel permanece dependiente del sistema operativo y "hardware" de la máquina particular.

Este trabajo se enfoca principalmente al desarrollo de programas residentes en memoria para computadores del tipo PC corriendo bajo DOS ("Disk Operating System"). Esto hace que se relacione fuertemente con el manejo de la arquitectura interna de la familia de los procesadores Intel 8086 en la que ésta máquina está basada. Su objetivo es el de sortear las dificultades técnicas que representa tal desarrollo y mostrar sus soluciones adecuadas de manera que a partir de esta tesis, cualquier programador con cierto nivel de conocimiento de lenguaje "C" y del lenguaje ensamblador del 8086 pueda crear fácilmente un ambiente de desarrollo similar siguiendo los lineamientos aquí descritos. La lectura de esta tesis será mucho más fructifera si se cuenta con un cierto nivel minimo de comprensión de dichos lenguajes,

aunque no es un requisito.

Un breve esbozo de lo que se tratará en cada uno de los capítulos de esta tesis se presenta a continuación:

Introducción.

Se da un panorama de lo que son los programas residentes en memoria, su importancia y se describen algunas de sus características. Se incluye un breve resumen de la tesis.

Capítulo 1.- Creación de un ambiente de desarrollo para programas residentes en memoria.

En el se describe como crear un ambiente para programación en lenguaje "C", que permita una interfaz de desarrollo de alto nivel para la creación de programas residentes en memoria.

Capítulo 2.- Instalación de programas residentes en memoria y manejo de interrupciones.

Aquí se describen los pasos necesarios a seguir para que un programa residente quede instalado correctamente en memoria. Se discute además el manejo apropiado de las interrupciones y de los problemas de incompatibilidad que se presentan entre diversos programas residentes en memoria.

Capitulo 3.- Utilización de Corrutinas.

Este tópico es general y tiene que ver no sólo con programas residentes. La aplicación aquí, resulta interesante dado que representa una forma muy elegante para reducir el tamaño del programa residente en memoria al tener un sólo programa ejecutando varios procesos simultáneamente, utilizando solo una copia de las funciones de biblioteca del lenguaje y de las variables estáticas del programa.

Capitulo 4.- Remoción de programas residentes en memoria.

Se describe el proceso que es necesario para remover apropiadamente los programas residentes en memoria.

Capitulo 5.- Aplicaciones.

Se mencionan algunas aplicaciones típicas que se pueden desarrollar ó se han desarrollado con base en esta tecnología.

Capitulo 6.- Esquema de Desarrollo de la tecnología.

Se presenta un resumen completo de los puntos necesarios para para la creación de esta interfaz de desarrollo de programas residentes en memoria.

Capitulo 7.- Conclusiones y discusión.

Finalmente se presentan las conclusiones del proyecto y se analizan varias posibilidades de generalización del mismo a futuro.

Apéndice A.- Programas residentes activados via teclado.

Se detalla el manejo de programas residentes en memoria activados por las interrupciones del teclado.

Apéndice B.- Ejemplos de rutinas criticas de bajo nivel

utilizadas por este tipo de desarrollo.

Se muestran ejemplos de las llamadas a DOS descritas a lo largo de este trabajo, así como las rutinas de intercambio de stacks para corrutinas y ejemplos de algunas de las rutinas de atención a interrupciones.

Apéndice C.- Ejemplos de programas y rutinas de alto nivel desarrolladas o usadas con esta tecnología.

Se muestran ejemplos sencillos de programas residentes en memoria desarrollados en lenguaje "C" que usan esta interfaz. Se incluye un ejemplo en seudocódigo de rutina despachadora de corrutinas y otro de como comunicar programas por medio de semáforos evitando el abrazo mortal ("Deadlock").

---

* PC es marca registrada de IBM Corporation.
* MacIntosh es marca registrada de Apple Computer.
* MSDOS es marca registrada de Microsoft Corporation.

# Capitulo 1

## Creación de un ambiente de desarrollo para programas residentes en memoria.

### 1.1 Antecedentes

Todo programa residente en memoria tiene parte de su código escrito en lenguaje ensamblador, so pena de tener que dejar residente en memoria, además del programa mismo, al ambiente de ejecución del programa instalador. La mayor parte de los programas residentes en memoria están hechos en su totalidad en lenguaje ensamblador, esto tiene como consecuencias importantes principalmente un incremento en la complejidad de el desarrollo, un incremento de la dificultad durante la fase de corrección ([MENC89]) y en el caso de futuras ampliaciones al sistema o bien de labores de mantenimiento del mismo, además de perderse la posibilidad de portabilidad de código alguno a otros ambientes de programación, por otra parte, la eficiencia de ejecución y tamaño del código es mayor.

De las anteriores consideraciones surgió la idea de crear

una interfaz entre lenguaje ensamblador y algún lenguaje de programación de alto nivel ([STEA88a] [STEA88b]) en nuestro caso el lenguaje "C", de manera que la parte que se escribe en ensamblador sea la encargada de hacer todas las tareas de bajo nivel, tales como el arranque del programa, reservar la memoria necesaria para que el programa corra, atrapar las interrupciones requeridas, hacer el intercambio de contextos entre procesos, etc., dejando para el desarrollo en "C", las tareas de alto nivel, que en este caso corresponden al código que sería necesario aún en el caso en que la aplicación que se tiene en mente desarrollar no fuera a ser un programa residente en memoria, es decir, el desarrollo que provee su funcionalidad al programa residente.

En el caso en que los desarrollos sean pequeños (por pequeño me refiero a un orden de 5 Kilobytes de código), el sobrepeso que significa el cargar con el ambiente de la interfaz, podría desalentar su utilización, dado que este representaría un incremento en memoria utilizada que sería del mismo orden que el tamaño del programa, sin embargo, por la facilidad de desarrollo, mantenimiento y corrección del programa y las herramientas de programación disponibles, resulta recomendable aún en tales casos utilizar la interfaz aquí propuesta.

Se eligió el lenguaje de programación "C" ([KERB78]) para este desarrollo debido a su portabilidad y a su

eficiencia en la generación de código, lo cual permite la creación de desarrollos comunes con otras máquinas a la vez que el código que se genera es compacto, lo que representa una gran ventaja tanto debido a la reducción en tamaño del mismo como a su eficiencia al tiempo de ejecución.

## 1.2 Descripción del desarrollo

El compilador que se utilizado para crear esta interfaz fue "Lattice C" (versión 2.15), pero desarrollar alguna interfaz similar para otros compiladores comerciales como "Microsoft C Compiler" y "Turbo C" no es más complejo, y se puede crear siguiendo básicamente los lineamientos aquí descritos.

La interfaz con el lenguaje "C" se compone de dos partes principalmente. La primera parte es una versión modificada de la interfaz original al tiempo de ejecución que brinda el compilador (C.ASM), a la cual se le debe adicionar un medio de controlar externamente el tamaño del stack del programa y el tamaño de memoria reservado para estructuras dinámicas, junto con la adición de algunas variables que serán utilizadas para el intercambio de contextos entre los diferentes procesos. La segunda parte es un conjunto de rutinas en lenguaje ensamblador, una para cada una de las interrupciones que se van a atrapar, con la lógica adecuada para evitar los posibles problemas de reingreso. Debe

incluirse además una pequeña rutina de instalación cuya función es básicamente la de substituir las rutinas originales de interrupción por las nuevas, después de lo cual se puede dejar el programa residente en memoria mediante una llamada a la función 31h de DOS ([IBMC87]).

Es posible desarrollar la rutina de sustitución de interrupciones desde lenguaje "C", siempre y cuando se hagan públicas las rutinas en ensamblador que manejan las interrupciones. Esto tendría el posible problema de que no contamos con la garantía de que las llamadas al DOS que hacen el manejo de vectores de interrupción serían ejecutadas con interrupciones deshabilitadas, pudiendo ocasionar algún mal funcionamiento de la máquina bajo ésas circunstancias.

El programa residente en memoria permanece inactivo hasta que ocurre algún evento que lo despierta, siendo este evento generalmente una interrupción del procesador. Dependiendo de la naturaleza del programa, este manejará las interrupciones del teclado, acceso a disco, reloj, sistema operativo, etc.

Nuestra discusión se va a centrar en programas residentes que son manejados a través de la interrupción del reloj, es decir de programas que a intervalos periódicos de tiempo ganan el acceso al procesador de la máquina y se ejecutan parcialmente o en su totalidad. Sin embargo, la discusión

del tema es general y en el apéndice "A" se amplia el tema de programas residentes en memoria activados por la vía de las interrupciones del teclado.

La interfaz debe estar provista de un mecanismo que le permita evitar o manejar apropiadamente las condiciones de reingreso de las rutinas utilizadas, que se presentan por ejemplo cuando la condición de activación del programa residente en memoria se cumple estando éste activo, pudiendo ocasionar mal funcionamiento del mismo. Esta consideración es de especial cuidado en particular, cuando se maneja la interrupción del reloj de la máquina, dado que esta se genera aproximadamente 18.2 veces por segundo ([IBMC85]), por lo que cualquier programa residente en memoria que utilice ésta interrupción para ganar control del procesador y cuyo tiempo de ejecución sea mayor a un dieciochavo de segundo se va a encontrar con esta condición.

En el momento en que la condición de activación se cumple, el programa residente en memoria debe tomar control de la máquina, sin embargo antes de comenzar a ejecutar el código propio de su aplicación, debe encargarse de salvar el ambiente de ejecución del proceso que estaba activo al momento de la interrupción que le cedió el control. Ahora bien, dado que cuando se toma control del procesador 8086 a través de una interrupción, en general sólo se tiene a mano información de los valores del segmento de código y del

apuntador a la instrucción propios, los lugares donde se debe guardar tal ambiente deben formar parte del mismo segmento de código, de manera tal que se pueda tener acceso a ellos para salvarlos o restablecerlos apropiadamente. La declaración de estas variables debe pues hacerse de manera que queden incluidas en el segmento de código, típicamente se pueden declarar junto con la rutina que proporciona el compilador utilizado como interfaz a tiempo de ejecución.

## 1.3 Medio ambiente de ejecución de DOS

Encargarse de salvar el ambiente de ejecución del proceso suena muy interesante y casi obvio, pero, ¿Que es lo que entendemos por esto?, ¿Que es el ambiente de ejecución bajo DOS?. Lo que entendemos como salvar el ambiente de ejecución del programa activo al momento de la interrupción es lo siguiente: guardar una copia de los valores de todos los registros del procesador en ese momento, junto con las estructuras globales de control que utiliza el sistema operativo y los elementos del contexto de ejecución que pudieran ser afectadas por nuestro programa. Los elementos del contexto deben salvarse si es el caso que se van a utilizar unos propios, estos elementos son el stack, el PSP (descrito mas adelante), el tipo de cursor en pantalla, el modo de video utilizado, la memoria de video, el manejo de Error Crítico y el de Control-Break.

El PSP (Program Segment Prefix) es una estructura de datos que utiliza el DOS entre otras cosas para el control de sus operaciones con archivos y que en las primeras versiones de DOS también se utilizó para mantener un cierto nivel de compatibilidad con el sistema operativo CP/M, su antecesor directo. El DOS proporciona un PSP nuevo a cada proceso que se inicia, siendo normalmente el último PSP que se creó el que está en efecto todo el tiempo. El PSP es una estructura de datos de 256 bytes, documentada en el manual de referencia técnica de DOS ([IBMC87]), aunque muchos de sus campos no lo están.

Entre la información que se sabe que está contenida en el PSP, encontramos desde código para la terminación de programas, información acerca de la cantidad de memoria accesible al programa, código para llamadas a DOS, copia de los vectores de atención a interrupción de la rutina de terminación de DOS, la rutina de manejo de Control-Break y la rutina de atención a error crítico, el valor del segmento asociado a la copia del medio ambiente ("environment") local del proceso, los bloques de control de archivos (FCB) utilizados usualmente, así como un área de 128 bytes que se conoce como área de transferencia de disco ("DTA") usada. Al comenzar la ejecución de un programa, el area reservada para el DTA contiene una copia de la linea de comandos que se le pasaron al programa para ser interpretados por el mismo.

La información anterior es con respecto a los campos documentados en el manual de DOS. Dentro de los campos no documentados, se encuentran la dirección del PSP del proceso padre, la tabla local de manejadores de archivos, así como el tamaño de esta tabla y su localización en memoria, y la dirección del stack local que guarda DOS cuando es invocado por un programa antes de cambiarse a un stack propio de DOS.

Cuando se ejecuta la instrucción de termina y permanece residente, el PSP del programa que esta dejando residente también permanece residente. Esto es importante al momento de instalar en memoria, porque como el PSP no es parte del programa, en la instalación hay que tenerlo en cuenta al hacer la petición de memoria necesaria al sistema operativo y en el proceso de remoción hay que hacer la operación de liberar la memoria tanto del bloque de memoria del programa incluyedo el bloque de memoria del PSP asociado. Es sumamente importante para aplicaciones que van a utilizar manejo de archivos que se salve el PSP del proceso activo como parte del ambiente de ejecución, pues de lo contrario se causa todo tipo de mal funcionamiento, como que DOS confunda la tabla de manejadores de archivos activa, o bien, como DOS guarda en una variable el valor del PSP que cree que está activo y que es normalmente el último en ser cargado, ocasionar mal funcionamiento al devolver control a un stack que no sea el adecuado.

Este manejo del PSP se puede llevar a cabo por medio de la función 62h de DOS ( DOS 3.00 ó mayor) que sirve para obtener la dirección del PSP activo, y con la función no documentada 50h de DOS ([DUNR88]), la cual se usa para cambiar el PSP activo. Para versiones de DOS menores a 3.00 el PSP activo se puede obtener a través de la llamada no documentada 51h de DOS.

En el programa residente mismo, debe tenerse cuidado de que las operaciones que utilicen el área de transferencia de disco, se aseguren de utilizar un área propia, la cual sea reestablecida inmediatamente después de la operación, un ejemplo de este tipo de operaciones, son las llamadas al sistema operativo para búsqueda de archivos con nombre fijo ó incluyendo caracteres comodines dentro de algún directorio especificado, si no se hace el manejo adecuado del área de transferencia de datos, seguramente se afectará el funcionamiento de los demás programas que estén corriendo en ese momento, incluso el del mismo COMMAND.COM, puesto que en el DTA se guarda información referente a los anteriores comandos a disco ejecutados. Para facilidad de esta operación DOS proporciona un par de llamadas que realizan las funciones de cambiar y de obtener la dirección del área de transferencia de disco actual que son respectivamente las funciones 1Ah y 2fh de DOS ([IBMC87]). Ejemplos de mal funcionamiento que pueden ocurrir si no se toma tal

precaución son por ejemplo que falle una operación de DIR, o que se abran archivos con el nombre o directorio equivocado así como el posible mal funcionamiento de las operaciones a disco de otros programas que se encuentren corriendo simultáneamente.

### 1.4 Acceso al programa residente

La segunda parte de la interfaz con el lenguaje de programación "C" es la que se encarga de atrapar las interrupciones utilizadas por el programa residente de manera que se le pase control al programa residente en memoria cuando así sea requerido, éste proceso se hace de manera que la rutina de acceso al programa en "C", a la cual denominaremos CallC, llame cuando se ejecuta por primera vez a la rutina principal del programa en "C", que es _main() a partir de la cual e incluyéndola, todas las subrutinas posteriores se encuentran al nivel del lenguaje "C". Las siguientes veces que se ejecuta CallC, esta transfiere el control al punto en el cual se le paso a ella la última vez.

La rutina de acceso CallC queda descrita esquemáticamente de la siguiente manera:

1.- Salva los registros al momento de la interrupción en el

stack del proceso activo en el momento en que la condición de activación del programa ocurrió.

2.- Salva los valores del segmento de stack y el apuntador a éste en variables del segmento de código de la rutina que maneja la interrupción.

3.- Hace el intercambio del PSP del proceso activo por el del proceso residente en memoria.

4.- Si esta es la primera vez que se llama CallC, se asignan los valores apropiados al segmento de stack y apuntador al stack de el proceso residente y se manda llamar a la función _main(). Después de la terminación de _main() se pasa al punto 7. (Es posible que _main() nunca termine, por ejemplo si el programa residente es un ciclo infinito).

5.- Si no era la primera vez que se llamaba CallC, se recuperan los valores de los registros, segmentos y banderas del procesador al momento de la última vez que éste cedió el control desde el programa residente, los cuales fueron salvados previamente en una estructura de datos que llamaremos progstat, ubicada en el segmento de datos el cual en este punto ya se encuentra accesible, y se transfiere control al programa residente en el punto en que este fué suspendido anteriormente. Programas que utilizan acceso a video en esto punto es donde deben de salvar el cursor, modo de video y memoria del mismo para

poderlo reestablecer a la salida de la rutina al final del paso 6.

6.- Cuando CallC recupera el control por parte del programa residente en memoria, se salvan los valores de los registros, segmentos y banderas del procesador en este momento en la estructura progstat, de donde se recuperarán posteriormente la siguiente vez que se llegue al paso 5. Aquí es donde se reestablece el ambiente de video si es que fue modificado.

7.- Se reestablece el PSP del proceso original.

8.- Se recuperan los valores del segmento de stack y del apuntador al stack del proceso original.

9.- Se recuperan de este stack, los registros al momento de la interrupción y CallC regresa el control a quien lo llamó originalmente.

Con la idea de hacer más limpio el funcionamiento de este mecanismo, se adicionó una función intermedia que es la que le devuelve el control a CallC desde el programa en "C", a la que llamaremos _check(t), y que utiliza un parámetro t, de tipo entero cuyo significado es el número mínimo de interrupciones del reloj de la máquina en los cuales el control no es necesario que regrese al programa residente en memoria.

De esta manera, el programa en "C" le proporciona una indicación a las rutinas que manejan el control de los procesos acerca de su estado de actividad. Por otro lado, no es necesario un conocimiento muy preciso de dicho estado, ya que una llamada a _check() con parametro 1, siempre será segura para el programa residente.

Con el anterior mecanismo, el programa residente en memoria gana control del procesador a través de las interrupciones del sistema y lo regresa por medio de una llamada a la función _check(t). Con objeto de que el programa residente en memoria no tome control absoluto del procesador, es necesario que la función _check(t) sea llamada frecuentemente a lo largo de todo el programa, de la misma manera, los valores con los que se debe llamar a esta función deben de ser ajustados de manera que se logre una eficiente distribución del tiempo de procesador entre ambos procesos, sin embargo, las llamadas a _check(t) por otro lado, no deben ser tan frecuentes como para hacer que el tiempo necesario para el intercambio de procesos sea menor que el control que gana el programa residente, haciendo que la eficiencia de ambos procesos se degrade forzando que el procesador pierda su tiempo solo en labores de sincronización e intercambio entre procesos.

Si el programa residente es tal que puede terminar su ejecución y recomenzarla otra vez sin necesidad de removerse de memoria y reinstalarse nuevamente; por ejemplo en programas residentes activados por medio del teclado que se ejecutan totalmente cada vez que son activados sin necesidad de tener que cargarlos otra vez, es fundamental no tener dependencias de asignaciones estáticas a variables. Esto es debido a que los valores iniciales de dichas variables son asignados al momento de compilación de manera que si sus valores se alteran durante la ejecución del programa, nadie va a reestablecer sus valores la siguiente vez que se quiera ejecutar el mismo.

## 1.5 Utilización de archivos

Un punto sumamente delicado para este tipo de programas tiene que ver con el manejo de archivos de que va a hacer uso, en particular si algunos de estos se comparten con algún otro proceso. Este proceso puede ser la aplicación que corre el usuario o bien algún otro programa residente. Para poder hacer un manejo apropiado de estos archivos, es necesario abrirlos utilizando los atributos apropiados para manejar archivos compartidos que se proporcionan en el DOS en las versiones mayores o iguales a la 3.00 y utilizando la utilería SHARE que forma parte del DOS en tales versiones.

Los archivos propios entonces se pueden abrir en el modo de acceso apropiado, pero prendiendo el atributo de exclusividad de la operación, es decir negando acceso de escritura y lectura a otros procesos. Los archivos compartidos deben ser abiertos por ambos procesos usando los atributos de permiso de lectura o escritura según sea el caso.

La utilización de SHARE es necesaria puesto que es una extensión al DOS que hace que tales atributos (atributos de compatibilidad), sean interpretados correctamente y que surgió como una ampliación necesaria cuando DOS comenzó a soportar redes de computadoras en las que la posibilidad de manejar archivos compartidos por ejemplo en una máquina con recursos accesibles a toda la red ("SERVER"), es una necesidad.

El número total de archivos abiertos que puede manejar DOS está dado por el parámetro FILES del archivo config.sys, el valor por omisión de este parámetro es 8, que en general resulta insuficiente. Por su parte, cada proceso puede manejar hasta 20 archivos abiertos a la vez (se incluyen los que abre DOS para su uso como stder, stdin, stdout, stdaux y stdprn), esto tiene como consecuencia que para lograr la operación adecuada de los procesos residentes junto con las diferentes aplicaciones que se pudieran correr en la máquina, es necesario modificar el parámetro FILES de manera que tenga

un valor igual al menos a 20 veces el número de procesos que van a correr en la máquina simultáneamente, de esta manera, si no vamos a tener programas residentes que manejen archivos abiertos, podemos tener FILES=20, si vamos a tener un programa residente en memoria que haga manejo de archivos abiertos, FILES debe ser 40 y asi sucesivamente. Hay que tener en cuenta que DOS reserva cierta memoria para cada uno de estos FILES por lo cual la solución de poner un número exageradamente grande en el parámetro FILES de config.sys no es apropiada.

Dado que el programa residente en memoria debe tener una operación de lo más transparente posible, normalmente no se escribe a la pantalla por ejemplo con la función printf. Como además es un requerimiento que el tamaño del programa residente en memoria sea lo mas pequeño posible es conveniente por ejemplo que se programe evitando las llamadas a printf, sprintf, fprintf y similares que añaden una porción significativa de código principalmente por la inclusión del analizador gramatical ("parser") de sus argumentos y formatos. Por el mismo argumento, es deseable utilizar las operaciones de archivos en modo directo ("unbuffered") y no cargar el código de operaciones de archivos en modo indirecto ("buffered"). Para poder lograr lo anterior es necesario además modificar ligeramente las funciones _main y _exit de manera que no se haga referencia a los archivos estandar predefinidos ( stder, stdin, stdout, stdaux y stdprn).

---

* Lattice es marca registrada de Lattice, Inc.
* CP/M es marca registrada de Digital Research, Inc.
* Microsoft C Compiler es marca registrada de Microsoft Corporation.
* Turbo C es marca registrada de Borland International, Inc.

## Capítulo 2

# Instalación de programas residentes en memoria y manejo de interrupciones.

## 2.1 Terminación y permanencia en memoria

La instalación de programas residentes en memoria es una adición al DOS introducida a partir de la versión 2.00. Básicamente hay dos maneras de hacer que un programa se quede residente en memoria, la primera es por medio de la rutina de la interrupción 27h y la segunda por la función 31h del DOS. Entre estas, es preferible la segunda porque permite entre otras cosas que el programa que se va a dejar residente sea mayor de 64 Kilobytes, así como pasar información de regreso al sistema operativo. Esta llamada permite tambien manejar adecuadamente las interrupciones de error crítico y control-break ([IBMC87]). En general, la primera se utiliza normalmente cuando el programa que se va a dejar residente es alguna adición al sistema operativo como manejadores de dispositivos (device drivers), mientras que la segunda se utiliza para programas residentes de propósito general.

El proceso de instalación de un programa residente en memoria se puede describir brevemente como sigue: el programa se carga en memoria cuando se ejecuta como cualquier otro programa, se encarga de hacer el manejo apropiado de las interrupciones que va a utilizar, apuntando las que le van a servir como activadores a las rutinas correspondientes, hace el cálculo de la cantidad de memoria que va a utilizar que es igual al tamaño del código del programa mismo, más el espacio para obtención de memoria dinámicamente, más el espacio de stack y datos que el programa vaya a utilizar y el tamaño de su PSP, despues de lo cual, ejecuta la función 31h del DOS pasándole como parámetro el tamaño en memoria que se requiere del sistema operativo, con lo cual el programa termina su ejecución regresando control a DOS y habiéndose quedado residente en memoria.

La manera de calcular el tamaño de memoria necesario para la instalación del programa residente está basada en la forma en que se organizan los segmentos del procesador al momento de iniciar el programa y en general depende del compilador y modelo de memoria que se esté utilizando, en este caso vamos a describir este proceso para el modelo intermedio o "M" del compilador y que es como sigue.

La parte más baja de la memoria que es utilizada por el programa residente es el PSP, a partir del cual comienza el

segmento de código, a continuación se encuentran el segmento de stack y el segmento de datos traslapados, los cuales contienen en el orden indicado al área estática de datos, seguida de el área para obtención de memoria dinámica y el stack. Para esta organización de memoria, el cálculo se hace como sigue: se resta del segmento de datos el segmento de código, obteniendo como resultado el tamaño (en bloques de memoria de 16 bytes), del segmento de código del programa. Después se calcula el tamaño restante utilizando para ello el hecho de que conocemos el tope del stack, que incluye el area de datos y de memoria dinámica, con lo cual convertimos a bloques y lo sumamos al tamaño del código más el tamaño del PSP que son 256 bytes o sea 16 bloques y ésta es toda la memoria que utilizará el programa. En el caso de programas que utilizan los modelos de memoria que permiten mas de 64K de datos, se hace un cálculo similar considerando que en este caso el segmento de datos y el segmento de stack no necesariamente se traslapan y que puede haber varios de ellos.

Los programas residentes en memoria, una vez que ya se ejecutó la interrupción que los hace residentes no deben modificar la cantidad de memoria que tienen reservada para su uso dado que si hace una petición de expansión de tamaño en memoria, debido a la manera en que DOS hace el manejo de memoria, basado en el PSP, lo más probable es que escriban sobre el espacio en el cual se está ejecutando otro programa, ocasionando el mal funcionamiento de este. El caso es el

mismo cuando se desea obtener memoria dinámica directamente a partir de DOS por ejemplo para algún buffer temporal mediante la llamada 2Dh ya que esto ocasionaría la fragmentación de la memoria de DOS y el mal funcionamiento de los programas no residentes. Toda la memoria que podemos obtener con seguridad es la que fue reservada para tal propósito en el cálculo que se hizo previamente a la llamada 31h de DOS ("termina y permanece residente").

## 2.2 Utilización de funciones de DOS

Cuando un programa llama a una función del DOS, éste salva los registros de stack y establece su propio stack, al terminar la función, DOS reestablece el stack del proceso que lo invocó, de manera que si un programa residente en memoria interrumpe una función del DOS que fue llamada por otro programa y el programa residente en memoria hace a su vez una llamada al DOS, DOS sobreescribe los registros donde tenía guardada la información del stack del primer proceso; cuando la operación invocada por el programa residente termina, se continúa la anterior pero los registros de stack guardados no son los que corresponden al primer proceso haciendo que el sistema se pare ([MICC88]). Esta es entonces una condición que se debe poder manejar.

Internamente, DOS utiliza varios stacks y areas para salvar registros para poder realizar sus operaciones, uno de estos stacks es utilizado para las funciones bajas, 0h a 0Ch y otro diferente para las funciones altas que son las demás, de esta manera si el programa residente interrumpe una de las funciones bajas de DOS, es libre de utilizar las funciones altas y viceversa. DOS internamente utiliza una variable para indicar cuando esta ejecutando alguna de las funciones altas, la dirección de esta variable que llamaremos DosCritical se puede conocer a través de una función no documentada de DOS que es la 34h y que utilizaremos para el control de el reingreso de DOS, las demás funciones del DOS (las bajas), son utilizadas para acceso a display y teclado, y se pueden reemplazar por llamadas directas al BIOS (Basic Input Output System) que se encuentra en la memoria ROM (Read Only Memory) de la máquina. DosCritical también es conocida como InDos.

Este control de reingreso se puede hacer de otra manera atrapando nosotros la interrupción 21h que es la que proporciona las funciones de DOS, de manera que desde ahi nosotros mismos tengamos una variable que nos indique estar ejecutando una función de DOS, sin embargo, esto implica la utilización de una interrupción extra sin una verdadera justificación. Si tenemos que atrapar dicha interrupción por algún motivo, la variable que utilicemos para control de reingreso debe ser limpiada en el caso de que ocurra una interrupción 24h, o critical error, de manera que rutinas de

atención a esa interrupción que no regresen el control a DOS, no den la impresión de que DOS esta aún activo.

Algunas veces es posible utilizar las funciones altas de DOS aún si la bandera DosCritical está prendida. Este es el caso por ejemplo cuando DOS está esperando que se oprima una tecla, cuando esto ocurre, DOS ejecuta llamadas frecuentes a la rutina de interrupción 28h indicando que el reingreso es seguro, y que es utilizada para poder activar operaciones de DOS aún cuando éste esté activo, esto permite entonces que un programa residente sea activado también a través de ésta interrupción, evitando sufrir graves retrasos en su ejecución por ejemplo si DOS esta esperando respuesta del teclado y el usuario no teclea nada durante un lapso de tiempo largo ([HYMM87]).

## 2.3 Manejo de interrupciones

Los programas residentes en memoria pueden interferir con las operaciones a disco de otros programas si se activan en medio de una de ellas y realizan alguna operación que altere la sincronía del mecanismo del disco o interfiriendo con su posicionamiento, lectura ó escritura. El mecanismo de protección proporcionado por DosCritical sería suficiente para protegernos de esta posible interferencia en el caso que todas las llamadas a operaciones de disco fueran ejecutadas a

través de DOS. Este no es el caso, cualquier programa puede llamar directamente operaciones de disco a través del BIOS o de lenguaje ensamblador. Así pues se debe proveer un mecanismo extra para protegernos del caso anterior.

La forma de protegerse contra lo anterior es atrapando la rutina de interrupción de acceso a disco de manera que prenda una bandera antes de ejecutar la operación y la apague una vez concluida esta. Siendo este el caso, el programa residente en memoria verificarÁ el valor de esta bandera de manera que si se encuentra prendida no se active. Como esta interrupción regresa condiciones de error en las banderas del procesador, hay que tener cuidado de manejarlas correctamente, haciendo la modificación en el lugar adecuado del stack, de lo contrario, programas que utilicen directamente esta interrupción y chequen sus condiciones de error dejarán de funcionar correctamente. No es posible protegerse contra programas que accesan directamente al controlador de disco, aunque en general nadie aparte de BIOS lo hace.

La rutina de acceso a video proporcionada por BIOS no puede ser interrumpida por el programa residente en memoria, ya que este puede iniciar sus propias llamadas a video causando mal funcionamiento de la máquina. Por lo anterior, se debe utilizar una técnica parecida a la descrita para las operaciones a disco. En este caso, como esta rutina usa en

general todos los registros, incluso el "BP" ([IBMC87]), es preciso que la llamada se haga sin dependencias del valor de éste como es el caso de las funciones en "C" que se les pasan parámetros. Esta precaución es necesaria siempre que el programa residente haga acceso al video a través de BIOS.

Las rutinas de interrupción son accesadas por medio del mecanismo mencionado a continuación, que es el proporcionado por el procesador 8086. En la parte más baja de la memoria, se encuentra una tabla de vectores de interrupción, de cuatro bytes cada una, que son en realidad las direcciones de cada una de las rutinas de atención a cada interrupción. Cada vez que el procesador tiene que ejecutar una llamada a una interrupción, pone las banderas en el stack y hace una llamada tipo FAR (que pone en el stack el segmento de código y el apuntador a la siguiente instrucción) a la dirección apuntada por el vector de interrupción correspondiente. La rutina de interrupción devuelve el control al proceso que la llamó, una vez que terminó sus funciones, por medio de la instrucción "IRET", que se encarga de regresar el control a la dirección de retorno y de sacar del stack a las banderas, o bien por medio de una instrucción "ret 2" en los casos en que se desee regresar el nuevo valor de las banderas del procesador.

La manera en que se atrapan las interrupciones es entonces utilizando el anterior mecanismo. Hay dos formas de

hacerlo, la primera forma es simplemente reemplazando el valor en memoria de la dirección de atención a la interrupción por la dirección de la nueva rutina de atención a la interrupción. Sabemos que la dirección en que se encuentra el vector de interrupción número intno es 0:(4*intno) ([INTC85]), así que se reemplaza el valor que ahí se encuentra por la dirección de nuestra rutina de atención a la interrupción en el formato apropiado, que es posición relativa, segmento de código. La segunda manera es utilizando una función que proporciona DOS para tal propósito. Esta es la función 25h, a la cual se le pasan como parámetros la interrupción que se desea atrapar así como la dirección de la nueva rutina de atención a la interrupción.

En el caso en que se atrapan interrupciones que tienen un uso específico (e.d. no son interrupciones disponibles para el usuario), es crucial el respetar la funcionalidad básica proporcionada por la interrupción que estamos reemplazando. Para realizar lo anterior, la manera más sencilla es que desde la nueva rutina de atención a la interrupción se haga una llamada a la antigua, para lograrlo es necesario que antes de reemplazar el vector de interrupción, salvemos el anterior valor del mismo, el cual además será utilizado cuando se descargue de memoria nuestro programa. El detalle importante a considerar aquí es que la dirección que tenemos guardada es la de una rutina de atención a interrupción, por lo cual al llamarla se debe hacer de manera que sea simulada

una interrupción, es decir, es necesario poner las banderas del procesador en el stack antes de hacer el llamado a la rutina. Además en el caso de funciones que regresan códigos de error en las banderas del procesador, hay que encargarse del manejo de la copia de las banderas que se puso en el stack al momento en que la interrupción fue provocada.

Cuando se entra en una rutina de atención a una interrupción, las banderas del procesador pueden cambiar con respecto al valor que tenían antes de la llamada y que es el que está en el stack, cuando hacemos la simulación de la interrupción original, las banderas que se deben poner en el stack antes de la llamada a la rutina original son precisamente aquellas con que nuestra rutina de atención a la interrupción fue llamada. En otras palabras, se tiene que obtener una copia de la que está en el stack. La forma de hacerlo es de la siguiente manera:

```
push    bp              ; Salvamos bp para poderlo usar
                        ; en la rutina como apuntador al
mov     bp, sp          ; stack.

push    ax              ; Salvamos el registro ax.
mov     ax, [bp+6]      ; Aqui deben estar las banderas.
push    ax              ; Se ponen en el stack.
popf                    ; Se sacan del stack.
pop     ax              ; Se recupera el registro.
```

Sin embargo, debido a un error ("bug") en el microcódigo de algunas de las primeras versiones del procesador 80286, utilizar la instrucción popf puede ocasionar cierto mal funcionamiento, descrito en el manual de referencia técnico de la IBM-AT ([IBMC87] página 9.8) , de manera que para evitarlo se debe de reemplazar la instrucción popf por una secuencia de instrucciones equivalente como lo es la siguiente:

```
    jmp     $+3             ; Brinca el iret.
xxx:                        ; Etiqueta para regresar.
    iret
    push    cs              ; Simulando llamada far.
    call    xxx             ; Ahora el iret hará el popf
                            ; por nosotros.
```

Como se mencionó anteriormente, la manera más sencilla de preservar la funcionalidad básica de la interrupción atrapada es llamar a la antigua rutina desde el interior del nuevo código de atención a la interrupción. Esa llamada se puede hacer de varias maneras dependiendo de cual sea el tipo de control que necesitamos recobrar de la interrupción. Si sólo necesitamos la interrupción para hacer algún procesamiento previo mínimo y despues ejecutamos el código original, esto

se puede hacer de la siguiente manera:

```
...                     ;
...                     ; Código del procesamiento previo
...                     ; debe dejar el stack en el
...                     ; estado en que lo encontró.
                        ; En IntForward está la dirección
                        ; de la rutina de interrupción
                        ; original.
pushf                   ; Pone las banderas en el stack
                        ; para simular una interrupción.

push    word ptr cs:IntForward+2
push    word ptr cs:IntForward

iret                    ; Regresa control a la rutina de
                        ; interrupción original.
```

De esta manera, el control regresará al programa que llamó la interrupción por medio de la ejecución del iret que la rutina original de atención a la interrupción debe ejecutar a su término.

En caso de que además se necesite algún procesamiento posterior a la ejecución de la rutina actual de interrupción,

la manera de hacer la llamada es la siguiente:

```
...                         ;
...                         ; Código del procesamiento previo
...                         ; debe dejar el stack en el
...                         ; estado en que lo encontró.
                            ; En IntForward está la dirección
                            ; de la rutina de interrupción
                            ; original.
pushf                       ; Pone las banderas en el stack
                            ; para simular una interrupción.

call    dword ptr cs:IntForward

...                         ;
...                         ; Código de procesamiento
...                         ; posterior.

iret                        ; Regresa control a la rutina de
                            ; interrupción original.
```

De esta manera, después de la ejecución de la rutina original, el control regresará a nuestra rutina para el proceso posterior, y después al programa que llamó la interrupción, por medio de la ejecución del "IRET" que nuestra rutina efectua a su terminación.

El reloj de la máquina genera una interrupción al nivel físico de la máquina (hardware) que es la interrupción 08h, la cual se genera aproximadamente 18.2 veces por segundo, y que tiene ciertas funciones básicas como son el chequeo de algunos puertos, la sincronización de los manejadores de disco, actualización de los valores de fecha y hora del DOS, etc., además llama a otra interrupción, esta vez al nivel lógico de la máquina (software) que es la interrupción 01ch, que puede ser utilizada como contador de pulsos del reloj por los programas que estén corriendo en la máquina.

Los programas que se activan por medio de alguna de estas interrupciones tienen que manejarse con cierto cuidado pues de otra manera se puede generar todo tipo de funcionamiento erróneo. Típicamente que el reloj de DOS se retrase, que los controladores de disco pierdan su sincronía, no ocurra refresco de memoria por parte del "DMA" ("Direct memory access") o bien la máquina simplemente deje de funcionar. En general, lo más adecuado es atrapar la interrupción de hardware de manera que se ejecute la rutina original antes que el código sustituto, previniendo así los diversos mal funcionamientos que pudieran provenir por no ejecutar en el momento adecuado el código original de las interrupciones 08h ó 1ch. Se debe manejar las condiciones de suspensión y reingreso de nuestro código siempre que su ejecución pudiera tomar más tiempo que el necesario antes de que otra interrupción 08h ocurra.

La condición de reingreso es sencilla de manejar, para hacerlo es necesario únicamente tener acceso a una variable booleana que hay que encender cada vez que se entra al proceso residente y se apaga cuando se regresa el control a el proceso en curso que puede ser DOS, alguna aplicación o incluso algún otro programa residente. Así, cada vez que se cumpla la condición de activación del proceso, se checa el valor de esta variable para saber si éste esta ejecutando actualmente, en cuyo caso no se entra. Un punto fino en este esquema es que la variable que se utilice para indicar si el proceso está activo debe ser accesible al momento de la interrupción, esto pone la restricción de que dicha variable pertenezca al segmento de código de la rutina de atención a la interrupción.

### 2.4 Despachador de procesos

Utilizando la interrupción del reloj de la máquina como mecanismo de activación del programa residente en memoria es posible simular multiproceso en DOS, la idea aquí es poner en la rutina de manejo de la interrupción 08h, un despachador de procesos que se encargue de distribuir el tiempo de proceso entre el programa residente en memoria y los demás procesos que pudieran estar activos. Lo anterior es necesario pues de

otra manera, una vez activado el programa residente en memoria acapararía toda la atención del procesador, que no es el efecto deseado si se quiere simular multiproceso.

El despachador se encarga en realidad de decidir a que proceso le toca el siguiente intervalo de tiempo de procesador, donde cada intervalo tiene la duración de un pulso del reloj. Una manera de hacer lo anterior es la siguiente: Se mantiene un contador del número de pulsos del reloj que han transcurrido desde la última activación del programa residente, se utiliza una variable que llamaremos "lambda", en la cual se guarda el número de pulsos del reloj que se desea tengan los demás procesos antes de activar al programa residente. Cuando la condición de activación del programa residente se cumple (en este caso cuando ocurre una interrupción 08h) se compara el contador contra el valor de lambda, si éste es mayor, se vuelve a hacer cero el contador y se cede control al programa residente.

La variable lambda se inicializa a un valor que depende de cual es el tiempo inicial que deseamos que transcurra antes de que el programa residente en memoria comience su ejecución, que puede ser cero por supuesto, lambda toma sus valores subsecuentes por indicación del programa residente en memoria, que cuando regresa el control a otro proceso se encarga de poner en lambda el número de pulsos que desea transferir el control a los demás procesos. Como se indicó

en la sección 1.4, el control se transfiere del proceso residente a través del una llamada a la función _check(t), de hecho, lambda toma su valor del parámetro t proporcionado a _check(t). La función _check(t) tiene entonces dos objetivos básicos y necesarios. Estos son, asignar a lambda el valor adecuado y ceder el control a los demás procesos y salvar el contexto de ejecución del programa residente para que la siguiente vez que éste se active, continúe su ejecución normalmente.

Para clarificar la manera en que trabaja el esquema propuesto elaboré el diagrama 1, que representa un intervalo hipotético de ejecución. Considérese que el tiempo transcurre en el eje vertical, y el eje horizontal representa los procesos que está ejecutando el procesador.

La interpretación del diagrama es la siguiente. En el punto 1, el procesador se encuentra ejecutando algún proceso como puede ser por ejemplo command.com ó algún otro programa de aplicación y el programa residente esta inactivo, suponemos que no se está ejecutando ninguna operación crítica de DOS. En el punto 2, se cumple la condición de activación del programa residente, por ejemplo por la ocurrencia de una interrupción 08h, de manera que éste toma control de la máquina. Esto se representa por un salto en el diagrama,

este nivel representa nuestro proceso que es el punto 3. El punto 4 representa una interrupción que ocurre cuando nuestro proceso se está ejecutando, por ejemplo un acceso a disco o un pulso del reloj, la condición de reingreso está bien manejada y por tanto no se intenta activar el proceso residente ya que éste se encuentra activo. Sin embargo, la funcionalidad básica que dicha interrupción proporciona se obtiene. En el punto 5, el control regresa al programa residente. En los puntos 6 y 7 se repite un ciclo similar al anteriormente descrito.

El punto 8 indica una llamada a DOS por parte de nuestro programa, dicha llamada no podría ejecutarse si el programa residente se hubiese activado cuando DOS estaba ejecutando una operación critica ó de reingreso inválido. El punto 9 representa el nivel en que la llamada a DOS ejecuta. En 10 ocurre nuevamente un pulso del reloj, el cual regresa el control a DOS en 11 para que complete su operación. En 12 aparece otro pulso del reloj que cuando termina regresa el control a DOS que a su vez termina regresando al punto 13 que es nuevamente al nivel dé nuestro programa residente. El punto 14 es una llamada a _check que regresa el control al proceso interrumpido por la activación del programa residente.

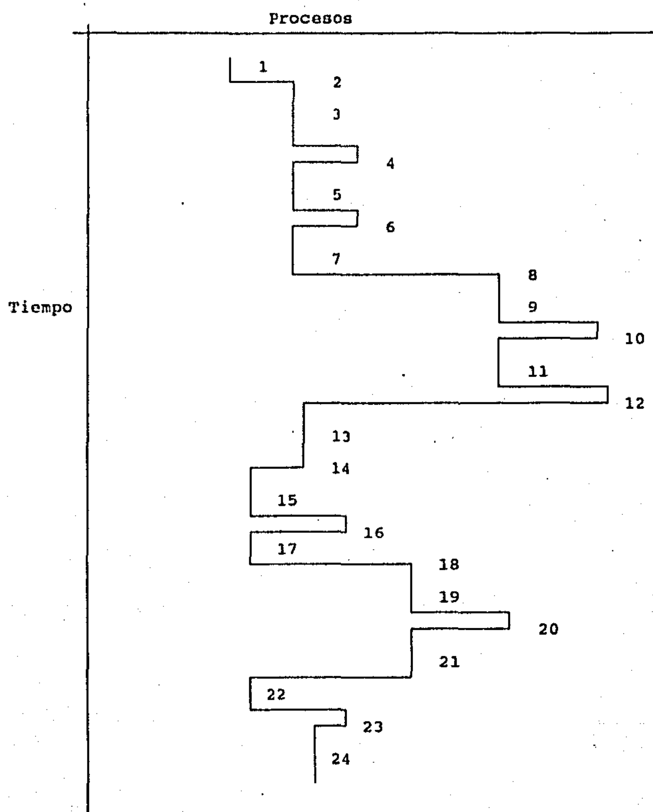


Diagrama hipotético de ejecución de procesos.

DIAGRAMA 1.

En el punto 16, ocurre un pulso del reloj, sin embargo, el contador de pulsos transcurridos desde la última llamada a _check es menor aún que el valor actual de lambda (que fue puesto en el punto 14 con la llamada a _check), así que el programa residente no retoma control y se regresa en 17 después de la ejecución de la rutina original de atención a la interrupción 08h. En 18, ocurre una operación crítica de DOS, la cual empieza su ejecución en el punto 19. En el punto 20 ocurre un puldo del reloj. Podemos suponer que en este momento, han transcurrido más de lambda pulsos desde la última vez que se llamó a _check, sin embargo, como esta en ejecución una llamada crítica de DOS, se inhibe la activación del proceso residente. En este caso, el contador de pulsos inactivo del programa residente se sigue incrementando hasta que éste retoma el control.

En el punto 21, DOS retoma el control para terminar su operación y regresa el control en 22 al proceso original. El siguiente pulso del reloj, representado por el punto 23, satisface las condiciones de activación por lo cual se ejecuta la rutina de reloj de DOS y el control vuelve otra vez al programa residente para su continuación, donde la rutina CALLC descrita en el anterior capítulo se encarga de reestablecer su ambiente adecuado de ejecución.

## 2.5 Comunicación entre procesos

Es necesario proporcionar alguna manera de interactuar con el programa residente en memoria, por ejemplo para indicarle diferentes tipos de operación tales como que suspenda momentáneamente su ejecución, removerlo de memoria, o que cambie sus opciones de configuración iniciales por otras. Debido a que el programa residente tiene referencia a una copia del medio ambiente local ("environment") que puede ser diferente a la que está actualmente en funcionamiento, dicha comunicación no se puede llevar a cabo por este medio. Para lograr esta comunicación se cuenta con varias opciones descritas a continuación.

Una de estas opciones es la comunicación entre programas utilizando el mecanismo de interrupción que tiene el sistema, es decir, el programa residente en memoria atrapa uno de los vectores de interrupción libres que están destinados para su uso por diferentes programas del usurario ( 60h a 67h ). La rutina que se instala para manejar esta interrupción tiene que proporcionar diferentes funciones para cada una de los casos en los cuales se desea establecer comunicación con el programa residente. La selección de cuál de estas funciones se va a ejecutar queda determinada por el valor de uno de los registros al momento de llamar la interrupción que sirve como código de acceso.

El manejo de estas interrupciones para los programas del usuario es diferente a la de las demás interrupciones por dos razones. Una de ellas es que estos vectores pueden estar desocupados, en cuyo caso apuntan inicialmente a la dirección 0000:0000 donde no hay ninguna rutina de atención a interrupción. A diferencia de los otros que al menos apuntan a alguna dirección que contiene una rutina de atención ficticia como puede ser un iret. De esta manera, si se ejecuta el procedimiento descrito al principio de este capitulo como rutina de interrupció, se generará una llamada a esa dirección dentro de nuestra rutina, en general contendrá instrucciones inválidas que harán que el procesador simplemente se pare. Ahora bien, en el caso en que algún otro programa esté usando el vector de interrupción que seleccionamos, éste contendrá la dirección de la rutina proporcionada por el otro programa y aquí aparece la otra diferencia, que es que no sabemos nada acerca de que haga dicha rutina y se puede ocasionar mal funcionamiento cuando en una llamada se pasa control a esa rutina sin que sea el caso que es necesario llamarla.

Mostramos dos maneras resolver dicho problema. Una posibilidad es que al momento de instalación, el programa residente se de cuenta de cuales interrupciones no estan en uso, identificables por que contienen el valor 0000:0000 y elegir una de ellas para nuestros propósitos. Si este es el

caso, al instalar el programa hay que poner una firma o identificación en algún lugar de la memoria a una distancia fija de la dirección de la llamada a la interrupción de manera que sea posible reconocer que ésa es la rutina esperada. Este enfoque puede fallar por ejemplo en el caso en que todas los vectores de interrupción del usuario se encuentren ocupados, o bien cuando un programa posteriormente ejecutado se encima en el vector de interrupción que elegimos, haciendo que se pierda la dirección de nuestra firma.

La segunda manera es proporcionando un número mágico en alguno de los registros que usaremos como llave y que nos indicará si la llamada a la interrupción esta dirigida a nuestro programa en cuyo caso ejecutamos la rutina adecuada y un "IRET", y en el caso en que el valor del registro llave no cheque ejecutamos, dependiendo si al momento en que instalamos el vector de interrupción estaba ocupado llamamos a la rutina que ocupaba el vector de interrupción previamente o bien ejecutamos simplemente un "IRET". Se puede regresar otro número mágico en alguno de los registros para asegurarse que la ejecución de nuestra rutina de atención fue exitosa.

El uso combinado de los dos enfoques es mejor porque minimiza la posibilidad de interferencia entre diferentes programas. El número de interrupción del usuario utilizado por el programa residente lo determina el programa de

aplicación que se desea comunicar con él, corriendo una rutina que hace una búsqueda de la firma del programa residente en las diferentes interrupciones. Desde luego la interferencia entre programas que manejen interrupciones sólo podrá ser evitada en general si hubiera un manejo estándar de interrupciones, ya que por ejemplo, el segundo enfoque no sirve de nada si un programa posterior simplemente llega y pone su rutina de atención sin tomar en cuenta si había alguien instalado anteriormente. De cualquier forma, un medio de comunicación de este tipo debe ser proporcionado al menos para la remoción de memoria del programa, ya que de esa manera se puede tener acceso fácilmente a los valores de las rutinas originales de atención a las interrupciones usadas y la remoción de memoria del programa no queda sujeta a la integridad de algún archivo que contenga dicha información.

DOS proporciona un mecanismo general para que los programas de aplicación verifiquen la presencia de ciertos programas residentes en memoria (en particular de Print y Share). Esto lo hace por medio de la interrupción 2Fh (multiplexor), a la cual cada programa residente que se instala se encadena con una rutina que informa acerca de la instalación del programa y le asigna un número de identificación. Este mecanismo sin embargo, es inadecuado porque los números de identificación son arbitrarios, de manera que si hay algún otro programa residente instalado que utiliza el mismo número de identificación, no se podrá realizar la instalación sino cambiando dicho número. Esto

con la consecuencia de que otros programas que deseen interactuar con el programa residente y chequen su presencia por medio de este mecanismo, pueden obtener confirmaciones incorrectas de instalación ó ausencia del programa residente con consecuencias obvias.

Otra forma de intercambiar información con el programa residente es por medio de un archivo que contiene cierta estructura y al cual se tiene acceso para lectura y escritura por parte tanto del programa residente como del programa de aplicación con el que se desea comunicación. La manera de asegurar la integridad de dicha información es abriendo dicho archivo para efectuar operaciones de lectura y escritura y poniendole las restricciones de acceso de lectura y escritura por parte de otros programas proporcionada por SHARE y haciendo que todas las operaciones a este archivo sean atómicas, es decir, tanto las lecturas como las escrituras al mismo tienen un protocolo que abre el archivo al principio de la operación, lo mantiente abierto durante la misma y lo cierra cuando se completa, impidiendo de esta manera que la información sea corrompida por otro de los procesos. Hay que asegurar además que dichas operaciones no hagan transferencia de control a otras rutinas mientras no se hallan completado. Por este medio es posible pasarle al programa residente información del tipo de configuración que se desea, o de ciertos valores por omisión que debe utilizar para algunas de sus operaciones.

Hay ciertas operaciones que necesitan de una mayor sincronización entre los diferentes procesos, es decir no pueden realizarse en cualquier punto de la ejecución del programa residente. Para lograr ese tipo de coordinación es necesario poner un mecanismo de semáforos con el cual se pueda controlar externamente el flujo de ejecución del programa residente hasta que este llega a un punto en el cual se garantiza la seguridad del tipo de operación que es requerido. El programa residente deberá estar constantemente checando la posibilidad de que se le pida una operación de este tipo. Debido a esto, el punto idóneo para hacer ese chequeo es en alguna función que se llama a intervalos muy frecuentes. Un ejemplo de este tipo de función es la función CheckTime que se discute en el siguiente capítulo y que es la que entre otras labores ejecuta la llamada a _check para la transferencia de control a los demás procesos.

El programa externo de aplicación que se quiere comunicar con el programa residente lo manda a un lugar seguro por medio de una función que podemos llamar StopBg y que consistiría en el encendido de un primer semáforo, en ese momento el programa de aplicación empieza en un ciclo de chequeo de un segundo semáforo que el programa residente encenderá cuando llegue al punto en cuestión. El programa residente se entera de que se quieren comunicar con él al encontrar el primer semáforo encendido, termina sus operaciones con seguridad y se mueve a un lugar seguro donde

enciende el segundo semáforo y empieza un ciclo checando hasta que el primer semáforo este apagado de nuevo.

Cuando el programa de aplicación encuentra al segundo semáforo encendido, se comunica con el programa residente por ejemplo a través del mecanismo de archivo descrito anteriormente, o a través de memoria compartida (accesada a través de alguna interrupción) y después libera el primer semáforo por medio de una operación que podemos llamar ReleaseBg y comienza otro ciclo ahora checando por el segundo semáforo que le indicará que el programa residente ejecutó la operación que le fue pedida, el programa residente entonces continúa su ejecución determinando cual es la operación a ejecutar a partir del archivo de comunicación. Ejecuta dicha operación y libera el segundo semáforo de manera que el otro programa pueda proseguir su ejecución. Opcionalmente puede utilizar el mismo archivo de comunicación para regresar algún código de error al programa de aplicación.

La ventaja primordial que se obtiene con el anterior mecanismo de comunicación entre procesos (usando semáforos), comparado con el de comunicación a través de memoria, es que si los archivos de semáforos y de comunicación se encuentran en una red local de computadoras, se puede interactuar de la manera descrita anteriormente desde una máquina que no sea necesariamente en la cual está instalado el programa residente. Lo anterior no sería posible si toda la

comunicación fuera por medio de memoria como es el caso cuando la comunicación es realizada por via de interrupciones.

En el apéndice "C" describo como estructurar un mecanismo de comunicación entre programas por medio de semáforos de manera que se evita el abrazo mortal ("deadlock"). Se incluye un esqueleto de código de la estructura de los programas que se comunican. Incluye el caso de corrutinas que se explica en el capitulo 3.

# Capítulo 3

# Utilización de Corrutinas

## 3.1 Conveniencia del uso de corrutinas

En el capítulo anterior se describe la mecánica con que el programa residente gana control de la máquina ejecutándose de manera "paralela" con los demás procesos activos. Si se desea tener dos procesos independientes corriendo al mismo tiempo además de alguna posible aplicación, esto se puede hacer de una manera sencilla corriendo dos programas residentes al mismo tiempo, cada uno de ellos encargado de uno de los procesos.

El problema que esto representa es doble. Por un lado, el deterioro en la capacidad de proceso útil de la máquina se ve incrementado conforme el número de procesos simultáneos aumenta, esto queda muy claro en nuestro caso en que conforme este número aumenta, la cantidad de procesamiento que le corresponde a cada uno de los procesos se reduce no sólo por que tiene que dividirse entre más de ellos, sino que además

el porcentaje de tiempo que el procesador utiliza haciendo el cambio de contexto entre los diferentes programas se incrementa.

Por otro lado, gran parte del código de cada uno de los programas residentes esta repetido en los otros, por ejemplo todas las librerias y funciones comunes de estos procesos utilizan, que en el caso en que los procesos estén muy relacionados entre si, puede llegar a ser la casi totalidad del código, así como las variables estáticas. Esto representa un problema serio puesto que la cantidad de memoria que se necesita para la ejecución de los diferentes procesos empieza a multiplicarse dejando cada vez menos memoria para las diferentes aplicaciones.

Hay dos formas de resolver este problema. La primera sería por ejemplo instalar un manejador ("driver"), al que se tenga acceso por medio de alguna interrupción y con el cual se accese todo el código común a los diferentes programas residentes. Este enfoque sin embargo da al traste con las ventajas de la interfaz con el lenguaje de alto nivel a menos que se desarrolle otro nivel intermedio que sirva de puente entre las rutinas del manejador y la programación en el lenguaje de alto nivel. Otra desventaja de esta solución es que necesita incluir mucho código para poder ser lo suficientemente general y hay casos de programas residentes muy simples para los cuales el pagar ese precio es

inaceptable.

La segunda manera es por medio de la utilización de corrutinas, esto es diseñar diferentes rutinas para cada uno de los procesos que se desea se ejecuten simultáneamente y proporcionar un medio para que entre ellas se transfieran el control unas a las otras retomándolo en el punto justo después de la última transferencia de control que hicieron ([WULW81]). De esta manera, no hay código duplicado ocupando innecesariamente espacio en memoria y además también se reduce el tiempo que el procesador pierde reestableciendo los diferentes estados de los procesos, ya que no es necesario dentro del mismo programa reestablecer sino los valores de los registros y segmentos para transferir el control a la otra corrutina.

Para la creación de las corrutinas, es necesario que cada una de ellas tenga su propio stack donde guarda el contexto de su ejecución y que utilizan los subprocesos mandados llamar por ella. Con objeto de proporcionar a cada uno de estos subprocesos su propio stack, el primer enfoque posible es declarar arreglos en el área de datos del programa para que cada proceso utilice el que le corresponda, aquí aparece el problema de que muchas de las funciones nativas del compilador incluyen código de chequeo de desbordamiento de stack ("stack overflow"), y la forma en que este chequeo se lleva a cabo normalmente a través de una comparación del

apuntador del stack con la siguiente localidad accesible de memoria para asignación dinámica, si ahora nuestro apuntador al stack cae en alguno de los arreglos, que sería el caso en que una de las corrutinas esté ejecutando, cuando se manda llamar a ésas funciones del compilador, un falso desbordamiento de stack se detecta y el programa se termina. Entonces, lo que se tiene que hacer es simplemente proporcionar al verdadero stack espacio suficiente para los stacks de cada una de las corrutinas y particionar este stack en varias partes, una para cada uno de los procesos necesarios.

Por supuesto que ahora ya no va a ser posible detectar una condición real de desbordamiento de stack de manera que hay que tener cuidado de proporcionar la cantidad adecuada de stack que va a utilizar cada proceso y ser muy cuidadosos en el uso de este.

Con objeto de determinar los requerimientos de stack de cada una de las corrutinas, es posible hacer un análisis de éstas aunque no es estrictamente necesario. Una manera sencilla de encontrar cuáles son las cantidades necesarias para cada uno de los procesos es totalmente empírica aunque bastante efectiva, esta basada en el siguiente mecanismo. En la rutina que hace la asignación inicial del stack (en el caso aquí descrito, es c.asm modificado), al momento de obtener el espacio para el stack del programa, se llena todo

el espacio del stack con algún valor elegido de antemano por nosotros tal como OFFh, se asignan valores grandes a los tamaños de stack requeridos por cada corrutina y se deja correr el programa durante un largo período, asegurandose de que el programa ejecute en toda su funcionalidad.

De esta manera, los valores con los que inicializamos el stack se van alterando y podemos tener una idea más clara de cual es el tamaño necesario por cada una de nuestras corrutinas observando los valores en memoria del espacio correspondiente al stack por medio de debug.com o alguna utilería semejante, ya que podemos conocer cual es su posición original en memoria.

Es preciso dejar cierto margen de tolerancia adicional en el cálculo de los tamaños de stack por dos motivos. El primero es que si nuestro proceso queda manejado por sucesos externos a este como la existencia de ciertos archivos, entradas de datos por algún puerto, etc., no podemos asegurar que durante la ejecución de la prueba que hicimos los stacks llegaron al máximo posible de utilización. El segundo es que hay que recordar que las rutinas de interrupción del sistema operativo y las funciones de BIOS utilizan muchas veces el stack del proceso que las llama para ejecutarse, y en el caso de las interrupciones de hardware, nunca sabemos en que momento pueden ocurrir.

### 3.2 Intercambio de control

Para que las corrutinas se sincronicen y empiecen a trabajar adecuadamente, es necesario construir unas funciones en lenguaje ensamblador que se encargan del intercambio de contextos entre las corrutinas. Llamaremos a estas funciones _func1 y _func2, la diferencia entre ellas es que _func1, antes de cambiar contextos, inicializa el stack destino y es utilizada sólo la primera vez que se va utilizar cada uno de los nuevos stacks. _func2 por su parte es la que normalmente hace el cambio de contextos de ejecución, aunque asume que los stacks fuente y destino estan propiamente acondicionados para un intercambio exitoso (es decir fueron inicializados por _func1 ó por el compilador. Ambas utilizan los mismos parámetros, que son la dirección de un apuntador, en el cual se va a guardar el valor del apuntador al stack del proceso que se está suspendiendo y un apuntador al stack que se intenta reestablecer en el caso de _func2 ó a un stack vacio en el caso de _func1.

Las anteriores funciones son llamadas siempre desde un procedimiento que llamaremos CheckTime(), descrito en la sección 3.3 y ejemplificado en el apéndice "C", que es el encargado de decidir a cual de las corrutinas hay que transferirle el control, estas funciones trabajan

esquemáticamente como se describe a continuación.

La funcionalidad de _func1 es la siguiente:

1.- Guarda en unas variables del segmento de código los valores de sus parámetros así como los valores de la dirección de retorno que están en su stack. Dependiendo de cómo quedó generado el código para CheckTime en función de sus variables locales es posible que el stack se tenga que ajustar tomando esto en cuenta.

2.- Escribe en el stack sobre las localidades donde se encuentran sus parámetros y dirección de retorno para simular el ambiente como si se encontrara dentro de la función CheckTime que es la que lo mando llamar, de manera que cuando el control vuelva a este stack a través de una llamada a CheckTime por otra de las corrutinas, se encuentre con las direcciones de retorno adecuadas.

3.- Salva todos los registros del procesador en el stack, a excepción de "CS", "IP", "SS" y "SP" que se recuperarán cuando así sea necesario de las variables del segmento de código donde fueron guardadas, "SS" no cambia.

4.- Se guarda el valor del apuntador al stack en este momento en la variable cuya dirección fue proporcionada como primer parámetro a la función, de manera que CheckTime tenga acceso a estos valores para futuras llamadas a _func2.

5.- Se cambia el valor del apuntador al stack por el proporcionado como segundo parámetro en la llamada, que corresponde a un stack vacio.

6.- Se empujan al stack la dirección de retorno de la llamada a la función que previamente salvamos en el paso 1 y se ejecuta un retf, con lo cual el stack en uso ahora es el nuevo. El registro "BP" se coloca al principio del nuevo stack siguiendo las convenciones de llamadas de funciones del lenguaje "C".

Así pues, despues de una llamada a _func1 para usar uno de los stacks nuevos, regresamos al punto en que fue llamado CheckTime pero ahora con el nuevo stack, de manera que ya podemos llamar a la corrutina correspondiente utilizando dicho stack. Se deben proporcionar los medios para asegurarse que _func1 siempre se llama para inicializar un nuevo stack y que esta llamada sólo ocurre una vez.

La manera de trabajar de _func2 es:

1.- Se ajusta el stack como el de CheckTime, de manera que los valores de retorno se encuentren en la posición esperada.

2.- Se salvan los registros del procesador en el stack para que cuando eventualmente se regrese a utilizar este stack nuevamente se recuperen de él el ambiente de ejecución de la corrutina.

3.- Se guarda el valor del apuntador al stack en este momento en la variable cuya dirección fue proporcionada como primer parámetro a la función, de manera que CheckTime tenga acceso a estos valores para las subsecuentes llamadas a _func2.

4.- Se cambia el valor del apuntador al stack al proporcionado como segundo parámetro en la llamada, que corresponde al stack de la rutina a la que se va a transferir control.

5.- Se recuperan del stack los valores de los registros y se ajusta el stack pointer de acuerdo a CheckTime si esto es

necesario.

La clave para que las corrutinas funcionen eficientemente es que todas ellas tengan frecuentes llamadas a CheckTime de manera que éste determine cuál es la siguiente a la cual hay que transferirle el control.

### 3.3 Despachador de corrutinas

La función despachadora de corrutinas, CheckTime utiliza un parámetro que indica durante al menos cuanto tiempo la corrutina que lo mando llamar no necesita el control de la máquina. Basado en esta información y en el momento cuando fueron llamadas por última vez cada una de las corrutinas, CheckTime determina a quien hay que cederle el control a continuación. Los candidatos para esto son las corrutinas del programa, y en el caso de programas residentes en memoria, la rutina despachadora de procesos _check(t). Esto lo consigue de la siguiente manera:

1.- Calcula el número de pulsos del reloj transcurridos desde la última vez que fue llamado. Mantiene contadores del tiempo que tiene cada una de las corrutinas sin ser llamada, esto no tiene que ser demasiado exacto, dado que sólo sirve

para que CheckTime se de una idea de que corrutina es la mas indicada para ejecutar a continuación.

2.- Si se tienen N corrutinas, las primeras N-1 veces que CheckTime es llamado, ejecuta la llamada a _func1 con los valores de sus apuntadores a stack correspondientes para ser inicializados y manda llamar a la corrutina asociada. El stack del programa principal (_main()) que también se considera como corrutina dentro de esta interpretación, no tiene que ser inicializado por CheckTime dado que el compilador ya se encargo de eso, por lo cual sólo son N-1 llamadas a _func1.

3.- En este punto checa por posibles peticiones de ejecución de algúna función por parte de programas externos de aplicación que así lo indiquen por medio del mecanismo de semáforos. Si éste es el caso, comienza el proceso de llevar la ejecución del programa a un lugar seguro en el cual se pueda suspender el proceso, para lo cual es posible que haya necesidad de salir de todas las corrutinas activas. Esto se logra encendiendo una variable global que las corrutinas checan a intervalos frecuentes y que indica que se deben de suspender de una manera limpia. A su vez, cada una de las corrutinas suspendidas, al terminar enciende una variable para indicar que su suspensión fue completada. El sistema debe quedar en posibilidad de reanudar las corrutinas pendientes posteriormente.

4.- Si ya todas las corrutinas tienen inicializado su stack, le cede el control a la corrutina que determinó que es la siguiente en ser ejecutada, mediante algún algoritmo que puede ser basado en la información acerca de la última vez que las corrutinas tuvieron el control del procesador y que parámetro le enviaron a CheckTime en aquella ocasión; o bien se puede desarrollar un algoritmo más complejo, basado en atributos de prioridades, tiempos, y dependencias en secuencias de ejecución entre las corrutinas, dependiendo de la naturaleza del proceso.

5.- En el caso de que se cumpla que para ninguna de las corrutinas ha transcurrido el tiempo que pidieron antes de volver a ser ejecutadas, se hace una llamada a _check pasandole el valor mínimo de pulsos que hay que esperar para que la condición de ejecución se cumpla para una de las corrutinas. De esta manera, los demás programas que están corriendo en la máquina obtienen el control de la misma.

La anterior función puede decidir en el caso de qùe alguna de las corrutinas termine, ya no pasarle el control nunca más, lo cual puede ser llevado a cabo por medio de una variable global que le indique que esa corrutina ya termino su ejecución. Si las corrutinas son ciclos infinitos, esto no es necesario. Además, es preciso que el ambiente propicio

de ejecución para cada corrutina esté puesto correctamente antes de que CheckTime le transfiera el control por medio de alguna llamada a _func2. Esto impone algunas restricciones adicionales al mecanismo de control de CheckTime a menos que el programa se diseño de manera que comience con una subrutina de inicialización encargada de preparar los ambientes de ejecución adecuados. Opcionalmente se puede usar CheckTime para envío de mensajes entre procesos mediante la adición de algunos parámetros si se desea esta funcionalidad.

El anterior mecanismo permite entonces ejecutar concurrentemente varios procesos. En todos los procesos concurrentes, hay que cuidar el manejo de los recursos que son compartidos por varios de ellos. Estos recursos son en nuestro caso, los archivos y las variables globales de memoria que sean utilizados para lectura y escritura.

### 3.4 Consideraciones de programación

El manejo que permita la integridad de los archivos es a través de la funcionalidad que proporciona SHARE. DOS permite el acceso a archivos por dos medios, los FCB's ("File Control Blocks") y los manipuladores de archivos ("Handles") ([DUNR86]). Para poder utilizar SHARE adecuadamente, es

necesario que todo el acceso a archivos sea hecho a través de manipuladores de archivos, puesto que la operación de abrir archivos usando FCB's es hecha siempre en el modo de compatibilidad que no permite el acceso a el archivo por varios procesos. Las funciones de SHARE son principalmente dos, una es extender la apertura de archivos de manera que éstos puedan ser compartidos por varios programas a la vez, y otra es permitir el bloqueo ("Lock") y desbloqueo ("Unlock") de regiones de archivos, con los cuales es posible desarrollar un esquema de semáforos para coordinación de procesos concurrentes. Cuando SHARE está en efecto, es posible abrir los archivos para compartir en diferentes modos dependiendo del acceso que deseemos restringir, que pueden ser ningún acceso, accesos de lectura, accesos de escritura y accesos de lectura y escritura. En particular, si deseamos que un archivo sea abierto solamente por un proceso a la vez, lo abrimos en modo exclusivo, es decir restringiendo el acceso de lectura y escritura al mismo.

Para el caso de las corrutinas, descritas anteriormente, si deseamos compartir archivos, esto puede hacerse si cada corrutina abre el archivo a ser compartido en el modo de no restringir ninguna operación sobre el archivo, este enfoque in embargo es limitado debido a que DOS en sus versiones previas a 3.30, permite un máximo de 20 archivos abiertos por programa, de manera que el utilizar diferentes manipuladores de archivo para cada corrutina que abre un archivo común puede resultar muy costoso en términos de el número de

manipuladores de archivo que quedan libres para la ejecución de todo el programa. Un enfoque que se puede tomar en este caso es abrir los archivos comunes utilizando manipuladores de archivo globales a los que tengan acceso todas las corrutinas que los vayan a utilizar. Una restricción aquí es que para mantener la integridad de estos archivos, no se transfiera el control de una corrutina a otra cuando estamos a la mitad de una operación de acceso a los archivos compartidos.

Como no se cuenta con un mecanismo de bloqueo de acceso para memoria, las variables globales utilizadas por mas de una corrutina, deben ser alteradas solo bajo ciertas condiciones especificas sobre algún semáforo, de manera que se conserve la consistencia y coordinación entre las corrutinas. En el caso en que no se tenga este tipo de variables o dependencias, es decir, si la ejecución de las corrutinas es independiente, estas precauciones no son necesarias.

Es sumamente importante para los programas que utilizan corrutinas que todas las condiciones de error sean manejadas adecuadamente, especialmente las de acceso a archivos, ya que si por ejemplo cada vez que hay un error de lectura o cada vez que se desea suspender la ejecución de los procesos no se cierran todos los archivos que se hayan abiertos en ese momento, como el número de archivos abiertos simultáneos que

se pueden tener es pequeño, el programa eventualmente los agotará y todas las operaciones de apertura subsecuentes empezarán a fallar. Del mismo modo, siempre que se suspenda alguna de las corrutinas, si esta utiliza algún apuntador al que se le proporcionó memoria dinámicamente, dicha memoria debe ser regresada al sistema, puesto que la cantidad de memoria accesible por este medio debe estar minimizada a sólo la estrictamente necesaria por consideraciones de tamaño del programa en memoria. Dado el manejo de varios stacks en el area de datos, una falla de uso de memoria de este tipo resultaría de consecuencias fatales.

Todo lo anterior aplica también para la creación de corrutinas en programas que no sean residentes, siendo la única diferencia que la llamada a _check dentro de la rutina CheckTime es una llamada ficticia y por tanto no transfiere el control a ningún otro programa. Con objeto de facilitar el trazo y corrección de las diferentes corrutinas durante la fase de desarrollo, se puede hacer que CheckTime también sea una rutina ficticia lo cual haría que la máquina ejecutara solo una corrutina y no se tuviera interferencia originada por las demás corrutinas.

Capítulo 4

# Remoción de programas residentes en memoria.

## 4.1 Condiciones para una remoción segura

Una facilidad que los programas residentes en memoria deben proporcionar es la capacidad de removerse de ésta, liberando de esta manera memoria en la máquina para que pueda ser utilizada por otros programas. Este proceso aunque es relativamente sencillo, tiene sus puntos finos.

La manera de remover un programa residente en memoria es a través de un programa que se encarga de el proceso de remoción del mismo y que puede ser el programa residente mismo cuando se cumpla cierta condición. Sin pérdida de generalidad vamos a tratarlo como un programa independiente en la discusión que sigue.

El programa removedor, debe averiguar antes que nada si el programa que se pretende remover se encuentra residente en memoria. Para este fin, se puede hacer una función que

busque para las diferentes interrupciones, alguna que tenga una firma que la identifique como utilizada por el programa residente. Esta firma, no es otra cosa que un pedazo de texto identificando el programa residente y que sea sencillo de localizar en memoria a partir de la dirección de atención a la interrupción.

En caso que no se detecte una interrupción con las anteriores caracteristicas, se tendría una indicación de que o bien el programa residente no se encuentra instalado en memoria o algún otro programa residente cargado en memoria posteriormente se encuentra ocupando la interrupción que tiene la firma. Cualquiera que sea el caso, si no se cumple alguna de las anteriores condiciones, no se debe proceder a la remoción del programa residente pues todo tipo de mal funcionamiento puede ocurrir al dejar ya sea vectores de interrupción apuntando a memoria ya liberada, ó interfiriendo con otros programas residentes cargados posteriormente.

Algunos programas residentes utilizados comercialmente tienen el inconveniente de no seguir este esquema y de efectuar el proceso de remoción de memoria aún en el caso en que hay algún otro programa residente que esta cargado en memoria posteriormente y que utiliza alguna de las interrupciones que ellos usan, cuando tal es el caso puede ocurrir que otros programas residentes dejen de funcionar correctamente o incluso que el sistema se pare.

Este tipo de programas "mal comportados" se deben de cargar siempre siguiendo las recomendaciones de los proveedores. El mecanismo más seguro de cargar y descargar programas residentes en memoria manera es al final de todos los demás y descargarse estrictamente en orden LIFO ("last in, first out") para garantizar un comportamiento adecuado. Cualquiera que sea el caso, es recomendable seguir siempre un orden de ese tipo para descargar los programas residentes, ya que debido al manejo que hace DOS de memoria, la memoria liberada por el programa removido quedará fragmentada mientras haya programas cargados en direcciones mas altas de memoria.

El programa residente que queremos remover de memoria, en el caso general, se puede encontrar en plena ejecución en el momento en que decidimos removerlo, de manera que antes de proceder a su remoción de memoria, es posible que sea necesario pararlo y llevarlo a un punto estable conocido, en el cual haya cerrado los archivos que tuviese abiertos y abortado ó terminado las demás tareas pendientes que pudiera tener y liberado los recursos físicos que pudiera estar utilizando en ese momento (puertos paralelos, seriales, tarjetas, dispositivos externos, etc).

Esto se puede lograr como se señala en la sección 2.5,

por medio de un mecanismo de sincronización basado en semáforos. Vamos a analizar el caso de sincronización cuando se usan corrutinas por ser mas general, ya que un programa que no tiene corrutinas se puede considerar como una sola corrutina.

Por medio del mecanismo de sincronización mencionado u otros, podemos parar a las diferentes corrutinas de el programa residente y llevarlas a un punto seguro. Cuando todas las corrutinas del programa han llegado al punto estable, podemos proseguir con el proceso completo de remoción.

Una manera de hacer que las corrutinas tengan un lugar "seguro" donde pararse es diseñarlas con un esquema parecido al siguiente:

```
corrutina1() (
    while (TRUE) (
        while (!alto) (
            /*                                  */
            /*                                  */
            /* Aqui va el código de la corrutina    */
            /*                                  */
            /*                                  */
        );
```

```
            siguel=FALSE;
            while (alto)
                CheckTime(1);
            siguel=TRUE;
        };
};
```

en donde 'alto' es una variable que indica que se desea parar al programa, y que cambia de valor dentro de CheckTime dependiendo de los valores que le proporciona el mecanismo de semáforos del programa. 'siguel' es una variable que indica cuando tiene el valor FALSE, que la corrutina 'corrutina1' se encuentra parada y en un lugar seguro. Es sumamente importante que el código de la corrutina esté hecho de tal manera que todos sus ciclos internos dependan además de su lógica interna, del valor de la variable 'alto', para evitar que alguna de las subrutinas se quede esperando indefinidamente en una condición de "deadlock".

En general, el programa residente no debe ejecutar operaciones síncronas que puedan tomar más de unos cuantos ciclos de reloj en ejecutarse, puesto que durante este tiempo no se regresará el control a los demás programas que estan corriendo simultáneamente. En el caso en que haya que hacer operaciones lentas con algunos dispositivos externos, estas deben de hacerse (si la naturaleza del dispositivo lo permite) en modo asíncrono usando un esquema parecido al siguiente:

```
/*                          */
/* operación asíncrona      */
/*                          */

completa=FALSE;
manda_asinc(&completa);
while (!completa && !alto)
    CheckTime(1);
if (!completa) {
    cancela_manda_asinc();

    /* Aquí se deben liberar los recursos como   */
    /* archivos abiertos, archivos temporales,   */
    /* memoria dinámica, y todo lo necesario     */
    /* para mantener la integridad del programa */
}

/*                          */
/* continua la rutina       */
/*                          */
```

Es importante que en el caso en que no se termina la operación asíncrona, esta sea cancelada como se muestra en el esquema, así como la liberación de los recursos compartidos o globales utilizados y las labores de limpieza para que el programa se mantenga íntegro.

## 4.2 Restauración de interrupciones y bloques de memoria

Con objeto de remover el programa residente, es necesario que tengamos acceso a las variables en que el programa residente guardó los valores originales de los vectores de interrupción para las interrupciones que esta utilizando, la manera mas sencilla de hacer esto es hacer que la rutina de reestablecimiento de las interrupciones sea atendida por una de las rutinas que el programa residente puso, por ejemplo, la interrupción usada para comunicarse con el programa residente. Se checa una por una todas los vectores de interrupción que el programa residente utiliza comparando su valor actual con el que tenían al momento de la instalación, si alguno de estos valores difiere, significa que algún otro programa cambió el valor que el programa residente puso.

En particular hay que tener cuidado cuando se esta haciendo acceso a manejadores de dispositivos, ya que hay algunos que durante algunas de sus funciones pueden atrapar interrupciones que no se reestablecen hasta que se ejecuta alguna otra función contraparte (ej. arranque y fin de operación), la cual si no se lleva a cabo como parte del proceso normal de desactivación de la corrutina que lo utiliza haría parecer como si algún otro programa estuviese actualmente cargado usando esa interrupción.

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

Si se da el caso anterior o el de otro programa residente utilizando alguna de las interrupciones del que usamos, nuestro programa no se debe remover, tenemos sin embargo manera de desactivarlo para que no se vuelva a activar. Esto se logra modificando la rutina que transfiere el acceso al programa residente haciendola que ejecute simplemente una instrucción de retorno (retf). Esto es sencillo de hacer ya que el programa residente tiene acceso a sus propias rutinas siempre que esta activo. Esta acción permite que el programa residente quede funcionalmente desactivado y tenga requerimientos mínimos de tiempo de proceso.

Si ninguno de los vectores de interrupción que puso el programa residente ha cambiado, el siguiente paso es el de restablecer los valores originales de dichos vectores. Esto se hace a partir de los valores salvados al momento de instalación y utilizando la función 25h de DOS. Las llamadas a DOS que se hacen al momento de remoción a partir del punto en que los vectores originales son restablecidos deben de ser protegidas de ser ejecutadas durante una llamada de DOS con reingreso no permitido, ya que en este punto ya no se tiene protección al respecto.

Los programas residentes en memoria cuando ejecutan la instrucción de termina y queda residente, dejan residentes

dos bloques contiguos de memoria que son por un lado, una copia de las variables del medio ambiente ("environment") de DOS al momento en que se ejecutó, y por el otro, el programa residente en sí, incluyendo su PSP. Aunque los bloques son contiguos, debido a la manera en que DOS hace su manejo de memoria, no se pueden regresar con una sola operación, sino que hay que hacerlo con dos, una para cada uno de los bloques ([IBMC87]), la función de DOS que hace el regreso de memoria es la 49h, y solo necesita conocer el valor del primer segmento de memoria del bloque. Estos dos valores por tanto se deben de guardar desde el momento de instalación que es cuando se conocen.

Podría preguntarse cómo es posible que un pedazo de código se remueva de memoria a sí mismo, puesto que una vez que se ejecuta el regreso de la memoria, se supone que también se devuelve la memoria donde se encuentran las siguientes instrucciones a ejecutar que deben ser al menos una instrucción de retorno y el stack de este proceso que debe tener al menos las direcciones de retorno. Esto sin embargo es posible debido a que ésta devolución de memoria es en realidad solo una marca que indica a DOS que la siguiente vez que necesite memoria ésta está accesible, y por tanto, no se borran inmediatamente los datos que dicha memoria contiene, dando asi tiempo a ejecutarse las instrucciones de retorno de la rutina de interrupción antes de que dicha memoria sea utilizada nuevamente.

Capitulo 5

# Aplicaciones.

Esta tecnologia abre la puerta a un gran número de nuevas posibilidades de programación y a una mejor utilización de recursos materiales tales como la planta instalada de PC para la investigación, la industria y oficinas. A manera de ejemplo describiré brevemente algunas de estas posibilidades de desarrollo.

## 5.1 Captura de datos experimentales.

Es el caso muy común en laboratorios de investigación, en los que se tiene montado algún experimento cuyos datos de salida se capturan por medio de una PC, ya sea via el puerto serial o algúna otra interfaz. En el caso de experimentos que tardan muchas horas o aún dias en llevarse a cabo, la PC cuya función es únicamente la de capturar datos que se producen esporádicamente o bien a ciertos intervalos de tiempo relativamente grandes comparados con la velocidad del procesador de la máquina, está dedicada por completo a dicha

tarea pasando la mayor parte del tiempo sin hacer nada útil, la creación de dicho tipo de programas de captura de datos residente en memoria por medio de la tecnología aquí descrita, permitiría la utilización de esa máquina simultáneamente para cualquier otro tipo de tareas como pueden ser edición de textos, proceso de datos, creación de programas, etc. permitiendo así un mucho mejor uso de los recursos de investigación del laboratorio.

### 5.2 Solución de problemas con métodos numéricos.

El cálculo en algunos problemas por medio de métodos numéricos puede tomar mucho tiempo dedicado por una máquina cuando se trata por ejemplo de resolver ecuaciones con derivadas parciales, cálculo de matrices, operaciones de muy alta precisión o de números enteros de muchas cifras, etc. Estos cálculos pueden tomar varias horas de máquina que si son ejecutadas en la noche cuando nadie mas puede necesitar el acceso a la máquina no representa problema, pero durante las horas de oficina, puede ser crítico el tener una máquina dedicada al cálculo. Haciendo el cálculo con la tecnología aquí descrita, evitaría ese problema y dejaría la máquina accesible a otros usuarios. Por supuesto, si el problema a resolver es de tal magnitud que requiera varios dias con la PC dedicada, una solución más adecuada es utilizar otra máquina mas rápida o especializada como puede ser un

supercomputador o mainframe.

### 5.3 Simulaciones, reportes.

Estos son casos tipicos de procesos largos que se pueden crear fácilmente con esta tecnología y para los cuales la máquina no necesita estar dedicada totalmente a ellos, en especial durante las horas de oficina en que puede ser utilizada para alguna otra actividad paralela.

### 5.4 Envio de mensajes en redes locales de computadoras (sin "SERVER").

Una aplicación altamente interesante de la tecnología descrita en esta tesis, es el desarrollo de un sistema de envio y recepción de mensajes de un correo electrónico en una red de computadoras sin "SERVER" (este tipo de programa se puede desarrollar por ejemplo utilizando los protocolos proporcionados por redes de computadoras como AppleTalk o NetBios). El programa consistiría de dos procesos, uno que envia los mensajes y otro que los recibe, los mensajes son creados por alguna aplicación externa que corre en la máquina y que simplemente deposita los mensajes en algun

subdirectorio específico de la máquina, estos mensajes contienen una estructura que les permite encontrar el nombre del destinatario a quien van dirigidos los mensajes. Antes de comenzar los procesos de enviar y recibir (que son programados como corrutinas para ahorrar memoria usada por el programa residente tal como se describe en los capítulos anteriores), el programa se registra en la red con algún nombre o identificador que sea válido dentro del protocolo de la red. A continuación, se inician los procesos.

El proceso de envio permanece en un ciclo buscando archivos para enviar los que estén en la estructura de directorios correspondiente, una vez que encuentra algún mensaje válido, lo lee y determina quien es el destinatario, y establece una sesión con éste en la cual por medio de algún protocolo de comunicación se envia el mensaje; una vez que este fue enviado exitosamente, se borra el mensaje para no enviarlo de nuevo en la siguiente iteración. Si por algún motivo la comunicación se rompiera o algo fallara, el mensaje no es borrado y se reintentará reenviar más tarde.

El proceso que recibe es aún mas simple, pues consiste solamente de un ciclo a la espera de que alguien trate de establecer sesión con él. Una vez que esto ocurre, utiliza un protocolo de comunicación para recibir el mensaje y lo deposita en otra estructura de subdirectorios diseñada para el efecto.

Este sistema tiene como ventaja sobre un programa de aplicación dedicado a lo mismo, en que estará vivo todo el tiempo a la espera de una posible comunicación sin necesidad de establecer explicitamente una conexión con cada uno de los demás usuarios del sistema, y además las máquinas no están dedicadas solamente a este proceso.

Con un poco más de sofisticación, se puede agregar al sistema un mecanismo de almacenaje y reenvio para el caso de mensajes entre usuarios cuyos horarios o máquinas no están activas simultáneamente nunca, el cual a su vez puede servir como puente para envío de mensajes fuera de la red local a otros sistemas de correo, si es que el correo proporciona la flexibilidad para este tipo de funcionalidad.

### 5.5 Sincronización de procesos en redes locales de computadoras.

En sistemas de redes de computadores que no proporcionen el servicio de reloj de la red, este servicio se puede simular usando esta tecnología y sin tener que dedicar una máquina para la prestación del mismo. El reloj de la red es simplemente un proceso que se registra con un nombre especial

(ej. NetWorkClock) y que esta todo el tiempo a la espera de establecer sesión con quien asi lo solicite, enviandole su hora local, que será considerada la hora de la red. En el caso de redes que proporcionan el servicio de "broadcast" o radiación de mensajes, es posible un desarrollo aún sin necesidad de establecer sesiones de comunicación, aunque esta solución es mas cara en terminos de tráfico en la red.

### 5.6 Sistema automático de respaldo de archivos.

Otra interesante posibilidad es la creación de un sistema automático de respaldo de archivos, el cual se podría activar por decir algo, cada 30 minutos, corriendo residente en memoria, de manera transparente y pudiendo servir tanto para máquinas dentro de una red como para estaciones individuales. Un sistema de este tipo, permitiría que en caso de alguna falla en el disco de trabajo, se pudiera contar con un respaldo relativamente reciente. Desde luego que este mecanismo en todo caso sería adicional a las labores rutinarias de respaldo que se pueden hacer con menor frecuencia por ejemplo diariamente o una vez a la semana.

### 5.7 Sistema de protección anti-virus.

Una extensión natural de la aplicación mencionada en el inciso 5.6 es la de crear un programa residente en memoria que tiene como entrada de datos un archivo de texto con información acerca de los tamaños y CRC de los archivos ejecutables del sistema (.EXE y .COM) y los manejadores de dispositivos del sistema de manera que le permita checar la consistencia de dicha información y detectar las posibles instancias de una infección o ataque viral en una etapa temprana de la misma. Especialmente en el caso de una red de computadoras, si esta se encuentra relativamente abierta a sus usuarios y/o a comunicación con otras redes remotas, este sistema se puede volver un salvavidas en caso de un ataque de este tipo. Desde luego que aquí la efectividad de la vacuna depende ampliamente en tener al día la lista de los programas y utlierías ejecutables desde el sistema. Nuevamente, la gran ventaja sobre un sistema de este tipo reside en que una máquina no es necesario dedicarla todo el tiempo a esta tarea, lo cual es crucial si se trata de una estación de trabajo individual independiente.

### 5.8 Alertas.

Como aplicación inmediata de esta tecnología esta la programación de Alertas, que son programas que inican algúna situación extraordinaria en la máquina de lo cual se desea

informar al usuario en el momento en que esta ocurre, que podría ser por ejemplo la creación de algún archivo en cierta estructura de directorios de la aplicación mencionada en el inciso 5.4, que indicaría la llegada de correo nuevo, o una alerta que notifique al supervisor de una red cuando alguna de las tablas del sistema o archivo importante haya sido alterado por algún usuario, etc.

### 5.9 Alarmas.

Otra aplicación típica y sencilla de programar, es un pequeño sistema de alarmas que permitiera al usuario mediante una sencilla interfaz, hacer que la máquina le recuerde sus citas y compromisos, de manera que por ejemplo si se tiene una junta a las 5:00 pm, poniendo una alarma en el sistema a las 4:45 pm, le recordará la junta y le dará tiempo de llegar, o cancelar en todo caso. Esta interfaz se puede construir de manera que desde otros programas de aplicación se tenga acceso a la modificación de las alarmas del sistema, de manera que haya un evento de este tipo pueda ser registrado tanto por un sistema para trabajo en equipo, una aplicación de tipo agenda y el mecanismo de alarma.

## Capítulo 6

## Esquema de desarrollo de la tecnología.

Los pasos necesarios para llevar a cabo una tecnología como la aquí descrita, se pueden separar en cuatro clases diferentes que corresponden a partes independientes de la misma. Estas partes son:

### 6.1 Desarrollo de programas residentes

a) Rutina de chequeo de instalación.
Verifica si el programa se encuentra instalado actualmente en memoria. Puede utilizarse para esto una marca o firma en memoria. Esto permite evitar que el programa residente se cargue en memoria más de una vez utilizando memoria inutilmente o causando mal funcionamiento de los mismos en el caso de acceso a recursos físicos de la máquina.

b) Rutinas de atención a interrupciones.

Para cada una de las interrupciones que se desean atrapar, se debe proporcionar una rutina que se encargue de proveer la funcionalidad de la rutina original de interrupción. Las interrupciones que se tienen que atrapar en el esquema aquí descrita son las siguientes:

08h : Interrupción del reloj, usada como medio principal para la obtención de acceso al control del procesador. Esta es una interrupción del nivel físico de la máquina y ocurre 18.2 veces por segundo. Es generada por el controlador de interrupciones de la máquina.

28h : Interrupción de señalamiento de libre reingreso a DOS, utilizada para obtener acceso al control del procesador desde dentro de llamadas a DOS que permiten reingreso y de larga duración. Es una interrupción que pasa al nivel lógico de la máquina y es generada por DOS. El atrapar esta interrupción junto con la 08h anteriormente descrita garantiza que el despachador de procesos obtendrá el control del procesador muy frecuentemente, quedando determinado por su lógica el decidir si el control pasa al programa residente o

no se lo pasa.

10h : Interrupción de acceso a video. Esta interrupción es parte del BIOS y es generada al nivel lógico de la máquina en general por los programas de aplicación via DOS o directamente. En esta tecnología es utilizada para evitar la activación del proceso residente cuando esta interrupción se encuentra activa evitando problemas de reingreso si es que se utilizan sus servicios desde el programa residente.

13h : Interrupción de acceso a disco. Esta interrupción también es parte del BIOS y también se genera al nivel lógico de la máquina. Esta interrupción la producen los programas de aplicación generalmente a través de DOS o bien directamente. En esta tecnología es utilizada para evitar la activación del proceso residente cuando esta interrupción se encuentra activa evitando problemas de reingreso si se utilizan sus servicios desde el programa residente tales como puede ser el afectar el mecanismo de sincronización del disco.

60h-67h : Interrupciones para programas del usuario. Estos vectores de interrupción se encuentran vacios al momento de arrancar la máquina y se proporcionan para que las aplicaciones que corren el la máquina hagan uso de ellos si es preciso. Es posible que sea necesario el atrapar alguna de las interrupciones del usuario para utilizarla como medio de comunicación con el programa residente y en particular para llevar a cabo la remoción de memoria del programa residente.

c) Rutina de instalación.

Esta rutina es la que deja el programa residente en memoria. Incluye la siguiente funcionalidad.

1.- Checa que el proceso no se encuentre instalado.

2.- Atrapa las interrupciones que va a utilizar y sustituye sus rutinas de atención por las nuevas, salvando las direcciones originales de las rutinas de atención dichas interrupciones para reestablecerlas cuando

sea necesario.

3.- Calcula el tamaño en memoria del programa.

4.- Ejecuta la función de terminar y quedar residente.

d) Rutina de devolución de control a otros procesos. Esta rutina es la que se encarga de transferir el control a los demás programas que se están ejecutando simultáneamente en la máquina. El programa residente gana el control via las interrupciones del procesador y necesita esta rutina como mecanismo de retorno que pueda ser llamada desde dentro del programa residente en los lugares adecuados. Un ejemplo de este tipo de rutina es la que mencionamos anteriormente como _check(). El control es regresado al proceso activo cuando la interrupción que le dio el control al programa residente ocurrió.

## 6.2 Interfaz con "C"

a) Adaptación de la interfaz de tiempo de ejecución del compilador para llevar a cabo las rutinas de instalación del programa residente. Esta adaptación es necesaria con objeto de contar con el ambiente de programación para que los programas en lenguaje "C" ejecuten residentes en memoria.

b) Adaptación de la interfaz de tiempo de ejecución del compilador para permitir controlar desde el programa el tamaño de stack y memoria dinámica que el programa residente va a utilizar. Este paso es necesario como medio para poder controlar y minimizar el tamaño del sector de datos que utiliza el programa residente, permitiendo de esta manera un manejo más eficiente de la memoria de la máquina.

c) Creación de la función exit() teniendo en cuenta la naturaleza residente del proceso. Es necesaria una función exit() que NO ejecute un exit de DOS, que mataría algun otro de los procesos activos en la máquina, sino que por ejemplo haga un llamando a la rutina de remoción en memoria o simplemente cerrando sus archivos y entrando en un ciclo que regrese el control inmediatamente cada vez que lo obtiene.

d) Es recomendable evitar la inclusión de funciones de acceso a archivos con buffer para evitar la duplicación o carga en memoria de código innecesario. Lo anterior se aplica también a las funciones printf(), sprintf(), scanf(), etc.

e) En el caso de programas que se ejecutan repetidas veces, se debe evitar la dependencia de los valores de inicialización de las variables estáticas. Esto es debido a que dichos valores son asignados en el código generado por el compilador, y si se altera su valor, la siguiente vez que el programa ejecuta tendrán valores incorrectos.

### 6.3 Manejo de corrutinas

a) Partición del stack de "C" en varios stacks.

Es necesario seccionar el stack del programa en "C" en varios stacks, uno para cada una de las corrutinas a ser utilizadas en el programa. De otra manera, no sería posible que cada una de las corrutinas tuviera su propio ambiente de

ejecución independiente.

b) Rutina de inicialización de un stack.

Esta rutina corresponde a la llamada _func1() descrita la sección 3.2 y un ejemplo de la misma se incluye en el apéndice B. Su objeto es inicializar los nuevos stacks que las corrutinas utilizarán durante su ejecución de manera que puedan ser utilizados apropiadamente por las mismas, esta inicialización incluye el almacenar en ellos direcciones de regreso apropiadas.

c) Rutina de intercambio de stacks.

El intercambio de contextos de ejecución es llevada a cabo desde esta rutina, corresponde a la que referimos como _func2() a lo largo de la exposición y un ejemplo de la misma se muestra en el apéndice B. Hace un intercambio de stacks similar al de _func1() pero asume que estos se encuentran inicializados.

d) Función despachadora de corrutinas.

Esta es la función que se encarga de decidir que corrutina necesita el control y por cuanto tiempo se le va a conceder este. En el capítulo 4 se hace referencia a una función de este tipo con el

nombre de CheckTime(). Puede usar diferentes tipos de algoritmo para esto como una cola, transferencia de control aleatoria, manejo de prioridades entre corrutinas, etc.

### 6.4 Comunicación entre programas

a) Rutina de atención a interrupción del usuario.

Esta rutina debe cargarse en una de las interrupciones disponibles para programas del usuario y proporcionar un medio de comunicación con el programa residente indicandole que tipo de acciones debe ejecutar por medio de comandos, tales como suspender su ejecución, removerse de memoria, etc.

b) Interfaz de comunicación via semáforos.

Esta interfaz nos permite una comunicación efectiva para ejecutar acciones que necesitan de una coordinación muy precisa entre los programas. Puede o no ser necesaria dependiendo del tipo de sincronización que los procesos necesiten llevar a cabo entre ellos.

Las anteriores partes son independientes entre ellas, ya que por ejemplo, se puede hacer un desarrollo residente sin lenguaje "C", sin uso de corrutinas y sin uso de comunicación entre programas, o se puede hacer un desarrollo con corrutinas pero sin quedarse residente en memoria, etc.

Las anteriores partes son independientes entre ellas, ya que por ejemplo, se puede hacer un desarrollo residente sin lenguaje "C", sin uso de corrutinas y sin uso de comunicación entre programas, o se puede hacer un desarrollo con corrutinas pero sin quedarse residente en memoria, etc.

Capítulo 7

# Conclusiones.

## 7.1 Discusión

Después de la anterior exposición, surgen como preguntas obvias las siguientes: ¿Que tan segura es la tecnología aquí descrita? ¿Que prácticas de programación son necesarias trabajando en este tipo de ambiente de desarrollo? ¿De que tamaño es el impacto en la capacidad de proceso corriendo con aplicaciones residentes en paralelo? ¿Que nuevas posibilidades surgen a partir de este desarrollo?, ¿Cual es la aportación del presente trabajo a la comunidad de programadores y usuarios de computadoras? ¿Está este desarrollo limitado por lo expuesto o puede ser expandido?. En el presente capítulo es donde se da respuesta a éstas y otras preguntas.

La tecnología se ha probado intensivamente en el campo en varias de sus aplicaciones, (envio de mensajes, reloj de red, alerta), en algunos casos durante periodos ininterrumpidos de

tiempo muy largos y en paralelo con todo tipo de aplicaciones sin que se le hayan encontrado problemas de operación que no hayan sido solucionados, sin embargo, como se explica en el capítulo 2 de esta tesis, no se está protegido contra el mal uso de los recursos de la máquina por parte de otros programas (como ningún otro programa lo está).

El tipo de programación necesario para un programa residente es programación a la defensiva, es decir, el programa residente debe de cuidarse contra cualquier suposición que otros programas puedan hacer sobre la utilización de los recursos de la máquina y manejar las condiciones de error de manera adecuada. Ciertos errores típicos de programación resultan fatales en estos ambientes de ejecución, tales errores son por ejemplo, el no liberar la memoria obtenida de manera dinámica, ya que esto hace que eventualmente se acabe este recurso y todo el programa comience a fallar. Dejar archivos abiertos hará que otros procesos no tengan acceso a ellos y eventualmente llevará al agotamiento de los manejadores de archivos disponibles para el programa, etc.

Si utiliza recursos físicos de la máquina tales como tarjetas de red, que puedan necesitar para su funcionamiento un proceso de inicialización, deberán ser devueltos a su estado original al final del programa. Lo mismo aplica también a recursos tales como locks sobre archivos, etc. Un

programa residente en memoria no se puede dar el lujo de parar el procesador y causar la pérdida de cuanto trabajo útil se esté llevando a cabo por otros procesos al mismo tiempo.

### 7.2 Degradación de la capacidad de proceso

Siempre y cuando desde el programa residente se llame frecuentemente a la rutina que cede el control al sistema operativo, la capacidad de proceso varia dependiendo de que tan intensivo es el trabajo que el programa residente efectua, variando desde menos de .05% (negligible para todo propósito práctico) en el caso de aplicaciones sencillas y de poco proceso como las alertas, hasta cerca de un 30% en programas de continuo acceso a disco flexible. Ahora bien, dada la implementación aquí descrita, debido a que el control al programa residente también se obtiene a través de la interrupción 28h, el programa residente recibe mucha mayor atención cuando el sistema se encuentra sin hacer nada. Los números anteriores fueron determinados en pruebas de capacidad de proceso con el procesador intensamente ocupado. La capacidad de proceso se ve también degradada conforme se utilicen mas programas simultaneos peleando por el control de la máquina. La capacidad de proceso a la que me refiero es a la cantidad de tiempo del procesador que es asignada a la aplicación externa o no residente en memoria que es la que

está visible normalmente al usuario.

Se llevaron a cabo pruebas para la medición de la degradación de la capacidad de proceso ocasionadas por la utilización de este tipo de programas residentes para dos panoramas, uno en el cual el programa residente la única tarea que realiza es ganar control y devolverlo solicitandolo lo más pronto posible, que llamaré procesos tipo "ResDummy" y otro que es un reloj el cual calcula la fecha y hora por medio de llamadas a DOS, lo convierte a un formato intelegible y despliega en pantalla, acto seguido devuelve el control solicitandolo inmediatamente a su vez. A éste último lollamaré "ResClock". La prueba consistia en medir el tiempo que tardaba en ejecutar un programa prueba ("Benchmark") que contenía una serie de ciclos de longitud fija, y comparando la variación de sus tiempos de ejecución conforme había mas programas residentes desarrollados con esta tecnología, similares a "ResDummy" ó a "ResClock" según fue el caso. La prueba se llevó a cabo en una máquina Micron AT Pro, procesador 80286, aunque esto es irrelevante pues los resultados se presentan en forma comparativa respecto a la capacidad original de proceso. Se obteniendo los siguientes resultados medidos con respecto a la capacidad de proceso original de la máquina en términos porcentuales:

| # de procesos | Capacidad de Proceso % "ResDummy" | Capacidad de Proceso % "ResClock" |
|---|---|---|
| 0 | 100.0 | 100.0 |
| 1 | 98.9 | 94.8 |
| 2 | 97.9 | 91.1 |
| 3 | 95.8 | 87.6 |
| 4 | 94.8 | 84.4 |
| 5 | 93.9 | 80.0 |
| 6 | 92.0 | 76.7 |
| 7 | 91.1 | 71.9 |
| 8 | 90.2 | 68.1 |
| 9 | 88.5 | 65.2 |
| 10 | 87.6 | 61.7 |
| 11 | 86.8 | 56.8 |

* Nota: Dado el rango de error de la prueba diseñada, la cifra decimal no es significativa y se presenta tan sólo como referencia.

Los resultados de estas pruebas son claros y nos muestran una degradación lineal de la capacidad de proceso. Este resultado es el esperado de acuerdo al razonamiento siguiente:

El tiempo que estos programas tardan en tomar el control, realizar su tarea y devolverlo es fijo (para cada uno de los programas prueba) con bastante aproximación, llamemosle "Tp" y midámoslo en segundos. Como los programas ganan acceso 18.2 veces por segundo (por su diseño en el que inmediatamente después de ejecutar solicitan el control de nuevo), y despreciando el tiempo de ejecución de la rutina de atención a la interrupción de BIOS, se ejecutarán una vez cada 0.055 segundos aproximadamente, tomando "Tp" segundos para su ejecución y dejando el resto para todos los demás procesos. Conforme haya más procesos similares residentes ejecutando, cada uno de ellos tomará a su vez "Tp" segundos, de los restantes, disminuyendo la capacidad de proceso accesible para los programas de aplicación a su vez en en un porcentaje igual a "Tp"/0.055. Cuando el número de procesos residentes similares corriendo simultáneamente sea igual a mc = 0.055/"Tp", la capacidad de proceso externo se habrá degradado totalmente, pues justo al terminar de atender a los mc procesos se generará otra interrupción que forzará nuevamente la atención a estos. Cuando el número de procesos residentes es mayor al de mc mencionado, se generará un desbordamiento de stack ya que la llamada a la interrupción se generará nuevamente antes de haber terminado y ejecutado el anterior "IRET".

### 7.3 Relevancia del desarrollo

A partir de esta tecnología se abre clara brecha para el desarrollo serio de programas residentes en memoria de aplicación científica o comercial. Es posible y directo el desarrollo tanto de utilerías corriendo paralelamente con otras aplicaciones así como utilerías activadas por el usuario (generalmente por la vía del teclado).

En conclusión, la tecnología mostrada representa una solución elegante al problema de desarrollo de programas residentes en memoria. Su interface con el lenguaje de programación "C" proporciona al programador la potencia de un lenguaje de programación de alto nivel para el desarrollo de aplicaciones, a la cual a su vez facilita la labor de corrección durante la fase de desarrollo. El campo que se abre a partir de este tipo de desarrollo es amplísimo, puesto que en principio, cualquier utilería desarrollada en lenguaje "C" se puede convertir en un programa residente en memoria proporcionando la comodidad y flexibilidad de estar accesible en todo momento y desde el interior de otras aplicaciones. El manejo de corrutinas le da especial fuerza para desarrollo de sistemas complejos en los que de otra manera sería necesario crear más de un programa residente.

La tecnología aquí descrita se presenta además como una

solución económica, práctica y relativamente sencilla al problema de proporcionar multiproceso a DOS, contando en cada proceso con toda la funcionalidad que el sistema operativo proporciona y sin la necesidad de utilizar un nuevo sistema operativo. Esta capacidad de multiproceso abre además nuevas posibilidades respecto al tipo de sistemas que se pueden desarrollar como por ejemplo las aplicaciones descritas en el capítulo cinco. Los desarrollos allí mencionados son especialmente valiosos en lugares donde los recursos de cómputo son limitados. Su viabilidad permite aprovechar al máximo lo recursos de computación accesibles y la implementación de soluciones alternativas a problemas que de otra manera no sería práctico tratar.

### 7.4 Posibilidades de extensión

Ahora bien, con el presente trabajo, se sientan además las bases de lo que puede ser utilizado como punto de partida para desarrollos mas sofisticados o complejos de programación bajo DOS. Entre las posibilidades de generalización y ampliación de este sistema se pueden mencionar las siguientes:

a) Una posible extensión a la tecnología desarrollada consiste en là generalización de ésta para formar el núcleo

de un sistema operativo multiproceso montado encima de DOS.

b) Otra posibilidad es la simplificación de la interfaz para ser usada para desarrollo de programas residentes en lenguaje ensamblador para programas realmente pequeños en que no se desea cargar con todo el peso de la interfaz que es de alrededor de 4 Kilobytes.

c) Permitir la utilización de memoria extendida y memoria expandida. Esto haría muy poderoso el sistema ya que ahora los programas residentes no ocuparían espacio en los 640 Kilobytes de la memoria principal de la máquina.

d) El módulo de corrutinas puede ser expandido para proporcionar un ambiente de programación concurrente para desarrollo de programas.

Finalmente, es posible concluir que la presente tesis ofrece soluciones alternativas en el campo del diseño, desarrollo y programación de sistemas orientados a una utilización óptima de sus recursos de computación, así como una plataforma de desarrollo para la creación de programas residentes en memoria y multiples posibilidades de extensión a futuro de la tecnología misma.

## Apéndice A

### Programas residentes activados via teclado.

Dentro de la familia de los programas residentes en memoria, nos encontramos con una serie de programas que no tienen que ver con multiproceso en DOS, sino que son simplemente utilerias accesibles al usuario que normalmente se encuentran durmiendo esperando su condición de activación cuando el usuario así lo determine. Este tipo de utilerías son activadas en general a través del teclado. En la discusión dentro del cuerpo de la tesis, se menciona poco acerca de este tipo de desarrollos y el objetivo de este apéndice es el de mostrar como desarrollarlos a través de la tecnología descrita.

Los programas residentes que se activan via teclado utilizan generalmente una combinación "mágica" de teclas que les permite reconocer que es necesario que se activen. Los mecanismos para la detección de dicha condición se cumple son los siguientes:

El primer mecanismo que hay para encontrar esta combinación se basa en atrapar las interrupciones de manejo de teclado de DOS que son dos, la de nivel físico (hardware) que es la interrupción 09h, y la de nivel lógico (software) que es la interrupción 16h. La interrupción al nivel físico de la máquina (09h) al ser ejecutada, proporciona como valor de regreso un código en el registro "AL", que es la llamada máscara de estado y que indica cual fue la combinación de teclas que fueron seleccionadas, basado en la siguiente tabla:

| Tecla | Código |
|---|---|
| Ins | 128 |
| Caps Lock | 64 |
| Num Lock | 32 |
| Scroll Lock | 16 |
| Alt | 8 |
| Ctrl | 4 |
| Left Shift | 2 |
| Right Shift | 1 |

TABLA 1

Esta distribución es tal que permite la utilización de cualquier combinación de ellas reflejándola en el registro "AL" y en la máscara de estado, de manera que atrapando dicha interrupción y comparando el valor de retorno de la misma en comparación con el código que hayamos elegido como combinación mágica para la activación de nuestro programa residente, en el caso de coincidir, prendemos una variable que nos indica que es hora de activar el programa residente (cuando ésto sea posible). La interrupción al nivel lógico del teclado (16h) se utiliza solamente como un medio adicional de transferencia de control al programa residente una vez que se cumplió la condición de ejecución por la recepción de la combinación mágica en la interrupción al nivel físico de la máquina.

Es claro entonces que la combinación mágica puede ser configurable al momento de instalación simplemente pasándole un parámetro a nuestro programa residente indicándole cual es dicha combinación, permitiendo de esta manera, interactuar sin interferencia con otros programas residentes que utilicen diferentes combinaciones de teclas para activarse.

La otra manera de activar programas residentes en memoria a través de teclado es utilizando el mecanismo de la interrupción del reloj de la máquina y utilizando el hecho de que BIOS guarda en una dirección fija de memoria el resultado

de la máscara de estado. Esta dirección es la 0:417 ([HOGT88]) y contiene los bits de acuerdo a la descripción que se muestra en la tabla 1. Así pues, es posible cada vez que se genera una interrupción del reloj, checar la máscara de estado y verificar si coincide con la máscara especificada como condición de activación para verificar que es necesario que sea llamado el programa residente.

En el caso de programas residentes en memoria que utilizan el teclado como via de activación como se dijo anteriormente no se trata de programas corriendo concurrentemente, dado que estos en el caso general no ceden el control a otras aplicaciones mientras se estan ejecutando. Por lo mismo, las consideraciones mencionadas en este trabajo con respecto a concurrencia en DOS se pueden relajar sensiblemente.

Podría ocurrirse que se utilizaran las interrupciones de teclado como un medio de comunicación con el programa residente. Sin embargo, este mecanismo es poco adecuado debido a las posibles interferencias con otros programas residentes ya que seguramente hace que haya incompatibilidades entre los que utilicen dicho medio de de comunicación entre ellos.

Este tipo de aplicaciones generalmente utiliza una interfaz de despliegue en pantalla para interacción con el usuario, de manera que si este es el caso, la definición de lo que es considerado como el ambiente de ejecución del proceso interrumpido es además la memoria de despliegue y sus atributos tales como el tipo de cursor, la página de video, el modo de la pantalla (como puede ser gráfico, color, VGA, EGA), etc. La restricción adicional de guardar este ambiente hace que dichos programas utilicen memoria extra para tales labores, ya que salvar una pantalla de video en modo gráfico puede significar en casos de programas pequeños un sobrepeso tal vez mayor que el del código mismo. Lo anterior hace que muchas de las utilerías de este tipo no soporten estos modos gráficos.

Dado que estos programas tienen además una interfaz de teclado para recepción de comandos ó edición de textos y a partir de allí acceso a DOS al nivel de command.com, es inprescindible que tales programas tengan sus rutinas propias de atención a control-break y error crítico.

Además, los programas activados via teclado pueden iniciar y terminar su ejecución cada vez que son llamados, lo cual hace también necesario que su funcionalidad no se encuentre atada a valores estáticos de inicialización de sus variables. Si el programa termina y reinicia su operación en el punto en que estaba inmediatamente antes de ser

suspendido, lo más recomendable es que abra y cierre los archivos que tiene en uso actualmente de manera que no interfiera con los demás procesos que se encuentran corriendo simultáneamente.

## Apéndice B

**Ejemplos de rutinas críticas de bajo nivel utilizadas por este tipo de desarrollo.**

A continuación se van a detallar las llamadas a DOS descritas a lo largo de este trabajo, así como las rutinas de intercambio de stacks para corrutinas y algunas de las rutinas de atención a interrupciones como ejemplo de su manejo adecuado.

A) Llamadas a DOS.

1) Terminación de un proceso permaneciendo residente.

```
mov ah, 31h         ; código de la función
mov al, xx          ; valor de regreso a DOS
mov dx, memsiz      ; tamaño necesario en memoria
int 21h             ; llamada a DOS
```

El valor que se pone en el registro "AL" es utilizado por el proceso padre a través de la llamada 4dh de DOS o por el mismo DOS a través del comando ERRORLEVEL de ejecución de archivos batch. "DX" contiene el tamaño en segmentos de la memoria que el programa residente solicita a DOS. Usa como unidades segmentos, que son bloques de 16 bytes. No proporciona ningún valor de regreso inmediato.

2) Obtener la dirección del PSP activo.

```
mov ah, 62h          ; código de la función
int 21h              ; llamada a DOS
mov cs:PSP, bx       ; valor del PSP activo
```

El segmento correspondiente al PSP activo es regresado en el registro "BX", en el código mostrado, este es guardado en una variable para su uso posterior. Nótese además que el valor del PSP activo se guarda en una variable del segmento de código para estar accesible al momento de alguna interrupción.

3) Asignar el PSP activo. (No documentada por DOS)

```
mov ah, 50h         ; código de la función
mov bx, cs:PSP      ; segmento del PSP
int 21h             ; llamada a DOS
```

Se le pasa a la llamada a DOS en el registro "BX", la dirección del segmento correspondiente al PSP que se desea activar ([DUNR88], [HYMM87]). Nuevamente, la variable que contiene la dirección del PSP activo se encuentra en el segmento de código y por tanto accesible al momento de una interrupción. No regresa ningún valor.

4) Obtener el DTA activo.

```
mov ah, 2fh         ; código de la función
int 21h             ; llamada a DOS
mov cs:CDTA, bx     ; offset del DTA activo
mov cs:CDTA+2, es   ; segmento del DTA activo
```

Regresa la dirección del DTA activo (área de transferencia de disco) al momento de la llamada. El código mostrado almacena la dirección del DTA en la

variable CDTA ( en el segmento de código ).

5) Seleccionar el DTA activo.

```
push ds                 ; salva el segmento de datos
mov ax, cs:CDTA+2       ; segmento del DTA que se
mov ds, ax              ; va a activar
mov dx, cs:CDTA         ; su posición
mov ah, 1ah             ; código de la función
int 21h                 ; llamada a DOS
pop ds                  ; restablece DS
```

Activa el DTA cuya dirección se encuentra en "DS:DX" al momento de la llamada a DOS. El código mostrado toma ese valor de la variable CDTA (del segmento de código). Salva y restablece el segmento de datos original por si es necesario.

6) Obtener la dirección de una rutina de interrupción.

```
push es                 ; salva ES
mov ah, 35h             ; código de la función
```

```
mov al, xx              ; número de interrupción
int 21h                 ; llamada a DOS
mov ax, cs:xxForward+2  ; segmento de la rutina
mov es, ax              ; de la interrupción
mov cs:xxForward, bx    ; posición de esta
pop es                  ; restablece ES
```

Regresa en "ES:BX" la dirección del manejador de la interrupción proporcionada en el registro "AL" al momento de la llamada a DOS. El código mostrado salva el valor en la variable xxForward. Como este valor va a ser utilizado al tiempo de interrupción para hacer una llamada desde la rutina del programa residente a la rutina original, dicha variable debe estar accesible y por tanto en el segmento de código. Salva y restablece el registro "ES".

7) Asignar rutina de interrupción.

```
push ds           ; salva DS
mov ax, xxInt+2   ; obtiene el segmento de la
mov ds, ax        ; rutina de interrupción
mov bx, xxInt     ; y su posición
mov ah, 25h       ; código de la función
mov al, xxh       ; número de la interrupción
```

```
int 21h             ; llamada a DOS
pop ds              ; recupera DS
```

Asigna al vector de interrupción número xx la rutina de atención guardada en xxInt. Salva y restablece el segmento de datos. No regresa ningún valor.

8) Liberar segmento de memoria a DOS.

```
push es             ; salva ES
mov ax, MemSeg      ; obtiene el segmento de memoria
mov es, ax          ; que se desea regresar a DOS
mov ah, 49h         ; código de la función
int 21h             ; llamada a DOS
pop es              ; recupera ES
jc error            ; checa si la llamada falló
```

Llamada utilizada para regresar la memoria utilizada a DOS durante el proceso de desinstalación del programa residente. El segmento de memoria que se desea regresar se pasa en el registro "ES" al momento de la llamada a DOS. Regresa código de error en el registro "AX" indicado por que la bandera de acarreo se enciende.

9) Modificar tamaño de segmento en memoria.

```
push es                 ; salva ES
mov ax, MemSeg          ; obtiene el segmento de memoria
mov es, ax              ; que se desea modificar
mov bx, NewSize         ; tamaño deseado en memoria
mov ah, 4ah             ; código de la función
int 21h                 ; llamada a DOS
pop es                  ; recupera ES
jc error1               ; checa si la llamada falló
```

Esta función se utiliza para modificar el tamaño de un segmento de memoria solicitado a DOS, en el desarrollo de programas residentes se puede utilizar para reducir el tamaño del segmento de datos del programa a sólo lo necesario para nuestro stack, memoria estática y memoria dinámica. El tamaño de la memoria que es solicitada se pone en el registro "BX" al momento de la llamada a DOS, y en "ES" el segmento de memoria cuyo tamaño se desea modificar. Al regreso de la llamada, la bandera de acarreo indica que algún error ocurrió, regresando en el registro "AX" un código de error y en el registro "BX" el máximo número de párrafos de memoria que fueron obtenidos.

10) Llamada para encontrar la dirección de la bandera de actividad de DOS. (No documentada)

```
push es                     ; salva ES
mov ah, 34h                 ; código de la función
int 21h                     ; llamada a DOS
mov cs:DosCritical, bx      ; Salva posición y segmento
mov cs:DosCritical+2, es;   de DosCritical
pop es                      ; recupera ES
```

Regresa en "ES:BX" la dirección de la bandera de DOS que indica que este está activo. Esta bandera es conocida como DosCritical o InDos ([BOID89] [DUNR88] [HYMM87]). El uso de funciones no documentadas de DOS tiene cierto riesgo en el sentido que los creadores de DOS quedan en libertad de inhabilitarlas en futuras versiones. Sin embargo, dado que estas funciones son utilizadas por algunas de las utilerías externas de DOS (ej. PRINT) esto es poco probable. Estas llamadas funcionan en las versiones de DOS 3.00 a 3.30 al menos.

11) Obtener número de versión de DOS.

```
mov ah, 30h             ; código de la función
```

```
int 21h                ; llamada a DOS
```

Dado que algunas de las funciones descritas en este trabajo son dependientes de la versión de DOS que se esta corriendo, es necesario que el programa residente cheque primero si la versión de DOS soporta la funcionalidad que este requiere. Como valores de regreso de esta llamada a DOS se obtiene en "AL" el número principal de la versión y en "AH" el número secundario. Así pues, corriendo bajo DOS 3.20 después de la llamada a DOS se tiene "AX"=1403h.

12) Obtener información extendida de error (Versiones 3.10 en adelante).

```
xor     bx, bx                      ; clean bx
mov     ah, 59h                     ; GetExtInfo
int     DosInt                      ;
mov     cs:ExtErrAX, ax             ; save info
mov     cs:ExtErrBX, bx             ;
mov     cs:ExtErrCX, cx             ;
mov     cs:ExtErrDX, dx             ;
mov     cs:ExtErrSI, si             ;
mov     cs:ExtErrDI, di             ;
mov     cs:ExtErrES, es             ;
```

A partir de la versión 3.10 de DOS, con la introducción de la información extendida de error, ésta se vuelve parte del ambiente de ejecución que el programa residente debe salvar y reestablecer. Aquí se salva en una estructura a propósito para asignarla luego por medio de la función 5d0ah descrita a continuación.

13) Asignar información extendida de error (Versiones 3.10 en adelante). No documentada por DOS.

```
push    ds                              ; save ds
push    cs                              ; change to cs
pop     ds                              ;
mov     dx, offset cs:ExtErrInfo        ; struct address
mov     ax, 5d0ah                       ; set error info
int     DosInt                          ;
pop     ds                              ; restore ds
```

Esta rutina no esta documentada por DOS ([DUNR88]). Asigna los valores que contiene la estructura ExtErrInfo a sus variables locales usadas para el manejo de la información extendida de error.

Como ejemplo de las rutinas para inicialización e intercambio de stack descritas en la sección de corrutinas se

muestran las siguientes en las que la rutina mostrada es una codifición para el modelo mediano del lenguaje "C":

_funcl hace la inicialización de un stack nuevo, regresa el control a la función que lo llamó. Es llamada desde "C" de la siguiente manera:

```
_funcl( viejo, nptr )
int **viejo, *nptr;
```

Aquí, * viejo es un apuntador al stack actual, nptr es un apuntador al stack nuevo que se va a inicializar. La variable viejo debe pasarse como parámetro por referencia para que pueda ser modificada con el valor actual al momento del intercambio de stacks.

```
        public _funcl
_funcl  proc    far
        mov     cs:savbp, bp     ; Hay que salvar el registro
        push    bp               ; bp actual en el segmento
                                 ; de código.
        sub     sp, 2            ; Alineación de stack, (igual
                                 ; al de la función que llamó)
        mov     bp, sp           ; Utilizamos bp para obtener
                                 ; nuestros parámetros.
```

```
push    ax              ; Segmento de código al que
mov     ax, [bp+6]      ; hay que regresar el control
mov     cs:savcs, ax    ; al final de _func1.
mov     ax, [bp+4]      ; Apuntador a la instrucción
mov     cs:savip, ax    ; para regresar al final.
mov     ax, [bp+10]     ; Stack nuevo a inicializar.
mov     cs:savp1, ax    ;
mov     ax, [bp+8]      ; Stack anterior o viejo.
mov     cs:savp2, ax    ;
pop     ax              ;
mov     sp, [bp+2]      ; Usa stack de quien llamó.
push    ax              ; Salva todos los registros.
push    bx
push    cx
push    dx
push    si
push    di
push    ds
push    es
mov     si, cs:savp2    ; Salva el valor viejo
mov     [si], sp        ; del stack.
mov     ax, cs:savp1    ; Obtiene el nuevo stack.
mov     sp, ax          ;

mov     ax, cs:savcs    ; Simula en el nuevo
push    ax              ; stack una llamada por
mov     ax, cs:savip    ; la función original.
push    ax              ;
mov     ax, cs:savp1    ; Pone bp apuntando al
```

```
        mov     bp, ax              ; tope del nuevo stack
                                    ; siguiendo convención
                                    ; de llamadas de "C".
        retf                        ; Regreso de función.
_func1  endp
```

_func2 salva el contexto de ejecución, intercambia stacks y recupera el contexto a partir de un estado viejo guardado en el stack que se acaba de obtener.

Es llamado desde "C" de la siguiente manera:

```
_func2( viejo, nptr )
int **viejo, *nptr;
```

Los parámetros aquí son los mismos que en _func1 y tienen el mismo sentido.

```
        public  _func2
_func2  proc    far
        push    bp                  ; Salva el valor de bp, ya
                                    ; puede ser en el stack.
        sub     sp, 2               ; Alineamiento.
```

```
mov     bp, sp          ; Usa bp para obtener sus
                        ; parámetros.
push    ax              ; Salva los registros.
push    bx
push    cx
push    dx
push    si
push    di
push    ds
push    es

mov     si, [bp+8]      ; Salva el valor del stack
mov     [si], sp        ; en *viejo.
mov     ax, [bp+10]     ; Obtiene el nuevo stack
mov     sp, ax          ; Lo cambia.

pop     es              ; Recupera registros del
pop     ds              ; nuevo stack.
pop     di
pop     si
pop     dx
pop     cx
pop     bx
pop     ax

add     sp, 2           ; Restablece el stack
pop     bp              ; Recupera el bp.

retf                    ; Regreso.
```

```
_func2  endp
```

Como ejemplo de rutinas de interrupción propias de un programa residente usadas como sustituto de las originales proporcionadas por DOS.

Las rutina de atención a la interrupción de disco y video aquí descritas son muy sencillas y son usadas simplemente para saber cuando se encuentra la máquina adentro de alguna de estas interrupciones.

```
DiskIntercept        proc    far
    pushf                               ; Salva las banderas.
    inc     word  ptr cs:DiskBusy       ; Indica disco en uso.
    call    dword ptr cs:DiskForward    ; Llama rutina original.
    dec     word  ptr cs:DiskBusy       ; Apaga disco en uso.
    ret     2                           ; Regresa interrupción
                                        ; respetando banderas
DiskIntercept   endp
```

```
VideoIntercept       proc    far
    pushf                                ; Salva las banderas.
```

```
    inc     word  ptr cs:VideoBusy     ; Indica video activo.
    call    dword ptr cs:VideoForward  ; Llama rutina original.
    dec     word  ptr cs:DiskBusy      ; Apaga video activo.
    iret                               ; Regresa interrupción.
VideoIntercept   endp
```

La rutina de atención a la interrupción del reloj es más compleja, ya que es la entrada principal de control a procesos residentes corriendo en paralelo con la aplicación.

```
TimerIntercept  proc    far
    push    bp                      ; Hay que recuperar las
    mov     bp, sp                  ; verdaderas banderas del
    push    ax                      ; stack, ya  que esta es
    mov     ax,  ss:[bp+6]          ; una interrupción de
    push    ax                      ; hardware.

    jmp     $+3                     ; Hay que darle la vuelta
caaaa:                              ; al bug del 80286, no
    iret                            ; sabemos en que máquina
    push    cs                      ; va a correr el programa
    call    caaaa                   ;
    pop     ax                      ; Reestablece todo.
    pop     bp                      ;

    pushf                           ; Simula interrupción.
```

```
        call    dword ptr cs:TimerForward;

        push    bp                      ; Esta danza es para
        mov     bp, sp                  ; guardar los flags que
        push    ax                      ; vamos a regresar al
        pushf                           ; final de la rutina de
        pop     ax                      ; regreso a quien tenía
        mov     ss:[bp+6], ax           ; el control al momento
        pop     ax                      ; de la interrupción.
        pop     bp                      ;
        sti                             ;

        cmp     cs:done, 0              ; Hay algo que hacer?
        jne     ti200                   ; No, termina.

        inc     cs:NeedPopup            ; Chequeo para manejo de
        cmp     cs:running, 1           ; cuanto tiempo ha tenido
        jne     lx                      ; el control el programa
        inc     cs:ticksrun             ; residente.
lx:

        cmp     cs:CriticalCount,0      ; Chequeo de reingreso
        jne     ti200                   ; BG activo, termina.
        inc     cs:CriticalCount        ; Prende control
        push    ax                      ; Tiempo transcurrido es
        mov     ax, lambda              ; mayor que lambda?
        cmp     cs:NeedPopup,ax         ;
        pop     ax                      ;
        jl      ti050                   ; Si, termina.
```

```
        push    bx                      ; Chequeo de DOS activo
        push    es                      ;
        les     bx,DosCritical          ;
        cmp     word ptr es:[bx-1],0    ;
        pop     es                      ;
        pop     bx                      ;
        jne     Ti050                   ; DOS activo, termina.

        push    ax                      ; Chequeo Disco activo.
        mov     ax, word ptr cs:DiskBusy;
        cmp     ax, 0                   ;
        pop     ax                      ;
        jg      ti050                   ; Ocupado, termina.

        push    ax                      ; Chequeo Video activo.
        mov     ax, word ptr cs:VideoBusy;
        cmp     ax, 0                   ;
        pop     ax                      ;
        jg      ti050                   ; Ocupado, termina.

        call    popup                   ; Cede control al
                                        ; programa residente.

        mov     running, 0              ; Señala no activo.

Ti050:
```

```
        dec     cs:criticalcount          ; Actualiza control de
                                          ; reingreso.
    Ti200:                                ;
TimerIntercept  endp
```

## Apéndice C

**Ejemplos de programas en lenguaje "C" y rutinas de alto nivel desarrolladas o usadas con esta tecnología.**

A continuación muestro el esqueleto de dos programas en, que intercambian información por medio de semáforos de manera que se evita el abrazo mortal o "DeadLock", uno de ellos con 2 corrutinas.

Las rutinas de semáforo o candados que se utilizan son:

initlocks() hace la necesaria inicialización para que sirvan las funciones de candados, (abre archivo de candados, inicializa contadores de semáforos, etc.).

deinitlocks() regresa el sistema a su estado original (cierra archivo de candados, etc.)

lock(n) Implementa internamente el algoritmo de semáforo con un contador en caso que sea éste el proceso que obtuvo el candado para esa localidad. Si el contador asociado a la localidad n es mayor que 0, lo incrementa y regresa TRUE, si no, trata de poner un candado en la localidad n de disco, reintenta varias veces antes de fallar. Regresa TRUE si lo consiguió incrementando el contador, y FALSE en otro caso.

release(n) Decrementa el contador asociado al semáforo n. Si el contador es igual a cero, libera el candado de la localidad n del archivo de candados.

test(n) Verifica si el semáforo está puesto y lo pone si no es el caso. Regresa TRUE si lo puede poner, FALSE si no. No hace reintentos.

Programa A. Este programa se sincroniza con el programa B por medio de el mecanismo de semáforos para intercambiar información con él. Es un ejemplo de programa externo que se comunica con un programa residente en memoria.

```
#include <stdhdr.h>

main() {
    initlocks(); /* hace la inicialización necesaria    */
```

```
                /* de manera que la función lock sirva   */
                /* como mecanismo de semáforos           */
    if (!lock(0)) {
                /* se evita deadlock en el caso en que   */
                /* dos programas del tipo de A quieren   */
                /* comunicarse simultáneamente con B.    */
        exit(error("Ocupado. Intente mas tarde.\r\n"));
    }
    lock(1);    /* indica al programa B que se quiere    */
                /* comunicar con él                      */
    while (test(2)) {
                /* ciclo de espera a que B se encuentre */
                /* en un lugar adecuado                  */
        release(2);
    }
    /* Aquí ocurre el proceso de comunicación de A hacia B  */
    while (!test2); /* Espera a que B continúe su operación */
    release(2);
    release(0);    /* Permite a otros procesos interacción  */
                   /* con el programa B                     */
    deinitlocks(); /* Reestablece el estado original        */
}
```

Programa B. Este es un ejemplo de un programa residente en memoria que se comunica utilizando el mecanismo de sincronización por medio de semáforos, con el programa A mostrado anteriormente. Para que sirva además como ejemplo

de la utilización de corrutinas, este cuenta con dos de ellas. Se incluye un ejemplo simple de despachador de procesos.

```
#include <stdhdr.h>


bool Corr1_activa  = TRUE; /* indican si la corrutina está parada */
bool Corr2_activa  = TRUE;

bool Corr1_primera = TRUE; /* indican si dicha corrutina no ha  */
bool Corr2_primera = TRUE; /* sido llamada anteriormente        */

ushort SizeMain    = 0x1000, /* tamaños de stack asignados al main, */
       SizeCorr1   = 0x1000, /* corrutina_1 y corrutina_2           */
       SizeCorr2   = 0x2000;

int _STACK = 0x4000;      /* esta variable debe tomar el valor de */
                          /* al menos la suma de los tamaños de   */
                          /* los stacks que se vayan a utilizar   */
                          /* es con la que se determina cuanta    */
                          /* memoria para stack usa el programa   */

int _POOLMAX = 0x2000;    /* tamaño de memoria dinámica que será  */
                          /* usada por el programa                */


extern int _TOP;          /* Valor Inicial del Stack del Programa */
```

```
long *StackMain, *StackCorr1, *StackCorr2;
                        /* variables usadas para                */
                        /* guardar los apuntadores a los stacks */
                        /* de cada una de las corrutinas        */

bool para_corrutinas=FALSE;

main() {
    initlocks();        /* prepara la utilización de candados   */
    while (TRUE) {      /* ciclo infinito                       */
        while (HayCorrutinaActiva()) {
                        /* espera el señalamiento de las        */
                        /* corrutinas para pasar al punto       */
                        /* de la intercomunicación              */
            CheckTime(1); /* despachador de corrutinas          */
        }
        lock(2);        /* señala que se esta en lugar seguro   */
                        /* aqui ocurre la comunicación          */
        while (!test(1)); /* espera el señalamiento de fin del  */
                          /* proceso de comunicación            */
        release(1);
        release(2);       /* libera candado para indicar que ya */
                          /* va a reiniciar su actividad normal */
    }
    para_corrutinas=FALSE; /* permite que se activen estas      */
}

HayCorrutinaActiva() {
```

```
    return (Corr1_activa || Corr2_activa);
}


Corrutina_1() {
    Corr1_activa=TRUE;
    while (!para_corrutinas) {
        Corr1_main();   /* operación normal de la corrutina_1   */
        CheckTime(1);   /* llama al despachador de corrutinas   */
    }
    while (para_corrutinas);    /* espera fin de comunicación   */
    Corr1_activa=FALSE;
}

Corrutina_2() {
    Corr2_activa=TRUE;
    while (!para_corrutinas) {
        Corr2_main();   /* operación normal de la corrutina_2   */
        CheckTime(1);   /* llama al despachador de corrutinas   */
    }
    while (para_corrutinas) ;    /* espera fin de comunicación  */
    Corr2_activa=FALSE;
}

int StackMode=0;

CheckTime(n)
int n;  {
    if (Corr1_primera) {
```

```
        StackCorr1= (long *) (_TOP - SizeMain); /* apunta al stack */
        _func1(&StackMain, StackCorr1); /* inicializa el stack     */
                        /* nuevo y los cambia, salva el actual     */
        StackMode=1;
        Corr1_primera=FALSE;     /* para no llamar dos _func1      */
                                 /* con el mismo stack             */
    }
    if (Corr2_primera) {
        StackCorr2= (long *) (_TOP - SizeMain - SizeCorr1);
                                 /* apunta el nuevo stack, lo   */
                                 /* inicializa, cambia y salva  */
        _func1(&StackCorr1, StackCorr2); /* el actual           */
        StackMode=2;
        Corr2_primera=FALSE;     /* para no llamar dos _func1   */
                                 /* con el mismo stack          */
    }
/* Este algoritmo hace un simple esquema de "round robin" o     */
/* todos contra todos. _check es el despachador de procesos     */
/* en el caso de un programa residente o bien una función no-op */
/* en el caso de programas con corrutinas no residentes         */

    switch(StackMode) {
    case 0:
        _check(n);               /* despachador de procesos     */
        _func2(&StackMain, StackCorr1;  /* intercambia stacks   */
        StackMode = 1;           /* indica nuevo stack activo   */
        break;
    case 1:
        _check(n);               /* despachador de procesos     */
```

```
        _func2(&StackCorr1, StackCorr2); /* intercambia stacks  */
        StackMode = 2;            /* indica nuevo stack activo   */
        break;
    case 2:
        _check(n);                /* despachador de procesos     */
        _func2(&StackCorr2, StackMain); /* intercambia stacks   */
        StackMode = 0;            /* indica nuevo stack activo   */
        break;
    }
}
```

Programa C. Este es un ejemplo simple de la aplicación de la tecnología descrita en la tesis. Se trata de un programa que calcula la hora y la despliega en la esquina superior derecha de la pantalla, puede opcionalmente recibir como parámetros para determinar la posición de éste. La escritura a pantalla es mediante BIOS para simplificar el programa y funcionar independientemente del modo gráfico.

```
#include <stdhdr.h>
#include <stdlow.h>

int _POOLMAX=0x100;      /* memoria dinámica utilizada   */
int _STACK=0x300;        /* stack utilizado              */

int Vrow=0,  Vcol=70;    /* posición default en pantalla del reloj */
```

```
char BGSIGNATURE[9]="ResClock";
                                /* firma para reconocer instalación     */

int atoin(s,n)
char *s;
int n; {
int r=0;
    while (n-- && isdigit(*s)) {
        r = r*10 + *s - '0';
        s++;
    }
    return r;
}

    /* Esta función se incluye para no cargar código de */
    /* la familia de printf, sprintf y compañía          */

numtostr(num,s)
int num;
char *s; {
    *s++=num/10+'0';
    *s++=num%10+'0';
}

showtime() {
struct xregs regs;
struct SREGS segregs;
int savcurpos;
```

```
int savcurmod;
char mode,page;
char datebuff[9], *p;
int hour, minute, seconds;

    p=datebuff;
    xbdos(0x2c,&regs); /* obtiene hora de DOS   */

                /* convierte la hora a un formato legible */

    hour=regs.cx>>8;
    minute=regs.cx & 0xff;
    seconds=regs.dx>>8;

    numtostr(hour,p);
    p+=2;
    *p++=':';

    numtostr(minute,p);
    p+=2;
    *p++=':';

    numtostr(seconds,p);
    p+=2;
    *p++=0;

              /* busca la página y modo activos de video */

    regs.ax=0x0f00;
```

```
    xbios(0x10,&regs);
    mode = regs.ax & 0xff;
    page = (regs.bx >> 8) & 0xff;

    segread(&segregs);

    regs.ax=0x1300;              /* escribe sin mover el cursor    */
    regs.bx=(page << 8) | 0x07;/* en la página y atributo        */
    regs.cx=strlen(datebuff);  /* tamaño del mensaje             */
    regs.dx=(Vrow<<8) | Vcol;  /* renglón, columna               */
    regs.bp=(bits) datebuff;    /* mensaje                       */
    regs.es= segregs.es;
    xbios(0x10,&regs);
}

main(argc, argv)
int argc;
char *argv[]; {
char *p, ch;

        /* proceso de la linea de comandos! (fácil gracias a "C") */
    while (--argc) {
        p=*++argv;
        if (*p++!='-') continue;
        ch=*p++;
        switch (toupper(ch)) {
        case 'C':
            Vcol=atoin(p,2);
            break;
```

```
        case 'R':
            Vrow=atoin(p,2);
            break;
        default:
            break;
        }
    }

    while (TRUE) {
        showtime();      /* muestra el reloj */
        _check(1L);      /* llama al despachador de procesos */
    }
}

exit() {          /* ficticias para no cargarlas con su código */
_exit() {         /* de manejo de archivos en modo indirecto   */
```

## Bibliografía.

[BOID89]

Boiling, D. "Background copying without OS/2", PC magazine, volumen 8, número 1, Enero 17, 1989, páginas 289-315, New York, NY: Ziff-Davis Publishing Company. (ISNN 0888-8507)

[DUNR86]

Duncan, R. "Advanced MS-DOS". Redmond, WA: Microsoft Press, 1986. (QA76.76.063D858 1986 005.4'46 86-8496) (ISBN 0-914845-77-2)

[DUNR88]

Duncan, R. "The MS-DOS Encyclopedia". Redmond, WA: Microsoft Press, 1988. (QA76.76.063M74 1988 87-21452 005.4'46-dc19 CIP) (ISBN 1-55615-049-0)

[HOGT88]

Hogan, T. "The programmer's PC sourcebook". Redmond, WA: Microsoft Press, 1988. (QA76.8.I1015H64 1988 87-36575 005.4'469--dc19 CIP) (ISBN 1-55615-118-7)

[HYMM87]

Hyman, M. I. **"Memory resident utilities, interrupts and disk management with MS & PC DOS".** Portland, OR: Management Information Source, Inc. 1987. (ISBN 0-943518-73-3)

[IBMC87]

IBM Corporation. **"Disk Operating System Version 3.30: Thecnical Reference".** First Edition. Boca Raton, FL: IBM Corporation, Abril, 1987. (IBM part number 80X0945)

[IBMC84]

IBM Corporation. **"Technical Reference PC Network".** First Edition. Boca Raton, FL: IBM Corporation, Septiembre, 1984. (IBM part number 6322916)

[IBMC85]

IBM Corporation. **"Technical Reference: Personal Computer AT".** First Edition. Boca Raton, FL: IBM Corporation, Septiembre, 1985. (IBM part number 6139362)

[INTC85]

Intel Corporation. **"iAPX 286 Programmers's Reference Manual**

including the iAPX 286 Numeric Supplement". Santa Clara, CA: Intel Corporation, 1985. (Intel part number 210498-003)

[KERB78]
Kernighan, B. W., Ritchie, D. M. "The C programming language". Englewood Cliffs, NJ: Prentice-Hall, Inc. 1978. (QA76.73.C15K47 001.6'424 77-28983) (ISBN 0-13-110163-3)

[MENC89]
Menico, C. "Debugging TSR programs". Dr. Dobbs's Journal of Software Tools, volumen 14, número 2, Febrero, 1989, páginas 67-70 y 104-106. Redwood City, CA: M&T Publishing Inc. (ISNN 0888-3076)

[MICC85]
Microsoft Corporation. "Microsoft Macro Assembler for the MS-DOS Operating System". Users Guide. Redmond, WA: Microsoft Corporation. 1985. (Part Number 016-014-023) (Document Number 410610001-400-R00-0985)

[STEA88a]
Stevens, A. "Writing Terminate and Stay-Resident Programs (Part I: TSRs in Turbo C)", Computer Language, volumen 5,

número 2, Febrero, 1988, páginas 37-49. San Francisco, CA: Miller Freeman Publications. (ISNN 0749-2839)

**[STEA88b]**

Stevens, A. **"Writing Terminate and Stay-Resident Programs (Part II: TSRs in C 5.0 and Quick C)"**, Computer Language, volumen 5, número 3, Marzo, 1988, páginas 67-76. San Francisco, CA: Miller Freeman Publications. (ISNN 0749-2839)

**[WULW81]**

Wulf, W. A., Shaw, M., Hilfinger, P. N., Flon, L. **"Fundamental Structures of Computer Science"**. Reading, MA: Addison-Wesley Publishing, 1981. (QA76.7.F86 001.6'42 79-12374) (ISBN 0-201-08725-1)