Universidad
Carlos III de Madrid

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA INFORMÁTICA

Proyecto Fin de Carrera

Diseño e implementación de un Sistema Operativo para fines didácticos

Autor: Aníbal Ramírez García
Tutor: Francisco Javier García Blas

Leganés, Mayo 2012

AGRADECIMIENTOS

En primer lugar quiero mostrar todo mi agradecimiento a mi tutor Francisco Javier García Blas por la confianza e interés que me ha prestado, pues pienso que posiblemente sin su apoyo no hubiera podido culminar este proyecto.

Agradezco también igualmente a José Daniel García Sánchez por sus consejos, su interés y su apoyo, porque me ha ayudado en gran medida a seguir adelante con el proyecto.

Deseo mencionar también el apoyo y soporte prestados por mi compañero de trabajo Pedro Pablo López Rodríguez, cuya ayuda ha sido inestimable y sin la cual este trabajo hubiera resultado bastante más arduo.

No quiero olvidar a mis compañeros y amigos de trabajo que me han mostrado en todo momento su confianza, no dejándome con sus palabras de ánimo, que desatendiera este trabajo.

A todos los que me he dejado sin mencionar, que sepan que también les tengo muy presentes y les agradezco todo su apoyo mostrado.

Y por supuesto, quiero agradecer también a mi hija y mi mujer todo el ánimo y apoyo que en todo momento me han dado.

Resumen

Actualmente podemos encontrar en el mercado una gran variedad de sistemas operativos que satisfacen múltiples propósitos. Algunos de estos sistemas fueron diseñados teniendo como uno de sus principales objetivos el servir de herramienta de apoyo en entornos educativos, y más precisamente, unos pocos de ellos se diseñaron con la finalidad básica de servir como herramienta de apoyo en la impartición de la asignatura "Sistemas Operativos". No obstante, debido a la diversidad de estos entornos educativos, en algunos casos es difícil, si no imposible, encontrar un sistema lo suficientemente satisfactorio. Para estos casos, puede estar justificado el desarrollo de un S.O. a la medida.

Este proyecto presenta el diseño e implementación de un sistema operativo con capacidades básicas, pero suficientes, como para poder ser utilizado como herramienta de apoyo a la enseñanza de la asignatura "Sistemas Operativos". El proyecto cubre las necesidades específicas de un entorno de enseñanza concreto. Algunas de éstas son: uso del lenguaje 'C'; un entorno de desarrollo amigable con facilidades para la edición, compilación y depuración integradas; uso de máquinas virtuales mediante software de emulación; diseño e implementación dando preferencia a la sencillez frente a la eficiencia; una amplia y detallada documentación; etc.

El sistema se ha desarrollado para una plataforma Intel x86 en modo real, pudiendo correr tanto en hardware desnudo tipo PC compatible, como en máquina virtual. Algunas de sus características más relevantes son: gestión de procesos con capacidad multitarea y reubicación dinámica, gestión de memoria con particiones de tamaño variable, sistema de ficheros tipo FAT12/16, sistema de múltiples ventanas terminal tipo texto, etc.

El trabajo finaliza presentando unas conclusiones y líneas de trabajo futuro, y también algunas ideas y trabajos prácticos a realizar por el estudiante sobre este sistema.

Abstract

Today we can find on the market a variety of operating systems which serve multiple purposes. Some of these systems were designed with the main target of being used as a teaching support tool, and more precisely, a few of them were designed with the basic purpose of serving as a support tool in teaching the subject "Operating Systems"; however, due to the variety of educational environments, in some cases, it is difficult if not impossible, to find a good-enough system to be pleased. In these cases, it may be justifiable to develop a customized O.S.

This work presents the design and implementation of an operating system with basic skills but good enough to be used as a teaching support tool for the subject: "Operating Systems". The project covers the specific needs of a particular environment teaching. Some of them are: use of "C" language; a friendly development environment with integrated features for editing, compiling and debugging; use of a virtual machine by means of software emulation; design and implementation preferring simplicity versus efficiency; wide and detailed documentation; etc.

The system has been developed for an Intel x86 real mode, being able to run both on bare hardware PC compatible as in a virtual machine. Some of its more relevant features are: process management with multitasking and dynamic relocation, memory management with variable sized partitions, FAT 12/16 file system, multiple terminal text windows, etc.

The project ends by presenting some conclusions and future lines of work, and also some ideas and practical works to be carried out by the student on this system.

ÍNDICE DE CONTENIDOS

ILUSTRACIONES Y TABLAS

1. INTRODUCCIÓN

El presente trabajo desarrolla e implementa un sistema operativo básico, cuyo objetivo primordial es su uso como herramienta en la enseñanza de los fundamentos del diseño y construcción de sistemas operativos para ordenadores. Este sistema tiene como alguna de sus características: estar programado en lenguaje "C", es multitarea, dispone de gestión de memoria dinámica, gestión de ficheros, operaciones de entrada/salida, múltiples ventanas de texto, etc.

Este sistema operativo está diseñado para correr en un tipo de ordenador "PC compatible", basado en el procesador Intel 8088/86, que aunque es ya bastante antiguo, sigue vigente como modelo usado en la enseñanza, dada su relativa simplicidad y la amplia gama de herramientas y documentación disponible, todo ello de fácil acceso. Por mencionar alguna de estas facilidades, cabe citar la gran disponibilidad actual de acceso al *software* de emulación de máquinas virtuales, como por ejemplo: DOSBox, Qemu, Virtual PC, VirtualBox, VMWare, etc. Gracias a ello, tanto el entorno de desarrollo, como la posible realización de prácticas para dicho sistema, se tornan muy asequibles, y aportan al conjunto características muy deseables e interesantes para su uso en el entorno enseñanza.

1.1 Motivación.

La idea de este proyecto surgió debido a la necesidad de contar con un sistema operativo lo suficientemente sencillo como para no perderse en los detalles y complejidades del software que lo compone, y lo suficientemente capaz y versátil como para ser puesto como ejemplo de las posibilidades, y capacidades que aporta un sistema operativo a la máquina, al transformarla en una máquina extendida de uso más simple y transparente desde el punto de vista del usuario.

La necesidad de un sistema así en la enseñanza parece natural, ya que los sistemas operativos comerciales son enormemente complejos, y es difícil mostrar los detalles de su diseño y construcción sin tener que entrar en muchas complicaciones. Además el acceso al código fuente de los mismos no suele estar muchas veces disponible, aunque es cierto, que en la actualidad existe bastante software abierto que podría utilizarse a tal efecto, si obviamos tanto la complejidad del mismo como la necesidad de tener que estudiarlo en profundidad para su adecuada exposición en clase.

Como se ha comentado anteriormente, existen ya algunos sistemas no muy complejos que están dedicados principalmente a la enseñanza. Un ejemplo claro de ello lo constituye el sistema operativo “Minix”, al menos en sus primeras versiones. Dicho sistema no es demasiado complejo, y existe bastante información sobre el mismo como para ser un buen candidato a ser utilizado como herramienta de apoyo en la enseñanza de los sistemas operativos. De hecho, el autor de este Proyecto Fin de Carrera lo ha utilizado para ese propósito durante bastante tiempo. No obstante, desde cierto punto de vista, adolece todavía del inconveniente principal, su complejidad. La realización de trabajos prácticos requiere un extenso conocimiento del mismo y se hace difícil pedir al alumno que realice modificaciones y mejoras sobre el código, pues como puede suponerse, pedir una modificación suele requerir el conocimiento previo de lo ya existente, y eso es algo que muchas veces no es trivial. Otro inconveniente que se ha encontrado es que las herramientas y útiles de desarrollo, tales como el compilador, editor, depurador, etc. suelen ser bastante complejas, constituyendo una barrera más a sortear por parte de alumno, para la adquisición de nuevo conocimiento sobre el sistema operativo. Todo esto es bastante accesorio, por lo que el alumno no debería tener que emplear demasiado tiempo en su aprendizaje, para así poder dedicarse plenamente al aprendizaje de los conceptos fundamentales.

Una vez asumidas las “limitaciones” que impone el uso de un sistema como “Minix”, y haber descartado otros sistemas de uso didáctico por razones más o menos similares, se consideró interesante y deseable, desarrollar como proyecto fin de carrera un Sistema Operativo propio. Se podría hacer a la medida de las necesidades específicas para la enseñanza que se habían encontrado, y podría darle un uso práctico real. No debería sufrir de los inconvenientes ya mencionados y podría servir de plataforma para una futura ampliación. Además al aportar suficiente información técnica, sería posible su utilización por parte de otros educadores, sin que ello tuviera que requerir un gran esfuerzo en su aprendizaje.

1.2 Objetivos.

El objetivo fundamental de este Proyecto Fin de Carrera es conseguir que el estudiante disponga de una herramienta útil y adecuada, la cual le facilite el aprendizaje de los fundamentos del diseño y construcción de los sistemas operativos.

Para llevar a cabo este objetivo, se diseñará e implementará un sistema operativo sencillo que pueda correr mediante software de virtualización tipo DOSBox, Qemu, etc. y que esté desarrollado en un entorno que permita efectuar las operaciones de compilación, edición y depuración de forma integrada, todo ello en aras a facilitar al estudiante las tareas de modificación y ampliación de las funcionalidades del sistema, facilitando así una comprensión más amplia del mismo.

Dicho sistema se completará, en la medida de lo posible, con la elaboración de unos trabajos prácticos basados en dicho sistema, con objeto de que los realice el estudiante, para que de este modo se faciliten, mejoren y amplíen los conocimientos del estudiante sobre los conceptos fundamentales en los que se basa el diseño y la construcción de los sistemas operativos.

1.3 Definiciones y Acrónimos.

A continuación, se definen las palabras técnicas y la nomenclatura utilizada.

1.3.1 Definiciones.

- *Sistema Operativo*: Programa o conjunto de programas que efectúan la gestión de los procesos básicos de un sistema informático, y permite la normal ejecución del resto de las operaciones.

- *Round-robin*: "Turno circular". Es un método de planificación para seleccionar todos los elementos en un grupo de manera equitativa y en un orden, comenzándose normalmente por el primer elemente de la lista hasta llegar al último y volviendo a empezar desde el primero.

- *Bus*: Sistema digital que transfiere datos entre los componentes de un ordenador o entre varios ordenadores.

- *Kernel*: Parte fundamental de un programa, por lo general de un sistema operativo, que reside en memoria todo el tiempo y que provee los servicios básicos. Es la parte del sistema operativo que está más cerca de la máquina y puede activar el hardware directamente o unirse a otra capa de software que maneja el hardware.

- *Driver:* programa informático que permite al sistema operativo interactuar con un periférico, haciendo una abstracción del hardware y proporcionando una interfaz posiblemente estandarizado para usarlo

- *Paragraph:* Unidad de memoria equivalente a 16 bytes u octetos.

- *Cluster:* Conjunto contiguo de sectores que componen la unidad más pequeña de almacenamiento de un disco.

1.3.2 Acrónimos.

Acrónimo	Significado
MS-DOS	*Microsoft - Disk Operating System.*
UNIX	*Uniplexed Information and Computing Service* (*no* es exactamente un acrónimo)
POSIX	*Portable Operating System Interface (la X viene de Unix)*
FAT	*File Allocation Table*

SCO	*Santa Cruz Operation*
BSD	*Berkeley Software Distribution*
RAE	*Real Academia Española*
GNU	*General Public License*
RAM	*Random Access Memory*
ROM	*Read Only Memory*
EEPROM	*Electrically Erasable ROM*
BIOS	*Basic Input Output System*
CMOS	*Complementary Metal Oxide Semiconductor*
MMU	*Memory Management Unit*
DMA	*Direct Memory Access*
GUI	*Graphics User Interface*
IDE	*Integrated Developing Environment*
USB	*Universal Serial Bus*
DLL	*Dynamic Link Library*
COCOMO	*Constructive Cost Model*
PYMES	*Pequeñas y Medianas Empresas*
RTI	*Rutina de Tratamiento de Interrupción*

Tabla 1: Definiciones y Acrónimos

1.4 Descripción de capítulos.

En este punto se presenta de forma resumida el contenido de cada uno de los capítulos que forman este trabajo.

En el Capítulo 1, "*Introducción*" se presenta este trabajo mostrando de forma resumida en qué consiste y cuáles son sus objetivos.

En el Capítulo 2, "*Estado de la cuestión*" se hace una rápida revisión de los fundamentos de los sistemas operativos, y se comentan las características más relevantes de algunos sistemas opera-

tivos próximos o relacionados con los objetivos de este proyecto. Especialmente de aquellos que se han desarrollado principalmente como herramienta de apoyo a la enseñanza de los principios básicos de diseño de estos sistemas.

En el Capítulo 3, "*Gestión de proyecto*" se comenta la metodología seguida para la realización del proyecto, así como el software utilizado para el control de versiones.

En el Capítulo 4, "*Objetivos*" se señalan los objetivos básicos del proyecto así como también los objetivos secundarios.

El Capítulo 5, "*Memoria del trabajo*" se realiza una exposición relativamente detallada del diseño e implementación del sistema operativo, así como detalles de funcionamiento, instalación, requisitos, etc.

El Capítulo 6, "*Resultados*", Se muestran algunas de las múltiples pruebas realizadas y resultados obtenidos por los comandos internos y externos del sistema, así como también algunos programas de usuario y trabajos prácticos a realizar por el alumno.

El Capítulo 7, "*Conclusiones*", se exponen las conclusiones obtenidas al final del desarrollo del proyecto, así como futuras líneas de desarrollo y ampliación y una valoración o presupuesto del proyecto.

El Capítulo 8, "*Referencias*", relaciona la bibliografía de consulta y enlaces URL a páginas web de interés para el proyecto.

El Capítulo 9: "*Anexos*", detalla el contenido del CD que se adjunta con esta memoria.

2. ESTADO DE LA CUESTIÓN

En este apartado se muestra en principio una visión general de los fundamentos de los sistemas operativos, y seguidamente se detallan las características más relevantes de algunos de ellos. Especialmente se profundizará en aquellos que se han utilizado principalmente como herramienta de apoyo a la enseñanza de los principios básicos de diseño de estos sistemas.

2.1 Fundamentos de los Sistemas Operativos.

2.1.1 Qué es un Sistema Operativo

Según el diccionario de la RAE, el Sistema Operativo podría definirse así: "*Programa o conjunto de programas que efectúan la gestión de los procesos básicos de un* ***sistema*** *informático, y permite la normal ejecución del resto de las operaciones*". No obstante aún dada esta breve y concisa definición, resulta difícil saber exactamente qué es lo que consideramos un Sistema Operativo. Ello es así porque entre otras cosas el Sistema Operativo realiza básicamente dos funciones no relacionadas. Por una parte extiende la máquina real, ampliando sus posibilidades y ocultando su complejidad, y por otra, realiza una gestión de los recursos de la misma [*TAWO98*].

Para entender mejor cómo se extiende una máquina real pensemos por ejemplo cómo hay que programar una operación de entrada y salida sobre un disquete en una máquina desnuda. Si estudiamos la documentación de la controladora de disco de un ordenador típico de sobremesa, veremos que ésta encierra una gran complejidad. Para llevar a cabo una simple operación elemental de lectura o escritura, hay que programar bastantes instrucciones en código máquina, y con todo ello sólo habríamos resuelto una pequeña fracción del problema. La visión física del disco la componen pistas, cabezas y sectores, y tanto escribir información, como su recuperación, acarrearía llevar el control de dónde se encuentra ésta. Un concepto tan familiar hoy día, como lo es la abstracción "fichero", es algo que no se encuentra a nivel máquina, y por tanto no es posible realizar una operación tan básica como es leer o escribir una determinada información en el fichero, sin tener que ser consciente de los innumerables detalles que exige el nivel de programación de lenguaje máquina. Esta abstracción y otras muchas, es algo que nos ofrece el Sistema Operativo. Nos presenta una máquina "extendida o virtual" que dispone de unas características mejoradas, con las que podemos trabajar de una forma mucho más cómoda y sencilla. En resumen, el sistema operativo ofrece al programador una gran variedad de servicios, que se pueden obtener mediante el uso de lo que denominamos comúnmente, "llamadas al sistema".

Por otro lado, también es posible contemplar al sistema operativo, fundamentalmente como un administrador de un sistema con un cierto grado de complejidad, lo que corrientemente conocemos como "ordenador", el cual generalmente está compuesto de procesadores, memorias, discos, temporizadores, ratones, conexiones de red, impresoras, y una gran variedad de otros dispositivos. La administración consistiría en asegurar el reparto equilibrado de estos recursos entre los diferentes procesos que se ejecutan en el sistema. Se hace necesario proteger la memoria, los dispositivos de entrada/salida, y muchos otros recursos de un acceso indiscriminado, sobre todo para evitar conflictos entre procesos (y usuarios, si el sistema operativo fuera multiusuario).

Otra posible forma de administración puede consistir en la multiplexación de un recurso en el tiempo o en el espacio, por ejemplo el procesador puede ser asignado alternativamente a diferentes procesos en intervalos de tiempo de una determinada duración, para así conseguir que los procesos se vayan ejecutando de un modo pseudo-concurrente. Otro ejemplo de multiplexación en tiempo puede ser el uso de una impresora compartida. Cuando se encolan varios trabajos en una impresora han de tomarse decisiones de qué trabajo se imprime primero y cuál después.

En la multiplexación en el espacio, en vez de asignarse turnos de uso, el recuso se divide entre los usuarios. Por ejemplo la memoria se reparte entre varios procesos obteniendo cada uno parte de la misma. De este modo pueden residir en memoria varios procesos y ejecutarse por ejemplo por turnos de uso del procesador. Otro ejemplo lo constituye el disco. En la mayoría de sistemas, el disco puede almacenar muchos ficheros de muchos usuarios a la vez. Controlar el espacio de disco usado y quine usa cada bloque del disco es una típica tareas de gestión del disco.

2.1.2 Breve descripción de los componentes de un ordenador

El sistema operativo debe conocer a fondo los componentes físicos del ordenador donde se va a ejecutar ("hardware"), para así poder presentar al programador una máquina "extendida".

De forma general y algo simple podemos describir un ordenador personal como una serie de dispositivos conectados todos ellos entre sí mediante un "bus" de comunicación. En realidad la estructura es bastante más compleja, ya que hay múltiples "buses", pero de momento como modelo es suficiente. A continuación se va a detallar brevemente los componentes más relevantes de cara al desarrollo de un sistema operativo [*TANE09*]:

El Procesador: Es el núcleo principal del ordenador. Se encarga de extraer instrucciones de la memoria y ejecutarlas. El ciclo básico de ejecución de un programa consiste en extraer una instrucción, decodificarla para determinar su tipo y operandos, ejecutarla, y seguidamente obtener la siguiente instrucción para repetir el ciclo.

El procesador tiene un conjunto de registros generales donde guardar resultados intermedios y variables. Dentro del conjunto de instrucciones del procesador, las hay que cargan el contenido de una palabra de memoria en un registro y viceversa, otras realizan operaciones aritméticas con dichos registros o posiciones de memoria, evalúan condiciones, bifurcan el flujo de ejecución, etc.

Además de esos registros, el procesador dispone de otros registros especiales. Uno de ellos es el contador de programa, que contiene la dirección de la siguiente instrucción del programa, otro registro es el puntero de pila, el cual señala la cima de la pila actual en memoria. En la cima se encuentra un conjunto dado de variables locales y parámetros asociados a un procedimiento. Por último otro registro especial muy importante es el "registro de estado del programa", el cual

guarda un conjunto bits que reflejan el estado y condición de ciertas operaciones efectuadas, como por ejemplo las de comparación. También pueden guardar el estado del permiso de interrupciones, etc.

Para realizar una llamada al sistema, el programa de usuario ejecuta una instrucción especial que irrumpe en el núcleo del sistema operativo invocando el servició asociado a la llamada. Esta instrucción especial conmuta el nivel de privilegio del modo usuario al modo supervisor, siempre y cuando el procesador dispusiera de esta característica, ya que por ejemplo el procesador Intel 8086 no la tiene.

La memoria: Es el segundo componente más importante en un ordenador. Como cualidades deseables de la memoria tenemos que; debería ser lo más rápida posible, existir en abundancia y de un coste lo más bajo posible. No hay tecnología que satisfaga todos estos requisitos simultáneamente y se ha de optar por una solución de compromiso, estableciendo jerarquías de memorias, atendiendo a su velocidad, tamaño y coste. La memoria más rápida la forman los registros del procesador, y suele ser de un tamaño inferior a 1KB. Seguidamente tenemos la memoria caché, controlada por el "hardware". Se usa como almacenamiento intermedio entre el procesador y la memoria de siguiente nivel, o memoria principal. Cuando se necesita un dato se busca primero en esta memoria y si se encuentra el dato no se requiere entonces el acceso al siguiente nivel de memoria, que es más lento. La caché puede tener más de un nivel, siendo sucesivamente los niveles superiores más rápidos que los inferiores. En el siguiente nivel tenemos la memoria principal o RAM (Random Access Memory), es más lenta y más barata y lo suele ser persistente. Los siguientes niveles lo componen dispositivos tales como los discos en primer lugar y cintas a continuación. Son memorias de gran tamaño, más económicas y de velocidad considerablemente menor. En estas memorias se suele soportar el almacenamiento del sistema de ficheros.

En el ordenador también suele haber un tipo de memorias conocidas como EEPROM y flash RAM, que son persistentes, aunque en contraste con las memorias ROM (Read Only Memory) pueden ser escritas más de una vez, y almacenan código y datos de rutinas básicas de manejo del hardware, como por ejemplo en el ordenador personal, la BIOS (Basic Input Output System). Otro tipo es la memoria CMOS, que es volátil, pero de muy bajo consumo. El ordenador suele mantener en ella la fecha y hora y la configuración del equipo, alimentada por una pequeña batería.

Con relación a los procesos y el uso de la memoria principal hay que precisar que normalmente es deseable guardar más de un programa en memoria, porque si un proceso se bloquea en espera de E/S, el procesador puede proseguir con la ejecución de otro proceso, aumentando con ello el rendimiento del procesador. No obstante en este caso, habría que resolver dos problemas:

1. Proteger la memoria del acceso indiscriminado de los procesos.
2. Manejar la reubicación del código y datos de los procesos.

Existen muchas posibles soluciones, pero todas ellas requieren que el procesador posea determinadas características hardware.

El primer problema es obvio, pero el segundo no lo es tanto. Cuando se compila y enlaza un programa, el compilador normalmente no sabe en qué dirección se cargará el programa, por ello, normalmente se asume como dirección de comienzo la 0. Supongamos que la primera instrucción carga una palabra de la dirección 1000, y que el programa se encuentra cargado en la dirección 2000. Sin reubicación, no se ejecutaría correctamente esta instrucción, porque cargaría la palabra de la dirección 1000, en vez de en la dirección 3000, que es de donde debería cargarla. Es necesario efectuar un proceso de reubicación del programa. La reubicación se puede hacer durante la carga, modificando todas las direcciones, lo que es costoso, o bien "al vuelo", en tiempo de ejecución, mediante el empleo de registros base en el procesador para formar las direcciones (también puede haber registros límite para protección). El proceso de mapeo resultante de convertir una dirección generada por el programa, llamada dirección "virtual", en una dirección de memoria llamada "física", lo realiza una parte del procesador llamada MMU (Memory Management Unit).

Los dispositivos de entrada/salida (E/S): Normalmente constan de dos partes, la controladora y el propio dispositivo. En la mayoría de los casos el control del dispositivo es bastante complejo, y el trabajo de la controladora es presentar una interfaz más sencilla al sistema operativo.

Cada tipo de controladora requiere un software diferente. El software que maneja la controladora se le conoce comúnmente en inglés como "*device driver*", término bastante conocido, aunque normalmente se le llama simplemente "*driver*", como por ejemplo: "*driver* de la tarjeta de video", o "*driver* de la tarjeta de red", etc.

El "*driver*" normalmente corre dentro del "*kernel*" en modo privilegiado, aunque teóricamente puede correr fuera del "*kernel*", son pocos los sistemas que optan por ello porque se requiere la habilidad de acceder al dispositivo de un modo controlado desde el espacio de usuario del "*driver*".

Normalmente hay tres formas de añadir un "*driver*" al sistema operativo. La primera consiste en compilar y enlazar el "*software"* del "*driver"* junto con el sistema operativo, tal que al arrancar el sistema, el "driver" forma parte del mismo. Muchos sistemas tipo UNIX funcionan así. La segunda forma consiste en crear una entrada en un fichero del sistema indicando que se requiere un "*driver*" y reiniciar. Durante el arranque el sistema operativo examina dicho fichero y cargue en ese momento el "*driver*". El sistema MS-DOS por ejemplo funciona así. La tercera y última es más sofisticada y menos corriente. Consiste en que el sistema operativo es capaz de instalar en cualquier momento un "*driver*" sin necesidad de re arrancar el sistema. Esta última forma cada vez es más corriente, y las últimas versiones de los sistemas operativos más populares ya la adoptan.

Una controladora suele disponer de un pequeño conjunto de registros que se usan para comunicarse con ella, por ejemplo para especificar direcciones, bien de memoria, de disco, etc. El "*driver*" recibe un comando del sistema operativo y lo traduce en los valores adecuados para escribir en estos registros. En algunos sistemas estos registros forman parte del espacio de direcciones de memoria, usando por tanto instrucciones de acceso a memoria también para este propósito. Otros en cambio, tienen un espacio de entrada/salida aparte, con instrucciones específicas como "IN" y "OUT" que requieren el modo de ejecución privilegiado. Ambos sistemas son ampliamente usados.

La entrada/salida se puede hacer de tres formas distintas. En el método más simple el proceso realiza una llamada al sistema, el cual a su vez invoca al "*driver*". El "*driver*" inicia la entrada/salida y permanece en un bucle de examen continuo del estado de finalización del dispositivo. Cuando el dispositivo cambia su estado indicando que ha acabado, el "driver" sale del bucle y retorna el resultado al sistema operativo el cual a su vez lo devuelve al proceso. Este método se llama "*espera activa*" y tiene la desventaja de que tiene ocupado al procesador en el bucle durante el tiempo que dure la operación de entrada/salida.

En el segundo método, el "*driver*" inicia el dispositivo pidiéndole que emita una interrupción cuando finalice. En este punto el "*driver*" retorna al sistema operativo, el cual bloquea al proceso

que hizo la petición y se pone a hacer cualquier otro trabajo pendiente. Cuando la controladora detecta el fin de la operación, emite una interrupción que hace que el sistema operativo recupere el resultado de la misma, y lo entregue al proceso que la solicitó, desbloqueando.

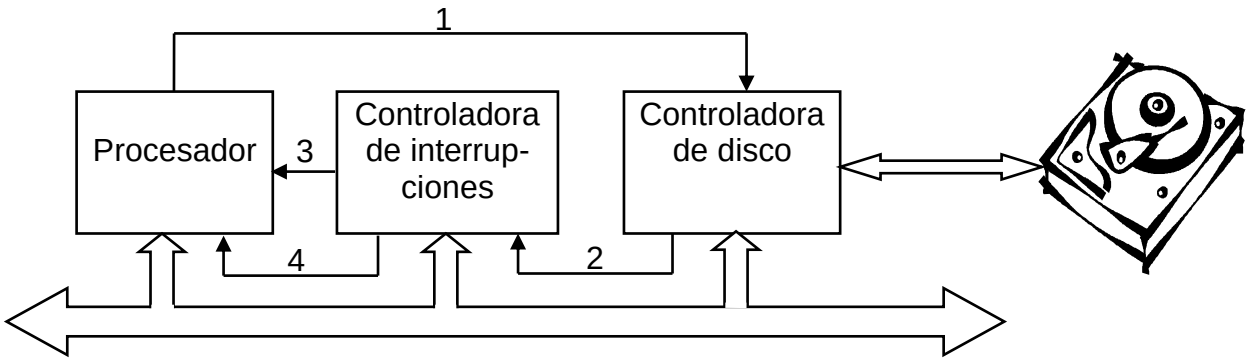


Figura 1: Etapas del inicio de una operación de E/S

Las interrupciones son muy importantes en los sistemas operativos. La fig.1 muestra un proceso de tres etapas para una operación de entrada/salida. En el paso 1, el "driver" escribe en los registros de la controladora la solicitud, y la controladora inicia el dispositivo. Cuando la controladora finaliza de leer o escribir la cantidad de información solicitada, señala al controlador de interrupciones mediante ciertas líneas del, es el paso 2. Si la controladora de interrupciones está preparada para aceptar la interrupción, entonces pone a 1 cierta línea del procesador informándoselo, es el paso 3. En el paso 4, el controlador de interrupciones pone el número de dispositivo en el bus para que el procesador pueda leerlo y saber qué dispositivo ha finalizado.

Cuando el procesador acepta la interrupción, el contador de programa y los "*flags*" de estado se meten en pila y se conmuta al modo supervisor. Se suele utilizar el número de dispositivo como un índice para acceder a una zona de memoria, el "vector de interrupción".

El tercer método de E/S usa un chip especial de DMA (Direct Memory Access), el cual puede supervisar la transferencia de bits desde el dispositivo a la memoria sin la intervención constante del procesador. El procesador programa el chip DMA diciéndole cuantos bits debe transferir a dónde. Al finalizar la operación el chip DMA produce una interrupción que informa al procesador de la conclusión de la misma.

Las interrupciones pueden suceder en cualquier momento, y en algunas ocasiones dicho momento puede ser inoportuno. Por dicha razón el procesador puede inhabilitar las interrupciones y habilitarlas de nuevo en otro momento. También es posible tener más de un dispositivo pendien-

te de atender su interrupción. El controlador de interrupciones en ese cado decide cuál atender basándose en una jerarquía de prioridades.

2.1.3 Conceptos básicos de los Sistemas Operativos

En prácticamente todos los sistemas operativos podemos encontrar ciertos conceptos o abstracciones tales como: Procesos, memoria, entrada/salida, ficheros, etc. [*TANE09*], [*CARR01*].

Procesos: Un proceso es básicamente un programa en ejecución. Los procesos tienen asociados un espacio de direcciones, formado por un conjunto de posiciones de memoria donde el proceso puede leer o escribir. En dicho espacio se encuentra el programa ejecutable, los datos y la pila. También hay asociados al proceso un conjunto de registros, tales como el contador de programa, el de pila, etc., así como el resto de información necesaria para ejecutar el programa.

Cuando se suspende temporalmente la ejecución de un proceso, para posteriormente reanudarse en el mismo punto donde se suspendió, el S.O. tiene que guardar toda la información necesaria en algún sitio. En muchos sistemas operativos dicha información se guarda, salvo el contenido de su espacio de direcciones, en la "tabla de procesos". Los elementos de dicha tabla (que podría ser una lista u otro tipo de estructura) se conocen generalmente como "descriptores de proceso".

Las llamadas al sistema relativas a procesos primordiales son las que se encargan de la creación y terminación de procesos. Cuando un proceso crea otro proceso, generalmente se le conoce como proceso hijo. Si a su vez éste crea otro proceso y así sucesivamente se obtiene una jerarquía de procesos en árbol. A menudo varios procesos relacionados pueden trabajar cooperativamente en algún trabajo y necesitan comunicarse para sincronizarse. El S.O. dispone para ello de un servicio de comunicación de procesos. Otras llamadas al sistema solicitan o liberan memoria, esperan por la terminación de un proceso, cambian la imagen de memoria del proceso, etc.

Una herramienta muy útil la constituye "*las señales*". Por ejemplo el S.O. puede enviar una señal de alarma a un proceso cuando ha transcurrido un cierto lapso de tiempo. La señal hace que el proceso suspenda temporalmente lo que está haciendo y se ejecute un procedimiento específico de tratamiento de señal. Cuando este tratamiento finaliza, el proceso reanuda lo que estaba haciendo, justo a continuación del momento en que fue suspendido. Las señales son el análogo a las "interrupciones *hardware*". Éstas pueden ser generadas por múltiples causas. Por ejemplo, si

se produce una excepción por una dirección inválida, el S.O. podría enviar una señal al proceso que la provocó.

A los usuarios de un sistema normalmente se les suele asignar un identificador (UID), igualmente sucede con los procesos (PID). El sistema sabe que usuario ejecutó cada proceso gracias a estos identificadores. Con ello se puede evitar que un proceso lleve a cabo funciones que no le son permitidas.

Cuando dos o más procesos interactúan se puede dar el caso que se produzcan interbloqueos. Una situación en la cual ningún proceso puede avanzar por estar bloqueado por el otro. Esta situación es similar al caso de la vida real en la que varios vehículos se aproximan a un cruce y se detienen justo en la intersección para verificar si pueden cruzar, bloqueando el paso a otros vehículos. No cruzan porque otro vehículo les bloquea el paso y esto mismo sucede con todos los demás, quedando todos ellos bloqueados. Para evitar estas situaciones existen herramientas ofrecidas por el S.O., tales como los semáforos, buzones, etc.

Gestión de la memoria: Un S.O. que se precie puede alojar en memoria más de un proceso. Esto significa que debe haber algún mecanismo de protección para evitar que los procesos interfieran entre ellos. Este mecanismo lo aporta el "*hardware*" pero lo controla el S.O. Por otro lado, otro concepto no menos importante relacionado con la gestión de memoria consiste en el manejo del espacio de direcciones de los procesos. Normalmente un proceso tiene un conjunto de direcciones máximo. En el caso más simple, todo el espacio de direcciones de un proceso cabe en la memoria principal disponible. Sin embargo, ¿Qué pasaría si un proceso tuviera un espacio de direcciones mayor que la memoria disponible y quisiera escribir en todo el espacio? En los sistemas antiguos eso no era posible, pero en la actualidad, una técnica llamada "memoria virtual", nos lo permite. Con esta técnica el S.O. gestiona el espacio, manteniendo parte del mismo en memoria principal y parte en disco, trayendo y llevando bloques de memoria desde un soporte físico al otro según sea necesario.

Entrada/Salida: En todos los ordenadores existen dispositivos físicos para poder realizar operaciones de entrada o salida de datos. Después de todo, ¿De qué serviría tener la posibilidad de hacer un buen trabajo si el usuario no pudiera indicar qué hacer, o recibir el resultado? Hay muchos tipos de dispositivos: teclado, pantalla, impresora, ratón, etc., y es responsabilidad del S.O. su gestión. Una parte del "*software"* encargado del mismo puede ser independiente del dispositi-

vo, y por tanto ser el mismo para todos ellos y otra parte puede ser específica para cada dispositivo concreto.

Ficheros: Otro concepto clave es el "sistema de ficheros". Tal y como ya se ha dicho, una parte importante del S.O. es ocultar las particularidades de los discos y otros dispositivos, presentado al programador un modelo abstracto limpio y agradable de ficheros independientes del dispositivo. Se necesitan servicios para crear, borrar, leer y escribir ficheros.

La mayoría de los sistemas operativos utilizan un concepto llamado "directorio", como una forma alojar ficheros y mantenerlos agrupados. Los directorios pueden formar una jerarquía en árbol, y aunque esto sucede también con los procesos, la similitud acaba aquí. La profundidad o niveles en los procesos suele ser corta, al contrario que con los ficheros donde son comunes niveles de profundidad de 4, 5 o más. La jerarquía en los procesos es de vida corta, al contrario que en ficheros. También es bastante diferente la gestión de la propiedad y la protección.

Todo fichero dentro de la jerarquía puede ser especificado por su "ruta" (pathname). Ésta puede ser especificada desde la raíz, por ejemplo: "\usrs\libros\curso\milibro.doc", o bien especificarse desde el "directorio de trabajo". Por ejemplo: "curso\milibro.doc", donde el directorio de trabajo sería "\usrs\libros". Todo proceso tiene asociado por tanto un directorio de trabajo y es posible mediante una llamada al sistema cambiarlo.

Un concepto muy importante en sistemas operativos tipo UNIX es el de "fichero especial". Este concepto sirve para poder trabajar con dispositivos como si fueran ficheros, de este modo, éstos pueden ser leídos y escritos usando los mismos servicios que el resto de ficheros. Existen dos tipos de ficheros: orientados a bloque, y orientados a carácter. Los orientados a bloque se usan para modelizar dispositivos consistentes en un conjunto de bloques con forma de acceso directo, tales como los discos. Los orientados a caracteres se usan para modelizar impresoras, módems, y otros dispositivos que aceptan una corriente de caracteres como entrada o salida.

2.2 Sistemas Operativos para la enseñanza.

En este apartado se mostrarán las características más importantes de algunos de los sistemas operativos existentes en la actualidad, que por su relativa sencillez se han venido utilizando como modelo y herramienta para prácticas en la enseñanza de sistemas operativos.

2.2.1 Minix

Minix es un sistema operativo de código abierto, diseñado para ser muy fiable, flexible y seguro. Este sistema operativo tipo UNIX fue desarrollado por Andrew S. Tanenbaum en 1987, el cual también distribuyó su código fuente. Fue creado principalmente para la enseñanza del diseño de sistemas operativos, ya que el sistema Unix estaba bajo restricciones de licencia de AT&T y era demasiado complejo. Gracias a su pequeño tamaño, su diseño basado en el concepto de "micro núcleo" y la amplia documentación disponible, resulta bastante adecuado para todo aquel que desee un sistema Unix para su PC, así como también para aprender sobre su funcionamiento interno.

La versión inicial, ***Minix 1,*** era compatible a nivel de llamadas con la 7ª edición de Unix. Fue desarrollada para correr sobre un PC con procesador 8088/86, aunque versiones posteriores fueron ampliando sus posibilidades al correr sobre procesadores más potentes como el 80286, 80386, etc. Junto con el sistema operativo, Tanenbaum & Woodhill publicaron también el libro "Sistemas Operativos: Diseño e Implementación" [*TAWO98*], en el que se recogía una parte del código en 'C' del núcleo, el gestor de memoria y el gestor de ficheros (pág. web no oficial: http://minix1.woodhull.com/index1.html).

En 1991 se lanzó la versión ***Minix 1.5***, con soporte para *micro-channel* y arquitecturas Motorola 68000 y SPARC, aunque también hubo versiones no oficiales para el i386 (en modo protegido), NS32532 de *National Semiconductor*, ARM e INMOS transputers. *Meiko Scientific* usó una versión previa de Minix en su sistema operativo "*MeikOS*", para sus ordenadores paralelos de cómputo superficial basados en transputers. También se creó un simulador llamado SMX que corría como un proceso de usuarios sobre los sistemas operativos *SunOS* y *Solaris*.

Posteriormente, en 1997 se lanzó **Minix 2** que era solo compatible con arquitecturas x86 y SPARC, dada la caída de las arquitecturas *68000*. Con esta versión se lanzó la segunda edición

del libro de Tanenbaum. En esta versión se añadió compatibilidad con POSIX1, soporte 32 bits (I386 y superiores) y protocolo TCP/IP que reemplazo al de *Amoeba* de Minix 1.5. Más tarde dos investigadores de la universidad *Vrije* de Ámsterdam realizaron la versión *Minix-vmd* para procesadores compatibles IA-32, a la que dotaron de memoria virtual y soporte para X-Windows. En general, Minix 2 puede ser una buena opción para alguien que desee entender casi todos los elementos del sistema operativo con sólo algunos meses de estudio. (pág. web no oficial: http://minix1.woodhull.com).

Finalmente, en Octubre de 2005, en la conferencia del "*ACM Symposium Operating Systems Principles*", Andrew Tanenbaum anunció **Minix 3**, que aunque sigue manteniéndose como ejemplo en su libro, se rediseñó para ser utilizado como un sistema serio para PC's con recursos limitados y aplicaciones que requieran una gran fiabilidad. Esta versión está disponible en forma de "LiveCD", lo que permite ser utilizado sin necesidad de instalación previa. También puede ser usado en sistemas de virtualización como BOCHS, Qemu, VirtualBox. Etc. La última versión en la actualidad es la 3.1.8, la cual se encuentra disponible en la página oficial www.minix3.org.

Por ser muy extensa la documentación en la red sobre este sistema, sólo se muestran sólo algunos de los enlaces donde se puede obtener más información.

- http://minix1.woodhull.com. (soporte para Minix 2). Desde esta página se pueden obtener otros muchos enlaces tales como:

  http://minix1.woodhull.com/hints.html. Información variada sobre Minix.
  http://minix1.woodhull.com/mxdownld.html Descarga de Minix 2.
  http://minix1.woodhull.com/contrib.html Paquetes de software (utils..).
  http://minix1.woodhull.com/mxinet.html. Minix como servidor ftp, http, etc.
  http://minix1.woodhull.com/docs.html. Documentación sobre Minix 2.
  http://minix1.woodhull.com/teaching. Recursos para la enseñanza de Minix
  http://minix1.woodhull.com/others.html. Otras versiones de Minix (Minix-vmd….)
  http://minix1.woodhull.com/links.html. Otras fuentes de información.

- http://www.dmoz.org/Computers/Software/Operating_Systems/Unix/MINIX.
- http://sopa.dis.ulpgc.es/ii-dso/lecminix/lecminix.htm.

Las características más relevantes de Minix 2 son:

- Similar a Unix V7 (procesador comandos y llamadas al sistema).
- Sacrifica "eficiencia" por "comprensión y modularidad".
- Soporta Multiproceso y Multiusuario.
- Diseñado para PC compatible sin uso de la BIOS. También con versiones para Atari, Amiga y Macintosh y emuladores sobre Unix y SUN
- Dispone de "*drivers*" de video para CGA, monocromo, Hércules y EGA.
- Requiere una partición de disco de al menos 30MB.
- Soporta hasta 16MB en un i286 y 4GB en i386 o superior.
- 2 Puertos RS232 (terminal y/o modem) y Centronics (impresora).
- Controladores para disco duro tipo XT, AT y BIOS.
- Manejadores de red TCP/IP.
- Llamadas al sistema compatibles POSIX.
- Escrito en 'C' (ANSI y IEEE POSIX: Amsterdam Compiler Kit).

El sistema está diseñado por niveles (que podríamos numerar arbitrariamente del 0 al 3 de abajo arriba), tal y como se muestra en la figura siguiente:

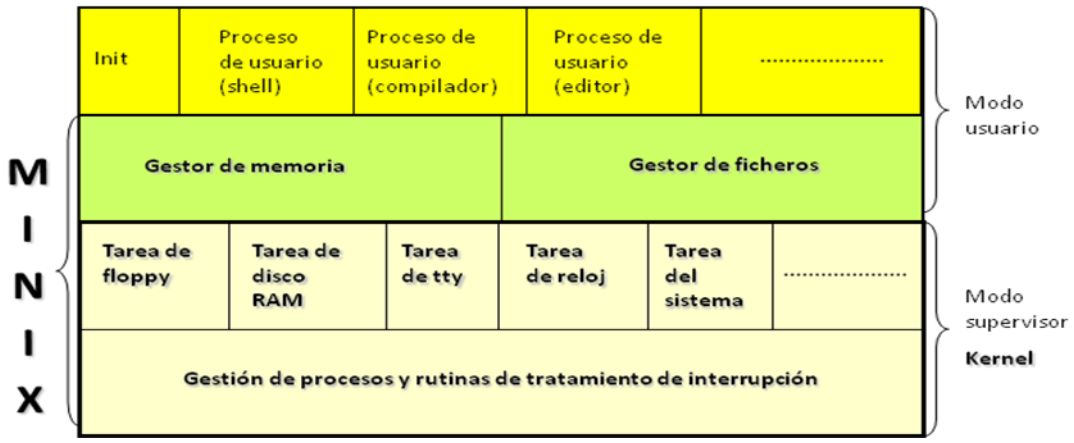


Figura 2: Estructura de niveles de Minix

Minix Está formado por un conjunto de procesos (denominadas tareas) que se comunican entre sí y con los procesos de usuario a través de unas primitivas de comunicación de paso de mensajes tipo "rendez vous" (comunicación síncrona). Existen restricciones de comunicación entre niveles, por ejemplo el nivel de usuario (3) no puede comunicarse con los niveles inferiores 0 ó 1. Seguidamente se detallan las funciones de cada una de los niveles:

Nivel 0: Atender Interrupciones, planificar tareas y procesos y manejo y validación del paso de mensajes.

Nivel 1: En el corren las tareas de entrada salida (manejadores de dispositivo) y la tarea del sistema (tiene funciones especiales no asociadas a ningún dispositivo). Tanto las tareas de nivel 0, como las del nivel 1, forman un único ejecutable (kernel), el cual corre en modo supervisor.

Nivel 2: En el corren los procesos servidores (gestores) de Minix. El Gestor de memoria, el Gestor de ficheros y el Gestor de red. Tiene un nivel de privilegio menor que las tareas pero mayor que los procesos de usuario. El Gestor de Memoria atiende todas las llamadas relacionadas con procesos y memoria: fork, exec, brk, etc. El Gestor de ficheros, atiende las de ficheros: open, read, etc.

Nivel 3: En este nivel corren los procesos "init" e "interprete de comandos" (Shell), así como el resto de aplicaciones de usuario con el nivel de privilegio más bajo.

2.2.2 Linux

Linux es una implementación de UNIX de libre distribución. Originalmente se desarrolló para el procesador de Intel 386, pero en la actualidad se encuentra desarrollado para multitud de procesadores, como: i486, Pentium (toda la gama y clones AMD y Ciryx), SPARC, DEC Alpha, PowerPC, PowerMac y Max, Motorola, etc. [*SARW03*]

Como sistema operativo es muy eficiente y de muy buen diseño. Es multitarea, multiusuario, multiplataforma y multiprocesador. Tiene protección de memoria, capacidad de compartición de memoria, memoria virtual mediante paginación, librerías estáticas y dinámicas, consolas virtuales, sistema avanzado de archivos pudiendo usar los de otros sistemas, y múltiples protocolos de red incluyendo TCP/IP.

Este sistema se usa ampliamente en el mundo de los negocios (hay varias empresas que comercializan soluciones basadas en GNU/Linux: IBM, Novell (SuSE), Red Hat (RHEL), Mandriva (Mandriva Linux), Rxart, Canonical Ltd. (Ubuntu), así como miles de PYMES que ofrecen productos o servicios basados en esta tecnología), aunque su código es de libre distribución. Esto último, lo convierte en un buen candidato para la enseñanza en muchas áreas de los sistemas

operativos, no obstante, y debido a su complejidad, podría no serlo tanto en algunas otras, sobre todo en aquellas cuyos conceptos esenciales requieren ser contados con la máxima sencillez.

Linux aparece al principio de los 90. Un estudiante de informática llamado Linux Torvalds comenzó, como una afición, a programar las primeras líneas de código de lo que llegaría a ser este sistema. Se inspiró básicamente en el proyecto MINIX de Andrew S. Tanenbaum y al principio el foro de discusión principal era para usuarios que deseaban algo más de lo ofrecido por MINIX.

La primera versión "oficial" fue la 0.02. En ella se podía ejecutar Bash (interprete de comandos Bourne) y gcc (compilador GNU de 'C'), pero no mucho más. Después de la versión 0.03 se saltó a la 0.10 y empezaron a trabajar más y más personas en el proyecto. En marzo del 92 la versión era la 0.95 y desde entonces no se ha parado de desarrollar este proyecto

A continuación se hace un resumen algo más extenso de sus características:

- Multitarea: La palabra multitarea describe la habilidad de ejecutar varios programas al mismo tiempo. LINUX utiliza la llamada multitarea expulsora, la cual asegura que todos los programas que se están utilizando en un momento dado serán ejecutados, siendo el sistema operativo el encargado de ceder tiempo de microprocesador a cada programa.
- Multiusuario: Muchos usuarios usando la misma máquina al mismo tiempo.
- Multiplataforma: Las plataformas en las que en un principio se puede utilizar Linux son i386, i486. Pentium (toda la familia y clones), Amiga y Atari, también existen versiones para su utilización en otras plataformas, como Alpha, ARM, MIPS, PowerPC y SPARC.
- Multiprocesador: Soporte para sistemas con más de un procesador.
- Funciona en modo protegido i386.
- Protección de la memoria entre procesos.
- Carga de ejecutables por demanda: Linux sólo lee del disco aquellas partes de un programa que están siendo usadas actualmente.
- Política de copia en escritura para la compartición de páginas entre ejecutables: esto significa que varios procesos pueden usar la misma zona de memoria para ejecutarse. Cuando alguno intenta escribir en esa memoria, la página (4Kb de memoria) se copia a otro lugar. Esta política de copia en escritura tiene dos beneficios: aumenta la velocidad, y reduce el uso de memoria.

- Memoria virtual usando paginación (sin intercambio de procesos completos) a disco o a una partición y/o archivo; con la posibilidad de añadir más áreas de intercambio sobre la marcha. Se pueden usar hasta un total de 16 zonas de intercambio de 128Mb de tamaño máximo en un momento dado con un límite teórico de 2Gb para intercambio. Este límite se puede aumentar fácilmente con el cambio de unas cuantas líneas en el código fuente.
- La memoria se gestiona como un recurso unificado para los programas de usuario y para el caché de disco, de tal forma que toda la memoria libre puede ser usada para caché y ésta puede a su vez ser reducida cuando se ejecuten grandes programas.
- Librerías compartidas de carga dinámica (DLL's) y librerías estáticas.
- Se realizan volcados de estado (core dumps) para posibilitar los análisis post-mortem, permitiendo el uso de depuradores sobre los programas no sólo en ejecución sino también tras abortar éstos por cualquier motivo.
- Compatible con POSIX, System V y BSD a nivel fuente.
- Emulación de iBCS2, casi completamente compatible con SCO, SVR3 y SVR4 a nivel binario.
- Todo el código fuente está disponible, incluyendo el núcleo completo y los drivers, las herramientas de desarrollo y programas de usuario; además todo ello se puede distribuir libremente, aunque existen algunos programas comerciales de los que no se ofrece libremente el código fuente.
- Control de tareas POSIX.
- Pseudo-terminales (pty's).
- Emulación de 387 en el núcleo, de tal forma que los programas no tengan que hacer su propia emulación matemática. Cualquier máquina que ejecute Linux parecerá dotada de coprocesador matemático. Por supuesto, si el ordenador ya tiene una FPU (unidad de coma flotante), esta será usada en lugar de la emulación, pudiendo incluso compilar tu propio kernel sin la emulación matemática y conseguir un pequeño ahorro de memoria.
- Soporte para muchos teclados nacionales o adaptados y es bastante fácil añadir nuevos dinámicamente.
- Consolas virtuales múltiples, con varias sesiones "*login*" a través de la consola, intercambiables mediante las combinaciones de teclas adecuadas (independiente del hardware de video). Se pueden crear dinámicamente hasta un máximo de 64.
- Soporte para varios sistemas de archivo comunes, incluyendo Minix-1, Xenix y el sistema de archivo típico de System V. Dispone de un avanzado sistema de archivos propio con una capacidad de hasta 4 TB y nombres de archivos de hasta 255 caracteres de longitud.

- Acceso transparente a particiones MS-DOS (o a particiones OS/2 FAT) mediante un sistema de archivos especial. No es necesario ningún comando especial para usar la partición MS-DOS, esta parece un sistema de archivos normal de Unix (excepto por algunas restricciones en los nombres de archivo, permisos, y esas cosas). Las particiones comprimidas de MS-DOS 6 no son accesibles en este momento, y no se espera que lo sean en el futuro. El soporte para VFAT (WNT, Windows 95) ha sido añadido al núcleo de desarrollo y estará en la próxima versión estable.
- Un sistema de archivos especial llamado UMSDOS que permite que Linux sea instalado en un sistema de archivos DOS.
- Soporte en sólo lectura de HPFS-2 del OS/2 2.1
- Sistema de archivos de CD-ROM que lee todos los formatos estándar de CD-ROM.
- TCP/IP, incluyendo ftp, telnet, NFS, etc.
- Appletalk.
- Software cliente y servidor Netware.
- Lan Manager / Windows Native (SMB), *software* cliente y servidor.
- Diversos protocolos de red incluidos en el kernel: TCP, IPv4, IPv6, AX.25, X.25, IPX, DDP, Netrom, etc.

En la actualidad existen muchos sitios en Internet donde se puede acceder a una gran diversidad de distribuciones Linux, muchas de ellas gratuitas y otras de propósito comercial de coste variable en función de múltiples factores. Si lo que se desea es obtener algo más de información sobre este sistema se pueden consultar en otras muchas páginas web, las siguientes:

- http://www.linux-es.org/sobre_linux
- http://es.wikipedia.org/wiki/Historia_de_Linux
- http://www.monografias.com/trabajos14/linux/linux.shtml - historia

2.2.3 Otros sistemas operativos

MikeOS: Este sistema operativo está diseñado para PC's de la familia i386 y está escrito completamente en ensamblador. Es una buena herramienta para demostrar lo simple que es el funcionamiento de un sistema operativo, con un código bien comentado y una amplia documentación. Sus características básicas son:

- Interfaz de menús y dialogo en modo texto.
- Arranque desde disquete, CD-ROM y USB.
- Sobre unas 60 llamadas al sistema para aplicaciones.
- Gestor de ficheros, Editor de textos, visor de imágenes, juegos…
- Incluye intérprete BASIC con 27 instrucciones.
- Sonido del altavoz del PC y conexión terminal serie .
- Código abierto con licencia tipo BSD.

El requisito mínimo para correr el sistema es un PC i386 con 1 MB de memoria y teclado, aunque en principio debería ser posible ejecutarlo en una PC más antiguo.

Un buen recurso para ejecutar el sistema consiste en emplear un entorno de máquina virtual como Qemu, VirtualBox, Virtual PC, Bochs…

Algunos de los programas que se incluyen con el sistema para realizar diversas tareas y demostrar sus características son:

- Edit.bin (editor de texto de pantalla completa).
- Example.bas (Demostración de las características del BASIC).
- Fileman.bin (Borrado, renombrado y copiado de ficheros).
- Hangman.bin (Adivina nombres de ciudades de todo el mundo).
- Keyboard.bin (Teclado musical).
- Monitor.bin (Monitor de código máquina simple).
- Serial.bin (Terminal serie del tipo “Minicom”).
- Viewer.bin (visor de ficheros de texto e imágenes “pcx”).

La página oficial es: http://mikeos.berlios.de

Otros proyectos basados o inspirados en MikeOs:

- MikeOs en modo protegido (32 bits).
- Atom Core Os (MikeOS 3.1 con un rudimentario soporte para DOS).
- TomOs (16 bits O.S. basado en MikeOs 2.0).

- BareMetal OS (64 bits O.S. basado en MikeOs).

MenuetOS: Es un sistema operativo en desarrollo para el "PC" escrito completamente en ensamblador para 32/64 bits. Menuet64 se distribuye bajo licencia y Menuet32 bajo "GPL". *Menuet* soporta la programación en ensamblador 32/64 bits (Intel x86) de aplicaciones más pequeñas, más rápidas y de menor consumo de recursos.

Menuet no está basado en otros sistemas operativos ni tiene sus raíces en estándares POSIX. El objetivo de diseño, desde su primera versión en el año 2000, fue eliminar las capas extra entre las diferentes partes de un S.O., las cuales normalmente complican la programación y son fuente de errores (*bugs*).

La estructura de una aplicación Menuet no está reservada específicamente para aplicaciones ***asm,*** ya que la cabecera puede ser producida por prácticamente cualquier otro lenguaje. Sin embargo, en conjunto, el diseño de programación de la aplicación está enfocado para la programación en ensamblador de 32/64 bits. Programar para Menuet es fácil y rápido de aprender, y su interfaz gráfico de usuario (*GUI*) es fácil de manejar desde lenguaje ensamblador. Menuet64 puede ejecutar aplicaciones para Menuet32.

Entre sus características destacan:

- Escrito totalmente en ensamblador
- Multitarea expulsora con planificador de 1000 hz, multihilo, multiprocesador, protección de anillo 3.
- Resolución del interfaz gráfico de 1280x1024 y 16 millones de colores
- Aplicaciones de ventana transparentes y configurables (*drag* & *drop*).
- Soporte multiprocesador (SMP) hasta 8 procesadores.
- Editor y ensamblador para aplicaciones (IDE).
- USB 2.0: Almacenamiento de alta velocidad, webcam, impresora y Radio/TV.
- USB 1.1: Soporte para teclado y ratón.
- Pila TCP/IP con *loopback* y *drivers* para Ethernet
- Clientes e-mail, ftp, http y servidores ftp, mp3 y http.
- Captura de datos en tiempo real (*hard real-time data fetch*).
- Cabe en un disquete, arranca desde unidades CD y USB.

Página oficial: http://www.menuetos.net

GeekOS: Es un pequeño núcleo de sistema operativo para PC's basados en el Intel x86. Su principal propósito es servir de ejemplo sencillo pero real de un núcleo de sistema operativo corriendo sobre un "hardware" real (En realidad, la mayor parte de su desarrollo se ha realizado en el emulador "Bochs" de hardware de PC). (Nota: "Geek" se podría traducir por: empollón, cerebrito, Pitagorín, friki, etc. En general, fascinado por la informática y tecnología)

El objetivo de GeekOS es ser una herramienta de aprendizaje sobre el diseño y construcción del núcleo de un sistema operativo. Con ese fin, la versión 0.2.0 lleva incluidos un conjunto de proyectos adecuados para ser usados en cursos sobre sistemas operativos, o también para autoaprendizaje. En la actualidad, este sistema se ha venido usando en diferentes centros de enseñanza y universidades (i.e: Universidad Maryland).

Las motivaciones para crear este sistema según explica su autor son:

1. El deseo de crear un ejemplo o patrón de programa que arranque (haga *boot*) desde un equipo desnudo "x86". Hacer que un programa haga "boot", incluso uno tan sencillo como el típico que muestra la frase: "Hola mundo", es un primer paso para construir el "kernel" de un sistema operativo. Poner en un ejemplo, simple y bien documentado, cómo hace boot el "kernel", podría ser una buena idea para animar a otros a escribir su propio kernel. Debido a la simplicidad de GeekOS, éste podría ser útil para desarrolladores de sistemas empotrados.
2. El deseo de actualizar los proyectos del curso (*under graduate*) de sistemas operativos de la universidad de Maryland. Los proyectos fueron desarrollados originalmente para MS-DOS, usando un compilador y ensamblador para el modo real de 16 bits. Mientras que usar MS-DOS como anfitrión para los proyectos permitía el acceso directo a los recursos, algo deseable, también dejaba al S.O. vulnerable a los "cuelgues" de los proyectos. Además los estudiantes acostumbrados a la programación en sistemas como Unix o Windows no encontraban muy agradable la experiencia del modo de direccionamiento artificioso del modo real. GeekOs es un repuesto lógico para los proyectos originales de 16 bits, porque sigue manteniendo el espíritu de la programación próxima al "hardware". Porque los PC's basados en la familia x86 son baratos y ampliamente disponibles, y porque existen herramientas libres de mucha calidad disponibles para ellos. Además existen excelentes emula-

dores para el hardware del PC, como Bochs, Qemu, etc. que se encuentran implementados para varias plataformas.

Hacer que GeekOS se mantenga simple es de vital importancia, y para ello se han limitado sus características a aquellas consideradas fundamentales. A continuación se enumeran algunas de ellas:

- Escrito básicamente en "C".
- Manejo de interrupciones.
- Gestión de memoria dinámica (*heap memory allocator*).
- Hilos (*threads*) de "*kernel*" por rodajas de tiempo con planificación de prioridad estática.
- *Mutexes* y variables de condición para la sincronización de hilos del *kernel*.
- Modo usuario con protección de memoria basada en segmentación y un interfaz simple de llamadas al sistema.
- Controladores de dispositivo para teclado y pantalla VGA en modo texto.

En esta lista no se han incluido memoria virtual paginada, controladores de dispositivo de almacenamiento ni sistema de ficheros. Se ha usado el mecanismo de segmentación del "x86" para implementar la protección de memoria para las tareas de modo usuario. Hay que señalar que los segmentos de modo usuario usan un modelo de direccionamiento de 32 bits, evitándose con ello los artificios requeridos en el direccionamiento de segmentos del modo real. GeekOS incluye un mecanismo para compilar un programa de usuario como un objeto de datos enlazados directamente dentro del "*kernel*", para sortear así la ausencia de almacenamiento de disco y sistema de ficheros. Esta técnica puede usarse también para implementar un sistema de ficheros sobre la RAM.

Página oficial: http://geekos.sourceforge.net

GNUFiwix:

Fiwix es un núcleo de sistema operativo basado en la arquitectura UNIX y enfocado totalmente para ser compatible con el núcleo de LINUX. Este sistema está siendo desarrollado fundamentalmente para uso educacional, y entre sus objetivos está el intentar mantenerse tan simple como sea posible, en aras de facilitar al máximo al estudiante la comprensión del mismo. *Linux* es de-

masiado grande y complejo para que un estudiante medio pueda entender fácilmente su estructura interna, y es por ello que este sistema, debido a su mínima estructura, puede ser más adecuado para la educación. Por razones obvias, este sistema está enfocado principalmente a estudiantes y entusiastas en sistemas operativos que deseen ampliar sus conocimientos en la materia.

La plataforma en la que corre es la de la familia de Intel "x86" de 32 bits, y es compatible con un buen número de aplicaciones GNU ya existentes. No hay que creer que *Fiwix* es una variante más de Unix con sus propias librerías, utilidades, aplicaciones, etc., sino más bien que es un nuevo núcleo de estilo Linux pero bajo la filosofía GNU/Linux y que se aprovecha de las aplicaciones GNU ya existentes. *Fiwix* es altamente compatible con la base del núcleo Linux (con sus limitaciones), permitiendo que cualquier programa ELF-i386 compilado en un sistema GNU/Linux puede ser ejecutado nativamente sin ningún tipo de emulación. El diseño de su núcleo es monolítico, y el lenguaje de desarrollo principal es 'C', con algunas partes críticas en ensamblador.

Algunas de sus características a destacar son:

- Especificación compatible "GRUB Multiboot".
- Modo protegido de 32bit. Núcleo no expulsor.
- Multitarea real (tareas de núcleo a nivel 0).
- Entorno de tareas protegido (direc. memoria independientes por proceso).
- Manejo de interrupciones y excepciones.
- Señales POSIX.
- Comunicación entre procesos con "*pipes*".
- Manejo de memoria virtual hasta 4GB (todavía sin intercambio).
- Paginación por demanda con "Copy-On-Write".
- Compatibilidad de llamadas al sistema Linux.
- Soporte formato ejecutable *Linux ELF-386* (enlazado estática y dinámicamente).
- Algoritmo de planificación *Round Robin* (todavía sin prioridades).
- Capa de abstracción *VFS*.
- Soporte sistema ficheros Ext2" (solo lectura) con bloques de 1, 2 y 4 KB.
- Soporte sistema ficheros "Linux PROCfs".
- Soporte pseudo sistema ficheros "PIPEfs".
- Soporte sistema ficheros "ISO9660" con extensiones "Rock Ridge".

- Soporte para dispositivos "RAMdisk".
- Soporte para aplicaciones basadas en "SVGAlib".
- Driver teclado con soporte predefinido para Inglés, Español/Catalán.
- Soporte para "*driver*" de impresora de puerto Paralelo.
- "*Driver*" para disquetera con manejo de "DMA".
- "*Driver*" para disco duro "IDE/ATA" (sólo lectura).
- "*Driver*" para CD-ROM IDE/ATA ATAPI.

Entre sus requisitos tenemos:

- Arquitectura estándar "PC".
- Procesador "Intel IA-32" y compatibles (80386 y superiores).
- 3MB de memoria RAM.
- Disquetera de 3.5" con 1.44MB de capacidad.
- Teclado USA o ES/CA.
- Adaptador gráfico VGA.

Página oficial: http://www.fiwix.org/

Minirighi:

El código de este sistema operativo está escrito en 'C' y ensamblador, partiendo totalmente desde cero, es decir sin reutilizar código de ningún otro sistema. Está basado en la arquitectura de 32 bits de Intel, es multihilo y compatible con POSIX. Su principal característica es su núcleo ligero, el cual es fácil de leer, al contrario que otros muchos núcleos de código abierto (*open source*), analizando simplemente su código.

Gracias a su sencillez, puede utilizarse con fines didácticos, bien para profundizar en el desarrollo de técnicas de sistemas operativos o para el estudio de arquitecturas particulares, dirigiéndose tanto a principiantes y/o estudiantes como a los usuarios más expertos. Desde este punto de vista, *Minirighi*, puede ser una herramienta muy útil para aquellos que deseen testear algoritmos de alto nivel, ya que las rutinas básicas o fundamentales ya están escritas; pero también puede ser útil para desarrolladores de controladores (*drivers)* que quieran comprobar y mejorar las rutinas ya existentes en un sistema relativamente simple. Por otro lado, es fácil modificar el código para

que el sistema se adapte a cualquier propósito; por ejemplo podría ser adaptado a sistemas de tiempo real con muy poco código adicional, ya que las operaciones para manejar las restricciones de tiempo ya existen. La ligereza y modularidad del núcleo también lo hacen útil para el desarrollo de sistemas dedicados a dispositivos y/o tecnologías particulares que no requieran interfaces amigables pesadas y/o gráficos, así como también lo hace muy fácilmente adaptable a las nuevas tecnologías.

Se puede encontrar información adicional en: http://minirighi.sourceforge.net/

3. GESTIÓN DEL PROYECTO

3.1 Introducción.

Podríamos decir que un proceso es el conjunto ordenado de pasos que hay que seguir para obtener un producto o alcanzar la solución de un problema, y en el caso concreto de ser un producto *software*, que éste resuelva un problema. El proceso de creación puede llegar a ser bastante complejo, por ejemplo, la creación de un sistema operativo suele requerir un proyecto y, la gestión de muchos recursos y en general todo un equipo de trabajo. Por otro lado, si se tratara de un programa sencillo, éste podría ser realizado por un solo programador fácilmente. Es por ello que el proyecto suele clasificarse según su tamaño en tres categorías: pequeño, mediano y de gran porte. Existen varias metodologías para estimarlo, siendo una de ellas el sistema **COCOMO**, el cual aporta métodos y un programa que calcula y realiza una estimación de los costes de producción [*BARR94*], pág. web en español: http://www.sc.ehu.es/jiwdocoj/mmis/cocomo.htm.

En los proyectos de gran porte es necesario realizar trabajos bastante complejos, tanto en el aspecto técnico como en el administrativo, por lo que se ha llegado a desarrollar una ingeniería para su estudio conocida como "Ingeniería de *Software*" [*SOMM05*]; mientras que en los de mediano porte basta un pequeño grupo de trabajo e incluso un único analista-programador eficiente

puede realizar el trabajo. En estos proyectos (incluso a veces en los de pequeño porte) es siempre necesario seguir ciertas etapas en la creación del *software*, si bien éstas, pueden ser flexibles de acuerdo a la metodología utilizada.

Los "procesos de desarrollo software" [*CUEV02*], tienen reglas establecidas que deber ser aplicadas en la creación del software de mediano y gran porte, ya que de lo contrario es muy posible que el proyecto no concluya satisfactoriamente. Entre tales "procesos" los hay livianos y pesados, con sus variantes intermedias. Por citar algunos, "Programación extrema", "Proceso unificado de Rational", "*Feature Driven Development (FDD)*", etc. Pero cualquiera que sea el proceso utilizado en el desarrollo del software, siempre se aplica un "modelo de ciclo de vida", siendo corriente en desarrollos de mediano porte el aplicar una metodología propia, normalmente un híbrido de los procesos anteriores (para más información se puede consultar la siguiente página web: http://es.wikipedia.org/wiki/Proceso_para_el_desarrollo_de_software).

Las etapas mínimas a cumplir durante el proceso de desarrollo son:

- Captura, análisis y especificación de requerimientos.
- Diseño
- Codificación
- Pruebas (unitarias y de integración)
- Instalación y paso a producción
- Mantenimiento

Aunque el nombre de éstas puede variar ligeramente, o pueden ser más generales o detalladas; por ejemplo, agrupando o diferenciando "Análisis y Diseño".

Estas etapas se subdividen a su vez en sub-etapas, siendo el "Modelo del proceso", o "Modelo de ciclo de vida" el que define el orden de las mismas, así como la coordinación, enlace y realimentación entre ellas. Algunos de los modelos más conocidos son: "M*odelo en cascada o secuencial*", "*modelo espiral*" y "*modelo iterativo incremental*", con sus variantes y alternativas más o menos atractivas en función la aplicación y sus requisitos. El "modelo en cascada" (también conocido como modelo clásico, tradicional, y lineal secuencial), difícilmente se utiliza tal cual, pues requiere un conocimiento previo y absoluto de los requisitos, que éstos no cambien y que las etapas posteriores estén libres de errores, lo cual hace que sólo sea aplicable a desarrollos

pequeños. En este modelo no hay vuelta atrás en el paso de una etapa a otra, por ejemplo, pasar del diseño a la codificación implica un diseño "perfecto", lo cual es bastante utópico. Cualquier cambio durante la ejecución de una etapa implica reiniciar el ciclo completo. Seguidamente se muestra un posible esquema de este modelo:

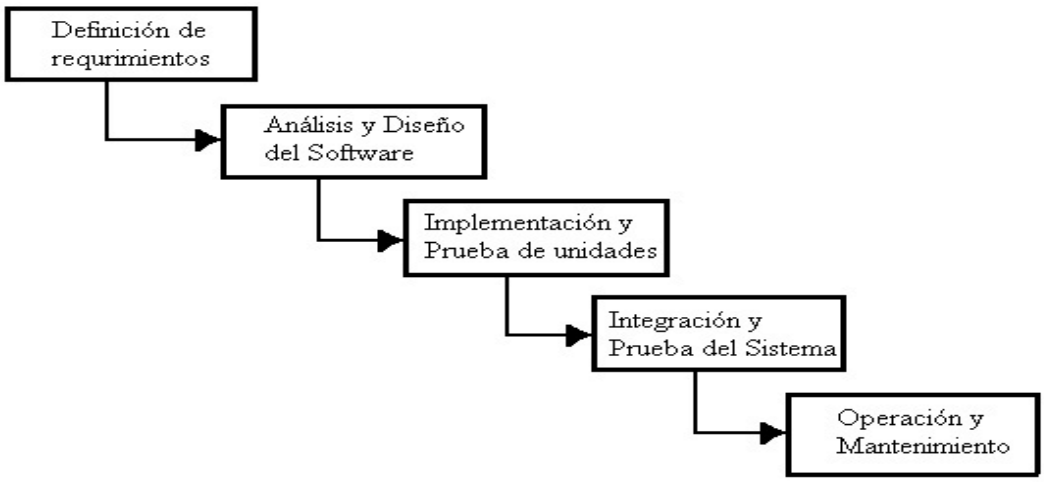


Figura 3: Modelo de cascada puro (ciclo de vida software)

Otro modelo más flexible es el "modelo en cascada realimentado". En éste se produce una realimentación entre etapas, lo que significa que se puede retroceder a una etapa anterior (saltándose a veces incluso más de una), si fuese requerido. Seguidamente se muestra un posible esquema:

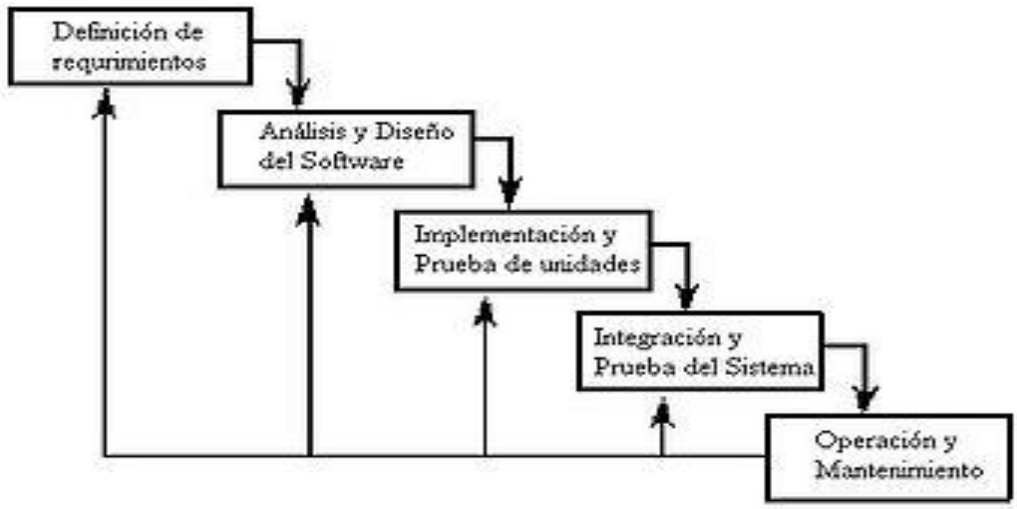


Figura 4: Modelo cascada realimentado (ciclo de vida software)

Este modelo es muy usado, es atractivo y hasta "ideal" si el proyecto presenta pocos o ningún cambio. Los requisitos son muy claros y están correctamente especificados. Sin embargo, pre-

senta algunos inconvenientes: los cambios en un etapa madura puede ser muy graves en un gran proyecto; no es frecuente que el cliente explicite los requisitos de una forma clara, completa y precisa; el producto final no estará disponible hasta muy avanzado el proyecto; etc.

Los modelos vistos no tienen en cuenta la evolución temporal del software. Por ello se requieren modelos diseñados para acomodarse a la evolución temporal, donde los requisitos centrales son conocidos de antemano aunque no así tanto sus detalles. Estos modelos conocidos como "evolutivos" permiten desarrollar versiones cada vez más completas y complejas hasta llegar al objetivo deseado. Los más conocidos son el "modelo iterativo incremental" y el "modelo espiral".

Figura 5: Modelo genérico evolutivo incremental (ciclo vida software)

El esquema de arriba nos muestra el funcionamiento de un ciclo iterativo incremental, el cual permite la entrega de versiones parciales a medida que se va construyendo el producto final. Cada versión emitida incorpora a las anteriores las funcionalidades y requisitos analizados.

El "modelo espiral" es un modelo evolutivo que conjuga la naturaleza iterativa del modelo de construcción de prototipos (MCP) con los aspectos controlados y sistemáticos del "modelo cascada". En este modelo se construye en una serie de versiones incrementales. El modelo se divide en un número de actividades de marco de trabajo, llamadas "regiones de tareas" (suele haber entre tres y seis). Las que se muestran en la figura de abajo son:

- Región 1: Tareas para establecer la comunicación entre el cliente y desarrollador.
- Región 2: Tareas inherentes a la definición de recursos, tiempo e información relacionada con el proyecto.
- Región 3: Tareas necesarias para evaluar los riesgos técnicos y de gestión.
- Región 4: Tareas para construir una o más representaciones de la aplicación.
- Región 5: Tareas para construir la aplicación, instalarla, probarla y proporcionar soporte al usuario (documentación y práctica.
- Región 6: Tareas para obtener la reacción del cliente, según la evaluación de lo creado e instalado en ciclos anteriores.

El modelo en espiral aporta un enfoque realista, que evoluciona igual que el software; se adapta muy bien para desarrollos en gran escala. Este modelo es particularmente apto para el desarrollo de Sistemas Operativos (complejos) y todos aquellos proyectos en los que sea necesaria una fuerte gestión del proyecto y sus riesgos, técnicos o de gestión.

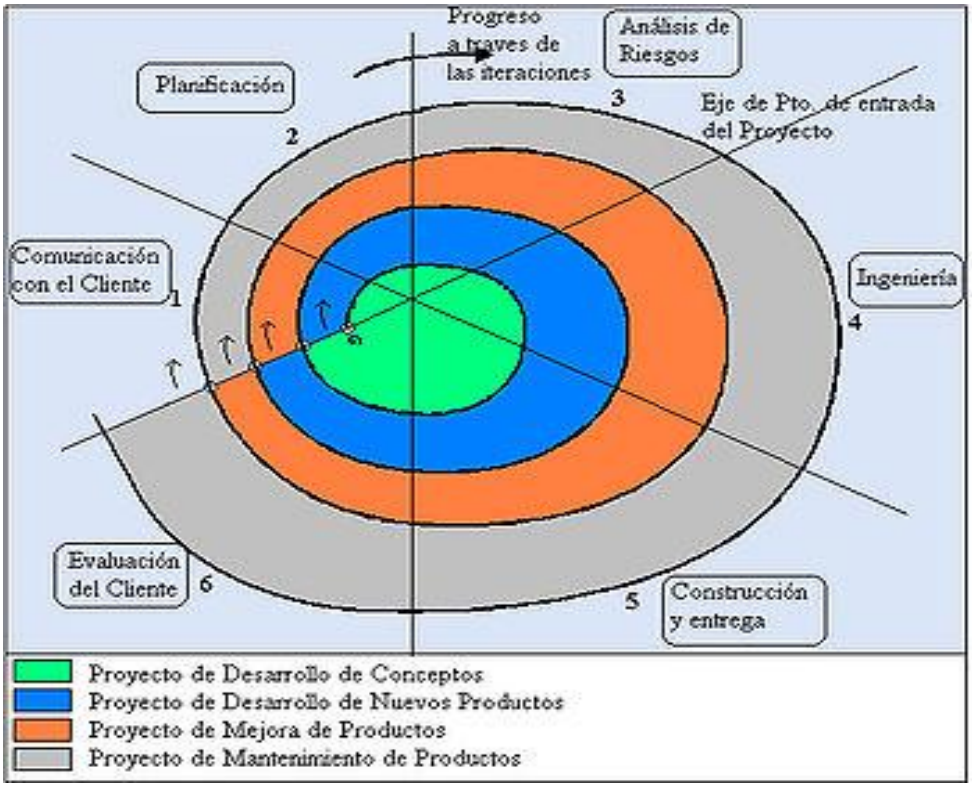


Figura 6: Modelo espiral (ciclo vida software)

3.2 Realización del proyecto

En este apartado se va a exponer cómo se ha realizado el proyecto, incluyendo las etapas y/o fases que seguidas para la realización del mismo.

En primer lugar, y a riesgo de ser reiterativo, diremos que este proyecto nació principalmente por la necesidad de contar con una herramienta muy personal de trabajo, para ayudar a la impartición de prácticas de sistemas operativos en un laboratorio de trabajo. Aunque las necesidades y requisitos mínimos eran en principio bien conocidos por el autor, y éstos se iban satisfaciendo a medida que transcurría el tiempo, globalmente se incrementaban, debido principalmente al deseo de introducir continuamente nuevas mejoras en el sistema.

En el proceso de creación del sistema se ha seguido un modelo del tipo "*evolutivo incremental*", aunque con muchos de sus aspectos muy particularizados y de un modo no muy ortodoxo. Todo ello debido fundamentalmente a las peculiares características del proyecto en cuanto al tiempo empleado en su desarrollo, su complejidad, los objetivos perseguidos y los recursos humanos disponibles (únicamente el autor).

En la primera etapa del proyecto, los objetivos no eran muy ambiciosos. En primer lugar se pretendía implementar, en primer lugar, el código de arranque de un sistema (*boot*), escrito en lenguaje "C", de un tamaño de código ejecutable no mayor de un sector (512 bytes), para que cupiera en el primer sector de un disquete, "el sector de arranque". Este "*boot*" se limitaría a cargar en un segmento de memoria dado, una cantidad dada de sectores consecutivos partiendo del primer sector de datos. Estos sectores, supuestamente, deberían contener el código del sistema operativo, para tras finalizar su carga, continuar la ejecución con la primera instrucción de la zona de memoria de carga. Por otro lado, y en segundo lugar, había que realizar un pequeño esqueleto de sistema operativo con funciones muy básicas, para ser cargado por dicho código '*boot*', siendo realmente esta segunda parte, el núcleo principal de este proyecto.

En esta segunda parte, inicialmente se pretendía tener un pequeño interprete de comandos y una gestión de ventanas con capacidad para visualizar datos de forma independiente, tal que se pudiese, en un futuro, asignar una ventana (y una entrada de teclas) independiente para cada uno de los procesos que se pudieran crear. En una segunda fase se implementaría la posibilidad de crear más procesos, con su ventana y teclado independientes, y con una planificación expulsora tipo

turno circular o "*round-robin*". En fases posteriores se irían creando las sucesivas llamadas al sistema y la especificación de los procesos de usuario, para poder ser cargados desde el pequeño intérprete o consola de comandos interna a 'SO'. A medida que el proyecto iba creciendo se pensó en la conveniencia de usar una herramienta de control de versiones, concretamente: '***git***'. Este producto está muy difundido, tiene un gran prestigio y es abierto. Es utilizado entre otros muchos proyectos para la propia gestión del kernel de "Linux". Este software y toda su documentación se puede encontrar en: http://git-scm.com, y se incluye como anexo junto con los fuentes del proyecto. Seguidamente, a título ilustrativo se muestran algunas de las pantallas que pueden obtenerse con 'git' sobre este proyecto:

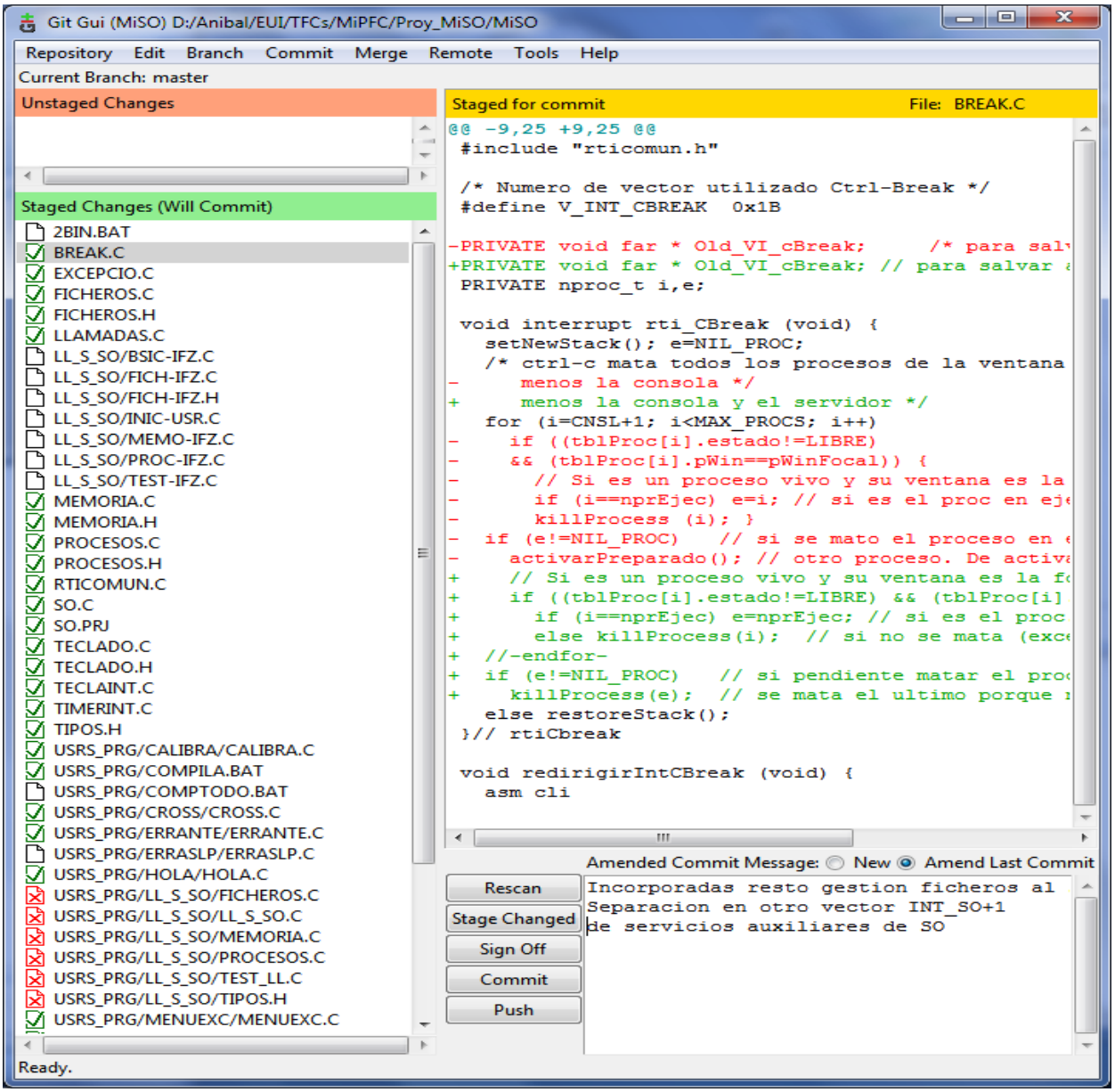


Figura 7: Ventana resultado ejecución comando 'git gui'

Desde esta ventana se pueden hacer nuevos "*commits*" (fijar cambios) al repositorio, o enmendar los ya realizados, crear nuevas ramas, fusionarlas, así como extraer o actualizar repositorios remotos. En la imagen anterior no se muestra la historia del proyecto. Para verla se puede arrancar la ventana del comando '**gitk**' la cual muestra la siguiente pantalla:

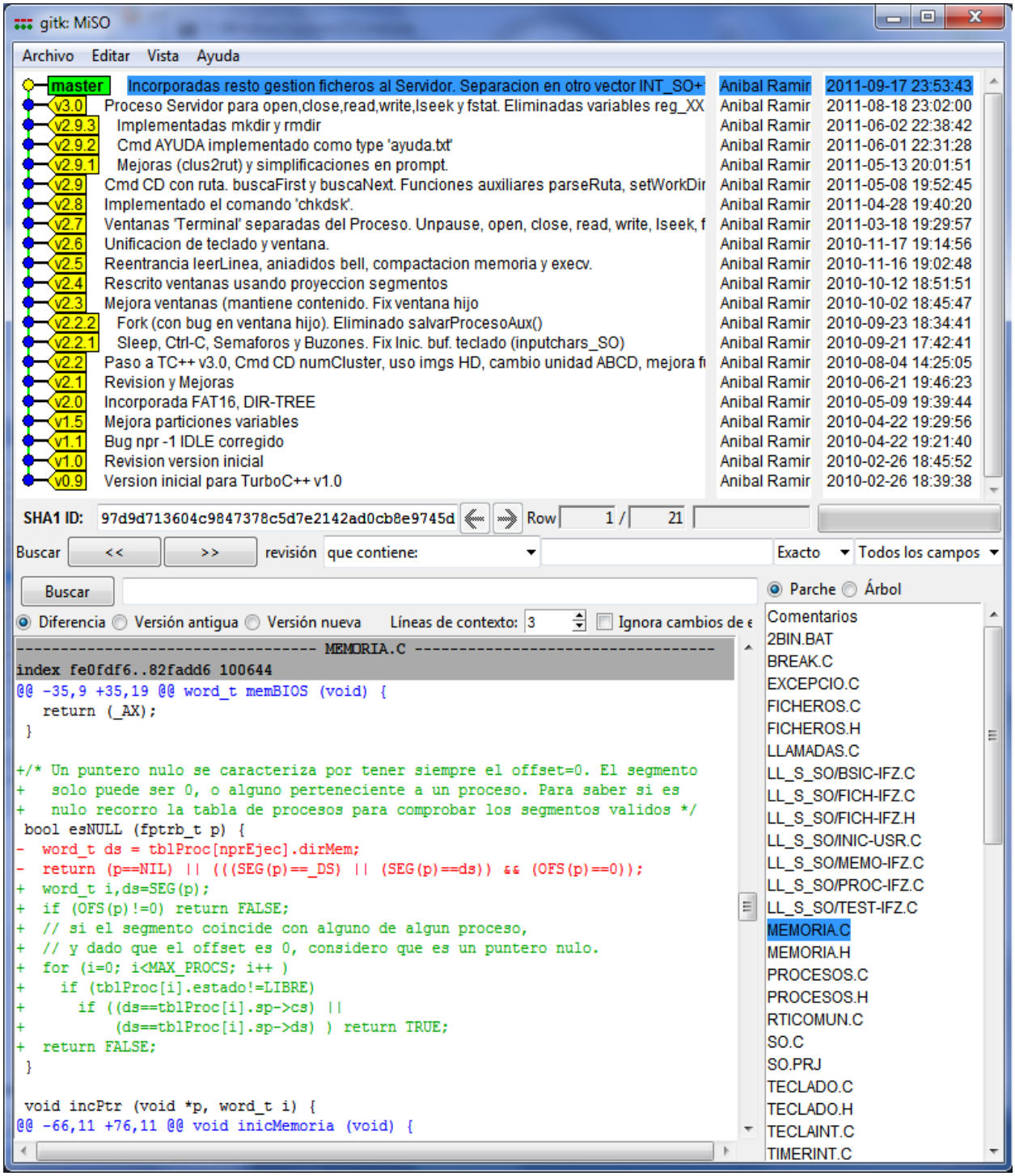


Figura 8: Ventana resultado ejecución comando 'gitk'

En la imagen superior se muestran los cambios de un repositorio. Esto incluye mostrar el grafo de "*commits*", mostrar información sobre cada cambio fijado ("*commit*"), los arboles de ficheros de cada revisión, etc. La siguiente imagen muestra la ejecución del comando "**git log**".

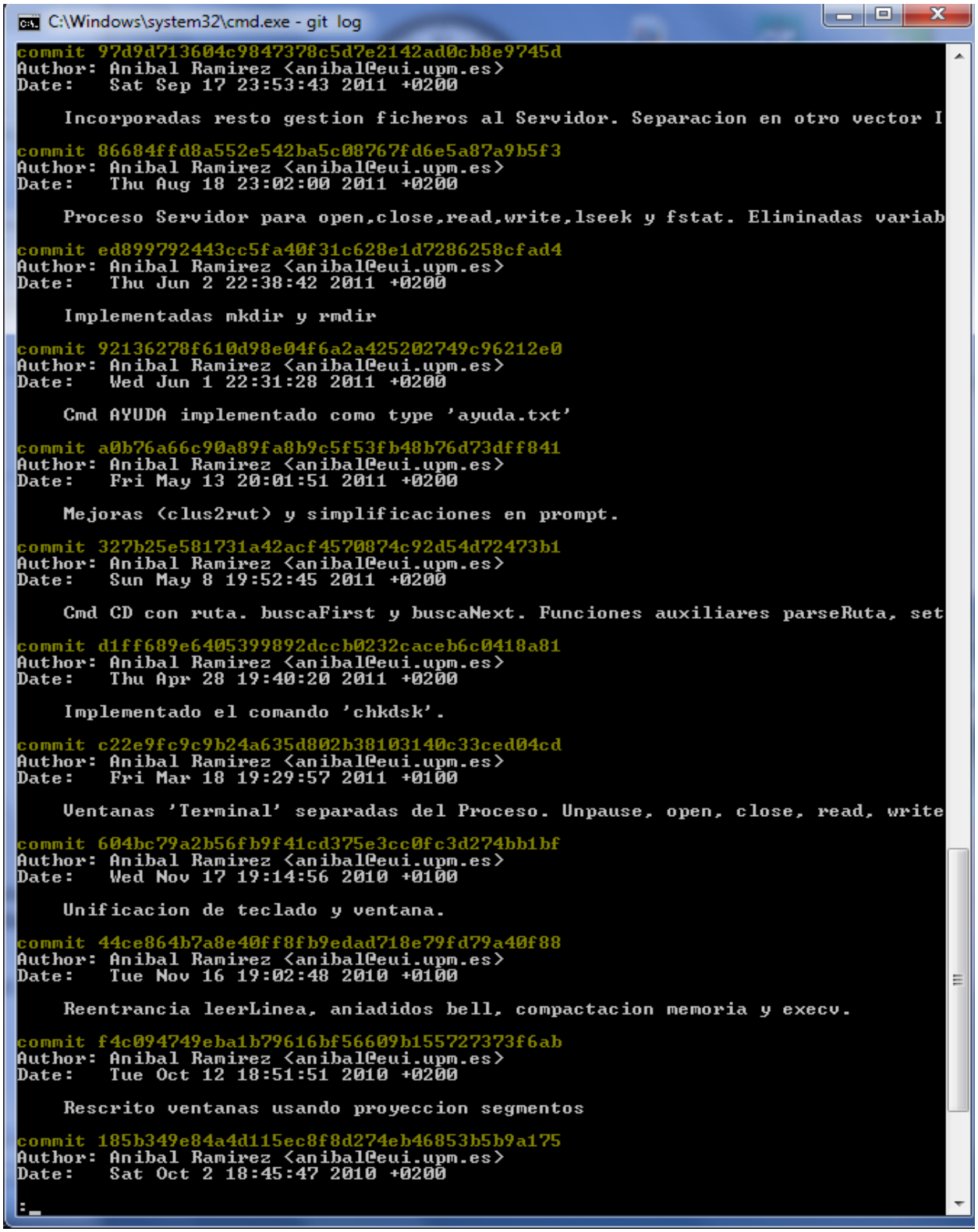


Figura 9: Ventana resultado ejecución comando 'git log'

4. ANÁLISIS DE REQUISITOS

El principal objetivo del proyecto, como ya se ha dicho, es que el estudiante disponga de una herramienta útil y adecuada para la comprensión de los principios y/o fundamentos de diseño y construcción de los sistemas operativos para ordenadores.

La herramienta fundamental para logar dicho objetivo es el propio sistema operativo a desarrollar, siendo una condición muy deseable el de poder utilizar dicho sistema con un software de emulación de maquina virtual (DOSBox, Qemu, etc.). Otra característica también muy deseable, es la de poder utilizar un entorno de desarrollo de aplicaciones para el desarrollo del mismo, tal que permita efectuar las operaciones de compilación, edición y depuración de forma integrada (IDE turbo C). Además, en aras a facilitar al máximo la sencillez del sistema, y por último, hay que citar también como un requerimiento fundamental el que el S.O. esté diseñado para funcionar en el modo "real" de la familia de procesadores i386.

A continuación, se van a relacionar los requisitos básicos que debe cumplir el S.O. para satisfacer las necesidades y requerimientos de este proyecto fin de carrera.

Identificación	ID-01	Tipo	Funcional	Nivel	Imprescindible
Descripción	Soporte de multitarea expulsora, con creación y destrucción de procesos				
Motivación	Tener la posibilidad de elaborar y modificar las políticas de planificación expulsoras, por prioridad, etc.				
Comprobación del cumplimiento	Crear y destruir procesos varios procesos, poniéndolos en ejecución concurrente, comprobando el buen funcionamiento.				

Identificación	ID-02	Tipo	Funcional	Nivel	Para prácticas
Descripción	Comunicación entre procesos mediante buzones de capacidad fija				
Motivación	Tener la posibilidad de sincronizar procesos así como de transferir información entre ellos mediante el uso de buzones. Esta característica permite el estudio de diferentes técnicas de implementación.				
Comprobación del cumplimiento	Envío y recepción de mensajes entre procesos, comprobando el bloqueo de los procesos en situaciones de llenado y vaciado del buzón.				

Identificación	ID-03	Tipo	Funcional	Nivel	Para prácticas
Descripción	Sincronización de procesos mediante semáforos				
Motivación	Tener la posibilidad de crear regiones críticas mediante semáforos. Esta característica permite el estudio de su implementación.				
Comprobación del cumplimiento	Uso de programas de usuario con alguna región críticas para implementar con algún semáforo.				

Identificación	ID-04	Tipo	Funcional	Nivel	Imprescindible
Descripción	Gestión de memoria mediante particiones de tamaño variable				
Motivación	La creación y destrucción de procesos requiere que se les asigne memoria a la medida de su tamaño y posteriormente se libere. También se requiere para que el estudiante pueda ampliar y modificar las políticas de asignación de la memoria.				
Comprobación del cumplimiento	Monitorizar la lista de huecos de memoria tras la creación y destrucción de procesos. Realización de programas que pidan y devuelvan memoria siguiendo diferentes patrones con comprobación de la consistencia de la memoria gestionada.				

Identificación	ID-05	Tipo	Funcional	Nivel	No Imprescindible
Descripción	Gestión de memoria con intercambio de procesos a disco				
Motivación	Aumentar el nivel de multiprogramación mediante el intercambio de procesos de memoria a disco.				
Comprobación del cumplimiento	Monitorizar la lista de huecos de memoria tras la creación de un proceso que no cabe en memoria requiriendo la expulsión a disco de algún otro proceso para aumentar el espacio de memoria disponible.				

Identificación	ID-07	Tipo	Funcional	Nivel	Imprescindible
Descripción	Gestión de ventanas terminal modo texto, de tamaño variable y con su propia cola de teclado, tal que cada proceso tenga asignada una de estas ventanas y una ventana pueda tener asignados más de un proceso.				
Motivación	Permitir a los procesos tener su propio espacio de entrada/salida en forma de ventana.				
Comprobación del cumplimiento	Al crearse un proceso, éste deberá tener asignada una ventana para su entrada/salida, pudiendo comprobarse que los mensajes en pantalla salen por dicha ventana y que cuando ésta se encuentre focal, las teclas pulsadas por el usuario se dirigen al proceso.				

Identificación	ID-08	Tipo	Funcional	Nivel	Imprescindible
Descripción	Gestión de ventanas terminal modo texto, de tamaño variable y con su propia cola de teclado, tal que cada proceso tenga asignada una de estas ventanas y una ventana pueda tener asignados más de un proceso.				
Motivación	Permitir a los procesos tener su propio espacio de entrada/salida en forma de ventana.				
Comprobación del cumplimiento	Al crearse un proceso, éste deberá tener asignada una ventana para su entrada/salida, pudiendo comprobarse que los mensajes en pantalla salen por dicha ventana y que cuando ésta se encuentre focal, las teclas pulsadas por el usuario se dirigen al proceso.				

Identificación	ID-09	Tipo	Funcional	Nivel	Imprescindible
Descripción	Gestión completa de ficheros tipo FAT12 y FAT16, incluyendo al menos llamadas al sistema básicas: crear, abrir, posicionar, leer, escribir, cerrar y borrar ficheros, y operaciones básicas sobre directorios tales como: crear y borrar. Soportará también el concepto de rutas relativas y absolutas de tanto uso en la actualidad en otros sistemas.				
Motivación	Permitir el acceso y escritura de información en disquete y en disco duro, con formato creado por sistemas tipo MS-DOS.				
Comprobación del cumplimiento	Escritura de aplicaciones y/o comandos con uso de llamadas al sistema relacionadas con ficheros, como: leer, escribir, crear directorio, etc.				

Identificación	ID-10	Tipo	Funcional	Nivel	Imprescindible
Descripción	Gestión completa de ficheros tipo FAT12 y FAT16, incluyendo al menos llamadas al sistema básicas: crear, abrir, posicionar, leer, escribir, cerrar y borrar ficheros, y operaciones básicas sobre directorios tales como: crear y borrar. Soportará también el concepto de rutas relativas y absolutas de tanto uso en la actualidad en otros sistemas.				
Motivación	Permitir el acceso y escritura de información en disquete y en disco duro, con formato creado por sistemas tipo MS-DOS.				
Comprobación del cumplimiento	Escritura de aplicaciones y/o comandos con uso de llamadas al sistema relacionadas con ficheros, como: leer, escribir, crear directorio, etc.				

Identificación	ID-11	Tipo	No Funcional	Nivel	Muy deseable
Descripción	Ejecución del sistema bajo MS-DOS sin uso de sus servicios.				
Motivación	Facilitar el desarrollo del sistema, ya que éste se realiza bajo DOS y con el entorno integrado de desarrollo de “Turbo-C”, el cual permite ejecutar y depurar programas, y no sería deseable tener que renunciar a esta característica porque el sistema no lo soportase (aunque existe un depurador de Borland: "*Turbo Debugger*", que es independiente del entorno integrado y permite depurar aplicaciones DOS, sería deseable que el sistema no lo requiriese; y pudiese ser ejecutado y depurado, desde el propio entorno integrado que acompaña al compilador de Turbo C).				
Comprobación del cumplimiento	Ejecución bajo DOS con intercepción de los servicios DOS.				

Identificación	ID-12	Tipo	No Funcional	Nivel	Muy deseable
Descripción	Manejo sencillo de tratamiento de interrupciones, tanto para implementar la multitarea, como para el teclado, ofreciendo posibilidades de ampliación a otros dispositivos como el puerto serie y el ratón.				
Motivación	Facilitar al máximo la compresión del tema al estudiante.				
Comprobación del cumplimiento	Valoración subjetiva expertos. Encuestas de opinión.				

Aunque los requisitos del sistema operativo que se pretende implementar reúne características bastante interesantes, éste no pretende ser usado para otro propósito diferente al de la enseñanza de los "Sistemas Operativos". En el análisis de requisitos se pueden encontrar muchas carencias , por ejemplo, no se considera en principio necesario que soporte muchas de las características que en otros sistemas son básicas, tales como: la "redirección" de entrada/salida, la incorporación de nuevos "*drivers"*, ni en tiempo de carga ni con posterioridad; la equiparación de dispositivo físico al concepto lógico de fichero; un diseño por niveles o capas; diferenciación entre niveles de privilegio de usuario y supervisor; líneas de comunicación serie, Ethernet; etc.

5. DISEÑO E IMPLEMENTACIÓN

En este capítulo se va a detallar principalmente y hasta cierto nivel, el diseño e implementación de los componentes y módulos del sistema operativo. Primeramente se verán algunos pormenores "básicos" sobre las herramientas del entorno de desarrollo del sistema, ya que ha sido necesario "retocar" algunos componentes del compilador empleado, para cumplir con algunos de los objetivos buscados por este proyecto. Posteriormente se mostrarán los aspectos más relevantes que se han tenido en cuenta en el diseño e implementación de este pequeño sistema operativo, pasando estos por sus componentes básicos, servicios ofrecidos, modelo de diseño, rutinas de interfaz, etc.

5.1 Estudio preliminar de las herramientas de desarrollo.

En primer lugar hay que decir que ha sido conveniente seleccionar un sistema de emulación de "máquina virtual" de entre las diversas opciones disponibles, ya que es más sencillo adaptar el proyecto a un sistema específico que dejarlo abierto y operativo a cualquier otra posibilidad, no obstante, aun ciñéndose a uno concreto, el proyecto no debe quedar relegado únicamente a dicho sistema, y ha de poder ser adaptado fácilmente a cualquier otro. De los sistemas de virtualiza-

ción estudiados, el que se ha juzgado más apropiado para el presente proyecto, ha sido "***DOS-Box***" junto con el entorno de interfaz gráfico "***D-Fend***", aunque no ha sido fácil dicha elección dada las buenas cualidades que ofrecen las otras opciones disponibles (*Qemu, Bosch, Virtual PC, Virtual Box,* etc.). En principio cualquiera de ellas podría ser apta para el proyecto, sin embargo, se escogió finamente "*DOSBox-DFend*", porque la conjunción de ambos ofrecía determinadas características deseables que no llegaban a aportar los otros productos por una u otra razón. De entre éstas se van a citar a continuación las más relevantes:

- Disponibilidad de un interfaz gráfico de ventanas (*D-Fend)* que facilita en gran medida la configuración de la máquina virtual.
- Gran fiabilidad en la emulación de la máquina real (También la ofrecen otras opciones como *Qemu, Virtual PC*, etc., pero adolecen de algún inconveniente, por ejemplo *Qemu* no presenta un teclado español totalmente compatible, lo que dificulta el uso del entorno de *turbo C++*, o *Virtual PC* no ofrece ventanas de distintos tamaños, etc.).
- El sistema es portable, es decir, por ejemplo, en "Windows", no se requiere instalación, pudiendo funcionar en cualquier carpeta y en cualquier unidad (un "*pendrive*" es muy práctico para el estudiante).
- El sistema está disponible para varias plataformas: Windows, Linux, Apple iMac y es *software* abierto.
- Ofrece la posibilidad de elegir el tamaño de la ventana de pantalla de la máquina virtual (disminuye la fatiga visual. Esta característica también la ofrecía "Qemu" en sus últimas versiones).
- Muy buena integración con el entorno de desarrollo integrado Turbo C++.

Una vez resuelta la decisión de qué *software* a utilizar como máquina virtual, como segundo paso, había que decidir también qué leguaje y compilador utilizar. La elección del leguaje fue "C", por ser la opción tradicional para el desarrollo de sistemas operativos. En cuanto al compilador, "*Borland Turbo C*" era un compilador que el autor conocía bastante bien y en varias versiones, y además éste reunía las cualidades necesarias para el desarrollo del proyecto, por lo que no se consideró necesario explorar más posibilidades. Esta herramienta aporta un entorno integrado de desarrollo (IDE) que permite compilar, ejecutar y depurar de forma muy cómoda, lo que hace que sea una herramienta muy buena para el desarrollo del sistema y para ser utilizada por los estudiantes. Sin embargo, ha sido necesario retocar el módulo de inicialización "***C0t.obj***", para

poder compilar programas que no fuesen a ejecutarse necesariamente bajo MS-DOS. En este módulo se incluía el código de inicialización del programa bajo MS-DOS y ha sido eliminado. En su lugar se ha dejado únicamente una inicialización muy básica para el nuevo sistema, consistente en guardar en la variable ***DGROUP@*** (que usa Turbo C) el valor del registro ***DS***, para que posteriormente, cuando se invoquen las rutinas de tratamiento de interrupción (tipo *void interrupt*), se restaure dicho registro ***DS*** al valor del segmento de datos de 'SO' (este código lo genera automáticamente el compilador en el modelo "*small*" para las funciones del tipo "*void interrupt()*". Una vez hecho esto, se invoca la función *main()* y finalmente la instrucción ***INT 20***, para contemplar el caso de ejecución como invitado de MS-DOS (ver fichero "***MI_C0t.asm***" en el directorio "***MI_C0***").

Otra particularidad a tener en cuenta respecto al uso del compilador "Turbo C++ v3.0", la emulación de máquina virtual "DOSBox" y el sistema 'SO' es la siguiente: cuando se ejecuta 'SO' como invitado de MS-DOS y desde el entorno integrado de Turbo C, el depurador no funciona adecuadamente, porque el vector de teclado que captura (y al salir restituye) 'SO', no es el original del MS-DOS (o BIOS), sino el que previamente ha establecido Turbo C, lo cual interfiere con el depurador provocando cuelgues, para evitarlo, el sistema intenta discernir si el sistema se ejecuta sobre en el entorno de Turbo C. Si es así, sólo se guarda el vector de teclado de DOSBox v.074, que es ***F000:E987***, en caso contrario, se limita a leer y guardar el vector de teclado anterior. Por todo ello, si se cambiara de versión de DOSBox o se usara otro emulador, habría que cambiar el valor mencionado o implementar otra solución (ver módulo "*teclaint.c*"). Estos detalles, sin ser esenciales para el proyecto, sí son importantes, porque la posibilidad de usar el depurador de Turbo C para detectar y corregir errores de programación del sistema es una cualidad muy deseable, tanto para ser usada por el estudiante como por cualquier programador que deseara trabajar en el proyecto.

5.2 Modelo de diseño.

Este pequeño sistema operativo se ha diseñado básicamente como un sistema monolítico, ya que en principio es más fácil de implementar, al estar todo el sistema en un único programa ejecutable, el cual reúne toda la funcionalidad. El programa en el que se ha compilado todo el sistema se denomina "SO.EXE" para el entorno de ejecución MS-DOS (aunque éste no se usa) o bien "SO.BIN" (se convierte a ".bin" mediante la conocida utilidad "*exe2bin.exe*", la cual básicamente

elimina la cabecera del fichero ejecutable requerida por MS-DOS), si se va a ejecutar directamente sobre el hardware tanto físico como virtual. El esquema de la estructura de este diseño es el siguiente:

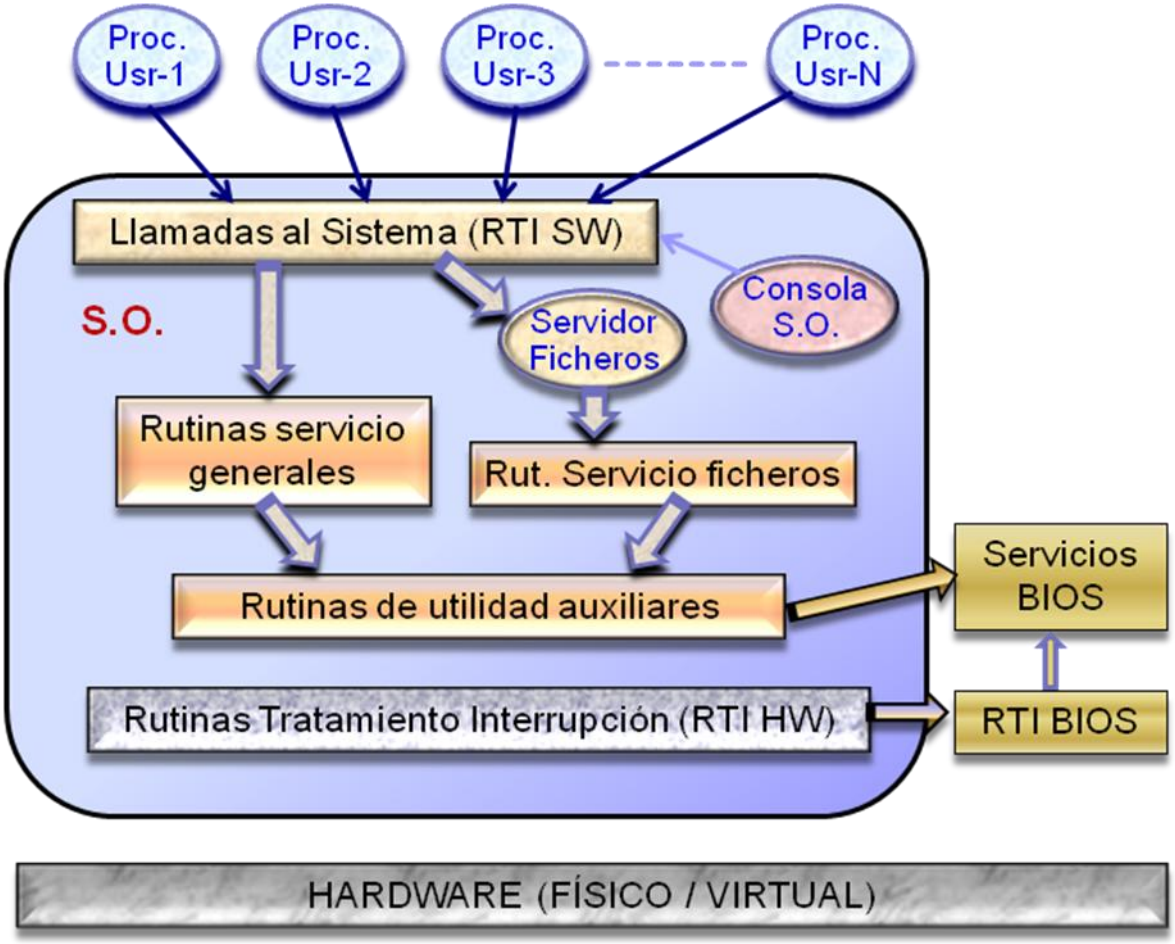


Figura 10: Esquema de diseño del S.O.

En el esquema se puede observar que hay dos procesos que corren en el espacio del sistema operativo ('SO'): el Servidor de Ficheros y la Consola (interprete de comandos interno). La Consola se limita a interpretar comandos y ejecutarlos recurriendo a llamadas al sistema como cualquier otro proceso de usuario, aunque es privilegiado porque tiene acceso a cualquier variable o función interna del sistema operativo. El otro proceso, el Servidor de Ficheros, es un proceso que sirve para atender peticiones de llamadas al sistema relacionadas con ficheros. El Servidor dispone de una cola de la cual va extrayendo y ejecutando peticiones por orden de llegada. Si la cola estuviese vacía el Servidor se duerme, despertando en cuanto se añade alguna nueva llamada a la cola. Con este método no se puede llevar a cabo más de una petición de ficheros a la vez, evitando así problemas de concurrencia, no obstante, sí permite que se lleve a cabo, concurrentemente, cualquier otra petición general (no de ficheros).

Las llamadas al sistema se reciben mediante la invocación de una rutina de tratamiento de interrupción (RTI) software dedicada a ello y en ella se determina de qué llamada se trata, redirigiendo la petición a la rutina de servicio apropiada, la cual a su vez se apoya en las rutinas de utilidad y funciones de la BIOS. En el caso particular de las llamadas relacionadas con ficheros, la petición es encolada, para ser posteriormente extraída y servida por el Servidor de Ficheros ya mencionado.

Por otro lado los dispositivos (*hardware*) producen interrupciones que procesan las rutinas de tratamiento de interrupción *hardware*, las cuales entre otras cosas, se encargan del teclado (interpretando y encolando teclas en cada uno de los *buffers* de teclado de las ventanas terminal), de reloj (controlando los "*quantum*" de tiempo, alarmas, etc.), y en general de cualquier dispositivo que haya que controlar (ratón, rs232, etc.).

5.3 Principales componentes del Sistema.

En este sistema se han implementado los típicos componentes de un sistema operativo: ***Gestión de Procesos*** (con multitarea expulsora), ***Memoria*** (particiones variables), ***Entrada/Salida*** y ***Ficheros*** (FAT-12 y 16) y una consola interna como intérprete de comandos (éstos y otros conceptos generales sobre sistemas operativos que se ven a lo largo de este capítulo se pueden consultar en la amplia bibliografía existente en la actualidad al respecto, como muestra pueden consultarse: [TANE09, CAND07, CARR01], etc.; en la sección final de bibliografía se citan algunas referencias más). Otros componentes como gestión de redes, protección, etc. se han dejado para futuras ampliaciones. Todos estos componentes se hayan integrados en el núcleo, ya que este sistema, como ya se ha dicho, tiene un diseño monolítico. No obstante, el código está estructurado en ficheros independientes (para compilación separada) que forman parte de un proyecto de Turbo C. Cada uno de estos ficheros agrupa las diferentes funciones de cada uno de estos componentes. En los sub-apartados siguientes se muestran las peculiaridades de diseño e implementación de cada uno de estos componentes.

5.3.1 Gestión de Procesos:

La gestión de procesos esta implementada en el módulo de proyecto "procesos.c", aunque las constantes, tipos de datos, declaración de variables externas y prototipos de función visibles a otros módulos se encuentran en el fichero "procesos.h", como es habitual en el lenguaje de programación 'C'. En "procesos.h" están declarados los siguientes tipos de datos:

- "estado_t". Para definir el estado de un proceso: LIBRE, EJECUCIÓN, PREPARADO y BLOQUEADO.
- "esperaPor_t". Para especificar la razón por la que un proceso está bloqueado, a la espera de un evento tal como: la pulsación de una tecla, vencimiento de un lapso de tiempo, apertura de un semáforo, recepción de un mensaje en un buzón, ser atendido por el servidor de ficheros, salir de un "*pause*", o por cualquier otra razón.
- "descrProc_t". Estructura descriptor de proceso. Contiene los siguientes campos: estado del proceso; razón de la espera (para el caso de bloqueo); dirección de la cola del recurso causa de bloqueo (o NULL si no está bloqueado); identificador del proceso (pid) y del proceso padre (ppid); siguiente proceso (si el proceso forma parte de una cola); dirección de la pila (donde se guardan los registros del procesador parte del contexto del proceso); comienzo y tamaño del bloque de memoria asignado al proceso; dirección de la ventana terminal del proceso; nombre y tamaño del fichero ejecutable; unidad de disco (*drive*) y *cluster* de trabajo del proceso; tiempo restante para despertar (para el caso de dormido), y por último; tabla de descriptores de ficheros abiertos.

Las variables que este módulo exporta son:

- Tabla de descriptores de proceso.
- Colas de procesos: preparados, dormidos y pendientes de ser atendidos por el servidor de ficheros.
- Numero de procesos vivos.
- Proceso en ejecución.
- Contador de tics por rodaja (para planificación round robin).

Las constantes declaradas y exportadas son:

TICS_POR_RODAJA (tamaño del quantum), origen o base de la pila para el proceso servidor de ficheros , y números de proceso de los procesos: ocioso (IDLE=-1), servidor de ficheros (SERV=0), y la consola interna (CNSL=1).

Las funciones exportadas son:

- *inicProcesos()*. Inicialización del módulo.
- *nuevoPid()*. Devuelve un nuevo identificador de proceso "pid".
- *nproc(pid)*. Devuelve el número de proceso (índice de la tabla de procesos) a partir de un identificador de proceso.
- *encolar(cola, npr)*. Encola un proceso en una cola.
- *desencolar(cola)*. Devuelve el primer proceso de la cola y lo quita de ella.
- *quitarDeCola(cola, npr, nprAnt)*. Suprime el proceso 'npr' de la 'cola' en cualquier posición. No lo busca, se ha de pasar el proceso anterior 'nprAnt'.
- *buscarEnCola(cola, npr, *pErr)*. Busca el proceso 'npr' y devuelve el anterior si lo encuentra, sino pone *pErr=TRUE.
- *activarProceso(npr)*. Pone en ejecución al proceso 'npr'.
- *activarPreparado()*. Pone en ejecución el primero proceso "preparado".
- *bloquearProceso(por, cola)*. Pone al proceso que está ejecutándose en estado BLOQUEADO con la razón especificada en 'por'; encola el proceso al recuso si se especificó 'cola' y por último pone en ejecución al primer proceso preparado.
- *crearProceso(part, size, *name)*. Crea un nuevo proceso en el bloque de memoria especificado en 'part', de tamaño 'size', con el nombre del fichero ejecutable 'name'. El proceso recibe una nueva recién creada ventana terminal.
- *killProcess(npr)*. Mata el proceso 'npr liberando todos sus recursos.
- *reboot()*. Reinicia la maquina llamando a la BIOS - INT 19
- *finProgDos()*. Finaliza 'SO' cuando corre sobre MS-DOS (aunque no lo usa)
- *getDate()*. Devuelve la fecha.
- *listarProcesos()*. Para uso interno de la consola para listar los procesos vivos.

De algunas de estas funciones conviene aclarar algo más su funcionamiento dada su importancia para el sistema:

La función "*inicProcesos()*" se encarga de preparar los datos relacionados con procesos. Inicializa la tabla de procesos a 0's, para después inicializar las entradas correspondientes a los procesos servidor (SERV) y consola (CNSL). Establece el número de procesos vivos a 2, deja como proceso en ejecución (nprEjec) la consola y pone en la cola de preparados como único proceso al proceso servidor de ficheros (SERV). Hay que señalar que estos dos procesos al ser internos a SO tienen algunos campos con valores atípicos, por ejemplo el valor del segmento de memoria donde están cargados (*dirMem*) es el mismo en ambos, que es el valor del registro **CS**. El campo tamaño de la consola tiene el valor de todo el sistema y el del servidor vale 0. Estos valores se usan en procesos normales para liberar la memoria ocupada al morir el proceso, pero estos procesos siempre están vivos y además no reciben la memoria como el resto de procesos, mediante asignación dinámica, por lo que el valor es puramente informativo. Por otro lado también hay que señalar que la ventana terminal es la misma para ambos y por último decir que tanto la consola como el servidor tienen su propia pila dentro del espacio de SO indicada por las constantes **SRV_PILA** y **BASE_PILA**.

La función "*activarProceso(npr)*" es bastante pequeña, sin embargo su contenido no es trivial. Se encarga de poner en ejecución el proceso "npr", del que se supone que debe tener en su pila el valor de los registros del procesador, tal y como estaban cuando sucedió la última pérdida del procesador por parte de dicho proceso, bien por cesión voluntaria, o bien por expulsión (fin rodaja o cualquier otra razón). Esta función pone el estado del proceso a "EJECUCIÓN", recupera la pila del proceso de su descriptor y tras una serie de instrucciones POP, acaba con la instrucción IRET, la cual sirve para retornar de una interrupción (hardware o software => INT). Conviene decir que en Turbo C existe el tipo de función "void interrupt()" que se utiliza para implementar las rutinas de tratamiento de interrupción. En este tipo de función, el compilador incluye implícitamente al comienzo de la función, una serie de instrucciones PUSH que se encargan de guardar en pila el valor de los registros del procesador, y análogamente, al final de la función, también incluye una serie de instrucciones POP para recuperar dichos valores. Son estas POP's la que se incluyen explícitamente en la función "activarProceso()" al no ser esta función del tipo "void interrupt()" y quererse obtener el mismo efecto.

La función "*activarPreparado()*" se utiliza cuando hay que poner en ejecución otro proceso, bien porque muere, se bloquea o vence el quantum de tiempo del proceso en ejecución (en este último caso sólo si hay algún proceso preparado). En todos estos casos hay que seleccionar otro proceso para ponerlo en ejecución. Si al tratar de seleccionar otro proceso la cola estuviese vacía, enton-

ces se para el procesador (instrucción ***HLT***) y se pone "IDLE (ocioso)" como proceso en ejecución. Cualquier interrupción reactivaría la CPU (sacándola de "halt" y reanudando el bucle). Si durante la ejecución de la rutina de tratamiento de interrupción (***RTI***) algún proceso pasase a preparado, entonces se abandonaría el bucle, seleccionando al primer proceso preparado y retornando. Seguidamente se activaría dicho proceso y continuaría su ejecución.

La función "*bloquearProceso(por, cola)*" pone el estado del proceso que está ejecutándose a "BLOQUEADO" especificando la razón del bloqueo. Si se especifica "cola", pone al proceso a bloquear en dicha cola (algunos recursos se pueden implementar sin uso de cola), y por último, tiene que poner a otro proceso en ejecución por lo que llama a activarPreparado(). Hay que señalar que la función "bloquearProceso()" no guarda el estado de los registros del procesador en pila. Estos registros se supone que ya están salvados en la pila del proceso, porque cuando se invoca esta función se hace durante la ejecución de una ***RTI*** (bien sea esta una interrupción hardware como el reloj, teclado, etc., o una llamada al sistema).

La función "*crearProceso(dirM, size, name)*" se encarga de crear un nuevo proceso que ubica en el segmento de memoria 'dirM', de tamaño 'size', (previamente solicitado al módulo de memoria), y le asigna el nombre dado en 'name'. Esta función debe encontrar un descriptor libre para el proceso y completar todos sus campos. Hay que destacar el campo 'pWin' que contiene la dirección de la ventana del proceso, la cual se crea solicitando memoria e inicializándola después mediante "initWin" del módulo de ventanas. El campo "sp" guarda la dirección de la pila del proceso. Su valor es la dirección más alta del segmento ubicado para el proceso menos el tamaño de la trama de registros del procesador que ha de introducirse en el momento de la creación del proceso. De los valores de estos registros sólo son relevantes los registros flags, CS e IP, sobre todo CS:IP ya que deben contener la dirección de comienzo del programa, la cual deber ser: 0 para IP y *dirMem* para CS. Hecho esto el proceso se deja en "preparados", para que cuando se active, se restauren los registros del procesador (inicialmente si importancia), y al ejecutar la instrucción IRET la CPU salte a la dirección del comienzo del proceso => dirMem : 0.

La función "*killProcess(npr)*" mata al proceso 'npr' teniendo en cuenta su estado. En primer lugar comprueba que el proceso sea uno válido, si es así, cierra cualquier fichero que pudiera estar abierto por él. Seguidamente comprueba que la ventana del proceso solo la usa él, para liberarla en ese caso o dejarla como está en caso contrario. A continuación, si el proceso estaba preparado o estaba bloqueado y encolado en alguna cola de recurso, lo quita de dicha cola. Para finalizar, y

entre otras cosas, si el proceso padre estaba en 'PAUSE' se pone en preparados, y si el proceso que muere estaba en ejecución se selecciona otro proceso a ejecutar. **Nota:** S*e despierta al padre si éste está en estado PAUSE porque cuando al crear un hijo mediante "fork()", el padre puede quedarse esperando finalización del hijo mediante "pause", por ello al morir el hijo debe despertar al padre si estuviera en dicho estado (Este comportamiento sería mejorable mediante el empleo de señales [todavía no implementadas]).*

Figura 11: Diagrama de estados de un proceso

En la imagen superior se muestra el típico diagrama con los estados en los que se puede encontrar un proceso particularizados para este sistema. El estado de bloqueado puede diferenciarse por el campo '***esperaPor***' del descriptor del proceso, pudiéndose además controlar, para algunos recursos, qué procesos esperan por un determinado recurso mediante una cola para el mismo, tal y como se puede ver en el esquema siguiente.

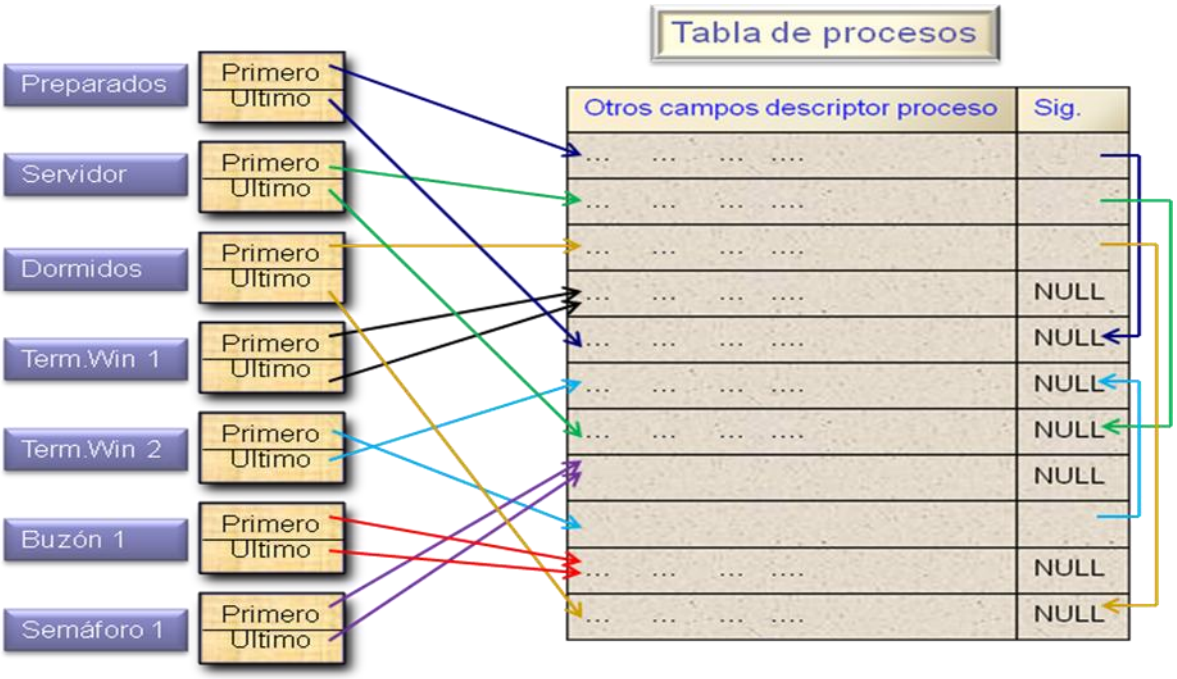


Figura 12: Esquema de colas de recursos

En la imagen de la figura siguiente se muestra un diagrama con la transición entre colas de espera de los diferentes recursos del sistema para un proceso. Sólo se muestra un recurso ventana terminal, al igual que un sólo semáforo y buzón. Además para este último caso, se muestran unificadamente los casos de envío y recepción de mensajes, con la precisión de que el encolamiento sólo se produce si el buzón está lleno o vacío según sea el caso de envío o recepción respectivamente.

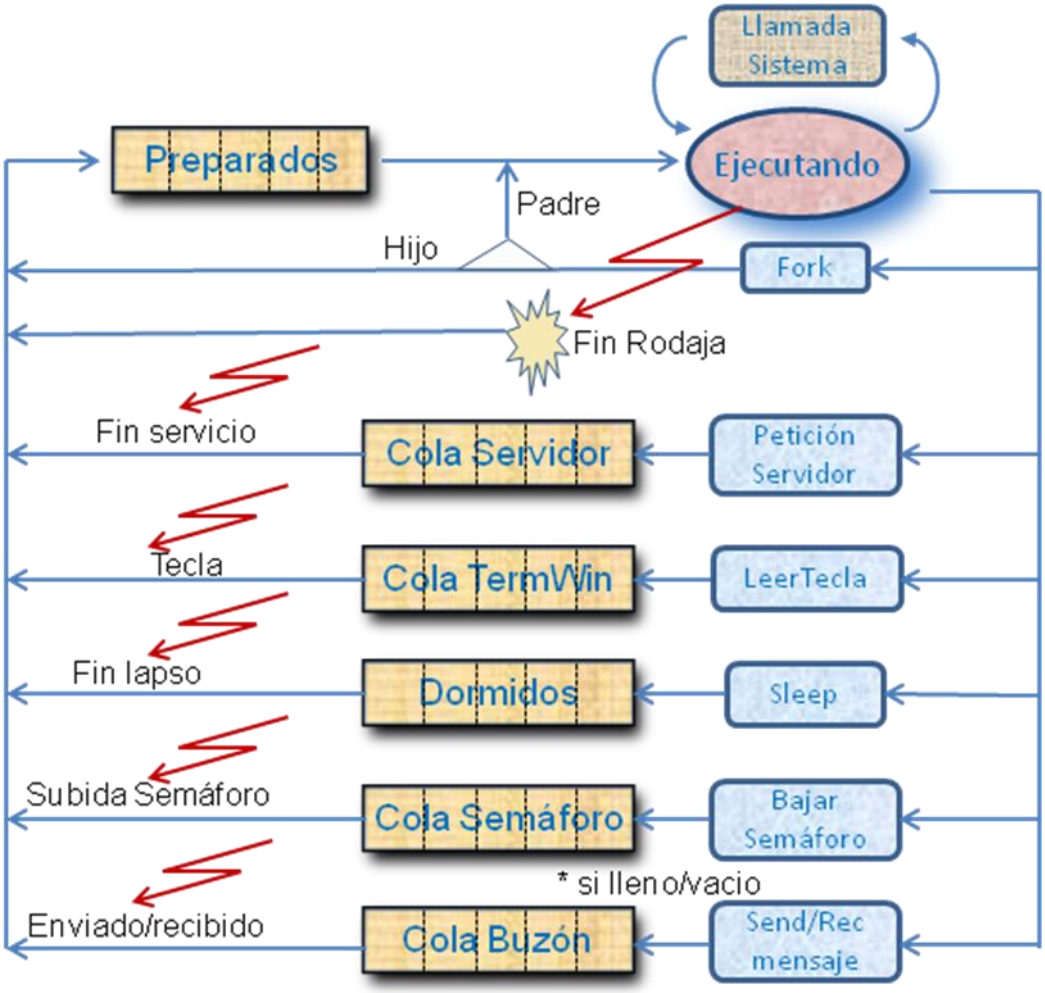


Figura 13: Transición de un proceso entre las colas del sistema

En la figura siguiente se muestra un esquema de cómo se lleva a cabo el cambio de contexto entre procesos. En dicho esquema se muestra cómo en un momento dado, durante la ejecución del proceso 'A', se produce una interrupción hardware, o bien éste realiza una llamada al sistema, la cual se realiza mediante la ejecución de una interrupción software. En ambos casos el flujo de ejecución se transfiere a la rutina de tratamiento de interrupción asociada, apilándose la dirección de la siguiente instrucción al momento en el que se produjo el evento, y el conjunto de los registros del procesador, para preservar el estado de ejecución del proceso. Esto se lleva a cabo de manera implícita en el código generado por el compilador Turbo C. A partir de aquí hay que realizar de forma explícita en el código lo necesario para llevar a cabo los cambios de contexto entre procesos. En primer lugar hay que salvar el puntero de pila en el descriptor de proceso interrumpido, ya que a continuación se va a establecer una nueva pila dentro del espacio de 'SO' (ya que más adelante, cuando se vaya a reanudar el proceso, habrá que recuperar dicha pila). Hecho esto, se puede llevar a cabo la ejecución del servicio de la llamada al sistema, o en el caso de ser una interrupción hardware, el tratamiento asociado a la misma. Durante la ejecución de la llama-

da al sistema es posible que el proceso se deba quedar bloqueado, por ejemplo si la llamada es "sleep()", o 'leerTecla()", o también podría ser un fin de proceso, teniendo en todos estos casos la necesidad de activar otro proceso. Igualmente, si se hubiera tratado de una interrupción hardware, durante el tratamiento de la misma podría haber cambiado el estado de un proceso, pasando de bloqueado a preparado. Además éste proceso podría ser prioritario, por ejemplo, si se pulsa un tecla por la que espera un proceso, podría interesar activar dicho proceso sin dilación para mejorar el tiempo de respuesta, algo deseable en sistemas interactivos. Otra posible situación podría ser una interrupción de reloj que hiciese que el "quantum" o rodaja, finalizase, en cuyo caso también habría que activar otro proceso. Cuando cualquiera de estas cosas sucede, para activar otro proceso, hay que recuperar el puntero de pila de su descriptor y seguidamente recuperar los registros del procesador que deben estar guardados en su pila y retornar al punto donde se produjo la última interrupción (o inicio de programa si es la primera vez).

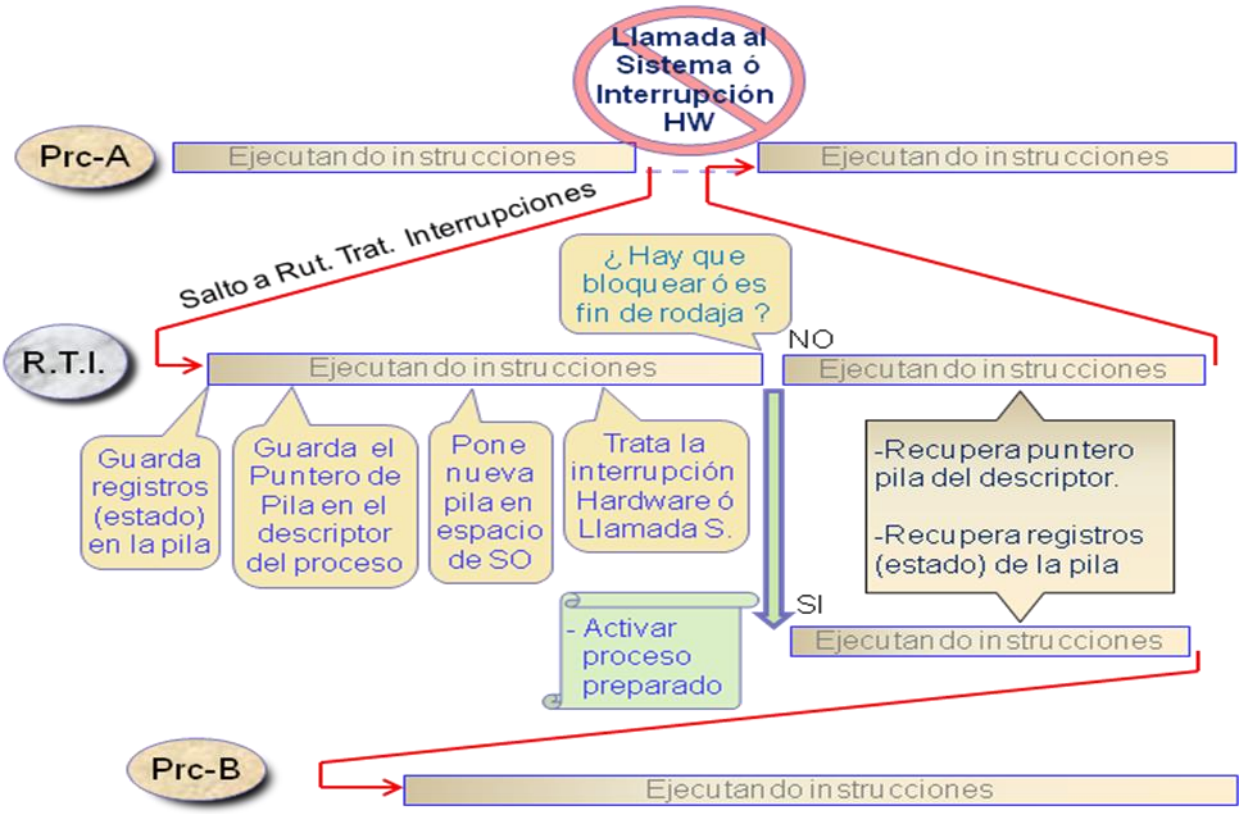


Figura 14: Esquema de cambio de contexto entre procesos

5.3.2 Gestión de Memoria:

La gestión de memoria esta implementada en el módulo de proyecto "memoria.c", y como es habitual, las constantes, tipos de datos, declaración de variables externas y prototipos de función

visibles a otros módulos se encuentran en el fichero "memoria.h", en el cual únicamente se define la constante "*CHUNK*" para establecer la cantidad de memoria que se le da a todo proceso para espacio (*'gap')* entre el área de datos y la pila. También se declaran las variables '***iniHeap***' y '***finHeap***', que contienen las direcciones (en "*paragraphs*") de comienzo y fin del "hueco" que inicialmente constituye toda la memoria disponible. Por último se declaran los prototipos de las funciones que exporta el módulo, la cuales son:

- *memBios().* Memoria de la que dispone el sistema según la BIOS.
- *inicMemoria ().* Inicializa los datos del módulo.
- *TomaMem* (tam). Asigna memoria.
- *SueltaMem* (dir, tam). Devuelve memoria.
- *incPtr* (ptr, v). Suma el valor 'v' al puntero 'ptr' evitando desbordamiento.
- *esNULL* (ptr). Comprueba exhaustivamente si 'ptr' es NULL
- *volcar* (ptr, nBytes). Muestra en pantalla 'nBytes' a partir de la dirección 'ptr'.
- *mostrarMemoria* (). Muestra en pantalla la lista de huecos de memoria.
- *compactaMem* (). Compacta la memoria.

De las funciones arriba señaladas las más interesantes a comentar son: **TomaMem()**, **SueltaMem()** y **compactaMem()**.

En primer lugar hay que decir que las unidades de memoria que se asignan y liberan son los "***paragraphs***" ("*clics*" en Minix), de tamaño igual a 16 bytes, pudiéndose entender por ello, que tanto los parámetros de entrada como los de salida sean del tipo "*word_t*" (el cual ocupa 2 bytes). Otro término que conviene aclarar es el de "***hueco***", usado aquí ampliamente para designar un bloque de memoria libre (o disponible).

En el fichero "memoria.c" se declara una estructura de datos para implementar los nodos de a lista de "huecos" llamada "*foo*" (ya que hay que poner algo, y este nombre no se usa nunca), compuesta por dos campos , "tam" y "uSig" que contendrán el tamaño y dirección del siguiente hueco de la lista. Sobre este tipo se declara el tipo "pHue_t" que es un puntero a dato del tipo anterior, que es el que realmente se usa. También existe declarada un tipo "unión" para facilitar la visión del dato "dirección de hueco" que a veces interesa tratarlo como un entero "w*ord_t*" y otras como un puntero especial de Turbo C (*_seg* *), el cual se caracteriza por ocupar sólo dos bytes, pero se considera prácticamente a todos los efectos un puntero largo (el cual ocuparía 4

bytes), en el que la parte "offset" de la dirección siempre es vale 0. Por último hay que decir que existen dos variables de ámbito privado a este módulo que son, "priHue" y "memDisponible", las cuales guardan respectivamente la dirección del primer hueco y la cantidad de memoria disponible inicialmente.

La función ***TomaMem()*** recorre la lista de huecos buscando uno de tamaño mayor o igual al solicitado. Si lo encontrase devolvería su dirección base, dejando la memoria sobrante, si la hubiere, en el propio hueco. Si el tamaño del hueco encontrado fuese exactamente el solicitado, el hueco se eliminaría de la lista. Si por otro lado tras el recorrido, no se hubiese encontrado un hueco de suficiente tamaño se devolvería 0.

A continuación se muestra un ejemplo de mapa de memoria en SO. En él se han mostrado unos bloques de memoria asignados para tablas FAT del sistema de ficheros de las unidades 0, 1 y 2, tres procesos y dos ventanas terminal. El resto de la memoria disponible está fragmentada en tres bloques o huecos.

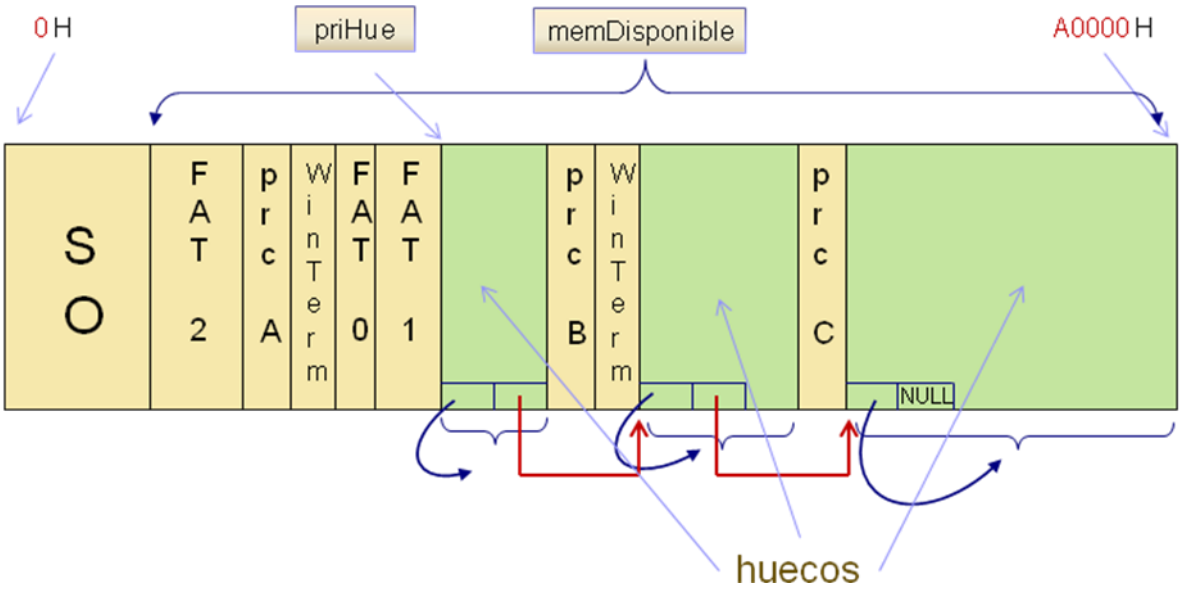


Figura 14: Esquema de cambio de contexto entre procesos

Algunos de los bloques asignados, tales como los de las tablas FAT y las ventanas terminal de los procesos se comentarán más ampliamente más adelante, en los módulos correspondientes de "ficheros" y "ventanas".

La función ***SueltaMem()*** se encarga de devolver un bloque previamente asignado, al conjunto de la memoria disponible. Al realizarse la devolución pueden producirse varias situaciones respecto a las posiciones de los huecos ya existentes. Si el bloque a devolver no tuviera ningún hueco por delante ni por detrás, entonces se produciría un nuevo hueco que habría que insertar en la lista, en una posición acorde a la dirección de memoria que ocupa el bloque, ya que la lista de bloques ha de estar siempre en orden ascendente de direcciones. Si por otro lado, el bloque a devolver tuviera algún hueco por delante o por detrás, entonces el bloque devuelto ha de fusionarse con el existente, no generándose en este caso ningún hueco nuevo, sino que sólo se modificaría la dirección o tamaño del bloque según fuera el caso. Si por último, el bloque a devolver se encontrara justo en medio de dos huecos, entonces se habría de fusionar con ellos y en este caso, tras la fusión habría desaparecido un hueco de la lista de huecos previa, ya que los dos huecos adyacentes y el liberado formarían un solo bloque o hueco, el cual quedaría registrado como tal en la lista resultante.

La función ***compactaMem()*** se encarga de agrupar todos los huecos en uno sólo, eliminando con ello la fragmentación de memoria. Para poder realizar este trabajo es condición necesaria que los objetos ubicados en memoria puedan reubicarse dinámicamente, es decir, mientras se ejecuta el sistema operativo. La reubicación de algunos objetos no presenta grandes problemas, ya que generalmente pueden ubicarse y funcionar en cualquier posición de memoria, siendo sólo preciso conocer ésta, y en caso de que cambie, podría ser suficiente el informar de la nueva ubicación o modificar en las estructuras de datos del sistema donde se guarde su posición para proceder a actualizarla. En este tipo de objetos podemos clasificar las "***ventanas terminal***" de los procesos y las "***tablas FAT***" de los sistemas de ficheros. Cuando estos objetos se han de reubicar basta con que el sistema modifique, en el caso de las tablas FAT, las variables que el sistema mantiene sobre las unidades de disco (en las que se guarda la dirección de memoria donde se encuentra la tabla FAT de cada unidad, y que se verá en detalle más adelante en el módulo "ficheros), y en el caso de las "*ventanas terminal*", modificar los descriptores de los procesos que usen dichas ventanas (pues en ellos se guarda la dirección de memoria de la ventana terminal del proceso, y por tanto han de reflejar la nueva ubicación), y los punteros que se usan en la lista de ventanas que mantiene el sistema para organizar la posición "**3D**" en pantalla de las ventanas (esta lista y organización se verá más adelante en detalle en el módulo de "ventanas"). Hay por otro lado otros objetos, como son los "***procesos***", los cuales pueden ser muy sensibles a su ubicación. El código de los mismos podría ser tal que contuviera direcciones estáticas ligadas a la dirección inicial de carga del proceso. Si esto fuera así, no sería fácil su reubicación, por tanto para poder reubicar

fácilmente los procesos, estos se han diseñado para que no contengan direcciones estáticas, por tanto, todas las direcciones son reubicables, es decir no están ligadas a la dirección de carga del proceso. Cuando un proceso cambia de sitio sólo se requiere modificar el descriptor del mismo, algunos registros de segmento, los cuales especifican segmentos de memoria donde se encuentra el proceso, y por último la trama de registros guardada en pila que forma parte del contexto del proceso.

A continuación se va describir a grandes rasgos el método seguido durante el proceso de compactación:

En primer lugar hay que recorrer la tabla que el SO mantiene con las unidades de disco para ir añadiendo en una tabla temporal de objetos a reubicar, inicialmente vacía, cada una de las tablas FAT de dichos discos. Sólo nos interesa la dirección y tamaño del objeto. La inserción en la tabla temporal de objetos se hace en orden ascendente de direcciones y se guarda también qué tipo de objeto es (FAT, ventana, o proceso) y un identificador del mismo (para poder identificar la unidad, y el proceso), aunque en este primer caso, obviamente los objetos incorporados serán todos del tipo FAT. A continuación se recorre la tabla de procesos, incorporando igualmente a la tabla temporal de objetos, los objetos que constituyen los bloques de memoria donde se encuentran ubicados los procesos. También se aprovecha cada iteración en la que se accede al descriptor de un proceso, para incorporar a esta tabla el objeto ventana terminal de dicho proceso, salvo que éste hubiese sido incorporado con anterioridad a la tabla temporal, por ser una ventana terminal compartida con otro proceso.

Una vez finalizado el relleno de la tabla temporal, y con ésta perfectamente ordenada por direcciones crecientes de memoria, hay que recorrer esta tabla desde el primer elemento al último para ir reubicando cada uno de los objetos, a la vez que se modifican las estructuras de datos que el sistema mantiene sobre ellos. Por cada objeto de la tabla se hace lo siguiente:

> Se compara la dirección de memoria donde se encuentra el objeto con una dirección "base" establecida al principio, con la dirección del primer hueco de la lista. Si la dirección del objeto es menor que dicha "base" se pasa al siguiente objeto, si es mayor, se copia (reubica) el objeto a la dirección "base", se modifican todas las estructuras de datos que el sistema tiene sobre el objeto y se incrementa la dirección "base", tantos *paragraphs* como tenga el objeto reubicado. Si el objeto es una FAT, la actualización de las estructuras de datos sólo requiere modificar la tabla informativa de discos (***infDrv[d].pFat***). Si el objeto es un proceso hay que actualizar va-

rias cosas: el campo del descriptor del proceso que señala dónde se encuentra el proceso (***tblProc[n].dirMem***); el puntero de pila guardado en el descriptor del proceso (***tblProc[n].sp***); y finalmente los registros de segmento incluidos en la trama apilada en la misma. Si por último el objeto es una ventana terminal, hay que modificar el campo del descriptor de proceso que contiene la dirección de la ventana terminal (***tblProc [n]. pWin***), de todos los procesos que usen dicha ventana terminal y por otro lado hay que retocar los punteros (***pWUp*** *y* ***pWdw***) que forman parte de la lista de ventanas que mantiene el SO sobre el orden "**3D**" que ocupan éstas en pantalla. Esta es quizá la parte más compleja de la implementación.

Una vez finalizado el recorrido de la tabla temporal de objetos, reubicados todos y ellos y con todas las estructuras de datos del sistema correctamente actualizadas sólo restaría actualizar la variable del sistema que señala la dirección y tamaño del único y primer hueco de memoria disponible (***priHue***).

Para acabar, también hay que señalar que durante todo el proceso de compactación las interrupciones deben estar inhibidas para evitar cambios de contexto o intentos de acceso a objetos que puedan estar en tránsito. Esto no supone un grave problema, ya que este proceso no suele durar mucho y de momento es una función reservada que sólo se puede invocar desde la consola.

5.3.3 Gestión de Entrada / Salida:

En este apartado se va describir la implementación de la entrada de teclado, el acceso a disco, el tratamiento del reloj y como colofón final, el manejo de ventanas.

ENTRADA DE TECLADO: Está repartida entre los módulos: "teclado.c/h" y "teclaint.c/h". El primero de ellos implementa los servicios de obtención de tecla y el segundo gestiona las interrupciones del teclado.

En "**teclado.h**" tenemos los prototipos de las funciones:

- *char leerTecla (void);*
- *char LeerTeclaLista (void);*
- *void LeerLinea (char far * lin, word_t size, bool mayus);*
- *int mibioskey (byte_t cmd);*

*"***leerTecla()**" es una función especial en el sentido de que no está implementada en este módulo sino en "llamadas.c", en el cual se encuentran implementados los servicios o llamadas al sistema y que veremos en otra sección. Esta función utiliza auxiliarmente LeerTeclaLista*()*. Si no hubiera ninguna tecla en el buffer de teclado del proceso, leerTecla() dejaría bloqueado al proceso. Esta es la principal razón por la que esta función, que usa la consola de SO, se encuentra implementada como una llamada al sistema.

"***LeerTeclaLista()***" comprueba si hay alguna tecla disponible en el buffer de teclado del proceso, el cual se encuentra en la ventana terminal del mismo. Si hay una tecla la extrae y la devuelve y si no la hay devuelve 0. El código de las teclas es el código ASCII que maneja la BIOS, y el 0 no se usa.

"***LeerLinea()***" obtiene caracteres del teclado hasta la pulsación de la tecla "Entrar" (**CR**, ASCII 13). Se admite la pulsación de la tecla "Retroceso" (**BS**, ASCII 8) para borrar el último carácter introducido. Tiene como límite "*size*"-1 (parámetro de entrada) y opcionalmente puede devolver los caracteres forzados a mayúsculas.

"***mibioskey()***" es una función clónica de la función de librería de turbo C "*bioskey()*". La razón de su existencia es porque la original de turbo C usa la interrupción INT 16 de la BIOS, la cual deja las interrupciones permitidas cuando se la llama, lo que no es deseable, ya que es preciso usar esta función desde el sistema con las interrupciones inhibidas. Esta función, no obstante, se beneficia del uso de la rutina de tratamiento de interrupción (**RTI**) de teclado que implementa la BIOS, así como del buffer de teclas que ésta mantiene.

En el módulo "***teclaint.h***" se encuentran los prototipos de establecer y restaurar la RTI de teclado que suplantará a la incluida en la BIOS: **redirigirIntTeclado*()*** y ***restablecerIntTeclado()***. Se requieren estos prototipos en este fichero porque estas funciones de usan en el código del módulo principal "**SO.c**".

En el modulo "***teclaint.c***" se encuentra el código de las funciones vistas arriba. Su implementación es similar al otras funciones cuyo cometido es el mismo, el de instalar y restablecer un vector de interrupción y que también existen para el reloj, las llamadas al sistema y tratamiento de excepciones; estas funciones se invocan justo en los momentos de arranque y finalización del sistema operativo. No obstante, debe hacerse una aclaración sobre el código de "*redirigirIntTeclado()*", ya que incorpora algo específico debido a una peculiaridad del entorno DOSBox y tur-

bo C. Si esta función se limitara simplemente a guardar el antiguo vector y establecer el nuevo, como es habitual, al ejecutar el S.O. desde el entorno integrado de turbo C, el depurador no funciona bien, porque el vector salvado no es el de la BIOS (o DOS) sino uno que previamente ha instalado el propio turbo C y esto crea conflictos con SO, que hace lo mismo. El caso es que para solucionar el problema, el código discierne entre dos posibles escenarios de ejecución, en el primero SO se ejecuta desde el entorno de turbo C, y en el segundo no. Si corre estando turbo C el vector que guardo como "anterior" es la dirección de la RTI que usa DOSBox v.74, y en caso contrario se aplica el tratamiento estándar.

En el fichero "**teclaint.c**" también se implementa la RTI de teclado, la cual es bastante más interesante. Esta RTI, al igual que la mayoría de ellas, comienza y acaba con la llamada a las funciones auxiliares: "***setNewStack()*** y ***restoreStack()***" (implementadas en el fichero "rticomun.c"), las cuales se encargan de establecer y restaurar una nueva pila respectivamente, dentro del espacio del S.O. La función "setNewStack()" se encarga de salvar la dirección de la pila del proceso interrumpido en su descriptor y de establecer los nuevos valores de los registros de pila **SS** y **SP**, los cuales apuntarán a una zona reservada a tal efecto dentro de SO. La otra función, "restoreStack()", simplemente restaura los valores de los registros **SS** y **SP**, a partir de los salvados previamente en el descriptor del proceso, para de este modo, poder acceder al resto de registros que forman la trama apilada del contexto del proceso, pudiendo con ello restablecer el contexto del proceso y retornar al punto donde se produjo su última interrupción. Conviene señalar algo importante relacionado con el empleo de estas funciones auxiliares, *dado que manipulan la pila*, y *las variables locales* que se declaran en las funciones *se ubican en ella*, no está permitido declarar variables locales en estas RTI's, pues dichas funciones auxiliares presuponen un marco específico de pila (sin variables locales) cuando son invocadas. Esta restricción es algo que debe controlar el/los programador(es) del sistema.

Una vez visto el preámbulo y fin de la RTI de teclado, se va a detallar el resto de la misma. En primer lugar, se invoca a la RTI original para que lleve a cabo su trabajo: gestión hardware de interrupciones, traducción de códigos de tecla "*scan*", y actualización del "*buffer"* de teclas y banderas ("*flags*") de estado de teclas de desplazamiento (*Mayus, Ctrl, Alt, etc.*). Hecho esto, se invoca (ya se puede invocar) la función "*mibioskey(1)*" para saber si hay alguna tecla disponible en el "buffer". Si no la hay, simplemente se abandona la RTI, aunque si en un futuro, el sistema implementara planificación por prioridades, este sería un buen momento para expulsar al proceso en ejecución por otro de la cola de preparados que tuviera mayor prioridad. En caso de que haya

alguna tecla disponible, se extrae mediante *mibioskey(0)*. En este punto se analiza si la tecla es especial, es decir, si se va a usar para controlar el movimiento o tamaño de una ventana. Si fuese una de esas teclas ("tab", "flechas" con o sin "mayus"…), se procedería a mover / redimensionar la ventana focal según la tecla de control pulsada y a continuación se abandonaría la RTI. Otra posible combinación de teclas de uso particular es Ctrl-C, la cual se usa para abortar programas. En este punto se detecta dicha combinación de teclas y se invoca la interrupción "**1B**", que técnicamente está reservada por la BIOS, para el evento de pulsación de la combinación de teclas Ctrl-Break. Este detalle conviene aclararlo un poco más: normalmente, la BIOS de un equipo o emulador, genera la interrupción "1B" cuando se pulsa la combinación "Ctrl-Break"; sin embargo, el emulador "DOSBox" no lo hace así. Para solventarlo, el sistema ha decidió reservar la combinación "Ctrl-C" para producir el mismo efecto, y es en este punto donde esto se lleva a cabo.

Si tras extraer una tecla del buffer mediante "*bioskey(0)*" la tecla no es especial, entonces se incorpora al buffer de la ventana focal (la que tiene el foco del teclado), a no ser que dicho buffer esté lleno, en cuyo caso se desecha dicha tecla. Tras realizar esto, y dado que en este momento se sabe que la ventana focal tiene en su buffer teclas disponibles, se comprueba si hay algún proceso bloqueado en espera de dicho evento, si es así se desbloquea dicho proceso quitándolo de la cola de procesos de la ventana terminal y aquí se ofrecen varias posibilidades: a) Poner el proceso recién despertado en al final de la cola de preparados, b) Ponerlo al principio de la cola de preparados y c) ponerlo en ejecución directamente expulsando al proceso actual en ejecución que se iría a preparados, bien a la primera posición, o a la última. La elección de alguna de estas opciones se hará en función del grado de interactividad que se desee tener con las aplicaciones que esperan por el teclado. En la versión actual se ha escogido la opción "c".

ACCESO A DISCO: Actualmente este código se encuentra incluido en el módulo de ficheros, fundamentalmente debido a su relativa sencillez. La única función a tener en cuenta cuyo prototipo se encuentra en "**ficheros.h**" es:

- *int leeSector (word_t sect, drv_t drv, void far *pBuf);*

El código de la misma se encuentra en "**ficheros.c**". Esta función lee un sector de 512 bytes (*sect*) de una unidad de disco (*drv*), y deja la información en la dirección (*pBuf*) que se le indica. La función se limita a llamar a otra más general llamada "leeEscr()", que es la que lleva a cabo el

trabajo. Esta segunda función admite un parámetro más (*cmd*) que indica si se desea leer o escribir el sector. La implementación no es complicada pues se apoya en la función de servicio de la BIOS **INT 16**, la cual es en realidad quien realiza todo trabajo de programación de la controladora de disquete y/o de disco duro. Básicamente lo que esta función hace es traducir el número de sector lógico a coordenadas físicas del disco: cabeza; pista y sector, que a la postre, son los parámetros con los trabaja la **INT 16**. Hay que señalar un detalle que debe tener en cuenta esta función; y es: si la unidad de disco es un disquete o es un disco duro. Si es un disco duro, se realiza un desplazamiento (suma) en el número de sector pedido para saltarse los 63 primeros sectores, los cuales suelen estar libres, ya que forman un hueco entre el sector maestro de arranque (*master boot record*) y el sector de arranque de la partición (boot sector). Otro detalle a tener en cuenta resulta del hecho de que la función **INT 16** de la BIOS activa las interrupciones, y éstas quedan permitidas al salir de ella. Si se desea evitar esto, se pueden enmascarar las interrupciones de reloj y teclado, para evitar posibles interrupciones no deseables durante el servicio. Esta solución se adoptó inicialmente, sin embargo, en la versión actual se han vuelto a dejar permitidas, pues se ha controlado de otro modo esta posibilidad; concretamente, cuando se añadió al sistema el proceso "Servidor", el cual se encarga en exclusiva de usar esta función. Gracias a este proceso se evita la concurrencia y las posibles condiciones de carrera.

Por último, hay que añadir un último detalle sobre el funcionamiento de la BIOS y la unidad de disquetes. En un principio, el hardware (controladora DMA) de los equipos sufría la restricción de no poder efectuar lecturas de sector, sobre posiciones de memoria que traspasasen una frontera de 64KB, y era responsabilidad del programador el evitar llamar a la **INT 16,** con una de esas direcciones de memoria. En esta función este problema todavía no está resuelto, ya que al estar destinado principalmente a ejecutarse bajo el emulador "DOSBox", y carecer éste, de dicho problema, se ha considerado innecesario introducir esta complicación en el código; no obstante, si se deseara portar el sistema a otros emuladores o equipos reales, habría que efectuar un parche en esta función; algo que en principio sería bastante sencillo; bastaría con efectuar siempre todas las lecturas de sector en un buffer del sistema ubicado en una dirección correcta y luego copiar los datos desde dicho buffer a la dirección pasada en la llamada.

TRATAMIENTO DEL RELOJ: Toda la programación relativa a reloj se encuentra en el módulo "*timerint.c/h*", en el cual se encuentra la rutina de tratamiento de interrupciones que se encarga de gestionar el tiempo transcurrido y fundamentalmente, dar soporte al cambio de contexto de procesos (expulsión) cuya rodaja de tiempo a vencido. En el fichero ".h" sólo se encuentran los

prototipos de las funciones que se encargan de instalar y restaurar los vectores de interrupción del reloj (**INT 8**). En el fichero ".c" está le código de estas funciones junto con el código de la rutina de tratamiento de interrupción del reloj. Esta RTI es invocada por el hardware de reloj cada 55 mseg. aproximadamente y se llevan a cabo las siguientes acciones:

- Se Invoca a la antigua RTI (la de la BIOS/DOS), para que pueda seguir llevando a cabo el recuento de "*tics*" en las variables de la BIOS (controlar el tiempo).
- Se muestra en la esquina superior derecha un símbolo que denota el progreso del sistema, así como también el proceso en ejecución y el contador de rodajas. Todo esto es opcional, sólo tiene una finalidad ilustrativa y/o en ocasiones, tras retocar el código, funciones de ayuda en la depuración de código.
- Se desactiva el altavoz si transcurrió el tiempo de duración del último pitido, si éste estaba activo. Esta acción es necesaria porque el típico pitido que se emite en determinadas ocasiones para indicar algún suceso, como por ejemplo una pulsación indebida de teclado, debe dejar de sonar, ya que en el momento en el que el sistema activa el altavoz, este empieza a sonar y se deja en ese estado, ya que no es conveniente efectuar una espera activa hasta la finalización del pitido. En su lugar lo que se hace es inicializar un contador con la duración del tiempo de pitido y proseguir con el código, dejando a la RTI de reloj encargada de la tarea de decrementar este contador con cada "*tic*" y desactivar el altavoz cuando finalmente llegue a 0.
- Se decrementa el lapso (tiempo restante para despertar) de los procesos dormidos, si los hay, y se comprueba si dicho lapso vale 0, en cuyo caso se despierta al proceso, quitándolo de la cola de dormidos y poniéndolo en la cola de preparados.
- Por último, se comprueba si en contador de "*tics*" por rodaja ha llegado al máximo, lo que significaría que habría que expulsar al proceso en ejecución. Esto se hace poniéndolo en la cola de preparados, y llamando la función activarPreparado(). Esta última función no retorna, ya que en ella se selecciona el primer proceso de la cola de preparados, se restaura su pila y se retorna a la siguiente instrucción donde se produjo el último cambio de contexto de dicho proceso. Si por otra parte no se hubiera alcanzado el límite de "*tics*" de la rodaja de tiempo, simplemente la RTI restauraría la pila y finalizaría.

MANEJO DE VENTANAS: Este sistema implementa algo que podríamos llamar "terminales virtuales" en la forma de "ventanas", las cuales sirven para que los procesos puedan mostrar información a la vez que permiten focalizar el teclado, de tal modo que cuando una ventana se

encuentra en primer plano, las pulsaciones de teclado en ese momento se dirijan a ella y quedan almacenadas en su "*buffer*", del cual extraerán teclas las aplicaciones cuando las soliciten al sistema.

La implementación se encuentra en los ficheros "windows.c/h". Encontrándose en el fichero ".h" los prototipos de las funciones que se usan en otros módulos, así como también la declaración de algunos tipos y constantes que también se requieren. Entre las constantes se encuentra el tamaño del buffer de teclas de las ventanas, y en forma de tipo enumerado, los atributos de color de ventanas y caracteres. El tipo más relevante es el puntero a ventana "*pWint_t*", que se apoya la estructura "*win_t*", la cual consta de los campos necesarios para definir las propiedades de una ventana. Por citar algunos de ellos: área (*plano*) para guardar la información visual, posición de la ventana relativa a la pantalla (*eSI. eID*), cursor, buffer de teclas (*keyBuf*), cola de procesos en espera de tecla, punteros a ventanas por encima y debajo, etc. Entre las variables a exportar a otros módulos tenemos: *bellTime*, que indica los tics de reloj que restan para apagar un posible pitido que está sonando. *videoDirecto*, es una variable booleana que sirve para conmutar la salida de la función printCar(), de la ventana del proceso en ejecución a la pantalla (fondo de escritorio) y viceversa. Esto se usa cuando el sistema quiere emitir un mensaje sin usar la ventana de ningún proceso, usando entonces la pantalla en forma directa. *curY*, *curX*, indican la posición de cursor de pantalla donde se realizará la salida cuando se usa videoDirecto. Por último, *pWinTop* indica las ventana que se encuentra en primer plano y *pWinFocal* la que tiene el foco del teclado.

Las funciones que se usan en otros módulos son:

- PrintXXX(). Sirven para visualizar información con diferentes formatos por la ventana del proceso en ejecución.
- inicWindows(). Inicializa el módulo. Determina el modo, color o monocromo y limpia la pantalla.
- initWin(). Inicializa una ventana situándola en unas coordenadas de pantalla.
- destroyWin(). Elimina una ventana liberando sus recursos.
- moverWin(). Mueve una ventana que se le indica con arreglo a un incremento/decremento (variación) de coordenadas que se le pasan.
- MoverWindow(). Mueve la ventana del proceso en ejecución a las coordenadas que se le pasan. (la usa una llamada al sistema)
- ColorWindow(). Cambia el atributo de color de una ventana.

- setWinFocal(). Establece la ventana con el foco del teclado.
- rotaWin(). Rota la lista de ventanas en un sentido y otro. Más adelante se explica con más detalle la disposición de las ventanas.
- listaWin(). Es una función que usa la consola de SO para mostrar la lista de ventanas. Se usa principalmente con fines ilustrativos.

En el fichero "windows.c" se encuentra el código que implementa las ventanas. Este código es relativamente complejo y sólo se explicará a grandes rasgos cómo se ha realizado la implementación, dejando los detalles a la consulta de los fuentes del mismo. En primer lugar hay que explicar que el sistema de ventanas utiliza un modelo "3D" en el cual las ventanas se proyectan en la pantalla quedando unas ocultas parcial o totalmente por otras, tal que podemos referirnos a la ventana que se halla en primer plano o la que se encuentra al fondo, e igualmente a cualquier posición intermedia entre éstas. El fondo está constituido por un relleno total de la pantalla con una celda concreta ('▒'), formada por un carácter y atributo de color. Esta celda puede modificarse fácilmente en el código según las preferencias del programador. A continuación se muestra una imagen con varias ventanas y el fondo de pantalla:

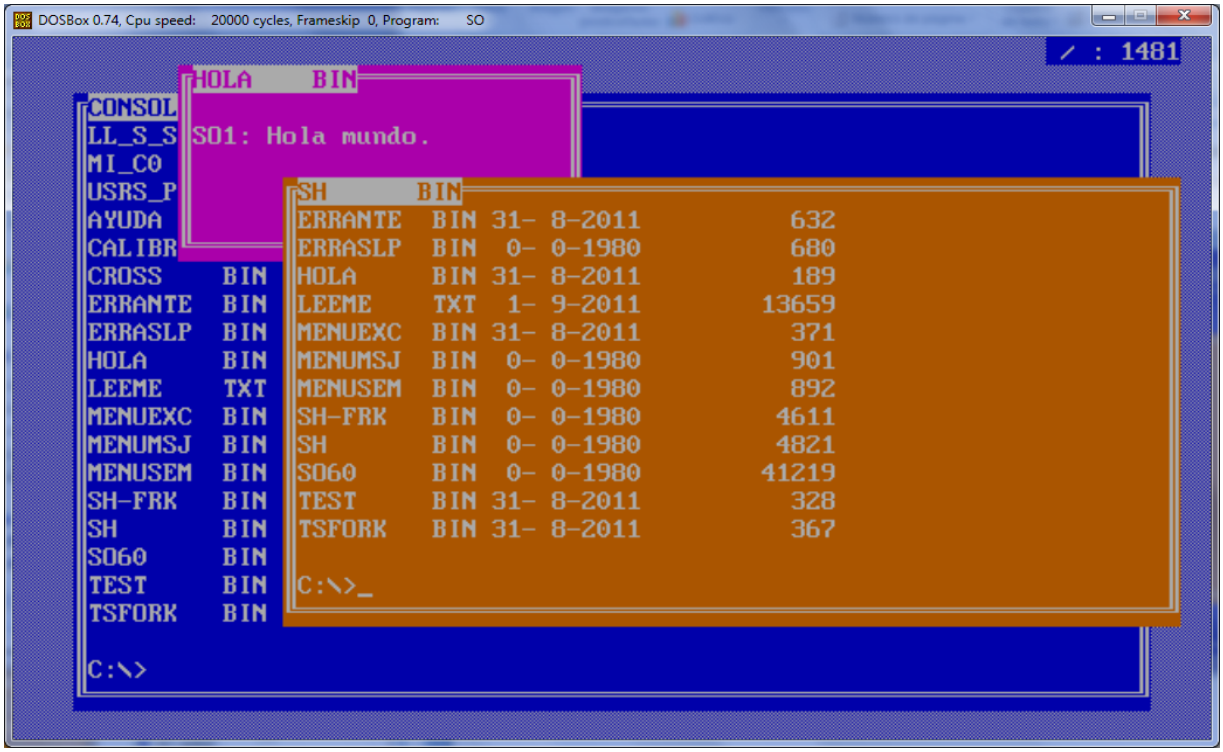


Figura 15: Pantalla con varias ventanas.

Las ventanas se encuentran formando parte de una lista circular doblemente enlazada, donde "*pWinTop*" es una variable global que apunta a la ventana que se encuentra en primer plano. To-

das las ventanas tienen dos punteros que, en general, señalan a la ventana de arriba y abajo, sin embargo en el caso de la ventana del fondo, el puntero que señala abajo en realidad apunta a la ventana de primer plano, y similarmente, en el caso de la ventana de primer plano, el puntero que señala a la de arriba, en realidad apunta a la del fondo, con ello se consigue la circularidad de la lista. Existen varias funciones que sirven para actuar sobre la lista de ventanas, entre estas se encuentran: *insTopLis() y insBottomLis()*, que inserta una ventana por la cima y fondo respectivamente, *delLis()*, que elimina una ventana de la lista y *pWinBottom()*, que retorna la ventana que se encuentra en el fondo.

Las ventanas tienen una propiedad llamada "*plano*" que se usa para guardar la información a visualizar. Este "plano" consta de una matriz de 25x80 elementos (celdas o posiciones de pantalla), y cada uno de estos elementos a su vez de un byte para el color y otro para el carácter. Cuando se hace "*print*" en una ventana, lo que se hace es guardar el(los) carácter(es) con su color en esa matriz, justo en la posición que indica el cursor de la ventana; además si la(s) posición(es) de la celda de esa ventana no está(n) oculta(s) por ninguna ventana que se encuentre encima, el(los) carácter(es) se mostrará(n) en pantalla. A continuación se muestra una figura con varias ventanas y la lista "3D" que las representa:

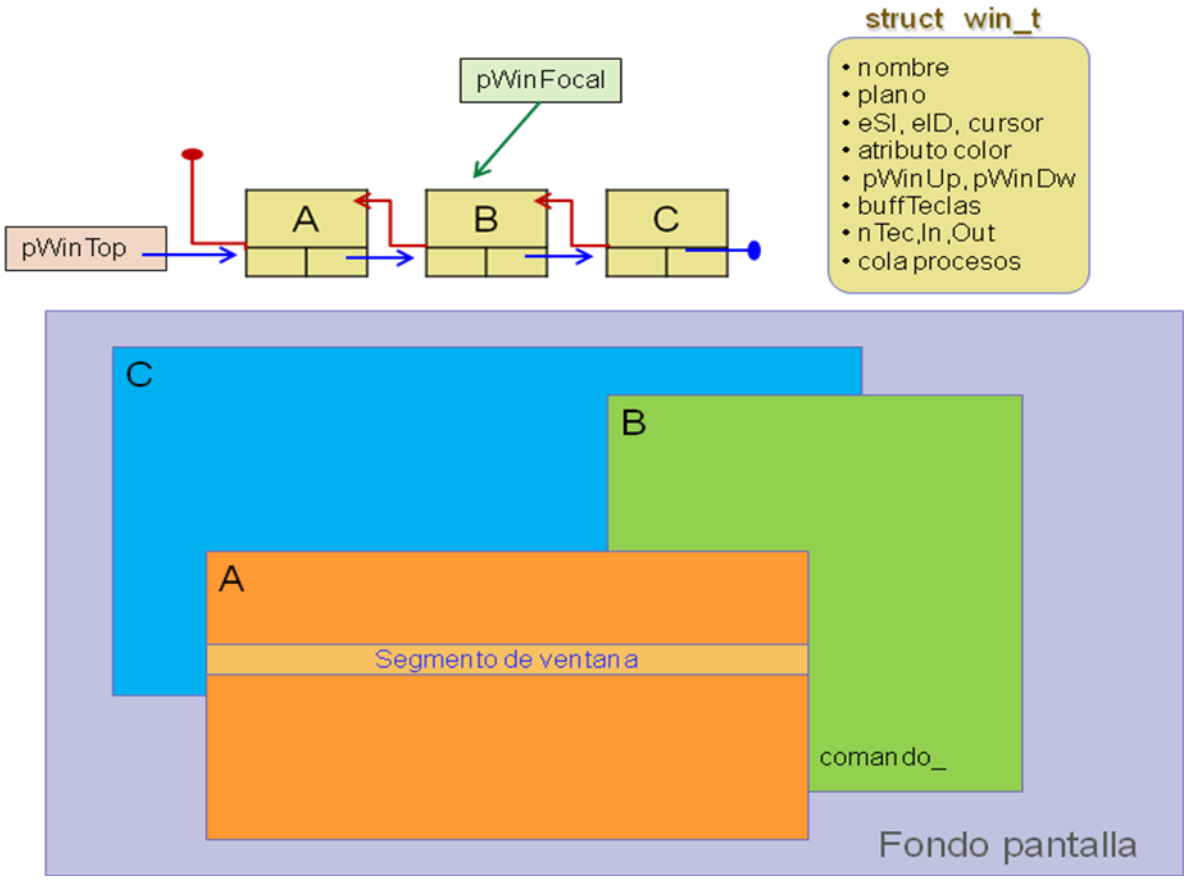


Figura 16: Lista "3D" de ventanas

Un aspecto bastante relevante sobre las ventanas es el cómo se ha resuelto precisamente el que una posición concreta de una ventana deba mostrarse o no. También cuando se ha de repintar una ventana surge dicho problema. La solución adoptada pasa por utilizar un concepto que se ha denominado "*proyección de segmentos*". Este consiste básicamente en utilizar "segmentos", los cuales tienen un atributo de posición, dado por las coordenadas "x, y", de su punto situado más a la izquierda, y de la longitud del segmento. Una ventana siempre puede descomponerse en varios segmentos, y estos segmentos se pueden proyectar en "3D" hacia el fondo de la pantalla o hacia arriba.

Durante la proyección de un segmento, que es un proceso iterativo, el segmento interseca con las ventanas que se va encontrando y se van produciendo como resultado de la intersección nuevos segmentos, que pueden ser: segmentos a la izquierda de la ventana (***s1***), a la derecha (***s2***), o interiores (***si***). Dependiendo de si el sentido de la proyección es hacia arriba o hacia abajo, los segmentos resultantes continúan o no su proyección de una forma recursiva, para finalmente determinar si el segmento final se ha de mostrar en pantalla o no. La siguiente figura muestra varios ejemplos de intersección de un segmento con una ventana:

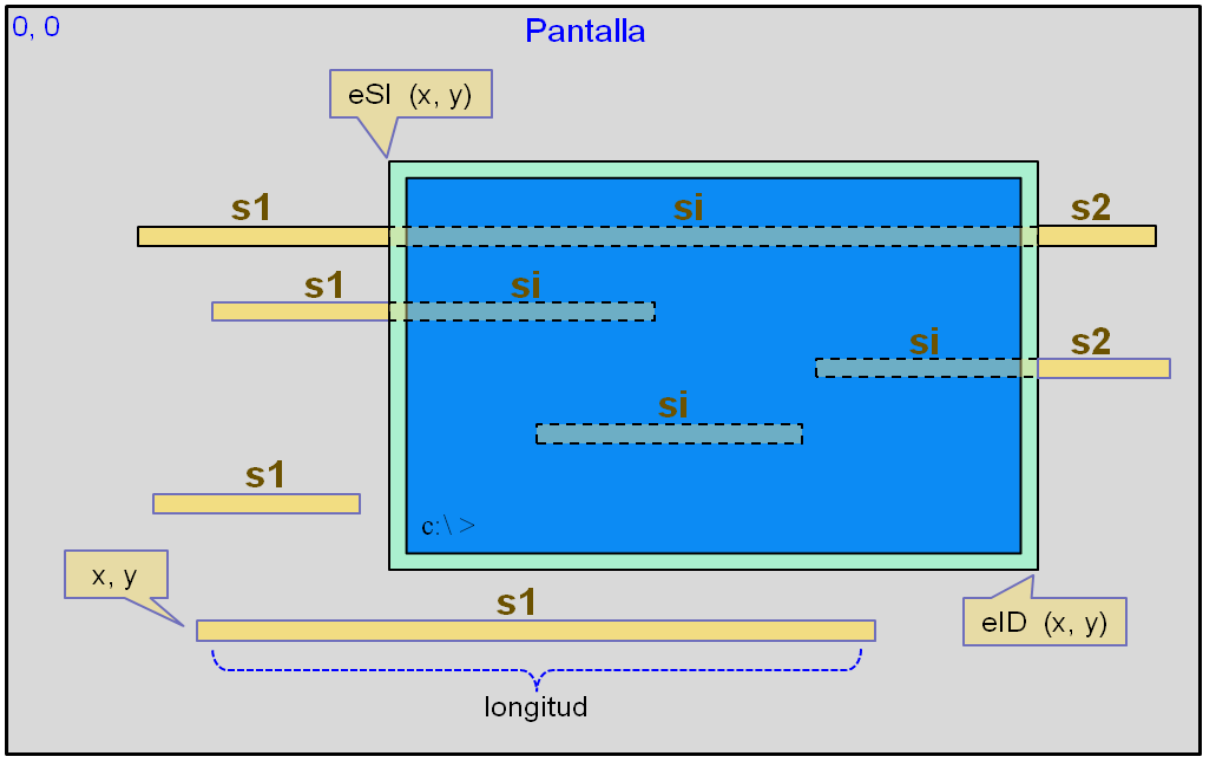


Figura 17: Varios ejemplos de intersección de segmento y ventana.

En la figura de arriba se muestra cómo un segmento al intersecar con una ventana puede producir hasta tres subsegmentos como ya se indicó antes, aunque el resultado también podría ser sólo uno; "***s1***" si no hubiera intersección, o bien "**si**", si la intersección fuera totalmente interior, o también podrían ser dos; "***s1***" *y "****si***", o "***si***" y "***s2***". La función que realiza esto es *obtenSegmentos()*, la cual es invocada repetidamente durante el proceso de proyección realizado por la función *proyectaSegmento()*, el cual a su vez invoca a *proyectaUp()* o *proyectaDown()*, dependiendo del sentido de la proyección. Para mejorar la comprensión de cómo funciona este sistema se señala la conveniencia de consultar los amplios comentarios existentes en los fuentes de este módulo.

Veamos qué se lleva a cabo cuando, por ejemplo, una ventana oculta se va a mostrar en una posición de profundidad intermedia. Lo que se hace es recorrer todas las filas de la ventana (abscisa y) y por cada una de ellas se toma el segmento que la conforma y se inicia la proyección del mismo hacia la cima. Con cada intersección de ventana se obtienen uno o más subsegmentos. Si un subsegmento es del tipo interior (***si***), significa que está oculto y por tanto no se muestra, acabando el proceso recursivo para él, si por otro lado, los subsegmentos son "***s1***" o "***s2***", entonces proseguirán su proyección hacia arriba, efectuándose todo esto mediante recursión. Si al final del proceso recursivo un subsegmento alcanza la cima, entonces se muestra en pantalla. Veamos ahora el caso en el que lo que se quiere hacer es ocultar una ventana que se encuentra en cualquier posición de profundidad. Este caso es algo más complejo pero se resuelve de un modo bastante similar. En este caso los segmentos en los que se descompone la ventana inicialmente inician su proyección hacia el fondo. Los subsegmentos interiores "***si***" son candidatos a ser mostrados, pero no el formado con la ventana a ocultar, si no uno nuevo que se forma con la ventana de la intersección. Para comprobar si se debe mostrar dicho segmento, ahora habría que comenzar un proceso análogo al visto anteriormente, es decir habría que proyectarlo hacia la cima para determinar que partes se deben mostrar o no. Volviendo a los posibles segmentos "***s1***" y "***s2***", estos deben proseguir su proyección hacia abajo para seguir comprobando si hay alguna ventana debajo de ellos. La proyección finaliza bien cuando alcanzan una ventana y se transforman en un subsegmento "***si***" (candidato a ser mostrado) e inician el proceso inverso ya visto, o bien cuando alcanzan el fondo de la pantalla, en cuyo caso también se inicia dicho proceso de proyección hacia arriba, pero el segmento a mostrar en caso de que se alcance la cima sería uno formado por los caracteres de relleno de fondo de escritorio. Todo lo visto constituye a grandes rasgos el método usado para la gestión de ventanas en "3D". Este método podría ser mejorado de varias formas, una de ellas podría consistir en usar también segmentos verticales, realizando una trans-

posición de coordenadas, con ello se podría disminuir el número de segmentos a proyectar en ventanas que fueran más altas que anchas. Otra mejora quizá aún superior sería utilizar intersecciones de ventana con ventana, que aunque pueda ser algo más complejo el resultado de las mismas, se mejoraría mucho la velocidad, aunque cuando se usan ventanas no gráficas, la velocidad actual es más que suficiente.

Pasemos a otro detalle importante con respecto a las coordenadas que se usan. El origen de éstas es x=0, y=0, y está situado en la esquina superior izquierda de la pantalla. Todas las coordenadas con las que se trabaja son relativas a este origen. Las coordenadas de las esquinas de la ventana "eSI" y "eID" no incluyen el marco. Para acceder a la información visual contenida en el plano partiendo de las coordenadas de la posición a mostrar, hay que efectuar una operación matemática en la que se suma a las coordenadas de la posición, el origen de la esquina superior izquierda de la ventana, el cual puede considerarse como el desplazamiento que experimenta la ventana cuando ésta se mueve. Esto es así porque la posición de un punto en la ventana cambia cuando ésta se mueve, en cambio su información visual, contenida en el plano, no. La operación matemática es la siguiente: *pWin->plano [y-pWin->eSI.y+1][x-pWin->eSIx+1].car = car*. Con este sistema, mover una ventana por la pantalla no requiere mover la información visual contenida en el plano, simplemente se cambian los valores de la esquina superior izquierda de la ventana.

Por último hay que añadir una propiedad de la ventana que no tiene que ver con la visualización. Consiste en un buffer de teclas y una cola de procesos en espera de tecla. El "*buffer*", que es circular, almacena las teclas que el usuario pulsa y que el sistema redirige a la ventana focal. Cuando un proceso pide una tecla al sistema, el sistema comprueba si hay alguna tecla disponible en dicho "*buffer*" y se la entrega de ser así, en caso contrario deja bloqueado al proceso por "`BLK_TECLADO`", y lo mete en la cola de procesos de su ventana.

5.3.4 Gestión de ficheros:

La gestión de ficheros se encuentra implementada íntegramente en "***ficheros.c y .h***", y únicamente contempla el sistema de ficheros **FAT** (variantes 12 y 16 bits**)** usado principalmente por sistema operativo MS-DOS. La razón por la que se ha usado este sistema es su sencillez y amplia difusión. En el fichero de cabeceras ".h" están definidas algunas de las estructuras de datos que se requieren para implementar la gestión de ficheros. La primera de ellas tiene que ver con la

información que el sistema debe guardar sobre una unidad de disco. En la estructura "*infDrv[]*" se recogen datos como el número de cabezas, sectores y pistas que tiene el disco, número de sectores por *cluster*, el primer sector de datos, primer sector del directorio raíz, tamaño de la FAT y puntero a ella, etc.

Otra estructura importante, "*infoFich_t*", recoge la información que el sistema debe tener sobre los ficheros que mantiene abiertos. Entre sus campos se encuentran:

- Copia de la entrada del directorio donde se encuentra el fichero (*entr*).
- Número de sector lógico donde se encuentra ésta (*nSect*) y
- Posición relativa dentro de dicho sector (*nEntr*).
- Unidad de disco donde se ubica el fichero (*drv*).
- Contador de referencias (*nRefs*) (indica cuantas veces está siendo referenciada esta estructura por otros objetos tales como los descriptores de fichero) y por último,
- "*flag*" (*modif*) para indicar si se ha modificado esta estructura y que por tanto ha de ser volcada a disco cuando el fichero se cierre.

Los procesos tienen que mantener cada uno de ellos una tabla con los ficheros que tienen abiertos. Cada uno de los elementos de esa tabla es un manejador de fichero (*file handle*), y generalmente suele ser un puntero o índice a una tabla de ficheros abiertos por el sistema cuyos elementos son conocidos comúnmente como descriptores de fichero, y el tipo o estructura que lo define es "*descrFich_t*", la cual tiene los siguientes campos:

- puntero a estructura informativa de fichero, vista en el párrafo anterior (*pInf*);
- posición del último byte leído o escrito (*fPos*), corrientemente conocido como "file pointer", y que se utiliza en la conocida llamada a sistema de ficheros "lseek"; y por último,
- número de veces referenciado (*nRefs*), el cual es un contador que especifica cuantas veces está siendo utilizado este descriptor por los procesos.

En la siguiente figura se muestra las diferentes tablas de datos y su relación entre ellas:

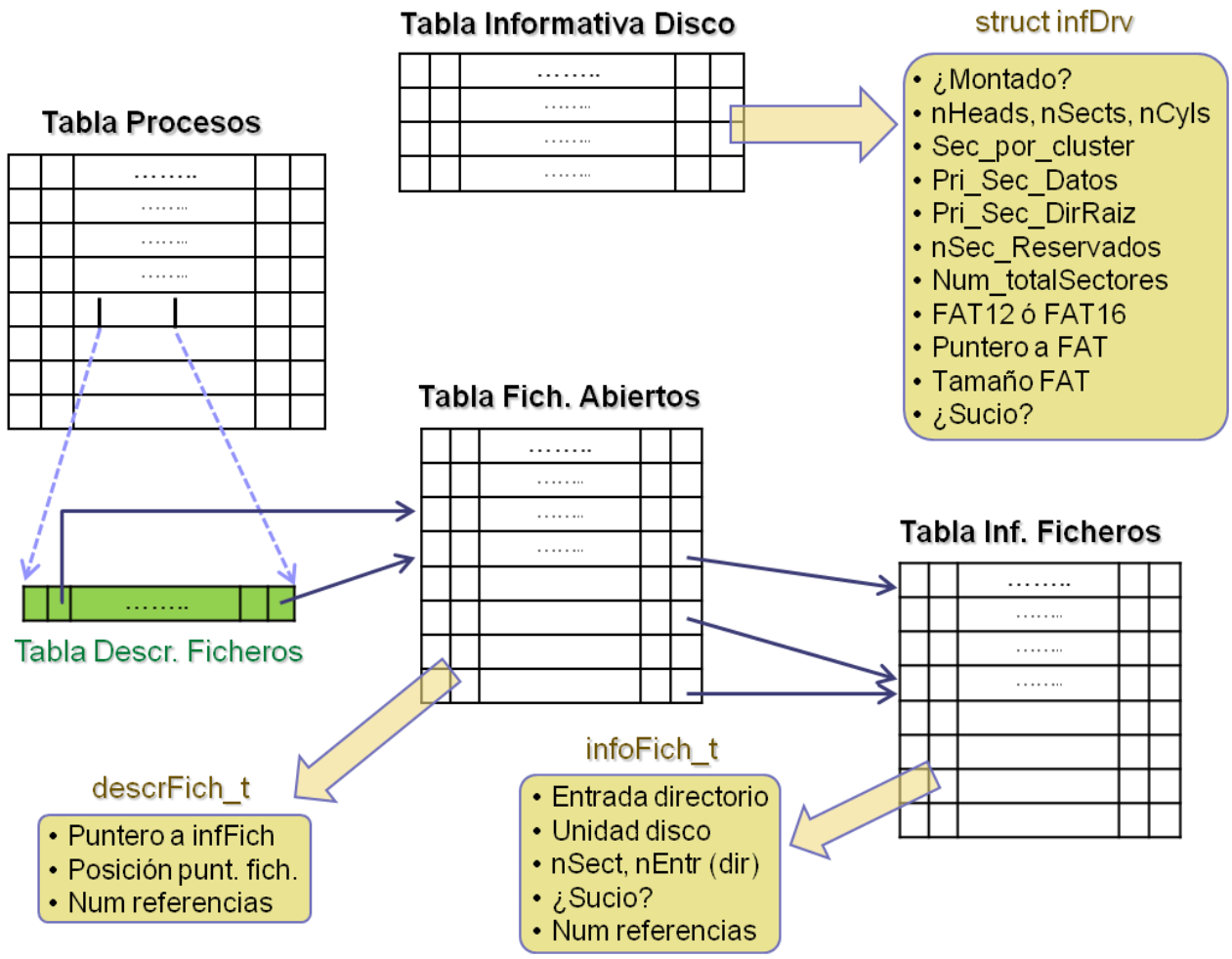


Figura 18: Principales tablas y estructuras del sistema de ficheros

Hay otra estructura de datos privada definida en "ficheros.c", que es "*boot_t*", la cual define la parte de datos que almacena un sector de arranque estándar. Una información técnica detallada se puede encontrar en el manual de Microsoft "MS-DOS Programmer's Reference", aunque de hecho esta información está muy difundida y existen múltiples documentos donde encontrar esta información.

Por otra parte, en el fichero de cabeceras también se declaran las variables y prototipos de funciones que exporta el módulo. Entre las variables tenemos la tabla de ficheros Abiertos (*tblFicAbiertos*), que lo requiere la llamada al sistema "fork", y la variable "*nprAtnd*" que indica qué proceso está siendo atendido por el proceso "Servidor", el cual como ya se dijo anteriormente, se encarga de atender las peticiones al sistema relacionadas con ficheros. Entre las funciones exportadas tenemos las que utiliza la consola: "*inicFicheros()*" y "*listarTFAbiertos()*", y las que se encargan de dar soporte a las llamadas al sistema, que son todas las demás y que veremos poste-

riormente desde la perspectiva de su propósito o funcionalidad en el capítulo de llamadas al sistema. Seguidamente se hace una descripción de las funciones usadas por la consola.

- *inicFicheros():* Inicializa el sistema de ficheros. Se limita a poner las tablas a 0. Esta función se usa durante la inicialización del sistema, en la función *main()* del fichero "so.c"
- *listarTFAbiertos():* Esta función muestra en pantalla los ficheros que el sistema mantiene abiertos. Es puramente informativa y es la encargada de llevar a cabo el comando de consola "tfa" (Tabla de Ficheros Abiertos).

Respecto a las funciones de soporte de las llamadas al sistema relacionadas con el sistema de ficheros, se evitará entrar en una descripción pormenorizada del código de cada una de ellas, en su lugar, se hará una descripción general de las partes de código más influyentes en la implementación del sistema de ficheros, el cual se encuentra como es fácil suponer en "ficheros.c".

En primer lugar, puesto que el sistema de ficheros está basado en el de MS-DOS, y éste se basa en el uso de *clusters*, es conveniente comentar un poco la función "*sigCluster()*", la cual, con la FAT cargada en memoria, devuelve el siguiente *cluster* a uno dado. Si la FAT fuese del tipo 16 bits, el cálculo sería muy sencillo, bastaría con acceder al elemento de la tabla cuyo índice fuese el parámetro de entrada, sin embargo, con una FAT del tipo 12 bits, el cálculo es más complicado. En el manual de referencia técnico de MS-DOS ***[MIC91]***, se detalla el cálculo, y esta función simplemente lo lleva a cabo. Existe también una función reciproca llamada "setSigCluster()", la cual se encarga de escribir en la FAT el valor del *cluster* siguiente a uno dado. Existen otras funciones de manejo de *clusters* como son "clus2sect(), getClusLibre(), getUltCluster() y liberaCadenaClusters()" cuyo propósito y modo de operación se puede consultar ampliamente en los comentarios incluidos en el código de dichas funciones.

Otras funciones interesantes son las de montar y desmontar una unidad de disco, "MontaDrv() y DesmontaDrv()". El montaje de una unidad consiste básicamente en cargar la información del sistema de ficheros de la unidad, en la tabla informativa de unidades: infDrv[]. Para ello, lee el sector 0, y si es válido, establece en dicha tabla valores como: la geometría del disco, el primer sector de datos, etc. También se encarga de solicitar memoria dinámica para leer la FAT del disco sobre ella, y si la unidad hubiera estado previamente montada la desmontaría antes. Desmontar la unidad consiste en liberar los recursos de la unidad, en especial el de la memoria dinámica

de la FAT, aunque previamente, comprobaría el estado del *flag* "sucio" (modificada), para en su caso, escribir los cambios en el disco.

Antes de proseguir con otras funciones hay que señalar la existencia de dos tipos de datos declarados en "fich-ifz.h" (incluido a su vez en "ficheros.h"). Estos tipos son: "entrada_t", que define la estructura de una entrada de directorio y que se utiliza ampliamente en todo el código; y el otro tipo de datos, "resulBus_t", cuya necesidad o naturaleza no es tan obvia. Este último tipo lo utiliza el usuario para poder realizar las llamadas al sistema "buscaPriEntDir()" y "buscaSigEntDir()", que se usan para iniciar, y posteriormente continuar, con la búsqueda de uno o más ficheros / directorios a partir de una ruta, la cual puede incluir comodines, y atributos de fichero. Los campos que incluye "resulBus_t" son los necesarios para poder iniciar una búsqueda de entrada de directorio, y también para poder retomar posteriormente dicha búsqueda partiendo del punto donde se quedó la última vez. Para esto último se requiere que la información persista, lo que hace gracias a esta estructura de datos. Este tipo de datos también se usa ampliamente en todo el código, porque la necesidad de buscar entradas de fichero en los directorios y proseguir la búsqueda es muy corriente, pues lo necesitan muchas funciones. Seguidamente se comentan los campos de esta estructura de datos tan importante.

- *entrsDir[]*. Tabla (*buffer*) para un sector de disco completo (512 bytes) con capacidad de hasta 32 entradas de directorio.
- *nombre[8]*, *ext[3]*. Nombre y extensión del fichero a buscar.
- atr. Atributo de fichero (oculto, sistema, sólo lectura, archivo, directorio, etc.) a usar como filtro en la búsqueda.
- *drv, nClus, nSect, nEnt*. Son los valores que permiten localizar la entrada objeto de la búsqueda. "nClus" toma en la primera búsqueda el valor del primer *cluster* del directorio, pero a medida que avanza la búsqueda, este campo se actualiza con el valor de los siguientes *clusters* que forman la cadena del directorio de búsqueda. "nSect" guarda el número de sector relativo al cluster (empezando por 0), pues como sabemos un *cluster* puede agrupar 2, 4, 8, etc. sectores. Por último, "nEnt" guarda el número de entrada relativa al sector, cuyo valor va de 0 a 31. Todos estos valores, que vienen a representar la dirección de una entrada de directorio, van actualizándose como ya se dijo a medida que la búsqueda progresa al efectuar sucesivas llamadas a la función busSigEntDir().
- secCLoDR. Este campo tiene un doble significado. A veces guarda el número de sectores por cluster y otras veces el número de sectores que ocupa el directorio raíz. La razón es

porque el directorio raíz no se trata igual que cualquier otro subdirectorio. Para empezar, el número de *cluster* de comienzo del directorio raíz es 0 por convenio, ya que este valor (junto con el 1) está reservado y no puede ser usado. Para recorrer al directorio raíz, al comprobar que es el *cluster* 0, se toma como primer sector el indicado en la información sobre la unidad de disco y se va incrementado éste hasta alcanzar el valor del campo "secCLoDR", que en este caso será el total de sectores que ocupa el directorio raíz. Los subdirectorios en cambio, se han de recorrer empezando por su primer cluster para seguir con la cadena de *clusters* guardada en la FAT. Se alcanza el final cuando la cadena así lo indica. El algoritmos que determina todo esto se encuentra en las funciones: "incEntDir()" y "setResulBus()", las cuales están bastante documentadas en el propio código.

Vistas las estructuras de datos más importantes, vamos a comentar a continuación algunas de las funciones más importantes en el código:

- *clus2ruta():* En el sistema de ficheros FAT, toda entrada de directorio contiene además del nombre del fichero/directorio, el número del primer *cluster* de la cadena de clusters que conforman el fichero/directorio. Esta función construye la ruta de un directorio a partir de este *cluster* de inicio. Este proceso es algo tedioso ya que debe ascender en la jerarquía de directorios buscando las entradas "..", que señalan al directorio padre, hasta que se alcanza finalmente el directorio raíz. Esta función es utilizada por "GetWorkDir()", que utiliza la unidad y cluster de trabajo que se guardan en el descriptor del proceso para a partir de esto y la ayuda de esta función devolver la ruta del directorio de trabajo. La función "parseRuta()" también requiere de esta función y se explica a continuación.
- *parseRuta():* Esta función "analiza" la ruta que se pasa como parámetro de entrada, que puede ser absoluta o relativa y construye una ruta absoluta sobre una variable local estática, de la cual devuelve su dirección. Otro efecto importante es que rellena la variable "*pResul" que se le pasa con la información requerida para poder realizar posteriores búsquedas o cualquier otro propósito. Esta función es bastante importante porque es la que realiza el trabajo principal de la llamada al sistema "buscaPriEntDir()", a la cual solo le resta añadir un filtro de selección de búsqueda y retornar el cluster inicial de la entrada buscada.
- *expandeEntrada()*: Cuando un fichero crece hay que ampliar su cadena de clusters. Esto sucede cuando se escribe al final de un fichero o también cuando se añaden ficheros a un directorio. Esta función facilita esta tarea permitiendo añadir 'n' *clusters* a una entrada de

directorio existente. Las funciones que llaman a esta función son "write()" y "creaEntrada()".

- *liberaCadenaClusters()*: Recorre la cadena de *clusters* a partir del cluster que se le pasa como parámetro, el cual normalmente es el primero, y va poniendo ceros en la FAT con lo que los deja marcados como libres. Esta función es usada por "truncaFichero()", "delEntrada()" y "rmDirEntradas()", que sirven para truncar el tamaño del un fichero, borrar una entrada de directorio y borrar todas las entradas de un directorio recursivamente.

- *abrirFichero()*: El S.O. mantiene una tabla en memoria con todos los ficheros con los que está trabajando en un momento dado llamada "tblFicAbiertos[]". Esta tabla guarda toda la información que necesita el sistema para poder acceder rápidamente a los datos del fichero. Esta función se encarga de rellenar esta información en esta tabla, admitiendo como parámetro de entrada la ruta con el nombre del fichero y devolviendo un número entero llamado descriptor de fichero que será utilizado en otras funciones para poder operar con el fichero. Esta función es usada por "RunFichero()", "Execv()" y "Open()" que se utilizan respectivamente, para ejecutar un programa (crear un nuevo proceso a partir de un fichero ejecutable), cambiar la imagen de memoria de un proceso por la que se encuentra en un fichero ejecutable y para dar soporte a la llamada al sistema "open()". Conviene señalar que el modulo de ficheros utiliza dos tipos de descriptor de fichero diferentes aunque están declarados del mismo tipo (*df_t*). Los descriptores (**df**) usados como índice en la tabla de ficheros abiertos (*tblFicAbiertos[df]*), del cuyo tipo es el descriptor devuelto por esta función, y los descriptores (**fd**) usados como índice para la tabla de ficheros abiertos del proceso (*tblProc[npr].tdf[fd]*), de cuyo tipo es el valor devuelto por la llamada al sistema open().

- *RunFichero()*: Pone en ejecución un fichero ejecutable. Para ello en primer lugar abre el fichero, obtiene su tamaño y solicita memoria mediante "TomaMem()". La cantidad pedida es la suma del tamaño del fichero ejecutable, el cual incluye el área de código y datos inicializados, y una cantidad fija (*CHUNK*), que está prefijada para dar cabida al área de datos no inicializados y la pila. Si tiene éxito en la obtención de la memoria, entonces carga el código ejecutable (imagen binaria del contenido del fichero) mediante la función "cargaFichero()", en la dirección de memoria previamente obtenida, y a continuación crea el proceso mediante "*creaProceso()*". Si se produce algún error en la carga del fichero, entonces se libera la memoria y retorna con un código de error. Esta función es utilizada por la consola cuando el operador teclea un comando y éste resulta ser el nombre de un fiche-

ro ejecutable. Existe otro modo de ejecutar programas que se corresponde mejor con el sistema usado en el mundo "Unix", el cual consiste en crear un proceso hijo mediante "fork()" y cambiar su imagen de memoria por la de un fichero ejecutable, lo cual se hace mediante la llamada al sistema "execv()", que se detalla a continuación.

- *Execv()*: Cambia la imagen de memoria del proceso en ejecución por la de un fichero ejecutable. Los pasos que sigue son:
  - Abrir el fichero ejecutable (si no existe retorna error),
  - Pedir memoria (si no hay suficiente memoria retorna error),
  - Cargar el fichero en la memoria solicitada (si no se consigue se retorna error y se libera la memoria que se solicitó), y por último,
  - Actualizar los campos del descriptor del proceso para que reflejen la nueva situación. Hay que cambiar: la dirección de memoria donde se encuentra el proceso, el tamaño del mismo y su nombre. También hay que cambiar el campo 'sp' (cima de la pila), y en la trama de pila: las posiciones de los registros ***CS:IP*** (contador de programa), el valor de la dirección de comienzo del código, y los "*flags*" de estado con sus valores iniciales por defecto.

  Los pasos que modifican estructuras de datos como la tabla de procesos o las variables de gestión de memoria deben hacerse con las interrupciones inhibidas, pues constituyen regiones críticas. Otros pasos que afectan a la gestión de ficheros no requieren de dicha medida, porque las operaciones con ficheros las lleva a cabo el proceso "servidor" y está garantizado que mientras se lleva a cabo un servicio por dicho servidor, no se lleva a cabo otro, por lo que no hay concurrencia, ni por tanto condiciones de carrera.

- *Read()*: Da soporte a la llamada al sistema read(). Esta función se comporta del mismo modo que viene asiendo habitual en otros sistemas. Lee de un fichero con descriptor 'fd', una cantidad de bytes pedidos sobre un buffer en memoria, a partir de la posición actual del apuntador del fichero, devolviendo como resultado el número de bytes leídos. Esta función se apoya en la función "pos2sec()" que devuelve el sector de disco donde se encuentra una posición de fichero dada. La dificultad principal de esta función se da por el hecho de poder solicitar una cantidad cualquiera de bytes a partir de cualquier posición del fichero, lo que obliga a tener que leer uno o más sectores dependiendo de donde esté posicionado el apuntador de fichero y la cantidad solicitada de bytes. Estos sectores a leer a su vez de deben ir obteniendo a partir de los *clusters* que forman la cadena del fichero,

lo cual se hace gracias a la función "pos2sec()" arriba mencionada, la cual si lo requiere puede ir obteniendo el siguiente cluster a uno dado y de "traducir" un número de cluster a número de sector, "función clus2sect()".

- *Write()*: Esta función es análoga a Read() y también funciona de un modo similar a otros sistemas, no obstante, es algo más compleja, ya que en este caso se presenta el problema de la escritura más allá del final del fichero. La escritura podría efectuarse justo al final del fichero, o lo que es aún peor, que se realice más allá del final del fichero, lo cual sucede cuando se mueve el apuntador de fichero más allá del final del fichero (*lseek*) y posteriormente se realiza una escritura. En estos casos el fichero debe crecer y hay que realizar una llamada a la función "expandeEntrada()", además y para el peor caso, hay que escribir ceros en los sectores en los que se ha expandido el fichero, concretamente. En los bytes que se encuentren entre el fin del fichero previo y la posición a partir de la que se efectúa la escritura, tal y como indica la especificación de esta función en otros sistemas operativos como MS-DOS o UNIX.

- *MkDir*: Crea un directorio a partir de una ruta. Esta función que da soporte a la llamada al sistema "mkDir()" tiene que buscar en el directorio de la ruta una entrada para comprobar que no exista previamente el directorio. Si no existe llama a la función "creaEntrada()" para crear una nueva entrada del tipo directorio, pero al ser de ese tipo, debe además de ubicar un cluster de datos para a su vez crear en él las entradas "." y ".." que deben incluir todos los subdirectorios, por ello hay que volver a llamar a "creaEntrada()" dos veces más para añadir estas dos nuevas entradas, aunque en este caso se crean en un directorio distinto al de la primera vez, como es evidente.

- *chkDsk()*: Esta función da soporte al comando de consola "c*hkDsk*" que sirve para comprobar y reparar el sistema de ficheros de un disco. El método que utiliza esta función es algo complicado, pero es inevitable si se quiere hacer una comprobación exhaustiva con opción de reparación del sistema de ficheros. Para comprender bien esta función o comando es indispensable un conocimiento profundo de la estructura de ficheros tipo FAT, pues sin ello, sería imposible efectuar una mínima comprobación de la coherencia del sistema y en su caso la reparación. A continuación se describe "grosso modo" el procedimiento seguido:

  Antes de comenzar el proceso hay que asegurarse que la unidad esta desmontada, ya que no se debe actuar sobre la estructura del sistema de ficheros mientras la unidad está sien-

do utilizada por algún otro proceso. Una vez hecha la comprobación la unidad se monta para que pueda trabajar sobre ella el procedimiento "*chkDsk*". Con la unidad ya montada, se tiene la primera FAT cargada en memoria y ningún fichero abierto, y con estas condiciones lo primero que se hace es comprobar que las dos FAT son iguales, si no lo fueran se emite un mensaje en pantalla y en cualquier caso se usa la FAT con la que se invocó el comando. Seguidamente se pide memoria para ubicar una tabla de bits que servirá para reflejar el estado de cada cluster durante el proceso de chequeo. Dicha tabla tendrá al menos tantos bits como clusters tenga la FAT. Para el cálculo del tamaño de la tabla se toma el numero de clusters de la FAT redondeado al siguiente CLIC (16 bytes), ya que las peticiones de memoria dinámica así lo requieren. En esta tabla cada bit se corresponde con un cluster, el primer bit de la tabla con el cluster número 1, el segundo con el cluster 2, etc. El significado de cada bit es el de informar de si su cluster correspondiente pertenece a una cadena o no. Para examinar si un bit de esta tabla vale 1 ó 0 se dispone de la función "*getBit(n)*" y para modificar su valor "*setBit(n)*". Esta tabla es usada por la función "*checkDir()*", empleada en este proceso y comentada a continuación.

El siguiente paso importante consiste en la comprobación de directorios y ficheros. Este trabajo lo lleva a cabo la función "*checkDir()*", la cual actúa recursivamente y se vale de la tabla de bits comentada anteriormente. Esta función realiza un bucle donde recorre todas las entradas de un directorio. Si la entrada está libre se la salta y en caso contrario la trata. El tratamiento consiste en primer lugar en iniciar un recorrido por la cadena de *clusters* de la entrada. Durante este recorrido se va marcando a '1' el bit de la tabla correspondiente al cluster de la cadena analizado, comprobando previamente que no lo estuviera ya, pues en ese caso sería un error y se reportaría como "cluster cruzado o en bucle" y se repararía truncando en ese punto la cadena. Una vez finalizado el recorrido, hay que preguntar si la entrada es un directorio, en cuyo caso la función se llama a sí misma iniciándose una recursión; en caso contrario la función retorna al punto de llamada. Si dicho punto fue una llamada recursiva la función continuaría con otras entradas del directorio superior. Este proceso continuaría hasta finalizar el recorrido de todos los directorios de la unidad y devolviendo el control a la función principal "*chkDsk()*".

Una vez ejecutada la función "*checkDir()*" desde el directorio raíz, en la tabla de bits estarán marcados todos los bits de aquellos *clusters* que pertenezcan a un fichero/directorio, sin embargo podría haber cadenas "huérfanas", es decir, que no se puede acceder a ellas a

partir de una entrada de directorio, que no tendrán su bits homólogos de la tabla marcados, pues no se habría accedido a ellas durante el recorrido de la estructura en árbol de directorios. Por tanto, se ha comenzar otra fase en la que hay que buscar dichas cadenas perdidas y a ser posible, si así se indicara en el comando, recuperarlas. A continuación se describe qué es lo que se hace para conseguir este objetivo:

Se inicia un recorrido de la FAT secuencial y progresivo partiendo del cluster número 2. Durante el recorrido se omiten los clusters libres (cuyo valor es cero), y aquellos cuyo bit en la tabla está marcado (lo que significa que pertenece a un fichero/directorio). Si el cluster no cumple las condiciones mencionadas, significa que es un cluster perdido (huérfano) y llamemos 'C' a dicho cluster, entonces iniciamos un proceso de búsqueda del primer cluster de la cadena a la que pertenece el cluster 'C', aunque bien podría ser el único. Para realizar esta búsqueda recorro la FAT desde el punto donde se encontró el cluster 'C' y se avanza secuencialmente buscando un cluster, llamémosle 'B', que sea justo su inmediato anterior, es decir cuyo siguiente sea precisamente el cluster 'C'. Si no se encuentra dicho cluster 'B' significa que el cluster 'C' es el primero de la cadena, si por el contrario se encuentra, entonces se marca el bit correspondiente a dicho cluster y se repite el proceso buscando ahora justo el inmediato anterior a 'B', y así sucesivamente hasta encontrar el principio de la cadena. Al final, además de haber encontrado el principio de la cadena, se habrán marcado todos los bits homólogos de todos los clusters previos al cluster que llamamos 'C'. Una vez realizado este paso, nos resta continuar con un recorrido de la cadena de clusters partiendo del cluster 'C', durante el cual se van marcando los bits homólogos de los clusters de la cadena. Concluida esta fase, tendremos una cadena perfectamente localizada y sólo quedará crear una entrada de directorio a la que adjudicarle la misma, quedando con ello recuperada la misma. Con todo esto se habría efectuado el tratamiento de recuperación de la cadena a la que pertenece el cluster 'C', pero evidentemente habría que continuar el proceso inicial, lo cual se hace partiendo del siguiente cluster a 'C' en orden secuencial en la tabla FAT y repitiendo todo lo ya contado hasta finalizar completamente el recorrido secuencial de toda la FAT. Hay que añadir un pequeño pero importante detalle, si durante el proceso se produce algún cruce de cadenas, entonces se trunca la cadena que se estuviera recomponiendo.

Una vez finalizado el proceso de comprobación y recuperación de cadenas perdidas se marca el "flag" de la unidad a "sucio", para forzar la escritura de las dos FAT cuando se desmonta la unidad, lo que se hace justo a continuación. Por último se libera la memoria pedida para ubicar la tabla de bits auxiliar.

Una última consideración. En este módulo de ficheros existe una variable global llamada "***escrDisable***" que se usa para inhibir o permitir la escritura en disco. El valor normal de esta variable es *"FALSE"*, es decir, escritura no inhibida. La función "*leeEscr()*", que realiza la escritura en disco, **no** efectúa ninguna escritura si esta variable vale "TRUE". Durante el proceso "chkDsk", esta variable se pone a *"TRUE"* si el comando se ejecutó con la opción de no grabar los cambios en el disco, consiguiéndose con ello simplemente la detección y **no** la corrección de los errores. Por supuesto, antes de finalizar la función, el valor de dicha variable se restituye a su valor previo.

5.4 Servicios ofrecidos por el Sistema (llamadas al sistema).

Las llamadas al sistema se encuentran implementadas principalmente en el módulo "llamadas.c/h", no obstante, algunas de ellas utilizan funciones auxiliares que se encuentran en otros módulos y que en muchos casos se encargan prácticamente de todo. En este módulo se encuentra la función primordial de atención y despacho de las llamadas soportadas por el sistema. Esta función está implementada como tipo "void interrupt()". Este tipo característico de Turbo-C se usa cuando se desea instalar una función en un vector de interrupción, y es el método comúnmente empleado para instalar las RTI's típicas, como la de teclado o reloj. En este caso lo que se ha hecho es utilizar un vector (60H) de los disponibles en el sistema para instalar en él la rutina de servicio del sistema operativo, "***void interrupt rti_SO()***". En el fichero "llamadas.h" se encuentran los prototipos de las funciones que se usan para instalar y desinstalar este vector de interrupción. Estas funciones, "*inicLlamadasSO()*" y "*restablecerLlamadas()*", se utilizan durante el arranque del sistema, para la instalación del vector, y al finalizar el sistema, para su desinstalación. Al igual que sucede con los otros vectores como los del teclado o reloj.

El trabajo de la función de reparto de llamadas al sistema, "rti_SO()", es bastante sencillo: En primer lugar establece una nueva pila dentro de SO, antes de hacer cualquier otra cosa, y restaurarla justo antes de finalizar. Hecho esto esta función básicamente lo que hace es determinar el

código de operación, que recibe en el registro **AH** pasado en la pila del proceso invocante, y en función de éste, llama a la rutina encargada de llevar a cabo el servicio. Todas las rutinas de servicio están definidas del mismo modo, y existe una tabla que contiene todas las direcciones de estas funciones. El número de índice que ocupa en la tabla cada función se corresponde exactamente con el código de operación. De este modo para invocar la rutina apropiada basta con utilizar la sentencia "*(*dirLlamada [cod_op])()*", donde "*dirLlamada*" es el nombre de la tabla que contiene los punteros a función de las funciones de servicio y "*cod_op*" es el código de operación recibido en el registro **AH**. En el caso especial de que el código de operación sea superior a 80h, el servicio lo atenderá el proceso servidor de ficheros y no se bifurca del modo habitual si no con una sentencia "if" previa. La función "so_serv()" es la encargada de atender estas peticiones y lo que hace básicamente es meter en la cola del proceso servidor la petición y dejar al proceso de usuario que la solicitó bloqueado en espera "BLK_SERV". Si el proceso servidor hubiera estado bloqueado en el momento del encolado entonces se le desbloquearía dejándolo preparado, para que pueda ser planificado y con ello atender la petición.

Las funciones de servicio cuya dirección se encuentra en la tabla reciben todas ellas los parámetros de entrada en los registros copiados en la pila del proceso de usuario. Para facilitar el acceso a los mismos se ha declarado la macro "RGP(N,R)", cuya declaración es:

```
#define RGP (N, R) (tblProc[N].sp->R)
```

Esta macro nos permite acceder a cualquier registro guardado en la pila usando para 'N' el número de proceso y para 'R' el nombre de un campo de una 'union' de 'C' declarada en "tipos.h". Por ejemplo para acceder al registro **AX** se pondría **A.X** y para acceder al registro **AH** se pondría **A._.H.** *Ej. RGP (npr, A.X).* A continuación se detallan las funciones de servicio que dan soporte a las llamadas al sistema actuales*:*

- (AH=0). *so_finProceso():* Llama a killProcess.
- (AH=1). *so_matarProceso():*
- (AH=2). *so_leerTecla():* Llama a "LeerTeclaLista()". Si esta función no retornara ninguna tecla y se optó por llamada bloqueante, se deja al proceso bloqueado.
- (AH=3). *so_printCar():* Muestra un carácter en la posición del cursor de la ventana del proceso en ejecución.

- (AH=4). *so_printStrHasta():* Muestra una cadena de caracteres en la posición del cursor de la ventana del proceso actual con un límite máximo de longitud.
- (AH=5). *so_printBase():* Muestra un número en base decimal o hexadecimal en la posición del cursor de la ventana del proceso actual.
- (AH=6). *so_moverWindow:* Mueve una ventana a unas coordenadas dadas.
- (AH=7). *so_coloWindow:* Cambia el color de fondo y tinta usado al mostrar caracteres por una ventana.
- (AH=8). *so_fotoWin:* Salva/restaura la posición y color de una ventana.
- (AH=9). *so_duerme:* Deja bloqueado un proceso durante un lapso de tiempo especificado en mseg. Si dicho lapso es 0 o negativo el proceso queda bloqueado indefinidamente. En el primer caso la razón de bloqueo es BLK_SLEEP y en el segundo BLK_PAUSE. Esta función únicamente calcula los tics de reloj equivalentes a los mseg. Especificados y los guarda en un campo del descriptor de proceso, para a continuación dejar bloqueado el proceso. La RTI de reloj es la encargada decrementar y comprobar que el lapso de tiempo especificado ha vencido para en ese caso despertar al proceso. Para el caso del que proceso se haya bloqueado por BLK_PAUSE. El mecanismo es distinto, en este caso la RTI no hace nada y es habitualmente un proceso hijo el que despierta a un proceso padre bloqueado en este estado.
- (AH=0Ah). *so_despierta:* Despierta un a proceso bloqueado por BLK_SLEEP o BLK_PAUSE.
- (AH=0Bh). *so_iniSemaforo:* Establece el valor inicial de un semáforo. También inicializa su cola asociada al estado de vacía.
- (AH=0Ch). *so_bajaSemaforo:* Efectúa la típica operación de bajar semáforo. Esta operación consiste en preguntar si el valor del semáforo es mayor que cero. Si es así lo decrementa y retorna, en caso contrario deja al proceso bloqueado con razón de bloqueo "BLK_SEMAFORO", lo añade al final de su cola y retorna.
- (AH=0Dh). *so_subeSemaforo:* Efectúa la típica operación de subir semáforo. Esta operación consiste en preguntar si hay algún proceso en la cola del semáforo, en cuyo caso deberá estar lógicamente bloqueado por "BLK_SEMAFORO", y de ser así lo quita de su cola y lo pone en la cola de preparados, en caso contrario se limita a incrementar el valor del semáforo, y retorna en ambos casos

- (AH=0Eh). *so_enviaMsjbuzon:* Este servicio implementa el envío de un mensaje de longitud fija a un buzón de capacidad fija. Estos tamaños están fijados previamente por el sistema en tiempo de compilación, así como también el número máximo de buzones que se implementa mediante una tabla. El comportamiento es sencillo. Si el mensaje enviado entra en el buzón por haber sitio en éste, el proceso deposita el mensaje en el buzón y continúa su ejecución, si por el contrario el buzón está lleno, entonces el proceso quedará bloqueado por "BLK_BUZON" y añadido a la cola del buzón. No saldrá de este estado hasta que le toque el turno de ser despertado, cuando algún otro proceso al recibir un mensaje de este buzón, habilite sitio en el mismo y haga que se despierte un proceso de la cola del buzón. Los detalles de implementación se pueden ver en el código pues está ampliamente comentado.

- (AH=0Fh). *so_recibeMsjBuzon:* Este servicio es recíproco al anterior. Si al intentar recibir un mensaje de un buzón, el buzón no está vacío, el mensaje es extraído del buzón y entregado al proceso receptor, continuando éste su ejecución. Si por el contrario estuviese vació, entonces el proceso quedará bloqueado por "BLK_BUZON" y puesto al final de la cola del buzón. De este estado saldrá cuando otro proceso haga un envío a este buzón y le toque el turno de ser despertado, en cuyo caso se le entregará el mensaje y se le pondrá en la cola de preparados.

- (AH=10h). *so_fork:* Esta llamada al sistema es ampliamente conocida en el entorno de UNIX. Sirve para clonar un proceso. Normalmente un proceso conocido por "padre" invoca esta llamada y a resultas de la misma, se crea un nuevo proceso nuevo llamado "hijo", el cual tiene el mismo código, datos y pila que su predecesor y su ejecución continúa, al igual que su "padre", justo después de esta llamada, con la diferencia de que al proceso padre se le devuelve el identificador del proceso hijo (pid) y al proceso hijo se le devuelve el valor 0. Los pasos groso modo seguidos por esta función son:

  - *Buscar un descriptor libre para el "hijo" en la tabla de procesos.*
  - *Copiar el descriptor del proceso "padre" en el del "hijo".*
  - *Pedir la cantidad de memoria que ocupe el "padre" para el "hijo".*
  - *Copiar la imagen de memoria del "padre" en la del "hijo".*
  - *Actualizar contador de referencias de ficheros abiertos.*

- *Modificar los valores que cambian en el descriptor del "hijo". El "pid", el "pPid", el puntero a trama de pila, "sp" (solo la parte segmento), y las copias de los registros CS, DS y ES guardadas en dicha trama de pila.*
- *Incrementar el número de procesos y dejar en preparados el proceso "hijo".*
- *Retornar al "padre" el pid del "hijo" y poner en la trama de pila del proceso "hijo", en la posición del registro AX, el valor 0, para que en su momento se le devuelva este valor al proceso "hijo".*

El proceso "hijo" hereda los ficheros abiertos y también la ventana terminal. El código se copia en una nueva área de memoria pero una posible mejora podría consistir en no duplicar dicho código sino compartirlo.

- (AH=11h). *so_tomaMem:* Solicita memoria al sistema. La función homóloga TomaMem(), que se encuentra implementada en el módulo "memoria.c/h" recorre la lista de huecos buscando uno de tamaño igual o superior al solicitado y si lo encuentra devuelve su dirección base (en *paragraphs*). Si sobra memoria en el hueco hallado, la deja en el mismo, modificando la lista de huecos para reflejar la nueva dirección del hueco resultante. Si el tamaño del hueco hallado fuera exactamente igual al solicitado, se elimina el hueco. Si no se encuentra ningún hueco de tamaño suficiente, la función retornaría 0.
- (AH=12h). *so_sueltaMem:* Devuelve memoria al sistema. Es una función reciproca a la anterior. Al devolver memoria al sistema puede suceder que el bloque devuelto tenga uno, dos o ningún hueco adyacente a él, si no tiene ninguno, entonces se genera un nuevo hueco, si tiene uno, tanto si su dirección es anterior o posterior a la dirección de la memoria devuelta, ésta se fusiona con el hueco adyacente, y si por último tiene dos huecos adyacentes, entonces ambos huecos se fusionan con la memoria devuelta y como resultado se produce un solo hueco donde antes había dos, reduciéndose en este caso el número de huecos.
- (AH=13h). *so_test:* Esta llamada tiene únicamente el propósito de ser usada para realizar pruebas durante la fase del desarrollo del sistema.
- (AH=14h). *so_printEntDir:* Muestra el contenido de una entrada de directorio por la ventana terminal del proceso actual.

Existe además otra RTI en el sistema 'SO', la función "**void interrupt rti_AuxSO()**". Esta función presta un servicio de apoyo a los procesos de usuario aportando funciones "auxiliares" y se

instala en el vector siguiente al de llamadas al sistema. Actualmente sólo incluye la función auxiliar “so_leerLinea()”, la cual se trata de forma análoga a cualquier otra llamada, aunque no igual, si bien para los procesos de usuario no hay prácticamente ninguna diferencia. La diferencia fundamental para el sistema 'SO' es que en la RTI no se establece una nueva la pila dentro del sistema, sino que se deja la de usuario, lo que permite invocar a la RTI de llamadas al sistema desde dentro del propio sistema, evitándose un problema de reentrancia. Por ejemplo, la función auxiliar “so_leerLinea()”, puede llamar al servicio de 'SO' “leerTecla()”, el cual puede dejar bloqueado al proceso. Esto puede hacerse, aún a pesar de que al invocar a “leerTecla()” se haga desde dentro del sistema, porque la pila continúa siendo la del proceso de usuario, y esto permite que el mecanismo usado para bloquear un proceso siga siendo válido.

Otro aspecto importante a señalar en este módulo es el referente a los servicios relacionados con ficheros que atiende el proceso servidor. La función "srv_main()" constituye el cuerpo principal del proceso servidor. Esta función contiene un bucle sin fin en el cual se está continuamente atendiendo peticiones de servicio de ficheros. En el bucle se pregunta si hay alguna petición en la cola de asociada al servidor y de ser así se extrae la petición de la cola y ésta se despacha, invocando la rutina de servicio asociada a la misma. El número de proceso que ha efectuado la petición y que permanece bloqueado se guarda en la variable del sistema "nprAtnd", la cual se usa en algunas partes del código de ficheros para conocer qué proceso realizó la petición, ya que la variable "nprEjec" no puede usarse para dicho fin al ser siempre en este caso su valor la del proceso servidor. Una vez atendida la petición se elimina ésta de la cola del servidor y se desbloquea al proceso de usuario poniéndolo en la cola de preparados para que pueda continuar con su ejecución. Si por otra parte en el bucle, al comprobar si hay alguna petición en la cola, se detecta que la cola está vacía, entonces el proceso servidor se duerme indefinidamente con razón de bloqueo "BLK_PAUSE". De este estado saldrá cuando un proceso de usuario al hacer una petición al sistema, provoque que la función de servicio "so_serv()" despierte al servidor, tal y como ya se ha comentado anteriormente.

El mecanismo de despacho de servicios es distinto al empleado en la RTI de llamadas al sistema. En este caso, y como modo alternativo, se utiliza una sentencia "switch" en vez de una tabla de punteros, y los códigos de operación de estos servicios empiezan en **80h**. Las llamadas al sistema relacionadas con ficheros que atiende el servidor son:

m*ontaDrv (ah=80h), drvMontado (ah=81h), runFichero (ah=82h), execv (ah=83h), setWorkDir (ah=84h), getWorkDir (ah=85h), setResulBus (ah=86h), buscaPriEntDir (ah=87h), buscaSigEntDir (ah=88h), open (ah=89h), close (ah=8Ah), read (ah=8Bh), write (ah=8Ch), lseek (ah=8Dh), fstat (ah=8Eh); erase (ah=8Fh), mkDir (ah=90h), rmDir (ah=91h), desmontaDrv (ah=92h), chkDsk (ah=93h), touch (ah=94h).*

La mayoría de las cuales ya se han comentado en el capítulo de ficheros previamente visto.

Hay que hacer notar por último un detalle importante. Hay que inhibir y permitir interrupciones en los momentos adecuados para crear regiones críticas y así evitar condiciones de carrera. En el código de "srv_main()" se ha comentado este punto con el suficiente detalle.

5.5 Rutinas de interfaz de llamadas al sistema.

Las rutinas de interfaz de llamadas se encargan básicamente de trasladar los parámetros de entrada que en "C" se reciben en la pila, a los registros pertinentes del procesador donde los espera la RTI de servicio del S.O., y por supuesto de invocar a dicha RTI, para tras retornar ésta, devolver el resultado que se encuentra en el registro AX en la pila, aunque el empleo de la pila es implícito por el uso del lenguaje "C". Cada rutina de interfaz es responsable de introducir el código de operación correcto asociado a la llamada en el registro AH, así como también utilizar los registros convenidos para paso de información a la RTI de servicio del sistema. Existe lógicamente una rutina de interfaz por cada llamada al sistema.

Estas rutinas forman parte del proceso de usuario, y al compilar un programa de usuario se incluyen en el mismo. Por tanto no tendrían por qué formar parte del sistema operativo, sin embargo, puesto que se ha incluido un intérprete (consola) dentro del sistema, y éste hace uso de estas funciones, como cualquier otro proceso de usuario, al final el código de estas funciones también forma parte de 'SO'.

El código de las rutinas de interfaz se haya distribuido en varios ficheros que se encuentran ubicados en un subdirectorio interno llamado "LL_S_SO". En este subdirectorio se encuentran los siguientes ficheros: "bsic-ifz.c/h", "memo-ifz.c/h", "proc-ifz.c/h", "fich-ifz.c/h" y "test-ifz.c/h". A continuación se detallan las rutinas de interfaz que contiene cada uno de estos ficheros:

bsic-ifz.c/h: finProceso, matarProceso, leerTecla, LeerTeclaLista, printCar, printDec, printHex, printStr, printStrHasta, printBase, printBase, MoverWindow, ColorWindow, fotoWin, y leerLinea.

memo-ifz.c/h: TomaMem y SueltaMem.

proc-ifz.c/h: duerme, despierta, iniSemaforo, bajaSemaforo, subeSemaforo, enviaMsjBuzon, recibeMsjBuzon y fork.

fich-ifz.c/h: printEntDir, montaDrv, desmontaDrv, drvMontado, setWorkDir, getWorkDir, runFichero, execv, open, close, read, write, lseek, fstat, erase, mkDir, rmDir, chkDsk y touch.

test-ifz.c/h: test_ll.

En el fichero "fich-ifz.h" además de encontrarse los prototipos de las rutinas de interfaz, se encuentran también algunas definiciones de tipos y constantes que se requieren tanto en 'SO' como en los procesos de usuario, siendo esta la razón por la que se encuentran aquí.

Por último hay que resaltar que en el subdirectorio "LL_S_SO", también se encuentra el fichero "inic-usr.c" que si bien no contiene rutinas de interfaz, si contiene algo fundamental para la compilación de procesos de usuario. Se trata del código inicial, incluido den la función "start()", con la que deben comenzar todos los programas de usuario. Esta función debe ser obligatoriamente la primera y por ello este módulo "inic-usr.c" debe aparecer en una directiva "#include" antes que cualquier otra línea relevante. La función "start()" se limita a llamar a la función "main()" y a continuación realizar la llamada al sistema "finProceso".

Otro detalle importante a comentar sobre el fichero "inic-usr.c", es sobre el uso de la directiva "**asm DGROUP GROUP _TEXT, _DATA**", que figura en el mismo. Esta directiva, tal y como podemos leer en los comentarios incluido en el código, es necesaria para que el compilador genere correctamente el *offset* de las direcciones de las variables globales, las cuales se ubican en el segmento de datos y son siempre relativas al registro 'DS'. Este *offset* debe generarse tomando como origen el registro 'CS', y esto se consigue gracias a esta directiva al agrupar los segmentos de código en datos en uno solo, consiguiendo con ello que los dos registros de segmento valgan lo mismo. Este detalle está relacionado con la opción del modelo de compilación usado por turbo-C, que en este caso resulta ser una especie de híbrido entre "*small"* y "*tiny"*.

6. RESULTADOS

El primer resultado que se ha obtenido con la realización de este proyecto ha consistido en la consecución de un sistema operativo de tamaño reducido y de poca complejidad, que corre sin problemas tanto en una máquina desnuda tipo 'PC' compatible, como en una máquina virtual tipo DOSBox, Qemu, etc., y que puede ser utilizado por un entorno concreto de enseñanza de la asignatura "Sistemas Operativos", con un alto grado de ajuste a sus necesidades.

Los ficheros fuente de este sistema se encuentran en el en el directorio "**MiSO**", dentro del 'CD 'de este proyecto; incluyéndose además estos otros subdirectorios:

- **.git**: Datos usados por el software "git" para la gestión del proyecto vista en el subcapítulo de "realización de #proyecto".
- **LL_S_SO**: Ficheros fuente con las rutinas de interfaz de llamadas vistas en el apartado "Rutinas de #Interfaz del Sistema".
- **MI_C0**: Código fuente ensamblador de inicialización usado por el compilador. Su utilidad y necesidad se vio en el "estudio preliminar de las herramientas de desarrollo", #mi_C0.

- **USRS_PRG**: Contiene los ficheros fuente de las aplicaciones de usuario. Estas aplicaciones se verán más adelante:

Otro resultado de la ejecución de este proyecto ha consistido en poder llevar a cabo el desarrollo de unas prácticas de laboratorio para su realización por parte del estudiante, lo que era además uno de sus principales objetivos. Estas prácticas han consistido por lo general, en solicitar al estudiante que lleve a cabo una modificación o mejora en el sistema. Más adelante, en uno de los apartados de este capítulo, haremos una breve descripción de algunas de ellas.

Por otro lado; con el ánimo de probar, depurar y mostrar la funcionalidad del sistema operativo, además de llevar a cabo un cometido; se han programado una serie de comandos que interpreta y ejecuta el proceso "consola", el cual se encuentra integrado en el sistema operativo; y también una serie de aplicaciones de usuario, que corren en el espacio de usuario y hacen uso de las llamadas al sistema. Estos comandos y aplicaciones se detallan en los siguientes apartados de este capítulo.

6.1 Comandos de consola.

La "consola" es un proceso del sistema que realiza las funciones de intérprete de comandos, con funciones de administración del sistema. Al contrario que los intérpretes de comandos en el espacio de usuario, la consola incluye algunos comandos que son exclusivos, debido principalmente a que éstos utilizan funciones internas; no disponibles como llamadas al sistema. Cuando el código del sector de arranque (*boot*) cede el control de ejecución al sistema operativo, lo primero que se hace es llevar a cabo la inicialización del Sistema. Entre otras cosas, se establecen determinados vectores de interrupción, se inicializan determinadas estructuras de datos del Sistema y eventualmente se prepara el proceso "consola" (CNSL), para que tome el control de la ejecución y comience a realizar su tarea básica, la cual consiste, tal y como ya se ha indicado, en interpretar y ejecutar comandos y en su momento, tras el comando pertinente, finalizar el sistema.

El fichero del sistema donde se encuentra este proceso (CNSL) es 'so.c', el cual es además, el que contiene la función "main()" con la que se inicia sistema operativo. Esta función comienza invocando las funciones de inicialización de las ventanas (i*nicWindows*), memoria, (*inicMemoria*), ficheros (*inicFicheros*), procesos (*inicProcesos*) y llamadas al sistema (*inicLlamadasSO*). Estas

funciones de inicialización se encargan de establecer los valores iniciales de las estructuras de datos del sistema en cada uno de los módulos. En general su código es bastante sencillo, aunque quizá convenga comentar algo más función "inicProcesos()". Para inicializar "procesos", no basta con poner la tabla de procesos a ceros, también hay rellenar las entradas correspondientes a los procesos "consola" y servidor". La parte más delicada quizá es la referente a la inicialización del puntero de pila que se guarda en el descriptor, y también el contenido de los registros que se guardan en ella. Los valores de los registros CS e IP guardados deben contener la dirección de la primera instrucción a ejecutar del proceso, para así poder iniciar correctamente su ejecución cuando el planificador seleccione al proceso. Para el caso del proceso "servidor" esta dirección será la de la función "srv_main()" y para el caso del proceso "consola", al ser éste un proceso especial, en el sentido de que es el proceso con el que arranca el sistema operativo, no hace falta inicializar estos valores, ya que el proceso nace ya en estado de ejecución y en cuanto se bloquee por primera vez, con la primera petición de teclado, el sistema guardará el valor de SS:SP el su descriptor, así como el resto de registros en su pila, lo que servirá posteriormente para restaurar el flujo de ejecución del proceso. Finalmente añadir que: la función "inicProcesos()" debe poner la variable "nprEjec" igual a CNSL, el numero de proceso a 2, e inicializar la cola de preparados para que esté en ella únicamente el proceso servidor.

Después de estas inicializaciones se ejecutan las funciones de redirección de los vectores de interrupción que usa el sistema: El teclado, el reloj, etc., y finalmente se muestran en pantalla los mensajes del sistema, se monta la unidad de trabajo de la consola y comienza un bucle sin fin de interpretación y ejecución de comandos. En este bucle la consola llama a "leerLinea()" y en esta función se llama a "leerTecla()", la cual deja bloqueada a la consola, desbloqueándose con cada pulsación de tecla. Cuando eventualmente se pulsa la tecla INTRO, como tecla final de la introducción de un comando, "leerLinea()" devuelve la cadena de caracteres que forman el comando y éste se convierte a una variable del tipo enumerado, que identifica el comando y se utiliza en una sentencia "switch" para su procesamiento. Los comandos actualmente implementados son: "EXIT", "CLS", "DIR", "MEM", "LSV", "TFA", "PS", "KILL", "DUMP", "MW", "CW", "CD", "DEL", "RD", "MD", "TOUCH", "COMPAC", "DESMON", "TYPE", "TST", "AYUDA" y "CHKDSK", y se comentaran a continuación:

- **EXIT**: Realiza los pasos necesarios para salir o finalizar el sistema de forma ordenada. Estos pasos son: 1) Recorrido de la tabla de procesos desde el proceso siguiente a la consola (CNSL+1), hasta el último, matando a cada uno de ellos. Esto se realiza con las interrupcio-

nes inhibidas. 2) Recorrido de la tabla de ficheros de la consola cerrando cada uno de ellos. 3) Se desmontan las unidades montadas. 4) se Elimina la ventana terminal de la consola. 5) Se restablecen los vectores de interrupción establecidos durante la inicialización del sistema. Este paso sólo es necesario si el sistema corre como invitado de MS-DOS (aunque no lo usa), sin embargo se realiza siempre pues en cualquier caso no es malo, por último, 6) se retorna al sistema anfitrión o se reinicia el sistema dependiendo del caso.

- **CLS**: Borra la pantalla. Lo hace mediante printCar(FF), el carácter "*Form Feed*".

- **DIR**: Muestra el contenido del directorio que se le especifica o del directorio actual de trabajo si no le especifica ninguno. La función que lo implementa 'Dir()" está declarada e implementada en este mismo fichero. Esta función utiliza las llamadas al sistema "buscaPriEntDir" y "buscaSigEntDir", y constituye un buen ejemplo del uso de estas funciones.

- **MEM**: Muestra la dirección de comienzo de la memoria de asignación dinámica y la lista de huecos disponibles. La función que implementa el comando es "mostrarMemoria() y se encuentra implementada en el fichero "memoria.c". Los detalles de cada elemento de la lista mostrado son: dirección de comienzo de cada hueco, tamaño en paragraphs y dirección del siguiente hueco.

- **LSV**: Muestra la lista de ventanas. La función que lo implementa es "listaWin()" y se encuentra en el fichero "windows.c". Los detalles de cada elemento de la lista mostrado son: Dirección del objeto ventana, dirección de la ventana que se encuentra justo arriba, dirección de la ventana que se encuentra justo debajo y el nombre con el que se creó la ventana.

- **TFA**: .Este comando muestra los ficheros abiertos de la tabla informativa de de ficheros. La función que lo implementa, "listarTFAbiertos()", se encuentra en "ficheros.c". la información que se muestra es: descriptor de fichero, nombre, extensión, atributos, primer cluster, tamaño, sector y número de entrada dentro del sector donde se encuentra el fichero.

- **PS**: Muestra los procesos vivos del sistema, El comando los implementa la función "listarProcesos" que se encuentra en "procesos.c". La información mostrada es: el número de proceso (índice dentro de la tabla de procesos), el identificador de proceso (pid), el estado (preparado, ejecución, etc.), el siguiente proceso de la posible cola en la que se encuentre el proceso, la dirección de memoria donde se ha cargado el proceso, la cantidad de "*paragraphs*" que ocupa el proceso, los registros CS, IP, DS, SS y SP y flags del procesador, número de caracteres del buffer de teclado de su ventana terminal y por último nombre del proceso. La función ob-

tiene la información principalmente de la tabla de procesos. Los registros se obtienen de la pila del proceso de usuario si el proceso no está en ejecución, porque en ese caso se obtienen directamente del procesador. Hay que hacer notar que actualmente, cuando se ejecuta el comando, se hace con el proceso consola en ejecución, y siempre se muestra éste como proceso en ejecución.

- **KILL**: Mata el proceso cuyo "pid" se le indica en el parámetro. El comando se implementa mediante la llamada al sistema "matarProceso".

- **DUMP**: Muestra el contenido de memoria. En los parámetros se puede expresar una dirección de comienzo como segmento : desplazamiento y una cantidad de bytes opcional. De forma implícita se muestran 64 bytes. La función que lo implementa es "volcar()" y se encuentra en "memoria.c".

- **MW**: Mueve o redimensiona una ventana. Se realiza mediante la llamada al sistema "moverWindow". Si las coordenadas indicadas en el comando no modifican ni el largo ni el ancho de la ventana, el comando se comporta como movimiento, en caso contrario, como redimensionamiento.

- **CW**: Cambia los colores de fondo y tinta de una ventana. Se implementa mediante la llamada al sistema "colorWindow".

- **CD**: Cambia el directorio de trabajo de la consola. Se implementa mediante la llamada al sistema "setWorkDir".

- **DEL**: Elimina un fichero o conjunto de ficheros según se especifique en el parámetro el nombre de fichero usando el "*" en el nombre o la extensión. Se implementa mediante la llamada al sistema "erase".

- **RD**: Elimina un directorio y todo su contenido procediendo de forma recursiva. Se implementa mediante la llamada al sistema "rmDir".

- **MD**: Crea un directorio nuevo. Se implementa mediante la llamada al sistema "mkDir".

- **TOUCH**: Pone como fecha del fichero / directorio especificado, la actual. Puede opcionalmente entrar recursivamente (opción /r) en los posibles subdirectorios. Este comando se apoya en la llamada al sistema "touch".

- **COMPAC**: Realiza una compactación o desfragmentación de la memoria. Este proceso consiste en mover los bloques de memoria asignados, como sea preciso, para conseguir que todos los huecos se unan formando uno sólo. Los detalles se han explicado con anterioridad en el capítulo de gestión de memoria (ver #Compacta).

- **DESMON**: Desmonta una unidad de disco liberando la memoria ocupada para la FAT de la misma. Esta implementada mediante la llamada al sistema desmontaDrv. La operación no puede llevarse a cabo si el sistema mantiene algún fichero abierto en la unidad

- **TYPE**: Muestra el contenido ASCII de un fichero. Esta implementado mediante la función "type()" que se encuentra en el propio fichero "so.c". Esta función se limita a abrir el fichero, para luego ir leyéndolo y mostrándolo en pantalla, cerrándolo finalmente.

- **TST**: Este comando se utiliza exclusivamente para realizar pruebas.

- **AYUDA:** Muestra la lista de comandos disponibles y los parámetros admitidos en cada uno de ellos.

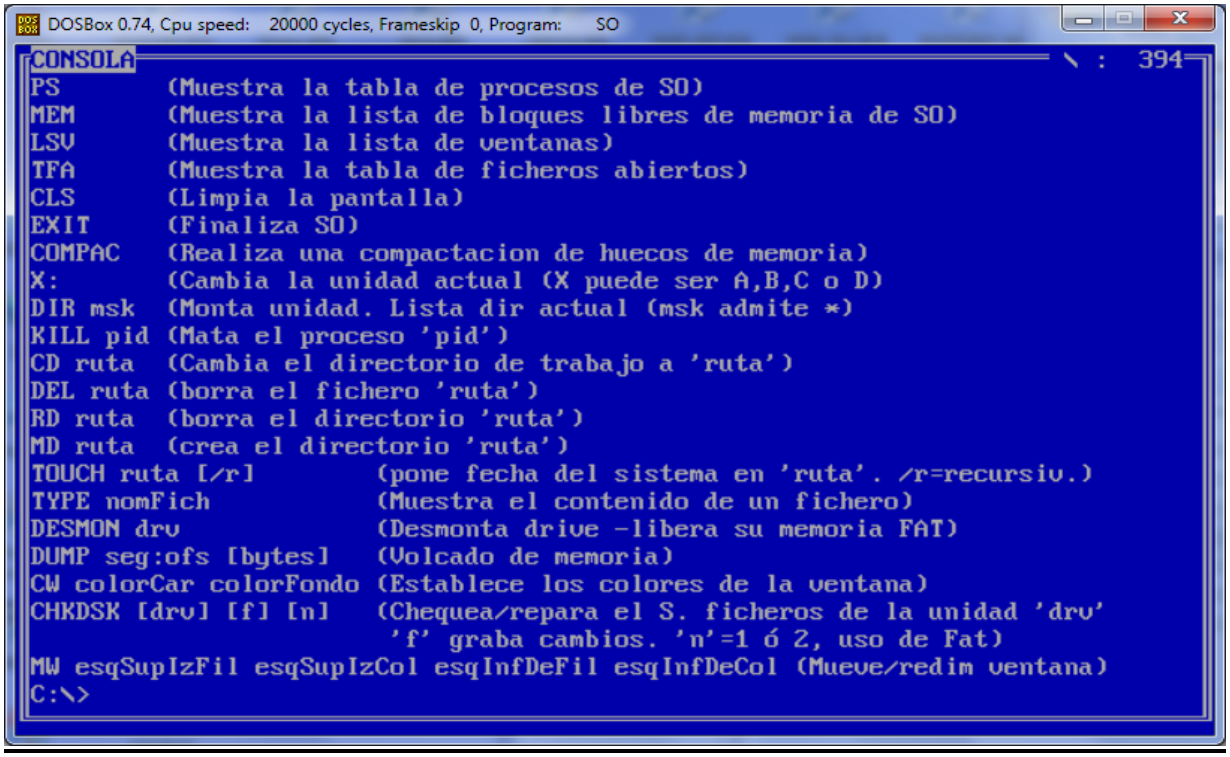


Figura 19: Ejecución comando ayuda

- **CHKDSK**: Realiza una comprobación/reparación de una unidad de disco. Se implementa mediante la llamada al sistema "chkDsk", cuyos detalles de funcionamiento se comentaron

en el capítulo de gestión de ficheros (ver #chkDsk). Para que se pueda hacer la comprobación la unidad tiene que ser desmontada. Si no se logra, no se realiza el chequeo. El comando admite que se le especifique en los parámetros: la unidad a comprobar, si se desea reparar (/F) o sólo comprobar, y cuál de las dos FAT se desea considerar como buena (1 ó 2). De forma implícita se consideran: la unidad de trabajo, la FAT 1 y el modo "sólo comprobación.

- Además de los comandos vistos hasta ahora, el interprete también admite el cambio de la unidad de trabajo introduciendo la letra de la unidad seguida de ":", ej. **A:** . Cualquier otro comando se interpretará como la ejecución de un fichero. El fichero puede tener la extensión "bin" que no es necesario teclear. Ej.: **A:>errante** <┘ .

6.2 Procesos de usuario.

Dentro del directorio "MiSO", que contiene las fuentes del proyecto, se encuentra el subdirectorio "USRS_PRG", el cual contiene el código de las aplicaciones de usuario que se han desarrollado para demostrar la funcionalidad del Sistema Operativo, y a la vez de servir de ejemplo para la realización de otros programas.

Antes de ver uno a uno los procesos de usuario, se va a explicar el proceso seguido para su compilación. En el directorio "USRS_PRG" se encuentra el comando *batch* "compila.bat" que contiene las órdenes necesarias para compilar un programa de usuario. Una de las ordenes invoca al compilador: "**tcc –B –C -1 –k- -c –mt –w+pro –g20 –j10 %1.c**". La información sobre las opciones de compilación empleadas se puede obtener de la documentación interna del compilador "Turbo-C". Después de compilar, el proceso por lotes hace una pausa que permite examinar los errores de compilación. El proceso por lotes se reanuda al pulsar la tecla "INTRO", y entonces se ejecuta la línea "**tlink /c/s %1.obj,,,\tc30\lib\cs.lib**", lo que lleva a cabo la fase de montaje. Una vez generado el ejecutable tras el proceso de montaje (*link*), se ejecuta la línea "**exe2bin %1.exe %1.bin**", lo que convierte el fichero ".exe" en un fichero ".bin". Este último paso elimina la cabecera del fichero ".exe". Las demás líneas son opcionales. Seguidamente vemos un ejemplo de compilación del programa de usuario "hola.c". En él se presupone que el fichero "hola.c" se encuentra en el directorio "HOLA".

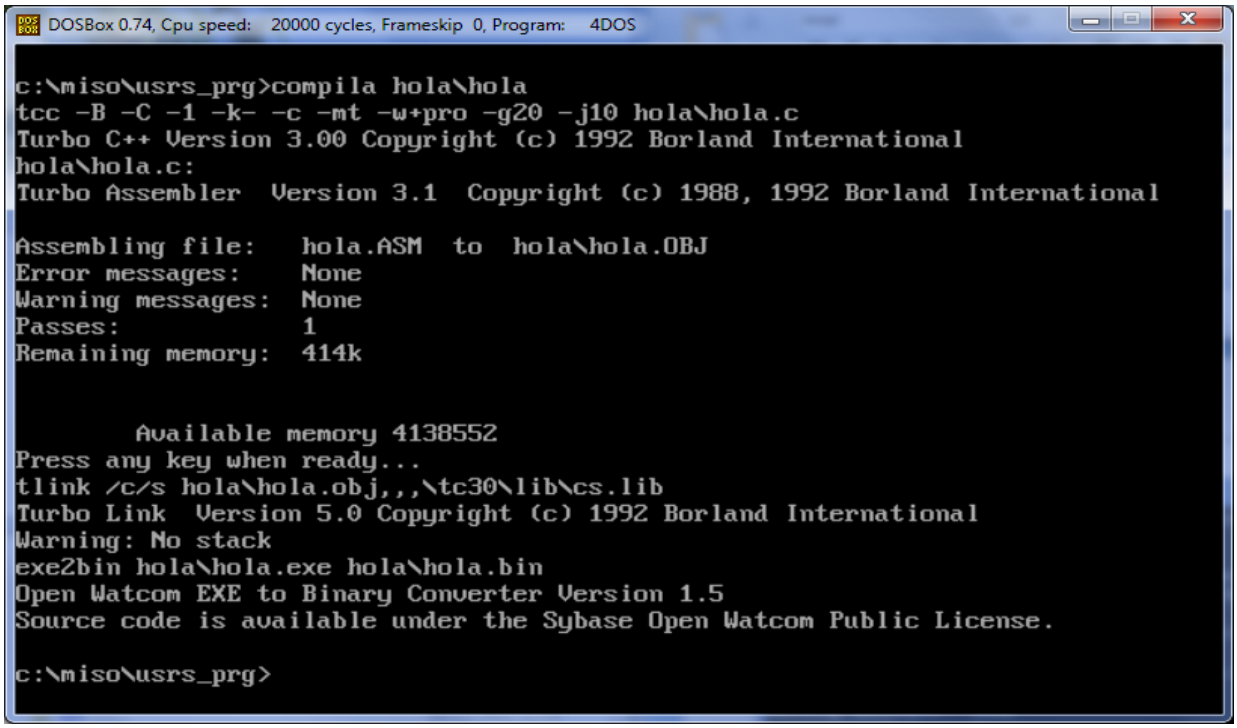


Figura 20: Ejemplo de compilación de un programa de usuario

Del mismo modo, sustituyendo simplemente el nombre "hola" por el del cualquier otro programa/directorio, se podría generar el ejecutable ".bin" de cualquier otro programa. También se dispone de un comando de procesamiento por lotes para facilitar la compilación de todos los programas de usuario del directorio USRS_PRG: "comptodo.bat".

Seguidamente se detallan las aplicaciones de usuario existentes:

- **Calibra**: Programa que solicita por teclado un tiempo en milisegundos y realiza una llamada a "sleep" con dicho tiempo. Esto permite comprobar aproximadamente si el sistema duerme al proceso durante el intervalo de tiempo correcto.

- **Hola**: Simplemente muestra en la ventana del proceso recién creado el mensaje "SO: Hola mudo.".

- **Test**: Realiza una llamada al sistema test. Actualmente se limita a pasar unos valores en los registros y espera esos mismos valores multiplicados por 2, lo que nos permite comprobar que el paso y retorno de parámetros es correcto.

- **Errante**: Es un programa que se limita a solicitar un valor numérico entre 0 y 9, el cual indica la velocidad a la que se tiene que desplazar la ventana del proceso por la pantalla describiendo una espiral cuadrada. Entre un movimiento y otro el programa realiza una espera activa. Es decir consume CPU. Al final de una serie de vueltas el programa finaliza.

- **Erraslp**: Es un programa similar a "errante", pero en este caso la espera no es activa. Ésta se hace mediante una llamada a "sleep", lo que libera la CPU. Este efecto puede constatarse fácilmente si se ejecutan varios programas "Errante/Erraslp" y se ejecuta el comando "*type*" para mostrar el contenido de un fichero de texto. En el caso de varios errantes el contenido del fichero se muestra a saltos, pues cuando le toca el turno a "*type*", éste muestra el contenido y cuando le toca el turno a los otros procesos, como consumen CPU en su turno y hay varios, el tiempo empleado por ellos es significativo y se percibe la pausa forzosa del comando "*type*". En el caso de "Erraslp", al ser una espera con bloqueo, la CPU se libera y puede asignarse al comando "*type*", que prácticamente no nota las interrupciones de los otros procesos, y por tanto lo que se muestra en pantalla aparece sin saltos.

- **Menuexc**: Este programa nos permite probar las excepciones de *overflow* y división por cero.

- **Menusem**: Este programa nos permite probar los semáforos. Nos solicita un número de semáforo, un valor para el mismo y una operación de subir o bajar. Con todo esto podemos comprobar cómo un proceso que quiere bajar un semáforo que no puede, se queda bloqueado, y también, a la contra, si sube un semáforo el cual tiene bloqueado a algún proceso, podemos ver cómo dicho proceso se desbloquea y continúa su ejecución.

- **Cross**: programa similar al errante pero con uso de semáforos. Este programa utiliza el valor 6 para inicializar dos semáforos que consultarán las instancias de este programa para determinar si las vueltas cuatro y ocho del circuito están ocupadas por algún proceso. Si se ejecutan varias instancias de este programa y una de ellas pone como velocidad '6' y las demás cualquier otro valor, entonces todas las ventanas empezaran a moverse realizando la espiral del circuito. Cuando cualquiera de ellas intenta entrar en la vuelta cuatro, realiza previamente la operación "bajar semáforo", si lo consigue, realiza la vuelta y al finalizarla, efectúa la operación "subir semáforo". Si durante el trayecto de dicha vuelta hubiera intentado entrar cualquier otro proceso, se habría quedado bloqueado en la operación bajar, y al salir el proceso que entró en la vuelta y suba el semáforo, provocará que el proceso bloqueado despierte y entre en la vuelta. Lo mismo sucede con la vuelta ocho usando otro semáforo.

- **Menumsj**: Programa demostración del paso de mensajes. El programa nos permite seleccionar un buzón (*B*), por defecto 0. Nos permite escribir un texto para mensaje (*M*), realizar una operación de envío (*E*) o recepción (*R*) de un mensaje, y/o terminar (*T*). Todo ellos nos permite probar el envío y recepción de mensajes a/de un buzón y también comprobar cómo se bloquea/desbloquea un proceso cuando se realizan dichas operaciones.

- **Tsfork**: Programa demostración de la llamada al sistema *fork*.

- **Sh** y **Sh-fork**: Estos programas son una *Shell* o intérprete de comandos en el espacio de usuario. Implementa muchos de los comandos de la consola de SO, aunque no todos. La diferencia entre ellos es que "Sh" ejecuta los programas utilizando la llamada "runFichero" y "Sh-fork" utiliza "fork". El comportamiento difiere en que "runFichero" ejecuta el proceso en una ventana nueva y propia y "*fork*" hereda la ventana del proceso padre y por tanto la comparte con él.

En la figura siguiente se muestran algunas de estas aplicaciones de usuario ejecutándose bajo el sistema 'SO'.

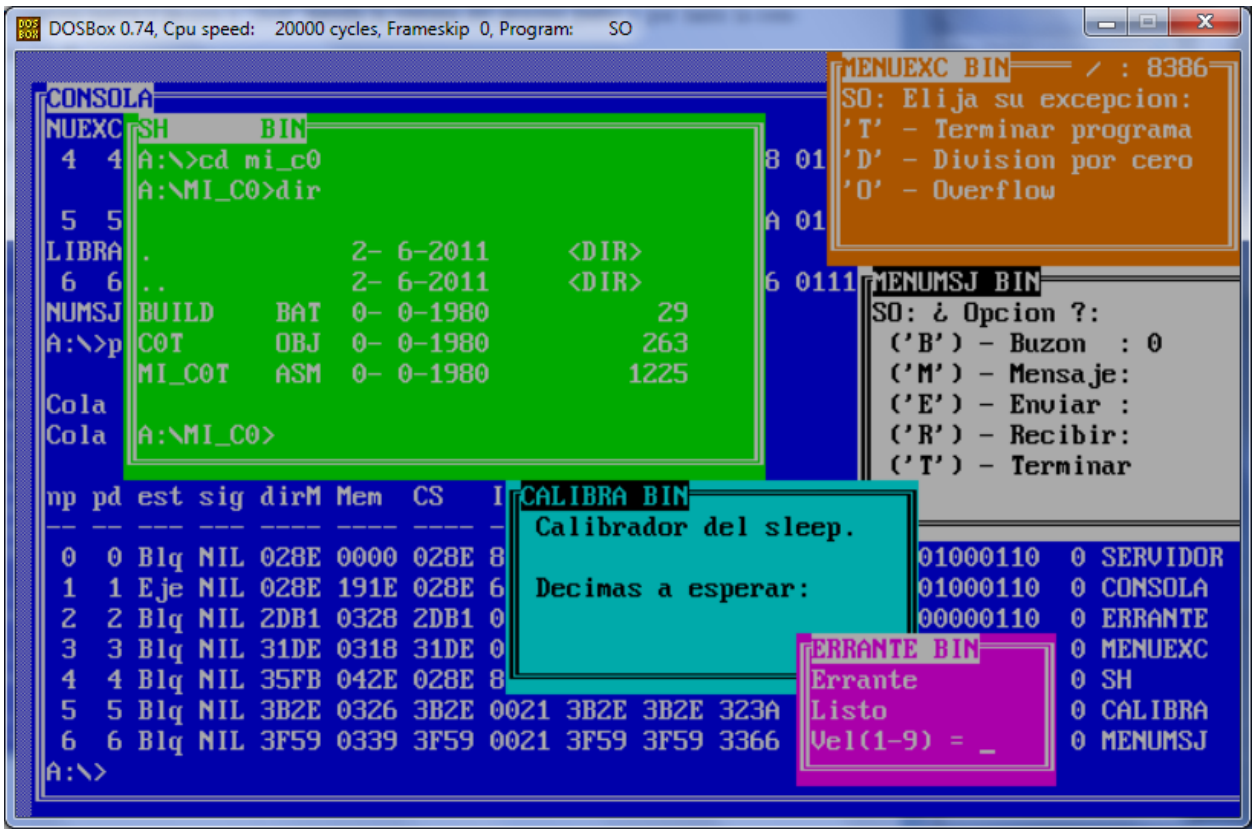


Figura 21: Ejecución de varias aplicaciones de usuario

6.3 Trabajos prácticos sobre el Sistema

En este apartado vamos a comentar brevemente algunas de las prácticas que se han desarrollado específicamente para este sistema operativo.

Práctica sobre tratamiento de interrupciones. El objetivo de esta práctica es que el estudiante sea capaz de programar una rutina de tratamiento de interrupciones. En este caso concreto se trata de la RTI correspondiente al vector '**1B**', asociado normalmente a la pulsación de la combinación de teclas Ctrl-Break, aunque en este S.O. particular será la combinación Ctrl-C. Esta RTI deberá "matar" inmediatamente todos los procesos vivos cuya ventana terminal sea la focal en el momento de la activación de la rutina (o sea, al pulsar Ctrl-C).

Práctica sobre manejo de excepciones. En esta práctica el estudiante aprenderá a tratar o manejar las excepciones. Concretamente se estudiarán las excepciones "división por 0" y "desbordamiento (*overflow*)". Esta práctica está muy relacionada con la anterior y es bastante similar salvo por los vectores de interrupción y el evento que los causa.

Práctica sobre la llamada "sleep". En esta práctica el alumno aprende a incorporar al sistema una nueva llamada al sistema, y en particular la conocida llamada "sleep" de Unix. Para llevar a cabo el trabajo el estudiante debe aprender a bloquear un proceso y despertarlo transcurrido un tiempo, para lo cual debe modificar las RTI's de servicio del sistema y del reloj. También aprende acerca de las librerías de interfaz de usuario de llamadas al sistema.

Práctica sobre semáforos. En esta práctica, ya con el conocimiento adquirido acerca de la incorporación de una nueva llamada al sistema, el estudiante aprende de forma práctica la implementación de las operaciones básicas sobre semáforos: "iniciar, subir y bajar".

Práctica sobre paso de mensajes. En esta práctica el estudiante profundiza en el conocimiento sobre la implementación de uno de los tipos de llamadas al sistema más complejos: el paso de mensajes. Tratar este tipo de llamadas tiene la ventaja de ilustrar la comunicación de datos entre los espacios de direcciones de un proceso de usuario a otro, pasando por el espacio del sistema operativo, algo muy interesante para un curso de sistemas operativos. El modelo seguido ha sido

el de una comunicación asíncrona, indirecta y simétrica, con buzones de capacidad limitada y mensajes de tamaño fijo.

Práctica sobre compactación de memoria. El objetivo de esta práctica es incorporar un comando nuevo a la "consola", el cual fusionará todos los huecos de memoria del sistema en uno sólo, eliminado completamente la fragmentación. Para conseguirlo, el estudiante tendrá que aprender a mover todos los objetos en memoria de forma transparente a los procesos, lo cual es posible gracias a que los procesos en este sistema admiten reubicación dinámica (con algunas restricciones). El estudiante deberá ser consciente de qué datos que mantiene el sistema sobre la ubicación de los objetos en memoria y de cómo modificar éstos para que reflejen su nueva ubicación.

Práctica sobre el proceso de arranque del sistema (boot). Esta práctica no trata sobre la mejora o modificación de ningún aspecto del sistema operativo, sin embargo, es fundamental entender el proceso de arranque en cualquier sistema operativo, y para ello, en este trabajo, el estudiante adquiere los conocimientos necesarios para realizar un arranque "en frío" del sistema, lo que además es válido para muchos otros sistemas.

7. CONCLUSIONES

En este capítulo se muestran: primero las conclusiones que se han obtenido como resultado de la realización de este proyecto y a continuación, las líneas de trabajo futuro a seguir para su ampliación y mejora.

7.1 Conclusiones finales.

Como primera conclusión podemos decir que se ha conseguido cumplir el objetivo principal del proyecto, consistente en la realización de un pequeño sistema operativo orientado a fines didácticos. El sistema conseguido es pequeño, manejable y de una relativa sencillez. El sistema tiene además entre sus cualidades, el haber demostrado que se pueden realizar unas "prácticas" para alumno, acerca de conceptos básicos e importantes en la comprensión de los "sistemas operativos". Por otro lado el sistema es suficientemente complejo y completo como para permitir la realización de nuevos trabajos prácticos dirigidos a la enseñanza, que abarquen muchos otros aspectos de los sistemas operativos de igual o incluso mayor relevancia.

Otra conclusión es que este tipo de proyectos, que en principio pueden ser abordables por una sola persona si se les limita suficientemente su funcionalidad, cuando se llevan a la práctica, es fácil que se tornen más complejos de lo previsto y se produzcan frecuentemente errores de análisis y de programación difíciles de depurar, con lo que los tiempos de desarrollo se incrementan en gran medida y de forma poco previsible. En estos sistemas es corriente que a medida que crece su tamaño y complejidad la introducción de nuevas características, o simplemente la corrección de los errores detectados, introduzca nuevos errores, aunque este hecho no ha llegado a ser muy preocupante en este sistema, dado que la complejidad del mismo se ha intentando mantener lo más baja posible y éste era uno de los objetivos principales del proyecto; no obstante; este indeseable efecto se ha hecho notar.

También podemos extraer otra conclusión relativa a las decisiones tomadas sobre el diseño del sistema operativo. Esta consiste en señalar la razón por la que no se ha buscado un diseño moderno o elegante, sino más bien uno anticuado como el "monolítico". Este último ha resultado ser el más sencillo de cara a su implementación y facilidad de depuración en un entorno cómodo como lo es el ofrecido por las herramientas de desarrollo escogidas. Es por ello por lo que se han sacrificado diseños más actuales, que aunque sin duda hubiesen sido mejores candidatos para realizar un sistema de uso más general, no lo han sido en este caso, pues como ya se ha dicho repetidas veces, el objetivo principal siempre fue el uso del sistema como herramienta de apoyo para la enseñanza de los fundamentos de los "sistemas operativos".

Por último, y también como conclusión, cabe señalar lo práctico que ha resultado ser el sistema una vez aplicado a la enseñanza gracias a las herramientas de trabajo requeridas, que son simples, de sencillo manejo, ampliamente disponibles y de coste muy económico o nulo en algunos casos, como son el emulador: "DOSBox" con entorno "D-Fend", el compilador "Turbo C", y otras varias de código abierto (*open source*). Todo ello ha permitido el montar unas prácticas en las que el esfuerzo de aprendizaje del alumno se ha focalizado principalmente en el conocimiento de los entresijos del sistema operativo y no de las herramientas empleadas. Por otro lado, hay que señalar que el desarrollo de las "prácticas" ha servido en gran medida para probar el sistema, ya que éstas han sido realizadas en la actualidad, no sólo por el autor sino también por muchos otros estudiantes, que en algunas ocasiones han aportado ideas, o simplemente han cometido errores que el sistema no tenía contemplados, y por ello han servido para que el autor efectuase cambios en forma de revisiones y/o mejoras del sistema.

7.2 Trabajos futuros de ampliación y mejora.

En principio habría que decir que en un proyecto como este, la lista de trabajos de ampliación y mejora a realizar en un futuro se podría hacer interminable, sin embargo, dado que el objetivo es mantener el sistema lo más simple posible, sólo se van a comentar algunas de las características que se podrían incorporar en un futuro al sistema sin que ello lo volviera demasiado complejo.

- Incorporar nuevas llamadas al sistema similares a las que ya ofrecen los sistemas UNIX como: wait, signal, sigaction, etc.
- Aumentar la lista de comandos internos.
- Gestión de ficheros especiales de dispositivo.
- Ofrecer posibilidad de redireccionar la entrada y la salida.
- Ofrecer comunicaciones al menos por el puerto serie y paralelo y deseablemente por red Ethernet.
- Incorporar el dispositivo ratón, bien mediante el puerto RS232 o PS/2.
- Gestión de ficheros especiales (dispositivos consola, impresora, rs232, etc.).
- Diseño del sistema para permitir que los drivers se instalen dinámicamente.
- Etc.

Otras ideas para la mejora del proyecto podrían incluir la realización de programas o herramientas de ayuda para la enseñanza que se apoyen en el sistema operativo, como por ejemplo un programa para la monitorización de procesos en ejecución. Este programa podría correr en una maquina física o virtual distinta y sería capaz de recibir información procedente del sistema 'SO" acerca de los procesos actualmente en ejecución a través de la línea de comunicaciones serie, paralelo o Ethernet; e iría mostrando un gráfico con un cronograma de los eventos que se producen en el sistema. De este modo el estudiante podría observar el comportamiento del sistema, viendo cómo evoluciona el estado de los procesos, los eventos que se producen, etc. De hecho este proyecto ya se encuentra implementado (no por el autor) y en fase de pruebas.

Por último y como una tarea quizá algo más compleja pero también muy interesante, sería la adaptación del sistema para corriese en el modo protegido del procesador 80386 y modelos superiores, para de este modo, crear así un sistema operativo que se pudiese beneficiar de las características avanzadas que ofrece dicho modo protegido.

7.3 Presupuesto

En este apartado se hace un pequeño análisis sobre los costes teóricos que ha requerido este proyecto fin de carrera.

En general, es corriente que previo a la realización de cualquier proyecto, se haya hecho antes un presupuesto para valorar su coste, sin embargo, para el caso particular de este proyecto fin de carrera, por diversas razones, no se ha hecho así, procediendo, a posteriori y en primer lugar, a efectuar una valoración que pretende ser más o menos objetiva, utilizando algunas de las herramientas disponibles en la actualidad para su medición y valoración; y en segundo lugar, se efectuara una valoración algo más subjetiva, y desde el punto de vista del autor.

7.3.1 Valoración mediante un modelo COCOMO.

Este modelo es quizá el más conocido, referenciado y documentado de todos los modelos de la estimación del esfuerzo de las actividades de diseño, codificación, pruebas y mantenimiento del *'software'*. Este modelo se creó a partir de la revisión de otros modelos existentes y con la participación de expertos en dirección de proyectos con experiencia en el uso de otros modelos de estimación. En un principio se creó bajo un único supuesto de modo de desarrollo, pero más tarde se consideró que era insuficiente y se amplió el supuesto a tres modos de desarrollo para intentar recoger mejor las diferencias entre la gran variedad de entornos de desarrollo. Estos tres modos de desarrollo son: **orgánico**, **semilibre** y **empotrado**. En el primero, el desarrollo del *software* se realiza en un entorno familiar y por un grupo reducido de programadores experimentados, yendo el tamaño del mismo desde unas pocas miles de líneas a algunas decenas de millar. El modo "semilibre" es un esquema intermedio en el orgánico y el empotrado; el grupo de desarrollo puede incluir una mezcla de personas experimentadas y no experimentadas. Por último, el modelo "empotrado" tiene fuertes restricciones, técnicas o funcionales. La resolución del problema es difícil pues puede ser único o no haber experiencia previa. Por otro lado, atendiendo al nivel de detalle empleado en el modo de desarrollo se distinguen a su vez tres modelos: **básico**, **intermedio** y **detallado**. En el modelo "básico" el esfuerzo de desarrollo es función del tamaño estimado del software en líneas de código, lo que es adecuado para realizar estimaciones de forma rápida pero sin gran precisión. En el modelo "intermedio", el esfuerzo es función del tamaño del producto, modificando los atributos directores del coste, los cuales incluyen una valoración

subjetiva del producto, *hardware*, personal, etc. Por último el modelo "detallado" tiene en cuenta la valoración de los atributos en cada una de las fases de desarrollo del proyecto.

Para la estimación objetiva de este proyecto se ha utilizado el modo de desarrollo "orgánico" siguiendo el modelo "básico". En primer lugar se ha utilizado un programa de *software* abierto llamado "**sloccount**". Este programa dispone de muchas opciones de ejecución, pero se ha limitado su funcionamiento al análisis del directorio donde se encuentran los fuentes del proyecto, ya que es la opción más sencilla y directa. Tras ejecutar dicha utilidad se obtiene el siguiente resultado resumido.

```
Computing results.

SLOC Directory SLOC-by-Language (Sorted)
3108    top_dir          cpp=3108
632     USRS_PRG         cpp=632
333     LL_S_SO          cpp=333
0       MI_C0            (none)

Totals grouped by language (dominant language first):
cpp:           4073 (100.00%)

Total Physical Source Lines of Code (SLOC)                = 4,073
Development Effort Estimate, Person-Years (Person-Months) = 0.87 (10.49)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.51 (6.11)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 1.72
Total Estimated Cost to Develop                           = $ 118,046
```

Figura 22: Resultado de estimación del coste por "sloccount"

En dicho resultado se puede ver que el esfuerzo estimado en hombres/mes es de 10,49. El cálculo se detalla en el propio resumen, aunque hay que precisar que '*KSLOC*' es el número de líneas de código (*SLOC*) expresado en miles, es este caso, 4,073, o sea: $2{,}4\ x\ 4{,}073^{\,1{,}05} = 10{,}49.$ El tiempo de desarrollo del proyecto es: $2{,}5\ x\ 10{,}49^{0{,}38} = 6{,}11$ meses. Las personas necesarias para su desarrollo son: 10,49 / 6,11 = 1,72, y el coste total del proyecto tomando como salario mensual el que utiliza el programa que es de: 50.286 /12 = 4.690,5 $, será el resultado de la ope-

ración: 4690,5 *x* 6,11 *x* 1,72 *x* 2,4 ≈ **118.046** $, donde 2,4 es la sobrecarga que introduce "sloccount" debida al mantenimiento del proyecto.

El programa "sloccount" tiene ciertas limitaciones al contabilizar las líneas de código, por ejemplo sólo contabiliza líneas de código físicas, no lógicas. Otros programas en cambio si lo hacen. Entre ellos se ha utilizado el programa "ucc release 2011.03", (Unified CodeCount)" para al menos disponer de una alternativa a la medición.

A continuación se muestra sólo un resumen del resultado obtenido mediante dicho programa. En él se observa que el número de líneas físicas es diferente del obtenido con "sloccount", aunque la diferencia no es demasiado grande. Tras comprobar manualmente algunas de las diferencias entre ellos en el resultado detallado por ficheros, parece ser que este último es más preciso, por lo que se reajusta el cálculo mediante las mismas fórmulas y utilizando las líneas lógicas en vez de las fisicas, se obtendría un esfuerzo hombre/mes de $2{,}4 \times 4{,}555^{1{,}05} = 11{,}79$, y por tanto el coste total sería de 132.722 $. Por otro lado, si tomamos como salario anual para un analista programador con experiencia, en la actualidad y en España, el valor de 50.400 €, obtenido tras haber consultado diversas fuentes, el coste total del proyecto sería: 4.200 x 11,79 x 2.4 = **118.843 €**

USC Unified CodeCount (UCC)
(c) Copyright 1998 - 2011 University of Southern California
SLOC COUNT RESULTS
Generated by UCC v.2011.03 on 4 15 2012
RESULTS SUMMARY

Total Lines	Blank Lines	Comments		Compiler Direct.	Data Decl.	Exec. Instr.	Number of Files	SLOC	File Type	SLOC Definition
		Whole	Embedded							
6819	674	1695	1417	336	402	3712	48	4450	CODE	Physical
6819	674	1695	1417	336	353	3866	48	**4555**	CODE	Logical

Figura 23: Recuento de líneas de código según UCC

Como ya se ha comentado, las constantes utilizadas en estos cálculos podrían modificarse ya que dependen del modo y modelo utilizados, si se desea conocer más sobre estos valores y otras opciones de configuración del coste se puede consultar en la guía de usuario de la documentación del programa "sloccount", la cual se puede encontrar en: http://www.dwheeler.com/sloccount. En dicha guía se detalla en cierta medida el modelo COCOMO. Otras páginas con más información al respecto son:

http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html y
http:// www.sc.ehu.es/jiwdocoj/mmis/cocomo.htm.

Al coste del proyecto visto hasta ahora habría que añadirle los gastos en material tanto software como hardware que se han requerido para su realización. En lo referente al hardware, lo único requerido ha sido un ordenador personal del tipo medio que se podría valorar en torno a los 750€, y en lo referente al software, no haría falta añadir ningún coste, pues todo el software empleado es del tipo "*freeware"* o similar, es decir de coste 0.

7.3.2 Valoración subjetiva del proyecto.

En el punto anterior hemos visto una estimación más o menos objetiva y simplificada utilizando herramientas de medición de líneas de código, sin embargo dado que la valoración del proyecto se ha efectuado una vez finalizado éste, en principio sería posible saber con bastante precisión la cantidad de tiempo empleado por el autor en el desarrollo del mismo, no obstante, dada la forma en la que se ha llevado a cabo el proyecto, que ha sido poco uniforme en el tiempo, y también debido a la falta de contabilidad del tiempo empleado en el desarrollo del mismo, no se dispone del tiempo real empleado, aunque siempre es posible considerar de forma algo subjetiva el tiempo que el autor estima que ha empleado en el proyecto. La primera versión registrada en la gestión del proyecto gestionada mediante el software "git" es de febrero de 2010, pero con seguridad dicha versión ya había requerido bastante trabajo antes de disponer de ella. Al menos el trabajo se inició seis meses antes. Posteriormente el sistema se fue ampliando y mejorando, invirtiendo aproximadamente hasta el día de hoy, unos dos años, aunque hay que decir que se estuvo trabajando todo el tiempo en el proyecto, ya que como ya se ha dicho, el desarrollo ha sido bastante irregular, con períodos intensos de trabajo y otros de total inactividad, no obstante, la apreciación subjetiva del autor sobre el tiempo empleado por día laborable es de unas 3 horas de media, lo que supondrían un porcentaje de un 37% sobre una jornada aproximada de 8 horas al día. Si consideramos como tiempo total de desarrollo del proyecto dos años y medio y aplicamos el porcentaje arriba calculado y aplicando el salario anual ya visto de 50.400 €, obtendremos un coste de: 50.400 x 2,5 x 0,37 = 46.620 €, antes de aplicar la sobrecarga de 2.4 puntos debidos al mantenimiento, tal y como ya se aplicó en el punto anterior, en la valoración objetiva. Una vez añadida dicha sobrecarga para una mejor comparación entre ambas valoraciones tendremos un costeo de trabajo de: 2.4 x 46.620 = **111.888 €**, lo cual no está muy alejado de la valoración anterior.

A continuación se muestra una pequeña tabla de la valoración del proyecto sin considerar el coeficiente 2'4 de factor de mantenimiento que aplica COCOMO. Tampoco se han añadido al presupuesto un porcentaje de beneficios por considerarse que éste será de código abierto:

Concepto	Importe (€)
Recursos Humanos…	47.250 €
Ordenador Personal..	750 €
Software (abierto)…	0 €
Total presupuesto…..	48.000 €

Tabla 2: Valoración subjetiva del proyecto

Sería bueno también añadir, que además del desarrollo del proyecto consistente en la implementación de un sistema operativo de tamaño reducido y orientado a la enseñanza, el autor también ha invertido un tiempo en desarrollar unas "prácticas" para ser realizados por los estudiantes. Éstas se adjuntan como apéndices al proyecto y no figuran en la estimación de coste por no incluir en ellas ninguna línea de código, puesto que las que resultarían de la realización de las mismas, ya se encuentran incluidas en el código del proyecto (aunque el trabajo de documentación de las mismas no es despreciable y habría que ser tenido en cuenta).

Por último señalar que para realizar este proyecto, ha sido necesario estudiar con bastante detalle muchos aspectos acerca del funcionamiento del compilador Turbo C y MS-DOS. Incluso ha habido que modificarse el código de inicialización estándar de Turbo C, para así poder conseguir que el sistema funcione tanto en modo independiente, como en modo anfitrión bajo MS-DOS y Turbo C. Con todo ello se ha conseguido que el estudiante pueda usar el entorno integrado de dicho compilador, y gracias a ello, beneficiarse de la edición, compilación y depuración que el entorno ofrece, logrando así, uno de los objetivos principales del proyecto.

8. REFERENCIAS

8.1 Bibliografía

[MICR91] Microsoft MS-DOS Programmer's Reference
Autor: Microsoft
Editorial: Microsoft Press, 1991. ISBN: 1-55615-329-5

[TAWO98] Sistemas Operativos: Diseño e implementación
Andrew S. Tanenbaum y Albert S. Woodhill
Prentice Hall, 1998. ISBN 970-17-0165-8

[TANE09] Sistemas Operativos Modernos
Autor: Andrew S. Tanenbaum
Editorial: Pearson Educación, 2009. ISBN 978-607-442-046-3

[CARR01] Sistemas Operativos. Una visión aplicada
Autores: Jesús Carretero Pérez, Pedro Miguel Anasagasti
Félix García Carballeria, Fernando Pérez Costoya
Editorial: McGraw-Hill/Interamericana de España S.A.U. ISBN: 84-481-3001-4

[SARW03] Linux: El libro de texto
Autores: Sarwar S. M., Koretsky R., Sarwar S.A.
Editorial: Pearson Educación, 2003. ISBN: 84-7829-060-5

[BARR94] Software Engineering Economics
Autor: Barry W. Bohem
Editorial: Prentice-Hall, 1994. ISBN: 0138221227

[SOMM05]Ingeniería del Software, 7ª Edición
Autor: Ian Sommerville
Editorial: Pearson Educación, 2005. ISBN: 84-7829-074-5

[CUEV02] Gestión del proceso software
Autor: Cuevas Agustín, Gonzalo
Editorial: C. Estudio Ramón Areces, 2002. ISBN: 8480045469, ISBN-13: 9788480045469

Referencias bibliográficas adicionales sobre Sistemas Operativos

Libro de prácticas de Sistemas Operativos
Autor: Jesús Carretero Pérez, Félix García Carballeira, Fernando Pérez Costoya
Editorial: McGraw-Hill, 2002. ISBN: 84-481-3662-4

Fundamentos de Sistemas Operativos (7ª Edicion)
Autor: A. Silberschatz, P. Galván y G. Gagne
Editorial: McGraw-Hill, 2006. ISBN: 84-481-4641-7

Sistemas Operativos. Aspectos internos y principios de diseño
Autor: William Stallings
Editorial: Pearson Educacion, 2005. ISBN: 978-84-205-4462-5

Sistemas Operativos. Conceptos y Diseño
Autor: Milan Milenkovic
Editorial: McGraw-Hill, 1994. ISBN: 978-84-481-1871-6

El entorno de programación UNIX
Autor: Brian. W. Kerninghan y Rob.Pike
Editorial: Prentice Hall, 1987. ISBN 968-880-067-8

The Design of the Unix Operating System
Autor: Maurice J. Batch
Editorial: Prentice Hall, 1986. ISBN 0132017997 (ISBN13: 9780132017992)

8.2 Enlaces URL:

Todas las páginas a la que se accede mediante los enlaces URL relacionados a continuación estaban vigentes y activas a fecha **3 Mayo 2012**.

http://www.minix3.org

http://minix1.woodhull.com

http://minix1.woodhull.com/index1.html (inicialmente: minix1.hampshire.edu)

http://www.linux-es.org/sobre_linux

http://www.monografias.com/trabajos14/linux/linux.shtml

http://es.wikipedia.org/wiki/Historia_de_Linux

http://mikeos.berlios.de

http://www.menuetos.net

http://geekos.sourceforge.net

http://www.fiwix.org

http://minirighi.sourceforge.net

http://www.sc.ehu.es/jiwdocoj/mmis/cocomo.htm

http://es.wikipedia.org/wiki/Proceso_para_el_desarrollo_de_software

http://git-scm.com

---

http://www.top4download.com/turbo-c-/aklqwuba.html (Descarga de Turbo C ++ v3.0)

http://dfendreloaded.sourceforge.net/Download.html (Descarga D-Fend)

---------- Este proyecto sourceforge y uno similar ----------

http://www.sop1.com

http://sourceforge.net/projects/so1/files/SO-0.1.2

ANEXOS

Junto con esta memoria se adjunta un CD en el que se encuentra la memoria del proyecto en el directorio raíz y todo el software utilizado en el proyecto listo para ser usado, una vez copiado a un dispositivo de almacenamiento con permiso de escritura en el directorio 'D-Fend'. En dicho directorio se encuentran también todas las fuentes del proyecto, junto con el fichero ejecutable para MS-DOS y la imagen binaria (.bin) para su instalación en un dispositivo de arranque. En un directorio aparte, se encuentra también la documentación sobre las "prácticas" que se han diseñado para ser realizadas por el estudiante sobre este sistema.

La jerarquía de directorios y contenido de los mismos es el siguiente:

Raíz

- ProyectoFC.docx *(memoria del PFC en formato 'Word')*
- ProyectoFC.doc *(memoria del PFC en formato 'Word')*
- ProyectoFC.pdf *(memoria del PFC en formato 'pdf')*
- PresentacionPFC.pptx *(presentación del PFC en formato 'Powerpoint)*
- PresentacionPFC.ppt *(presentación del PFC en formato 'Powerpoint)*
- PresentacionPFC.pdf *(presentación del PFC en formato 'pdf')*
- **D-Fend** *(Software interfaz gráfico de DOSBox)*
  - mapper.txt

```
├─ Dfend.exe      (Ejecutable entorno gráfico D-Fend)
├─ Defend.dat
├─ AutoSetup      (carpeta interna de D-Fend)
├─ Bin            (carpeta interna de D-Fend)
├─ Capture        (carpeta interna de D-Fend)
├─ Confs          (carpeta interna de D-Fend)
├─ DOSBox         (carpeta interna de D-Fend con emulador DOSBox)
├─ IconLibrary    (carpeta interna de D-Fend)
├─ IconSets       (carpeta interna de D-Fend)
├─ Lang           (carpeta interna de D-Fend)
├─ Settings       (carpeta interna de D-Fend)
├─ Templates      (carpeta interna de D-Fend)
└─ VirtualHD      (carpeta disco duro para la maquina virtual)
   │
   ├─ FreeDos     (carpeta comandos FreeDos)
   ├─ CN          (carpeta con un intérprete de comandos amigable para DOS)
   ├─ TC30        (carpeta compilador Turbo C 3.0 ++)
   ├─ TASM        (carpeta Turbo Assembler [no es estrictamente necesaria])
   ├─ InfoTech    (carpeta información técnica PC, BIOS, etc.)
   ├─ Utils       (carpeta con algunas utilidades)
   ├─ IMGS4SO     (carpeta con algunas imágenes de disco y boot para SO)
   └─ MiSO        (Proyecto Mi Sistema Operativo)
      │
      ├─ SO.exe          (fichero ejecutable Sistema Operativo para MS-DOS)
      ├─ SO60.bin        (imagen binaria Sistema Operativo para hacer boot)
      ├─ Ficheros.c      (módulo para el sistema de ficheros FAT)
      ├─ Llamadas.c      (módulo para las llamadas al sistema)
      ├─ Memoria.c       (módulo para la gestión de memoria)
      ├─ Procesos.c      (módulo para la gestión de procesos)
      ├─ RTIcomun.c      (código común a las RTI's)
      ├─ SO.c            (módulo de inicio y consola del S.O.)
      ├─ Teclado.c       (módulo para tratamiento del teclado)
      ├─ TeclaInt.c      (módulo RTI del teclado)
      ├─ TimerInt.c      (módulo RTI del reloj)
      ├─ Windows.c       (módulo para la gestión de ventanas)
      ├─ SO.dsk          (configuración escritorio de Turbo C)
      ├─ .gitignore      (configuración 'ignore' para el software 'git')
      ├─ Ficheros.h      "          "          "
      ├─ Llamadas.h      "          "          "
      ├─ Memoria.h       "          "          "
      ├─ Procesos.h      "          "          "
      ├─ RTIcomun.h      "          "          "
      ├─ Teclado.h       "          "          "
      ├─ TeclaInt.h      "          "          "
      ├─ TimerInt.h      "          "          "
      ├─ Tipos.h         "          "          "
      ├─ Windows.h       "          "          "
      ├─ SO.map          (config. map proyecto SO de Turbo C)
      ├─ 2Bin.bat        (comando batch para convertir 'exe' a 'bin')
      ├─ SO.prj          (config. prj proyecto SO de Turbo C)
      ├─ Ayuda.txt       (auxilar de S.O. para mostrar ayuda)
      ├─ Leeme.txt       (notas internas sobre SO)
      ├─ .git            (carpeta del proyecto que mantiene el software 'git')
      ├─ LL_S_SO         (carpeta con la liberaría de rutinas de interfaz)
      │  │
      │  ├─ Bsic-Ifz.c   (rutinas de interfaz básico)
      │  ├─ Fich-Ifz.c   (rutinas de interfaz de ficheros)
      │  └─ Inic-Usrc.c  (código inicial obligatorio proceso. usuario)
```

```
        ├ Memo-Ifz.c    (rutinas de interfaz de memoria)
        ├ Proc-Ifz.c    (rutinas de interfaz de procesos)
        ├ Test-Ifz.c    (rutinas de interfaz para pruebas)
        ├ Bsic-Ifz.h
        ├ Fich-Ifz.h
        ├ Memo-Ifz.h
        ├ Proc-Ifz.h
        └ Test-Ifz.h

    ├ Mi_C0         (carpeta con código init en asm para Turbo C)
    ├ Disquete      (carpeta con los binarios de los programas de ususario)
    └ Usrs_Prg      (carpeta con programas de usuario)

├ Prácticas
    ├ Practica-1.doc    (boot loader)
    ├ Practica-2.doc    (Interrupciones, Excepciones y llamadas)
    ├ Practica-3.doc    (paso de mensajes)
    └ Practica-4.doc    (compactación de memoria)

├ Git-1.7.7
└ CodeCounters
    ├ slocdata          (carpeta resultados 'sloccount')
    ├ outfile_cplx.csv  (fichero resultados 'ucc')
    ├ sloccount-2.26-1.i386.rpm
    └ ucc_2011.03.rar
```

APÉNDICE I. Práctica: Arranque del Sistema.

ÍNDICE

1) OBJETIVOS

Los objetivos de esta práctica son:

- Que el alumno conozca el nivel preciso sobre el que se implementa el sistema operativo de un ordenador real (es decir inmediatamente por encima del nivel de lenguaje máquina y de la BIOS).
- Que el alumno sea capaz de implementar el arranque de un sistema operativo desde un disquete estándar utilizando programación a bajo nivel en C.

2) INTRODUCCIÓN

Al encenderse el ordenador la CPU empieza a ejecutar instrucciones en memoria (ROM) a partir de una dirección prefijada. En la ROM toma el control lo que se denomina la BIOS (Sistema de Entrada/Salida Básico) que es una especie de rudimentario sistema operativo proporcionado por el fabricante del ordenador para poder operar con los dispositivos de E/S de una forma estandarizada. Gracias a que cada fabricante incorpora su propio BIOS a sus modelos clónicos del IBM PC, ofreciendo todos ellos la misma interfaz de llamadas (al sistema BIOS), es posible que un mismo programa funcione correctamente en cualquiera de esos ordenadores, a pesar de contar con dispositivos periféricos que pueden ser completamente diferentes, y sin tener que hacer un reconocimiento exhaustivo del hardware del ordenador.

La BIOS, una vez que toma el control, reconoce e inicializa los controladores de los dispositivos más usuales (teclado, tarjeta de vídeo, controladores de disquete y disco duro, etc.) estableciendo los vectores de interrupción (en la RAM) con las direcciones de las rutinas de tratamiento de interrupción correspondientes (residentes en la ROM). A continuación la BIOS intenta cargar el sistema operativo desde alguno de los dispositivos de E/S que ha reconocido (disquetera, cdrom, disco duro, usb o tarjeta de red). En caso de que ninguno de esos dispositivos aporte el sistema operativo, la BIOS simplemente se queda bloqueado.

En el caso de la disquetera, la BIOS espera que tenga introducido un disquete cuyo primer sector lógico (el sector lógico 0 que está en el sector físico 1, cabeza 0 y pista 0) tenga en algunos de sus 512 bytes ciertos valores. Si tras leer el sector lógico 0 la BIOS encuentra esos valores correctamente establecidos, deduce que ese sector contiene además el código del sistema operativo que se ocupa de realizar su carga desde el disquete, por lo que le cede el control, dando por terminado su cometido.

El código contenido en el sector de arranque tiene en cuenta hasta cierto punto la estructura del sistema de ficheros existente en el disquete (sector de arranque, sectores reservados, la(s) FAT(s), el directorio raíz y los sectores de datos) para realizar la carga del sistema operativo en la memoria RAM, ubicándolo en una zona apropiada para su ejecución. Finalmente el código del sector de arranque cede el control al sistema operativo ya cargado en la memoria.

En la programación del sector de arranque se manejan ya a un nivel elemental todos los conceptos básicos de un sistema operativo: la activación de diferentes programas, la gestión de la memoria, la reubicación de código, el manejo de los dispositivos de E/S, las llamadas al sistema y el

sistema de ficheros. Por ese motivo este tipo de ejercicios de programación juega un papel muy importante en un primer curso de sistemas operativos.

3) TRABAJO A REALIZAR

La tarea a desarrollar en este tutorial es justamente la programación en C (con recursos de bajo nivel) del sector de arranque de un disquete de 3 ½ pulgadas de doble cara y doble densidad (1,44 MBytes) cumpliendo las siguientes **especificaciones**:

1) Por supuesto, la BIOS debe reconocer el disquete donde se escriba ese sector de arranque como un disquete correcto con un sector de arranque correcto. Aquí juegan un papel fundamental los bytes 510 y 511 del sector de arranque (es decir la signatura del sector de arranque `55 AA`). Además MS-DOS debe reconocer también el disquete como un disquete correcto con un sistema de ficheros FAT12 correcto, de modo que por ejemplo el comando **`dir a:`** se ejecute sin problemas.

2) Una vez que la BIOS ceda el control al código del sector de arranque, ese código deberá realizar la carga de lo que haya en los primeros 320 sectores lógicos consecutivos siguientes al espacio ocupado por el directorio raíz del disquete. Tras la lectura de cada uno de esos sectores debe escribirse en la pantalla un carácter `'.'`, con el fin de poder seguir el avance de la carga de los 320 sectores.

3) La carga de los 320 sectores mencionados debe tener como destino final las 160 K direcciones de memoria consecutivas que están a partir del tercer KByte de la memoria principal (dirección de comienzo 000600H).

4) La carga de los 320 sectores debe respetar las siguientes zonas de memoria: la tabla de vectores de interrupción (de 000000H a 0003FFH), el área de trabajo de la BIOS (de 000400H a 0004FFH) y las direcciones utilizadas por los controladores y la ROM (de 0A0000H a 0FFFFFH). Dicho de otro modo, durante la carga no debe escribirse en ninguna de las zonas de memoria mencionadas.

Es natural que la especificación anterior plantee al alumno muchas dudas sobre multitud de pequeños detalles que significan una laboriosa tarea de documentación. Por ese motivo vamos a hacer a partir de aquí una exposición pormenorizada de las tareas que es necesario que el alumno lleve a cabo. Dichas tareas son las siguientes:

Actividad 1: Entender cómo debe funcionar el disquete de arranque

Actividad 2: Entender la generación de código necesaria

Actividad 3: Esqueleto para el sector de arranque

Actividad 4: Llamadas a la BIOS de manejo de teclado y pantalla

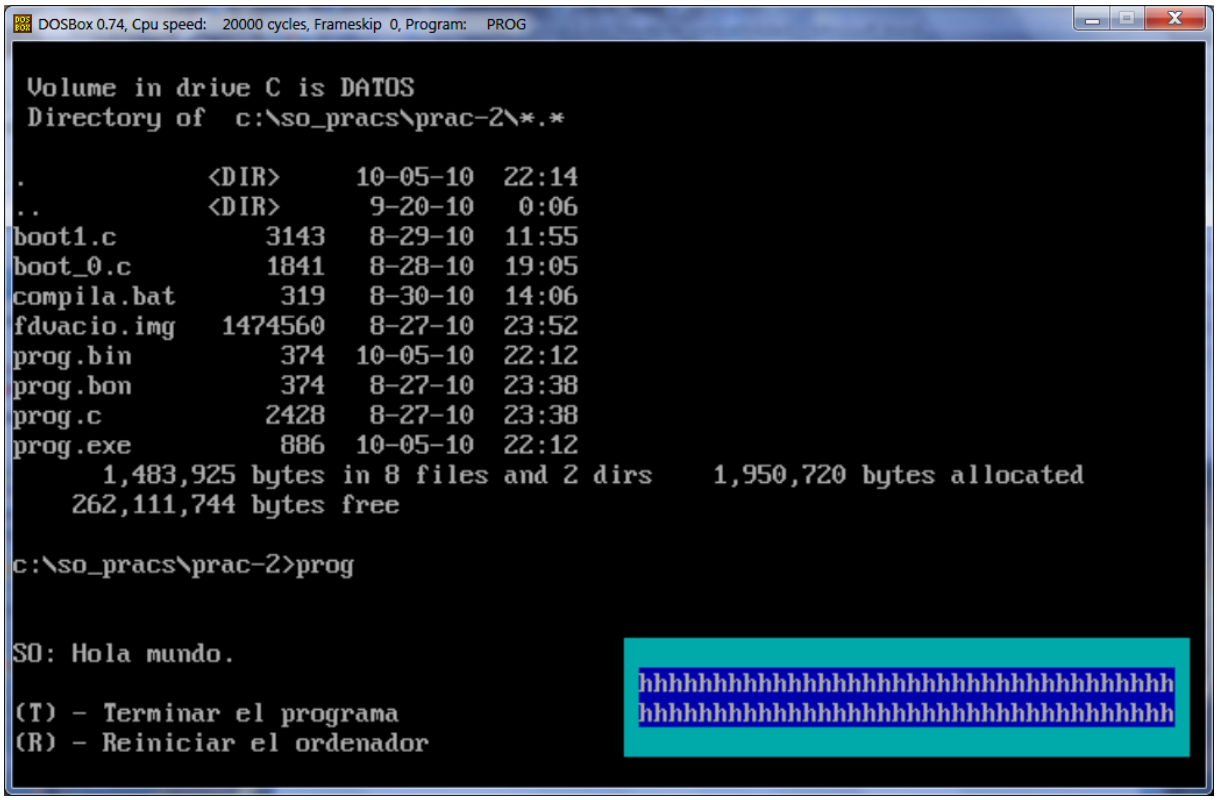
Actividad 5: Llamadas a la BIOS de acceso al disco

Actividad 6: Manejo de direcciones de memoria

Actividad 7: Implementación final del sector de arranque

4) ACTIVIDAD 1: Entender cómo debe funcionar el sector de arranque.

Para entender lo que se está pidiendo no hay mejor manera que ver un sector de arranque ya hecho que cumpla las especificaciones del apartado anterior. En el directorio c:\imgs4so se encuentra el fichero "mi_boot.bin" que contiene el código de sector de arranque que cumple las especificaciones del ejercicio. Para comprobar esto, vamos a utilizar un programa que haga las veces de S.O. para ser arrancado desde disquete con dicho cargador (boot loader). En el directorio de la práctica, "c:\practi.cas\prac-2" se encuentra el programa "prog.c", que una vez compilado (mediante el comando batch: compila progc) y ser ejecutado (programa generado: prog.exe) bajo MS-DOS, se mostraría la siguiente pantalla:



Como puede verse el programa "prog.exe" muestra en pantalla un saludo con menú, donde las opciones son: terminar el programa (volviendo a MS-DOS) o reiniciar el ordenador. En la esquina inferior derecha de la pantalla se muestra además un rectángulo de color azul claro cuyo interior se rellena con el último carácter que se haya tecleado.

El programa anterior no es, ni pretende ser, un sistema operativo, pero nos permitirá comprobar el buen funcionamiento del sector de arranque contenido en `"mi_boot.bin"`, a la vez que permitirá que el alumno se dé cuenta de cómo es la programación en el nivel más básico de la implementación de un sistema operativo.

El primer problema que se nos presenta es que el fichero ejecutable `"prog.exe"` no comienza directamente con la primera instrucción a ejecutar, sino que comienza con una cabecera de 512 bytes cuyo formato puede consultarse con el programa "*techelp*" y de la que podemos obtener fácilmente su información si tecleamos:

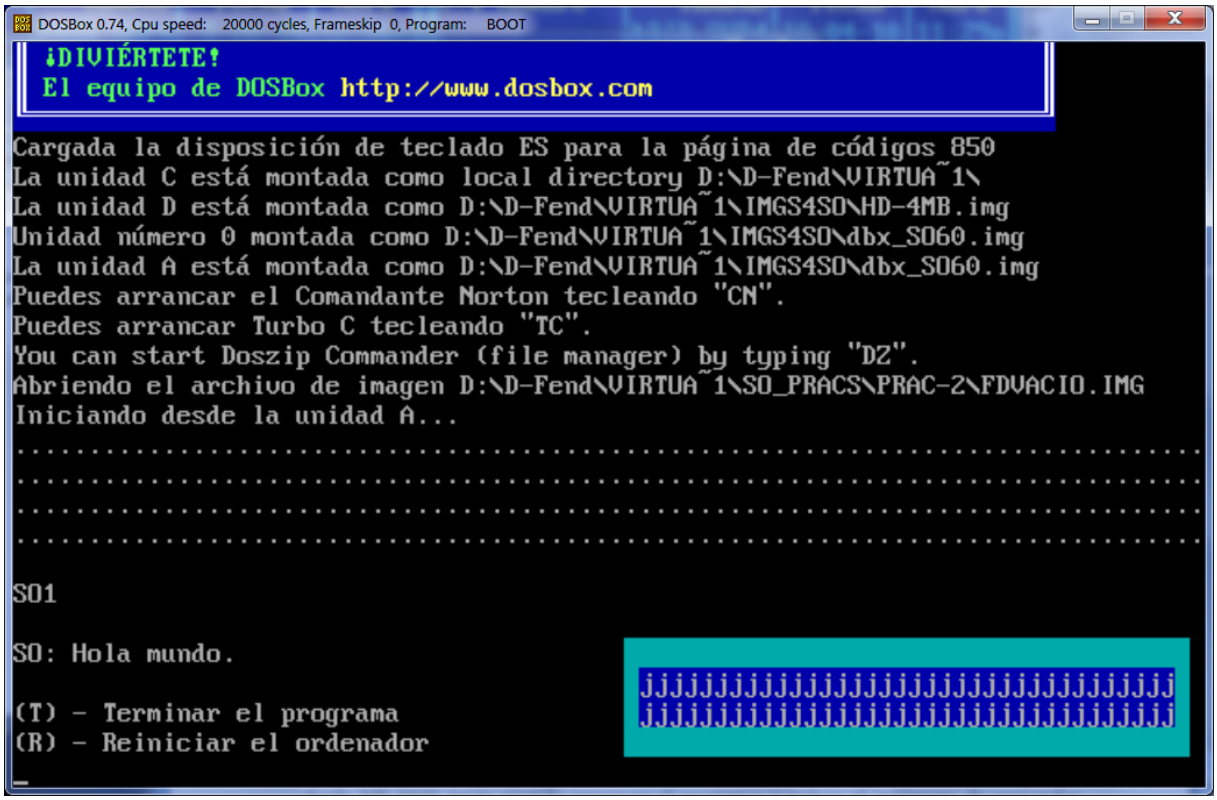
C:\practi.cas\prac-2\exe2bin /h prog.exe

Esta cabecera del fichero es algo necesario para MS-DOS, pero para nuestro sistema, no debe estar en los programas generados, por tanto debe ser suprimida. Existen muchos métodos para hacer esto. En nuestro caso utilizaremos el programa conversor de formato "*exe2bin*", incluido en el directorio "c:\utils". La supresión de la cabecera se realiza de un modo transparente, ya que en el comando "*compila.bat*" que se encuentra en el directorio de la práctica, se encuentra ya incluida la línea que invoca a dicho programa para llevar a cabo tal propósito. Si dentro del directorio "*c:\practi.cas\prac-2",* ejecutamos el comando: *compila prog*, se generarán los ficheros ***prog.exe*** y *prog.bin*. Este último es la versión compilada de *prog.c*, sin la cabecera. Se verá que el tamaño es justo 512 bytes menos que la versión "exe".

Una vez obtenido el programa `"prog.bin"` por el método que sea, tendríamos ya todos los elementos necesarios para construir nuestro disquete de arranque. Lo primero es dar formato a una imagen de disquete (***Se recuerda que la imagen debe estar montada como unidad física [0 ó 1] ya que DosBox usa también unidades lógicas que no admiten el comando tradicional del DOS "format". Para dicho propósito puede montarse por ejemplo la imagen de disquete del direcotorio de la práctica "fdvacio.img"***). Por rapidez se recomienda utilizar el comando "formatea.bat" que se encuentra en el directorio C:\practi.cas". Este comando es muy simple, únicamente escribe el fichero "formato.bin" en los primeros sectores de un disquete, utilizando el programa "escr_s" ya conocido de la práctica anterior. El fichero escrito, formato.bin, es una imagen binaria de un sistema de ficheros Fat12 de un disquete vacío. Con esto conseguimos tener un disquete libre de información, aunque sobre el resto de sectores no se escribe nada, y no valdría para borrar toda la información de un disquete, pero eso es algo que no nos preocupa.

El siguiente paso sería copiar el fichero `"mi_boot.bin"` (que se encuentra en el directorio c:\imgs4so) en el sector de arranque del disquete (sector 0), utilizando el programa "escr_s" ya visto. Por último, hay que copiar el fichero "prog.bin" en el disquete. *Es crucial que este fichero sea el primer fichero que se copia en el disquete*, ya que así se garantiza que ocupe los primeros sectores del disco (a partir del 33, que es el primer sector de datos tras el directorio raíz) y el disquete pueda arrancar.

Sólo nos falta comprobar que el disquete que hemos preparado arranca (hace *boot*) el programa `"prog.bin"` escrito en él. Con este fin, seleccionamos con "*D-fend*" alguno de los perfiles ya existentes (también se podría crea uno nuevo para tal efecto), y vamos haciendo "click" sobre el botón "editar", opción "Al arrancar" (abajo izquierda), "arrancar imagen de disquete" y seleccionamos nuestra imagen: por ejemplo "fdvacia.img". El resultado es el que se muestra en la siguiente imagen de pantalla:



El alumno debe comprobar que el programa lee y visualiza correctamente los caracteres que se introducen por el teclado, y que pulsando la tecla `'R'` (mayúscula o minúscula) se reinicia el ordenador. Igualmente debe comprobar que pulsando la tecla `'T'` el sistema no funciona. La razón es porque la operación de terminar el programa no tiene ningún sentido en este momento al ser `"prog.bin"` el único programa existente en el sistema. Cuando ejecutábamos `"prog.exe"` desde MS-DOS tenía sentido terminar el programa y volver al intérprete de comandos (haciendo una llamada al sistema operativo MS-DOS), pero ahora el único "sistema operativo" es el propio `"prog.bin"` quien tiene que resolver todos los problemas por sí mismo.

Esa situación de desamparo se da exactamente igual en el código de arranque contenido en `"mi_boot.bin"`. Ese código tiene que leer sectores del disquete que lo contiene, tiene que cargar en memoria dichos sectores, tiene que escribir en la pantalla y tiene que ceder el control al programa una vez cargado. Para hacer todo eso sólo dispone de su propio código máquina y de la BIOS presente en la ROM.

5) ACTIVIDAD 2: Entender la generación de código necesaria

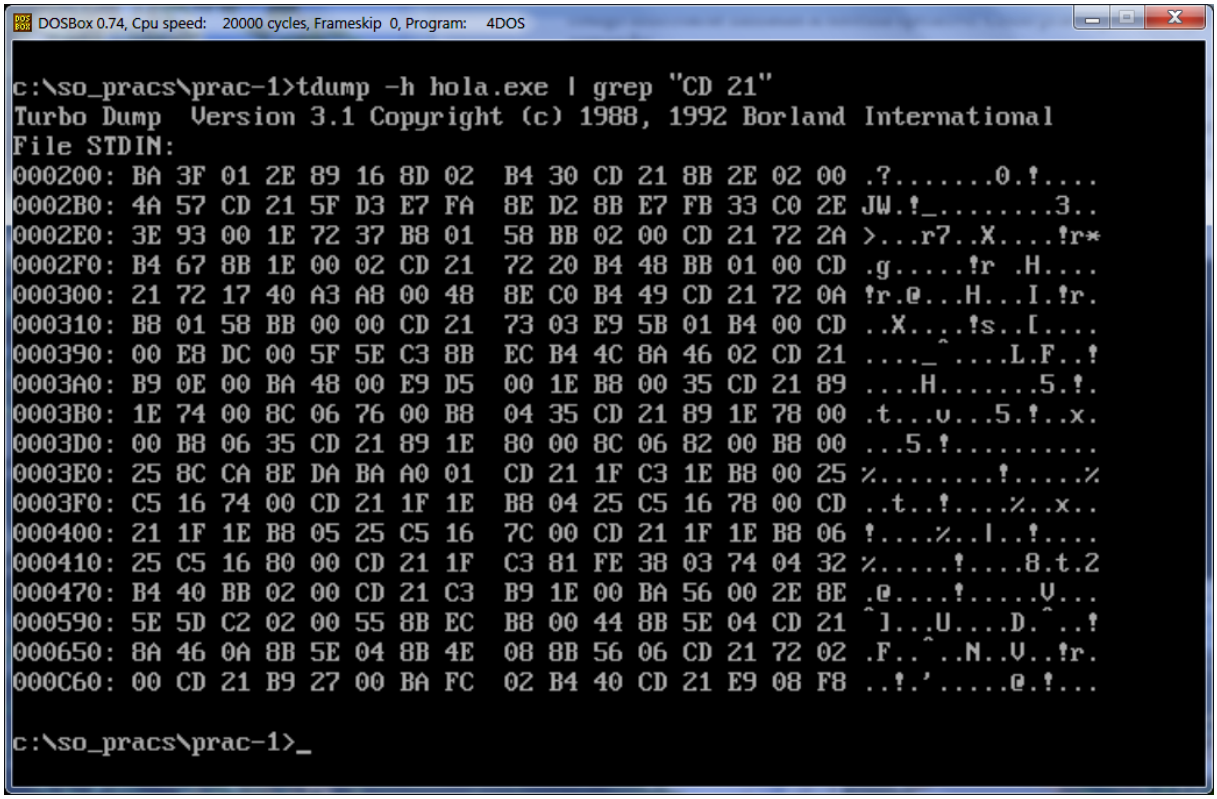
Es muy importante que el alumno se dé cuenta de que tanto el código del sector de arranque `"mi_boot.bin"` como el fichero `"prog.exe"` no son programas de usuario normales escritos en C. En los programas normales el compilador genera código que se apoya sobre el sistema operativo en aspectos como los siguientes:

- la asignación de memoria estática y dinámica al programa
- la entrada/salida y el manejo del sistema de ficheros
- la terminación del proceso asociado al programa en ejecución

Programas aparentemente tan sencillos como el típico `"hola.c"` ya visto incluyen en su código multitud de llamadas al sistema operativo. Como prueba podemos ejecutar el siguiente comando:

```
C:\PRACTI.CAS\PRAC-1>tdump -h hola.exe | grep "CD 21"
```

que muestra las líneas del volcado hexadecimal de `"hola.exe"` que contienen los bytes `"CD 21"`, los cuales corresponden al código máquina de la instrucción `INT 21` (interrupción software o trap 21H) utilizada para invocar las llamadas al sistema operativo MS-DOS:



Como vemos `"so1_hola.exe"` contiene 20 instrucciones `INT 21`, y por tanto ¡invoca 20 llamadas al sistema operativo! De ahí se deduce que `"hola.exe"` nunca podría funcionar tras ser cargado por el sector de arranque del disquete. Por el contrario `"mi_boot.bin"` no contiene ninguna de esas llamadas al sistema, y lo mismo sucede con `"prog.bin"` (exceptuando la llamada al sistema para terminar el programa que se ha dejado a propósito).

Si queremos que el compilador tcc de Turbo C no genere por defecto llamadas al sistema tendremos que establecer una serie de opciones de compilación especiales, como las que utilizaba el comando ***`compila`*** del que debemos echar mano para compilar `"prog.c"`:

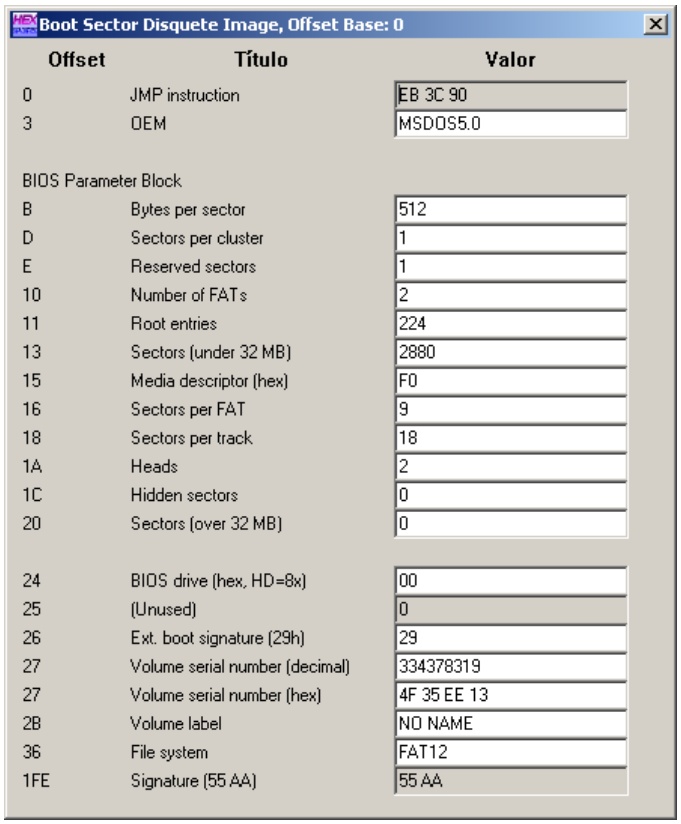
```
C:\PRACTI.CAS\PRAC-2>compila prog
```

Utilizando el comando **`compila`** conseguimos eliminar las llamadas al sistema operativo en el código generado, pero eso tiene como consecuencia que vamos a perder todas las comodidades de que disfrutábamos. Por lo pronto, ahora el punto de entrada del flujo de control no va a ser la función *`main`* correspondiente al programa principal, sino que será la primera instrucción que genere el compilador, es decir la correspondiente a la primera función que se declare en el fichero fuente. Eso no es muy grave ya que desde esa primera función siempre podemos hacer una llamada a la función *`main`*.

Otra incomodidad es que en algunos casos tendremos que ajustar nosotros mismos los segmentos de código, de datos y de pila del programa, cosa que normalmente realiza de una forma automática el compilador haciendo uso del sistema operativo.

6) ACTIVIDAD 3: Esqueleto para el sector de arranque

La estructura del sector de arranque, que se resume en la siguiente plantilla tomada del programa WinHex :



En el sector de arranque coexisten juntos código y datos entremezclados. Concretamente los primeros bytes corresponden a una instrucción de salto `JMP` que evita que se ejecuten los campos de datos del `Bloque de Parámetros de la BIOS`. El salto se dirige al byte situado en el desplazamiento `3E` que es justamente el byte siguiente a la cadena de 8 caracteres que describe el tipo de sistema de ficheros `"FAT12 "`. Por tanto, a partir de ese byte y antes del campo de signatura (`55 AA`) debe haber una serie de instrucciones que realicen la carga del sistema operativo y le cedan el control. En un programa normal el compilador separaría el código de los datos, por lo que vamos a explicar a continuación una manera sencilla de evitar ese problema insertando instrucciones de ensamblador dentro del código C.

```
#define nSectsSO 320          /* Numero de sectores a cargar (S.O.) */
void main () ;                /* declaracion forward de la funcion main */
void start () {
  asm jmp short inicio        /* instruccion JMP */
  asm nop                     /* instruccion NOP (90H) de relleno */
  asm db 'SO v2.00'           /* OEM, 8 caracteres */
  asm dw 512                  /* Bytes por sector */
  asm db 1                    /* Sectores por cluster */
  asm dw 1                    /* Sectores reservados */
  asm db 2                    /* numero de FATs */
```

```
  asm dw 224                  /* Entradas del directorio raiz */
  asm dw 2880                 /* numero de sectores en total (16 bits) */
  asm db 0xF0                 /* descriptor de medio */
  asm dw 9                    /* SectPorFAT */
  asm dw 18                   /* Sectores por pista */
  asm dw 2                    /* Cabezas */
  asm dd 0                    /* Sectores ocultos */
  asm dd 0                    /* numero de sectores en total 32 (bits) */
  asm db 0x00                 /* indica la unidad A: */
  asm db 0                    /* Byte que no se usa */
  asm db 0x29                 /* Extension de la signatura */
  asm dw nSectsSO             /* Numero de sectores a cargar */
  asm dw 0x0000               /* Disponible para otros usos */
  asm db 'ETIQUETA   '        /* Etiqueta de volumen, 11 caracteres */
  asm db 'FAT12   '           /* Tipo de sistema de ficheros */
inicio:
  main() ;
}

/* Declaracion de funciones auxiliares */

void main ( ) {
  /* codigo que realiza la carga y cede el control al s.o. */
}
/* Sino se incluyen las directivas _TEXT la directiva org actua sobre
   el segmento de datos y no saldria bien */
asm _TEXT segment byte public 'CODE'
asm     org 01FEh
asm     db 55h, 0AAh
asm _TEXT ends
```

El esqueleto anterior está disponible en `C:\PRACTI.CAS\PRAC-2` como `"boot_0.c"`. La primera línea del programa contiene la directiva `#define` que permite que nos refiramos mediante nombres simbólicos a valores o expresiones (análogamente a como se hace en Pascal definiendo constantes con la cláusula `CONST`). La constante que se define es el número de sectores que ocupa el sistema operativo en el disquete, y por tanto el número de sectores que habrá que cargar. Se estableció en el apartado 3 que serían 320 sectores (120 KBytes).

Tras el `#define` viene una declaración de la función `main` en la que simplemente se muestra su encabezamiento. Esa declaración avisa al compilador de que existe la función `main` por lo que podemos hacer llamadas a ella a pesar de que en ese punto del programa no se haya indicado todavía cuál es su cuerpo.

Luego viene la declaración de una función que hemos llamado a nuestro antojo `start`, y que contiene la instrucción de salto del sector de arranque y el `Bloque de Parámetros de la BIOS`. Es crucial que esa primera función no contenga variables locales, ya que en ese caso el compilador generaría código para asignarles espacio en la pila, y ese código desplazaría la instrucción de salto inicial a una dirección que ya no sería la cero (dentro del segmento de código).

La palabra reservada `asm` de Turbo C indica que va a insertarse en ese punto del programa el código correspondiente a una instrucción en ensamblador o los datos correspondientes a una directiva del ensamblador. Vemos que primero se inserta la instrucción de salto que requiere el sector de arranque y luego, mediante directivas, se insertan uno a uno todos los campos del `Bloque de Parámetros de la BIOS`. La directiva `db` (define byte) permite introducir una secuencia de caracteres (y cadenas de caracteres) separados por comas. La directiva `dw` (define word)

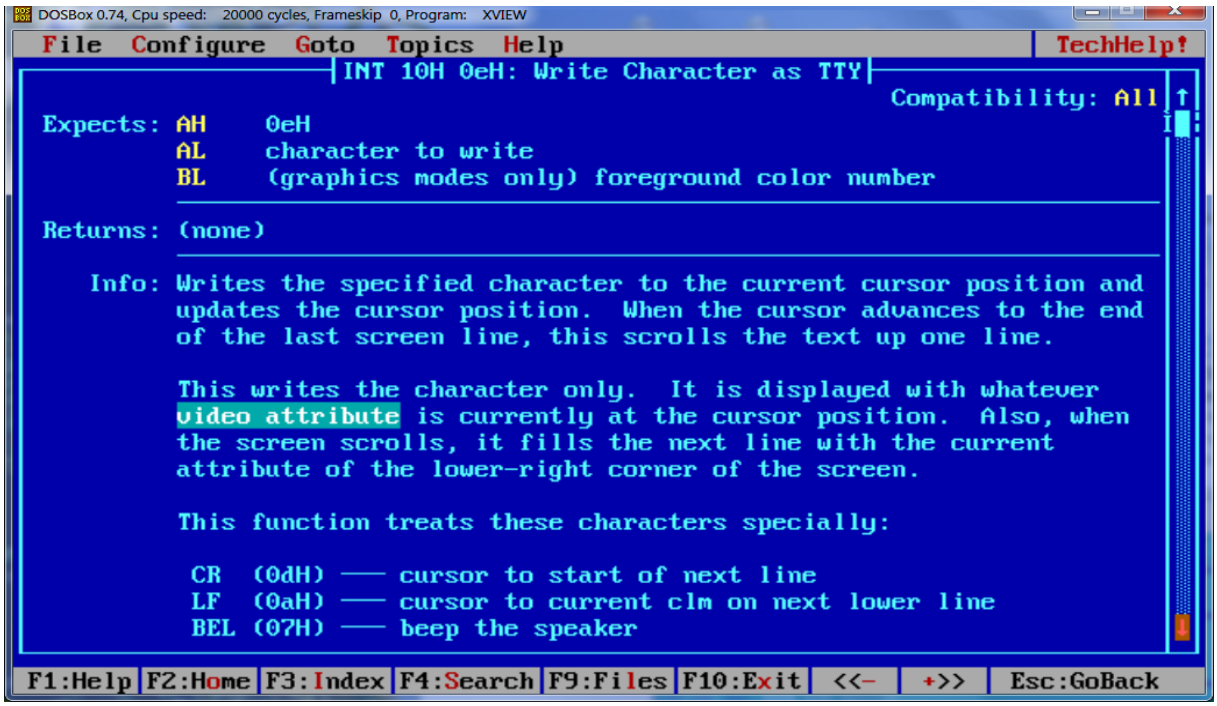
permite introducir una secuencia de palabras (16 bits) separadas por comas. Finalmente la directiva `dd` hace lo mismo con dobles palabras (32 bits).

A continuación aparece la etiqueta `inicio:` que es a donde se dirige el salto del principio del programa, llamándose inmediatamente a la función `main` donde en principio debería encontrarse el código que se ocupa de la carga del sistema operativo y de cederle el control. Con ese fin la función `main` puede hacer uso de funciones auxiliares que se declararían antes de ella y después de la función `start`. El formato del sector de arranque impone como requisito que termine con la signatura del sector de arranque (bytes `55 AA`) que debe ocupar exactamente los dos últimos bytes del sector. En otro caso la BIOS rechazaría ese sector de arranque. Por ese motivo hemos introducido las directivas `_TEXT` y `org 01FEH` antes de introducir los dos bytes de la signatura. Habrá que tener mucho cuidado de que el código del programa principal no sea tan grande que esa dirección esté ya ocupada por las instrucciones anteriores del programa principal.

7) ACTIVIDAD 4: Llamadas a la BIOS de manejo de teclado y pantalla.

Nuestro sector de arranque va a necesitar escribir en la pantalla algunos caracteres, por lo que vamos a tratar aquí el modo de hacerlo sin utilizar llamadas al sistema operativo MS-DOS y de manera que invirtamos pocos de los preciosos 512 bytes del sector de arranque en el código correspondiente. Una solución es recurrir a la BIOS. El programa `techelp` documenta los servicios que nos ofrece. La BIOS ofrece sus servicios mediante interrupciones software (*traps*). En el caso de los servicios de manejo de la pantalla el número del vector de interrupción que se utiliza es el 10H (16 en decimal).

Para cada tipo de servicio hemos de indicar como parámetro en el registro AH (byte de mayor peso del registro acumulador AX del microprocesador 8086) un número que identifica la operación concreta que queremos llevar a cabo. En nuestro caso deseamos ser capaces de escribir un carácter por la pantalla como si se tratara de un terminal, por lo que hemos de indicar en AH el código de operación 0E hex. Veamos los detalles sobre esa operación:



Los detalles son mínimos: para escribir un carácter en la pantalla utilizando el servicio 0eH de la interrupción 10H de la BIOS hemos de poner el carácter en el registro AL (byte de menor peso de AX) y el byte de atributo de pantalla en BL (byte de menor peso del registro BX). El atributo determina el color del carácter y el color de fondo del espacio sobre el que se escribe. El atributo normal (carácter blanco sobre fondo negro) corresponde al byte 07H, que es el que vamos a utilizar.

Con lo anterior podemos implementar la siguiente función auxiliar que nos permite visualizar un carácter por pantalla:

```
void printCar ( char car ) {
  asm mov al,car       /* car -> caracter a escribir */
  asm mov bl,07H       /* 07H -> atributo normal */
  asm mov ah,0eH       /* 0eH -> escribir caracter */
  asm int 10H          /* 10H -> servicios BIOS de manejo de la pantalla */
}
```

En cuanto a la lectura de caracteres desde el teclado podemos hacer exactamente lo mismo pero ahora utilizando la interrupción 16H (22 decimal) de la BIOS. En la siguiente pantalla del `techelp` se muestran los servicios ofrecidos a través de esa interrupción. Así, haciendo uso de la BIOS podemos programar del siguiente modo una función auxiliar que espera a que se pulse una tecla y retorna el carácter correspondiente:

```
char leerTecla ( ) {
  char car ;
  asm mov ah,00H       /* 00H -> leer siguiente tecla pulsada */
  asm int 16H          /* 16H -> servicios BIOS de teclado */
  asm mov car,al       /* El caracter ascii se nos devuelve en AL */
  return(car) ;
}
```

Vamos a ver otra posible función auxiliar que nos permita reiniciar el ordenador desde el programa. En un PC, cuando se enciende el ordenador o se pulsa el botón de reset se cede automáticamente el control a la instrucción que se encuentra en la dirección: 0FFFF0H (1 Mega menos 16 bytes), que expresada en forma de par segmento:desplazamiento del 8086 corresponde (por ejemplo) a la dirección FFFF:0000. Esa dirección es el punto de entrada de la BIOS que está en la ROM, el cual inicia el ordenador. Por tanto en cualquier momento podemos reiniciar el ordenador cediendo el control a esa dirección. Podemos programar del siguiente modo la función de reinicio poniendo la dirección a la que queremos ir en la pila y ejecutando una instrucción de retorno:

```
void reboot ( ) {
  asm push 0FFFFH       /* apilamos el numero de segmento */
  asm push 0000H        /* apilamos el numero de desplazamiento */
  asm retf              /* hacemos un retorno lejano (far) */
}
```

En este momento estamos ya en condiciones de escribir un sector de arranque que se limite a hacer eco en pantalla de los caracteres que se introduzcan a través del teclado. En el directorio `C:\PRACTI.CAS\PRAC-2` se encuentra el programa `"boot_1.c"` que es una primera aproximación al objetivo de este ejercicio y que nos permitirá probar las funciones auxiliares anteriores. Se puede ha añadido la función auxiliar:

```
void finProgDOS () {
  asm mov ah,4ch         /* Llamada al sistema MS-DOS: Terminar Programa */
```

```
  asm mov al,00h          /* Codigo de retorno 00h (como con exit(0))     */
  asm int 21h
}
```

con el fin de poder ejecutar primeramente el programa desde MS-DOS terminando el programa normalmente, a la vez que ilustramos cómo se hace una llamada al sistema operativo MS-DOS concreta (véase su documentación en el `techelp`). Esta llamada al sistema deberá suprimirse del código del sector de arranque definitivo, dejando en su lugar la función `reboot`. A continuación mostramos el programa `"boot_1.c"` completo:

```
#define nSectsSO 320          /* Numero de sectores a cargar (S.O.) */
#define CR        13          /* Retorno de carro */
#define LF        10          /* Avance de linea */
#define ESC       27          /* Tecla ESC */

void main () ;                /* declaracion forward de la funcion main */

void start () {
  asm jmp short inicio        /* instruccion JMP */
  asm nop                     /* instruccion NOP (90H) de relleno */
  asm db 'SO v2.00'           /* OEM, 8 caracteres */
  asm dw 512                  /* Bytes por sector */
  asm db 1                    /* Sectores por cluster */
  asm dw 1                    /* Sectores reservados */
  asm db 2                    /* numero de FATs */
  asm dw 224                  /* Entradas del directorio raiz */
  asm dw 2880                 /* numero de sectores en total (16 bits) */
  asm db 0xF0                 /* descriptor de medio */
  asm dw 9                    /* SectPorFAT */
  asm dw 18                   /* Sectores por pista */
  asm dw 2                    /* Cabezas */
  asm dd 0                    /* Sectores ocultos */
  asm dd 0                    /* numero de sectores en total 32 (bits) */
  asm db 0x00                 /* indica la unidad A: */
  asm db 0                    /* Byte que no se usa */
  asm db 0x29                 /* Extension de la signatura */
  asm dw nSectsSO             /* Numero de sectores a cargar */
  asm dw 0x0000               /* Disponible para otros usos */
  asm db 'ETIQUETA   '        /* Etiqueta de volumen, 11 caracteres */
  asm db 'FAT12   '           /* Tipo de sistema de ficheros */

inicio:
  main() ;
}

void printCar ( char car ) {
  asm mov al,car       /* car -> caracter a escribir */
  asm mov bl,07H       /* 07H -> atributo normal */
  asm mov ah,0eH       /* 0eH -> escribir caracter */
  asm int 10H          /* 10H -> servicios BIOS de manejo de la pantalla */
}

char leerTecla ( ) {
  char car ;
  asm mov ah,00H       /* 00H -> leer siguiente tecla pulsada */
  asm int 16H          /* 16H -> servicios BIOS de teclado */
  asm mov car,al       /* El caracter ascii se nos devuelve en AL */
  return(car) ;
}
```

```
void prompt ( ) {
  printCar('S') ;
  printCar('O') ;
  printCar('1') ;
  printCar('>') ;
  printCar(' ') ;
}
void reboot ( ) {
  asm push 0FFFFH       /* apilamos el numero de segmento */
  asm push 0000H        /* apilamos el numero de desplazamiento */
  asm retf              /* hacemos un retorno lejano (far) */
}

void finProgDOS () {
  asm mov ah,4ch         /* Llamada al sistema MS-DOS: Terminar Programa */
  asm mov al,00h         /* Codigo de retorno 00h (como con exit(0))     */
  asm int 21h
}

void main ( ) {
  char car ;
  prompt() ;
  while ((car = leerTecla()) != ESC) {
    printCar(car) ;
    if (car == CR) {
      printCar(LF) ;
      prompt() ;
    }
  }
  finProgDOS();
  reboot() ;
}
/* Si no se incluyen las directivas _TEXT la directiva org actua sobre
   el segmento de datos y no saldria bien */
asm _TEXT segment byte public 'CODE'
asm     org 01FEh
asm     db 55h, 0AAh
asm _TEXT ends
```

A continuación compilamos el programa del siguiente modo:

```
C:\PRACTI.CAS\PRAC-2>compila boot_1
```

Este comando genera los ficheros “boot_1.exe y boot_1.bin”. Probamos que boot_1.exe se ejecute correctamente y tras comprobar que todo va bien quitamos del fuente la función “finProgDOS” y su llamada y recompilamos, ya que ahora vamos a utilizar el fichero generado “boot_1.c”, que no tiene la cabecera de 512 bytes para probarlo como sector de arranque. El tamaño del fichero debe ser exactamente 512 bytes y si editamos o vemos s interior con cualquier utilidad (cn, tdump..) veremos también que los dos últimos bytes son la signatura 55AA. Por ejemplo:

```
C:\PRACTI.CAS\PRAC-2> tdump -h boot_1.bin | more
```

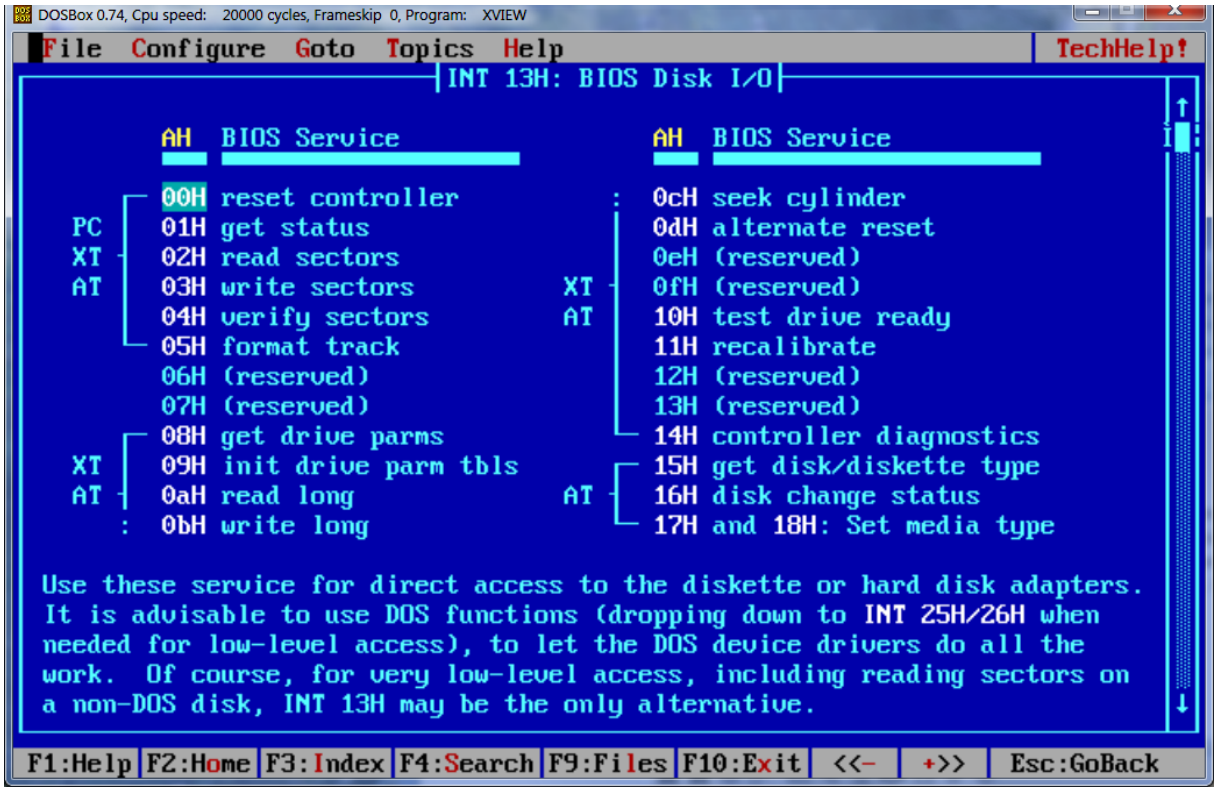
Tras la comprobación sólo nos queda escribir el sector de arranque en un disquete previamente formateado y reiniciar la máquina virtual DosBOX como se ha hecho en otras ocasiones.

```
C:\PRACTI.CAS\PRAC-2>escr_s boot_1.bin 0 0
```

Se verá que el programa despliega enseguida su prompt `"SO> "` y que realiza perfectamente el eco del teclado, volviendo a mostrar el prompt tras cada pulsación de la tecla de retorno de carro. La única salida posible del programa es presionando la tecla de escape ESC, lo que da lugar al reinicio del ordenador.

8) ACTIVIDAD 5: Llamadas a la BIOS de acceso al disco

Vamos a tratar ahora la forma en la que debe realizarse la lectura de sectores del disquete desde el código del sector de arranque. En la práctica 1 utilizamos la función `biosdisk`, ofrecida por la biblioteca `bios.h`, para leer sectores. Ahora vamos a implementar nosotros mismos el acceso al disquete utilizando directamente llamadas a la BIOS. Comenzamos echando un vistazo a la documentación que nos proporciona el `techelp`:



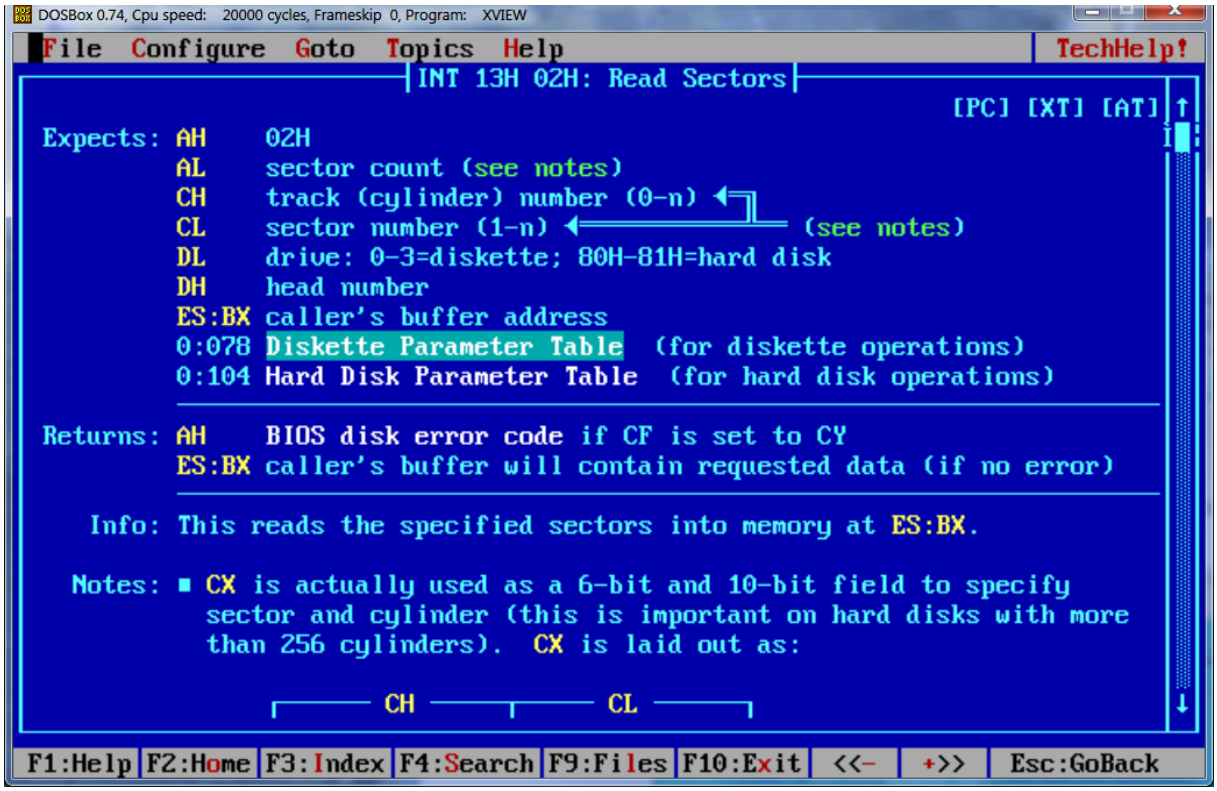
La interrupción 13H de la BIOS ofrece las siguientes operaciones:

El sector de arranque necesita tan solo dos operaciones: el reset del controlador (subfunción 00H) y la lectura de un sector del disco (subfunción 02H). El número de la subfunción debe escribirse en AH antes de ejecutar la instrucción INT 13H. Por su simplicidad vamos a comenzar implementando el reset del controlador de disquete.
En el arranque es necesaria la operación de reset porque el controlador podría haberse quedado en un estado incorrecto tras la ejecución de un programa que deje colgado el ordenador. La documentación dice que aparte de poner 00H en AH, es necesario poner en DL (parte baja del registro DX) la unidad sobre la que va a realizarse el reset (0 = A:, 1 = B:, 80H = C:, 81H = D:).

```
int resetDisquete ( char unidad ) {
  asm mov dl,unidad
  asm mov ah,00H       /* 00H -> reset del controlador */
  asm int 13h          /* BIOS: E/S de disco */
```

```
  asm jc resetError   /* el flag de acarreo C se activa en caso de error */
  return(0) ;
resetError:
  return(1) ;
}
```

Pasamos a ocuparnos de la operación de lectura de un sector. La documentación nos dice lo siguiente:



Vamos a explicar un poco cada uno de los parámetros de la llamada a la BIOS para leer un sector. Está claro que en el registro DL hemos de poner el número de la unidad. En los registros DH, CH y CL hay que codificar la información correspondiente al número de cabeza, pista y sector físico del sector lógico que se desea leer. El número de cabeza (0 o 1) se pone tal cual en DH. El número de sector físico (de 1 a 18) debe ponerse en CL. Finalmente el número de pista, que sería más correcto denominar número de cilindro (de 0 a 79) debe ponerse en CH. Se apreciará que la explicación anterior parece diferir de lo que señala el `techelp`. El motivo es que el `techelp` describe el caso general en el que el número de cilindro de un disco duro puede requerir hasta 10 bits, debiendo aprovecharse los dos bits superiores de CL para contener los dos bits que no caben en CH. En el caso de un disquete, con 80 cilindros, no tenemos ese problema, de ahí que todo sea mucho más sencillo.

Siguiendo con los parámetros de la llamada a la BIOS para leer un sector, vemos que en AL hay que indicar el número de sectores consecutivos que queremos leer (sin exceder un cilindro), por lo que pondremos en AL un 1. En cuanto a la dirección de memoria donde debe quedar el sector leído, vemos que hay que indicarla en los registros ES (registro de segmento extra) y BX. En ES debe ponerse el número de segmento de la dirección, y en BX el desplazamiento. Con esto hemos terminado de explicar la información que es estrictamente necesario indicar a la BIOS en esta llamada. Como sucedía con la operación de reset, si la BIOS detecta algún error en la lectura del sector, activa el flag C de acarreo para indicarnos ese hecho. A continuación mostramos el

esqueleto de la función que realiza la lectura de un sector lógico del disquete, dejándose como tarea al alumno el completar los detalles:

```
/* leerSector: sect es el numero de sector logico                          */
/*             unidad es el numero de la unidad (0 = A:, 1 = B:)           */
/*             dir es la direccion del bufer de memoria                    */

int leerSector (int sect, char unidad, void far * dir) {

  char sector ;            /* numero de sector fisico (1..nSectPorPista) */
  char cabeza ;            /* numero de cabeza (0 o 1) */
  char pista ;             /* numero de pista (cilindro) */

  sector = ... ;           /* calculo de (s,c,p) a partir de sect */
  cabeza = ... ;
  pista = ... ;

  asm les bx,dir           /* pone en ES:BX la direccion del bufer */
  asm mov dl,...
  asm mov dh,...
  asm mov ch,...
  asm mov cl,...
  asm mov al,...
  asm mov ah,02h           /* 02H -> lectura de un sector */
  asm int 13h              /* BIOS: E/S de disco */
  asm jc errorAlLeer  /* el flag de acarreo C se activa en caso de error */
  return(0) ;
errorAlLeer:
  return(1) ;
}
```

Con el fin de comprobar que la función anterior se comporta correctamente el alumno puede ir al programa `"l_s.c"` que implementó en la práctica 1, y reemplazar la llamada a la función `biosdisk`:

```
biosdisk(cmd_read_sector, unidad, c, p, s, 1, &bufer) ;
```

por una llamada a la función `leerSector`:

```
leerSector(sectorlogico, unidad, &bufer) ;
```

A continuación debe recompilarse el programa con el `tcc` (ya que `"l_s.c"` utiliza bibliotecas como `stdio.h` dependientes de MS-DOS) y comprobarse que el programa `"l_s.c"` sigue funcionando bien a pesar del cambio realizado.

9) ACTIVIDAD 6: Manejo de direcciones de memoria

Tras el apartado anterior sabemos ya cómo podemos leer los 320 sectores lógicos correspondientes al sistema operativo. Esos sectores ocuparán 160K en memoria principal, por lo que el búfer de memoria donde leemos cada uno de esos sectores debe ir desplazándose 512 bytes tras la lectura de cada sector. Los punteros que se manejan en el 8086 normalmente son desplazamientos de 16 bits relativos al segmento de datos. Con tan solo 16 bits no es posible direccionar los 160K de memoria donde va a cargarse el sistema operativo, por lo que necesitamos direcciones completas formadas por un número de segmento y un desplazamiento, las cuales requieren 32 bits en

total. En Turbo C el tipo de datos correspondiente a esas direcciones son los punteros lejanos (*far*) como por ejemplo los siguientes punteros `origen` y `destino` que apuntan a un carácter:

```
char far * pOrigen;
char far * pDestino;
```

A estos punteros lejanos se les puede asignar punteros normales (*near*), puede escribirse su valor con `printf` y el formato `%08lX`, y puede operarse con ellos con sumas y restas. Por ejemplo podemos copiar las 1000 posiciones de memoria que comienzan en la dirección segmentada `8765H:4321H` (que corresponde en el 8086 a la dirección lineal `8B971H = 87650H+4321H`) llevándolas a partir de la dirección `8900H:1234H` mediante el siguiente bucle:

```
pOrigen  = (char far*) 0x87654321L;
pDestino = (char far*) 0x89001234L;
contador = 1000 ;
while (contador-- > 0) *destino++ = *origen++ ;
```

Aunque en este caso concreto no se da el problema, hay que tener mucho cuidado de que en las sumas con punteros no se desborden los 16 bits correspondientes al desplazamiento, ya que por ejemplo si un puntero far vale `0x0000FFFF` y le sumamos 1, lo que se obtiene en Turbo C es `0x00000000`, en vez del valor esperado en el 8086 que es `0x10000000`. El alumno puede comprobarlo con el siguiente programa:

```
void main ( ) {
  char far * ptr;
  ptr = (char far *)0x0000FFFFL;
  printf(" ptr = %08lX antes de incrementarse\n", ptr++);
  printf(" ptr = %08lX despues de incrementarse\n", ptr);
}
```

Con el fin de solucionar el problema anterior al copiar zonas de memoria extensas, vamos a proporcionar una función de incremento de la dirección contenida en un puntero lejano.

```
void inc (void *p, unsigned i) {
  if (i > 0xFFFF - *(unsigned *)p) ((unsigned *)p)[1] += 0x1000;
  *(unsigned *)p += i;
}
```

Con esta función podemos realizar correctamente el incremento de la dirección contenida en un puntero lejano 'p', en una cierta cantidad 'i' (de 16 bits). Ahora el programa equivalente al anterior funciona como se espera en un 8086:

```
void main ( ) {
  char far * ptr ;
  ptr = (char far *) 0x0000FFFF ;
  printf(" ptr = %08lX antes de incrementarse\n", ptr) ;
  inc(&ptr, 1) ;
  printf(" ptr = %08lX despues de incrementarse\n", ptr) ;
}
```

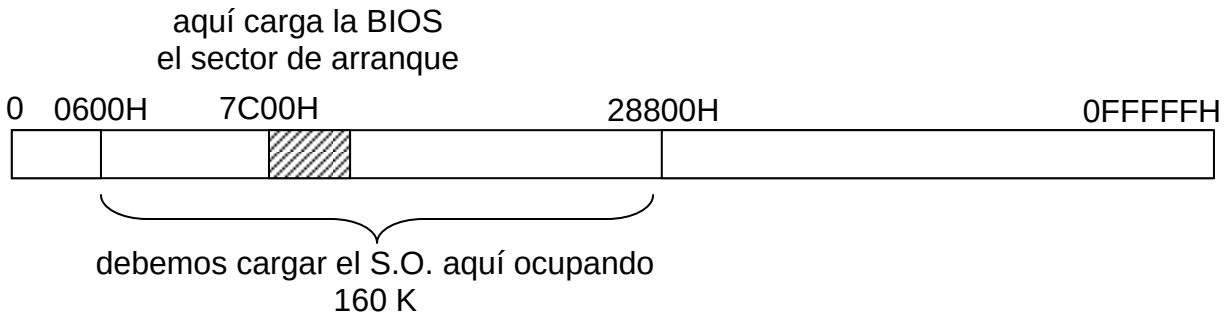
Una vez que se dispone de la función auxiliar '`inc`' podemos programar la carga desde el disquete de los 320 (`nSectSO`) sectores del sistema operativo de la siguiente manera, donde utilizamos un puntero para indicar en cada momento la dirección de comienzo del búfer:

```
#define PRI_SEC_DATOS 33
char far * ptr = (char far *) 0x00600000L;     /* 0060:0000 = 00600H */
```

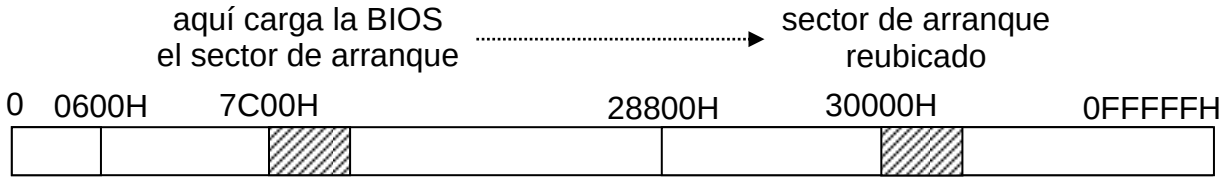
```
int i;
...

...
for (i = PRI_SEC_DATOS; i < nSectsSO+PRI_SEC_DATOS; i++ ) {
   leerSector(i++, 0, ptr);
   printCar('.');
   inc(&ptr, 512);
}
```

10) ACTIVIDAD 7: Implementación final del sector de arranque

Un hecho que complica la carga del sistema operativo es que la BIOS carga los 512 bytes del sector de arranque a partir de la dirección de memoria **007C00H**. El problema es que esa dirección cae dentro del rango de direcciones que queremos que ocupe el sistema operativo, como muestra la figura siguiente. Por tanto llegaría un momento durante la carga en el que el sistema operativo sobrescribiría el código del sector de arranque, lo que tendría consecuencias desastrosas.



La solución al problema consiste en que el sector de arranque se reubique desde su posición inicial a una zona de memoria segura que no sea recubierta por el sistema operativo. Esa nueva zona de memoria puede ser cualquiera a partir de la dirección 28800H, que es donde acaba el sistema operativo. Vamos a elegir para reubicar el sector de arranque la dirección 30000H que está suficientemente alejada.



Además de reubicar el sector de arranque, necesitaremos que la pila que se utilice tampoco sea sobrescrita durante la carga del sistema operativo, lo que sería también catastrófico (obsérvese que no utilizamos variables globales). Con ese fin el sector de arranque debe establecer una nueva pila en una zona segura. Vamos a elegir como valor para el segmento de pila SS el mismo que donde se copia el código del sector de arranque, es decir el 3000H. En cuanto al puntero de pila SP vamos a darle también el mismo valor 3000H, de manera que la cima de la pila estará en la dirección SS:SP = 3000:3000. El establecimiento de la pila debe hacerse en la función `start` antes de llamar a la función `main`, ya que la función **`main`** necesita la pila para asignar memoria a sus variables locales. El código que se necesita al final de la función `start` es el siguiente:

```
inicio:

  asm cli                       /* inhibimos las interrupciones */
  asm mov ax,3000h              /* Establecemos la pila en la dir. 33000H */
  asm mov ss,ax                 /* Segmento de pila SS = 3000H */
  asm mov sp,ax                 /* SS:SP = 3000:3000 = 33000H */
  asm sti                       /* permitimos las interrupciones */

  main();
```

La razón por la que inhibimos temporalmente las interrupciones (intrucción `cli`) es que la pila interviene decisivamente en la gestión de las interrupciones, que no deben aceptarse mientras modificamos la ubicación de la pila.

El punto más crítico del código del sector de arranque es el momento en el que el código del sector de arranque (cargado a partir de 07C00H) debe ceder el control a su copia (a partir de la dirección 30000H) por lo que vamos a indicar esos pasos en detalle:

```
void main ( ) {

    char far *pOrigen = (char far*)0x07C00000;  /* 0000:7C00 = 07C00H */
    char far *pDestino= (char far*)0x30000000;  /* 3000:0000 = 30000H */
    int i;

    . . .   /* Declaracion de otras variables locales que se necesiten */

    for (i=0; i < 512; i++ )  /* reubicacion del sector de arranque */
      *destino++ = *origen++;

    asm push 3000H /* Cedemos el control al sector de arranque reubicado */
    asm push OFFSET($+4)         /* justo despues de la instruccion retf */
    asm retf    /* $ = dirección a la que apunta el contador de programa */
    asm push cs     /* establecemos el segmento de datos DS = CS = 3000H */
    asm pop ds

    ...                                 /* Hacer un reset de la unidad A: */

    ...     /* Cargar el S.O. a partir de la dirección 0600H escribiendo */
            /* un punto tras la lectura de cada sector                   */

    ... /* Escribir en pantalla los caracteres: CR y LF, seguidos de SO */

    asm push 0060H  /* Cedemos el control al S.O. en 0060:0000 = 000600H */
    asm push 0000H
    asm retf

}
/* Si no se incluyen las directivas _TEXT la directiva org actua sobre
   el segmento de datos y no saldria bien */
asm _TEXT segment byte public 'CODE'
asm     org 01FEh
asm     db 55h, 0AAh
asm _TEXT ends
```

Vemos en el programa principal que lo primero es reubicar el sector de arranque. Tras la reubicación se cede el control a la copia del sector de arranque justo después de la instrucción `retf`. El signo $ hace referencia en ensamblador al contador de programa, por tanto si sumamos 4 al CP obtenemos un dirección 4 bytes posterior, que es justo la dirección de la siguiente instrucción a retf. Esto puede comprobarse examinando el código maquina con el debugger del DOS, o bien

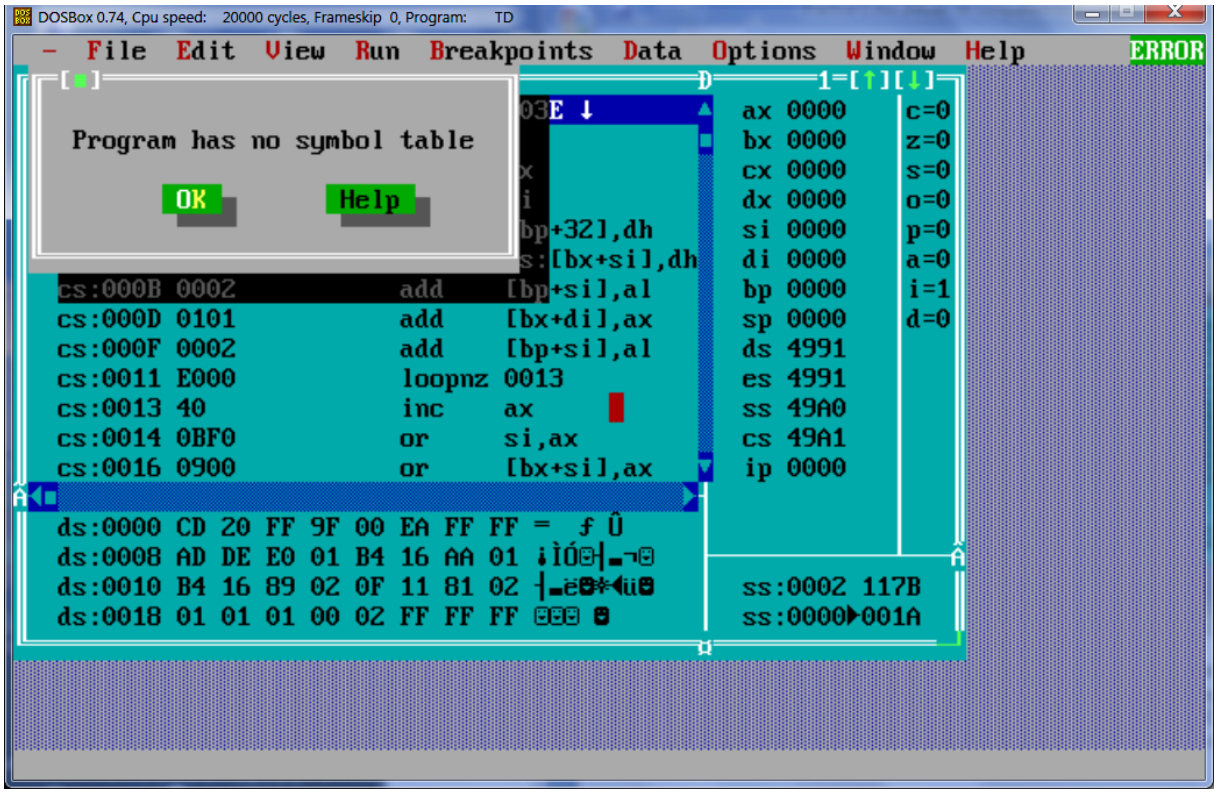
consultando un manual de ensamblador, en el que se especifica que la instrucción push OFFSET X es 68XXXX, ocupando 3 bytes y la instrucción retf es CB, ocupando un byte.

El resto de tareas que debe realizar el sector de arranque están indicadas como comentarios y se dejan como tarea para el alumno. Al final se cede el control al sistema operativo en la dirección 00600H como se exigía en la especificación. Para ello se apila (`push`) primero el valor del segmento (0060H), luego el valor del desplazamiento (0000H) y después se ejecuta otra instrucción `retf`. La instrucción `retf` saca de la pila la primera palabra (0000H) poniéndola en el contador de programa (IP) y saca luego la otra palabra (0060H) poniéndola en el registro del segmento de código (CS), de manera que la siguiente instrucción a ejecutar será la de la dirección CS:IP = 0060:0000 que es precisamente la primera instrucción del sistema operativo. Obsérvese que hemos ajustado el valor del segmento de código a 60H para que la primera instrucción del sistema operativo corresponda al desplazamiento 0000H.
Con esto hemos terminado de explicar el código del sector de arranque compuesto por la función `start`, las funciones auxiliares (actividad 5 y 6) y la función `main`. Con todo ese código el alumno deberá crear un fichero llamado `"boot_2.c"` que incluso sin depurar va a permitirnos ilustrar el flujo de control que se produce durante el arranque. Con ese fin procedemos a compilar el programa:

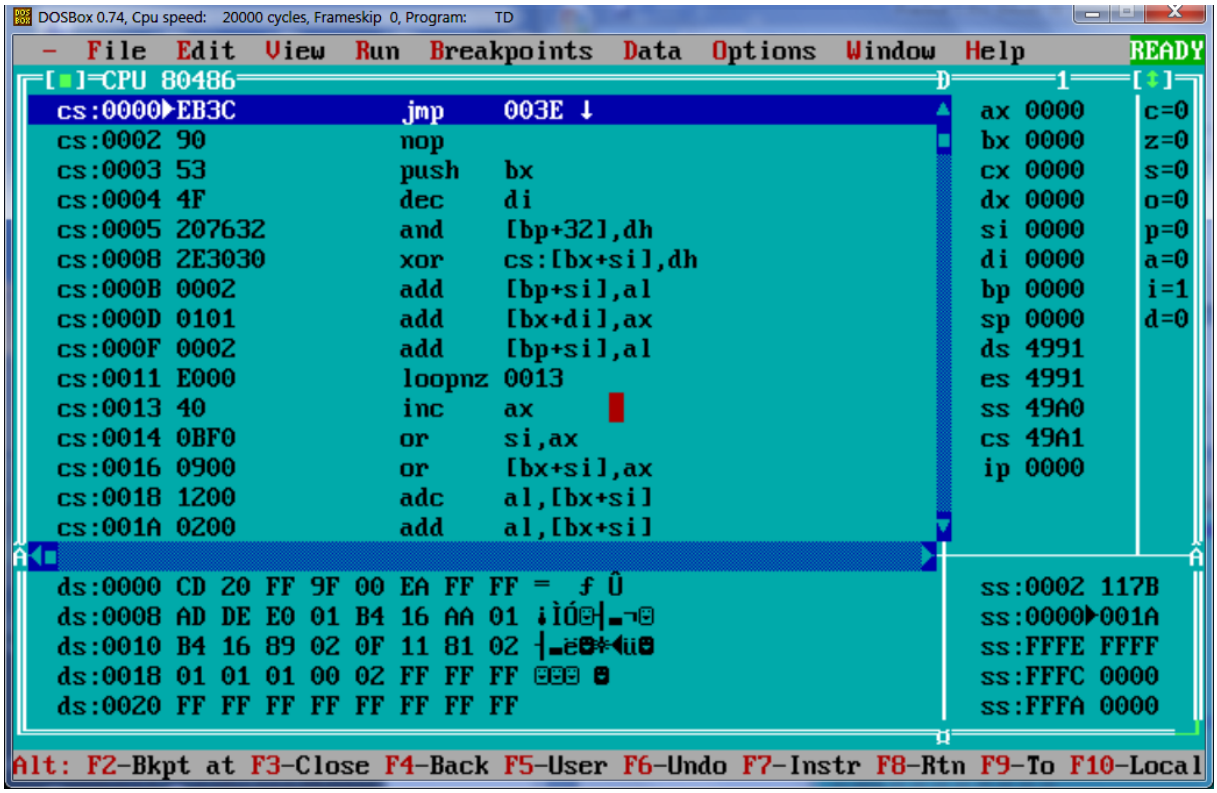
```
C:\SO1\PRACT2> compila boot_2                                    [tcc -v boot_2.c]
```

Ahora ejecutamos el depurador `td` (Turbo Debugger):

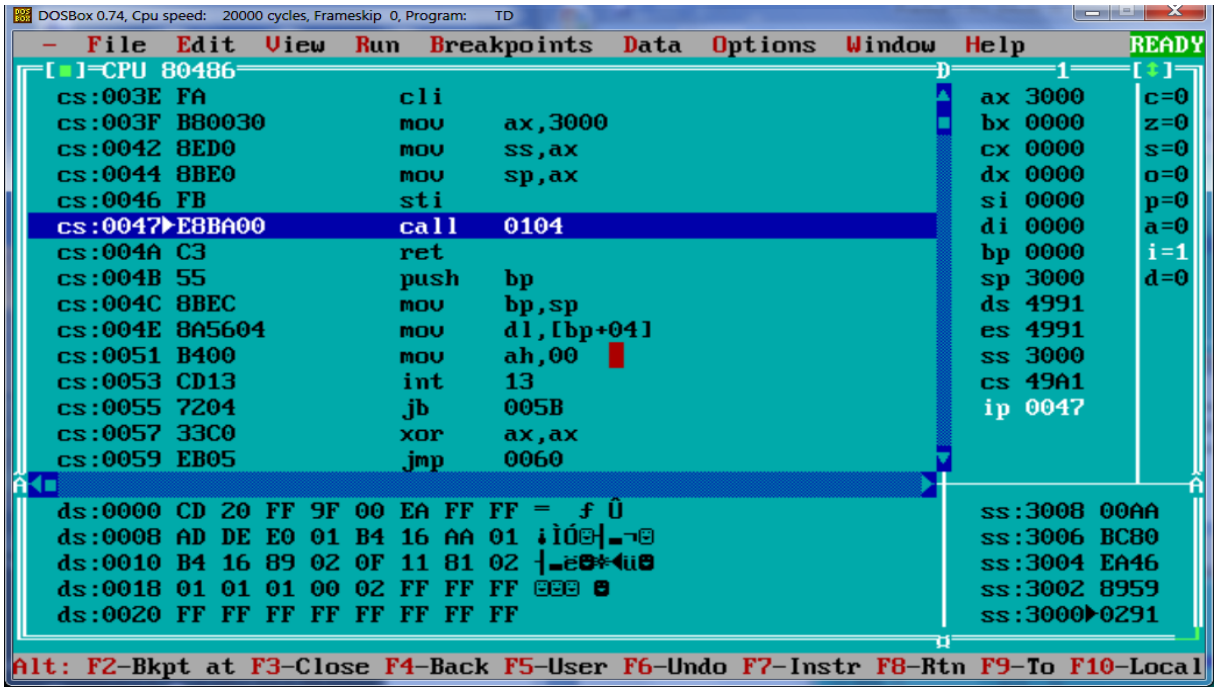
```
C:\SO1\PRACT2> td boot_2.exe
```



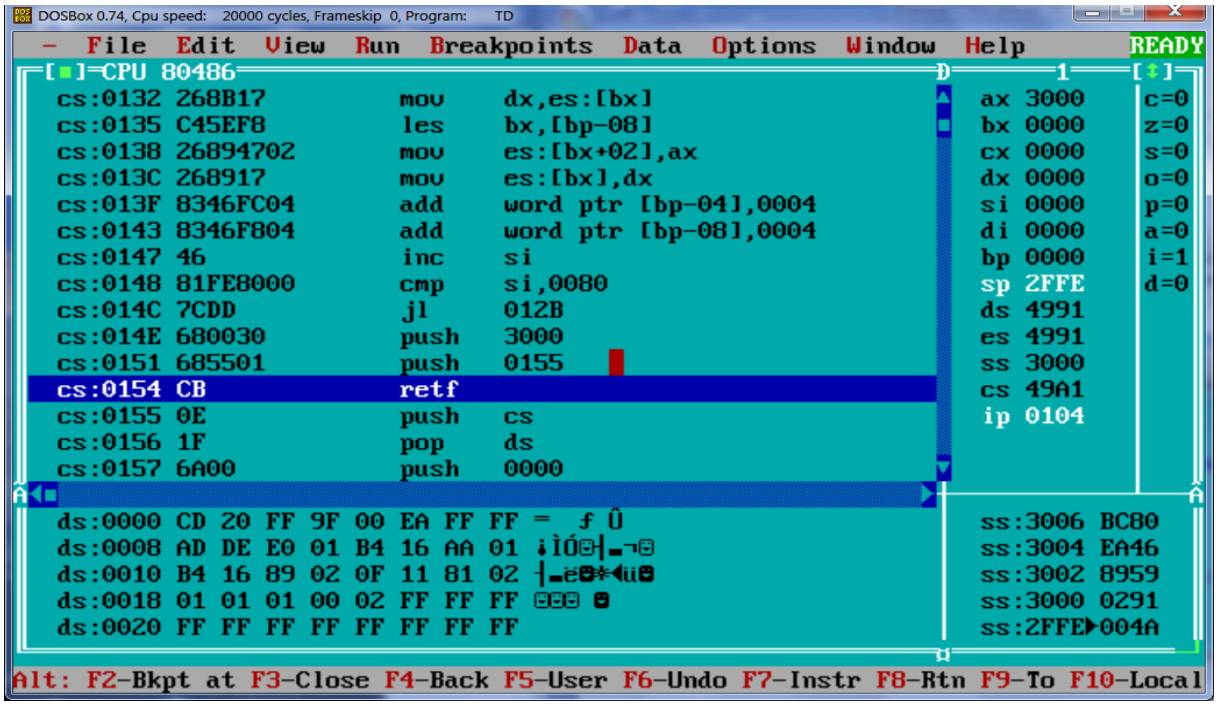
Pulsando la tecla INTRO aceptamos el mensaje. A continuación presionamos F5 para agrandar la ventana. Veremos que todo está listo para seguir la ejecución del programa a partir de la dirección 0000 del segmento de código (CS:0000). Dicha instrucción es precisamente la instrucción de salto que está al principio del sector de arranque.



Pulsando F7 se provoca la ejecución de la instrucción actual. Pulsando F8 se consigue ejecutar la instrucción actual o toda una llamada a un procedimiento (instrucción `call`) o toda una interrupción software (instrucción `int`). Vamos a ir presionando F7 hasta llegar al primer `call` que corresponde a la llamada a la función `main`.

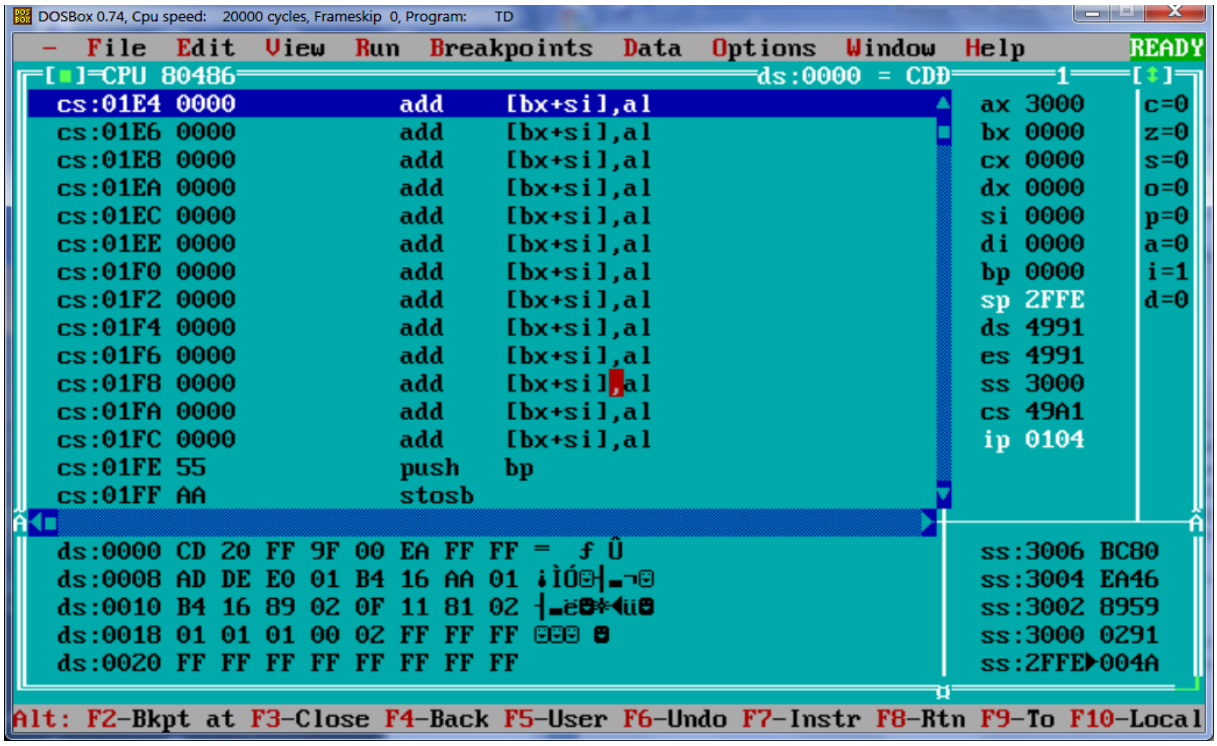


Pulsamos otra vez F7 para saltar al programa principal en la dirección CS:0104. Ya desde esa posición lo mejor es que nos limitemos a buscar la primera instrucción `retf` del programa principal utilizando la tecla de cursor abajo o `Avance de Página`.



Una vez localizada la primera instrucción `retf` vemos que la instrucción siguiente está en la dirección CS:0155, que efectivamente coincide con el segundo de los valores que se apilan (que en el programa expresamos mediante `OFFSET($ + 4)`)

Obsérvese que algo más adelante todo son ceros 0000, hasta llegar a la signatura (55 AA) del sector de arranque:



Tras las explicaciones anteriores el alumno ya puede desarrollar el sector de arranque cuya especificación se dio en el apartado 3. Se advierte de que hay que limitar al máximo el tamaño del código que se genere, ya que el sector de arranque no puede ocupar más que los 512 bytes del primer sector. En particular debe suprimirse cualquier función auxiliar que no se utilice desde el programa principal, aunque de eso ya se encarga automáticamente el compilador. Los requisitos para poder evaluarse de este ejercicio son:

- haber desarrollado completamente el fichero `"boot_2.bin"` cumpliendo todas las especificaciones del apartado 3

- haber comprobado que `"boot_2.bin"` funciona correctamente cuando se graba en un disquete de arranque con el fichero `"prog.bin"` (o cualquier otro) similar

- tener suficientemente claros todos los aspectos que recoge esta práctica

Por otra parte hay que decir que sería conveniente (si queda espacio libre en el sector de arranque) realizar alguna detección de errores a la hora de hacer el reset de la unidad A:, o a la hora de leer los sectores que contienen el sistema operativo. En caso de detectarse un error sería deseable escribir algunos caracteres indicándolo y esperar a la pulsación de una tecla para reiniciar el ordenador. En la entrega del trabajo para su evaluación podrán plantearse (en caso de duda sobre los conocimientos del alumno) cuestiones relativas a esas mejoras.

APÉNDICE II. Práctica: Interrupciones, excepciones y llamadas al sistema.

ÍNDICE

1. OBJETIVOS

Los objetivos de esta práctica son:

- Que el alumno conozca en detalle cómo se implementa el modelo de los procesos sin entrar en algoritmos de planificación complejos.
- Que el alumno entienda el funcionamiento de las interrupciones y excepciones como parte del sistema operativo.
- Que el alumno entienda cómo funciona sobre una máquina concreta el mecanismo de las llamadas al sistema.
- Que el alumno entienda el funcionamiento de la interrupción de reloj al nivel necesario para implementar la llamada al sistema ***sleep***.
- Que el alumno sea capaz de implementar en un sistema operativo una herramienta de sincronización sencilla (semáforos con las operaciones: ***iniSemaforo***, ***bajaSemaforo*** y ***subeSemaforo***).
- Que el alumno escriba algún programa de usuario que haga uso de las llamadas al sistema disponibles y conozca todos los pasos necesarios para su ejecución.

2. INTRODUCCIÓN

En la primera práctica el alumno aprendió a utilizar las llamadas al sistema de gestión de ficheros desde un programa escrito en C tomando conciencia de las operaciones básicas de implementación del sistema de ficheros, que son la lectura y escritura de sectores lógicos del disco.

En la segunda práctica el alumno se enfrentó con la ingrata situación de tener que escribir un programa capaz de ejecutarse sin recurrir a un sistema operativo, y desarrolló un sector de arranque capaz de cargar desde un disquete o CDROM cualquier sistema operativo sencillo.

Una vez superados estos desafíos el alumno está preparado para dar el salto y meterse de lleno en lo que realmente es un sistema operativo. Dada la complejidad del tema no podemos pretender comenzar abordando un sistema tan acabado como MINIX. Por el contrario hemos de conformarnos con un sistema más humilde y transparente que nos permita alcanzar los objetivos expuestos en el apartado anterior. El sistema 'SO' que va a utilizarse es lo suficientemente sencillo como para poder trabajar en él con la máquina virtual "DosBox" sin necesidad de ninguna herramienta adicional, aparte del entorno de Turbo C utilizado desde la primera práctica.

3. TRABAJO A REALIZAR

Para empezar hemos de entender el funcionamiento del sistema operativo 'SO', tanto a nivel de usuario, introduciendo comandos a través de la consola o haciendo llamadas al sistema desde un programa, como a nivel de la estructura de dicho programa 'SO' en correspondencia con la de un *sistema monolítico*. En relación con este segundo nivel se mostrará la implementación concreta de los procesos (descriptores, tabla de procesos, cola de preparados, planificador y despachador) y de la gestión de memoria (particiones fijas).

A continuación se emprenderá el camino de las ampliaciones del sistema 'SO' original. Comenzaremos ilustrando la gestión de las interrupciones mediante la escritura de una rutina de tratamiento para la interrupción del teclado generada por la combinación de teclas Ctrl-C. Seguidamente trataremos brevemente las excepciones, escribiendo las rutinas de tratamiento de las excepciones de división por cero y de desbordamiento aritmético (instrucción `INTO`). A partir de ahí vamos a concentrarnos en las interrupciones software (instrucción `INT`) que son el mecanismo mediante el cual los programas de usuario invocan las llamadas al sistema.

La primera llamada al sistema que se implementará será sleep, que permite a un proceso de usuario bloquearse durante un cierto intervalo de tiempo. En su implementación el alumno deberá alterar la declaración de los descriptores de proceso, modificar la rutina de tratamiento de la interrupción de reloj y escribir el manejador de la nueva llamada al sistema.

Lo siguiente será implementar el grupo de llamadas al sistema: ***iniSemaforo***, ***bajaSemaforo*** y ***subeSemaforo***, que corresponden a las operaciones del tipo abstracto de datos de los semáforos, dotando así al sistema 'SO' de sus primeras primitivas de sincronización. Será necesario definir en el núcleo del sistema las estructuras de datos que representen a los semáforos. Los manejadores de las llamadas al sistema ***bajaSemaforo*** y ***subeSemaforo*** invocarán al planificador y dispatcher del sistema con el fin de bloquear y desbloquear correctamente a los procesos.

4. RELACIÓN DE ACTIVIDADES A DESARROLLAR

Actividad 1: El sistema 'SO' a nivel de usuario

Actividad 2: El sistema 'SO' a nivel de programación de aplicaciones

Actividad 3: Estructura e implementación del sistema 'SO'

Actividad 4: Implementación de la interrupción Ctrl-C.

Actividad 5: Compilación del sistema 'SO'

Actividad 6: Implementación de excepciones

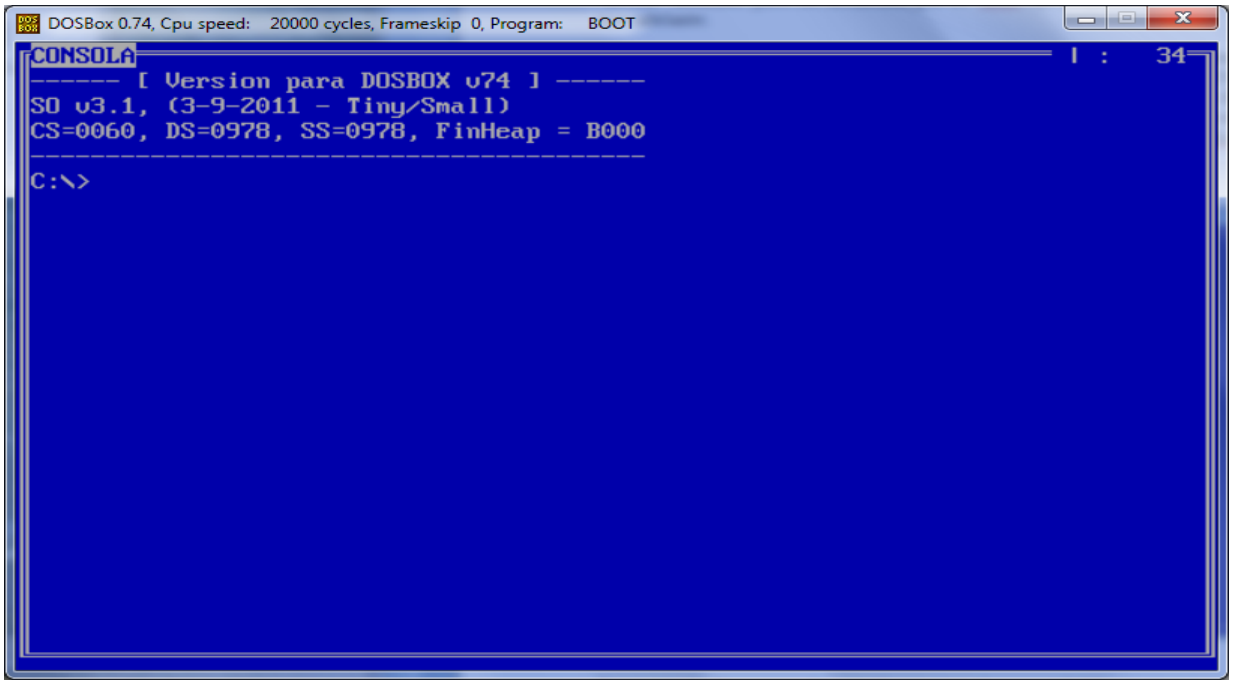
Actividad 7: Implementación de la llamada al sistema ***sleep***

Actividad 8: Implementación de los semáforos

5. ACTIVIDAD 1: El sistema 'SO' en el de nivel de usuario

Los fuentes del sistema 'SO' se encuentran en el directorio virtual "`c:\miso`". La generación del fichero ejecutable para DOS "so.exe", en sus modalidades de depuración integrada, depuración para el Turbo Debugger y sin depuración, así como la creación del fichero binario "so.bin" utilizado para ser arrancado por el "boot" de disquete, es muy sencilla y se explicará mas adelante.

Vamos a ejecutar por primera vez el sistema 'SO', para lo cual primeramente arrancamos el entorno D-Fend y ejecutamos cualquiera de los dos perfiles "DBxBoot-SO" o MiBoot_SO". Ambos perfiles inician el sistema 'SO' contenido en una imagen de disquete, en la cual además se encuentran algunos programas de usuario. La única diferencia entre ambas consiste en el boot de inicio del disquete. En una de ellas es el estándar de DosBox y en la otra es el boot de prácticas. Aparecerá la siguiente pantalla:



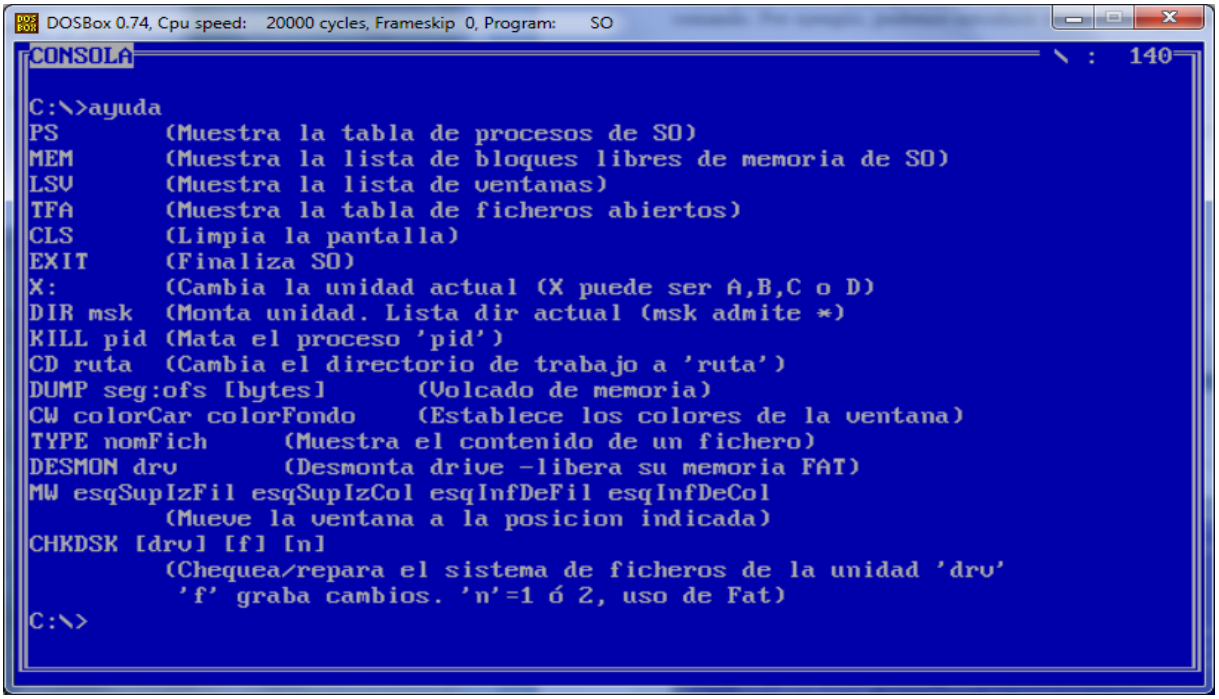
Como puede observarse, la consola muestra el *prompt* C:\> y espera a que se introduzca un comando. Por ejemplo, podemos introducir el comando siguiente:

`C:\>`**`cls <┘`**

que permite borrar la pantalla de la consola (*clear screen*).

Puede obtenerse la lista completa de los comandos internos de 'SO' introduciendo el comando "*ayuda"* ante el *prompt*:

`C:\>`**`ayuda <┘`**

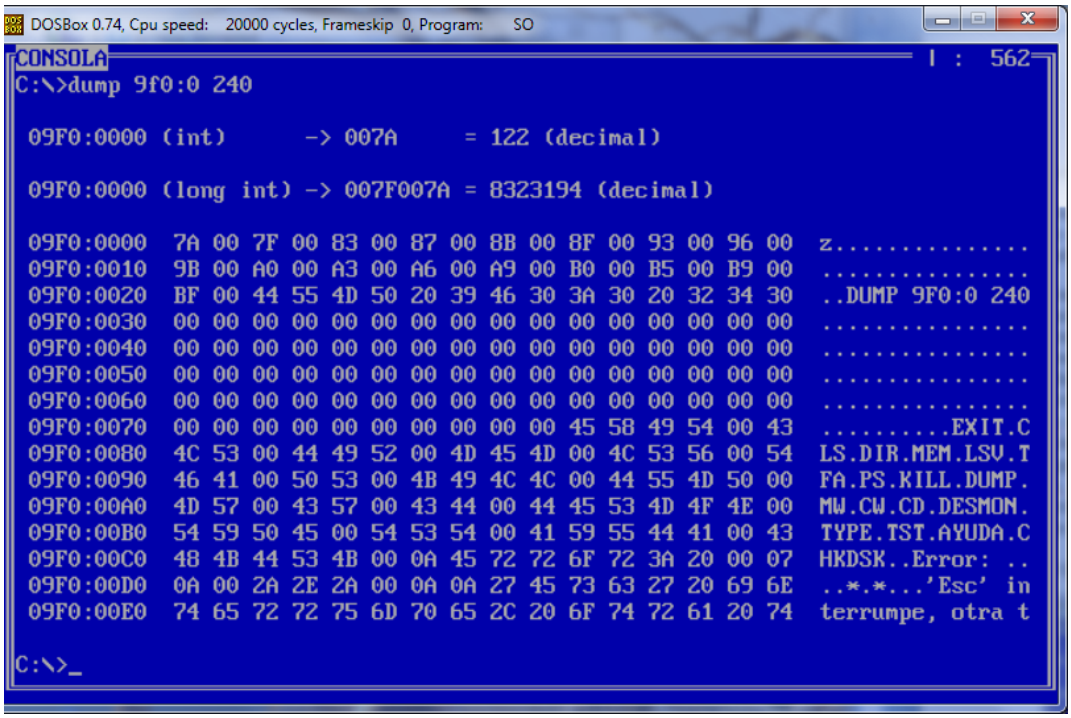


La denominación de *comandos internos* alude a que son comandos que el intérprete ejecuta cediendo el control a una función que forma parte del código del intérprete. Un *comando externo* sería uno que el intérprete ejecuta leyendo un fichero ejecutable, cargándolo en la memoria y cediéndole el control. La desventaja de los comandos externos es que tienen un peor tiempo de respuesta, al tener que leerse su código desde el disco. Su gran ventaja es que el código correspondiente a esos comandos puede actualizarse inmediatamente sin más que sustituir el fichero correspondiente por la última versión del comando, evitando así tener que recompilar el intérprete. En Linux podemos encontrar los comandos externos más utilizados en los directorios `/bin`, `/sbin` y `/usr/bin`.

'SO' es un sistema con multiprogramación, lo que queda patente ejecutando el comando `ps`, el cual muestra el estado de los procesos que están residiendo en la memoria. En este momento sólo hay dos procesos, que se corresponden con el proceso servidor (servicios relacionados con ficheros) y al intérprete de comandos de la consola de 'SO', los cuales forman parte del propio sistema 'SO' cargado en la dirección 0x00600, o segmento **0x60**.

El comando '*mem*' muestra el espacio disponible de memoria para la carga de programas en forma de una lista encadenada llamada "de huecos". Inicialmente toda la memoria a partir de donde finaliza 'SO' hasta el segmento dirección A000, estará disponible para programas y se reportará en forma de un solo hueco, por ejemplo: 2A5D (comienzo segmento), 85A3 (tamaño del hueco en "*paragraphs*" - unidades de 16 bytes) y por último la dirección del siguiente hueco: NIL.

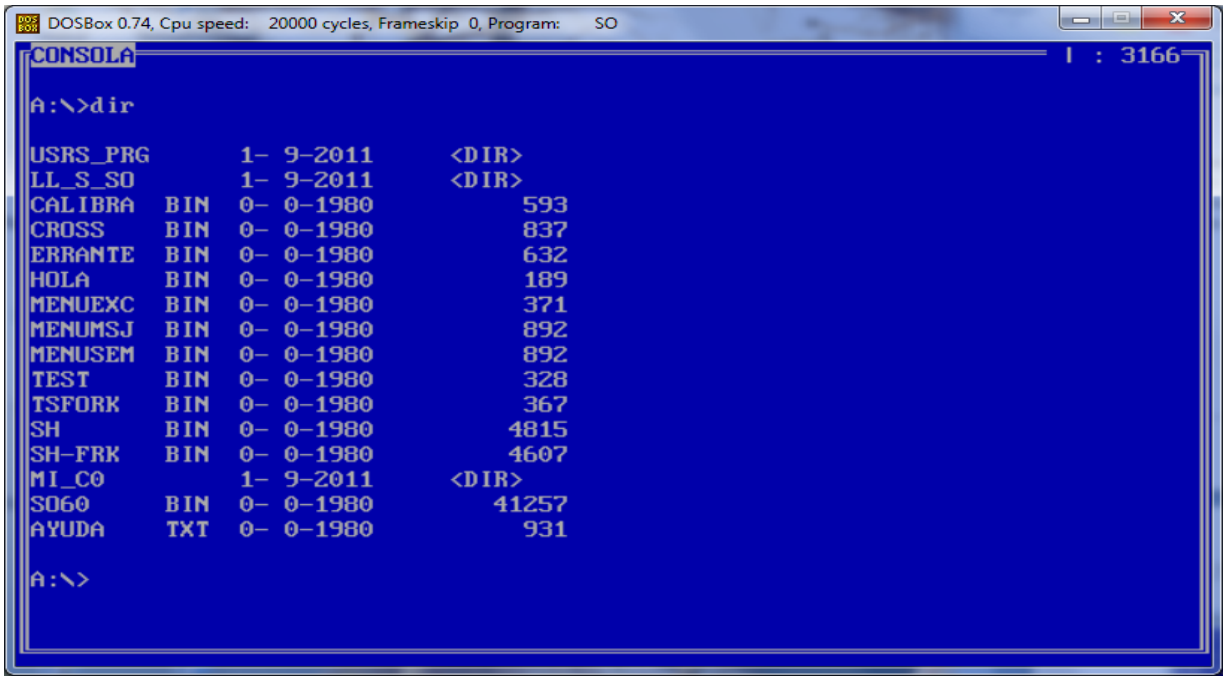
El comando '*dump*' (volcado de memoria) permite visualizar partes de la memoria del mismo modo que el programa "*ver_fich.exe*" visto anteriormente mostraba partes de un fichero binario. Si tecleamos por ejemplo el comando '*dump*' siguiente, obtendremos una pan talla similar a la siguiente:



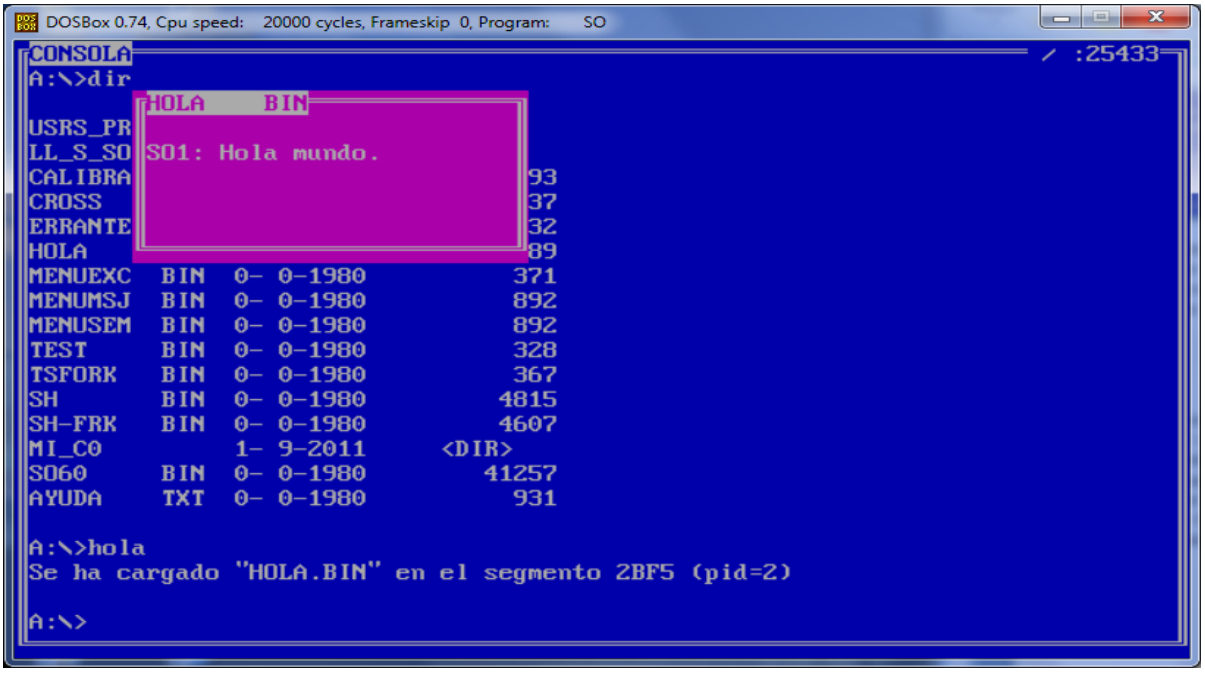
La dirección `09F0:0` es el principio del segmento de datos (DS) del programa 'SO'.

Como curiosidad en la columna de la derecha se reconocen varias cadenas de caracteres (cuyo fin está indicado por el byte 00) como son las correspondientes a los comandos internos mostrados por el comando “ayuda”.

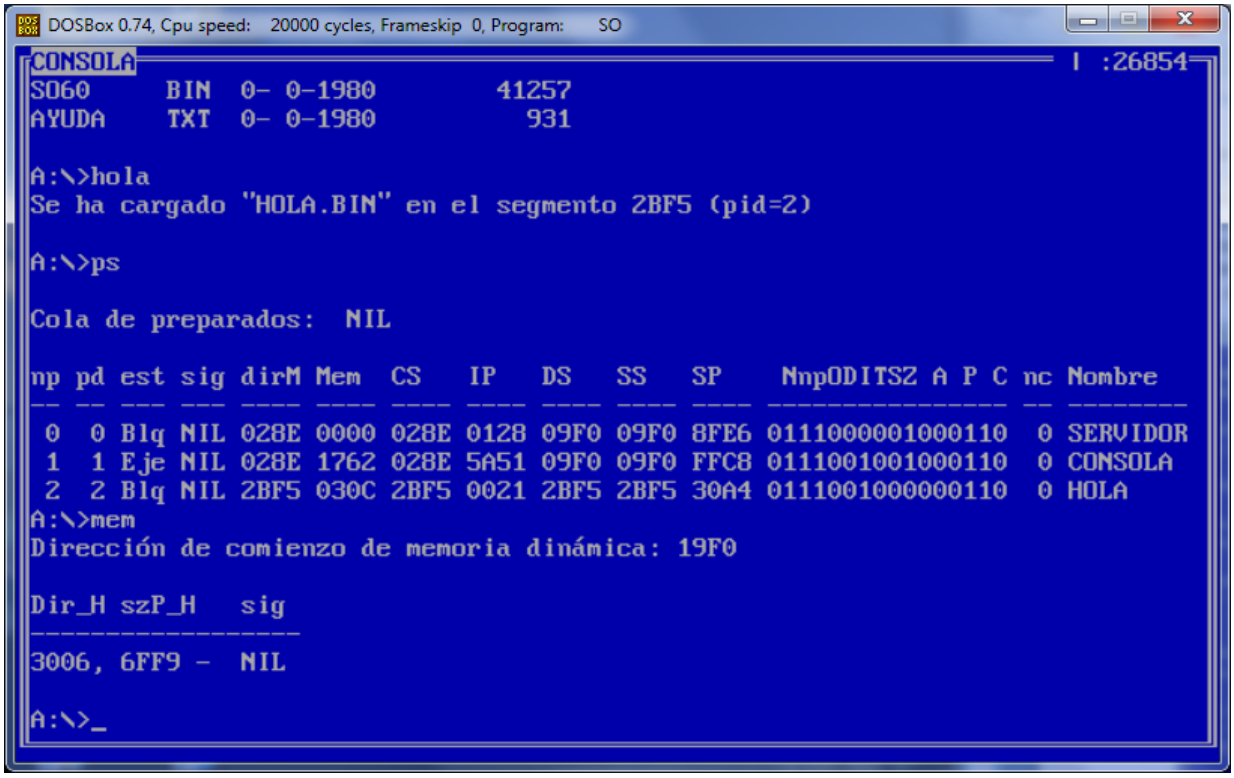
Veamos ahora cómo podemos ejecutar algún programa presente en el sistema de ficheros, el cual podría implementar algún comando externo de ‘SO’. Con ese fin lo más rápido es cambiar primero la unidad actual por la “A”, para ello tecleamos A:<intro> y seguidamente usamos el comando interno *dir* el cual nos mostrará el contenido del disquete:



En este caso vemos que hay varios ficheros en el disquete, uno de ellos: "so60.bin", se corresponde con el sistema ‘SO’ cargado durante el arranque, los demás ficheros son programas de usuario que hacen uso de diferentes llamadas a sistema, algunas de las cuales habrá que realizar en estas prácticas. Si ejecutamos el programa “hola”, obtenemos lo siguiente:

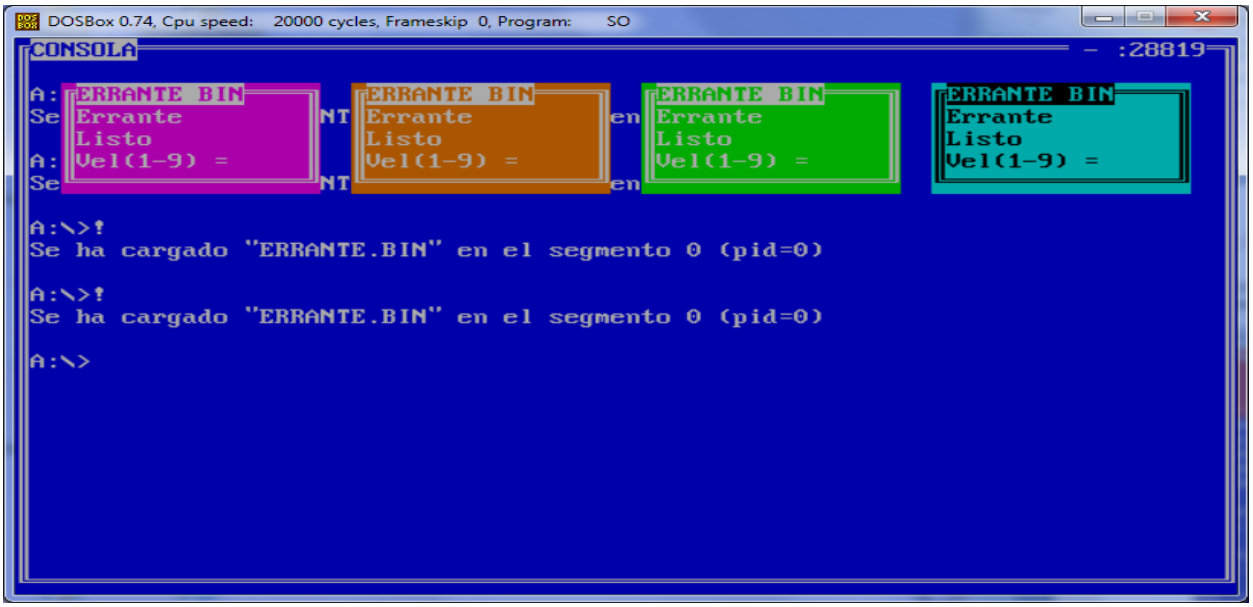


Vemos que el sistema 'SO' ha cargado el fichero "*hola.bin*" desde el disquete y lo ha ubicado en el segmento **2BF5**, creando un nuevo proceso cuyo ***pid*** (identificador de proceso) es 2. Además se ha abierto una ventana (violeta en este caso) que sirve de vía de comunicación entre el usuario y el proceso. La ventana no tiene el foco del teclado ya que el cursor "_" lo sigue mostrando la consola. Para tomar el foco del teclado pulsamos la tecla **TAB** que sirve para cambiar dicho foco entre todas las ventanas (procesos) en ejecución. Una vez retomado el foco, si ejecutamos los comandos '*ps'* y '*mem'*. Veremos la siguiente pantalla:

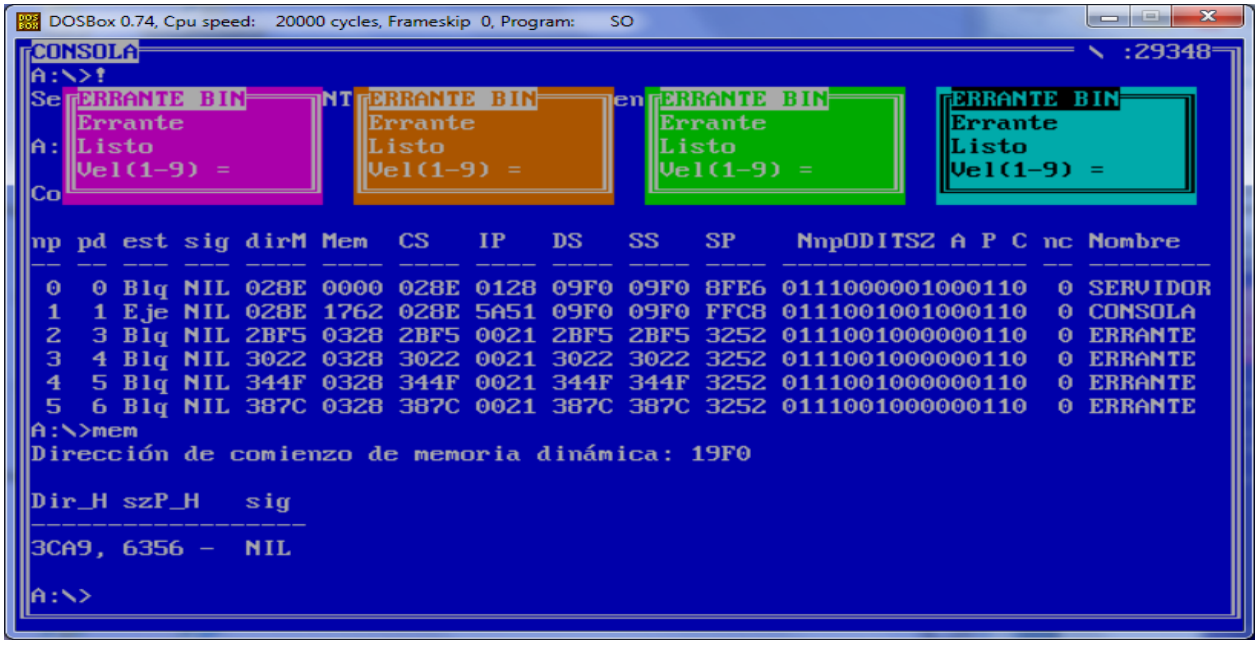


Las combinaciones de teclas **SHIFT IZDA-TAB** y/o **SHIFT DCHA-TAB** sirven para cambiar el plano de la ventana sin cambiar el foco del teclado. Si arrancamos mas procesos podemos probar estas teclas y comprobar su funcionamiento. El programa "*hola*" finaliza con la pulsación de una tecla, pudiendo comprobarse con '*ps'* cómo desaparece el proceso y con '*mem*' si se ha generado un nuevo hueco en el lugar que ocupaba dicho proceso o se ha fusionado con alguno ya existente. Por último, respecto al uso de teclas especiales, hay que decir que las teclas de cursor "⇦", "⇨", "⇧" y "⇩", sirven para mover la ventana que aparece en primer plano y si se usan conjuntamente con la tecla **SHIFT** *(Mayus)* redimensionan la ventana.

Vamos a mostrar ahora la ejecución de varios procesos concurrentemente. Con este fin introducimos desde la consola cuatro veces el comando externo "*errante*" (el carácter '**!**' tiene la función especial de repetir el último comando). Recuérdese el uso de **TAB** para retomar el foco y las teclas de flecha de cursor para mover las ventanas:



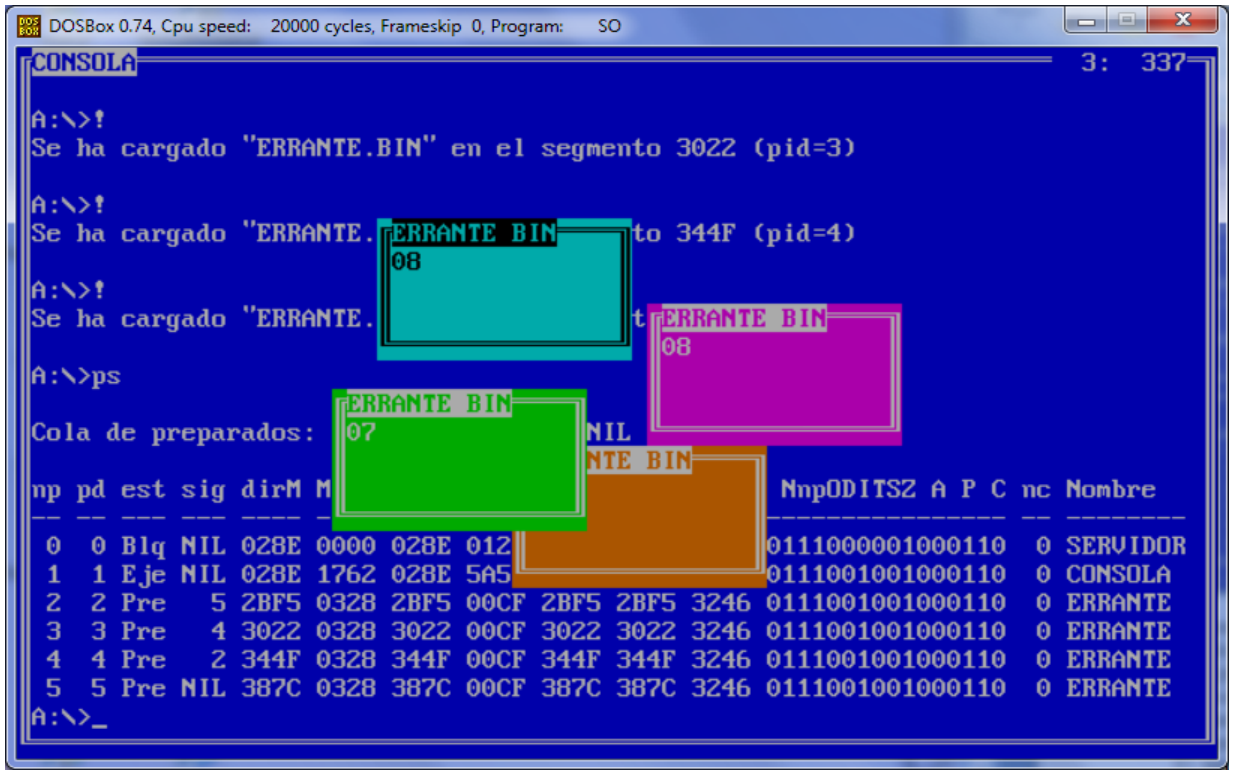
Si ejecutamos los comandos '*ps*' y '*mem*' obtenemos:



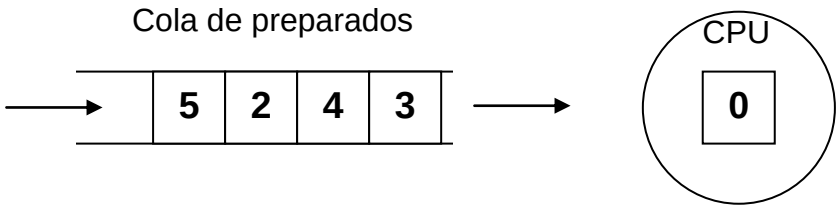
Pueden apreciarse los espacios de memoria que han ido ocupando los procesos "*errante*", la memoria que ocupa cada uno de ellos y el hueco de memoria final resultante.

La función del programa "errante" es muy simple, se limita a solicitar un número entre 1 y 9 y a continuación comienza a desplazarse por la pantalla a una determinada velocidad que viene indicada por el número introducido, cuanto más alto más rápido se mueve. Este programa hace uso reiterado de la llamada a sistema "*moverWindow*", y tras realizar una serie de vueltas, acaba parado en la esquina inferior izquierda, muestra un mensaje y finaliza.

Con los programas "errante" cargados en memoria y listos para empezar su función, tecleamos en cada uno de ellos un número entre 1 y 9 y seguidamente vamos a la consola y ejecutamos el comando '*ps*', obteniendo algo similar a esta pantalla:



Es interesante observar el estado de los procesos mientras se ejecutan haciendo uso del comando '*ps'* desde la consola. En la pantalla anterior se aprecia que se estaba ejecutando la consola (procesando el comando '*ps*') y que el resto de procesos estaban preparados para ejecutarse, para proseguir con la ejecución de su movimiento. Podemos deducir el estado de la cola de preparados a partir del valor de los campos '`sig`' de los descriptores de los procesos. Vemos que el siguiente del proceso ***np = 5*** es NIL, lo que indica que se trata del último proceso preparado. El siguiente del proceso 2 es 5, lo que indica que el proceso 2 es el penúltimo de la cola. El siguiente del proceso 4 es 2, por lo que el proceso 4 es el segundo proceso de la cola. Finalmente el proceso 3 tiene como siguiente al proceso 4, por lo que el proceso 3 es el primer proceso de la cola de preparados.



El comando '*kill*' permite destruir un proceso identificado por su '*pid'*. Por ejemplo desde la consola podemos matar cualquier proceso errante antes de que finalice éste, ejecutando por ejemplo: `c:\>kill 3` ↵

6. ACTIVIDAD 2: El sistema 'SO' en el nivel de programación de aplicaciones

Esta actividad se realizará usando principalmente el perfil de D-Fend "DBOX-SO", en el que tenemos el entorno de desarrollo ya familiar de TC30 y demás utilidades, y con el que se han desarrollado las prácticas anteriores, aunque en esta ocasión utilizaremos fundamentalmente el directorio *c:\miso* donde se encuentra el fuente del sistema 'SO', así como también el subdirectorio *usrs_prg*, en el que se encuentran los programas de usuario necesarios y el subdirectorio *ll_s_so* con los ficheros que forman la librería de interfaz de llamadas al sistema.

A nivel de programación de aplicaciones, el sistema 'SO' nos ofrece el repertorio de llamadas contenido en los ficheros del directorio *"c:\miso\ll_s_so"*, que podemos utilizar desde cualquier programa diseñado para ejecutarse en el sistema 'SO' como un proceso de usuario. Como ejemplo a continuación se muestra el contenido del fichero "*bsic-ifz.c*", el cual contiene un conjunto de llamadas básicas generales:

```
/* --------------------------- finProceso() ----------------------------
   Esta llamada al sistema es llamada cuando un proceso quiere finalizar
   ---------------------------------------------------------------------- */
void finProceso (void) {
  asm { MOV AH,0; INT VINT_SO }
}

/* ------------------------- matarProceso()-----------------------------
   Esta llamada al sistema elimina al proceso cuyo identificador de
   proceso es pid. No se permite matar al proceso con pid == 0 que
   corresponde a la consola del sistema SO. La ventana del proceso se
   cierra y la memoria que ocupa el proceso queda libre para cargar otro.
   Si se detecta algun error la funcion devuelve un -1, sino 0.
   ---------------------------------------------------------------------- */
int matarProceso (word_t pid) {
  asm { MOV AH,1; MOV BX,pid; INT VINT_SO }  return _AL;
}

/* ------------------------- leerTecla() -------------------------------
   Esta llamada al sistema lee un caracter del teclado virtual del
   proceso. Cada proceso tiene su propio teclado virtual que corresponde
   al teclado fisico cuando la ventana del proceso esta seleccionada.
   Si al ejecutar leerTecla() no hay ningun caracter disponible en el
   bufer del teclado fisico o en el bufer del teclado virtual, el proceso
   que hace la llamada se bloquea hasta que se presione una nueva tecla
   estando la ventana del proceso seleccionada. La funcion devuelve como
   resultado el caracter ascii correspondiente a la tecla pulsada.
   ---------------------------------------------------------------------- */
char leerTecla (void) {
  asm { MOV AX,0x0200; INT VINT_SO } return _AL;
}

char leerTeclaLista (void) {
  asm { MOV AX,0x0201; INT VINT_SO } return _AL;
}

/* ------------------------- printCar() --------------------------------
   Esta llamada al sistema escribe un caracter ascii en la ventana del
   proceso en la posicion actual del cursor.
   ----------------------------------------------------------------------*/
void printCar (char car) {
  asm { MOV AH,3; MOV BL, car; INT VINT_SO }
}
```

```
#define printDec(num,l) printBase(num,10,l)
#define printHex(num,l) printBase(num,16,l)
#define printStr(str)   printStrHasta(str,0xFFFF)

void printStrHasta (char far *str, word_t lon) {
  asm { MOV AH,4; LES SI,str; MOV CX,lon; INT VINT_SO }
}

void printBase (word_t num, byte_t base, byte_t lon){
  asm { MOV AH,5; MOV AL,base; MOV BX,num; MOV CL,lon; INT VINT_SO }
}

/* --------------------------- moverWindow() ----------------------------
   Esta llamada al sistema cambia los limites de la ventana del proceso
   correspondientes al numero de la primera fila, el numero de la primera
   columna, el numero de la ultima fila y el numero de ultima columna.
   Los numeros de fila deben de estar en el rango 0..24 y los numeros de
   columna deben estar en el rango 0..79. En otro caso o si la ventana
   resulta ser demasiado peque¤a, la funcion no tiene ningun efecto.
   Tras moverse la ventana se borra su contenido. Devuelve 0 si se pudo
   pudo llevar a cabo la operacion con exito o 0 en caso contrario.
   ------------------------------------------------------------------- */
void moverWindow (char esqSupIzF, char esqSupIzC,
                  char esqInfDeF, char esqInfDeC) {
  asm { MOV AH,6;
        MOV BL,esqSupIzF;  MOV BH,esqSupIzC;
        MOV CL,esqInfDeF;  MOV CH,esqInfDeC; INT VINT_SO }
}

/* -------------------------- colorWindow() ----------------------------
   Esta llamada al sistema cambia el color que se utiliza al visualizar
   los caracteres en la pantalla, tanto el color del caracter en si como
   el color de fondo utilizado. Los valores concretos de los colores
   pueden encontrarse en el fichero "colores.h". Esta funcion no devuelve
   ningun resultado.
   ------------------------------------------------------------------- */
void colorWindow (char colorCar, char colorFondo) {
  asm { MOV AH,7; MOV BH,colorCar; MOV BL,colorFondo; INT VINT_SO }
}

void fotoWin (bool salvar) {
  asm { MOV AH,8; MOV BL,salvar; INT VINT_SO }
}

// --- Funciones auxiliares de SO. Utilizan otro vector ----
void leerLinea (char far *lin, word_t size, bool mayus) {
  asm { MOV AH,0; LES BX,lin; MOV CX,size; MOV AL, mayus; INT VINT_SO+1 }
}
```

La implementación de estas funciones de interfaz de llamadas al sistema se limita a:

- Pasar los parámetros que reciben en la pila a los registros donde el sistema operativo ‘SO’ espera encontrarlos,

- Pasar al sistema operativo el código de la llamada al sistema en cuestión en el registro `AH` (byte alto del registro acumulador `AX`) del 8086,

- Realizar la llamada al sistema operativo 'SO' mediante la correspondiente instrucción `INT` del 8086, y

- Devolver, tras el retorno de la llamada, los resultados que corresponda.

Es conveniente que el alumno relacione esto con lo visto en la teoría, por lo que se reproduce aquí la figura 1-17 del libro de *Tanenbaum*.

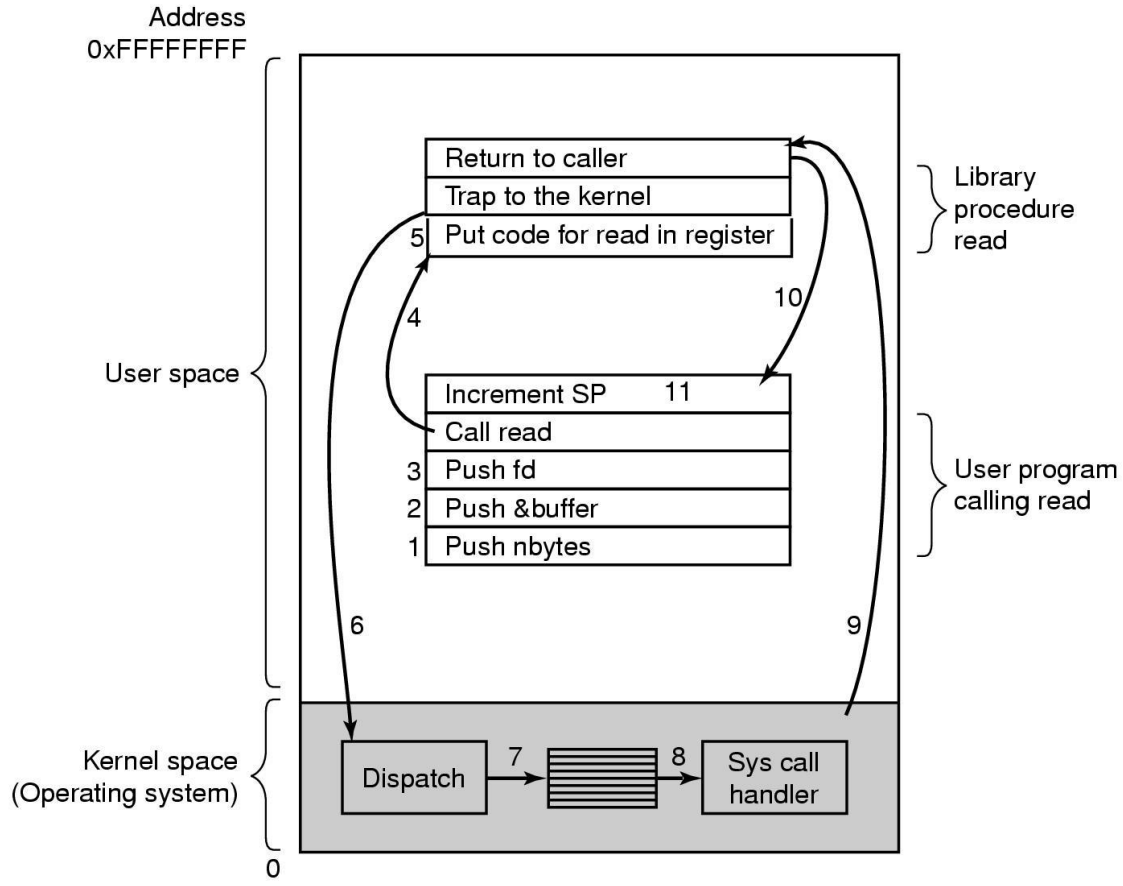


Figura 1-17. Los 11 pasos para hacer la llamada al sistema read(fd, &buffer, nby-

Si vemos ahora por ejemplo la implementación del procedimiento de librería `matarProceso`, seguramente quedará todo más claro:

```
int matarProceso (word_t pid) {
  asm { MOV AH,1; MOV BX,pid; INT VINT_SO }  return _AL;
}
```

El paso 5 de la figura corresponde a la instrucción `MOV AH,1` ya que el sistema 'SO' interpreta el código 1 como que el programa de usuario desea llevar a cabo la llamada al sistema *matarProceso*, la cual mata a un proceso. El paso 6 de la figura corresponde a la ejecución de la instrucción `INT VINT_SO`, donde `VINT_SO` es un símbolo definido como `0x60`, que es el número del vector de interrupción utilizado por el sistema 'SO'. Finalmente el paso 10 de la figura se corresponde con la instrucción de retorno `return _AL`.

Otro aspecto a señalar respecto a las llamadas al sistema y procesos de usuario es el fichero "*inic_usr.c*". Este fichero contiene el código de inicialización necesario para generar los programas de usuario, así como el interfaz de llamadas al sistema básico, el cual se haya incluido dentro de él mediante la directiva `#include "..\ll_s_so\bsic-ifz.c"`.

Su contenido es el siguiente:

```
/* ------------------------------------------------------------- */
/*      Funciones de interfaz de llamadas al sistema básicas     */
/* ------------------------------------------------------------- */

asm DGROUP GROUP _TEXT, _DATA
/* Esta directiva es vital para que el compilador genere codigo en
   el que las variables, que usan el segmento DS, tomen un offset
   adecuado, es decir, para que DS y CS sean iguales (mismo segmento).
   Si se omite, se asume que DS va a tomar un valor distinto a CS,
   concretamente CS + tama¤o codigo en paragraphs, o sea, modelo small.
   Todo esto es debido a que compilamos desde linea de comando no
   desde el entorno integrado y sin el codigo inicial C0.obj, el cual
   se encarga entre otras cosas de ello. Los offset de las variables
   empienzan donde acaban los de las funciones. Si se usa TCC 3.0 hay
   que usar la opcion -B para que use TASM en vez de el de inline BASM */

#include "..\tipos.h"

void main (void);              /* declaraciones forward */
void finProceso (void);

/* ---------------------------- start() ---------------------------------
   Esta funcion debe estar situada al principio de cualquier otra funcion
   de todo proceso de usuario, ya que en ella se invoca a la funcion main()
   y despues se efectua la llamada a sistema de finalizacion de proceso */
void start (void) { main(); finProceso(); }
```

#include "..\ll_s_so\bsic-ifz.c"

Este fichero contiene la función `start`, la cual debe ser la primera función en el fichero y dado que hace uso de las funciones `main` y `finProceso`, éstas deben de estar declaradas anteriormente como prototipos de 'C'. El propósito de ***start*** es servir de código inicial de todo proceso de usuario. Este código inicial lo que hace es invocar a la función `main` y tras finalizar ésta, realizar la llamada al sistema `finProceso`, de este modo las aplicaciones de usuario tienen garantizado arrancar por la función `main`, y también que al finalizar ésta, se invoque la llamada al sistema `finProceso`. Por supuesto es condición necesaria que los programas de usuario tengan al principio de todo, como su primer directiva "include", precisamente la de este fichero: *#include "..\ll_s_so\inic-usr.c"* , ya que es el modo más sencillo de garantizar que la primera función que el compilador encuentre sea precisamente la función *`start`*. Por último conviene aclarar un poco el uso de la directiva `asm DGROUP GROUP _TEXT, _DATA` que aparece en el fichero "*inic-usr.c*". Esta directiva se usa para informar al compilador que debe usar un solo segmento para código y datos, al igual que se hace en el modelo '***tiny***' de 'turboC'. Con ese modelo se simplifica el funcionamiento del arranque de las aplicaciones de usuario por parte de 'SO', ya que los segmentos CS, DS y SS tienen así el mismo valor inicial que es la dirección de memoria donde se carga del proceso.

Una vez presentada la interfaz de programación con el sistema operativo es conveniente que nos ejercitemos en su utilización poniendo a punto un programa de usuario para el sistema operativo

'SO'. Como viene siendo habitual, vamos a empezar por el programa de usuario para el sistema 'SO', "*Hola mundo*", cuyo fuente se encuentra en el fichero "*hola.c*" dentro del directorio: "*c:\miso\usrs_prg\hola*". Una cosa que debe quedar clara es que los programas ejecutables de 'SO', en general, no van a funcionar en otros sistemas operativos, ya que las llamadas al sistema en los que se basan hacen uso de la instrucción de *interrupción software* (*trap*) `INT 60h`, y el vector de interrupción `60h`, no tiene porqué soportar las funciones de servicio de llamadas al sistema características de 'SO'. Seguidamente se muestra el código de este programa:

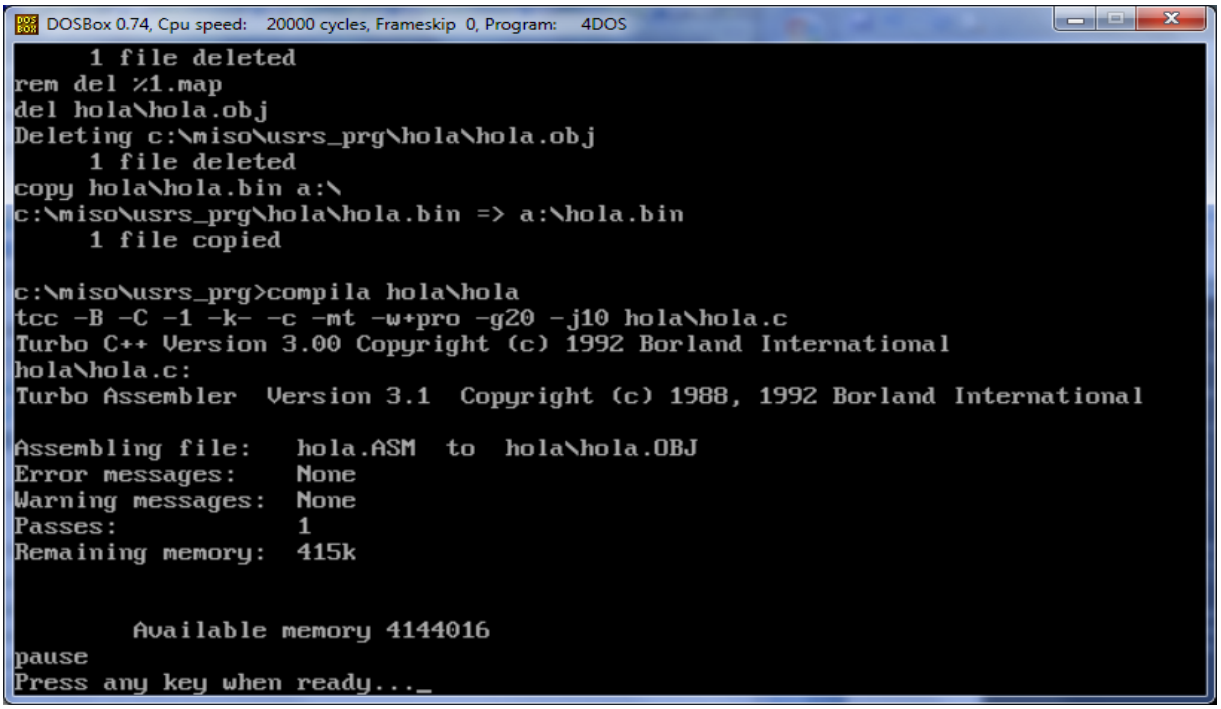
```
/* ------------------------------------------------------------------ */
/*        hola.c (programa Hola mundo para el sistema 'SO')           */
/* ------------------------------------------------------------------ */

#include "..\ll_s_so\inic-usr.c" //Cod.inic.prog.usuario y llam. basicas

void main () {
  printStr("\nSO: Hola mundo.\n"); leerTecla() ;
}
```

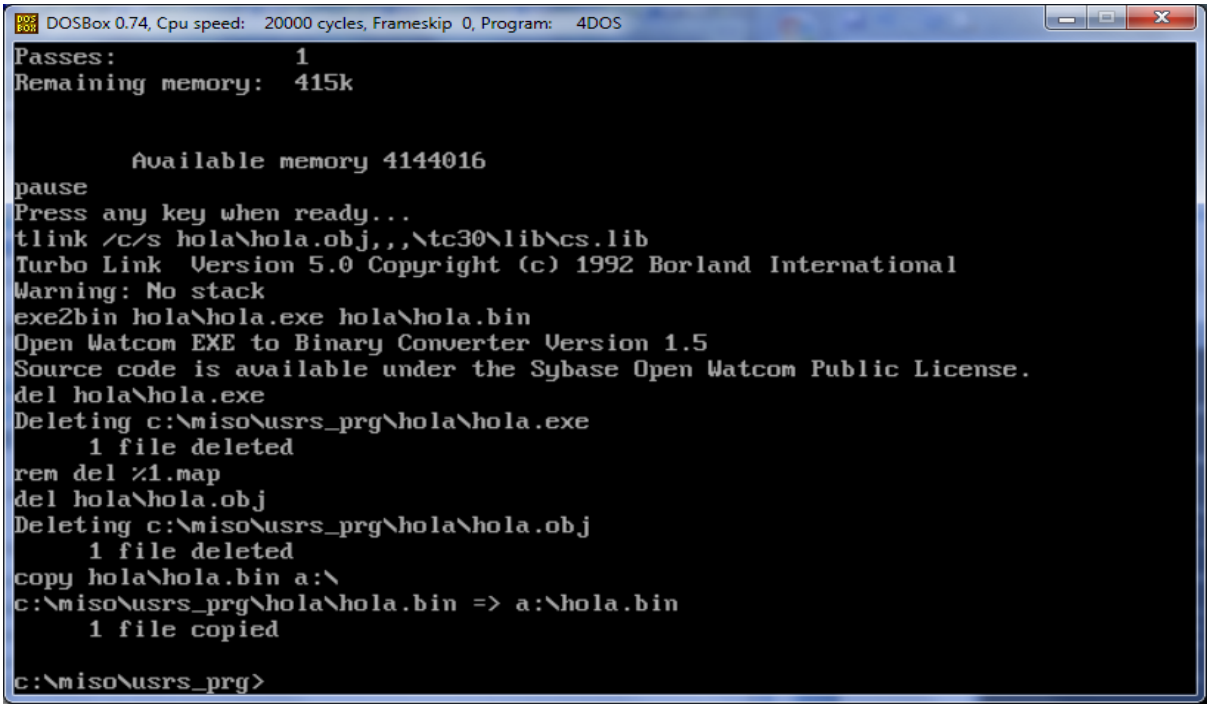
Un pequeño detalle en cuanto a la función `main` es que debe declararse como una función sin parámetros que devuelve un valor de tipo `void`. Después de escribirse la cadena de caracteres `"SO: Hola mundo."` se espera a que se pulse una tecla, con el fin de poder ver tranquilamente lo que se ha escrito. Finalmente el proceso sale de `main` y finaliza.

Para compilar y generar el ejecutable del programa "*hola.c*" la forma más sencilla es situarse en el directorio *c:\miso\usrs_prg* y ejecutar el siguiente comando:

```
c:\miso\usrs_prg>compila hola\hola <┘
```

Este comando se ejecuta en dos partes separadas por una pausa que exige la pulsación de una tecla por parte del usuario. La razón de esta pausa es para poder comprobar si hubo errores de compilación, ya que sin ella los mensajes de error desaparecerían por la parte superior de la pantalla. Tras la pulsación de dicha tecla habremos obtenido las siguientes pantallas:

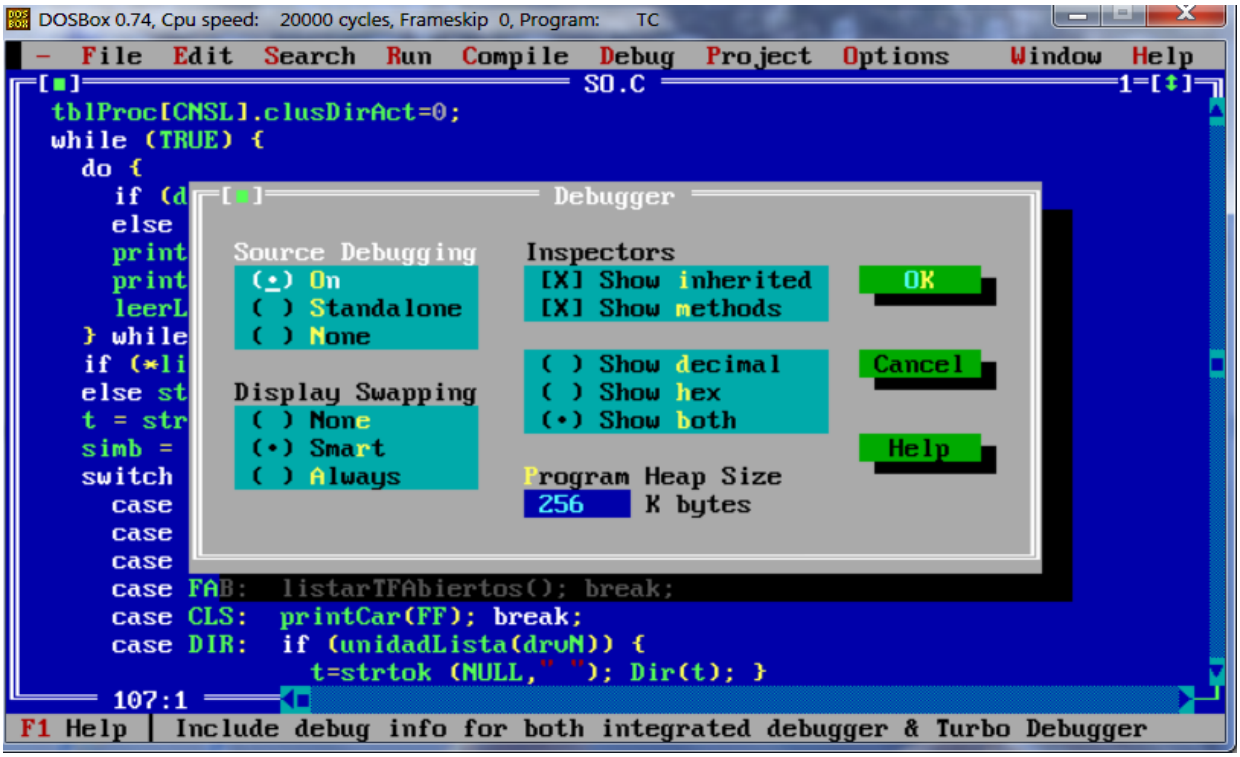




Pantallas con el resultado de la compilación.

El comando compila es un fichero '*.bat*' típico de MSDOS (procesamiento por lotes) que invoca al compilador de línea de comando '*tcc*' y al montador '*tlink*', con una serie de parámetros ya establecidos y probados. Con estos dos programas se genera un fichero ejecutable '*.exe*' apto en principio para MSDOS, pero no para el sistema 'SO', (por tener una cabecera, etc.), por ello el comando también invoca a la utilidad '*exe2bin*', que se encarga de quitar la cabecera y convertir el fichero a un formato binario '*.bin*'. El resultado final debe ser la creación del fichero '*hola.bin'* que se encontrará en el directorio USRS_PRG\HOLA dentro de MISO. Este fichero también se copia a la unidad de disquete "A:"

Para probar el programa hay que arrancar 'SO' y ejecutarlo desde él. El sistema 'SO' se puede arrancar desde el perfil visto en la actividad anterior '*DBxBoot-SO*', sin embargo es más sencillo seguir usando el perfil que se está usando y aprovechar la característica de que el sistema 'SO' también puede ser arrancado desde el propio DOS e incluso desde el propio entorno de desarrollo de *TurboC*. Arrancar 'SO' desde este perfil es muy sencillo, simplemente nos situamos en el directorio *c:\miso* y tecleamos *so*, ya que el sistema se encuentra también compilado y en forma de ejecutable para MSDOS (*so.exe*). Otra posibilidad consiste en arrancar 'SO' desde el entorno *TurboC*. Desde este entorno podemos abrir el proyecto 'so.prj' y utilizar las teclas de F9/ctrl F9 para compilarlo/ejecutarlo. Usar esta posibilidad nos permite incluso utilizar el *debugger* integrado para depurar el sistema 'SO'. El compilador de *TurboC* nos permite generar el ejecutable, sin código de depuración o con código de depuración incluido para usarse desde el propio IDE o desde el programa externo *TurboDebugger*.



Pantalla de selección de modo depuración integrado.

El programa "*hola.c*" sólo utiliza 3 de las 6 llamadas al sistema que ofrece 'SO'. Vamos a proporcionar un segundo programa algo más completo. Se trata de un programa, "*errante.c*", que implementa un proceso de usuario errante que da un montón de vueltas por la pantalla antes de morir.

El fichero "*errante.c*" está disponible en el mismo directorio `C:\MISO\USRS_PRG\ERRANTE`, donde el alumno puede editarlo, compilarlo y ejecutarlo de la misma manera que hizo con el programa '*hola.c*'. El alumno deberá cargar en memoria cuatro procesos errantes, poniéndolos todos a funcionar simultáneamente presionando en secuencia *"TAB", "6", "TAB", "7", "TAB", "8", "TAB" y "9"*. Después de esto, podremos ver (al igual que antes) a los cuatro procesos con sus ventanas desplazándose por la pantalla describiendo un movimiento del tipo espiral cuadrada a diferentes velocidades. Se sugiere al alumno que utilice desde la consola el comando "*ps*" para observar cómo se alternan los procesos errantes en el uso de la CPU al más puro estilo *Round-Robin*.

7. ACTIVIDAD 3: Estructura e implementación del sistema 'SO'

Una vez descrita la máquina virtual que nos ofrece el sistema 'SO', vamos a pasar a comentarlo brevemente como programa. 'SO' es un sistema monolítico parcialmente estructurado. Los ficheros fuente del sistema están disponibles en el directorio C:\MISO. El fichero que contiene el programa principal '*so.c*', se ocupa de la inicialización del sistema (creación procesos consola y servidor), y eventualmente de la interpretación de los comandos de la consola.

En este apartado vamos a comentar brevemente cómo está implementada la gestión de memoria y la gestión de los procesos. En las siguientes actividades nos fijaremos en las rutinas de tratamiento de interrupciones y excepciones, así como en la implementación de las llamadas al sistema.

En cuanto a la gestión de la memoria, hay que decir que el método elegido en cuestión ha sido el de *particiones variables*. El sistema utiliza una lista para gestionar los espacios de memoria libres (huecos). Al principio la lista sólo tiene un elemento, el cual hace referencia a toda la memoria disponible (que el sistema previamente ha determinado). El sistema cuando por ejemplo crea un proceso, necesita ubicarlo en memoria y para ello utiliza unas funciones de gestión de memoria que se encargan de reservar y/o liberar cantidades de memoria arbitrarias (bloques) según se les solicite. También es posible reservar memoria para otros propósitos, como por ejemplo la tabla FAT del sistema de ficheros.

La estructura de datos que permite gestionar la memoria está en el fichero *'memoria.c'* y es la siguiente:

```
/* ------------------------------------------------------------
   Tipos de datos para crear una lista de huecos en la memoria
   disponible para programas. Los "links" se guardan en los
   propios huecos. Especifican el tamaño del hueco y un puntero
   al siguiente. Este puntero es del tipo "_seg" que es especial
   ya que tiene siempre un offset implicito 0 (no ocupa). Se hace
   una unión con un tipo word para operar mejor con el */

typedef struct foo _seg *pHue_t; // ptr. offset siempre 0 (2 bytes)

typedef union {
  pHue_t p;    /* se puede usar como puntero */
  word_t w;    /* sin offset (siempre 0) o como un word */
} uHue_t;
struct foo {  word_t tam;  uHue_t uSig; };
word_t iniHeap, finHeap; // comienzo y fin memoria disponible
PRIVATE uHue_t priHue;      // primer elemento de la lista huecos
PRIVATE word_t memDisponible;   // en paragraphs
```

En el fichero *'memoria.h'* tenemos los prototipos de las funciones de gestión de memoria "`tomaMem()`" y "`sueltaMem()`" ya comentadas, además de otras como "`memBIOS()`" para determinar la memoria disponible que reporta la BIOS, "`inicializarMemoria()`", para inicializar las estructuras de datos empleadas, "`inc`()" para incrementar en una cantidad un puntero largo, y por último "`volcar()` y `mostrarMemoria()`" para mostrar en pantalla el contenido de la memoria. También se declara mediante #define CHUNK 768, la cantidad de *paragraphs* que se le asigna a todo proceso para datos + pila + buffer de teclas + video de ventana. Esta cantidad fija se suma a la que requiera el código del programa según su tamaño.

En cuanto a la gestión de los procesos la implementación es estándar tal y como se explica en la teoría, poniéndose en práctica el algoritmo de planificación *Round-Robin*. La parte de código correspondiente a las operaciones básicas con procesos está en el fichero `'procesos.c'`. Las definiciones básicas están en *"procesos.h"* y son:

```
#ifndef PROCESOS_H
/* ------------------------------------------------------------ */
/*                          procesos.h                          */
/* ------------------------------------------------------------ */

#include "windows.h"
#include "ficheros.h"

#define IDLE -1  /* num. proceso idle */
#define CNSL 1   /* num. proceso consola de 'SO'*/
#define SERV 0   /* num. proceso servidor */
```

```
#define PWIN_SO tblProc[CNSL].pWin  /* ptro. a ventana de la consola */
#define SRV_PILA 0x9000    /* origen de la pila para el servidor */
#define TICS_POR_RODAJA 2  /* num. tics por rodaja round robin */

typedef int pid_t;              // identificador de proceso

typedef enum {
  LIBRE, EJECUCION, PREPARADO, BLOQUEADO   /* estados de un proceso */
} estado_t;

typedef enum { // razon por la que un proceso esta bloqueado
  BLK_NO, BLK_TECLADO, BLK_SERV, BLK_PAUSE, BLK_OTROS
} esperaPor_t;

typedef cola_t far * fpCola;

#define MAX_TDF 10
typedef struct {
  estado_t     estado;   // LIBRE, EJECUCION, PREPARADO, BLOQUEADO...
  esperaPor_t  esperaPor;// En caso de bloqueo, razon del mismo
  fpCola       pColaBlk; // En caso de bloqueo, puntero a cola espera
  pid_t        pid,pPid; // Pid del proceso y pid del padre
  nproc_t      sig;      // Siguiente proceso de posible lista
  fpRegsPila_t sp;       // Los registros (status) se guardan en la pila
  word_t       dirMem;
  word_t       tamMem;   // Comienzo y tama¤o del bloque memoria
  pWin_t       pWin;     // Ventana y terminal del proceso
  byte_t       nombre[12];    // nombre fichero ejecutable
  long         tamFichero;    // tama¤o  "        "
  drv_t        drvWork;       // Drive actual (¢ de trabajo)
  word_t       clusWork;      // cluster del dir. de trabajo del proceso
  df_t         tdf[MAX_TDF];  // tabla de descriptores de fichero     @@
} descrProc_t; // descriptor de proceso

extern descrProc_t tblProc [MAX_PROCS];   /* tabla de procesos */
extern cola_t preparados;
extern cola_t aServir;
extern word_t  numProcesos;     // num. procesos en cada momento
extern nproc_t nprEjec,nprAtnd,nprAtBk;// procesos en ejecucion atendido
(+bak)
extern word_t contTicsRodaja;   // contador de tics de una rodaja

/* --------- funciones ------------*/

pid_t nuevoPid (void);
nproc_t nproc (pid_t pid);
void encolar (fpCola cola, nproc_t npr);
void colar (fpCola cola, nproc_t npr);
nproc_t desencolar (fpCola cola);
void quitarDeCola (fpCola cola, nproc_t npr, nproc_t nprAnt);
nproc_t buscarEnCola (fpCola cola, nproc_t npr, bool *pErr);
void activarProceso (nproc_t pid);
void activarPreparado (void);
void saveStack (void);
void restoreStack (void);
void bloquearProceso (esperaPor_t por, fpCola cola);
nproc_t crearProceso (word_t part, word_t size, char * name);
void inicProcesos (void);
int killProcess (nproc_t npr);
void listarProcesos (void);
void reboot (void);
void finProgDOS (void);
```

```
date_t getDate (void);
void pause (void);  // implementada como llamada al sistema

#define PROCESOS_H
#endif
```

El tipo de datos `descrProc_t` corresponde al descriptor de proceso. Los campos que incluye son: el estado de ejecución (*`estado`*), la razón por la que el proceso está bloqueado, si fuera el caso (`esperaPor`), la cola donde se pone el proceso cuando está bloqueado, si fuera el caso (*pColaBlk*), el identificador del proceso (`pid`) y el de su padre (`pPid`), el campo `sig` (para enlazar descriptores y formar listas), un puntero a la pila donde se guardan los registros generales para cambios de contexto (`sp`), la dirección y tamaño de la ubicación en memoria del proceso (`dirMem` *y* `tamMem`), puntero a la ventana (incluye teclado) de trabajo del proceso (*pWin*). El cluster y drive de trabajo del proceso (`drvWork` *y* `clusWork`), la tabla de descriptores de ficheros abiertos por el proceso (`tdf[]`), y por último a título informativo, el nombre del fichero ejecutable.

El tipo `cola_t` se utiliza para representar colas de procesos y en concreto la cola de preparados. Las declaraciones `extern` se encuentran declaradas nuevamente en el fichero "*procesos.c*". La directiva "`extern`" se utiliza para informa al compilador de que no reserve en ese momento el espacio de la variable, únicamente tome nota de su tipo.

En el sistema 'SO' cada proceso tiene su ventana, que es algo así como su pantalla virtual (aunque es posible que varios procesos usen la misma ventana). Por otro lado, como sólo hay un teclado físico es necesario repartirlo entre todos los procesos recurriendo a la multiplexación en el tiempo. Con ese fin se implementa el concepto proceso focal, de manera que los caracteres procedentes del teclado se asignan en cada momento al proceso focal, que es el que se considera que tiene su ventana con el foco del teclado. Como ya vimos, es posible cambiar la ventana focal con la tecla TAB.

Las funciones internas más relevantes de gestión de procesos implementadas son:

```
void activarProceso (nproc_t npr);
void activarPreparado (void);
void bloquearProceso (esperaPor_t por, fpCola cola);
nproc_t crearProceso (word_t part, word_t size, char *name);
int killProcess (nproc_t npr);
```

La función `activarProceso` reanuda la ejecución del proceso cuyo número se le pasa como parámetro, restaurando su estado (contexto) al que tenía cuando fue interrumpido o cedió el control de la cpu al S.O. En consecuencia, al llamar a esa función si no hay error, no habrá retorno.

La función `activarPreparado` reanuda la ejecución del primer proceso de la cola de preparados, invocando a la función `activarProceso`. Si la cola de preparados estuviese vacía, y puesto que no existe ningún *proceso ocioso*, el sistema activa (permite) las interrupciones (instrucción `STI`) y detiene el procesador (instrucción halt: `HTL`).

La función `bloquearProceso` pone al proceso que se le pasa como parámetro en estado de "bloqueado", especificando la razón de bloqueo que también se le pasa como parámetro. Si el puntero a cola que se le pasa no es NIL pondría en dicha cola al proceso bloqueado. Esta función

presupone que el estado (contexto) del proceso se encuentra ya salvado en la pila, de la cual el descriptor del proceso guarda su dirección.

La función `crearProceso` se invoca una vez que se ha conseguido memoria para el proceso y se ha cargado el código del mismo en ella. Básicamente esta función inicializa los campos del descriptor del proceso y pone éste en la cola de preparados, para cuando el planificador lo seleccione, éste pueda empezar su ejecución.

Finalmente, `killProcess` mata el proceso que se le pasa como parámetro. Si fuera el que está en ejecución, al final, pondría en ejecución el primero de la cola de preparados sin retornar al punto donde se hizo esta llamada. Al eliminar el proceso realiza las siguientes acciones: Libera su descriptor; devuelve la memoria al sistema, cierra su ventana terminal, si fuera el último proceso en usar dicha ventana; cierra sus ficheros abiertos; y por último, quita el proceso de la posible cola donde pudiera estar bloqueado (si fuera el caso).

8. ACTIVIDAD 4: Implementación de la interrupción Ctrl-C

En el PC la rutina de interrupción del teclado (`INT 09H` *Keyboard*) correspondiente a la BIOS, se encarga, entre otras cosas, de detectar por software si se está presionando simultáneamente las dos teclas Ctrl y Break, en cuyo caso simula una nueva interrupción que utiliza como vector de interrupción, el **1B**h (`INT 1BH` *Keyboard Break*). Este hecho está documentado en el *techelp*, no obstante, en el caso de la máquina virtual *DosBox*, esto no sucede, es decir, la combinación Ctrl-Break no produce dicha interrupción '**1B**', por esta razón y para conseguir un resultado similar, en el sistema 'SO' se ha reemplazado esta combinación por la combinación Ctrl-C, la cual se detecta en el módulo "*teclaint.c"* y se encarga de generar dicha interrupción, por lo que a todos los efectos el resultado será similar.

El objetivo de esta actividad es que el alumno programe la rutina de tratamiento de interrupción del vector '**1B**', para que en ella se "maten" inmediatamente todos los procesos vivos cuya ventana terminal sea la focal en el momento de la activación de la rutina (o sea, al pulsar Ctrl-C).

El programa "*techelp*" nos indica que la dirección del vector que utiliza la interrupción correspondiente al Ctrl-Break es la dirección de memoria 0000:006C (que sale de multiplicar el número del vector '**1B**' por 4, ya que cada vector de interrupción ocupa 4 bytes).

Vamos a centrarnos ahora en qué es lo que debe hacer la rutina de tratamiento de la interrupción '**1B**' para matar todos los procesos de la ventana focal. La respuesta no es complicada, ya que acabamos de ver en la actividad anterior que el sistema operativo 'SO' cuenta ya con la operación que necesitamos (ver fichero: "procesos.c") que es:

```
int killProcess (nproc_t npr);
```

La ventana focal se guarda en la variable "`pWinFocal`" declarada en el fichero "*windows.h*".

```
pWin_t pWinFocal;
```

En la tabla de procesos "`tblProc[]`" declarada en "*procesos.h*", tenemos los descriptores de todos los procesos. El campo "`pWin`" del descriptor nos dice cual es la ventana terminal del proceso, y con ello podemos saber si un proceso es focal o no, por ejemplo:

```
If (tblProc[npr].pWin==pWinFocal) …
```

Nos diría si el proceso "npr" tiene como ventana la focal, es decir si es focal o no. Por otro lado, el campo "estado" del descriptor nos indica el estado del proceso: LIBRE, PREPARADO, etc. Si el estado es LIBRE, sabemos que no es un proceso vivo, y por tanto, al recorrer la tabla de procesos, podríamos descartar estos descriptores, para evitar "matar" un proceso "no vivo". Sin embargo, aunque se intentara matar un proceso "no vivo" (estado == LIBRE), no pasaría nada grave, ya que la función "killProcess()" ignora estas peticiones.

Resumiendo: En la rutina de tratamiento de interrupción 'Ctrl-C' hay que recorrer la tabla de procesos, seleccionando los vivos y focales para matarlos, aunque hay que añadir un detalle importante: Si se llega a matar el proceso en ejecución antes de finalizar el recorrido de la tabla, y dado que la función "killProcess()" no retorna, podríamos no haber conseguido el objetivo, ya que podría haber más procesos focales en la tabla de procesos.

Pasemos ahora a otra cuestión. ¿Cómo incorporar la modificación/mejora dentro del código del sistema 'SO'? La respuesta la podemos encontrar fijándonos en la estructura de alguno de los ficheros de 'SO' donde se tratan las rutinas de tratamiento de interrupción, por ejemplo "teclaint.c", del que vamos a indicar a continuación su organización. Antes de eso conviene aclarar que aunque en alguna parte del código de 'SO' que veamos haya instrucciones en ensamblador, no se va a exigir al alumno que programe en ensamblador. Dicho esto a continuación se muestra el código básico empleado en dicha rutina:

```
/* ------------------------------------------------------------------- */
/*           Rutina de tratamiento de la interrupción xxx              */
/* ------------------------------------------------------------------- */

#include "tipos.h"
#include "rticomun.h"

/* Numero de vector utilizado para la interrupción xxx */
#define V_INT_xxx  0x??  // Sustituir las ?? Por su valor

PRIVATE void far * Old_VI_xxx; // para salvar antiguo vector de cBreak

/* ------------------------------------------------------------------
   las rutinas de tratamiento de interrupción (RTI) salvan automáti-
   camente los registros en la pila del proceso interrumpido y ade-
   más restablecen el registro DS al valor original del S.O., con ello
   los registros quedan preservados, pero como se va a establecer
   una nueva pila dentro del espacio del S.O., tenemos que salvar los
   punteros de pila (SS:SP) en el descriptor del proceso interrumpido.
   Cuando se desee volver al punto de interrupción se debe restaurar
   antes dicha pila y después efectuar el fin del tratamiento de la RTI
  (pop's e IRET). Para establecer la nueva pila dentro de 'SO' voy a
   considerar una nueva base con un valor inicial BASE_PILA. Se dejan
   0xFFFF-BASE_PILA KB para la consola cuya pila empieza en la FFFE.
   ------------------------------------------------------------------*/

void interrupt rti_xxx (void) {
  static int i;  /* ejemplo de variable local static permitido */
  setNewStack(); /* se establece una nueva pila dentro de 'SO'*/
  /* Ctrl-c debe matar todos los procesos de la ventana focal (de teclado),
     excepto la consola y el servidor */
  /* -----  Aquí vendría el código propiamente dicho de la RTI ------ */
     ...
     etc…
```

```
  /* ------------------- fin del código especifico --------------------*/
  restoreStack();  /* se restaura la pila antes de volver */
}// rti_xxx

void redirigirInt_xxx (void) {
  asm cli
  Old_VI_xxx = ((ptrTVI_t) 0) [V_INT_xxx];
  ((ptrTVI_t) 0) [V_INT_xxxx] = (void far *) rti_xxx ;
  asm sti
}

void restablecerInt_xxx (void) {
  asm cli
  ((ptrTVI_t) 0) [V_INT_xxx] = Old_VI_xxx;
  asm sti
}

/* ------------------------------------------------------------------- */
```

Vamos a explicar un poco este código: Comienza con la directivas #include del fichero "*tipos.h*" y "*rticomun.h*". En el primero se encuentra en tipo "`ptrTVI_t`" que se usa más abajo y en el segundo, se encuentran los prototipos de las funciones "`setNewStack()`" y "`restoreStack()`" empleadas para establecer la nueva pila dentro de 'SO' y restaurarla. Seguidamente encontramos la directiva `#define V_INT_xxxx`, que se utiliza simplemente por legibilidad y mantenibilidad. Las "xxx" deben ser sustituidas por un identificador más o menos nemónico de la interrupción con la que tratamos. Después se declara una variable del tipo puntero lejano a función, la cual servirá para guardar la dirección del vector de interrupción que hubiera antes de que 'SO' establezca el suyo propio. Esta variable se usa en las funciones "`redirigirInt_xxx()`" y "`restablecerInt_xxx()`" que se encargan, la primera, de salvar y establecer el nuevo vector de interrupción, y la segunda, de restaurar el vector original. Es conveniente aclarar la razón de la existencia de estas funciones. La función "`redirigirInt_xxx`" se requiere para establecer en el vector adecuado la dirección de la rutina de tratamiento que se desea incorporar a 'SO, sin embargo no está tan claro la necesidad de la funcion "`restablecerInt_xxx`", ya que en principio, se puede pensar que cuando el sistema 'SO' acabe, no hay necesidad de restaurar los vectores modificados durante el arranque. Si bien esto es cierto, no hay que olvidar que por varias razones, el sistema 'SO' también puede funcionar como invitado del sistema MS-DOS, y en ese caso sí es necesario restaurar los vectores a su estado original para no dañar el sistema anfitrión.

Antes de seguir es conveniente recordar un poco el mecanismo de las interrupciones del procesador. Cuando se produce una interrupción, el procesador pasa a ejecutar la instrucción que se encuentra en la dirección apuntada por el vector asociado a la interrupción que se ha producido, apilando previamente la dirección de retorno y los "*flags*" (banderas) de estado del procesador, e inhibiendo además las interrupciones. Una vez dentro de la "RTI", lo primero que se hace es salvar el contenido de los registros en la pila. Aunque el código que se encarga de ello no aparece explícitamente en el fuente "C", el compilador lo genera automáticamente cuando se especifica que la función es de tipo "`void interrupt`", como es el caso de la rutina de tratamiento de interrupción vista arriba: "`rti_xxx`()". Estas funciones, además de tener al comienzo una serie de instrucciones de apilamiento de todos los registros del procesador, también restauran el valor del registro de segmento de datos 'DS' al valor que debe tener para direccionar el área de datos del sistema 'SO'. Esto nos facilita el trabajo ya que la función preserva el estado de los registros y establece el valor del DS para poder trabajar con las variables globales, todo ello de manera implícita. A partir de ese momento habría que añadir el código necesario para lo que se requiera, pero normalmente la RTI lo que suele hacer justo a

continuación, es establecer una nueva pila dentro del espacio de 'SO', lo que le permitiría **no depender** del espacio disponible de pila del proceso interrumpido, y por simetría, también justo antes de finalizar, tendría que restaurar la pila dejándola como estaba antes de la interrupción. Este comportamiento, que es el habitual para cualquier RTI, **no debe realizarse en este caso**, ya que la RTI de Ctrl-C se invoca desde dentro de otra RTI, la de teclado, la cual a su vez ya ha realizado esta acción, y volver a realizarla corrompería la pila previamente establecida dentro de SO, por tanto, en el código listado arriba, el cual se puede tomar como ejemplo básico de una RTI, hay que eliminar para este caso concreto las llamadas a las funciones setNewStack() y restoreStack(). Por último un detalle importante a tener en cuenta a la hora de implementar una RTI: "No se deben declarar variables de ámbito local porque se ubican en pila. Si se requisiera alguna debe declararse global ó local con el atributo '**static**', lo cual hace que se ubiquen en el segmento de datos".

Una vez programada la rutina de tratamiento (rti_xxx), necesitamos programar también tanto la redirección del vector de interrupción para que apunte a dicha rutina, como su posterior restauración. Para ello usaremos las funciones ya vistas arriba: redirigirInt y restablecerInt, personalizándolas para nuestro caso, quedando únicamente pendiente la cuestión de dónde invocarlas. La respuesta es sencilla. Dentro del fichero "so.c" se podrá ver justo al principio, de la función "`main()`", cómo se invocan el resto de funciones de redireccionamiento de los vectores de interrupción, y también de igual modo, se puede ver cuando se trata el comando "EXIT" dentro del "*switch*" general, que también se invocan todas las funciones de restauración de los vectores de interrupción. En estos puntos es donde debemos insertar nuestros cambios o mejoras.

Una vez hechos estos cambios, si compiláramos el proyecto 'SO' sin más medidas, aparecerían errores indicando que no se encuentran los prototipos de las funciones *redirigirInt* y *restaurarInt*. El motivo es muy sencillo. Dichos prototipos se han de incluir en un fichero tipo "*.h*" característico de "C" para poder ser usado donde se requiera. Este fichero lógicamente debería llamarse "*break.h*" y debe ser incluido en el fichero fuente "*so.c*". Podemos tomar como ejemplo otros ficheros "*.h*", como por ejemplo "*timerint.h*".

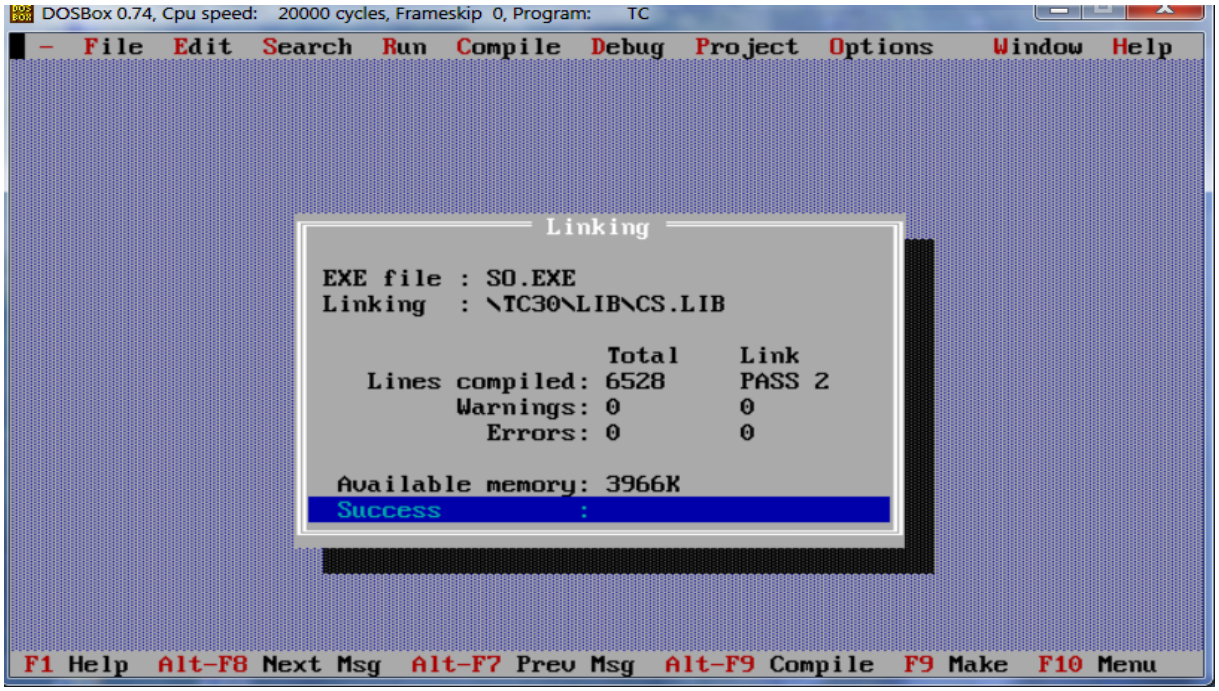
Por último sólo nos falta integrar estos ficheros en el proyecto "*so.prj*". Para ello basta con insertar el fichero *"break.c"* en el fichero del proyecto, abriendo la ventana de proyecto e insertando dicho fichero. Una vez hecho todo esto, ya sería posible compilar nuevamente el proyecto 'SO' mediante la tecla "F9" (ó Ctrl-F9 para compilar y ejecutar).

En la evaluación de este apartado se comprobará que, en el sistema 'SO' modificado por el alumno, la combinación de teclas 'Ctrl-C' mata todos los procesos cuya ventana es focal, tanto si están en ejecución como si están preparados o bloqueados.

Para poder probar que la mejora pedida al alumno funciona hay que conseguir que más de un proceso use una misma ventana. Para este propósito se ha incluido en el comando de ejecución de procesos, un parámetro adicional numérico que indica el 'pid' del proceso de cuya ventana queremos usar como anfitriona, por ejemplo, si tecleamos: ***A:>hola 2***, querríamos ejecutar el programa "hola" pero en la ventana del proceso cuyo 'pid' fuera 2. Al ejecutarse en la ventana de otro proceso, de algún modo, las pulsaciones de tecla se han de repartir entre los procesos que comparten la ventana. En este sistema se ha optado por hacer que *el último proceso que pida una tecla se ponga el primero en la cola de teclas de su ventana terminal*, lo cual no tiene porque ser la mejor política pero puede valer para realizar pruebas.

9. ACTIVIDAD 5: Compilación del sistema 'SO'

Aunque ya se ha visto algo de esto en apartados anteriores vamos a explicar cómo se compila el programa correspondiente al sistema operativo. Utilizaremos el perfil D-Fend "DBOX-SO" desde el cual tenemos acceso a la carpeta "virtualHD" que se monta como unidad de disco duro virtual "C". Los ficheros fuente que componen el programa del sistema están en el directorio '*c:\miso'*. En la fase de desarrollo lo normal es ejecutar el sistema partiendo de MS-DOS y desde el entorno de desarrollo de Turbo-C, todo ello sin necesidad de crear un disquete de arranque en ningún momento. El proceso es muy sencillo y ya hemos visto algo de ello anteriormente. Estando situados en el directorio que contiene los fuentes arrancamos Turbo-C, abrimos el proyecto "*so.prj*" (menú Project->Open Project), si no lo hubiera abierto ya automáticamente el entorno integrado de Turbo-C, y pulsamos "F9":



Una vez compilado podemos ejecutar el sistema 'SO' pulsando la combinación de teclas Ctrl-F9 o bien usar el ratón para abrir el *menú -> Run -> Run*. Esta opción, al ejecutar 'SO' sin salir del entorno TC, no deja mucha memoria disponible y si queremos probar 'SO' con más memoria es preferible salir de TC y ejecutar 'SO' desde la línea de comando:

```
C:\MISO>SO <┘
```

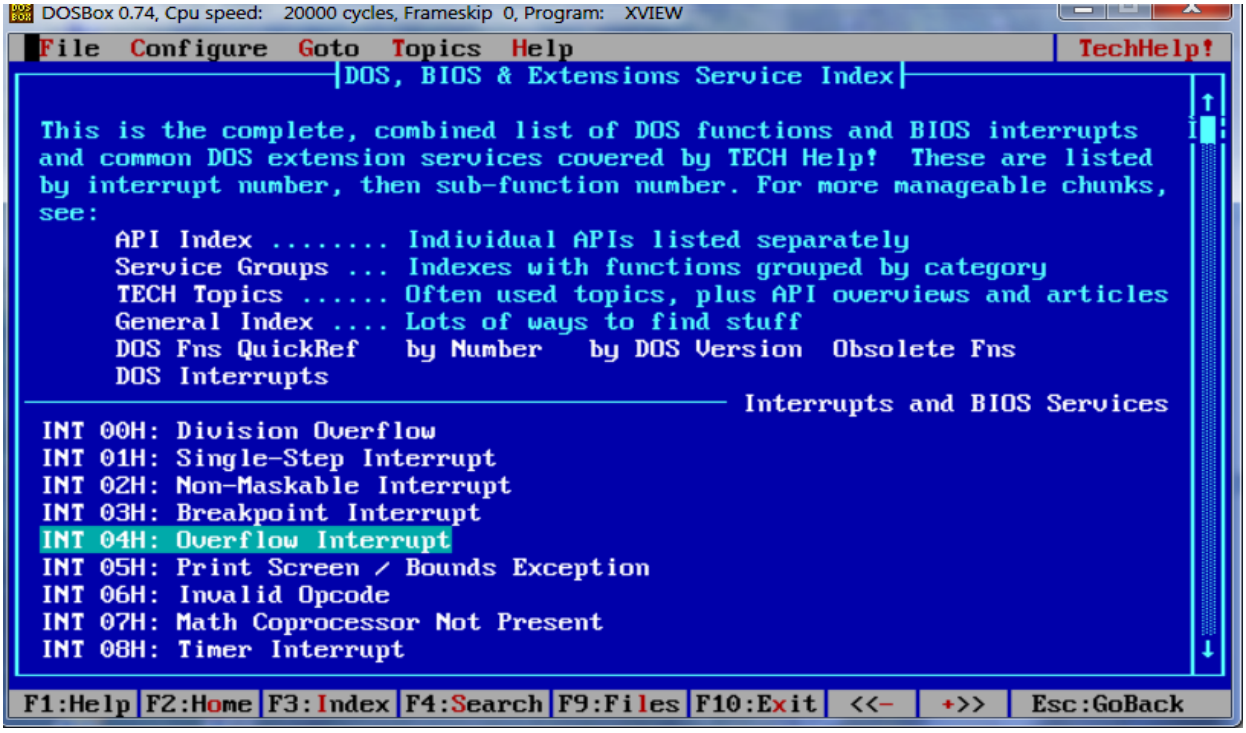
(Aunque ejecutemos 'SO' desde MSDOS sólo podremos acceder desde 'SO' a los ficheros que estén en las unidades montadas físicamente en Dos-Box como 0, 1, 2 y/o 3.)

Se recuerda que la velocidad de ejecución del programa puede variar mucho de ejecutarlo dependiendo de la selección que se haya hecho para la CPU *speed*.

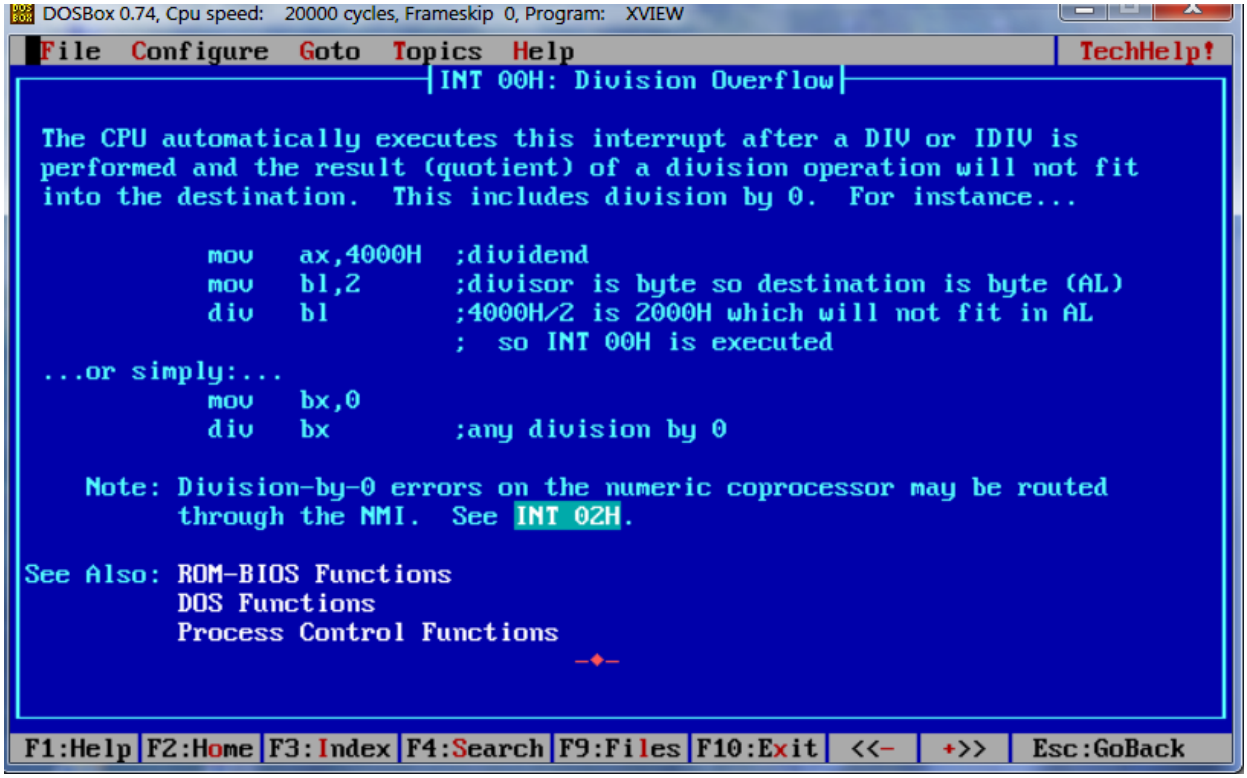
10. ACTIVIDAD 6: Implementación de excepciones (división por 0 y overflow)

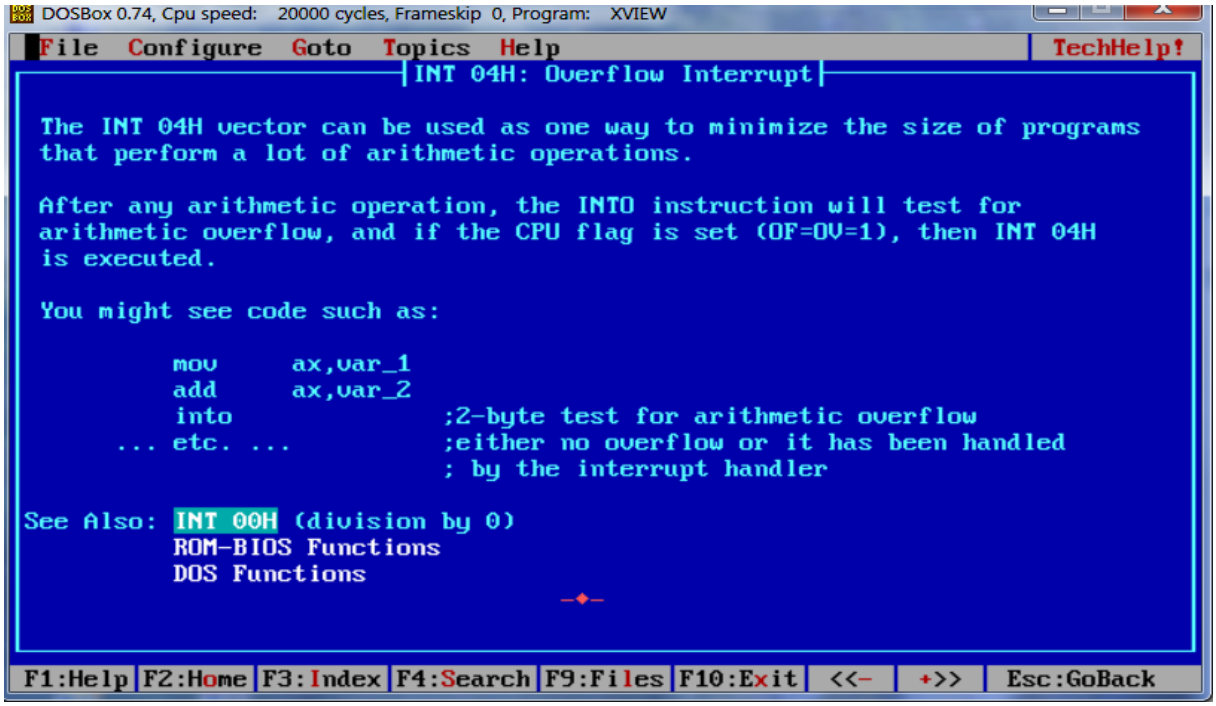
En este apartado el alumno programará las rutinas de tratamiento de las excepciones de división por 0 y *overflow*. Dado que la programación de esas rutinas es muy parecida a la de las rutinas de

tratamiento de interrupción, tenemos ya mucho camino recorrido gracias al apartado anterior. Los números de los vectores de interrupción que utilizan estas excepciones son los indicados en el *techelp*:



Más explícitamente son: el vector `00H` para el caso de la excepción de división por 0, y el vector `04H` para el caso del overflow. Podemos conocer cómo funcionan esas excepciones consultando los enlaces correspondientes:





Según las pantallas anteriores las rutinas de tratamiento de las dos excepciones toman el control automáticamente al ejecutar la instrucción `div` (o `idiv`, división con signo) y la instrucción `into` (interrupción si hay *overflow*) respectivamente bajo ciertas condiciones. En el primer caso la condición que dispara la excepción es que el divisor valga 0, o que el cociente no quepa en el registro destino de ese cociente (`AX` o `AL`). En el segundo caso la excepción se produce al ejecutar `into` estando activado el *flag* de *overflow*.

En cuanto al tratamiento de la excepción, deberá ser el siguiente:

- En la consola de 'SO' debe aparecer un mensaje que avise de que se ha producido una excepción y muestre cuál es su tipo (división por 0 u *overflow*) y cuál es el identificador del proceso (pid) responsable de esa excepción.

- La excepción debe provocar la muerte inmediata del proceso responsable de la excepción excepto si es la consola o el servidor, y

- El siguiente proceso a ejecutar debe ser el que determine la política de planificación existente actualmente en 'SO'.

Para escribir datos en la consola podemos hacer uso de algunas de las siguientes funciones auxiliares internas del sistema 'SO' (declaradas en '*windows.h*'):

```
PrintStr(str) ;                  /* str es una cadena de caracteres */
PrintDec(numero, ancho);         /* numero es un entero sin signo */
PrintHex(numero, ancho);         /* muestra el valor en hexadecimal */
```

En las funciones anteriores que escriben números debe indicarse el número de cifras (`ancho`) con el que va a escribirse el número. Si el número de cifras especificado es insuficiente para el número que se quiere escribir, la función utiliza como `ancho` el número de cifras necesario. Por ese motivo normalmente se indica como `ancho` una única cifra (es decir `ancho` 1). En el caso de

escritura hexadecimal, se rellena automáticamente con ceros a la izquierda hasta completar el número de cifras indicadas en `ancho`. Un detalle muy a tener en cuenta es que estas funciones internas muestran el texto en la ventana del proceso que estuviese en ejecución en el momento de la interrupción y si dicho proceso muere, el mensaje no podrá ser visto por la rapidez con la que desaparece la ventana del proceso. En estos casos es mejor que estos mensajes salgan directamente en la pantalla de fondo (no es ninguna ventana). Esto puede hacerse poniendo simplemente la variable global de 'SO' "`videoDirecto`" a TRUE inmediatamente antes de la función `Print…()`, y restaurándola a FALSE justo a continuación. De este modo el sistema 'SO' realiza la salida a dicha pantalla en vez de a la ventana del proceso en ejecución. Si queremos situar el mensaje en alguna posición concreta de la pantalla, utilizaremos las variables globales `curX` (0..79) y `curY` (0..24) antes de dicha salida para situar el cursor. (En general, un buen sitio para los mensajes sería: curX=14 y curY=24)

El esqueleto de las rutinas de excepción es similar al que se uso para Ctrl-C. Es decir habrá que crear los ficheros "*excepcio.h*" y "*excepcio.c*" y añadirlos al proyecto.

Con el fin de comprobar el correcto funcionamiento de las rutinas de tratamiento de excepción de división por cero y *overflow*, proponemos el siguiente programa de usuario: "*menuexc.c*", el cual se encuentra en el directorio *c:\miso\usrs_prg\menuexc*, y que a través de un sencillo menú, nos da la opción de producir cualquiera de las dos excepciones:

```
/* ------------------------------------------------------------------------*/
/*                                menuexc.c                                */
/* ------------------------------------------------------------------------ */
/*       programa de usuario con un menu para provocar excepciones          */
/* ------------------------------------------------------------------------ */

#include "..\ll_s_so\inic-usr.c" //cod.inic.prog.usuario y llam. basicas

void main () {
  char car=0;
  while (car != 'T') {
    printCar(FF);    /* borra el contenido de la ventana */
    printStr("SO: Elija su excepción:\n") ;
    printStr("'T' - Terminar programa\n") ;
    printStr("'D' - División por cero\n") ;
    printStr("'O' - Overflow\n") ;
    car = leerTecla() & 0xDF;   /* convierte a mayúsculas */
    switch (car) {
      case 'D': asm { MOV BX,0; DIV BX }; break;
      case 'O': asm { MOV AX,0x7fff; INC AX; INTO }; break;
      case 'T': break;
      default: printCar(BEL);
    }/* switch */
  }
}
```

11. ACTIVIDAD 7: Implementación de la llamada al sistema 'sleep'

En los sistemas tipo UNIX existe un servicio o llamada al sistema llamado "sleep" que permite bloquear a un proceso durante un cierto intervalo de tiempo especificado en segundos. En el sistema 'SO' se ha echado en falta este servicio cuando se programó el programa "`errante.c`", ya que allí hubo que ajustar la velocidad de los procesos a base de retardos implementados con bucles de espera:

```
void retardo (unsigned n) {   /* espera activa 'n' ticks */
  unsigned long r, far *ptime= (void far *) 0x046C;
  r = *ptime + n;
  while ( *ptime < r ); /* espera activa */
}
```

En un sistema multiprogramado como 'SO' la implementación de los retardos como bucles de espera activa significa un desperdicio de tiempo de CPU que perjudica a todos los procesos. La solución al problema consistirá en conseguir que los procesos esperen de forma pasiva como procesos bloqueados y no como procesos preparados o en ejecución, aprovechándose con ello el procesador.

El objetivo de este tutorial es la implementación de '***sleep***' como una nueva llamada al sistema que permita llevar a cabo retardos en los procesos de forma pasiva. La implementación se basará en las ideas presentadas en la solución del siguiente problema escogido del tema de Gestión de Procesos:

Procesos	02/09/99	Implementar: dormirse (segundos)	9899se3

Lo primero será establecer la interfaz de la llamada al sistema con los programas de usuario (ver los ficheros "*c:\miso\ll_s_so*.c*"). Vamos a convenir que para hacer uso de esta nueva llamada al sistema habrá que añadir en el fichero "*proc-ifz.c*" la siguiente función de biblioteca:

```
/* ----- Interfaz de llamada a sistema sleep ------- */
void sleep (unsigned decimas) {
  asm { MOV AH,9;
        MOV BX,decimas;
        INT VINT_SO }  // VINT_SO : vector llamada al sistema 'SO'
}
```

Lo anterior quiere decir que los programas de usuario podrán solicitar al sistema operativo que los mantenga bloqueados durante un cierto número de décimas de segundo, para lo cual ejecutarán la función anterior `sleep(decimas)`. En cuanto al sistema operativo (el cual toma el control tras la instrucción (*trap*) `INT VINT_SO,` en la función "`rti_SO`" del fichero "*llamadas.c*"), vemos que debe interpretar el código `09h` que le llega en el registro `AH,` como una petición de servicio "`sleep`", debiendo ceder el control a la función encargada de implementar dicho servicio y que vamos a llamar `so_sleep` (por mantener el mismo criterio de denominación de las funciones de servicio). Esta función tomará el valor del parámetro `'decimas'` entregado en el registro `BX,` donde lo dejó la función de interfaz '`sleep`', y se procederá a bloquear al proceso en ejecución y a activar al siguiente preparado.

Para conseguir este primer objetivo el sistema 'SO' nos ofrece en el fichero "*llamadas.c*" la macro ***RGP(N,R)***, la cual nos facilita el acceso al contenido de cualquier registro guardado en la pila de un determinado proceso (recordemos que cuando se realiza una llamada al sistema, la rutina de tratamiento de interrupción (RTI) guarda en la pila todos los registros). Con esta macro podemos acceder a su contenido fácilmente, por ejemplo: `RGP(nprEjec,B.X)` nos daría el valor guardado en pila del registro **BX** del proceso al que se está atendiendo. De este modo, acceder al parámetro "`decimas`" es sencillo, bastaría poner la macro tal y como se ha mostrado en el ejemplo anterior.

Pero hay que hacerse otra pregunta. ¿Cómo se bloquea el proceso y se activa el siguiente preparado?, afortunadamente la respuesta es sencilla. El sistema 'SO' dispone de la función de uso interno "`bloquearProceso(razon,&cola)`", la cual nos permite en una sola llamada hacer lo siguiente: a) Dejar bloqueado al proceso especificando la causa (si ha sido por espera de teclado, por dormir, etc.), b) Poner en una cola de procesos bloqueados a dicho proceso, y por último, c) Activar el siguiente proceso preparado disponible en la cola de preparados.

Relacionado con la función de arriba, el sistema 'SO' tiene definido un tipo en "*procesos.h*" denominado "`esperaPor_t`", el cual sirve para especificar la razón de bloqueo. Al crear el servicio 'sleep', hay que aumentar la lista de valores de este tipo, añadiendo el valor '`BLK_SLEEP`' para la razón "durmiendo". Hay que preguntarse también, qué cola habría que pasar como parámetro a la función "`bloquearProceso` ()". Es posible no pasar ninguna utilizando NIL, en cuyo caso los procesos dormidos no formaría parte de ninguna cola. No obstante, sería preferible añadir una nueva cola para este propósito, que podríamos declarar en los ficheros "*procesos.h*" y "*procesos.c*", tomando como modelo la forma en la que están declaradas las otras dos colas: "`preparados`" y "`aServir`".

Hasta aquí ya tenemos suficiente información para poder dejar un proceso bloqueado y en una cola de espera, pero tenemos que resolver más problemas. El proceso debe ser despertado transcurrido un lapso de tiempo especificado en el parámetro "`decimas`". Para resolverlo, en primer lugar debemos pensar dónde guardar el tiempo que falta para que un proceso despierte. El sitio más evidente es el descriptor de proceso. Podemos añadir un campo llamado "`lapso`" que mantendría los *ticks* de reloj que faltan para que el proceso despierte. El tipo de datos del descriptor se llama "`descrProc_t`", y está declarado en el fichero "procesos.h". Una vez resuelto esto, se nos plantea la necesidad de convertir las décimas de segundo en *ticks* de reloj, para lo cual necesitamos saber cuánto tiempo es un *tick* de reloj. Vamos o considerar que dicho tiempo es aproximadamente 55 ms, y que al hacer la conversión realizaremos un redondeo, tal que si la fracción de *tick* es menor de 0,5 la despreciamos y si es superior incrementamos en uno el resultado.

Una vez resuelta implementación de la función de servicio "`sleep()`", para completar el servicio, tenemos que acometer el problema de despertar al proceso en el tiempo establecido. Parece evidente lo más sencillo es modificar la rutina de tratamiento de interrupción del reloj, para que lleve la cuenta de los *ticks* que le quedan a un proceso para ser despertado. Esto no es complicado, bastaría con decrementar el campo lapso del proceso y al llegar a 0, despertar al proceso. ¿Cómo despertamos a un proceso dormido?; respuesta: lo quitamos de la cola de dormidos y lo ponemos en la cola de preparados. En el fichero "*procesos.h*" se encuentran disponibles las siguientes funciones relacionadas con colas:

```
void encolar (fpCola cola, nproc_t npr);
void colar (fpCola cola, nproc_t npr);
nproc_t desencolar (fpCola cola);
void quitarDeCola (fpCola cola, nproc_t npr, nproc_t nprAnt);
nproc_t buscarEnCola (fpCola cola, nproc_t npr, bool *pErr);
```

La función `encolar()`, añade un proceso a una cola por el final y la función `colar()` por el principio. La función `desencolar()` quita y devuelve el primer proceso de la cola. La función `quitarDeCola()` quita el proceso especificado en `npr` de cualquier posición en la que se encuentre en la cola. Es preciso pasarle también en `nprAnt` cual es el proceso anterior. Se usa normalmente durante el recorrido de una cola. Por último, la función `buscarEnCola()`, busca

el proceso `npr` en la cola y devuelve el proceso anterior en la misma. Si no estuviera en la cola devuelve `FALSE` en `*pErr`.

Aunque se han expuesto todas estas funciones de apoyo a la gestión de colas, normalmente no se requiere usar todas ellas para la modificación de la rutina de tratamiento de interrupción de reloj requerida por el servicio "*sleep*". Finalmente quisiera resaltar un error que se comente a menudo y que consiste en que al eliminar un elemento de una cola durante el recorrido de la misma se intente continuar el bucle del recorrido utilizando el campo siguiente del proceso recién eliminando (mediante quitarDeCola), lo cual sería un error porque al quitarlo de la cola, dicho campo se pone a NIL_PROC. (En otras ocasiones, si se elimina el proceso manualmente pero se inserta en la cola de preparados, el campo 'sig' no sería NIL_PROC pero tampoco apuntaría al siguiente de dormidos).

La incorporación de la nueva llamada al sistema *sleep* debe ser consistente con las llamadas al sistema anteriormente existentes. Por ejemplo, podría ser necesario modificar la rutina de tratamiento de la interrupción del Ctrl-C para que funcione consistentemente con los procesos bloqueados a causa de `sleep`, ya que la destrucción de un proceso dormido debe hacerse eliminándolo previamente de la cola de procesos `dormidos`. También habrá que comprobar la función `killProcess`.

La información dada hasta ahora debe ser suficiente como punto de partida para el alumno. Se recomienda al alumno que tome como modelo la implementación de las demás llamadas al sistema que figuran en el fichero "*llamadas.c*". Ya que el alumno va a tener que modificar varias partes del código de 'SO', deberá identificar los fragmentos de código que añada encerrándolos entre comentarios del tipo:

```
/* inicio modificacion sleep */
...                                (Instrucciones añadidas por el alumno)
/* fin modificacion sleep */
```

Resulta muy útil el uso de '`grep`' para localizar las líneas de los ficheros del código de 'SO' que contienen una determinada palabra. Esta opción se encuentra integrada en el menú de TurboC en: menú –, GREP.

Para comprobar la corrección de la implementación de *sleep* se propone utilizar el siguiente programa de usuario (disponible en *c:\miso\usrs_prg\calibra*):

```
/* ------------------------------------------------------------------------*/
/*                                calibra.c                                */
/* ------------------------------------------------------------------------ */
/*     programa de usuario para calibrar la llamada al sistema sleep      */
/* ------------------------------------------------------------------------ */

#include "..\ll_s_so\inic-usr.c" //cod.inic.prog.usuario y llam. basicas
#include "..\ll_s_so\proc-ifz.c" //llamadas al sistema de procesos

int leerDec (void) {      /* leee un entero sin signo */
  unsigned int acum = 0;
  char car = '0';
  /*-------------------*/
  while ((car >='0') && (car<= '9')) {
    if (acum>(0xFFFF-(car-'0'))/10) return -1; /* num. demasiado grande */
    acum = 10 * acum + (car - '0');
    car = leerTecla();
```

```
    printCar(car);
  }//while
  if (car != CR) return -1;                    /* cifra erronea */
  return acum;
}//leerDec

void main (void) {
  int decimas=-1;
  while (decimas) {
    printCar (FF);
    printStr (" Calibrador del sleep.\n\n Decimas a esperar: ");
    if ((decimas=leerDec()) != 0) {
      if (decimas==-1) {
        printStr ("\n\007 ! Número incorrecto !\n"); }
      else { /* decimas > 0 */
        printStr ("\n Dormido zz...!");
        duerme (decimas);
        printStr ("\n\007 Ya estoy despierto :-)");
      }//else-if
      printStr ("\n Pulsa una tecla."); leerTecla();
    }//if
  }//while
}//main
```

También podrá utilizarse el programa "*erraslp.c*" (donde se han sustituido los retardos mediante bucles de espera por llamadas a `sleep`) para comprobar cómo cambia el funcionamiento respecto de "*errante.c*", el cual utiliza espera activa. (Si se teclea el comando "type leeme.txt" mientras se mueven, por ejemplo, cuatro errantes se observará la diferencia)

12. ACTIVIDAD 8: Implementación de los semáforos

El objetivo de esta actividad es dotar al sistema 'SO' de los objetos abstractos "semáforos". El sistema va a proporcionar a los procesos un cierto número de semáforos con el fin de que los procesos puedan sincronizarse. Las funciones a implementar son: Inicializar, bajar y subir semáforo. Limitaremos la cantidad de semáforos a un máximo de 10. Las funciones de interfaz, al igual que se hizo anteriormente con "*sleep*", se incluirán en el fichero "*proc-ifz.c*".
El procedimiento es similar y basta repasar la actividad anterior para repetirlo sin problemas, aunque hay que precisar cómo serán los prototipos de estas funciones y los valores de los registros a emplear en cada una de ellas. A continuación se muestra dicha información:

```
/* En todas estas funciones el parámetro 'sem' se pasa en el registro BX,
   y el código de operación en AH. En iniSemaforo CX pasa el valor */
int iniSemaforo(unsigned sem, unsigned valor); //AH=0x0B, BX=sem, CX=valor
int bajaSemaforo(unsigned sem);                //AH=0x0C, BX=sem
int subeSemaforo(unsigned sem);                //AH=0x0D, BX=sem
```

Para los detalles sobre la realización de estas funciones de interfaz, úsese como modelo la función de interfaz de '*sleep()*' vista con anterioridad.
Conviene aclarar que el uso de estos registros es arbitrario, pero hay que respetarlo si deseamos utilizar los programas de usuario ya compilados que se entregan con la práctica, ya que estos programas utilizan de este modo los registros. Evidentemente si los recompilamos utilizando otra convención de registros no habría ningún problema.

A continuación vamos a comentar algunos aspectos de la implementación dentro 'SO'. En primer lugar hay que decir que el funcionamiento de los semáforos debe ser coherente con lo visto

en la parte de "Teoría" correspondiente al apartado 2.3 de comunicación entre procesos. Por otro lado, y ya referente a los cambios a realizar en el código de 'SO', ya sabemos que por lo general, cada nueva llamada al sistema debe tener su propia función de servicio, la cual ha de añadirse a las ya existentes en el fichero `"llamadas.c"`, del mismo modo en el que se hizo con la llamada "*sleep*" y que por seguir el criterio de nombres ya existente, deberían llamarse: `so_iniSemaforo`, `so_bajaSemaforo` y `so_subeSemaforo`. Recuerdese además el uso de la macro ***RGP(N,R)*** vista en '*sleep'*, para acceder a los registros pasados en la pila cuando se efectuó la llamada.

Parece natural el declarar una tabla en 'SO' para mantener el estado o situación de los semáforos. Lógicamente el número de elementos de la tabla será el máximo número de semáforos soportados; en nuestro caso 10. El número de semáforo que se pasa como parámetro en estas llamadas al sistema será precisamente el índice de esta tabla, por tanto un número de semáforo podrá valer entre 0 y 9. ¿Cómo deben ser los elementos de esta tabla? Como respuesta podemos decir que, en principio, sería suficiente con que tengan un campo numérico para guardar el valor del semáforo y otro del tipo "cola" (`cola_t`) para registrar los procesos bloqueados en el semáforo.

No hay que olvidar inicializar la tabla de semáforos. Bien en algún punto del arranque del sistema 'SO' ó bien en la declaración estática de la misma.

Como especificación del funcionamiento de los semáforos se nos pide lo siguiente: Cuando la operación '`subeSemaforo`' deba desbloquear un proceso y haya varios procesos en la cola del semáforo, se desbloqueará al primer proceso de la cola, es decir al que lleva más tiempo metido en ella (política FIFO). Se recuerda que todas las colas del sistema 'SO' enlazan los descriptores haciendo uso del campo `sig` del descriptor de proceso (que está definido en "*procesos.h*" y que ya existen en 'SO' funciones que facilitan el trabajo con colas).

Debido a que se explica en la parte de teoría cómo debe ser la implementación de los semáforos, y a que ya se han descrito las operaciones básicas de manejo de procesos dentro del sistema 'SO', es el momento de que el alumno ponga en práctica todo lo que sabe.

Para probar la corrección de la implementación de los semáforos se proporciona en el directorio "c*:\miso\usrs_prg\menusem*" el programa de usuario "*menusem.c*" que muestra un menú para realizar cada una de las operaciones propias de los semáforos. Con él podemos crear varios procesos que ejecuten el programa, y conmutar entre ellos con la tecla TAB para que unos procesos se bloqueen con la operación "***bajaSemaforo***", mientras que otros procesos se encarguen de desbloquearlos mediante operaciones "***subeSemaforo***".

Otro programa de usuario que nos permite probar el funcionamiento de los semáforos es el programa "c*:\miso\usrs_prg\cross\cross.c*". Modificación de "*errante.c*", en el que se cumple la siguiente condición:

> ***La cuarta y octava vueltas al circuito corresponden a un tramo de circuito muy estrecho y sólo se permite realizar la vuelta a un único proceso a la vez.***

APÉNDICE III. Práctica: Paso de mensajes (buzones).

ÍNDICE

1. OBJETIVOS

Los objetivos de esta práctica son:

- Que el alumno profundice sobre el mecanismo de las llamadas al sistema con parámetros que representan direcciones de memoria.
- Que el alumno sea capaz de implementar en un sistema operativo, un tipo de paso de mensajes sencillo a través de buzones (*enviaMsjBuzon, recibeMsjBuzon*), con una capacidad fija que viene establecida por una constante interna del sistema operativo.

2. INTRODUCCIÓN

En esta práctica se pretende profundizar en la implementación de uno de los tipos de llamadas al sistema más complejos, como son las correspondientes al paso de mensajes. El tratar este tipo de llamadas al sistema tiene la ventaja de permitirnos ilustrar la comunicación de datos desde el espacio de direcciones de un proceso de usuario hasta el espacio de direcciones de otro usuario distinto pasando por el espacio de direcciones propio del sistema operativo común a ambos, algo que resulta muy interesante para un primer curso de sistemas operativos.

El modelo de comunicación que se va a implementarse será el de la comunicación asíncrona, indirecta, y simétrica, con buzones de capacidad limitada y mensajes de tamaño fijo de 16 bytes transmitidos mediante copia (ver paso de mensajes en el apartado 2.3 del tema de procesos).

3. TRABAJO A REALIZAR

Vamos a presuponer que el alumno conoce ya el funcionamiento del sistema 'SO', tanto a nivel de usuario, introduciendo comandos a través de la consola o haciendo llamadas al sistema desde un programa, como a nivel de la estructura del programa 'SO' en correspondencia con la de un sistema monolítico. En relación con dicha estructura, se supone el conocimiento de la implementación concreta de los procesos (descriptores, tabla de procesos, cola de preparados, el planificador y el despachador) y de la gestión de memoria.

En pocas palabras el trabajo a realizar consiste en la implementación de los manejadores de las llamadas al sistema cuyas rutinas de interfaz vamos a denominar *enviaMsjBuzon* y *recibeMsjBuzon*, las cuales permitirán a los procesos comunicarse a través de un esquema de paso de mensajes basado en buzones (comunicación asínicrona, indirecta y simétrica). Los buzones son de capacidad limitada, debiendo admitirse también el caso particular de que sean de capacidad nula (comunicación síncrona, *rendez-vous*).

4. ESPECIFICACIÓN DEL PASO DE MENSAJES

Nuestro objetivo es añadir un mecanismo de paso de mensajes al sistema 'SO', ampliando su conjunto de llamadas al sistema con las siguientes llamadas, de las cuales indicamos los prototipos de las correspondientes funciones de interfaz:

```
/* ------------------------------------------------------------------- */
/*                              buzones.h                              */
/* ------------------------------------------------------------------- */
/* Especificación de las funciones de interfaz  para el paso de mensajes */
/* ------------------------------------------------------------------- */

#define CAPACIDAD 2  // Capacidad de los buzones, puede ser 0, 1, 2, ...
#define MBOX_MAX 10  // máximo 10 buzones

typedef unsigned char mboxDesc_t;  // descriptor de buzón

int enviaMsjBuzon(mboxDesc_t mbox, void far *msj);  // cod. op. AH = 0Ch
int recibeMsjBuzon(mboxDesc_t mbox,void far *msj);  // cód. op. AH = 0Dh
```

La idea es que el sistema operativo va a ofrecer a los programas de usuario hasta 10 buzones identificados cada uno de ellos por un número del 0 al 9 que denominaremos '*descriptor del buzón*'. Los mensajes se envían/reciben a/de los buzones por lo que la comunicación es indirecta. La comunicación es asíncrona porque el uso del buzón permite que el proceso receptor no tenga que estar listo para recibir, ya que la entrega se deposita en el buzón. Sólo en el caso de que el buzón sea de capacidad nula, el mensaje se entrega directamente al proceso, en este caso la comunicación es síncrona, ya que se requiere que el proceso esté listo para recibir. Un mensaje queda identificado por su dirección de comienzo (que podemos ver como un puntero lejano), entendiéndose que el mensaje está físicamente compuesto por el valor de los 16 bytes consecutivos que se encuentran a partir de dicha dirección de memoria. Por tanto los mensajes son de tamaño fijo igual a 16 bytes. A continuación se muestran varios ejemplos del envío de diferentes tipos de mensajes:

```
mboxDesc_t buzon = 5;
char str[16] = "123456789012345";    // los strings terminan con '\0'
int tabla[8] = { 60, 100, 200, 400, 800, 1500, 3000, 5000 };
struct {
  char car1, car2 ;
  int i;
  long long1, long2, long3 ;
} registro = { 'A', 'B', -77, 100000, 1000000, 10000000 } ;
...

enviaMsjBuzon (buzon, str);       // el mensaje es una cadena de caracteres
enviaMsjBuzon (5, tabla);         // el mensaje es una tabla de 8 enteros
enviaMsjBuzon (buzon, &registro); // el mensaje es un registro
```

Para recibir un mensaje, el proceso receptor debe indicar el buzón del cual quiere recibir el mensaje, así como la dirección de destino de los 16 bytes correspondientes al mensaje. La interpretación de la estructura del mensaje es competencia del proceso receptor, y lo normal es que el proceso emisor y el receptor se hayan puesto de acuerdo de alguna manera en el formato de los mensajes. Lo lógico es que los mensajes se depositen en estructuras de datos análogas a las utilizadas por el emisor. Por ejemplo el proceso receptor de los tres mensajes enviados anteriormente podría proceder de la siguiente manera:

```
mboxDesc_t mailbox = 5;
char cadena[16];
int bufer[8];
struct {
  char c1, c2;
  int e;
```

```
  long l1, l2, l3;
} reg;
...

recibeMsjBuzon (5, cadena);
recibeMsjBuzon (mailbox, bufer);
recibeMsjBuzon (5, &reg);
```

Tras la ejecución de esas instrucciones por parte de los procesos emisor y receptor se cumplirían las siguientes igualdades:

```
cadena[0]=='1', cadena[1]=='2', ..., cadena[14]=='5', cadena[15]=='\0'
bufer[0]==60, bufer[1]==100, ..., bufer[7]==5000
reg.c1=='A', reg.e==-77, ..., reg.l3==10000000
```

Además de esas condiciones, el paso de mensajes impone la sincronización de los procesos que se comunican, y en esta sincronización interviene decisivamente la capacidad de los buzones (constante `CAPACIDAD`).

Una consecuencia de que *enviaMsjBuzon* y *recibeMsjBuzon* sean llamadas al sistema es que el sistema operativo asegura que se ejecuten como acciones atómicas e indivisibles. Dicho de otro modo, no puede haber dos procesos que estén ejecutando simultáneamente dichas funciones. Si dos procesos intentan ejecutar a la vez estas funciones, el sistema asegura que primero un proceso ejecutará su función y sólo después, el otro proceso podrá ejecutar la suya. Por tanto, el emisor ejecuta primero *enviaMsjBuzon* y el receptor ejecuta después *recibeMsjBuzon*, o sucede al contrario.

Cuando un proceso llama a *recibeMsjBuzon*, si hay algún mensaje en el buzón, el proceso recibe el mensaje (transfiriéndolo desde del buzón a la dirección de memoria especificada en la llamada) y sale de la llamada al sistema continuando su ejecución. Si por el contrario, no hay mensajes en el buzón, el proceso se bloquea y se queda al final de la cola de procesos que esperan recibir un mensaje de dicho buzón. El proceso debe permanecer bloqueado hasta que llegue un mensaje al buzón y le toque el turno de la cola de dicho buzón, es decir, que sea el primero de dicha cola. Cuando esto suceda, el mensaje debe ser transferido, al igual que en el caso anterior, desde el buzón a la dirección de memoria del proceso receptor y éste debe reanudar su ejecución (pasar a preparado). De lo dicho hasta ahora es fácil deducir que en la implementación de buzones debe haber un buffer para los mensajes (de tamaño máximo "`CAPACIDAD`") y una cola (tipo "`cola_t`") para los procesos en espera de mensaje.

Análogamente, cuando un proceso llama a *enviaMsjBuzon*, si hay espacio en el buffer del buzón, el mensaje se transfiere desde la dirección de memoria especificada en la llamada a dicho buffer, y el proceso sale de la llamada y continúa su ejecución. Si por el contrario, el buffer del buzón está lleno, el proceso se bloquea y se pone al final de la cola de procesos que esperan enviar un mensaje al buzón. El proceso debe permanecer bloqueado hasta que un proceso al llamar a *recibeMsjBuzon*, habilite espacio en el buffer del buzón para que el mensaje pendiente de entrega de este proceso pueda depositarse en el buzón y con ello reanudar su ejecución.

Como especificación del servicio de paso de mensajes hay que decir que los mensajes deben recibirse y enviarse por estricto orden cronológico. Es decir, si por ejemplo un proceso 'A' envía un mensaje a un buzón y posteriormente lo hacen los procesos 'B' , 'C' y 'D', cuando cualquier otro proceso intente recibir de dicho buzón, debe recibir el mensaje del proceso 'A', y si posteriormente este proceso o cualquier otro intente recibir más mensajes, recibirá respectivamente y

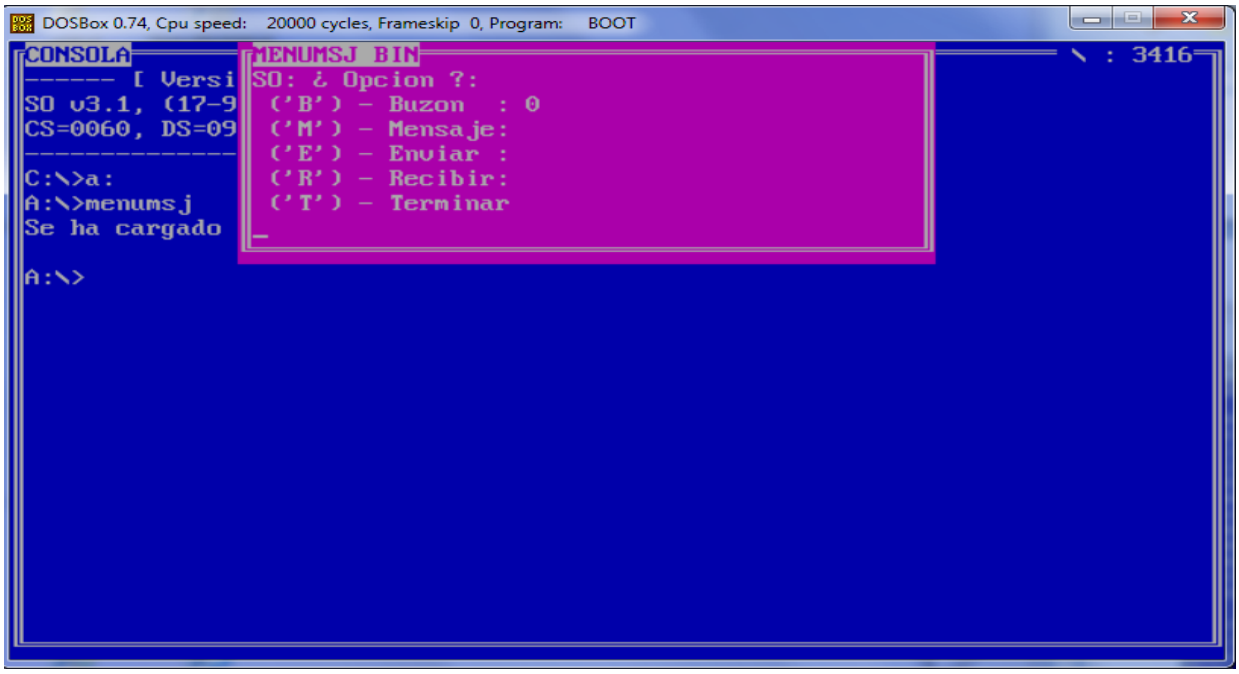
por este orden, los mensajes de los procesos 'B', 'C' y 'D'. Este concepto es análogo al caso recíproco de intentos de recepción de los procesos 'A', 'B', 'C' y 'D' y de envíos posteriores de cualquier otro proceso.

De la exposición anterior se deduce inmediatamente una estructura de datos válida para representar los buzones en el núcleo de 'SO', utilizando en cada buzón una tabla circular (campo `bufer`) para representar la cola de mensajes contenidos en el buzón. Nótese que la cola del buzón puede contener en un momento dado procesos que están esperando dejar un mensaje en el buzón y en otro momento dado, procesos que están esperando recibir un mensaje del buzón. Nunca puede darse el caso de que haya procesos esperando dejar un mensaje y en el mismo momento, que haya procesos esperando recibir un mensaje del mismo buzón, por ello no es necesario implementar dos colas en el buzón (una para procesos receptores y otra para procesos destinatarios).

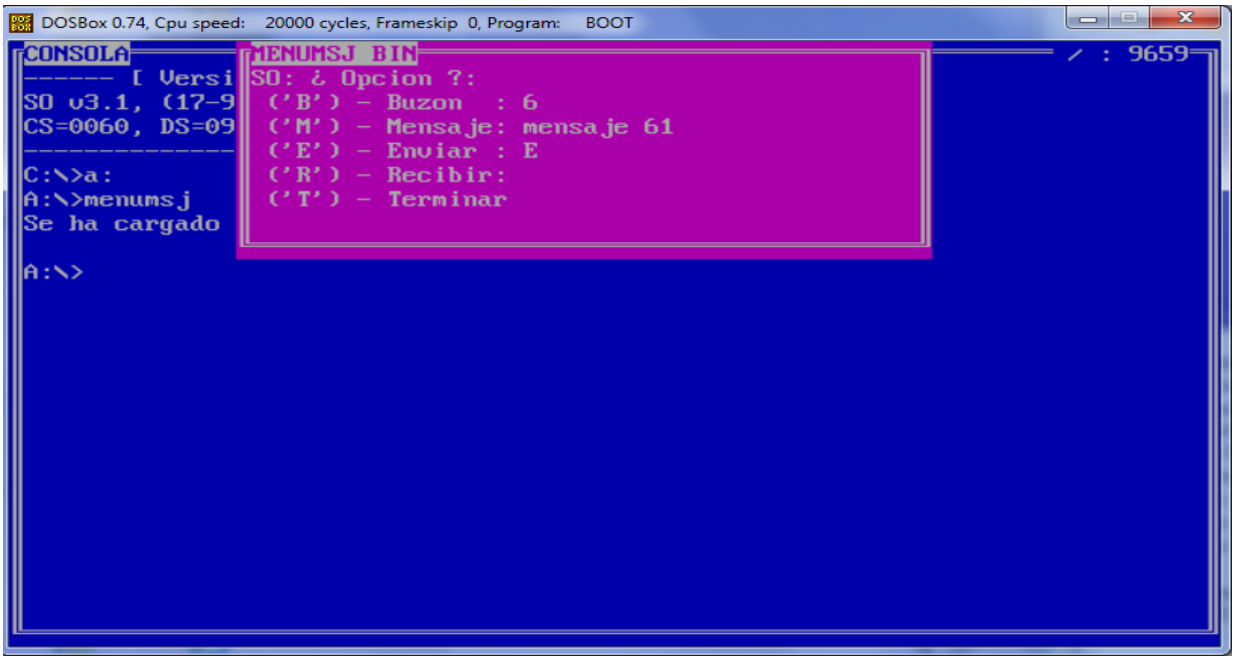
```
#define CAPACIDAD  2       // Capacidad de los buzones,
#define MBOX_MAX  10       // 10 buzones (mailbox)
#define LONG_MSJ  16       // longitud de los mensajes

struct {
  cola_t cola;
  unsigned char buffer[Capacidad][LONG_MSJ];
  unsigned in,out;   // buffer[in/out] primera entrada libre/ocupada
  unsigned numMensajes;                        // si numMensajes > 0
} buzon [MBOX_MAX];
```

5. EJERCITÁNDONOS CON EL PASO DE MENSAJES

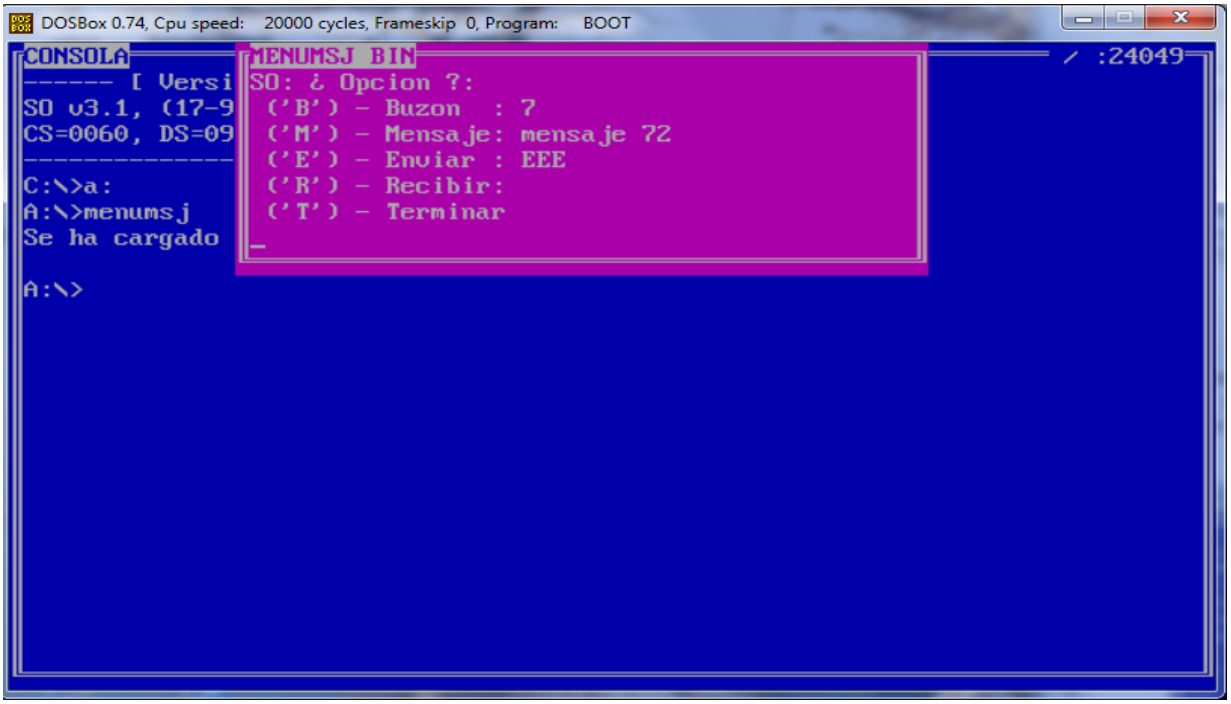
Para que el alumno capte de la forma más clara posible cómo debe funcionar el paso de mensajes lo mejor es que se ejercite con él dentro de 'SO'. El perfil *D-Fend* "***DBxBoot-SO***" ya conocido, permite arrancar desde disquete una versión de 'SO' que tiene implementadas todas las modificaciones solicitadas en las prácticas 3 y 4 (break, excepciones, sleep, semáforos y buzones). La capacidad de los buzones ha sido establecida en 'SO' mediante la declaración de la constante `CAPACIDAD` con el valor 2. Seguidamente arrancaremos desde dicho sistema el programa de usuario "*menumsj.bin*" situado en la unidad '**A:**', el cual proporciona un menú para que realicemos todas las operaciones relacionadas con el paso de mensajes:



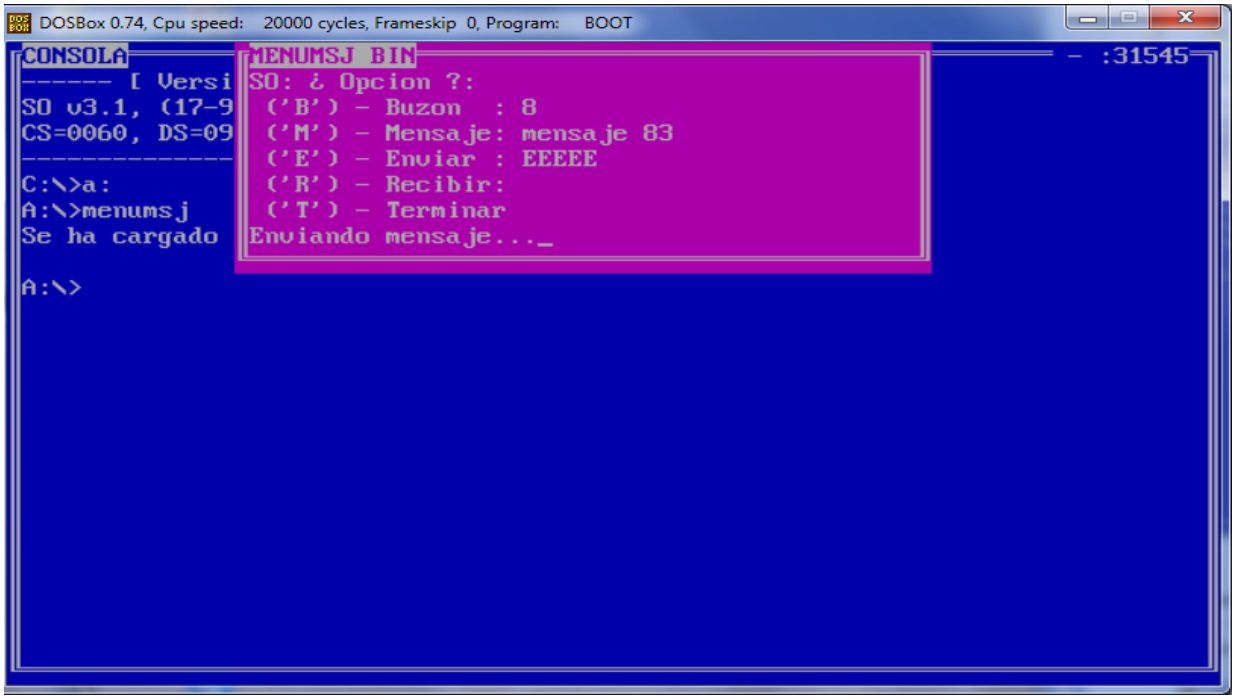
El buzón al que se envía o del que se recibe un mensaje figura tras los '**:**' de la opción **'B'**, la cual nos permite seleccionar cualquier otro buzón, mediante un número entre el 0 y el 9. El mensaje que se envía es la última cadena de caracteres (de longitud máxima 15) que se haya establecido con la opción **'M'**, el cual se muestra también tras los '**:**' en esta opción. Después de recibir un mensaje con la opción **'R'**, el mensaje recibido también se muestra en el mismo sitio, pero aparecerá una 'R' tras los '**:**' de esta opción. Del mismo modo, cuando se envía un mensaje, aparecerá una 'E' tras los ':' de la opción '**E**'. Vamos a empezar enviando al buzón 6 el mensaje `"mensaje 61"` (pulsar: B, 6, M, "mensaje 61" <┘, E):



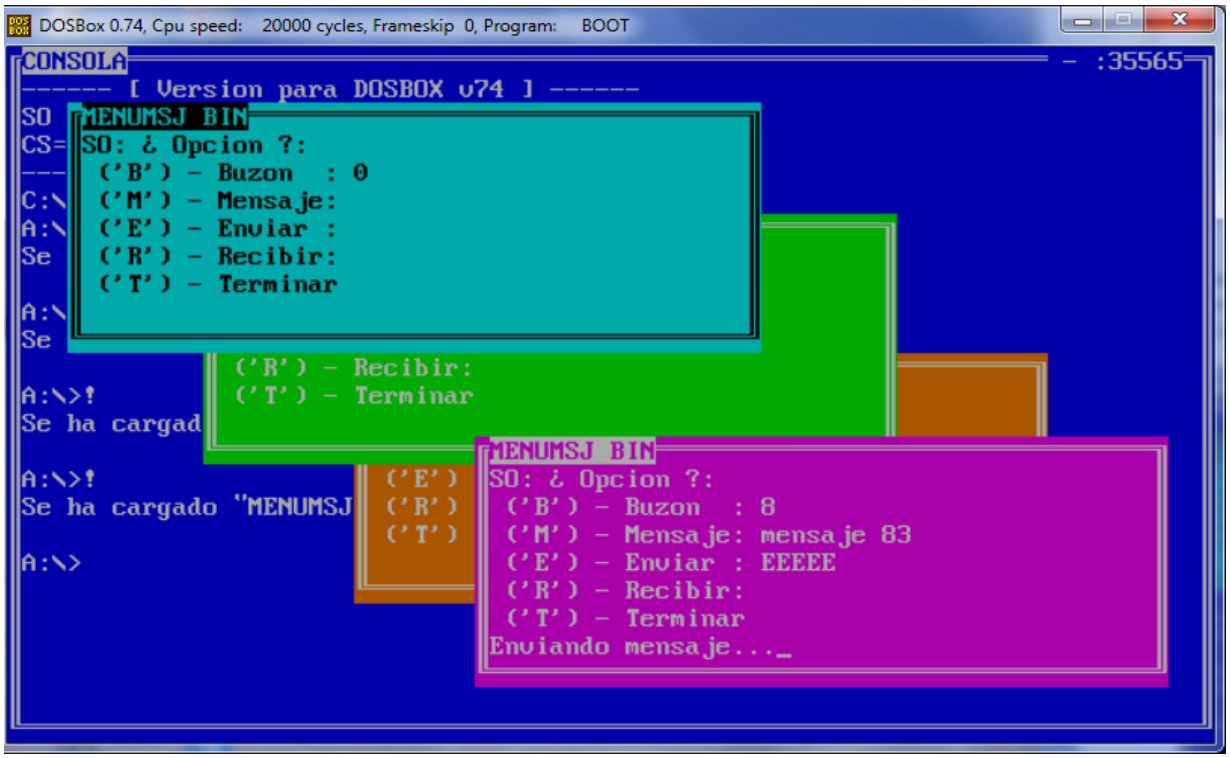
Vemos que el proceso 1 realiza el envío sin problemas, y el mensaje queda en el buzón 6.
Ahora enviamos al buzón 7 los mensajes `"mensaje 71"` y `"mensaje 72"`, operaciones que también se realizan sin problemas, quedando los dos mensajes en el buzón 7 (pulsar: B, 7, M, "mensaje 71" <┘, E, M, "mensaje 72" <┘, E):



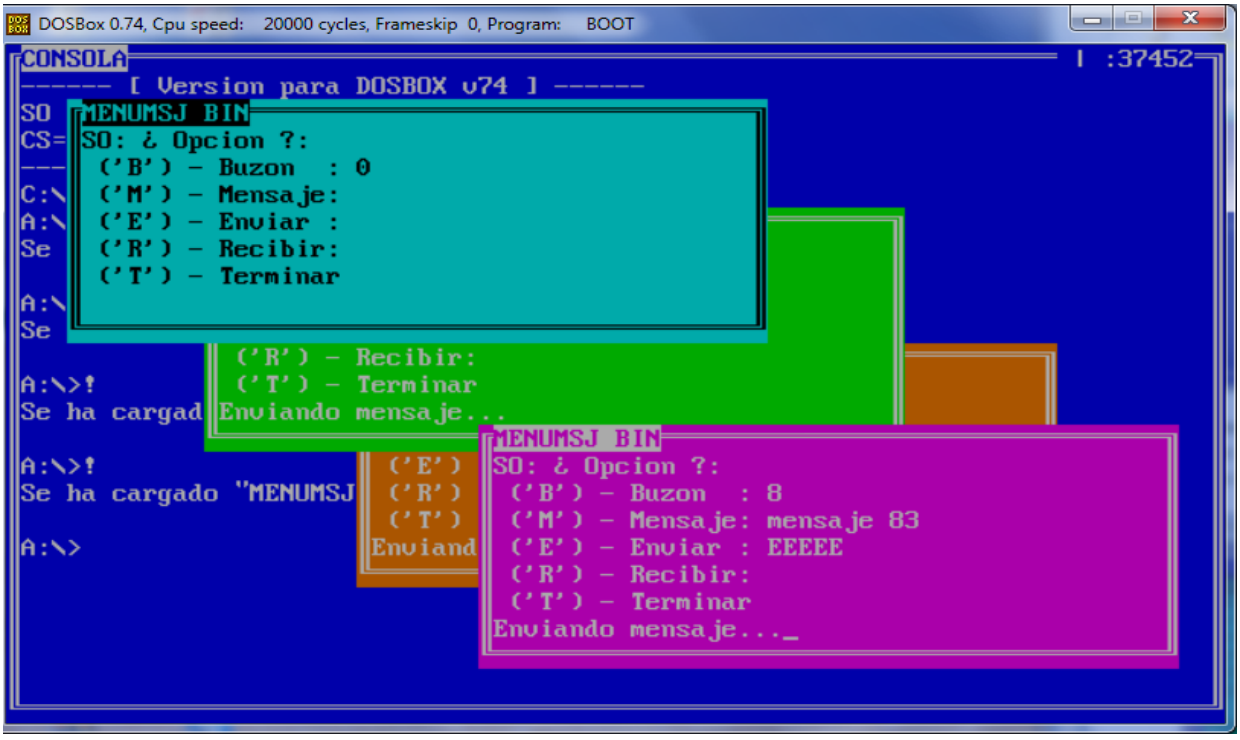
Seleccionamos ahora el buzón 8 y le enviamos los mensajes `"mensaje 81"`, `"mensaje 82"` y `"mensaje 83"`. Observamos que ahora el proceso queda bloqueado al intentar enviar el último mensaje ya que el buzón 8 está lleno a causa del envío de los dos primeros mensajes.



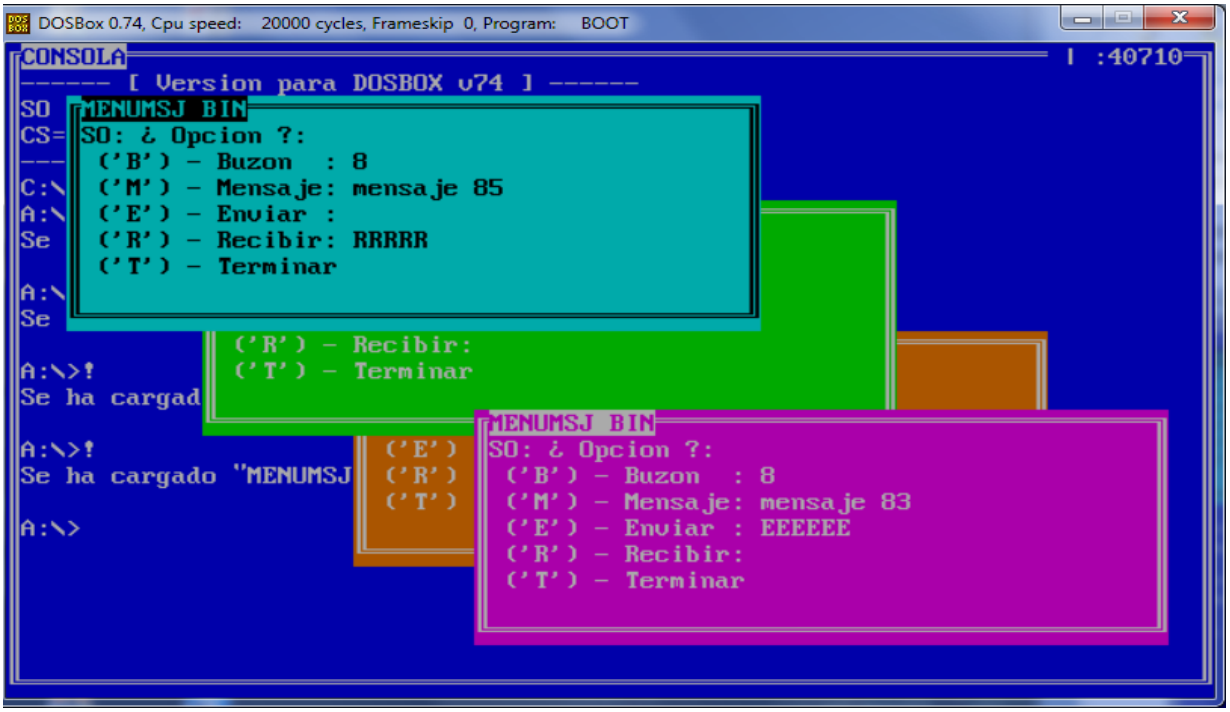
La única manera posible de desbloquear al proceso violeta (pid=2) es que otro proceso reciba un mensaje del buzón 8. Vamos a crear tres procesos más que ejecuten también el programa *"menumsj.bin"*.



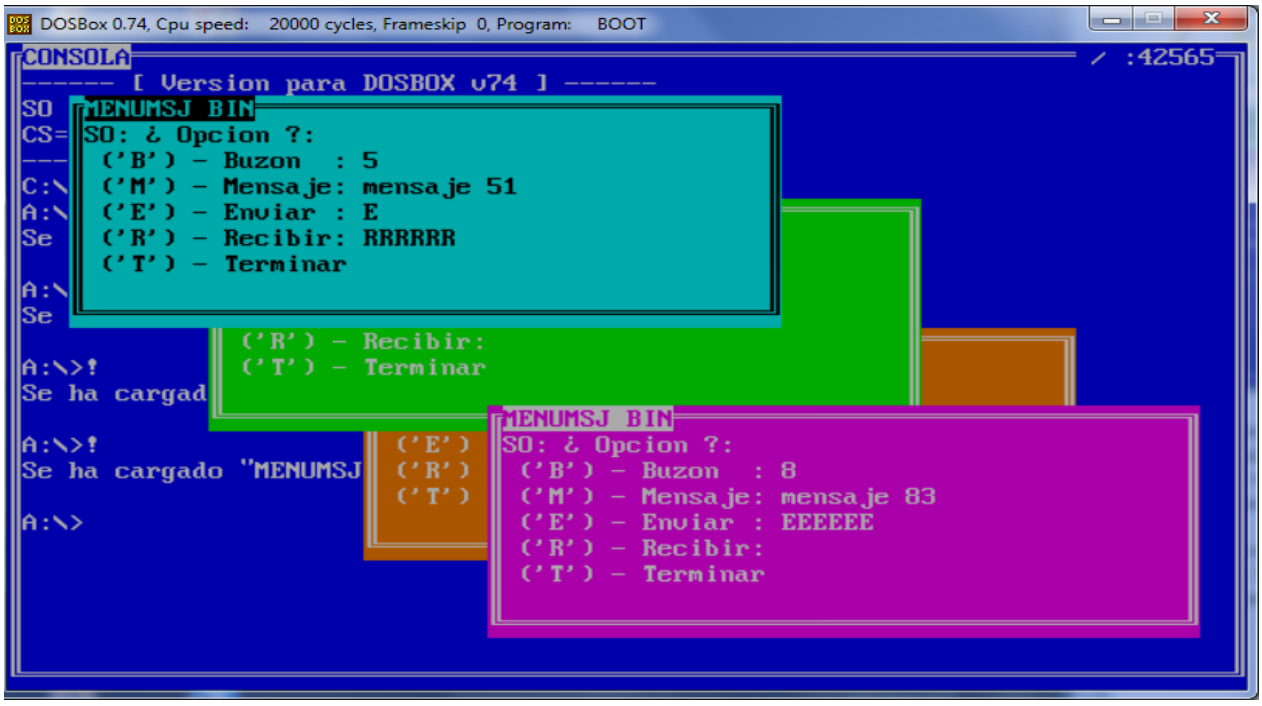
Ahora vamos a hacer que el proceso marrón (pid=3) envíe al buzón 8 el mensaje `"mensaje 84"`, y que el proceso verde (pid = 4) envíe al buzón 8 el mensaje `"mensaje 85"`. Evidentemente esos dos procesos quedarán también bloqueados.



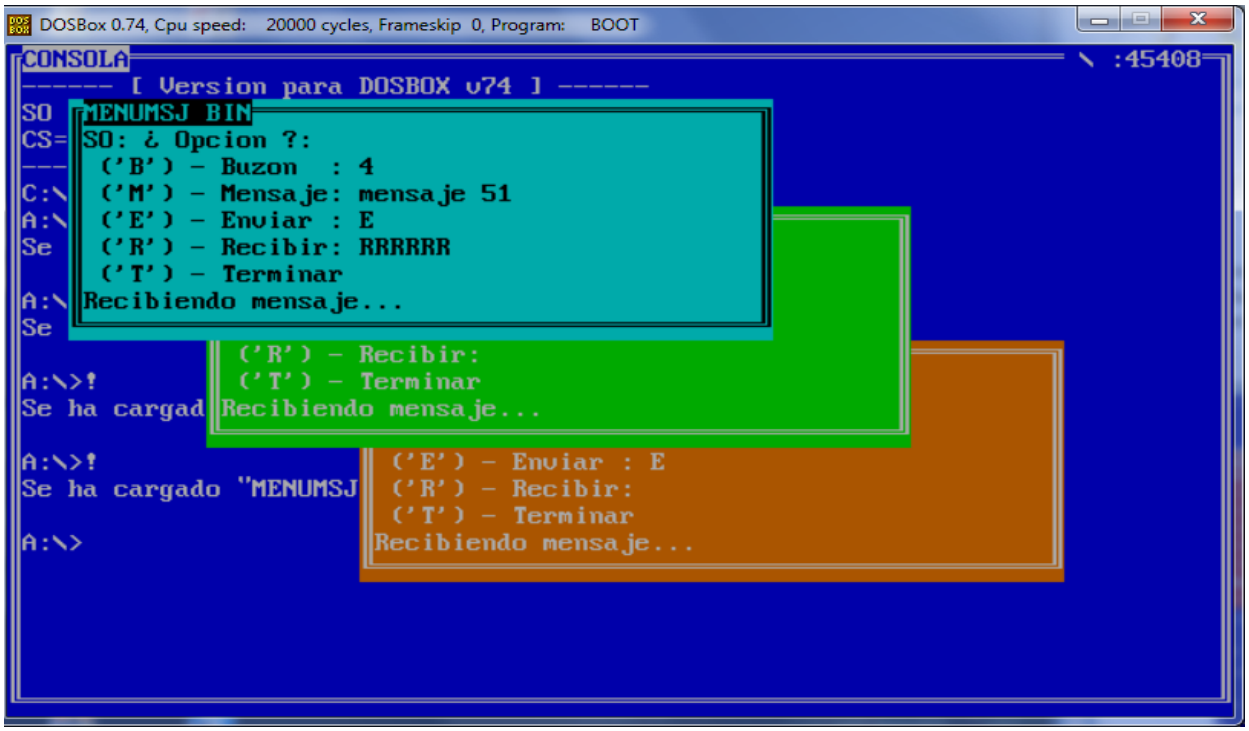
Ahora vamos a hacer que el proceso azul (pid=5) reciba del buzón 8 cinco mensajes. Observaremos que los mensajes se reciben exactamente en el orden en el que se han enviado (`"mensaje 81"`, `"mensaje 82"`, `"mensaje 83"`, `"mensaje 84"` y `"mensaje 85"`) y que en las tres primeras operaciones de recepción desbloquean sucesivamente a los procesos violeta, marrón y verde, siguiendo estrictamente la secuencia temporal en la que se bloquearon.



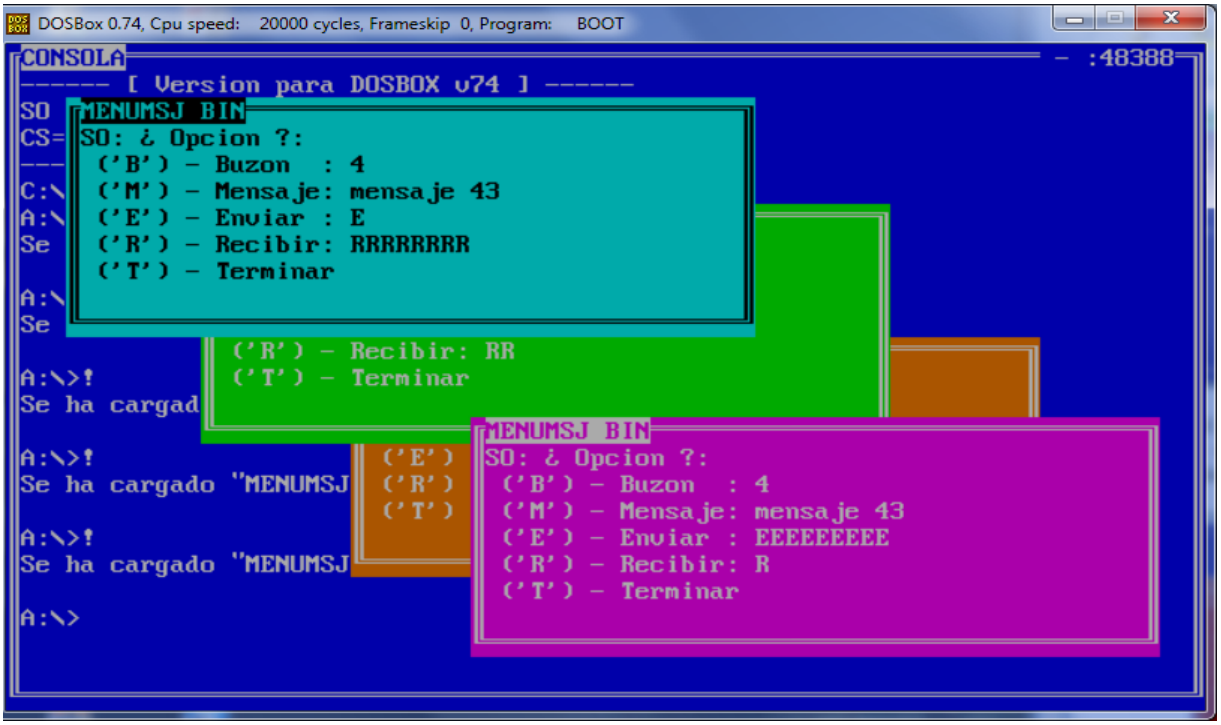
Ahora vamos a hacer que el proceso azul se envíe a través del buzón 5 el mensaje `"mensaje 51"` a sí mismo. Tras el envío debemos utilizar la opción `'m'` para introducir una cadena de caracteres vacía, borrando así el mensaje visualizado. Al recibir del buzón 5 debemos obtener el mismo mensaje que enviamos (`"mensaje 51"`).



Vamos a ver qué sucede ahora cuando varios procesos se quedan bloqueados intentando recibir de un buzón que está vacío. Para ello vamos a hacer que los procesos marrón, verde y azul intenten recibir (en ese orden) del buzón 4.



Si ahora el proceso violeta envía al buzón 4 los mensajes `"mensaje 41"`, `"mensaje 42"` y `"mensaje 43"` lo que debe suceder es que los procesos marrón, verde y azul se desbloquearán por ese orden al recibir los mensajes respectivos.



Lo visto hasta aquí debe ser suficiente para que el alumno se haya familiarizado suficientemente con las operaciones de envío y recepción de mensajes utilizando buzones de capacidad limitada, por lo que vamos a dejar el nivel del usuario, para pasar a ver a continuación algunos aspectos sobre el desarrollo de los buzones para el sistema 'SO'.

6. RUTINAS DE INTERFAZ DEL PASO DE MENSAJES

La implementación de las rutinas de interfaz que facilitan al usuario la realización de las llamadas al sistema correspondientes al paso de mensajes desde un programa en C son las siguientes:

```
/* ------------------------------------------------------------
       Funciones de llamadas al sistema para mensajes
   ------------------------------------------------------------ */
int enviaMsjBuzon (unsigned nBuz, void far *msj) {
  asm { MOV CX,nBuz
        LES BX,msj          // ES = SEGMENT msj, BX = OFFSET  msj
        MOV AH,0xE
        INT VINT_SO }  return _AX;
}//enviaMsjBuzon

int recibeMsjBuzon (unsigned nBuz, void far *msj) {
  asm { MOV CX,nBuz
        LES BX,msj          // ES = SEGMENT msj, BX = OFFSET  msj
        MOV AH,0xF
        INT VINT_SO }  return _AX;
}//recibeMsjBuzon
```

Estas funciones podrían definirse en un fichero independiente, pero por razones prácticas, es mejor incluirlas en el fichero de rutinas de interfaz ya existente "*c:\miso\ll_s_so\proc-ifz.c*".

Como puede apreciarse, en ambas funciones de interfaz el descriptor del buzón (`nBuz`) se pasa a través del registro **CX**, mientras que el segmento de la dirección del mensaje (`msj`) se pasa a través de **ES**, y su desplazamiento a través de **BX**. Como siempre el código de la operación se pasa a través del registro **AH**, siendo su valor **`0x0E`** para el caso de *enviaMsjBuzon*, y **`0x0F`** para el caso de *recibeMsjBuzon*.

Como ejemplo de programa de usuario que utiliza estas llamadas al sistema tenemos el programa "*c:\miso\usrs_prg\menumsj\menumsj.c*", el cual hemos utilizado en el apartado 6 para ejercitarnos en el uso del paso de mensajes.

7. IMPLEMENTACIÓN DE LAS FUNCIONES DE SERVICIO

Por seguir con la normativa sobre los nombres de las funciones de servicio es apropiado que los nombres de los manejadores de las llamadas al sistema correspondientes a *enviaMsjBuzon* y *recibeMsjBuzon* sean **`so_enviaMsjBuzon()`** y **`so_recibeMsjBuzon()`** respectivamente. Estas funciones, al igual que las demás, deben ubicarse en el fichero *"c:\miso\llamadas.c"*.

El significado del descriptor de un buzón (`nBuz`) utilizado en las funciones `enviaMsjBuzon` y `recibeMsjBuzon` debe ser el índice del buzón en la tabla `buzon`, es decir si `nBuz` es un descriptor de buzón, el acceso al mismo sería: `buzon[nBuz]`.

Tal y como ya se comentó para la llamada "*sleep*", es posible acceder fácilmente a cualquiera de los registros pasados en la pila a través de la macro:

```
#define RGP(N,R)(tblProc[N].sp->R)
```

Que se encuentra declarada en el fichero "*llamadas.c*".

Por ejemplo, desde la función de servicio `so_enviaMsjBuzon()`, se puede acceder al parámetro '`nBuz`' pasado en el registro **CX** del siguiente modo:

```
unsigned miBuz = RGP (nprAtnd,C.X)
```

El valor del parámetro '`msj`' que es un puntero, se envía a través de los registros **ES** y **BX**, y para formar un puntero mediante dos "*words*" disponemos de la macro PTR(S,O). En este caso por ejemplo podríamos usarlo del siguiente modo:

```
char far *miMsj = PTR(RGP(nprAtnd,es),RGP(nprAtnd,B.X))
```

Si deseamos conocer los nombres de todos los registros que se pueden usar con el parámetro '**R**' de la macro, podemos ver los campos de la estructura de datos "`regsPila_t`" declarada en el fichero "*c:\miso\tipos.h*".

Para realizar las copias de los mensajes desde una zona de memoria cualquiera a otra, necesitamos emplear punteros largos y sería deseable disponer de alguna función que realizara dicho trabajo. Afortunadamente disponemos de ella. La función '`_fmemcpy()`' de *turboC* que utiliza punteros largos y cuyo prototipo podemos incluir con `#include <mem.h>`. Por ejemplo, para copiar el mensaje cuya dirección está en '`pMsj`', al buffer '`i`' del buzon '`n`', haríamos lo siguiente:

```
_fmemcpy(&buzon[n].buf[i], pMsg, LONG_MSJ);     //LONG_MSJ=16
```

Un aspecto muy interesante es el siguiente ¿Dónde está (dónde se guarda) el mensaje de un proceso que se ha bloqueado al intentar enviar un mensaje? La respuesta es que el mensaje está en el espacio de direcciones del proceso emisor, y que la dirección de comienzo de ese mensaje está en los registros **ES:BX** de la trama de registros guardados en pila pertenecientes al contexto del proceso emisor. El sistema operativo no tiene ningún problema en acceder a esos campos para posteriormente recoger el mensaje y completar ese envío. La forma de hacerlo recomendada es mediante el uso de la macro que ya hemos visto: RGP (N,R). En este caso pondríamos en el parámetro '**N**' el numero de proceso que obtengamos de la cola del buzón, y como siempre en el '**R**' el registro, bien '**es**' o bien '**B.X'**.

Y análogamente, el problema anterior se vuelve a plantear y se resuelve de forma similar, para el caso en el que lo que queremos es obtener la dirección de la variable destino donde dejar un mensaje dirigido a un proceso receptor cuando éste está bloqueado.

Por último y para acabar, se recuerda que no hay que olvidar la inicialización de la tabla de buzones del sistema 'SO', estática o dinámicamente.

APÉNDICE IV. Práctica: Compactación de memoria.

ÍNDICE

1. OBJETIVOS

Los objetivos de esta práctica son:

- Que el alumno profundice sobre el conocimiento de la gestión de memoria de un pequeño sistema operativo.
- Que el alumno sea capaz de implementar en un pequeño sistema operativo, un procedimiento para compactar los huecos de memoria que se van produciendo durante el ciclo de creación y destrucción de procesos, ó asignación y liberación del recurso memoria.

2. INTRODUCCIÓN

En esta práctica se pretende implementar un comando de la consola del sistema operativo de prácticas que compacte los huecos de memoria.

Durante la vida del sistema el recurso memoria es asignado en bloques de tamaño variable, que normalmente se utilizan para ubicar el código y los datos de los procesos. Cuando estos procesos mueren el sistema libera la memoria ocupada por ellos, produciéndose un hueco, o espacio de memoria contigua, de tamaño igual al bloque de memoria que se le asignó cuando fue creado. Estos huecos acaban eventualmente estando dispersos por toda la memoria, produciendo lo que se conoce como fragmentación externa de memoria. El objetivo de este comando es fusionar todos estos huecos dispersos en uno sólo, eliminando así dicha fragmentación.

3. TRABAJO A REALIZAR

Vamos a presuponer que el alumno conoce ya el funcionamiento del sistema 'SO', tanto a nivel de usuario, introduciendo comandos a través de la consola o haciendo llamadas al sistema desde un programa, como a nivel de la estructura del programa 'SO', puesto que se da por hecho que ha realizado todas las prácticas anteriores.

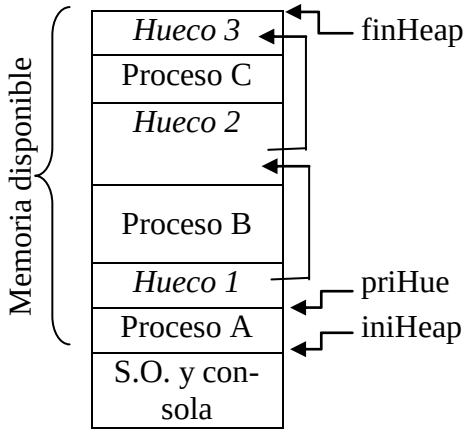
Tal y como ya se comentado en la introducción, el trabajo a realizar consiste en la implementación de un nuevo comando, al que llamaremos "*compac*" que se encargará de fusionar todos los huecos de memoria en uno sólo, eliminando completamente la fragmentación. Para conseguirlo, el comando tendrá que mover los objetos ubicados en la memoria asegurándose de que el cambio sea totalmente transparente para los procesos. Ello es posible gracias a que los procesos en 'SO' admiten reubicación dinámica, aunque eso sí, asumiendo algunas pequeñas restricciones. Actualmente todos los procesos de usuario disponibles como ejemplo las cumplen. En general la principal restricción consiste en que los procesos no deben emplear punteros largos para acceder a zonas de memoria que no le corresponden y que pudieran formar parte de algún objeto ubicado en memoria con posibilidad de reubicación. Esta restricción podría incluso superarse si se asumieran algunas reglas de comportamiento a seguir por dichos procesos y el propio 'SO'.

4. IMPLEMENTACIÓN DEL COMANDO 'COMPAC'

Nuestro objetivo es añadir este comando a la consola de 'SO', cuyo código se encuentra fundamentalmente dentro del fichero "so.c" perteneciente a los fuentes de 'SO'. En dicho fichero se declara el tipo "*simb_t*" que es un enumerado con todos los posibles comandos. Igualmente, se declara la tabla "*cmds[]*" que contiene los literales de estos mismos comandos, existiendo una correlación directa fácilmente visible entre ambas declaraciones. El añadir a estas declaraciones el nuevo comando es algo sencillo y un primer paso. También es posible y apropiado, añadir dicho comando a los mensajes informativos que muestra el comando ya existente "ayuda" en este momento, aunque esto por supuesto no es estrictamente necesario, ya que no influye directamente en los objetivos del nuevo comando.

Con los cambios indicados en el punto anterior ya podemos incorporar una nueva etiqueta "*case*" al "*switch*" principal de la función "*main()*". Este "*switch*" se encarga de distribuir la ejecución de cada uno de los comandos de 'SO'. La modificación podría consistir simplemente en invocar una nueva función, llamada por ejemplo "compactaMem()", que sería la que tendría que llevar a cabo todo el trabajo. El código de esta función debería estar incluido por coherencia con el resto en el fichero, en el módulo "*memoria.c*", y su prototipo en "*memoria.h*". De este modo podría ser utilizado desde "so.c". Hasta aquí con todo esto, habríamos realizado todos los cambios necesarios en el modulo "so.c", aunque hay que señalar que esta parte es la más sencilla del trabajo a realizar.

En el módulo "*memoria.c*" se encuentra el código fuente que contempla todos los aspectos relativos y relevantes acerca de la gestión de memoria en 'SO', aunque para entender y llegar a implementar el trabajo pedido tendremos que estudiar también los módulos: "*ficheros.c*", "*procesos.c*" y "*windows.c*", a los que se hará referencia más adelante, indicando aquellos aspectos que hay que conocer para poder llevar a cabo la implementación del comando "*compac*". En la figura siguiente se muestra un ejemplo de mapa de memoria.



En él se muestran tres procesos cargados en memoria y también tres huecos. En la zona inferior, en las direcciones más bajas de memoria se encuentra 'SO', donde a su vez se halla en él, en su espacio de memoria, el proceso consola. Justo donde acaba la última dirección de memoria del 'SO' empieza la memoria de uso dinámico para asignación. El valor de comienzo de esta zona lo indica la variable "*iniHeap*". La variable "*finHeap*" indica el final de la memoria disponible. La operación "*finHeap*"-"*iniHeap*" nos da el valor de la variable "*memDisponible*". Todas estas variables la establece 'SO' durante el proceso de inicialización. El tipo de datos de estas varia-

bles es "*word_t*" (unsigned int) y por tanto lo que representan son direcciones de segmento en el caso "*iniHeap* y *finHeap*" y *paragraphs* (16 bytes) en el caso de "*memDisponible*".

En el módulo memoria están implementadas las funciones "T*omaMem()*" y "S*ueltaMem()*" que sirven respectivamente para solicitar memoria y devolverla, y aunque son muy importantes, en principio no es necesario conocerlas en profundidad para realizar el trabajo pedido. La función "*mostrarMemoria*", que se encarga de mostrar en pantalla la lista de huecos de memoria a petición del comando "*mem*", es interesante estudiarla, ya que nos permite ver como se recorre la lista de huecos, no obstante tampoco esto es estrictamente necesario para cumplir nuestros objetivos, aunque eso sí, usaremos el comando "*mem*" durante el desarrollo de la práctica para ver los resultados de nuestro trabajo.

La lista de huecos está implementada usando los propios huecos para guardar los enlaces de la lista. El primer hueco lo indica la variable "*priHue*" y los siguientes se obtienen accediendo a la zona del hueco mediante punteros y utilizando el campo "*uSig*" para acceder al siguiente hueco y el campo "*tam*" para conocer el tamaño del hueco en "*paragraphs*" (unidades de 16 bytes). Los tipos de datos empleados se muestran a continuación:

```
/* ------------------------------------------------------------------------
   Tipos de datos para crear una lista de huecos en la memoria disponible pa-
   ra programas. Los "links" se guardan en los propios huecos. Especifican el
   tamaño del    hueco y un puntero al siguiente. Este puntero es del tipo
   "_seg" que es especial ya que tiene siempre un offset implícito '0' que no
   ocupa espacio. Hago una unión con el tipo "word_" para operar mejor con
   él.
   ------------------------------------------------------------------------ */

typedef struct foo _seg *pHue_t; // offset siempre = 0 (ocupa un word)
typedef union {
  pHue_t p;    // dato que se puede ver como puntero sin offset
  word_t w;    // ó como un word
} uHue_t;
struct foo {  word_t tam;  uHue_t uSig; };
/* ------------------------------------------------------------------------*/
```

El empleo de la estructura "*foo*" es por motivos sintácticos. Nos permite la declaración de campos del tipo "*pHue_t*" dentro de la unión "*uHue_t*". Lo que nos interesa es saber que tenemos un tipo de datos "*uHue_t*" que es una unión (es decir los dos campos dentro de ella, "p" y "w" permiten el acceso a una sola cosa, pero de dos formas distintas, como puntero y como entero sin signo), y otro tipo "*pHue_t*" puntero a "*struct foo*", que usaremos para acceder al interior de los huecos, para obtener el siguiente de la lista y el tamaño del mismo. Por ejemplo la variable "*priHue*" que es del tipo "*uHue_t*" podemos usarla escribiendo "*priHue.p*" para acceder a la dirección del primer hueco. Para acceder al tamaño del primer hueco pondríamos: "*priHue.p->tam*", ó bien para acceder al siguiente hueco al primero: "*priHue.p->uSig*", etc. Si por otra parte queremos obtener el valor del segmento de la dirección contenida en dicho puntero para copiarlo a una variable tipo "*word_t*" para operar con él usaríamos entonces "*priHue.w*". Para familiarizarse con estas operaciones es aconsejable estudiar un poco el código de este módulo.

A continuación vamos a explicar a grandes rasgos una posible forma de llevar a cabo la compactación.

Para compactar es necesario conocer las direcciones y tamaños de todos los objetos ubicados en la memoria de asignación dinámica. Estos objetos actualmente son sólo las FAT de los drives,

los procesos y las ventanas. Para obtener estos datos hay que examinar el contenido de la tabla de información de las unidades de disco (***infDrv[]***) y la tabla de procesos (***tblProc[]***), para ir rellenando, en una nueva tabla de objetos, las direcciones y tamaños de los objetos ubicados en memoria que hemos obtenido de dicho examen. Estos objetos son, en el caso de las unidades de disco, la '**FAT**' de cada unidad montada, en el caso de los procesos, las áreas de código y datos del mismo, y en el caso de las ventanas, el objeto '**win_t**' que guarda toda la información sobre una ventana terminal. La introducción de estos datos en la nueva tabla debe ser de tal modo que la tabla debe quedar al final ordenada por direcciones, preferiblemente de menor a mayor.

Una vez que tengamos registrados en la nueva tabla todas las direcciones y tamaños de todos objetos que el sistema 'SO' tiene ubicados en memoria dinámica, y además tengamos los elementos de esta tabla ordenados por su dirección de memoria, podremos recorrer esta tabla siguiendo su ordenación, e iremos comprobando por cada elemento de la tabla, si su dirección es menor que una previamente establecida a la que llamaremos "**base**", cuyo valor inicial es la del primer hueco de la lista. Si la dirección del objeto fuera menor que la de la "base" no habría que reubicar el objeto y pasaríamos al siguiente objeto de la tabla, si por el contrario fuera superior, habría que reubicar el objeto, es decir habría que mover el objeto (FAT, proceso o ventana) desde su posición de memoria actual, a la dirección de "base", y en este caso actualizaríamos al nuevo valor "base = base + tamaño del objeto reubicado". Cuando hayamos llegado al último objeto de la tabla tendríamos en "base" la dirección del único hueco resultante de este proceso, siendo su tamaño igual a la diferencia entre *finHeap* y *base* (tam = finHeap - base).

El procedimiento visto hasta ahora reubicaría los objetos, pero habría que hacer más cosas para que el sistema siga funcionando correctamente. Habría que modificar también todas las direcciones o punteros que el sistema 'SO' mantiene sobre aquellos objetos que se han reubicado. De estos punteros sólo haría falta cambiar la parte de "segmento" de la dirección, ya que la reubicación se ha efectuado siempre con alineamiento a "*paragraph*".

Seguidamente vamos a comentar en función del objeto reubicado, qué punteros hay que modificar y dónde se encuentran éstos dentro de 'SO'.

Si el objeto reubicado es una 'FAT' basta con modificar el puntero a la misma que se encuentra en la tabla de información de unidades de disco (*infDrv[]*). Esta tabla está definida en "*ficheros.h*", y en ella tenemos el campo "**pFat**" que es un puntero largo a byte (*fptrb_t*) que apunta a la FAT de la unidad (si es que está montada, sino valdría NIL). Si durante la reubicación hemos tomado nota del nuevo segmento de reubicación y de qué FAT se trata, podríamos hacer la modificación utilizando la macro *SEG(p)* que nos permite acceder a la parte "segmento" de un puntero. Por ejemplo, supongamos que "*newSeg*" es el nuevo segmento de dirección de memoria del objeto FAT reubicado correspondiente a la unidad "*drv*", entonces podríamos escribir:

```
SEG(infDrv[drv].pFat) = newSeg;
```

Alternativamente, también podríamos usar la macro *PTR(S,O)*, que nos construye un puntero a partir de un segmento (O) y un desplazamiento (offset, O), y podríamos escribir:

```
infDrv[drv].pFat = PTR(newSeg,0);
```

Ya que las direcciones de todos los objetos ubicados en memoria dinámica tienen siempre el offset a 0 (ya que las funciones de tomar memoria y soltar memoria devuelve un valor del tipo word que resulta ser el segmento de la zona de memoria que se asigna o libera).

Pasemos a analizar el caso de que el objeto reubicado fuera un 'proceso'. En este caso habría que hacer los siguientes cambios:

a) Modificar la dirección de memoria (segmento) especificada en el descriptor del proceso que indica dónde está cargado el proceso en memoria (***dirMem***).

b) Modificar el segmento del puntero (***sp***) que guarda la dirección de la cima de la pila (puntero de pila o *stack pointer*) donde se guardan los registros del contexto del proceso.

c) Modificar los registros **CS, DS** y **ES**, que se hallan en la trama de pila del contexto del proceso con el nuevo valor de segmento reubicado (*dirMem*). Para acceder a estos valores podemos utilizar el valor de '***sp***' del siguiente modo: `tblProc[np].sp->cs=dirMem;`
donde '***np***' es el numero de proceso reubicado y '***dirMem***' la nueva dirección de segmento.
Los otros dos registros, DS y ES, dado el modelo de memoria que utilizan los procesos de usuario, podemos asumir perfectamente que tendrán el mismo valor que el registro CS, por ello la modificación es similar a la ya vista.

En este momento conviene hacer la siguiente reflexión: El descriptor de proceso tiene también un campo puntero llamado '***pWin***' que contiene la dirección de la ventana terminal del proceso, la cual es uno de los objetos de posible reubicación. En el momento en el que se están haciendo los cambios relacionados con la reubicación del proceso, hemos modificado los punteros que hacían referencia a la nueva posición del proceso, retocando ciertos campos del descriptor del proceso, sin embargo este campo '**pWin**' hace referencia a la ventana del proceso, la cual puede haber cambiado de posición o no. Este cambio se tratará cuando se efectúe el proceso de retoque de los punteros mantenidos por 'SO' referentes a las ventanas (lista de ventanas), lo cual se comentará en el punto siguiente.

Vamos a tratar por último el caso en que el objeto reubicado sea una ventana terminal. Este caso es algo más complejo ya que las ventanas forman parte de una lista que esta ordenada por la posición que ocupa la ventana en la pantalla (primer plano, segundo plano, etc.) y hay que modificar los punteros de dicha lista referentes a las ventanas que se hayan reubicado. En primer lugar vamos a ver algunas de las estructuras de datos usadas por 'SO' para el manejo de ventanas, las cuales se encuentran definidas en el fichero "*windows.h*":

```
typedef struct wfoo win_t;
typedef win_t _seg *pWin_t;

#define SIZE_KEYBUF 127  // no pasar de 255
struct wfoo {
  char nombre[12];       // nombre de la ventana
  // --- Información de ventana ---
  pantalla_t  plano;     // guarda la información visual (una pantalla)
  int         dx, dy;    // desplazamiento del plano respecto a la pantalla
  pos_t       eSI, eID;  // coordenadas del marco de ventana
  byte_t      atr;       // atributo normal de la ventana
  pos_t       cursor;     // posición del cursor
  pWin_t pWDw, pWUp;      // ventana de debajo y de encima
  // --- Informacion de teclado ---
  byte_t keyBuf [SIZE_KEYBUF];
  byte_t nKeys;           // cantidad de teclas en el buffer
  byte_t ent,sal;         // posiciones de tecla entrante y saliente
  cola_t cola;            // cola de procesos esperando recibir tecla
};
extern pWin_t pWinTop;
extern pWin_t pWinFocal;  // ventana con el foco del teclado
```

Las ventanas de los procesos están organizadas formando una lista que está ordenada con arreglo a la posición que ocupan a la hora de ser mostradas en pantalla. Si suponemos una tercera dimensión, la profundidad, las ventanas se muestran y en pantalla ocultándose unas a otras. Podemos hablar de las ventanas que están encima y de las que están debajo, o también la ventana de primer plano, de la ventana del fondo, del fondo de pantalla, etc. Pues bien las ventanas forman una lista donde la primera es la que se presenta en primer plano y la última la que se encuentra más al fondo o en último plano.

Del listado de código de arriba, nos interesa sobre todo la variable "***pWinTop***" que es un puntero a la primera ventana de la lista (ventana de primer plano) y los campos '***pWDw***' y '***pWUp***' de la estructura '***win_t***', que son los punteros que tiene toda ventana apuntando a la ventana que se encuentra debajo y arriba de ella respectivamente. Son estos punteros los que hay que modificar para que se ajusten a las nuevas posiciones de memoria de las ventanas cuando éstas se reubican.

No hay que olvidar que además de los cambios en esta lista, también habría que cambiar el campo '***pWin***' del descriptor del proceso, tal y como ya se ha comentado con anterioridad.

Es conveniente por último hacer dos consideraciones. 1) El procedimiento de compactación debe efectuarse con las interrupciones inhibidas, ya que si durante la reubicación se intentase efectuar un cambio de proceso el resultado podría sería desastroso, y 2) A la hora de mover los objetos en memoria hay que tener en cuenta el sentido de la copia por posibles solapamientos. TC aporta una función que tiene esto en cuenta "***movemem()***" pero sólo admite como parámetros punteros cortos, y necesitamos punteros largos y por otro lado, las funciones de copia que admiten punteros largos como parámetros (***_fmemcpy()***..), dejan indefinido el resultado si hay solapamiento, por lo que tampoco pueden usarse, por lo tanto a falta de una solución mejor, habrá que implementar una función específica de copia para conseguir nuestros propósitos. Para facilitar esta tarea al alumno, se muestra a continuación una posible implementación de esta función:

```
/* Mueve 'sz' paragraphs desde el segmento 'o' al segmento 'd'. lo
   hace en unidades dword (4 bytes) para mejorar la velocidad. lo movido
   debe ser menor de 64KB y 'd' debe ser menor que 'o' si se solapan */

void mueveMem (word_t d, word_t o, word_t sz) {
  word_t i;
  sz <<= 4; // multiplico por 16 para pasar a bytes
  for (i=0; i < sz; i+=4)
    *((fptrd_t)PTR(d,i)) = *((fptrd_t)PTR(o,i));
}//mueveMem
```