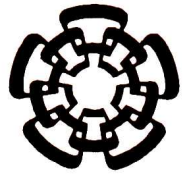


CENTRO DE INVESTIGACIÓN Y DE
ESTUDIOS AVANZADOS DEL INSTITUTO
POLITÉCNICO NACIONAL

DEPARTAMENTO DE INGENIERÍA
ELÉCTRICA
SECCIÓN COMPUTACIÓN

Emulación Prototipo de Sistemas DOS/8086

Tesis que presenta

Nélida Alicia Casas Reyes

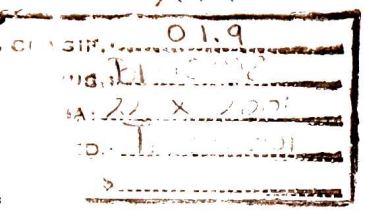
Para Obtener el Grado de

MAESTRA EN CIENCIAS
En la Especialidad de Ingeniería Eléctrica con Opción en Computación

Director de tesis: Dr. Jorge Buenabad Chávez

México, Distrito Federal Abril, dos mil uno.

XM



AGRADECIMIENTOS

Agradezco:

Al CINVESTAV, por permitirme el uso de su infraestructura.

Al Consejo Nacional de Ciencia y Tecnología, por su apoyo económico.

Al doctor Jorge Buenabad Chávez, por su invaluable apoyo y tiempo dedicado a la dirección y revisión de esta tesis.

Al doctor Arturo Díaz Pérez, por su apoyo y tiempo dedicado a la revisión de este documento.

Al doctor J. Oscar Olmedo Aguirre, por el tiempo dedicado a la revisión de este documento.

A la señora Sofía Reza Cruz, por su apoyo durante mi estancia en este centro y en el proceso de trámites finales.

A Hugo García Monroy, por la aportación de ideas para el desarrollo de esta tesis.

A Amilcar Meneses Viveros, por su apoyo y la aportación de ideas para el desarrollo de esta tesis.

Dedico esta tesis:

A mis padres.

Resumen

El diseño y desarrollo de un nuevo sistema de cómputo contempla el ofrecer compatibilidad con otros sistemas con el propósito de ganar aceptación general. Un nuevo sistema A debe comportarse como, o emular a, otro sistema B; de tal manera que los programas desarrollados para el sistema B puedan correr también en el sistema A. En efecto, el sistema A ofrece un sistema o máquina virtual B.

El diseño y desarrollo de una máquina virtual no es trivial. Es necesario un conocimiento profundo del hardware y del sistema operativo del sistema a emular y del sistema que ofrecerá la emulación. Y hasta donde sabemos no hay un documento disponible que describa el desarrollo completo de una máquina virtual.

Esta tesis presenta el diseño y desarrollo de una máquina virtual, un emulador prototipo de la plataforma conformada por el sistema operativo DOS y el procesador 8086/88, corriendo en un ambiente multitarea dentro del sistema operativo NeXTSTEP sobre el procesador 80486 y compatibles. Nuestro emulador soporta operaciones de entrada/salida sobre la terminal y archivos en disco. Y con algunas limitaciones, es capaz de correr programas ejecutables desarrollados en DOS sin modificarles.

Nuestro emulador completará la práctica de cursos sobre Sistemas Operativos y Arquitectura de Computadoras. Y con algunas extensiones, permitirá al sistema NeXTSTEP ofrecer compatibilidad completa con el sistema DOS. Además, los conceptos que presentamos son útiles para el desarrollo de aplicaciones sobre sistemas heterogéneos, donde la compatibilidad es fundamental.

Contenido

Lista de Figuras

Capítulo 1

Introducción

La aceptación general de un nuevo sistema de cómputo no sólo depende del tipo y la calidad de los servicios que ofrece para el desarrollo de aplicaciones. Muchas aplicaciones desarrolladas en sistemas ya establecidos presentan una inversión costosa que no es fácil de cambiar o actualizar.

La compatibilidad con otros sistemas es uno de los aspectos más importantes considerados en el diseño de los nuevos sistemas de cómputo. Microsoft siempre ha cuidado mucho el ofrecer compatibilidad con las aplicaciones de sus versiones anteriores. Algo similar ocurre en el caso de Intel en el desarrollo de sus procesadores, donde cada actualización soporta el conjunto de instrucciones de las versiones previas. Por su parte *soffice* [SUN 01] aún cuando no puede correr programas DOS/Windows, permite procesar archivos desarrollados en éstos sobre UNIX, ofreciendo así una compatibilidad restringida. Y *Samba* [REDHAT 99] es una colección de programas que permite a clientes y servidores Linux y Microsoft Windows (3.11, 95, NT) intercambiar archivos y servicios de impresión.

Para que un sistema A sea compatible con otro sistema B, A debe emular el comportamiento de B, ofreciendo a aplicaciones los servicios que B ofrece. La emulación de un sistema puede desarrollarse en software, en hardware, o ambos, y es conocida como *máquina virtual*. La implementación de máquinas virtuales no es una tarea fácil. Se requiere un amplio conocimiento del hardware sobre el cual se encuentra el sistema operativo que permitirá la emulación y del sistema operativo mismo, así como del sistema a emular. Y aunque existe bibliografía con conceptos y algunos aspectos de diseño e implementación, no es posible imaginarse como realmente se contruyen las máquinas virtuales. De ahí que el objetivo primordial de esta tesis sea de carácter educativo, pues pretende ejemplificar todo el proceso seguido para lograr

la construcción de una máquina virtual.

Esta tesis presenta el diseño y desarrollo de una máquina virtual, un emulador prototipo de la plataforma conformada por el sistema operativo DOS y el procesador 8086/88, corriendo en un ambiente multitarea dentro del sistema operativo NeXTSTEP [APPLE 01] sobre el procesador 80486 y compatibles. Nuestro emulador soporta operaciones de entrada/salida sobre la terminal y archivos en disco. Y con algunas limitaciones, es capaz de correr programas ejecutables desarrollados en DOS sin modificarles.

Debido a la complejidad de los sistemas operativos, los cursos de este tema son en general teóricos, aún cuando ahora ya se dispone del código fuente de algunos de ellos, cosa que no era así desde que los fuentes de UNIX dejaron de estar disponibles a instituciones educativas. Con esta tesis se pretende reforzar los cursos de sistemas operativos, como ya se ha hecho con sistemas operativos como MINIX sobre 8086 [TANENBAUM 94] y XINIX [BUENABAD 89]. Esta tesis podría aportar ideas para la construcción de una máquina virtual que soporte a MINIX sobre WINDOWS. Además, con cierto esfuerzo se podría lograr la compatibilidad total de DOS sobre el sistema operativo NeXTSTEP.

El resto de la tesis está organizada como sigue.

El capítulo 2 presenta un panorama general del concepto de máquinas virtuales, con la finalidad de ubicar al lector en el contexto del trabajo realizado.

El capítulo 3 describe las características del micronúcleo Mach, plataforma sobre la cual funciona el sistema operativo NeXTSTEP, sobre el cual desarrollamos nuestro emulador.

El capítulo 4 describe la arquitectura del procesador 80486 de Intel, destacando su operación en el modo virtual 8086. Esta arquitectura constituye la plataforma de hardware sobre la que se realizó la emulación.

El capítulo 5 explica los detalles de diseño e implantación de las estructuras del sistema de archivos que emulamos y la emulación de los servicios básicos para el manejo de archivos en DOS.

El capítulo 6 presenta el diseño e implantación del monitor que logra la ejecución de aplicaciones DOS sobre el sistema NeXTSTEP.

El capítulo 7 presenta los resultados obtenidos de nuestra emulación y pruebas realizadas sobre el mismo.

Finalmente, en el capítulo 8, presentamos las conclusiones y sugerimos trabajo futuro.

Capítulo 2

Máquinas Virtuales

Una máquina virtual es un mecanismo de hardware y software que funciona de manera idéntica al hardware de una computadora particular. Un sistema de cómputo puede correr concurrentemente una o más máquinas virtuales, posiblemente diferentes, y así ofrecer compatibilidad con otros sistemas o un ambiente para desarrollar software para otros sistemas, entre otras ventajas. Los conceptos y terminología de máquinas virtuales presentados en este capítulo son esenciales para entender los conceptos presentados en el resto de la tesis.

2.1 Introducción

Conceptualmente, un sistema de cómputo está construido mediante capas. El hardware es el nivel más bajo de todos estos sistemas. El núcleo ejecutándose en el siguiente nivel, usa instrucciones de hardware para crear un sistema operativo o conjunto de llamados al sistema operativo que pueda ser usado por las capas exteriores. Sin embargo, los programas del sistema que están arriba del núcleo pueden usar tanto llamados al sistema como instrucciones de hardware, y, de alguna manera estos programas no distinguen entre estos dos. Aunque son accedidos de diferente manera, ambos proporcionan la funcionalidad que un programa puede aprovechar para crear funciones aún más avanzadas. Los programas del sistema, a su vez, tratan el hardware y los llamados al sistema como si ambos estuvieran en el mismo nivel y los programas de aplicación pueden ver todo lo que se encuentra bajo ellos como si fuera parte de la máquina misma.

Con esta aproximación en capas, la mayoría de los sistemas operativos ofrecen a

los procesos una visión enriquecida y simplificada del hardware. Sin embargo, algunos sistemas operativos hacen que la interfaz a los procesos sea vista como la interfaz de hardware, es decir, un proceso tiene permitido usar todas las instrucciones, incluyendo las privilegiadas. La interfaz del proceso es llamada **máquina virtual** debido a que se visualiza como la máquina principal, y el núcleo de tal sistema operativo es llamado **núcleo virtualizante** o **máquina monitor virtual**.

Los sistemas operativos de máquina virtual son útiles debido a que permiten a los diseñadores de sistemas operativos experimentar con nuevas ideas sin interferir con la comunidad de usuarios. Antes de los sistemas operativos de máquinas virtuales, todas las pruebas a los sistemas tenían que ser realizadas sobre máquinas separadas dedicadas a tales propósitos. Cualquier error en el núcleo, sin importar lo trivial que pudiera ser, podría traer como consecuencia la caída total del sistema y cualquier usuario desafortunado que pudiera estar corriendo sus programas en ese momento, perdería una significativa cantidad de trabajo. Con un sistema operativo de máquina virtual, la versión de prueba del sistema operativo puede correr como un proceso controlado por el núcleo virtualizante. Los otros procesos que estan corriendo programas ordinarios no son afectados por errores en la versión de prueba. El unico efecto es que ellos no corren tan rápido debido a que la versión de prueba compite con ellos por los recursos. Este esquema se muestra en la figura 2.1.

Otro uso de los sistemas operativos de máquinas virtuales es integrar bloques y modos interactivos de diferentes máquinas virtuales. Este esquema permite colocar juntos dos sistemas operativos fundamentalmente diferentes en la misma máquina.

La posibilidad de correr varios sistemas operativos en la misma máquina a la vez tiene también otras ventajas [FINKEL 86]:

- Puede aliviar el trauma del cambio a nuevas versiones de sistemas operativos, dado que la versión anterior puede ser usada sobre una de las máquinas virtuales hasta que los usuarios conmuten a la nueva versión, que está corriendo sobre una máquina virtual diferente bajo el control del núcleo virtualizante.

- Puede permitir a estudiantes escribir sistemas operativos reales sin interferir con los otros usuarios de la máquina.

- Puede incrementar la confiabilidad en el software aislando componentes de software en diferentes máquinas virtuales.

- Puede incrementar la seguridad aislando programas y datos sensitivos en su propia máquina virtual.

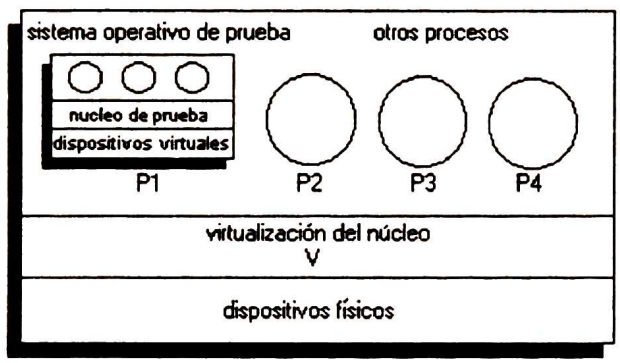


Figura 2.1: Probando un nuevo sistema operativo sobre una máquina virtual

- Puede probar facilidades de red simulando comunicación máquina a máquina entre diferentes máquinas virtuales sobre una sola máquina física.

- Puede proporcionar a cada usuario una máquina virtual separada en la cual corra un sistema operativo de un solo usuario. El diseño e implementación de este sistema operativo simple puede ser mucho más fácil que el diseño de un sistema operativo multiusuario, dado que puede ignorar detalles de protección y multiprogramación.

IBM introdujo las máquinas virtuales con el programa de control CP-67 sobre el Sistema/360 modelo 67, enfocándose en el aislamiento de una máquina virtual de otra ([MACKINNON 79]). CP-67 estaba disponible para proveer al hardware del Sistema/360 con una variedad de ambientes de sistemas operativos o máquinas virtuales independientes todas unas de las otras. Esto fue completado con hardware para la traducción dinámica de direcciones y el uso de múltiples espacios de direcciones virtuales supervisadas por CP, y el manejo por parte del CP de la máquina real. El Sistema/360 divide las instrucciones en instrucciones privilegiadas de control del sistema e instrucciones de cálculo no privilegiadas, permitiendo a CP alcanzar sus objetivos mediante la explotación del hardware real cuando sea posible.

Las máquinas virtuales se han vuelto a poner de moda como el medio para resolver problemas de compatibilidad entre sistemas. Por ejemplo, existen miles de programas disponibles para MS-DOS sobre sistemas basados en procesadores Intel. Los fabricantes de otros sistemas como Sun Microsystems y Digital Equipment Corporation (DEC) usan otros procesadores más rápidos, pero quisieran que sus clientes pudieran correr estos programas MS-DOS. La solución es crear una máquina virtual Intel encima del procesador nativo, de manera que el programa MS-DOS pueda hacer sus llamados al sistema como es usual. El resultado final es un programa que parece estar corriendo sobre un sistema basado en Intel pero que realmente está ejecutándose sobre un procesador muy diferente. Si el procesador es suficientemente rápido, el programa MS-DOS correrá rapidamente aún cuando cada instrucción esté siendo traducida a varias instrucciones nativas para su ejecución. De manera similar, el sistema Apple Macintosh basado en Power-PC incluye una máquina virtual Motorola 68000 para permitir la ejecución de código binario escrito para el Macintosh anterior al 68000 [SILBERSCHATZ 98].

2.2 Organización

Usando la planificación del CPU y técnicas de memoria virtual, un sistema operativo puede crear la ilusión de múltiples procesos ejecutándose cada uno de ellos en su propio procesador con su propia memoria (virtual). Por supuesto, normalmente los procesos tienen características adicionales, tales como llamados al sistema y un sistema de archivos que el hardware desnudo no proporciona. Por otro lado, el enfoque de máquinas virtuales no proporciona ninguna función adicional pero sí una interfaz que es idéntica al hardware desnudo subyacente. Cada proceso es provisto con una copia virtual de la computadora subyacente.

2.2.1 Aspectos de Diseño

Los recursos de la computadora física son compartidos para crear las máquinas virtuales. Se puede usar planificación del CPU para compartir la CPU y crear la ilusión de que los usuarios tienen su propio procesador. Mediante *spooling* y un sistema de archivos se pueden proveer de lectores de tarjetas virtuales e impresoras de líneas virtuales. Una terminal normal de usuario de tiempo compartido da la funcionalidad de consola del operador de la máquina virtual.

Una de las mayores dificultades del enfoque de máquina virtual está relacionada

con los sistemas de disco. Si una máquina física tiene tres unidades de disco y se quieren soportar siete máquinas virtuales evidentemente no se puede asignar una unidad de disco a cada máquina virtual, de tal manera que el software de la máquina virtual necesitará considerable espacio en disco para emular cada uno de los siete discos requeridos. La solución es proveer de discos virtuales idénticos en todo menos en tamaño. Éstos reciben el término de *minidiscos* en el sistema operativo VM de IBM [SILBERSCHATZ 98]. El sistema implementa cada minidisco reservando tantos sectores como el minidisco necesite en los discos físicos, siendo la suma del tamaño de todos los minidiscos menor que la cantidad actual de espacio físico disponible en disco. Así, cada usuario recibe su propia máquina virtual y puede ejecutar cualquiera de los sistemas operativos y paquetes de software que estén disponibles en la máquina. En el caso del sistema VM de IBM, los usuarios normalmente ejecutan CMS, un sistema operativo monousuario. El software de máquina virtual se ocupa de multiprogramar varias máquinas virtuales dentro de una máquina física, pero no necesita considerar software de apoyo a los usuarios. Esta organización podría representar una división útil, en dos partes más pequeñas, del problema de diseñar un sistema interactivo multiusuario.

2.2.2 Aspectos de Implementación

Aunque el concepto de máquina virtual es útil, es difícil de implementar. Para crear un duplicado exacto de la máquina subyacente se requiere un gran esfuerzo. Recordar que la máquina subyacente tiene dos modos: el modo de usuario y el modo de monitor. El software de máquina virtual se ejecuta en modo monitor, ya que éste es el sistema operativo. Así como la máquina física tiene dos modos, la máquina virtual debe tenerlos también. Como consecuencia se deben tener un modo usuario virtual y un modo monitor virtual ambos ejecutándose en un modo usuario físico. Aquellas acciones que causen una transferencia del modo de usuario al modo monitor sobre una máquina real (tales como llamados al sistema o intentos de ejecución de instrucciones privilegiadas) deben también causar una transferencia del modo de usuario virtual al modo monitor virtual en una máquina virtual.

Los sistemas operativos de máquinas virtuales son complejos. Para proporcionar velocidad aceptable, el hardware ejecuta la mayoría de las instrucciones directamente. Podría pensarse que el núcleo virtualizante **V** puede ejecutar todas sus máquinas virtuales en un estado privilegiado y dejarlas usar todas las instrucciones de hardware. Sin embargo es demasiado riesgoso dejar a los procesos ejecutarlas directamente. En vez de esto, **V** debe correr todas las máquinas virtuales en modo de usuario para evitar

interferencias accidentales o maliciosas entre ellas y con V mismo [FINKEL 86]. De hecho, los sistemas operativos de máquinas virtuales no pueden ser implementados en computadoras donde las instrucciones peligrosas son ignoradas en el modo de usuario, por ejemplo, el PDP-11/45 en modo usuario falla sus trampas sobre varias instruciones peligrosas. En general, una instrucción es peligrosa si realiza entrada y salida, manipula registros de traducción de direcciones o manipula el estado del procesador (incluyendo instrucciones de retorno de interrupciones y asignación de prioridad).

Se permite a las máquinas virtuales imaginar que tienen el control de los estados del procesador aunque en realidad no es así. V guarda el estado del procesador virtual de cada máquina. Esta información es almacenada en el bloque de contexto de las máquinas dentro de V. Todas las instrucciones privilegiadas ejecutadas por las máquinas virtuales causan trampas para V, quien emula el comportamiento del hardware representativo de la máquina virtual.

- Si una máquina virtual estaba en modo de usuario, V emula una trampa para la máquina poniéndola en modo de núcleo virtual, aunque realmente aún corre en modo físico de usuario. Al contador de programa de la máquina virtual se le asigna la dirección de la trampa apropiada. Se dice que V ha **reflejado** la trampa para la máquina virtual.

- Si la máquina estaba en modo virtual de núcleo, V emula la acción de la instrucción misma. Por ejemplo, termina la máquina con una instrucción *halt* y ejecuta instrucciones de entrada y salida interpretativamente.

El núcleo virtualizante puede querer simular un dispositivo por otro, simulando, por ejemplo, una impresora sobre un disco o un disco pequeño sobre una parte de otro más grande. Una interrupción de un dispositivo puede indicar que una operación de entrada y salida iniciada sobre alguna máquina virtual ha terminado. En este caso, la interrupción debe ser reflejada a la máquina virtual apropiada. Pero la emulación de una simple operación de entrada y salida para una máquina virtual puede requerir varias operaciones; por lo que la interrupción de un dispositivo a menudo indica que V puede proceder al siguiente paso de la emulación de una operación que está ya en progreso. Tales interrupciones no son reflejadas. Si la computadora se comunica con dispositivos a través de registros con direcciones en memoria principal, todos los accesos a esa región de la memoria deben causar trampas para que V pueda emular la entrada y salida. La traducción de direcciones también llega a ser muy compleja.

2.2.3 Micronúcleos

El potencial para el desarrollo de sistemas operativos distribuidos se alcanzó al inicio de los años 70's, pocos años después de haber surgido las minicomputadoras. El uso de minicomputadoras como estaciones de trabajo de un solo usuario y su efectividad para el desarrollo de software y otras tareas interactivas fueron influencias importantes que motivaron la búsqueda de un ambiente de cómputo más eficiente que el ofrecido por sistemas multiusuario de tiempo compartido, pero no se contaba con el hardware y software necesarios para hacer a las computadoras de un solo usuario completamente efectivas, ni se tenían las facilidades de comunicación que les permitiera ser usadas de manera cooperativa.

Desde inicios de 1980 se dió una rápida expansión de investigación y desarrollo en sistemas distribuidos. Mach y Chorus son dos ejemplos de núcleos de estos sistemas operativos. El *núcleo* de un sistema operativo se distingue porque su código es ejecutado con privilegios totales de acceso a los recursos físicos de la computadora. En particular, puede controlar la unidad de manejo de memoria y manipular los registros del procesador.

Además de Mach y Chorus, Amoeba, Clouds y System V son ejemplos de interés debido a sus contribuciones técnicas [COULOURIS 94]. Todos estos proyectos emplean un núcleo mínimo o micronúcleo como base de sus diseños.

Un micronúcleo proporciona los mecanismos básicos sobre los cuales las tareas de manejo de recursos en general deben ser ejecutadas. No existe una definición fija sobre qué funciones deben ser realizadas por un micronúcleo, sin embargo, todos ellos incluyen manejo de procesos, manejo de memoria y paso de mensajes. Los micronúcleos pueden variar en tamaño desde alrededor de diez kilobytes hasta varios cientos de kilobytes de código ejecutable y datos estáticos.

Las principales ventajas de un sistema operativo basado en micronúcleo respecto a aquellos que cuentan con un núcleo monolítico son su apertura y posibilidad de reforzar la modularidad, además de los mecanismos de protección de los límites de memoria. Por otro lado, un núcleo relativamente pequeño es más fácil de ser liberado de errores que uno más grande y complejo. Actualmente, las estructuras de los micronúcleos hacen posible la ejecución simultánea de múltiples sistemas operativos. Por ser los más usados, sistemas operativos como UNIX y DOS generan la necesidad de ser emulados, y un medio para lograrlo es la implantación de estos sistemas como aplicaciones en ejecución sobre un micronúcleo.

2.3 Ejemplos

En esta sección presentamos ejemplos de la organización de máquinas virtuales.

2.3.1 Máquina Virtual/370

La máquina virtual/370 (VM/370) es un ejemplo de un sistema monitor de máquina virtual (VMM) [STUART 74]. Un VMM es una forma especial de sistema operativo que multiplexa sólo los recursos físicos entre los usuarios. En particular, la máquina virtual provista por una VMM es idéntica a la máquina desnuda sobre la cual la VMM se ejecuta. De esta manera, VM/370 por ejemplo, hace parecer a un Sistema/370 ser muchos Sistemas/370, como se ve en la Figura 2.2. VM/370 completa este esquema controlando la multiplexión de los recursos físicos del hardware de manera análoga a la forma en la que la compañía de teléfono multiplexa las comunicaciones, permitiendo conversaciones separadas y aisladas sobre los mismos alambres. VM/370 consiste de dos componentes principales: (1) El programa de control (CP), el monitor de la máquina virtual, que realiza las funciones del procesador, memoria y multiplexión de dispositivos de entrada y salida para producir las máquinas virtuales; y (2) el sistema monitor conversacional (CMS), un sistema operativo simple que realiza las funciones de procesamiento de comandos, manejo de información y manejo limitado de dispositivos (por ejemplo, todas las funciones necesarias por un usuario en una consola de un sistema de tiempo compartido). Dado que el CMS solo usa asignación de memoria contigua y no proporciona multiprogramación, tiene un manejo extremadamente simple de la memoria y el procesador.

Típicamente, el CP y el CMS son usados juntos, aunque un individuo puede elegir usar cualquier otro sistema operativo Sistema/360 o Sistema/370, tales como OS/360, OS/VS-1 u OS/VS-2 en lugar de CMS. CP y CMS representan una clara división en el diseño de VM/370. Dividiendo los sistemas en dos partes iguales, el diseño y la implementación son simplificados sustancialmente. Además, dado que CP y CMS son construídos sobre una interfaz de máquina desnuda, pueden ser implementados y depurados en paralelo en vez de serialmente.

En CP-40/CMS y CP-67/CMS, el componente CMS primero fue depurado sobre una máquina real antes de haber sido usado sobre una máquina virtual. De hecho, la mayor parte de CMS fue operacional antes de que el componente CP fuera completado. Esta es la forma en que muchos sistemas operativos explotarán en el futuro el enfoque de máquinas virtuales.

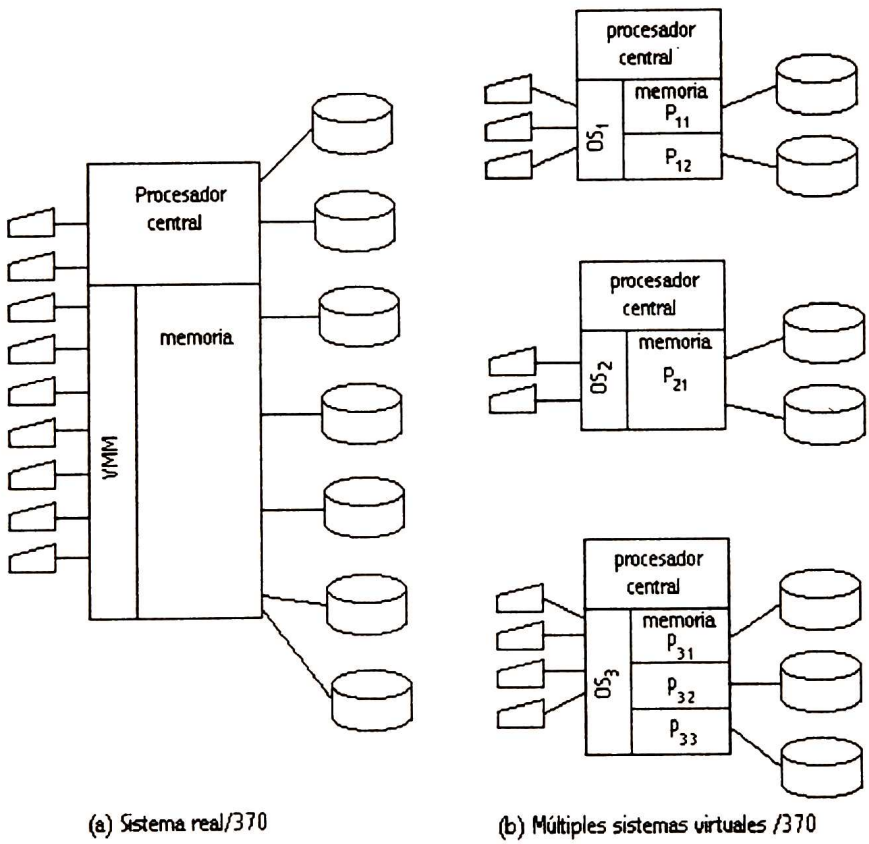


Figura 2.2: Máquinas real y virtual en el Sistema VM 370

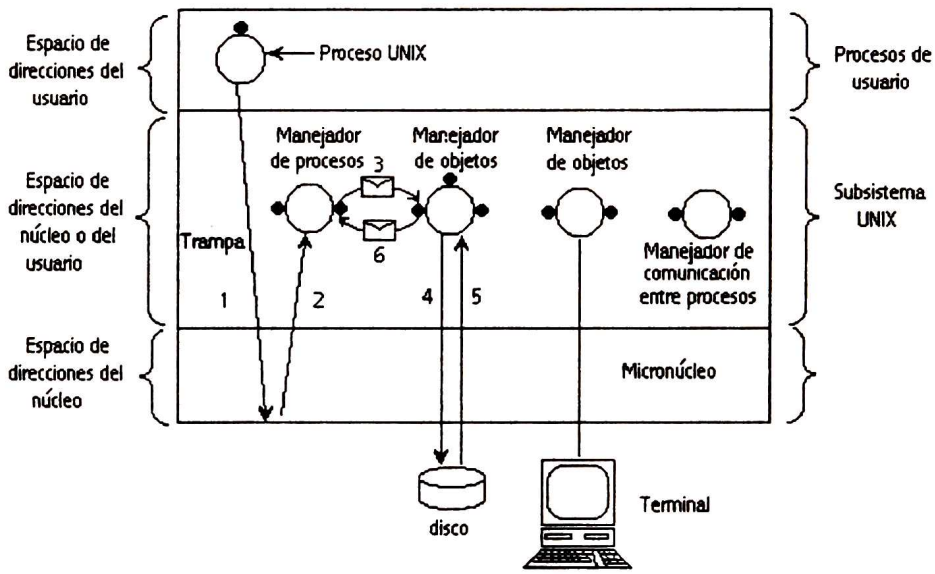


Figura 2.3: Estructura de la emulación de UNIX en Chorus.

2.3.2 Emulación de UNIX en Chorus

Chorus es un sistema operativo basado en micronúcleo para usar en sistemas distribuidos. Proporciona compatibilidad binaria con UNIX System V, soporte para aplicaciones de tiempo real y programación orientada a objetos. Chorus consta de tres capas conceptuales: la capa del núcleo, los subsistemas y los procesos de usuario. La capa del núcleo contiene el micronúcleo, algunos procesos que corren a nivel del núcleo y comparten el espacio de direcciones del micronúcleo. La capa media contiene los subsistemas que son usados para proveer soporte a los programas del usuario que residen en la capa superior.

La implementación del subsistema UNIX es construida a partir de cuatro componentes principales: el **manejador del procesador**, el **manejador de objetos**, el **manejador de flujo** y el **manejador de la comunicación entre procesos**, como se observa en la Figura 2.3. Cada uno de estos componentes tiene una función específica en la emulación. El manejador de procesos captura los llamados al sistema y procesa su manejo. El manejador de objetos manipula los llamados al sistema de

archivos y la actividad de paginación. El manejador de flujo se ocupa de la entrada y salida. El manejador de comunicación entre procesos realiza la comunicación entre procesos (IPC) de System V. El manejador de procesos es código nuevo, el resto es tomado en su mayoría de UNIX mismo para minimizar el trabajo de diseño y maximizar la compatibilidad. Estos cuatro manejadores pueden manipular cada uno múltiples sesiones, de manera que sólo uno de ellos esté presente en un sitio dado, sin haber problema de cuantos usuarios estén activos en el sistema. En el diseño original, los cuatro procesos deberían haber estado dispuestos a correr en modo núcleo o en modo de usuario. Sin embargo, al ser agregado código más privilegiado, esto fue más difícil de hacer. Ahora en la práctica todos ellos normalmente se ejecutan en modo núcleo, lo cual también es necesario para dar un desempeño aceptable. Para ver como se relacionan las partes, se examinará como son procesados los llamados al sistema. Al momento de la inicialización del sistema, el manejador de procesos dice al núcleo que quiere manejar los números de trampa estándar que UNIX AT&T utiliza para hacer los llamados al sistema (para lograr compatibilidad binaria). Cuando un proceso UNIX más tarde envía un llamado al sistema para ser atrapado por el núcleo, un hilo en el manejador de procesos toma el control. Este hilo actúa como si fuera el mismo hilo que hizo el llamado, de forma análoga a la que son hechos los llamados al sistema en sistemas UNIX tradicionales. Dependiendo del llamado al sistema, el manejador de procesos puede realizar el llamado al sistema mismo o enviar un mensaje al manejador de objetos solicitándole hacer el trabajo. Para llamados de entrada y salida, el manejador de flujo es invocado. Para llamados IPC, se utiliza el manejador de comunicaciones.

2.3.3 Emulación de UNIX en Mach

Mach es un sistema operativo basado en micronúcleo. Fue diseñado como base para la construcción de nuevos sistemas operativos y emulación de algunos ya existentes. También proporciona una manera flexible de extender UNIX a multiprocesadores y sistemas distribuidos.

Mach tiene varios servidores que corren encima de él. Probablemente el más importante es un programa que contiene una gran cantidad del UNIX de Berkeley (por ejemplo, esencialmente el código completo del sistema de archivos) dentro de sí mismo. Este servidor es el principal emulador de UNIX. Este diseño es un legado de la historia de Mach como una versión modificada de UNIX de Berkeley.

La implementación de la emulación de UNIX en Mach consiste de dos partes, el servidor UNIX y una biblioteca de emulación de llamados al sistema, como se observa

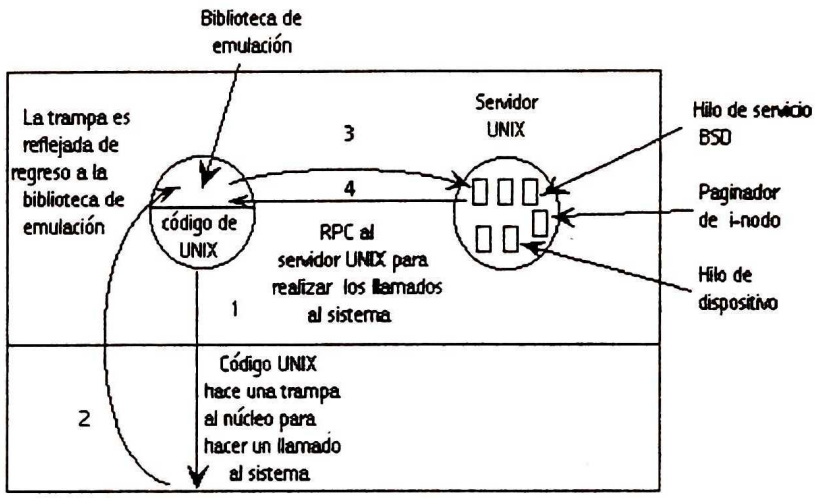


Figura 2.4: La emulación en Mach usa el mecanismo de trampolín.

en la Figura 2.4. Cuando inicia el sistema, el servidor UNIX instruye al núcleo para capturar todos los llamados al sistema, lo cual genera una trampa que los vectoriza a una dirección dentro de la biblioteca de emulación del proceo UNIX que está haciendo el llamado. A partir de ese momento, cualquier llamado al sistema hecho por un proceso UNIX dará el control temporalmente al núcleo pasando inmediatamente después a su biblioteca de emulación. Al momento de que el control es dado a la biblioteca de emulación, todos los registros de la máquina tienen los valores que tenían al momento de la trampa. Este método de rebote desde el núcleo hacia el espacio del usuario es conocido como **mecanismo de trampolín**.

Una vez que la biblioteca de emulación tiene el control, examina los registros para determinar qué llamado al sistema fue invocado. Entonces, hace un RPC (llamado a procedimiento remoto) a otro proceso del servidor UNIX para hacer el trabajo. Cuando finaliza, el programa de usuario obtiene el control de nuevo. Esta transferencia del control no necesita ser a través del núcleo.

Cuando el proceso **init** genera procesos hijos, automáticamente heredan la biblioteca de emulación y el mecanismo de trampolín, así que ellos también pueden hacer

llamados al sistema UNIX. El llamado al sistema **exec** ha sido cambiado para no reemplazar la biblioteca de emulación sino solo parte del programa UNIX del espacio de direcciones.

2.3.4 Emulación de DOS en Windows NT

El sistema operativo Windows NT es un sistema operativo multitareas expropiativo de 32 bits para procesadores modernos. NT es portable a varias arquitecturas de procesador. Más de una versión de NT han sido portadas a Intel 386 y superiores, MIPS R4000, DEC Alpha y el Power PC. Los objetivos claves del sistema son la portabilidad, seguridad, adhesión a la Interfaz de Sistema Operativo Portable (POSIX) o cumplimiento del estándar 1003.1 de IEEE [IEEE 96], soporte a multiprocesadores, extensibilidad, soporte internacional y compatibilidad con aplicaciones MS-DOS y MS-Windows. NT utiliza una arquitectura de micronúcleo (como Mach), lo que permite hacer mejoras a una parte del sistema operativo sin afectar demasiado las demás partes del sistema.

La arquitectura de NT es un sistema de módulos en capas. Las capas principales son la capa de abstracción del hardware, el núcleo y el ejecutivo que se ejecutan en modo protegido, y una gran colección de subsistemas que se ejecutan en modo usuario. Los subsistemas en modo de usuario pertenecen a dos categorías. Los subsistemas de entorno emulan diferentes sistemas operativos; los subsistemas de protección ofrecen funciones de seguridad.

2.3.5 Java

Java es un lenguaje que goza de gran popularidad, diseñado por Sun Microsystems y que se implementa con un compilador que genera *códigos de bytes* como salida. Estos códigos de bytes son las instrucciones que se ejecutan en la *Máquina Virtual Java* (JVM, *Java Virtual Machine*). Para que los programas Java se ejecuten en una plataforma, es necesario que dicha plataforma esté ejecutando una JVM. La JVM se ejecuta en muchos tipos de computadoras, incluidos los compatibles con IBM-PC, Macintosh, estaciones de trabajo y servidores Unix, y minicomputadoras y macrocomputadoras IBM. La JVM también se implementa en el pequeño JavaOS, pero aquí la implementación es directamente en hardware para evitar el gasto extra que implica ejecutar Java en sistemas operativos de propósito general. Por último, dispositivos de propósito específico, como los teléfonos celulares, se pueden implementar

a través de Java utilizando microprocesadores que ejecutan códigos de bytes Java como instrucciones nativas.

La Máquina Virtual Java implementa un conjunto de instrucciones basado en pila que incluye las instrucciones aritméticas, lógicas, de movimiento de datos y de control de flujo. Al ser una máquina virtual, también puede implementar instrucciones que son demasiado complejas para incorporarse en hardware, incluidas instrucciones para crear y manipular objetos e invocar métodos. Los compiladores de Java se limitan a emitir estas instrucciones de códigos de bytes y la JVM debe implementar la funcionalidad necesaria en cada plataforma.

El diseño de Java aprovecha el entorno completo que una máquina virtual implementa. Por ejemplo, se verifica que los códigos de bytes no incluyan instrucciones que pudieran poner en peligro la seguridad o confiabilidad de la máquina subyacente. Si el programa Java no satisface esta verificación, no se permitirá su ejecución. Mediante la implementación de Java como lenguaje que se ejecuta en una máquina virtual, Sun ha creado un recurso orientado a objetos eficiente, dinámico, seguro y transportable. Si bien los programas Java no son tan rápidos como los que se compilan para el conjunto de instrucciones del hardware nativo, sí son más eficientes que los programas interpretados y tienen varias ventajas respecto a los lenguajes de compilación nativa como C.

2.4 Resumen

La construcción en capas de la mayoría de los sistemas de cómputo, da a los procesos una visión simplificada del hardware. En algunos sistemas operativos, esta interfaz de los procesos con el hardware se conoce como **máquina virtual** y el núcleo de tales sistemas como **monitor virtual**. La utilidad de los sistemas de máquinas virtuales consiste en permitir a los diseñadores experimentar con nuevas ideas sin intervenir con la comunidad de usuarios. Otro uso es la integración de bloques y modos interactivos que permitan ocupar diferentes máquinas virtuales sobre la misma máquina física. Debido a la variedad de ventajas que genera el uso de máquinas virtuales, diversas compañias han incursionado en el diseño de este tipo de sistemas, siendo precursor IBM que introdujo CP-67 sobre Sistema/360, la máquina virtual 370, la emulación de UNIX en diferentes micronúcleos como Chorus y Mach, la emulación de DOS en Windows NT y la maquina virtual Java.

Capítulo 3

Mach

En este capítulo se describen las principales características del micronúcleo Mach. Los conceptos mencionados en este capítulo son necesarios para comprender el diseño del proyecto desarrollado.

3.1 Desarrollo de Mach

El proyecto Mach inició su desarrollo en la Universidad de Carnegie-Mellon (CMU); fue el sucesor de dos proyectos previos, RIG y Accent [BOYKIN 93]. RIG fue desarrollado en la Universidad de Rochester en los años 70's y Accent fue desarrollado en Carnegie-Mellon durante la primera mitad de los 80's. En contraste a sus predecesores RIG y Accent y al sistema Amoeba, [TANENBAUM 92], el proyecto Mach nunca fue diseñado para ser un sistema operativo completo, sino para ofrecer compatibilidad directa con UNIX BSD, proporcionando facilidades de núcleo avanzadas que pudieran complementar a las de UNIX y permitir a las implementaciones UNIX ser distribuidas a través de una red de multiprocesadores o de computadoras de un solo procesador. La intención de los diseñadores era implementar muchas de las funcionalidades de UNIX como procesos a nivel de usuario, es decir ejecutándose fuera del núcleo.

Han habido varias versiones de Mach. La primera, 1.0 fue distribuida por CMU para permitir a otros investigadores seguir el progreso del desarrollo de Mach. Esta versión tenía memoria virtual y subsistemas de comunicación entre procesos, pero no tenía soporte para hilos y tareas. Siguió la 2.0, una versión mas funcional que contenía soporte total para hilos y tareas. Esta última versión formó la base para los primeros productos comerciales. La versión 2.5 de Mach fue la primera de las dos mayores ver-

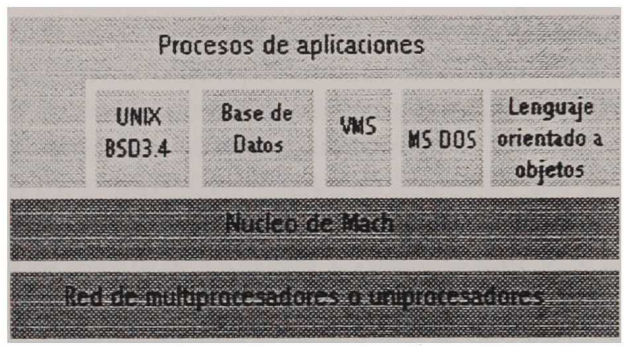


Figura 3.1: Soporte de Mach para sistemas operativos, bases de datos y otros subsistemas

siones e incluye gran parte del código de UNIX dentro del núcleo mismo. Esta versión corre sobre SUN-3, computadoras NeXT, el multiprocesador IBM RT PC, sistemas de multiprocesador y uniprocesador VAX y los multiprocesadores Encore Multimax y Sequent, entre otras computadoras. Desde 1989, Mach 2.5 fue incorporado como tecnología base para OSF/1, la Fundación de Software Abierto.

El código de UNIX fue removido desde la versión 3.0 del núcleo de Mach. Esta versión corre sobre PC's basadas en procesadores Intel 386 en adelante, las series de computadoras DEC 3100 y 5000, algunas computadoras basadas en Motorola 88000, estaciones SUN SPARC, computadoras Macintosh y algunas computadoras con los procesadores RS6000 de IBM, *Precision Architecture* de Hewlett Packard y Alpha de DEC. La versión 3 de Mach es la base para la construcción de emulaciones de sistemas operativos a nivel de usuario, sistemas de bases de datos, lenguajes orientados a objetos y otros elementos de software llamados subsistemas (Figura 3.1). Han sido ya implementadas las emulaciones de sistemas operativos tales como UNIX (BSD4.3, System V.4 y AIX de IBM), OS/2, MSDOS y VMS. La emulación de sistemas operativos convencionales hace posible la ejecución de código binario desarrollado para estos sistemas, además de que tales aplicaciones pueden tomar ventaja de los beneficios del

procesamiento distribuido que Mach ofrece.

Los principales objetivos de diseño y características de Mach son los siguientes:

Operación Multiprocesador: Mach fue diseñado para ejecutarse sobre multiprocesadores con memoria compartida, de tal manera que, tanto los hilos del núcleo como los hilos en modo usuario pudieran ser ejecutados por cualquier procesador. Mach proporciona un modelo multi-hilo de procesos de usuario, con ambientes de ejecución llamados *tareas*.

Extensión Transparente a operación en red: Para permitir a los programas distribuidos extenderse transparentemente entre uniprocesadores y multiprocesadores a través de una red, Mach ha adoptado un modelo de comunicación independiente de la localización involucrando puertos como destinos. El diseño de Mach confía totalmente en los procesos servidores de red a nivel de usuario para realizar el paso de mensajes de manera transparente a través de la red.

Servidores a nivel de usuario: Mach soporta un modelo basado en objetos en el cual los recursos son manejados por el núcleo o por servidores a nivel usuario. Como se ha mencionado, un objetivo primario ha sido implementar la mayoría de las facilidades UNIX a nivel de usuario, proporcionando compatibilidad con aplicaciones binarias UNIX. A excepción de algunos recursos manejados por el núcleo, los recursos son accedidos uniformemente mediante el paso de mensajes. A cada recurso le corresponde un puerto manejado por un servidor. El Generador de Interfaz de Mach (MIG) fue desarrollado para generar interfases RPC (*Remote Procedure Calls*) para ocultar accesos basados en mensajes a nivel de lenguaje [COULOURIS 94].

Emulación de Sistemas Operativos: Para soportar la emulación de UNIX y otros sistemas operativos a nivel de código binario, Mach permite la redirección transparente de llamados al sistema operativo a llamados a funciones en bibliotecas de emulación y de ahí a servidores del sistema operativo a nivel de usuario.

Implementación flexible de Memoria Virtual: Mucho del esfuerzo de diseño de Mach fue mejorar su manejo de memoria virtual para soportar la emulación de UNIX y otros subsistemas. Esto incluyó el uso de técnicas de mapeo de memoria como el mecanismo *copy-on-write* [RASHID 87]. *Copy on write* mapea dos o más páginas virtuales a una misma página física para evitar el costo de copiar los datos a menos estos sean modificados, por ejemplo, cuando los mensajes son pasados entre tareas. Finalmente, Mach fue diseñado para permitir que servidores corriendo a nivel de usuario implementaran respaldo de almacenamiento para páginas de memoria virtual, donde las regiones pueden ser mapeadas a datos manejados por paginadores externos.

Portabilidad: Mach fue diseñado para ser portable a una gran variedad de plataformas de hardware. Por esta razón, el código independiente de la máquina

fue aislado tanto como fuera posible. En particular, el código de la memoria virtual está dividido en una parte independiente y otra parte dependiente de la máquina.

3.2 Conceptos Fundamentales de Mach

Mach ofrece cinco abstracciones de programación que son la base para la construcción de un sistema. Estas abstracciones fueron escogidas como las mínimas necesarias para producir un sistema útil sobre el cual pudieran ser construidas las operaciones típicas. Estas abstracciones son:

Tareas. En Mach, el proceso tradicional UNIX es dividido en dos componentes separadas (Figura 3.2). La primera de estas componentes es la *tarea*, que contiene todos los recursos para un grupo de entidades cooperativas. Ejemplos de recursos en una tarea son los puertos de comunicación y la memoria virtual. Una tarea es una colección pasiva de recursos; no se ejecuta en un procesador.

Hilos. El segundo componente de un proceso en Mach es el ambiente de ejecución activo. Cada tarea puede soportar uno o más estados de ejecución, corriendo concurrentemente, llamados *hilos*. Gran parte del poder del modelo de programación de Mach viene del hecho de que todos los hilos de una tarea comparten los recursos de la tarea; por ejemplo, todos ellos toman el mismo espacio de direcciones virtuales de la tarea. Sin embargo, cada hilo en una tarea tiene su propio estado de ejecución privada. Este estado consiste de un conjunto de registros, tales como registros de propósito general, un apuntador a la pila, un contador de programa, etc.

Puertos. El canal a través del cual se comunican los hilos unos con otros se denomina *puerto*. Un puerto es un recurso propiedad de una tarea, por lo tanto, los hilos tienen acceso a un puerto en virtud de pertenecer a dicha tarea. Los programas cooperativos pueden permitir a hilos de una tarea tener acceso a puertos de otra tarea. Una característica importante de los puertos es que son localidades transparentes, lo cual facilita la distribución de servicios por la red sin modificación a los programas.

Mensajes. Los hilos en diferentes tareas se comunican entre ellos mediante el intercambio de mensajes. Un *mensaje* contiene una colección de datos de cierto tipo.

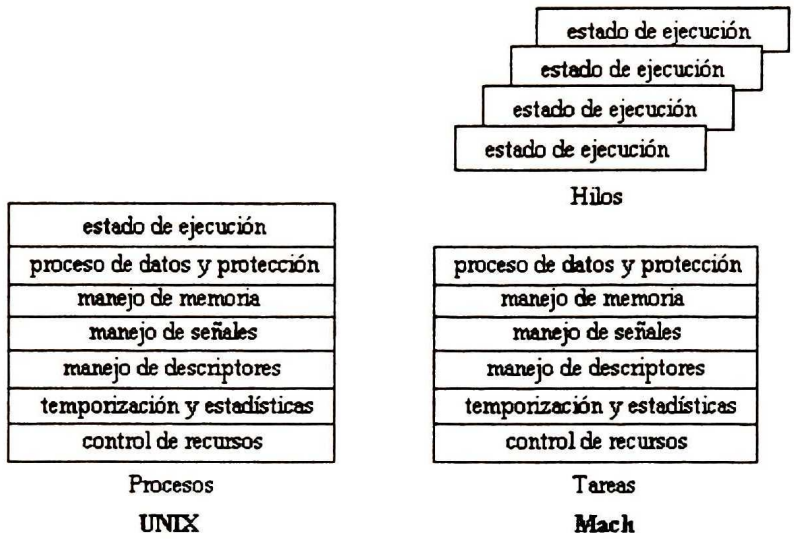


Figura 3.2: Proceso UNIX e Hilos y Tareas Mach

Estos datos pueden ser desde datos específicos de un programa tales como cadenas o números hasta datos relacionados con Mach tales como los parámetros de un puerto.

Objetos de Memoria. La posibilidad de realizar funciones tradicionales de un sistema operativo fuera del núcleo del sistema es una característica clave de Mach. Por ejemplo, Mach soporta políticas de paginación de memoria virtual en programas a nivel de usuario. Los objetos de memoria son una abstracción para soportar esta capacidad.

Excepciones. Las *excepciones* son interrupciones al procesamiento normal de las instrucciones de un programa causadas por el programa mismo. Las excepciones pueden ocurrir por varias razones que van desde el diseño explícito de un programa hasta errores en el programa, o intervención humana. El manejo de excepciones tiene gran importancia para las aplicaciones que las procesan como parte integral de su operación.

Todos los conceptos recién mencionados son parte fundamental de la programación de Mach y además son recursos usados por el núcleo mismo.

3.3 Tareas e Hilos

Uno de los elementos centrales de Mach es su ambiente de ejecución. Este ambiente permite a las aplicaciones optimizar su desempeño usando cómputo de ejecución concurrente, logrado a través de lo que se conoce como *hilos*. Un hilo esencialmente puede considerarse un procesador virtual, con su propio conjunto de registros y su propia información de estado. Para explotar esta capacidad, debe ser posible compartir datos y otros recursos de los hilos conveniente y eficientemente. Mach implanta esta compartición agrupando los recursos en una tarea. Cada hilo pertenece a una sola tarea y tiene acceso a todos los recursos de la tarea. A una tarea pueden estar asociados cualquier número de hilos. Tareas e hilos permiten el diseño de aplicaciones que automáticamente tomen ventaja de múltiples procesadores durante su ejecución.

Este modelo de tareas e hilos de Mach proporciona ventajas sobre el modelo correspondiente de ejecución de programas en UNIX debido a que el procesamiento paralelo es más natural con tareas e hilos; además de que la creación de hilos es menos costosa en tiempo de CPU que la creación de procesos UNIX.

Una tarea de Mach puede verse como un contenedor para una colección de recursos (Figura 3.3). Las tareas de Mach son pasivas, no ejecutan el código de los programas. Así como la mayoría de aplicaciones de UNIX usan un proceso, la mayoría de aplicaciones de Mach usan una tarea, aunque Mach también permite el diseño de aplicaciones con múltiples tareas, cuando tales aplicaciones requieren lo siguiente:

- Protección para un conjunto de recursos.
- Un conjunto diferente de recursos.
- Recursos adicionales

Esto se justifica de la siguiente manera: una tarea encapsula a un programa para brindar protección a sus límites, de manera que cuando un programa requiere protección contra otro *software*, puede usarse una tarea separada, ya que no es posible para una tarea referirse directamente a la memoria de otra tarea sin mediación del núcleo de Mach. Ejemplo de estas aplicaciones son los programas cliente/servidor, que se implementan como tareas separadas, de tal manera que malfunciones en el cliente no pueden afectar al servidor, y viceversa. Otro ejemplo es un depurador que necesita protección sobre el depurador mismo y el programa que está siendo depurado.

Otra razón para tener múltiples tareas es obtener recursos adicionales o diferentes, como puede ser un mayor espacio de direcciones de memoria.

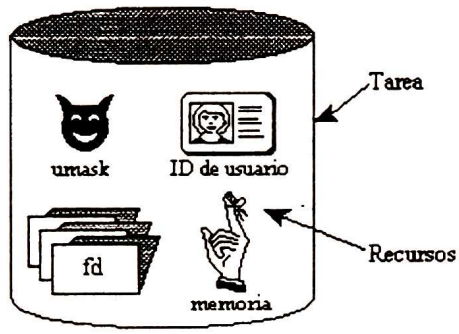


Figura 3.3: Tareas de Mach vistas como contenedores de recursos

En vez de crear múltiples tareas cuando se requiere hacer más de una actividad a la vez, Mach ofrece la posibilidad de emplear más de un hilo. Los hilos de Mach son activos, pues ejecutan código de programas almacenado en el espacio de direcciones de una tarea de Mach. Las tareas proporcionan el conjunto de recursos usados por sus hilos. El efecto más notable de la compartición de estos recursos es que todos los hilos de una tarea usan la misma memoria. Por supuesto la compartición de datos acarrea la responsabilidad de sincronizar los cambios hechos a los datos. Aunque toda la memoria de una tarea es compartida para todos sus hilos, en la práctica cada hilo no usa el espacio entero de direcciones. Por convención, cada hilo tiene un área del espacio de direcciones designado, así como su propia pila privada. Todas las pilas de los hilos son físicamente accesibles a todos los hilos de la tarea, pero típicamente cada hilo referencía solamente a su propia área asignada.

3.3.1 Llamados para el Manejo de Tareas e Hilos

A continuación se presentan algunos llamados al sistema para la manipulación de tareas e hilos en Mach.

La creación de una nueva tarea puede hacerse mediante un llamado al sistema **fork()**. A diferencia del *fork* tradicional de UNIX, este llamado en Mach realiza los siguientes pasos:

- Se crea una nueva tarea, la hija.
- Se copia la memoria de la tarea madre en la hija.

- Se crea un nuevo hilo perteneciente a la tarea hija.
- Se inicializa la información UNIX relacionada a la hija, tal como el ID del usuario e ID del proceso.
- Iniciar la ejecución del hilo recién creado.

De manera alternativa, existe el llamado al sistema **task_create()**, que permite la creación de una nueva tarea hija, devolviendo como resultado el identificador de la tarea recién creada. Los llamados siguientes utilizan este identificador (del tipo *task_t*) como argumento.

Mach permite suspender todos los hilos de una tarea a través del llamado **task_suspend()** que se considera síncrono pues no regresa hasta que todos los hilos de la tarea hayan sido suspendidos.

Las tareas suspendidas pueden ser puestas en ejecución nuevamente usando el llamado **task_resume()**.

El llamado al sistema **task_threads()** obtiene una lista de todos los hilos de una tarea; esta lista no sigue un orden en particular.

Es posible obtener información sobre cierta tarea usando el llamado **task_info()**. La información obtenida es similar a la del programa *ps* de UNIX, tal como la utilización de memoria de la tarea, el tiempo de ejecución de usuario y el tiempo de ejecución total del sistema, etcétera.

Al crear una nueva tarea, Mach le asigna un par de puertos. Uno de estos es el *puerto de notificación de la tarea*, al cual el núcleo dirige mensajes de advertencia indicando que un puerto al cual la tarea cedió derechos de envío ha sido destruido. El otro puerto es el *puerto de excepción de la tarea*, al cual Mach direcciona mensajes cuando los hilos generan excepciones.

Para registrar un puerto como de notificación o de excepción, Mach utiliza el llamado al sistema **task_set_special_port()**, en el cual se define qué tipo se le asigna a cierto puerto especificado como parte de los argumentos del llamado.

Mach proporciona el llamado **thread_create()** para la creación de un nuevo hilo, si el llamado se completa sin ningun error, el núcleo regresa el valor KERN_SUCCESS. El llamado puede fallar cuando el argumento pasado como tarea padre no es válido o si hay recursos insuficientes en el núcleo.

Aunque un hilo se haya creado todavía no puede ejecutarse, antes deben prepararse varias piezas de información, como el apuntador a la pila, la dirección de la primera instrucción a ejecutar, el estado de varias banderas, (bit del acarreo o indicador de desbordamiento), etc.

El llamado **thread_set_state()** especifica información del estado para un hilo. Dicho estado está contenido en una estructura *thread_state_t* que por convención incluye el nombre de la arquitectura, por ejemplo *i386_thread_state_t* para sistemas basados en arquitectura Intel 386. Una aplicación que emplea este llamado no es portable al hardware de múltiples arquitecturas.

Algunos programas, depuradores por ejemplo, necesitan obtener el estado actual de un hilo para poder realizar futuras modificaciones. El llamado **get_thread_state()** recupera esta información.

Cuando un hilo es creado, Mach lo inicializa en un estado suspendido; bajo esta condición, el hilo no se ejecuta sino hasta que se hace un llamado a **thread_resume()**.

La manera de detener temporalmente la ejecución de un hilo es mediante el llamado a Mach **thread_suspend()**. Cuando se desea terminar definitivamente la ejecución de un hilo se utiliza un llamado a **thread_terminate()**.

Cada hilo en Mach tiene un conjunto de puertos IPC que pueden servir para funciones especializadas. Dos de tales puertos son:

Puerto de respuesta. Los llamados al sistema Mach generalmente son implantados a través del intercambio de mensajes entre un hilo y el núcleo. Cuando una aplicación haga un llamado a **port_allocate()**, el núcleo de Mach utilizará el puerto ahí definido para comunicarse con la aplicación.

Puerto de excepción. Cuando una aplicación encuentra una excepción tal como un desbordamiento aritmético, Mach reacciona enviando un mensaje a dicho puerto.

Para establecer que un puerto (previamente reservado) sea de excepción o de respuesta, se utiliza el llamado **thread_set_special_port()**. Si se desea saber si cierto puerto es de excepción o respuesta se recurre al llamado **thread_get_special_port()**.

3.4 Comunicación Entre Procesos (IPC)

La comunicación entre dos programas, algunas veces llamados procesos, es llamada **comunicación interprocesos**, o IPC.

Conceptos Generales

Mach define a cada extremo de un canal de comunicación usado para pasar mensajes como un *puerto*. Los puertos son implementados en una cola de mensajes que el

núcleo maneja. Los mensajes contienen una colección de objetos de datos de cierto tipo. El programador define qué datos son pasados en un mensaje y el tipo de los datos, tales como caracteres de 32 bits, etcétera.

Derechos de Acceso

Las operaciones de comunicación entre procesos que una tarea puede realizar sobre un puerto definen el conjunto de **derechos de acceso**. Una tarea puede tener cualquier combinación de derechos de envío o recepción a un puerto. Las tareas que tengan derechos de envío sobre un puerto pueden enviar mensajes a tal puerto; de manera similar, aquellas tareas con derecho de recepción de un puerto pueden recibir mensajes del mismo. Debido a que los puertos son recursos pertenecientes a las tareas, cualquier hilo dentro de una tarea puede enviar o recibir mensajes a los puertos de esa tarea.

Una tarea automáticamente adquiere todos los derechos de acceso cuando crea un puerto. Estos derechos pueden ser pasados a otra tarea en cualquier momento enviando un mensaje a la tarea destino.

Cómputo Distribuido

A menudo, los modelos de cómputo distribuido ocultan la existencia de la red. Un programa que envía un mensaje no sabe ni tiene cuidado de que el receptor esté en la misma máquina o en una diferente. La comunicación entre tareas puede ejecutarse tanto en máquinas remotas como en locales. La transmisión proporciona un modelo simple para el cómputo distribuido. Para iniciar una conexión entre tareas que desean comunicarse, un servidor Mach debe "anunciar" un puerto sobre el cual acepta peticiones de servicios. Un programa cliente "advierte" al servidor, obtiene derechos de envío al puerto y usa el puerto para comunicación. Un servidor especial de Mach maneja ambas actividades, "anunciar" y "advertir", y aisla a los clientes y servidores regulares de todo conocimiento de la red. Mediante este intermediario, un cliente puede obtener derechos de envío y recepción del puerto de un servidor. Mach asegura que la computadora remota estará dispuesta a leer los datos, aún cuando las computadoras tengan diferentes representaciones de los mismos. El programador debe incluir el tipo de datos de cada bloque a transmitir. Con esta información, Mach traduce los datos de su representación en una máquina a otra.

A diferencia de otros mecanismos tradicionales de comunicación interprocesos, el de Mach no pone restricción en la cantidad de datos que pueden ser enviados en un mensaje.

Nombres de Puertos

Cada puerto es conocido por un *nombre* único, que es representado como un valor entero. Diferentes tareas pueden tener su propio nombre para el mismo puerto. Por ejemplo, si las tareas A y B están enviando mensajes a un servidor, la tarea A puede estar usando un puerto nombrado "doce", mientras que la tarea B conoce a tal puerto como puerto "siete". El receptor de mensajes puede conocer al puerto como puerto "cuarenta y dos".

3.5 Memoria Virtual

Así como Mach da al programador control total sobre tareas, hilos, mensajes IPC y puertos, también lo hace sobre la memoria de un programa a través de los llamados al sistema de memoria virtual. Usando estas interfases, un programa puede reservar, liberar, proteger y compartir rangos arbitrarios de memoria en cualquier tarea. Estos rangos de memoria no necesariamente deben ser contiguos.

El subsistema de memoria virtual de Mach combina funcionalidad con facilidad de uso y alto desempeño. La memoria puede ser manipulada rapidamente desde una sola página hasta el espacio de direcciones completo de manera transparente al usuario.

Existen algunos problemas que se presentan cotidianamente y pueden resolverse con el manejo de memoria virtual como los que a continuación se mencionan:

- Cuando un programa llega a ser demasiado grande para caber en memoria principal, el diseñador debe dividirlo en partes y arreglárselas para saltar de una parte a otra de memoria cuando sea necesario.

- Alternativamente, cuando un programa carga en memoria aquellas porciones que no están siendo usadas, se gastan valiosas porciones de memoria.

- Por otro lado, los programas cargados simultáneamente no tienen más alternativa que la confianza mutua, aunque por malicia o error, un programa puede corromper los datos de otro. Un programa también puede intentar acceder memoria que no existe o involucrarse inadvertidamente con dispositivos tales como controladores de disco, con resultados potencialmente desastrosos (Figura 3.4)

Para sobreponerse a estos problemas, muchas computadoras proporcionan una unidad manejadora de memoria (MMU). Mach realiza la configuración de esta unidad,

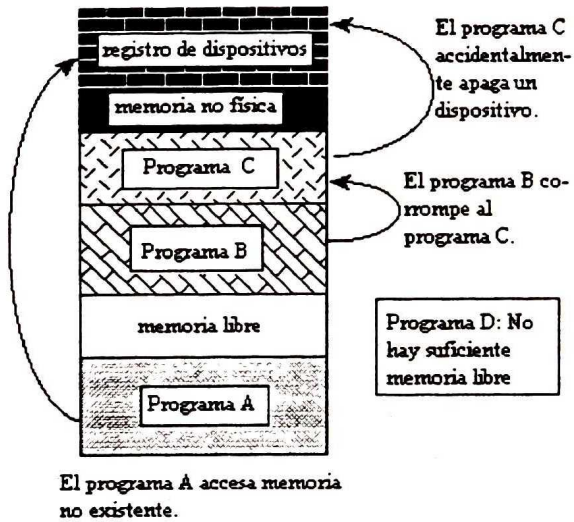


Figura 3.4: Problemas con el acceso a memoria

para traducir direcciones de memoria virtual generadas por un programa en direcciones físicas. Cada programa tiene asociado un mapa de traducción que el núcleo mantiene de direcciones virtuales a físicas. La MMU rechazará el intento de algún programa de acceder memoria que el núcleo no haya mapeado. Además, el núcleo programa la MMU para prohibir el acceso de los programas a memoria no existente, o regiones sensitivas tales como dispositivos mapeados en memoria.

En particular, Mach tiene soporte para el manejo de **unidades paginadas de memoria** (PMMU's). Este hardware divide el espacio de direcciones en porciones de tamaño fijo. De acuerdo a la plataforma, la longitud de las porciones varía desde los 512 hasta 8192 o más bytes; cada una de estas porciones constituye lo que se conoce como **página**. El uso de la PMMU permite a Mach manejar la memoria de los programas automáticamente de la forma que a continuación se describe. El núcleo, de manera invisible, divide a los programas en páginas, y no todas las páginas necesitan estar simultáneamente residentes en la memoria física. Cuando un programa referencia a una dirección de una página que no está en memoria, la PMMU alerta a Mach y éste carga a la memoria física la página que incluye la dirección referenciada obteniéndola del medio de almacenamiento secundario, actualiza la tabla de traducción de la

PMMU apropiadamente y continúa con la ejecución del programa en el punto en que lo dejó.

Mach también emplea estrategias adicionales para minimizar el uso de la memoria; por ejemplo el uso del mecanismo denominado *copy-on-write* (COW), optimiza la operación de copias frecuentes del contenido de una página a otra. En el caso tradicional, el hacer una copia de una página requiere la reservación de una segunda página (virtual) y la asignación de una página física en la cual a través de un ciclo, se efectúa la escritura de cada uno de los bytes de la primera página La estrategia *copy-on-write* difiere de la forma tradicional en que a ambas páginas virtuales se les asocia la misma página física siempre y cuando no se intente modificar el contenido de ninguna de ellas. Cuando la PMMU detecta un intento por modificar alguna de las páginas, el núcleo reserva una nueva página física y hasta este momento realiza en ella la copia. La técnica copy-on-write ahorra además de memoria, tiempo al evitar copias innecesarias.

Las tareas en Mach tienen su propio espacio de direcciones compartido por todos los hilos de la tarea. El modelo UNIX del espacio de direcciones está orientado a utilizar memoria contigua, de tal manera que los segmentos de texto, datos y bss (*Block Started by Symbol*, una región que se extiende después del segmento de datos, también conocida como *heap*) se encuentran juntos y mantienen un intervalo entre el segmento bss y la memoria no reservada. Cuando se hace una solicitud de memoria, esta memoria dinámica puede ser obtenida moviendo el apuntador bss sobre este intervalo. También se tiene un segmento de pila, que crece hacia abajo desde la parte superior de la memoria. A diferencia de UNIX, Mach ha sido diseñado para manejar grandes espacios de memoria eficientemente, los cuales se pueden encontrar *esparcidos* a lo largo del espacio de direcciones virtuales; es decir, puede haber varios rangos de memoria virtual separados por grandes intervalos.

3.5.1 Asignación de Memoria

Aún cuando todos los programas al iniciar su ejecución tienen asignada una porción de memoria virtual que contiene su código, datos y pila, la mayoría de los programas en algún momento requieren de memoria virtual adicional. Esta memoria extra requerida en tiempo de ejecución se conoce como memoria dinámica. Para hacer esto posible Mach cuenta con el llamado **vm_allocate()** que asigna nueva memoria para una tarea. Una vez que la memoria ya no es necesaria, es recomendable liberarla, para lo cual debe hacerse el llamado **vm_deallocate()** que sólo removerá memoria de la tarea solicitante y no de otras tareas que tengan compartida o copiada dicha memoria.

Las operaciones de transferencia de datos entre diferentes espacios de direcciones, se realizan mediante los llamados **vm_read() y vm_write()**. **Vm_read()** toma los datos de cierta tarea fuente y los coloca en un nuevo rango de páginas de la tarea que ejecutó el llamado; la cantidad de datos leídos debe ser múltiplo del tamaño de la página. El llamado **vm_write()**, que realiza una escritura de datos del espacio de direcciones de la tarea actual al de otra tarea, tambien tiene como restricción que la dirección de inicio donde se efectuará la escritura esté alineada con una página.

3.5.2 Protecciones de Memoria

Mach permite manejar protecciones o "permisos" sobre un rango de memoria con el llamado del sistema **vm_protect()**. Las protecciones a la memoria pueden ser cualquier combinación de:

VM_PROT_READ: permiso para leer el contenido de una página.

VM_PROT_WRITE: permiso para modificar el contenido de una página.

VM_PROT_EXECUTE: permiso para usar el contenido de una página como instrucciones de un programa.

VM_PROT_ALL: permiso de lectura, escritura y ejecución.

VM_PROT_NONE: la página no puede ser leída, escrita o ejecutada aunque ya contenga datos válidos.

La forma de combinar los valores de protección es colocándolos separados por barras formando un OR de los valores individuales, por ejemplo:
VM_PROT_READ|VM_PROT_WRITE.

3.5.3 Herencia de Memoria entre Tareas Relacionadas

Durante la creación de una tarea, Mach copia el contenido del espacio de direcciones de la tarea madre en el espacio de direcciones de la tarea hija, usando copy-on-write. En ocasiones, es útil poder compartir áreas de memoria entre tareas relacionadas; mediante la interfaz **vm_inherit()**, Mach permite ajustar el atributo de herencia para una región de memoria. Un rango de memoria marcado como "no heredable" en la tarea padre, estará ausente de la tarea hija, así como una área marcada como compartida podrá ser leída y escrita por ambas tareas, padre e hija, con todos los

cambios visibles para las dos. El atributo de herencia puede ser cualquiera de los siguientes:

VM_INHERIT_NONE: la región de memoria no existirá en una nueva tarea.

VM_INHERIT_SHARE: la región de memoria será compartida con tareas creadas subsecuentemente.

VM_INHERIT_COPY: la región de memoria será insertada copy-on-write en una tarea recientemente creada.

3.5.4 Mapeo de Archivos

Los archivos mapeados en memoria son uno de los beneficios del sistema de memoria virtual de Mach. Esta técnica consiste en que la totalidad o una parte de un archivo puede estar mapeado en una sección de memoria virtual. Posterior al mapeo, una referencia a una posición dentro de esta sección es equivalente a una referencia a la misma posición en el archivo físico. Si tal porción del archivo no se encuentra actualmente en memoria, ocurre un fallo de página que indica al núcleo solicitar del sistema de archivos la sección requerida del archivo para mapearla en la memoria física. Desde el punto de vista del proceso, el archivo entero se encuentra en la memoria a la vez.

3.6 Manejo de Excepciones

Las excepciones son una categoría de las interrupciones causadas por un programa durante su ejecución. Las causas de estas excepciones pueden variar desde el diseño explícito del programa, errores del mismo o intervención humana, por ejemplo el exceso en los límites de los rangos de datos, intento de acceder memoria virtual no existente, o puntos de ruptura dentro de un depurador.

El manejo de excepciones es de gran importancia para aplicaciones que las procesan como parte integral de su funcionamiento.

3.6.1 Filosofía en el manejo de excepciones

Mach separa la respuesta a una excepción del ambiente de ejecución donde ésta ocurrió. El núcleo suspende la ejecución del hilo víctima que encontró la excepción

y notifica a un hilo manejador de excepciones de la condición. Este diseño resuelve varios problemas inherentes del hilo víctima en relación a las excepciones:

- El hilo suspendido puede estar en un estado inconsistente.
- Puede ser difícil manejar múltiples excepciones al mismo tiempo debido a que un programa multihilo puede generar excepciones concurrentes.
- El código que maneja la excepción puede necesitar alguno de los recursos usados en el hilo víctima.

Estos problemas son resueltos por el mecanismo de excepciones de Mach de la siguiente manera. Para el primer caso, el código que corresponde a la excepción corre en un hilo diferente al hilo víctima de la excepción, y por lo tanto no es vulnerable a inconsistencias en el estado de la víctima. Para el segundo caso, Mach notifica al hilo manejador la ocurrencia de una excepción mediante un mensaje IPC, permitiendo al manejador serializar el procesamiento de múltiples excepciones concurrentes o bien tener varios hilos manejadores de excepciones procesándolas de manera simultánea. Y para el tercer caso, el uso de un hilo manejador separado también reduce la dependencia de los recursos usados en el hilo víctima, pues este hilo manejador cuenta con sus propios recursos.

Es notorio entonces que las aplicaciones que usan la facilidad para manejar excepciones siempre contienen al menos dos hilos: aquel en el cual ocurre la excepción y el hilo manejador que recibe el mensaje IPC advirtiendo de la excepción en la víctima. Típicamente, la víctima y el manejador se ejecutan en la misma tarea.

3.6.2 Pasos en el procesamiento de excepciones

Cuando ocurre una excepción, el sistema operativo suspende al hilo víctima y envía un mensaje IPC a un puerto especial llamado **puerto de excepción** del hilo, el cual pertenece a la víctima.

El segundo paso ocurre cuando el hilo manejador recibe un mensaje de notificación de excepción desde el puerto de excepción del hilo.

En el paso tres, se toma la acción apropiada en respuesta a la excepción. El manejador lleva a cabo el paso cuatro respondiendo al mensaje de notificación del núcleo con el resultado de la acción tomada; de esta manera, el núcleo decide entre continuar o terminar con la ejecución del hilo víctima. Este proceso se advierte en la figura 3.5

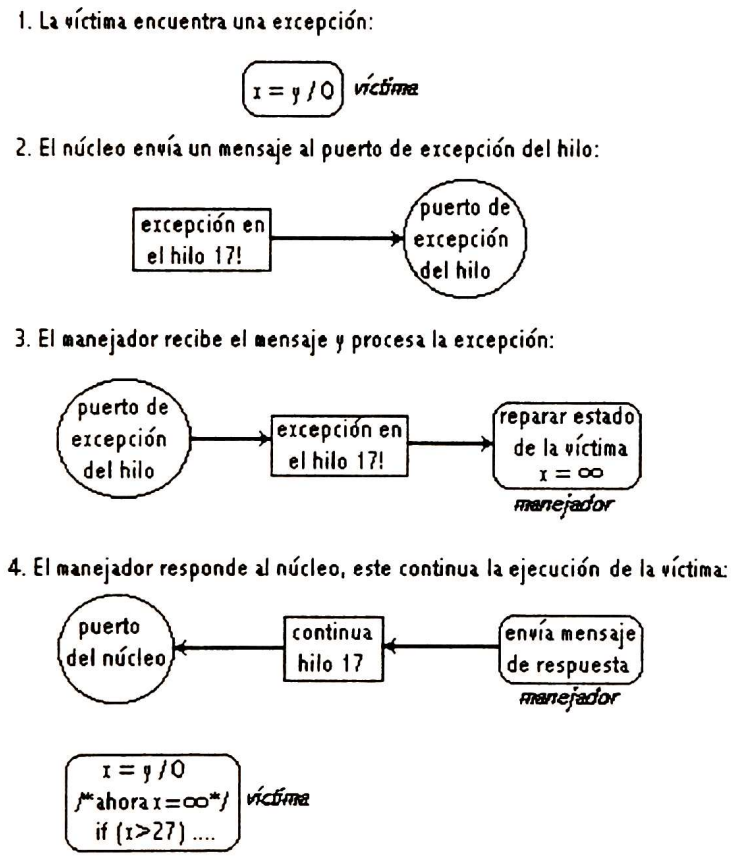


Figura 3.5: Pasos en el procesamiento de excepciones

Código	Significado
EXC_BAD_ACCES	No es posible acceder memoria
EXC_BAD_INSTRUCTION	Instrucción falló
EXC_ARITHMETIC	Excepción aritmética
EXC_EMULATION	Instrucción de emulación
EXC_SOFTWARE	Excepción generada por software
EXC_BREAKPOINT	Traza, punto de ruptura, etc.

Tabla 3.1: Significado de algunos códigos de excepción

Puertos de excepción

Un hilo designa un puerto como su puerto de excepción con el llamado al sistema **thread_special_port()**, especificando el identificador del hilo, la constante THREAD_EXCEPTION_PORT y el identificador del puerto previamente reservado que funcionará como puerto de excepción.

Para mostrar las operaciones que se realizan con los puertos de excepción, considerar una tarea con dos hilos: el que lleva a cabo los cálculos de la aplicación, que será denominado hilo A y el manejador de excepciones, el hilo B.

Inicialmente, el hilo A reserva un puerto y lo asocia a la variable global *exc_port*; posteriormente, ejecuta el llamado **thread_set_special_port** para registrar tal puerto como el puerto de excepción. el hilo B entra a su ciclo infinito efectuando las operaciones recibir, procesar y responder; la variable *exc_port* será usada para determinar qué puerto especificar en el llamado **msg_receive()**. Cuando el hilo A genera una excepción, el núcleo encuentra que A ha registrado un puerto de excepción y envía un mensaje a ese puerto notificando la ocurrencia de la excepción; este mensaje será recibido por el hilo B. Todas estas operaciones se ilustran en la figura 3.6

3.6.3 Mensajes de excepción

El mensaje enviado al puerto de excepción proporciona la información que el hilo manejador necesita para procesar la excepción. Este procesamiento requiere de la identificación del hilo víctima y el conocimiento de qué excepción ocurrió. Algunos códigos de excepción se muestran en la Tabla 3.1

El manejador de excepciones regresa un código de respuesta al núcleo de Mach,

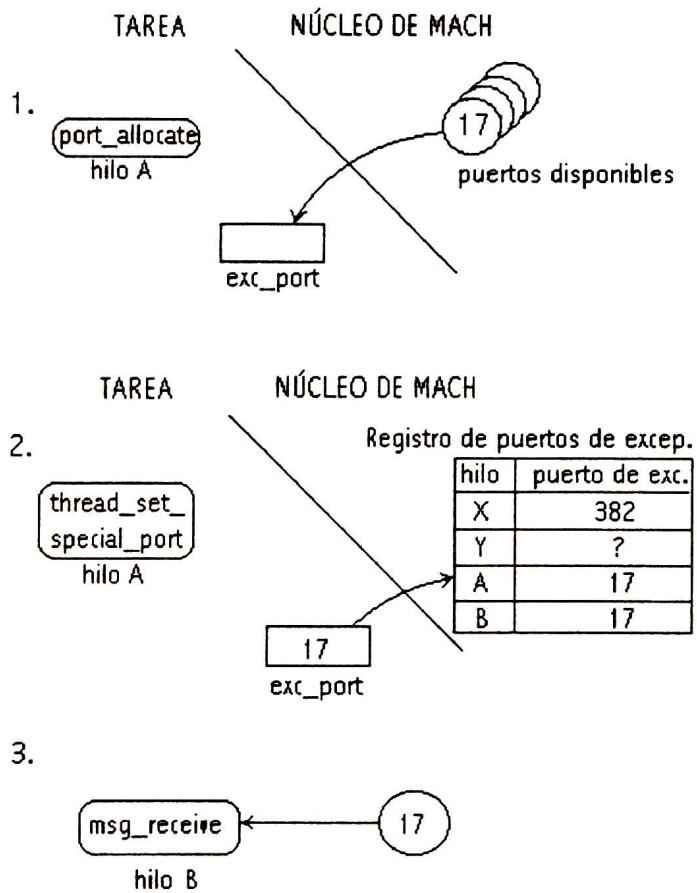


Figura 3.6: Uso del puerto de excepción

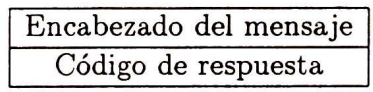


Figura 3.7: Formato del código de respuesta a excepciones

con el que indica si la excepción ha sido o no procesada. El formato de éste código de respuesta se muestra en la figura 3.7

Para el procesamiento de las excepciones es posible el uso de la rutina **exc_server()** para minimizar el trabajo involucrado en el análisis del mensaje de notificación y la construcción del mensaje de respuesta. Esta rutina llama a otra rutina proporcionada por el usuario, **catch_exception_raise()**, para actuar sobre la excepción. La función *exc_server()* entiende el formato del mensaje de notificación, extrae la información relevante (hilo víctima, tipo de excepción, etc.) del mensaje y la proporciona a **catch_exception_raise()**. La rutina **exc_server** coloca el estado de regreso de **catch_exception_raise()** en el mensaje de respuesta regresado al núcleo. Un estado con valor TRUE indica al núcleo que la excepción fue procesada exitosamente y FALSE indicará que la excepción no fue procesada.

3.7 Resumen

Mach ofrece cinco abstracciones de programación básicas en la construcción de un sistema. Estas abstracciones son: la *tarea*, que es uno de los componentes en los que se ha dividido a los procesos UNIX tradicionales. Una tarea contiene todos los recursos (como puertos de comunicación y memoria virtual) para un grupo de entidades cooperativas. La tarea es una colección pasiva de recursos; no se ejecuta en un procesador.

Otro componente de un proceso en Mach es el ambiente de ejecución activo, llamado *hilo*. Cada tarea puede soportar uno o más hilos de ejecución, ejecutándose concurrentemente y compartiendo los recursos de la tarea a la que pertenecen. El estado de ejecución de un hilo consiste de un conjunto de registros, tales como registros de propósito general, un apuntador a la pila, un contador de programa, etc. La tercera abstracción se refiere al canal a través del cual se comunican los hilos unos con otros y se denomina *puerto*. Un puerto es un recurso propiedad de una tarea, por lo tanto, los hilos tienen acceso a un puerto en virtud de pertenecer a dicha tarea. Los programas

cooperativos pueden permitir a hilos de una tarea tener acceso a puertos de otra tarea. Una característica importante de los puertos es que son localidades transparentes, lo cual facilita la distribución de servicios por la red sin modificación a los programas. Los hilos en diferentes tareas se comunican entre ellos mediante el intercambio de mensajes. Un *mensaje* contiene una colección de datos de cualquier tipo.

Mach cuenta con un subsistema de *memoria virtual* que combina facilidad de uso y alto desempeño. La memoria puede ser manipulada rapidamente desde una sola página hasta el espacio de direcciones completo de manera transparente al usuario. Así un programa puede reservar, liberar, proteger y compartir rangos arbitrarios de memoria en cualquier tarea.

Las *excepciones* son interrupciones al procesamiento normal de las instrucciones de un programa causadas por el programa mismo y tienen gran importancia para las aplicaciones que las procesan como parte integral de su operación.

Mach dispone de una gran variedad de llamados al sistema que el programador puede utilizar para manejar las abstracciones mencionadas.

Capítulo 4

El procesador i486

Muchos de los servicios que ofrece un sistema operativo requieren de la disponibilidad del hardware adecuado. Así, existe siempre una relación estrecha entre el sistema operativo y la arquitectura de la computadora sobre la cual trabaja. Este capítulo presenta la arquitectura del i486, el procesador en el que corre el emulador de DOS descrito en capítulos posteriores.

4.1 Organización

El i486 es un procesador CISC (*Complex Instruction Set Computer*) de 32 bits que ofrece capacidades avanzadas de manejo de memoria, compatibilidad con otros procesadores de la familia x86 de Intel, y soporte para ambientes multiprogramados.

La memoria puede manejarse como segmentada, paginada o segmentada-paginada, con o sin manejo de memoria virtual. También contiene una memoria caché en el chip que guarda los datos más recientemente utilizados.

Los segmentos pueden ser tan grandes como 4 Gbytes; las páginas son de 4 Kbytes.

El i486 es compatible con otros procesadores de la familia Intel, seleccionando su modo de operación. En el modo real se comporta como un procesador 8086/88 de alta velocidad con extensiones en su conjunto de instrucciones y registros. Su mayor velocidad se debe no sólo a su mejor diseño y tecnología, sino al uso de una memoria caché dentro del procesador. En el modo protegido es posible seleccionar cualquier tipo de manejo de memoria mencionado antes, y utilizar soporte para sistemas multiprogramados.

Además el i486 ofrece facilidades para depuración, coprocesamiento y multiproce-

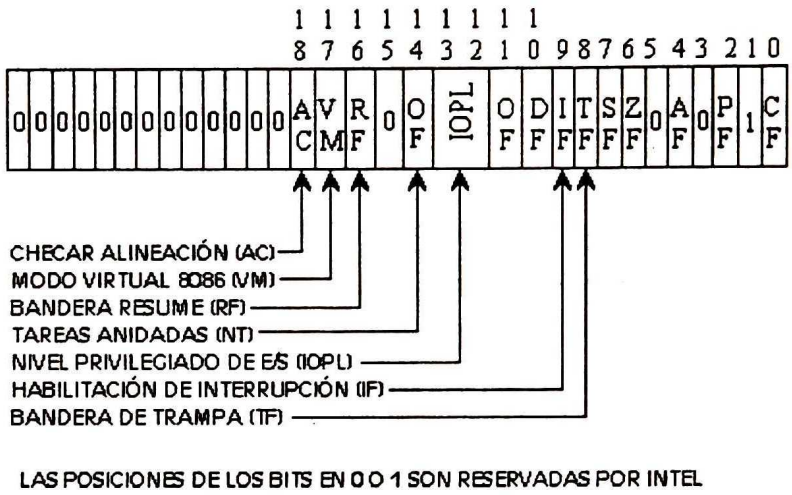


Figura 4.1: Banderas del Sistema i486.

samiento.

4.2 Registros de Control

Las diferentes opciones de manejo de memoria y modo de operación son seleccionados al escribir en diferentes registros de control del i486.

El registro de banderas EFLAGS, por ejemplo, controla la entrada y salida, interrupciones enmascarables, depuración, conmutación entre tareas y el modo virtual-8086. Un programa de aplicación debería ignorar estas banderas y no intentar cambiar su estado [INTEL 90]. En la mayoría de los sistemas, un intento del programa por cambiar el estado de las banderas genera una excepción. Las banderas son mostradas en la Figura 4.1. Una descripción completa de cada bandera se encuentra en [INTEL 90]. Las más significativas a nuestro trabajo son:

AC (Modo de revisión de alineación) La asignación de un 1 en AC y en el bit AM del registro CR0, habilita la revisión de alineación en las referencias a memoria. La interrupción de revisión de alineación puede ser usada para verificar

la alineación de los datos. Esto es útil durante el intercambio de datos con otros procesadores como el microprocesador de 64-bits i860, que requiere que todos los datos sean alineados a frontera de n bits. La interrupción de revisión de alineación también puede ser usada para indicar que algún apuntador tiene una dirección desalineada con el tipo de datos que debe apuntar, eliminando posibles traslapes de los datos.

VM (Modo virtual-8086) Poniendo a 1 la bandera VM se coloca al procesador en modo virtual que es una emulación del ambiente de programación de un procesador 8086.

RF (Bandera de continuación) La bandera RF deshabilita temporalmente las excepciones de depuración; una instrucción puede así ser restaurada después de una excepción sin causar ninguna otra excepción de depuración. La bandera RF no es afectada por la instrucción POPF, pero sí por las instrucciones POPFD e IRET.

NT (Tareas Anidadas) El procesador usa la bandera de tareas anidadas para controlar el encadenamiento de tareas llamadas e interrumpidas. Esta bandera afecta la operación de la instrucción IRET. Es afectada por instrucciones POPF, POPFD e IRET. Las excepciones inesperadas en los programas de aplicación pueden ser originadas por cambios inapropiados al estado de esta bandera.

IF (Habilitación de Interrupción) Coloca al procesador en un modo en el cual responde a solicitudes de interrupciones enmascarables (INTR). Poniendo a cero esta bandera, se deshabilitan las interrupciones enmascarables.

TF (Bandera de Trampas) Pone al procesador en modo de ejecución a pasos para depuración. En este modo, el procesador genera una excepción de depuración después de cada instrucción.

Otros registros de control son el CR0, CR1, CR2 y CR3, los cuales pueden ser leídos por los programas de aplicación para determinar si existe la presencia de un coprocesador numérico, o por el sistema operativo para inicialización y gestión del estado del procesador, por ejemplo. El registro CR0 contiene banderas de control del sistema que controlan los modos de operación o indican los estados del procesador. Un programa no debe intentar cambiar cualquiera de los bits en posiciones reservadas. A continuación se describen algunos de los bits del registro CR0, el resto de ellos son bits reservados:

PG Paginación, bit 31. Este bit en 1, habilita la paginación. Cuando se genera una excepción durante la paginación, el registro CR2 tiene en sus 32 bits la dirección lineal que causó la excepción de fallo de página.

WP Protección de Escritura, bit 16. Cuando está en 1 este bit, las páginas a nivel de usuario se acceden en modo supervisor. Cuando este bit está en 0, las páginas a nivel de usuario solo pueden ser escritas por un proceso supervisor. Esta característica es útil en la implantación del método *copy-on-write* para la creación de un nuevo proceso.

MP Coprocesador Matemático Presente, bit 1. En caso de valer 1, este bit indica la presencia del coprocesador matemático.

PE Habilitación de Protección, bit 0. Poniendo este bit a 1, se habilita la protección a nivel de segmento.

Los registros de depuración brindan capacidades avanzadas de depuración para el procesador i486, incluyendo puntos de ruptura sin modificación de los segmentos de código. Solo los programas ejecutándose en el más alto nivel de privilegio pueden acceder a estos registros (ver Figura 4.2).

4.3 Manejo de Memoria

El procesador i486 ofrece manejo de memoria segmentada, memoria paginada, y memoria segmentada-paginada. Sin embargo, su disponibilidad depende del modo en que esté operando el procesador.

En el **modo real**, el procesador se comporta como una versión de 32 bits del 8086 y solo está disponible el manejo de memoria segmentada con limitaciones.

En el **modo protegido** están disponibles los tres tipos de manejo de memoria. En este modo, el i486 tiene un submodo denominado **modo virtual-8086** (V86). Tanto en este modo como en el modo real, el i486 se comporta como un 8086; la diferencia entre ambos es que el cambio a modo V86 puede realizarse a nivel de tarea. Así, un sistema puede ejecutar concurrentemente (bajo multitarea) tanto programas 8086 en modo V86 como programas i486 en modo protegido. Los modos de operación son vistos en secciones posteriores en este capítulo. En esta sección presentamos los tipos de manejo de memoria.

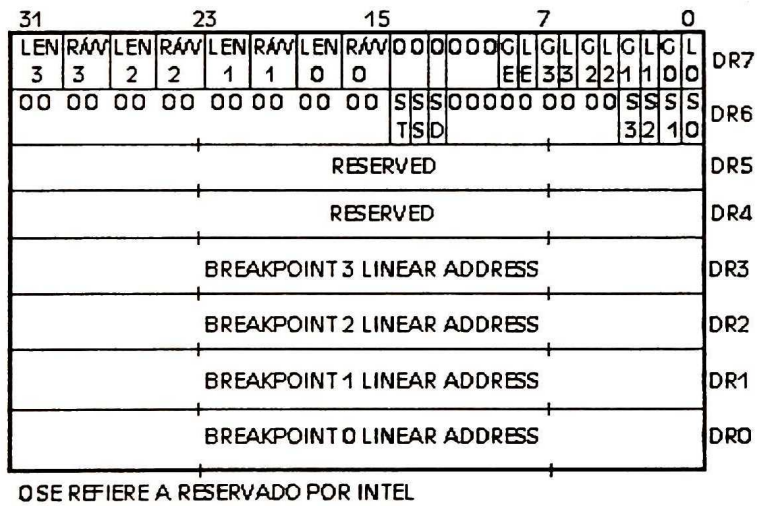


Figura 4.2: Registros de Depuración

4.3.1 Segmentación

La segmentación es una forma de dividir las áreas de memoria y de datos del programa en unidades llamadas segmentos. Estas unidades poseen las siguientes características:

- Están dedicadas a funciones específicas de los programas, tales como código, datos o pila.
- Pueden tener cualquier tamaño, hasta un máximo determinado por el modo de operación del procesador.
- Se aplican a la memoria lógica tal como el programador los ve, no a la memoria física.

El i486 permite que el programa acceda hasta a seis segmentos a la vez a través de los registros: CS, SS, DS, ES, FS y GS.

Segmentación en el Modo Real y el Modo virtual-8086.

Como en un procesador 8086/88, en el modo real y el modo virtual-8086, los segmentos pueden tener una longitud de hasta 64 Kbytes. Un segmento está definido por un número de 16 bits, y su dirección base es ese número multiplicado por 16. Una dirección lógica se compone de un número de segmento y de un desplazamiento (otro número de 16 bits) dentro del segmento. El 8086, y cada modo traduce las direcciones lógicas en direcciones físicas de la siguiente forma:

1. Multiplica el número de segmento por 16.
2. Suma el desplazamiento al resultado.

La traducción es siempre la misma en las máquinas basadas en 8086/8088, independientemente del sistema operativo o de cualquier otro software.

Segmentación en Modo Protegido

En el modo protegido, la segmentación trabaja de modo diferente. Aquí, un registro de segmento contiene un selector, un número de descriptor almacenado en una tabla de descriptores en memoria. El descriptor contiene la dirección base del segmento, el límite y otros atributos:

- Dirección base (32 bits). A la dirección base, el procesador le suma un desplazamiento. Con manejo de memoria segmentada, la dirección obtenida corresponde a una dirección física de la memoria principal. Con manejo de memoria segmentada-paginada, la dirección obtenida es una dirección en un espacio de direcciones de 4 Gbytes, y es traducida de nuevo utilizando tablas de páginas (ver paginación).

- Límite de segmento (20 bits). Define la longitud del segmento.

- Nivel de privilegio de descriptor (2 bits). Indica el privilegio de un segmento de código.

- Bit de segmento presente (P). Un sistema puede utilizar este bit para implementar memoria virtual a nivel de segmento, aunque en la práctica, muchos sistemas implementan la memoria virtual mediante cambios de páginas.

- Bit de segmento accedido (A). El procesador pone a 1 este bit cada vez que se accede al segmento, y el sistema operativo lo revisa y limpia para detectar segmentos más recientemente usados para implementar una política de reemplazo.

Los descriptores de segmentos pueden estar situados en dos tipos de tablas de descriptores: globales (aquellos que aplican a todas las tareas del sistema) y locales (que solo aplican a la tarea en curso). La tabla es elegida por el selector mismo, ver Figura 4.3.

La dirección base lineal de 32 bits de la tabla de descriptores globales está en el registro de la tabla de descriptores globales (GDTR). La dirección base lineal de 32 bits de la tabla de descriptores locales está en el registro de la tabla de descriptores locales (LDTR). Las direcciones base son lineales, no relativas a segmento y corresponden a direcciones de memoria principal. Los registros GDTR y LDTR contienen además 16 bits que especifican el tamaño de la tabla respectiva.

Un selector debe especificar una tabla y proporcionar además un índice a un descriptor dentro de la misma. El índice debe multiplicarse por 8 antes de que el procesador lo utilice para acceder a la tabla. Los selectores no ocupan los registros de segmento completamente. Los registros poseen una parte visible que contiene al selector y una parte invisible que contiene al descriptor correspondiente: la dirección base, el límite, el tipo y otros atributos del segmento que se obtienen de la tabla de descriptores. Las partes ocultas son cargadas por el procesador cuando este carga el selector. Así se ahorra tiempo cada vez que el procesador utiliza el selector. El

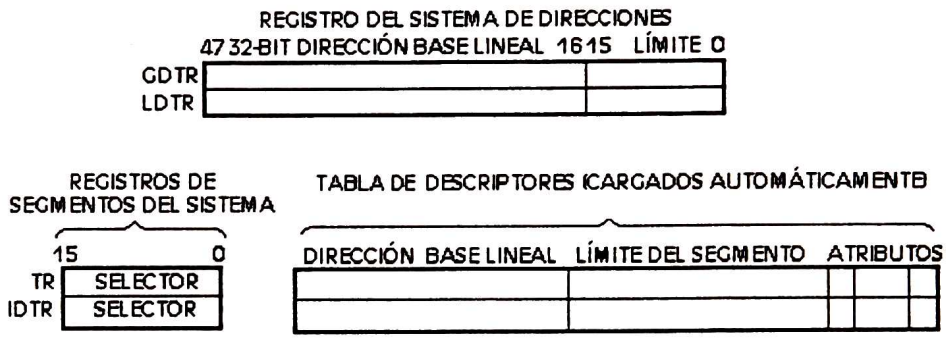


Figura 4.3: Registros de Manejo de Memoria

procesador no necesita leer el descriptor en memoria para calcular la dirección lineal o para realizar comprobaciones de validez, ya que la información está disponible en la parte oculta del registro de segmento.

4.3.2 Paginación

Contrario a la segmentación, la paginación es opcional en el i486. El bit PG (bit 31 del registro de control 0) determina si está activa. El sistema operativo solo puede poner a uno el bit PG en el modo protegido. El modo real no admite paginación pero el modo V86 puede admitirla.

Traducción de Direcciones Lineales a Direcciones Físicas

La traducción bajo paginación involucra dos niveles de tablas:

1. Un directorio que contiene las direcciones físicas base de tablas de páginas. Los directorios permiten que el sistema tenga varias tablas de páginas, para mantener separadas a las tareas y/o ahorrar memoria física.

2. Tablas de páginas, que contienen las direcciones físicas base de las páginas de código o datos.

Para calcular una dirección física, el procesador divide una dirección lineal en tres campos, como se muestra en la figura 4.4. Los campos refieren:

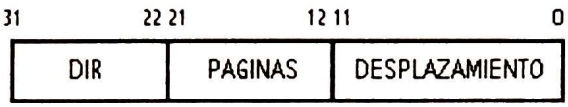


Figura 4.4: Formato de una dirección lineal

- Al índice del directorio, bits 22 al 31. El procesador lo utiliza para seleccionar una entrada en el directorio, la cual contiene una dirección física base de una tabla de páginas. El campo de 10 bits, implica que el directorio puede tener hasta 1K (1024) entradas para direccionar tablas de páginas.

- Al índice de la tabla de páginas, bits 12 al 21. El procesador los utiliza para seleccionar una entrada en la tabla de páginas la cual contiene la dirección física base de una página de datos. Como es un campo de 10 bits, la tabla de páginas puede contener hasta 1024 entradas para direccionar páginas de datos.

- Los bits 0 al 11 corresponden al desplazamiento dentro de la página. El procesador los suma al contenido de la entrada de la tabla de páginas.

Tablas de Páginas

Además de una dirección física, cada entrada en las tablas de páginas tienen también los siguientes campos:

- Bit D (página sucia). El procesador pone el bit D a 1 cada vez que escribe en la página correspondiente. D indica así al sistema operativo si debe escribir la página al disco cuando sea retirada de memoria. Si el procesador no ha escrito nunca en una página, no hace falta almacenarla de nuevo en disco.

- Bit A (página accedida). El procesador pone A a 1 cada vez que accede a la página. El sistema operativo puede utilizar el bit A para identificar las páginas que no se han accedido ultimamente y son candidatas para cambiarlas por otras en disco.

- Bit U/S (usuario/supervisor). Este es un bit de protección que determina si la página es accesible para programas que corran con privilegio de usuario o unicamente con privilegio de supervisor.

- Bit R/W (lectura/escritura). Este bit determina si la página es sólo para lectura (0) o para lectura y escritura (1) en el nivel de usuario.

- Bit P (presente). P vale 1 si la página o tabla de páginas está en memoria física. El procesador indica un fallo de página si intenta utilizar una entrada en la que P = 0.

El directorio y la tabla de páginas en curso deben permanecer siempre en memoria, así como la tabla de descriptores globales y las funciones de servicio de excepciones e interrupciones básicas.

El Caché de Entradas de Tablas de Páginas

La paginación podría llevar mucho tiempo si la traducción de direcciones requiriera que el procesador accediese al directorio de páginas y a la tabla de páginas en la memoria; para evitar esto, el procesador guarda los datos de la tabla de páginas más recientemente utilizados en una memoria caché, interior al chip, lo cual hace que sólo tenga que acceder a memoria si la información de una determinada página no está en la memoria caché.

Este caché, denominado *Translation Lookaside Buffer*, o **TLB**, contiene 32 entradas, cada una de ellas consiste de un campo de etiqueta de 24 bits y un campo de datos de 20 bits. El campo de etiqueta contiene los 20 bits más significativos de la dirección lineal, el bit de validez, y los tres bits de atributos (presente, lectura/escritura, usuario/supervisor). El campo de datos contiene los 20 bits más significativos de la dirección física.

Con 32 entradas, el TLB puede proporcionar acceso a 32 páginas o 128 Kbytes de memoria. Si un programa utiliza un área de memoria de tamaño menor de, o igual a 128 Kb (denominada ventana de trabajo), podrá ejecutarse sin necesidad de que el TLB se actualice. Los fallos de página causarán inicialmente que el procesador cargue en memoria la ventana de trabajo. El procesador la utilizará entonces de modo continuo sin necesidad de accesos extra a memoria para realizar la paginación.

4.4 Excepciones e Interrupciones

Las interrupciones y excepciones son transferencias de la ejecución a una tarea o un procedimiento llamado manejador. Las interrupciones ocurren durante la ejecución de un programa en respuesta a señales de hardware o para invocar un servicio del sistema operativo. Las excepciones ocurren como resultado de la ejecución anómala de las instrucciones de un programa (por ejemplo, la división por cero). Usualmente, los servicios de interrupciones y excepciones son realizados de manera transparente a los programas de aplicación.

Hay dos fuentes de interrupciones y dos de excepciones:

1. Interrupciones

   - Las interrupciones enmascarables son recibidas por la entrada INTR del procesador i486. Estas interrupciones corresponden a señales de dispositivos y no se atienden a menos que la bandera (IF) esté habilitada.
   - Las interrupciones no enmascarables son recibidas por la entrada NMI del procesador y se utilizan para solicitar servicios al sistema operativo. El procesador no proporciona un mecanismo para prevenir interrupciones no enmascarables.

2. Excepciones

   - Excepciones detectadas por el procesador. Son clasificadas como *fallos*, *trampas* y *aborts*. Los fallos son la categoría menos grave, generados por ejemplo por puntos de ruptura, o divisiones por cero. El procesador los detecta antes de terminar de ejecutar la instrucción. La dirección de retorno del manejador del fallo apunta a la instrucción que lo generó. Una trampa, es una excepción que es reportada al inicio de la instrucción siguiente a la que generó la excepión. Un abort es una excepción que no siempre reporta la dirección de la instrucción que causó la excepción. Los aborts son usados para reportar errores severos, tales como errores de hardware y valores ilegales o inconsistentes en las tablas.
   - Excepciones programadas. Las instrucciones INTO, INT 3, INT n, y BOUND pueden generar excepciones. Estas instrucciones a menudo son llamadas "interrupciones de software", pero el procesador las maneja como excepciones.

4.4.1 Atención a Excepciones e Interrupciones

El procesador asocia un número de identificación a cada tipo de interrupción o excepción. Este número es conocido como **vector**.

Las interrupciones y las excepciones tienen asignados vectores en el rango de 0 a 31. No todos esos vectores son utilizados por el procesador, algunos son reservados para posibles usos futuros. Los vectores para interrupciones enmascarables son determinados por hardware. Los controladores externos de interrupciones ponen el vector en el bus del procesador i486 durante su ciclo de reconocimiento de interrupción. Cualquier vector en el rango de 32 a 255 puede ser usado.

La tabla de descriptores de interrupciones (IDT) asocia cada excepción o vector de interrupción con un descriptor al proceso o tarea que dará servicio al evento asociado. El IDT es un arreglo de descriptores de 8 bytes cada uno. Para formar un índice en el IDT, el procesador multiplica el número de excepción o interrupción por ocho, el número de bytes de cada entrada.

La IDT puede residir en cualquier parte de la memoria física. Como se muestra en la Figura 4.5, el procesador localiza la IDT usando el registro IDTR. Este registro guarda la dirección base (de 32 bits) y el tamaño (de 16 bits) de la IDT. Si un vector hace referencia a un descriptor más allá del límite de la IDT, el procesador entra en modo *shutdown*, es decir, detiene la ejecución de instrucciones hasta que una interrupción NMI sea recibida o se invoque una inicialización. El procesador genera un ciclo de bus especial para indicar que ha entrado en modo *shutdown*.

4.5 Control de Tareas

Una *tarea* es un programa junto con los datos que usa y el espacio de memoria que se le ha asignado. El i486 ofrece las siguientes características de manejo de tareas:

- Segmentos de estados de tareas
- Descriptores de segmentos de estados de tareas
- Registros de tareas
- Descriptores de puertos de tareas
- El indicador NT (Tarea Anidada) en el registro de indicadores extendido para uso en el regreso de una tarea que ha sido llamada por otras. NT es el bit 14 de EFLAGS.

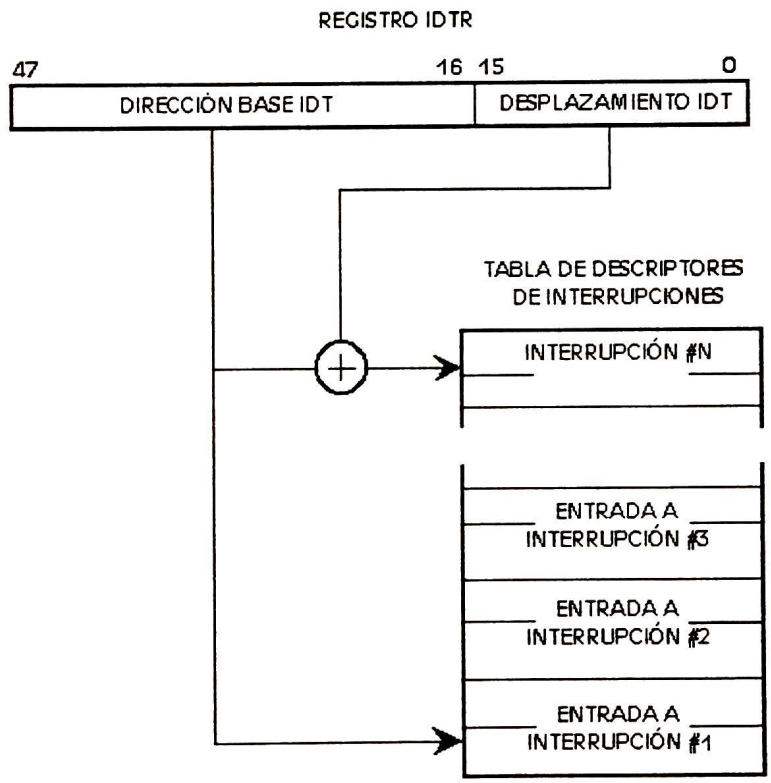


Figura 4.5: Registro IDTR localizando a la IDT en Memoria

La gestión de tareas se aplica en el modo protegido (incluido el modo V86) pero no en el modo real. Las tareas V86 deben tener el bit VM de EFLAGS puesto a uno y deben correr en el nivel de privilegio 3. Un monitor de máquina virtual maneja excepciones especiales V86 tal como la INT 21h para entrar en MS-DOS.

4.5.1 Segmentos de Estados de Tarea (TSS)

Los segmentos de estado de tareas especifican el estado actual de las tareas en un formato predefinido. Los TSSs permiten al sistema operativo cambiar fácilmente el estado de las tareas. El sistema operativo puede activarlas, suspenderlas o terminarlas manipulando su segmento de estado. El segmento de estado de la tarea activa es apuntado por un registro de tarea.

Un segmento de estado de tarea contiene información sobre la globalidad del entorno el estado actual de una tarea. Cargar el segmento de estado de la tarea, no solamente sustituye al método de iniciar los registros que arranca una tarea, sino también permite devolver a una tarea suspendida su estado antiguo antes de rearrancarla.

4.6 Modos de Operación

El i486 tiene dos modos de operación: el modo real y el modo protegido, y dentro de este último un submodo llamado Virtual-8086.

En el modo real se comporta como un procesador 8086 de alta velocidad con extensiones a su conjunto de instrucciones y registros. Es decir, el valor en un registro de segmento no representa un selector que refiere a un descriptor, sino que forma direcciones lineales como lo haría un procesador 8086: recorriendo el selector cuatro bits hacia la izquierda para formar una dirección base de 20 bits. La dirección efectiva, es extendida con cuatro bits en cero en las posiciones correspondientes a los bits superiores y sumada a la dirección base para crear una dirección lineal.

El i486 inicia su operación en modo real. Para dejarlo y entrar al modo protegido, se debe mover un 1 a la bandera PE del registro CR0.

El modo protegido ofrece todo el potencial del i486 para manejo de memoria, multiprogramación e incluso la emulación concurrente del procesador 8086. Como ya vimos el manejo de memoria, nos concentraremos en el modo virtual-8086.

4.6.1 Modo Virtual-8086

El procesador corre en modo virtual-8086 cuando el bit VM (máquina virtual) del registro EFLAGS está en 1. El procesador prueba esta bandera bajo dos condiciones generales: durante la carga de los registros de segmentos, para saber si usar traducción de direcciones estilo 8086; durante la decodificación de instrucciones, para determinar qué instrucciones no son soportadas (como en el modo real).

El procesador i486 soporta la ejecución concurrente de uno o más programas en el modo virtual-8086 dentro del modo protegido. No solo pueden haber múltiples programas virtuales 8086, sino tambien otros corriendo en modo protegido.

En realidad, un programa a correr en modo virtual-8086 requiere de otros dos componentes: un programa monitor a correr en modo protegido, y un conjunto de procedimientos que realizan los servicios que el programa virtual 8086 pueda solicitar. A estos tres componentes nos referimos como tarea virtual 8086. El procesador entra en modo virtual-8086 para correr el programa 8086 y regresa a modo protegido a correr el monitor o alguna otra tarea del procesador.

El monitor consiste mayormente de procedimientos de inicialización y manejo de excepciones. Las direcciones lineales arriba de 10FFEFH, están disponibles para el monitor virtual-8086, los procedimientos de servicio y otro software del sistema.

En general, hay dos opciones para la implantación de los procedimientos de servicio:

1. Pueden correr como parte del programa 8086.

2. Pueden ser implantados o emulados en el monitor.

4.6.2 Paginación para Tareas Virtuales 8086

La paginación no es necesaria para una sola tarea virtual 8086, pero es útil o necesaria en los siguientes casos:

- En la creación de múltiples tareas virtuales 8086. Cada tarea debe mapear el megabyte inferior de direcciones lineales a diferentes localidades físicas.

- Creación de un espacio de direcciones virtuales mayor que el espacio de direcciones físicas.

- La compartición de los procedimientos de servicio o código ROM, común a varios programas corriendo en multitareas.

- Redireccionamiento o referencias a dispositivos de E/S mapeados en memoria.

4.6.3 Protección dentro de una Tarea Virtual 8086

Para proteger el software de una tarea virtual 8086, los diseñadores de software pueden seguir alguna de estas recomendaciones:

- Reservar el primer megabyte (más 64 kbytes) del espacio lineal de direcciones de cada tarea para el programa 8086. Un programa 8086 no puede generar direcciones fuera de este rango.
- Usar el bit U/S de la tabla de entradas de páginas para proteger al monitor. Cuando el procesador está en modo virtual, el nivel de privilegio (CPL) es 3 (el menos privilegiado); así, un programa 8086 solo tiene privilegios de usuario. Si las páginas del monitor de la máquina virtual tienen privilegios de supervisor, no podrán ser accedidas por el programa 8086, solo por el monitor, que corre en el nivel 0, o el más privilegiado.

4.6.4 Entrando y Dejando el Modo Virtual-8086

La figura 4.6 muestra la manera de conmutarse a un programa 8086. El modo virtual-8086 se accede poniendo a uno la bandera VM. Hay dos maneras de hacer esto:

1. La tarea carga la imagen del registro EFLAGS del nuevo TSS. El TSS de la nueva tarea debe ser un TSS del i486, no del 80286, debido a que éste no carga la parte alta del registro EFLAGS que contiene la bandera VM. Un valor de uno en el bit VM, indica que la nueva tarea está ejecutando instrucciones 8086.
2. Una instrucción IRET de un procedimiento i486 carga el registro EFLAGS del stack. Un valor de uno en la bandera VM indica que el procedimiento al cual se regresa el control es un procedimiento 8086. Cuando la instrucción IRET es ejecutada, CPL debe ser 0, de otra manera el procesador no cambia el estado de la bandera VM.

Cuando se usa una conmutación de tarea para entrar a modo virtual-8086, los registros de segmento son cargados desde un TSS. Pero, cuando se usa una instrucción IRET para poner 1 en la bandera VM, los registros de segmento deben haber sido cargados con los valores adecuados para el modo virtual-8086 durante el modo protegido.

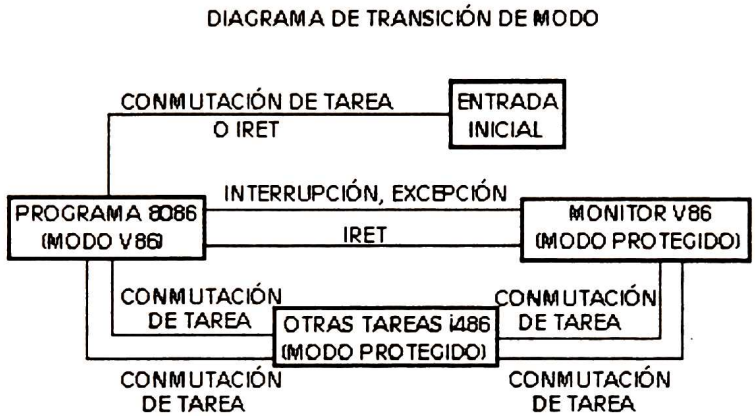


Figura 4.6: Entrando y dejando el modo virtual-8086

El procesador deja el modo virtual-8086 cuando ocurre una interrupción o excepción. Existen dos casos:

1. La interrupción o excepción causa una conmutación de tarea. Una conmutación de una tarea virtual 8086 a cualquier otra tarea, carga el registro EFLAGS del TSS de la nueva tarea. Si el nuevo TSS es un i486 y la bandera VM en el nuevo contenido del registro EFLAGS está en 0, el procesador limpia la bandera VM del registro EFLAGS, carga los registros de segmento del nuevo TSS usando formación de direcciones estilo i486 y empieza la ejecución de instrucciones de la nueva tarea en modo protegido i486.

2. Interrupciones o excepciones llaman a un procedimiento con nivel de privilegio 0 (el más privilegiado). El procesador almacena el contenido actual del registro EFLAGS en la pila y limpia la bandera VM. El manejador de interrupción o excepción, sin embargo, corre como código i486 en modo protegido. Si una interrupción o excepción llama a un procedimiento en un segmento en un nivel de privilegio distinto a 0, el procesador genera una excepción de protección general, el código de error es el selector del segmento de código que intentó el llamado.

El software del sistema no cambia el estado de la bandera VM directamente, cambia el estado en la imagen del registro EFLAGS almacenado en el stack o el TSS. El monitor virtual 8086, pone a uno la bandera en la imagen de EFLAGS en el stack o el TSS, durante la primera creación de una tarea virtual 8086. Los manejadores de excepciones o interrupciones pueden examinar la bandera VM en el stack, si el procedimiento interrumpido estaba corriendo en modo virtual-8086, el manejador puede necesitar llamar al monitor virtual 8086.

4.6.5 Configuración de los Vectores de Interrupción

Debido a que un programa 8086 es diseñado para correr en un procesador 8086, una tarea virtual 8086 tendrá una tabla de interrupciones al estilo 8086 que inicia en la dirección lineal 0. Sin embargo, el procesador i486 no usa esta tabla directamente; para todas las excepciones e interrupciones que ocurren en modo virtual-8086, el procesador llama manejadores a través de la IDT.

Las interrupciones y excepciones que llaman una trampa o interrupción, usan un nivel de privilegio 0. Los contenidos de los registros de segmento son almacenados en la pila con este nivel de privilegio, después de esto, el procesador limpia los registros de segmento antes de que el procedimiento manejador inicie su ejecución, esto permite al manejador de interrupción guardar y restaurar los registros DS, ES, FS y GS

Un manejador de interrupción pasa el control al monitor virtual 8086 si la bandera VM está habilitada en la imagen del registro EFLAGS guardado en la pila. El monitor virtual 8086 puede:

- Manejar la interrupción dentro del monitor virtual 8086.
- Llamar el manejador de interrupción del programa 8086.

El regreso de una interrupción o excepción al programa 8086 involucra los siguientes pasos:

1. Recuperar de la pila el estado almacenado del programa 8086.
2. Cambiar la liga de retorno en el nivel de privilegio 0 de la pila (más privilegiado) para apuntar al procedimiento original interrumpido, procedimiento con nivel de privilegio 3.
3. Ejecutar una instrucción IRET para pasar el control al manejador.

4. Cuando la instrucción IRET del manejador con nivel de privilegio 3, llama nuevamente al monitor 8086, restaura la liga de retorno sobre el nivel de privilegio 0 de la pila, para apuntar al procedimiento original interrumpido.

5. Regresar el control al procedimiento interrumpido.

4.7 Resumen

El i486 es un procesador de 32 bits con capacidades avanzadas de manejo de memoria, compatibilidad con otros procesadores de la familia x86 de Intel, y soporte para ambientes multiprogramados. La memoria puede manejarse como segmentada, paginada o segmentada-paginada, con o sin manejo de memoria virtual. Mediante la selección de su modo de operación, es compatible con otros procesadores de la familia Intel. En el modo real se comporta como un procesador 8086/88 de alta velocidad con extensiones a su conjunto de instrucciones y registros. Su mayor velocidad se debe en gran medida al uso de una memoria caché dentro del procesador. En el modo protegido es posible seleccionar cualquiera de los tipos de manejo de memoria antes mencionados, y utilizar soporte para sistemas multiprogramados. Las opciones de manejo de memoria y modos de operación se seleccionan mediante su conjunto de registros de control.

La segmentación da una forma de dividir las áreas de memoria de los programas en unidades llamadas segmentos que están dedicadas a funciones específicas de los programas, pueden tener cualquier tamaño y se aplican a la memoria lógica, no a la física. El i486 permite el acceso hasta a seis segmentos a la vez a través de los registros CS, SS, DS, ES, FS y GS. Al contrario que la segmentación, la paginación es opcional en el i486. La traducción bajo paginación involucra dos tablas: un directorio con las direcciones físicas base de las tablas de páginas y las tablas de páginas, que contienen las direcciones físicas base de las páginas de código o datos. El modo real no admite paginación pero el modo V86 puede admitirla.

Interrupciones y excepciones son transferencias de la ejecución de una tarea a un procedimiento manejador. Las primeras ocurren en respuesta a señales de hardware o para invocar un servicio del sistema operativo. Las excepciones son resultado de la ejecución anómala de las instrucciones de un programa. Los servicios de interrupciones y excepciones son realizados de manera transparente a los programas de aplicación. La conmutación entre los diferentes modos de operación del i486 es en base a los mecanismos que soportan el manejo de interrupciones y excepciones.

Capítulo 5

Emulación Prototipo del Sistema de Archivos DOS

5.1 Introducción.

En el capítulo anterior se describió la manera general de correr un programa 8086 en el modo virtual-8086. Además del programa 8086, se requiere un programa monitor y los procedimientos que realizarán los servicios que solicite el programa 8086. El programa monitor corre en modo protegido y se encarga de la inicialización y manejo de excepciones. La inicialización consiste en la creación de un hilo que se ejecuta en modo virtual del procesador. Este modo de ejecución se logra mediante la manipulación de los registros del procesador y el bit VM de las banderas. El manejo de excepciones consiste en canalizar los servicios que solicite el programa 8086 a los procedimientos que los realizarán.

Los procedimientos que realizan los servicios que pueda solicitar el programa 8086 pueden implantarse como parte del programa 8086 o como parte del programa monitor. La ventaja de ser parte del programa 8086 es que no es necesario cuidar los detalles de implementación de estos servicios para que sean reentrantes; sin embargo el tamaño de los programas es mayor al tener que ligar los servicios a cada uno de ellos. Si tales procedimientos se implantan como parte del programa monitor, solo se necesita una copia para varios programas 8086, pero debe garantizarse que estos sean reentrantes.

Además de los aspectos anteriores, el diseño del programa monitor y de los procedimientos de servicio debe tomar en cuenta, respectivamente la manera en que los

servicios son solicitados por el programa 8086 y la funcionalidad que estos esperan de tales procedimientos.

Todos estos aspectos han sido considerados para diseñar e implantar una emulación prototipo del sistema de archivos DOS dentro del sistema operativo NeXTSTEP sobre el procesador 80846. Nuestro programa monitor es presentado en el capítulo 6. En este capítulo presentamos el diseño de nuestra emulación del sistema de archivos DOS y de los procedimientos de servicio para utilizarlo. Para entender nuestra emulación es necesasario conocer la estructura del sistema de archivos DOS y por esta razón iniciamos con una descripción del mismo.

Posteriormente presentamos nuestra emulación de un disco duro virtual con los componentes de todo disco con formato DOS. Finalmente, presentamos la programación de servicios básicos de manipulación de archivos a través de la interrupción 21h, dentro de los cuales se incluyó: creación, lectura y escritura y búsqueda de archivos en el directorio.

5.2 Estructura del Sistema de Archivos DOS

Las capas del sistema de archivos DOS incluyen las siguientes partes [FORIN 93]: la sección inicial del disco contiene el **registro de arranque**, formado por un número variable de sectores. Esta sección es seguida por una o más copias de la **Tabla de Asignación de Archivos** (FAT, del inglés *File Allocation Table*), que a su vez es seguida por las entradas del **directorio** raíz. El resto del disco está dividido en *clusters*, grupos de sectores dinámicamente reservados para almacenar datos de archivos. La figura 5.1 muestra gráficamente estas capas.

El registro de arranque inicia en el primer sector del disco, y contiene el código de arranque del sistema (*bootstrap*), así como información sobre la geometría del disco y posiblemente información del particionamiento. También contiene la localización, tamaño y número de FAT's, el tamaño del directorio raíz y el tamaño de los sectores físicos y los *clusters*.

La FAT es una tabla de tamaño fijo que describe el estado de asignación de los *clusters* del disco. Sus entradas son de 12 o 16 bits de longitud, dependiendo del número total de *clusters* en la partición del disco. Un disco flexible tiene una FAT con entradas de 12 bits, un disco duro con entradas de 16 bits. Las primeras dos entradas de la FAT son reservadas. DOS usa una sola copia de la FAT en discos flexibles y dos copias en discos duros para proteger los datos de posibles corrupciones en el disco.

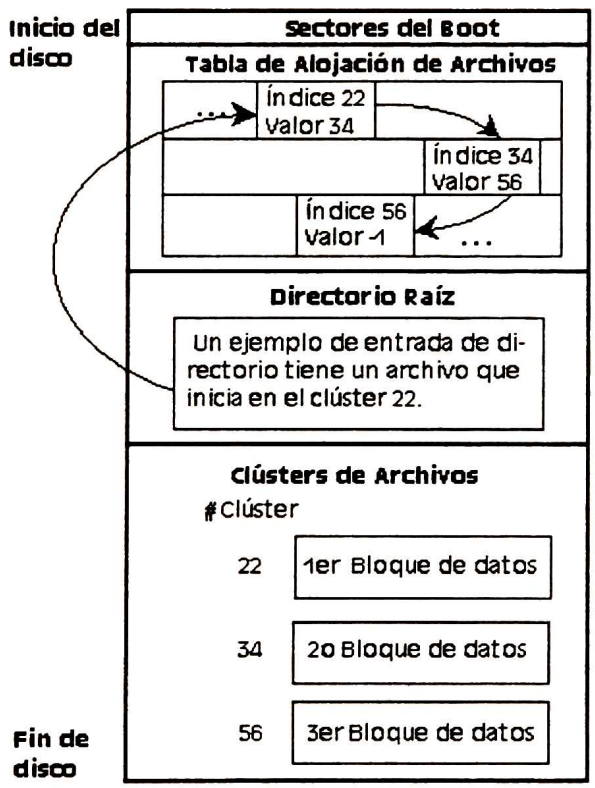


Figura 5.1: Organización del Sistema de Archivos de DOS

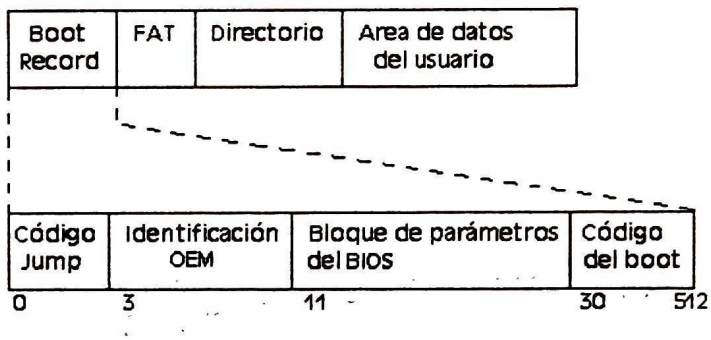


Figura 5.2: Estructura del Boot DOS

Para determinar qué bloques del disco están reservados para un archivo (o directorio) se consulta la respectiva entrada del archivo en el directorio. Una entrada de directorio contiene el número de *cluster* inicial del archivo, es decir, la localización del primer bloque de datos del archivo. Para encontrar el siguiente bloque, usamos el número de *cluster* como un índice en la FAT. El valor en tal entrada es el número de *cluster* del segundo bloque, y así sucesivamente. El último bloque del archivo tiene una entrada en la FAT con un valor de 0FFF8H. Si un *cluster* no está asignado, su entrada en la FAT es cero. Existen otros valores especiales para indicar *clusters* reservados (bloques dañados, por ejemplo).

Los *clusters* son bloques lógicos múltiplos de sectores físicos. El tamaño de un sector físico es típicamente de 512 bytes en discos flexibles pequeños y 2048 bytes en discos duros. Actualmente, el tamaño del *cluster* se define como una potencia de dos múltiplo del tamaño del sector físico; el límite superior soportado por DOS es de 64 KB.

5.2.1 Registro de Arranque

Como ya se había descrito de manera breve, el registro del *boot*, o de arranque, contiene dos partes: la primera de ellas corresponde al programa que carga el sistema DOS a la memoria, la segunda parte contiene la información que describe la organización del disco. Algunos de los componentes de este registro constituyen lo que se conoce también como Bloque de Parámetros del BIOS [JAMSA 94], ver Figura 5.2. A continuación se describen los componentes del registro de arranque.

Instrucción JMP. La primera entrada del registro de arranque es una instrucción JMP. El BIOS carga el registro de arranque en la memoria y después transfiere el control al programa que está dentro de él. En realidad transfiere el control del primer byte del registro de arranque, que contiene la instrucción JMP, al cargador de DOS real (código del boot en la Figura 5.2).

Nombre OEM (identificación). El campo OEM del registro de arranque es un campo de 8 bytes, dentro del cual el fabricante del disco puede colocar su identificación.

Los siguientes campos constituyen el Bloque de Parámetros del BIOS:

Bytes por sector. El campo bytes por sector del registro de arranque describe el tamaño de cada sector en bytes. La mayor parte de los discos utilizan sectores de 512 bytes.

Sectores por *cluster*. Para una mayor eficiencia, DOS asigna espacio en el disco a los archivos en grupos de sectores contiguos denominados *clusters*. Este campo, especifica el número de sectores que contiene cada *cluster* del disco. El número es siempre una potencia de 2.

Número de sectores reservados. Este campo especifica el número de sectores del disco que están reservados para un uso específico. Generalmente este valor es 1, siendo el sector de arranque el único sector de disco reservado.

Número de tablas de asignación de archivos. DOS se mantiene informado de dónde residen los bloques de datos de archivos en el disco utilizando la tabla de asignación de archivos (FAT). En general, la tabla de asignación de archivos sirve como un mapa de ligas que se utiliza para localizar los archivos. Si una tabla de asignación de archivos llega a dañarse, posiblemente por un error del disco, DOS no podrá ubicar sus archivos en el disco. Para reducir la posibilidad de este error normalmente se colocan dos copias de la tabla FAT en el disco. Si la primera tabla llegara a dañarse, el sistema puede utilizar la segunda. Este campo especifica el número de tablas de asignación de archivos contenidas en el disco.

Número de entradas del directorio raíz. Un disco con formato DOS tiene reservado un número específico de sectores para almacenar el directorio raíz del disco. Cada entrada del directorio raíz ocupa 32 bytes. Este campo especifica el número de entradas para las que DOS ha reservado espacio.

Total de sectores del disco. Este campo especifica el número total de sectores del disco.

Descriptor de soportes. Este campo contiene un valor en bytes que identifica el tipo de disco, el valor correspondiente al disco duro es de 0F8H.

Sectores de la Tabla de asignación de archivos. Especifica la cantidad de espacio de disco en sectores que consume cada tabla de asignación de archivos.

Sectores por pista. Para almacenar la información en el disco, éste se encuentra dividido en varias pistas concéntricas, cada una de estas a su vez está dividida en sectores; este campo especifica cuantos de estos sectores contiene cada pista.

Número de cabezas. Indica el número de cabezas de disco que soporta la unidad; por ejemplo, una unidad de disco flexible puede soportar una o dos cabezas, y un disco duro cuatro cabezas o más, dependiendo del tamaño.

Número de sectores ocultos. Todo disco duro inicia con un grupo de sectores que contienen información de las particiones de disco, denominada registro de arranque maestro (*boot master*). Este campo describe el número de sectores de disco ocultos para información de la tabla de particiones.

5.2.2 Tabla de Asignación de Archivos

Como se ha mencionado, DOS emplea el valor 0 para indicar en la tabla de asignación de archivos que un *cluster* está disponible y el valor FFF8H para indicar que se trata del último *cluster* de una cadena de *clusters*. También emplea el valor FFF7H para indicar que se trata de un *cluster* marcado como no utilizable. La tabla 5.1 muestra los posibles valores de las entradas de una FAT. La figura 5.3, muestra las entradas de la FAT correspondientes a un archivo que está en los *clusters* 4, 5, 8 y 9.

Dependiendo del tamaño del disco, DOS puede emplear entradas de 16 o 12 bits. En la tabla 5.2 se muestran los tamaños de las entradas según los tamaños más comunes de disco.

5.2.3 Directorio

El directorio raíz es un área de sectores reservada para almacenar la información relativa a los archivos almacenados en el directorio principal del disco. Las entradas

Entrada	Siguiente entrada
2	3
3	FFF8
4	5
5	8
6	0
7	0
8	9
9	FFF8
⋮	⋮

Figura 5.3: Entradas de la FAT para un archivo de 4 bloques.

Valor de la entrada	**Significado**
00H	Entrada disponible
FFF7H	Entrada dañada
FFFFH	Entrada reservada
FFF8H	Último bloque de la cadena de un archivo
No. de entrada	Siguiente bloque del archivo

Tabla 5.1: Posibles valores de las entradas de la FAT

Tamaño del disco	Tipo de entrada en la FAT
160Kb	12 bits
180Kb	12 bits
320Kb	12 bits
360Kb	12 bits
720Kb	12 bits
1.2Mb	12 bits
1.44Mb	12 bits
2.88Mb	12 bits
Disco duro	12 bits si tiene menos de 4096 *clusters*
	16 bits si tiene más de 4096 *clusters*

Tabla 5.2: Tamaños de las entradas de la FAT

de directorio tienen un tamaño fijo de 32 bytes. Cada entrada contiene el nombre y extensión, un byte de atributos (que indica, entre otras cosas, si es un directorio o un archivo) una estampa de tiempo, el tamaño del archivo (cero para los directorios), y el número inicial de *cluster*. Los dos bits superiores del byte de atributos, también son reservados o indefinidos. En las versiones iniciales de DOS, el directorio raíz era unico y no podía crecer. Las últimas versiones de DOS son compatibles con las anteriores y también tienen un directorio raíz fijo y predefinido contiguamente. Otros directorios pueden crecer o truncarse como sea necesario. Algunas entradas de directorios son especiales, e identificadas como tales por el byte de atributos. La propia imagen de DOS, por ejemplo, está marcada como invisible y de solamente lectura. La entrada de etiqueta (*label*) contiene información sobre el tiempo de creación del sistema de archivos y el nombre del dispositivo. Existe solo una entrada de tal etiqueta en el directorio raíz. Existen 10 bytes más en cada entrada reservados para especificaciones del sistema de archivos.

Los campos que constituyen una entrada de directorio, pueden tener el siguiente formato:

```
struct DirectoryEntry {
        char filename[8];
        char extension[3];
        char atributes;
        char reserved[10];
```

```
        unsigned time_stamp;
        unsigned date_stamp;
        unsigned starting_cluster;
        long file_size;
}
```

5.3 Organización de la Emulación

En esta sección presentamos la organización de nuestra emulación del sistema de archivos DOS dentro del sistema operativo NeXTSTEP sobre el procesador 80486. NeXTSTEP está construido sobre el micronúcleo Mach (descrito en el capítulo 3) y hemos utilizado algunas de sus facilidades en el diseño de nuestra emulación.

Los archivos mapeados en memoria son uno de los beneficios del sistema de memoria virtual de Mach. Esta técnica consiste en que la totalidad o una parte de un archivo puede estar mapeado en una sección de memoria virtual. Posterior al mapeo, una referencia a una posición dentro de esta sección es equivalente a una referencia a la misma posición en el archivo físico. Si tal porción del archivo no se encuentra actualmente en memoria, ocurre una falla de página que indica al núcleo leer la sección requerida del archivo en la memoria física. Desde el punto de vista del proceso, el archivo entero se encuentra en la memoria principal a la vez gracias al manejo de memoria virtual.

Para la emulación del sistema de archivos DOS se creó un archivo UNIX de 3 megabytes. Este archivo contiene las estructuras mencionadas en las secciones previas, es decir, el registro del boot, con su bloque de parámetros del BIOS, una FAT de 6144 entradas para mapear cada uno de los *clusters* del disco virtual, un arreglo de 512 entradas de directorio para el directorio raíz y un área de datos del usuario, limitada a 6144 sectores de 512 bytes cada uno. Las siguientes definiciones conforman la información contenida en el archivo del disco virtual:

```
WORD            FAT[6144];      /*Tabla FAT de 16 bits por entrada*/
DirectoryEntry  dir[512];       /*512 entradas al directorio raiz*/
BYTE            data[3145728]; /* area de datos del usuario*/
```

El bloque de parámetros del BIOS (dentro del regsitro del boot) tiene el siguiente formato:

```
typedef struct BPB
   {
     char    jump_code[3];
     WORD    ss; /*tamaño del sector en bytes*/
     char    vendor_id[8];
     BYTE    au; /*tamaño de la unidad de asignacion
                 (sectores por cluster)*/
     WORD    rs; /*numero de sectores reservados*/
     BYTE    nf; /*numero de FATs en el disco*/
     WORD    ds; /*numero de entradas al directorio raiz*/
     WORD    ts; /*numero total de sectores*/
     BYTE    md; /*descriptor de medio*/
     WORD    fs; /*numero de sectores en la FAT*/
     WORD    st; /*numero de sectores por track*/
     WORD    nh; /*numero de cabezas*/
     DWORD   hs; /*numero de sectores ocultos*/
   }BIOS_BP;
```

La FAT fue definida como un arreglo de 6144 palabras de 16 bits, cada una de ellas representa un *cluster* del disco, con valores determinados que indican el estado del *cluster* real en el disco. Así es posible saber si dicho *cluster* está libre, ocupado o dañado; en caso de estar ocupado, la entrada contiene el número de *cluster* correspondiente al siguiente bloque de datos de un archivo o el valor 0FFFH, que indica el bloque final de un archivo. Por ejemplo, la búsqueda de un bloque de datos libre en el disco se realiza de acuerdo al siguiente código, incluido en el programa implantado:

```
ent = (WORD *)(memfile+OFFSFAT);

for (i = 0; i<6144; i++)
 if (*ent == 0) break;
    else ent++;
```

Suponiendo que *memfile* es la dirección inicial del archivo mapeado en la memoria virtual y *OFFSFAT* se refiere al desplazamiento donde inicia la tabla FAT del disco. Cuando se encuentra el primer espacio libre en el disco, termina el ciclo.

Las características del disco virtual se resumen a continuación.

- Capacidad para datos del usuario: 3Mbytes

- Tamaño del sector: 512 bytes
- Tamaño del *cluster*: 1 sector (512 bytes)
- Número de entradas de la FAT: 6144
- Tamaño de cada entrada de la FAT: 16 bits
- Tamaño de la FAT: 24 *clusters*
- Número de entradas del directorio raíz: 512

Cuando el sistema se ejecuta por primera vez, se crea el archivo UNIX y se inicializa con la información anterior (estructura del disco para el sistema de archivos DOS). Las veces posteriores, el archivo solo se abre y se deja listo y con los primeros sectores mapeados en la memoria virtual para poder accederlo mediante los respectivos servicios de la interrupción 21H.

El proceso de acceso al disco virtual puede ser resumido en los siguientes puntos:

- Al iniciar la ejecución de la aplicación, se abre el archivo HARDDISK.HDF, que contiene la estructura de un disco duro DOS.
- Se hace un mapeo en memoria virtual de las primeras cuatro páginas del archivo, la longitud de cada página de memoria virtual es de 8 Kbytes, de ahí que el espacio total mapeado en memoria sea de 32 Kbytes. En este espacio se comprende el sector del bloque de parámetros del boot, el directorio y la FAT.
- Cuando se pretende acceder, mediante cualquiera de las funciones implantadas, el espacio de datos del usuario, se realiza un mapeo de la dirección requerida al desplazamiento equivalente del espacio mapeado en memoria.
- Si al hacer el mapeo se detecta que el bloque solicitado no se encuentra mapeado en memoria, se lleva a cabo el mapeo de la página que lo contenga, posiblemente liberando otra página no utilizada.

5.4 Servicios Básicos de Manipulación de Archivos

Una vez creadas las estructuras del sistema de archivos, se desarrolló el código de las funciones que manejan estas estructuras para crear, leer y escribir archivos mediante

el uso apropiado de la tabla de asignación de archivos. En DOS estas funciones se llaman a través de la interrupción 21H como sigue.

Cuando ocurre el llamado a cualquiera de las funciones de la interrupción 21H, es necesario consultar el estado de los registros, pues estos constituyen los parámetros necesarios para la realización de la función solicitada. En el caso de los servicios implantados los registros DX, AH y AL, contienen los parámetros relevantes. Una estructura de datos con el estado de los registros es pasada como argumento a la función encargada de simular un servicio de la interrupción 21H. En nuestra emulación, a partir de los parámetros se hace la conversión de la dirección al desplazamiento dentro del área mapeada en memoria.

Las dos macros siguientes convierten una dirección formada por segmento y desplazamiento a una dirección lineal:

```
#define Addr_8086(x,y) ((((x)& 0xffff)<< 4)+((y)& 0xffff))
#define Addr(s,x,y) Addr_8086(((s)->x),((s)->y))

dir = (u_char *) Addr(state, ds, edx);
```

Después de obtener la dirección lineal a partir del valor de los registros de segmento DS y desplazamiento DX en la estructura de registros, se verifica si tal dirección cae dentro del área mapeada en memoria. Para esto se calcula la página que corresponde a la direción lineal obtenida. En caso de no estar mapeada la página, se hace una lectura de ésta, como se muestra a continuación.

```
np = (int) ((int)dir_lineal / 8192); /*8192 tamaño de la página*/
if (np!=np1) /*np1: número de página en memoria, np: nueva página*/
  { ip2 = 8192 * np;
    vm_read(task, (vm_address_t)(addr+ip2), vm_page_size,
           (pointer_t *)&page, &cnt);
    if (error != KERN_SUCCESS)
       mach_error("Call to vm_read() failed", error);
  }
```

Entonces se hace apuntar la dirección inicial de la página recién obtenida y se calcula el desplazamiento del dato dentro de la misma.

```
    ap = (char *)page;
    index =(int)((int)dir % 8192);
    np1 = np;
```

Una vez localizado el dato dentro de la página adecuada, se obtiene el código de la instrucción, en este caso interesa conocer el número de servicio solicitado, para efectuar la emulación de ese servicio en caso de tratarse de un llamado a la interrupción 21H. La decisión del servicio a realizar, depende del valor del registro AL

```
switch(HIGH(state->eax)) {
        ...
```

5.4.1 Definición de la DTA

La función 1AH de la interupción 21H, permite definir el área de transeferencia de datos para un archivo (DTA), esta área debe estar definida previamente a una operación de entrada y salida a un archivo y consiste de un bloque de 512 bytes en el cual deberán colocarse los datos a escribir a un archivo o donde quedará la información obtenida después de una operación de lectura. Los parámetros utilizados por esta función son los siguientes:

Registro	**Contenido de la entrada**
AH	1AH Servicio definición de la DTA
DS:DX	es la dirección inicial del espacio designado como DTA.
	DS es el valor del segmento; DX es el desplazamiento

5.4.2 Creación

La función que crea un archivo corresponde al número de servicio 16H de la interrupción 21H. Esta función utiliza los siguientes parámetros:

Registro	**Contenido de la entrada**
AH	16H Servicio FCB creación de archivo
DS:DX	es la dirección del bloque de control de archivo FCB.
	DS es el valor del segmento; DX es el desplazamiento
	a partir de la dirección especificada por DS

El FCB es el bloque de control de archivo. Esta estructura contiene información para manejar un archivo. Cuando se crea o abre un nuevo archivo, el FCB proporcionado por el usuario debe contener solamente el número de unidad de disco, el nombre y la extensión del mismo. Posterior a la solicitud de algún servicio sobre el archivo, el FCB llena el resto de sus campos de acuerdo a las operaciones realizadas sobre el mismo. Los campos que constituyen un FCB son los siguientes:

```
struct FCB {
  char numero_unidad;
  char nombrearchivo[8];
  char extension[3];
  unsigned bloque_actual;
  unsigned tamaño_registro;
  long tamaño_archivo;
  unsigned fecha_archivo;
  unsigned hora_archivo;
  char reservado[8];
  char numero_registro_actual;
  long numero_registro_aleatorio;
}
```

Una vez detectado que el código de la instrucción que generó la excepción al núcleo correspondía al servicio 16H de la interrupción 21H, se realizan los siguientes pasos:

1. Se revisan los datos referentes a nombre y extensión del área de FCB, en caso de no ser correctos, se coloca el código del error correpondiente en el registro AL.

2. Se busca una entrada libre en el directorio raíz para colocar los datos del archivo recién creado.

3. Se busca espacio en el área de datos del usuario; en caso de no haber espacio libre, se coloca el código del error correspondiente en el registro AL.

4. Se llenan los datos de la entrada de directorio encontrada para el archivo, colocando nombre, extensión, longitud en cero y sector inicial con el número de entrada encontrada disponible en la FAT, a su vez, a esta entrada de la FAT se le asigna el valor de fin de archivo.

Al término de este servicio se obtiene uno de los siguientes resultados:

Registro	**Contenido a la salida**
AL	Estado de creación
	00H Con éxito
	FFH Con error

Además de los resultados reportados en los registros, se espera que el FCB sea actualizado como sigue:

El campo número_unidad especifica el número de unidad deseada, donde 0 es la unidad actual, 1 es la unidad A, 2 es la unidad B, y así sucesivamente. Este campo es llenado por el programador para indicar donde ubicar su archivo.

El campo nombrearchivo contiene el nombre del archivo deseado. Si el nombre tuviera menos de ocho caracteres, se rellena con espacios.

El campo extensión contiene la extensión del nombre del archivo deseado. Si la extensión tuviera menos de tres caracteres, se rellena con espacios; este dato también es proporcionado por el programador.

El campo bloque_actual es el número del bloque que contiene al registro actual.

El campo tamaño_registro especifica el tamaño de cada registro en bytes, en este caso, consideramos registros de 512 bytes.

El campo tamaño_archivo especifica el tamaño del archivo en bytes.

El campo fecha_archivo contiene la fecha de creación o última modificación del archivo. Los bits de este campo representan la fecha como sigue:

Bits	Fecha
0-4	Valor del día del 1 al 31
5-8	Valor del mes del 1 al 12
9-15	Valor del año relativo a 1980

El campo hora_archivo contiene la hora de creación o última modificación del archivo. Los bits de este campo representan la hora como sigue:

Bits	Hora
0-4	Dobles segundos de 0 a 29
5-10	Minutos de 0 a 59
11-15	Horas de 0 a 23

El campo número_registro_actual especifica el registro actual dentro del bloque actual.

El campo número_registro_aleatorio especifica el número relativo de registro para el acceso aleatorio al archivo.

En el caso de la creación de un archivo, los campos del FCB que se actualizan son: bloque_actual, tamaño_archivo, fecha_archivo, hora_archivo y número_registro_actual.

5.4.3 Apertura

La función para abrir un archivo por medio de un bloque de control de archivo corresponde al número de servicio 0FH de la interrupción 21h. Esta función busca en el directorio en uso el archivo especificado en el FCB y, si el archivo existe, lo abre. Los datos requeridos por esta función son:

Registro	**Contenido de la entrada**
AH	0FH Función para abrir un archivo
DS:DX	es la dirección del bloque de control de archivo FCB.
	DS es el valor del segmento; DX es el desplazamiento
	a partir de la dirección especificada por DS

Los pasos realizados por nuestra función son los siguientes:

1. Se revisan los datos referentes a nombre y extensión en el FCB; en caso de no ser correctos, se coloca el código del error correpondiente en el registro AL.

2. Se recorre el directorio en uso en busca de una entrada cuyos campos nombre y extensión coincidan con los especificados en el FCB.

3. En caso de no encontrarse el archivo especificado, se coloca el código de error correspondiente en el registro AL.

4. Se obtiene de la entrada de directorio el valor referente al primer bloque del archivo en el disco y se coloca en el FCB como registro actual.

Al término de este servicio se obtiene uno de los siguientes resultados:

Registro	**Contenido a la salida**
AL	Estado de búsqueda
	00H Con éxito
	FFH Con error

De manera similar al servicio de creación, se actualizan los campos del FCB de acuerdo a la información obtenida de la entrada de directorio asociada al archivo.

5.4.4 Lectura

El servicio de lectura secuencial de archivos corresponde al número de servicio 14H de la interrupción 21H. Este servicio lee el registro actual preparado por el bloque de

control de archivo (FCB).

Registro	**Contenido de la entrada**
AH	14H Servicio FCB lectura secuencial
DS:DX	es la dirección del bloque de control de archivo FCB. DS es el valor del segmento; DX es el desplazamiento a partir de la dirección especificada por DS

Los pasos seguidos por nuestra rutina de atención de este servicio son los siguientes:

1. A partir del FCB se obtiene el campo correspondiente al número de *cluster* a ser leído.

2. Se mapea el número de *cluster* a la dirección correspondiente en el archivo UNIX que emula al sistema de archivos DOS.

3. Si la dirección obtenida corresponde a una porción no mapeada del archivo, se realiza el mapeo en memoria virtual de la página que contenga este sector del archivo.

4. Si la información encontrada en este bloque de datos indica final de archivo, se coloca el código correspondiente en el registro AL.

5. Si no se ha encontrado el fin de archivo, se copia del *cluster* de disco correspondiente hacia el área de transferencia de disco.

6. Consultando el índice del registro en la FAT, se actualiza en el FCB el número de registro corriente, para hacer la siguiente lectura.

Al término de la atención de este servicio, se pueden tener los siguientes resultados:

Registro	**Contenido a la salida**
AL	Estado de lectura 00H Con éxito 01H Se encontró fin de archivo y no hay datos 02H Area de transferencia de disco sobrepasa los límites de un segmento 03H Datos parciales y se alcanzó el fin de archivo

5.4.5 Escritura

La función de manejo de archivos que realiza el servicio de escritura secuencial usando el bloque de control de archivo, es la función 15H de la interrupción 21H. Los datos requeridos en un programa para efectuar esta operación son los siguientes:

Registro	**Contenido de la entrada**
AH	15H Servicio FCB de escritura secuencial
DS:DX	es la dirección del bloque de control de archivo FCB. DS es el valor del segmento; DX es el desplazamiento a partir de la dirección especificada por DS

Los pasos que realiza la rutina de atención de este servicio son los siguientes:

1. Se obtiene en la FAT un espacio libre en el disco.

2. Si se cuenta con espacio disponible, se actualiza con el número de *cluster* libre encontrado en la FAT, el registro actual en el FCB.

3. Se aplica la función de transformación que mapea el número de *cluster* a la dirección correspondiente en el archivo UNIX que emula al sistema de archivos DOS.

4. Si la dirección obtenida corresponde a una porción no mapeada del archivo, se realiza el mapeo en memoria virtual de la página que contenga este sector del archivo.

5. Se hace la copia de los datos apuntados por el área de transferencia de datos al área correspondiente en archivo UNIX.

6. Se señala esta entrada de la FAT como el fin de archivo.

7. Si todas las operaciones son exitosas, se reporta el resultado de la escritura en el registro AL.

8. En caso de no haber encontrado espacio libre en la FAT, se coloca el código de error correspondiente en el registro AL.

Al término de la atención de este servicio, se pueden tener los siguientes resultados:

Registro	Contenido a la salida
AL	Estado de escritura 00H Con éxito 01H Disco lleno 02H Area de transferencia de disco sobrepasa los límites de un segmento

5.4.6 Búsqueda de la primera ocurrencia en el directorio

La función para buscar en el directorio la primera ocurrencia de un archivo mediante su FCB se realiza a través del servicio 11H de la interrupción 21h. Esta función busca en el directorio la primera entrada cuyo nombre y extensión correspondan con los valores especificados en el FCB, esta información puede contener caracteres comodín (? ó *) por lo que puede haber más de un archivo que corresponda al nombre almacenado en el FCB. Si hay coincidencia, el nombre y extensión de la respectiva entrada del directorio son colocados en el área DTA definida previamente. Los datos requeridos por esta función son:

Registro	Contenido de la entrada
AH	11H Función para buscar la primera ocurrencia de un archivo en el directorio
DS:DX	es la dirección del bloque de control de archivo FCB. DS es el valor del segmento; DX es el desplazamiento a partir de la dirección especificada por DS

Los pasos realizados por nuestra función son los siguientes:

1. Se revisan los datos referentes a nombre y extensión en el FCB; en caso de no ser correctos, se coloca el código del error correpondiente en el registro AL.

2. Se recorre secuencialmente el directorio en uso en busca de una entrada cuyos campos nombre y extensión coincidan con los especificados en el FCB hasta encontrarse la primera coincidencia o terminar el recorrido del directorio.

3. En caso de no encontrarse el archivo especificado, se coloca el código de error correspondiente en el registro AL.

4. Si se encuentra la primera coincidencia, se obtiene de la entrada de directorio el nombre y la extensión del archivo buscado y se copia al área DTA.

Al término de este servicio se obtiene uno de los siguientes resultados:

Registro	**Contenido a la salida**
AL	Estado de búsqueda
	00H Con éxito
	FFH Con error

5.4.7 Búsqueda de la siguiente ocurrencia en el directorio

La función para buscar en el directorio la siguiente coincidencia de un archivo mediante la información del FCB se realiza a través del servicio 12H de la interrupción 21h. Esta función continúa la búsqueda en el directorio de la siguiente entrada cuyo nombre y extensión correspondan con los valores especificados en el FCB y si hay coincidencia, el nombre y extensión de la respectiva entrada del directorio son colocados en el área DTA definida previamente. Los datos requeridos por esta función son:

Registro	**Contenido de la entrada**
AH	12H Función para buscar la siguiente ocurrencia de un nombre de archivo en el directorio
DS:DX	es la dirección del bloque de control de archivo FCB. DS es el valor del segmento; DX es el desplazamiento a partir de la dirección especificada por DS

Los pasos realizados por nuestra función son los siguientes:

1. Se continúa el recorrido del directorio en uso a partir de la entrada siguiente a donde se encontró la primera ocurrencia y hasta encontrar otra entrada que coincida.

2. En caso de no encontrarse el archivo especificado, se coloca el código de error correspondiente en el registro AL.

3. Si se tuvo éxito en la búsqueda, se obtiene de la entrada de directorio el nombre y extensión del archivo encontrado y se copia al área DTA.

Al término de este servicio se obtiene uno de los siguientes resultados:

Registro	**Contenido a la salida**
AL	Estado de la búsqueda
	00H Con éxito
	FFH Con error

5.5 Servicios Básicos de Entrada y Salida Estándar

Para poder mostrar al usuario resultados visuales de las operaciones sobre archivos, se realizó la simulación de dos funciones de entrada y salida estándar DOS. Estas funciones se refieren a los servicios 9 y 10 de la interrupción 21H: escribir una cadena de caracteres al video y leer una cadena de caracteres a través del teclado.

5.5.1 Escribir una cadena en el dispositivo de salida estándar

La función 09 de la interrupción 21H muestra en el dispositivo de salida estándar la cadena pasada en el área apuntada por el registro DX, tal cadena debe terminar con el caracter del signo de dólar $ (ASCII 36).

Registro	**Contenido de la entrada**
AH	09H
DS	Dirección del segmento de la cadena a presentar
DX	Dirección del desplazamiento de la cadena a presentar

Registro	**Contenido a la salida**
Ninguno	

5.5.2 Entrada desde el teclado a la memoria

Esta función lee caracteres procedentes de un dispositivo de entrada estándar hacia una área de memoria definida por el usuario, haciendo eco de los caracteres en la pantalla.

Registro	**Contenido de la entrada**
AL	Máximo número de caracteres
AH	0AH
DS	Dirección del segmento del área de memoria del usuario
DX	Dirección del desplazamiento del área de memoria del usuario

Registro	**Contenido a la salida**
DS:DX	Apunta a la cadena leída

Ambas funciones fueron implantadas mediante las funciones *printf()* y *scanf()* respectivamente, por lo cual no se hace mayor referencia a ellas.

5.6 Resumen

El sistema de archivos DOS consiste de: el **registro de arranque** (*boot*), formado por un número variable de sectores que se encuentran al inicio del disco; una o más copias de la **Tabla de Asignación de Archivos**; el **directorio** y el área de datos para los archivos.

La emulación del sistema de archivos DOS se implementó mediante la creación de un archivo UNIX (nuestro *disco virtual*) con estructuras representando al registro del *boot*, con su bloque de parámetros del BIOS, una FAT de 6144 entradas para mapear cada uno de los *clusters* del disco virtual, un arreglo de 512 entradas para el directorio raíz y un área de datos del usuario, limitada a 6144 sectores de 512 bytes cada uno. Este archivo es abierto cuando se ejecuta el sistema monitor y se mapea en memoria virtual, posteriormente su información es accedida y modificada como respuesta a los llamados al sistema (interrupción 21h) que los programas realizan.

La otra parte de nuestra emulación prototipo fue la creación de algunos servicios de la interrupción 21h relacionados con el manejo del sistema de archivos. Los servicios emulados fueron:

1. Definición del área de transferencia DTA, servicio 1AH.
2. Creación de archivo mediante FCB, servicio 16H.
3. Apertura de archivo mediante FCB, servicio 0FH.
4. Lectura secuencial de archivo mediante FCB, servicio 14H.
5. Escritura secuencial a archivo mediante FCB, servicio 15H.
6. Búsqueda de la primera ocurrencia del directorio, servicio 11H.
7. Búsqueda de la siguiente ocurrencia del directorio, servicio 12H.

Capítulo 6

Monitor DOS

Este capítulo describe el diseño e implantación de un programa monitor que constituye un servidor DOS, encargado de canalizar las solicitudes de servicios (DOS) hechos por programas 8086, a los procedimientos de emulación de servicios presentados en el capítulo anterior. Nuestro diseño utiliza varias de las facilidades de Mach y del procesador 80486.

6.1 Panorama general

Para comprender nuestra emulación prototipo de servicios DOS sobre Mach es necesario considerar los aspectos siguientes: la manera en que los programas DOS llaman al sistema operativo, la organización de la memoria física disponible bajo DOS, y la operación del modo virtual 8086 del procesador 486 de Intel.

Los programas DOS utilizan dos módulos del sistema para realizar sus tareas [MALAN 91]: el BIOS ROM y el núcleo de DOS. El BIOS es un conjunto de rutinas del ROM que soportan llamados al sistema para dispositivos específicos. El núcleo de DOS está localizado en la RAM y proporciona una interfaz general de sistema operativo. El núcleo de DOS usa las rutinas del BIOS para realizar muchas de las operaciones; sin embargo, frecuentemente los programas DOS pasan por alto el núcleo de DOS y llaman directamente al BIOS.

Los programas DOS realizan los llamados al sistema cargando los parámetros a los registros de la máquina y generando una interrupción no enmascarable. Los registros usados para pasar los parámetros varían dependiendo del llamado al sistema. Las interrupciones son generadas por los programas DOS con la instrucción de máquina

INT. La instrucción INT toma el número de interrupción como un argumento que es usado como índice en la tabla vector de interrupciones localizada en memoria baja. El vector de interrupción seleccionado por la instrucción, determina qué rutina de servicio manejará la interrupción. Diferentes números de interrupción son usados por diferentes llamados al sistema de DOS y el BIOS, y cada uno de estos números tiene un punto de entrada en el vector.

Otro punto central para emular DOS en Mach es el modo de operación virtual 8086 del procesador (**V86**). En este modo, el procesador interpreta la memoria y ejecuta instrucciones como si fuera una máquina nativa 8086, pero un conjunto de instrucciones se consideran privilegiadas y generan fallas. Nuestra emulación de DOS sobre Mach crea dos hilos. Un hilo opera en modo V86 y corre el programa DOS del usuario, generando fallas de Protección General por aquellas instrucciones que manipulan el estado de la máquina, generan o regresan de las interrupciones y accesan puertos privilegiados de E/S.

El otro hilo, llamado monitor, corre en modo usuario y se encarga de servir/procesar las fallas generadas por el hilo corriendo en modo V86 (ver Figura 6.1). Cuando ocurre una falla, el registro IP se queda apuntando a la instrucción que generó la excepción, por lo que se hace necesario consultar el estado del hilo para saber el valor de tal registro. Una vez obtenida la dirección donde se encuentra la instrucción, se analiza el código de ésta y se determina de qué manera deberá atenderse. En caso de tratarse de una interrupción 21H, por ejemplo, se analiza el contenido de los registros para saber de qué servicio se trata, consultando el registro AH, y se invoca a la función que simula tal servicio.

6.2 Modo virtual del procesador

Como ya se ha mencionado en un capítulo previo, el procesador soporta la ejecución de uno o más programas en un ambiente en modo protegido, permitiendo que las tareas virtuales 8086 puedan ejecutarse concurrentemente con otras tareas i486.

El procesador corre en modo virtual 8086 cuando el bit VM (máquina virtual) del registro EFLAGS es puesto a uno. El procesador prueba esta bandera bajo dos condiciones generales:

1. Durante la carga de los registros de segmento para saber si debe usar la traducción de direcciones estilo 8086.

2. Durante la decodificación de instrucciones, para determinar cuales son sensitivas

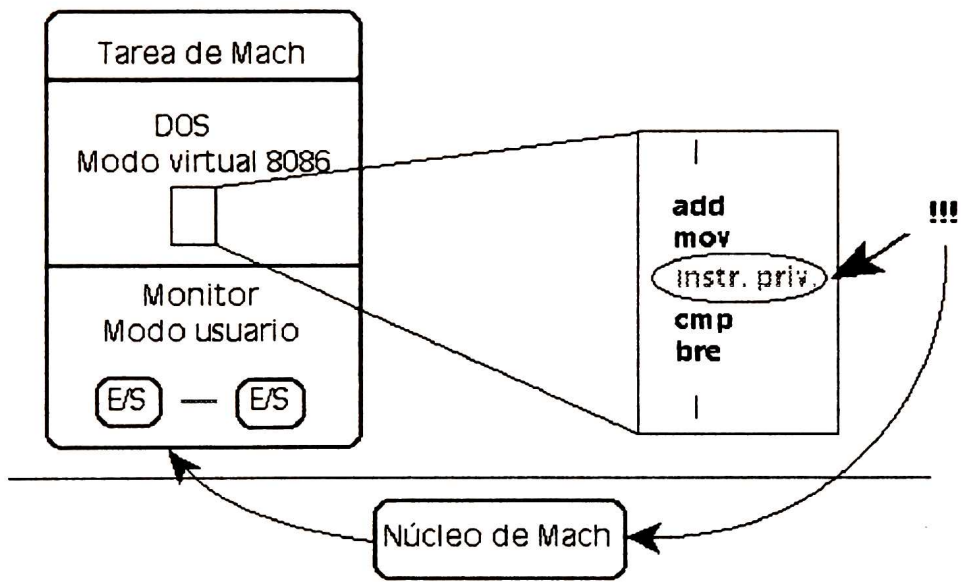


Figura 6.1: Emulación de DOS sobre Mach.

a IOPL (*Input Output Level*), y qué instrucciones no son soportadas (como en el modo real)

Para cambiar el procesador al modo virtual 8086, se incluyeron dentro del código del monitor las siguientes operaciones:

- Se crea el hilo virtual; la dirección pasada al llamado para la creación de este hilo corresponde a la de la función *thread86()*. Dentro de esta función se hacen los pasos necesarios para la ejecución de este hilo en modo virtual del procesador.
- Se reserva una dirección de puerto y se asocia esta dirección al hilo virtual para que funcione como puerto de excepción.
- Se obtiene el estado del procesador y se asocia un valor de cero a cada uno de los registros del procesador: cs, ds, ss, gs, es, bp, ax, bx, cx, dx, di, si y sp. Al

registro de banderas se le habilita el bit VM, que indica que el procesador está en modo virtual.

- El registro ip se deja apuntando a la dirección de memoria donde se encuentra cargado el código de la aplicación DOS a ejecutar en este modo.
- Una vez asignados los valores a todos los registros, se asigna este nuevo estado al procesador.

El conjunto de pasos listados permite hacer la conmutación de modo real a modo protegido, es decir, el procesador estará en modo virtual, y el hilo correspondiente al código de la aplicación DOS será el que se ejecute en el modo protegido.

6.3 Creación del hilo virtual

El hilo V86 es el unico hilo que opera en el modo virtual 8086. Este hilo ejecuta las aplicaciones DOS y el código que genera los fallos de protección general. Estos fallos son instrucciones privilegiadas interceptadas por el núcleo. La creación y mantenimiento del hilo virtual (V86), se realizan en el núcleo de Mach haciendo uso de los llamados al sistema para obtener y asignar el estado de los hilos. Para la creación del hilo virtual se realizan los siguientes pasos:

- Crear una tarea.
- Asignar un espacio de direcciones de memoria a la tarea recién creada. Este espacio será ocupado por el código del programa DOS indicado por el usuario; su tamaño debe ser dado como un múltiplo del tamaño de una página de memoria virtual.
- Se abre el archivo obtenido del usuario, que corresponde al programa DOS a ejecutar.
- Se lee el archivo, por buffers del tamaño de una página de memoria virtual, y se copia al espacio de direcciones asignados a la tarea recién creada. Esta operación se efectúa sucesivamente hasta completar la carga del programa en el área de memoria correspondiente.
- Se obtiene la dirección de carga del programa, que será considerado el punto de entrada de ejecución para el hilo virtual.

- Se llama a la función *thread86* para crear el hilo virtual.
- Al regreso de la función, y si la creación fue exitosa, se pone en estado de ejecución el hilo virtual.

Una vez que el hilo virtual se encuentra activo, es capaz de atender las excepciones que puedan ser generadas por el intento de ejecución de alguna de las instrucciones privilegiadas, ya que proporciona el soporte para la emulación de éstas instrucciones.

6.4 Detección de excepciones

El hilo monitor es el manejador de excepciones del hilo virtual, proporcionando la emulación de los servicios solicitados por las diferentes instrucciones del grupo de instrucciones privilegiadas. La emulación es dividida en grupos dependiendo del tipo de servicio que deberá ser proporcionado.

Cuando el programa DOS se encuentra en ejecución, es necesario detectar el llamado a las instrucciones que queremos capturar para redirigirlas a las funciones de emulación. La manera de llevar a cabo esto es enterando de las excepciones al monitor.

Para procesar las excepciones dentro de la tarea principal de la aplicación, se hace un llamado a la función implantada *process_exceptions()*. En esta función se definen las estructuras necesarias para la recepción y envío de mensajes de excepción. Dentro de estas estructuras debe colocarse la dirección previamente definida como puerto de excepción para el hilo virtual. Las operaciones realizadas dentro de la función son las siguientes:

- Se hace el llamado al sistema que dejará al hilo monitor en espera de que ocurra una excepción (llamado *msg_receive()* de Mach).
- El hilo entra a un ciclo infinito dentro del cual permanece haciendo lo siguiente:
  - Llama a la función de Mach *exc_server()*. Esta función se encarga de despachar un mensaje recibido por el puerto de excepción, llamando al manejador de excepción apropiado. Este manejador de excepciones se encuentra en el cuerpo de la función *catch_exception_raise()* que es la encargada de la emulación de las instrucciones que generaron la excepción.
  - Se envía la respuesta indicando que el mensaje ha sido adecuadamente procesado.

- Si la instrucción recién procesada fue equivalente al fin del programa en ejecución (se sabe mediante el valor de la bandera *terminate*), se concluye el ciclo.
- Se queda en espera de la siguiente excepción.

De los pasos descritos anteriormente, es importante destacar el funcionamiento del llamado a *exc_server()*. Esta función debe usarse después de haber recibido un mensaje en un puerto de excepción previamente definido; una vez recibido el mensaje de excepción, la función ejecuta el código contenido en la función *catch_exception_raise()*, que constituye el manejador de excepciones. En esta función el monitor lleva a cabo diversos pasos, de los cuales se destaca la obtención del estado del procesador cuando ocurrió la excepción. Con los valores almacenados en cada uno de los registros del procesador, es posible calcular la dirección de memoria absoluta correspondiente a la instrucción que generó la excepción, a partir de los valores de los registros CS e IP. Las características de los modos de direccionamiento virtual y 8086 son:

Direccionamiento 8086:

- 16 bits del registro de segmento
- 16 bits del registro de desplazamiento
- Direcciones de 20 bits = segmento << 4+desplazamiento

Modo Virtual 8086:

- El hardware opera con memoria virtual
- Emula ejecución de instrucciones y direccionamiento 8086
- Operaciones privilegiadas causan excepciones de protección general.
- Excepciones manejadas por el núcleo

El código empleado para hacer la conversión de los valores de CS e IP a la dirección de memoria 8086 correspondiente se lleva a cabo a través de Addr(), de acuerdo a los siguientes cálculos:

```
#define Addr_8086(x,y) (( ((x) & 0xffff) << 4) + ((y) & 0xffff))
#define Addr(state,cs,ip) Addr_8086(((state)->cs),((state)->ip))
```

el valor correspondiente al segmento se corre cuatro lugares a la izquierda y se suma con el valor del desplazamiento.

Se determina entonces en qué número de página de la memoria virtual asignada a la tarea se encuentra la dirección (de la instrucción que generó la excepción), y se lee la página correspondiente. También se calcula el desplazamiento relativo, a la página leída, de la instrucción y se hace almacenar en una variable que se considera un apuntador a la instrucción. El código que hace estos cálculos es el siguiente:

```
dir8086 = (u_char *) Addr(state, cs, ip);
np = (int) (dir8086 / SIZEPAG);
index = (int)((int)dir8086 % SIZEPAG);
```

El número de página asociado a la dirección calculada se determina a través de la división de tal dirección por el tamaño de la página. Entonces se determina si la página se encuentra en memoria, y en caso de no ser así, se realiza la lectura. Con el residuo de la dirección 8086 y el tamaño de la página se obtiene el desplazamiento donde se ubica la instrucción a partir del inicio de la página actual de memoria virtual.

El tipo de excepción ocurrido puede ser cualquiera de los siguientes: EXC_BAD_INSTRUCTION, EXC_BREAKPOINT, EXC_BAD_ACCESS y EXC_ARITHMETIC, de todas estas excepciones, la unica emulada en el proyecto fue EXC_BAD_INSTRUCTION, que es atendida a través de un llamado a la función *do_bad_instruction()*.

Dentro de *do_bad_instruction()*, se clasifica la atención a la excepción generada por alguna "instrucción mala", es decir una de las pertenecientes al conjunto de instrucciones privilegiadas, como puede ser: el llamado o retorno de una interrupción, alguna instrucción que manipule el registro de banderas, etc. En esta función se realiza la emulación de cada una de estas instrucciones. En el caso de la instrucción INT se invoca a la función *do_int()* que atiende a cierto número de interrupciones, el proceso de atención para esta instrucción se da de la siguiente manera:

- Se incrementa el apuntador al código de la instrucción para obtener el número de interrupción requerido.

- Si el número de interrupción se refiere a la INT 21H, de la estructura que apunta el estado del procesador se obtiene el contenido de la parte alta del registro AX, donde se encuentra el servicio solicitado. Entonces se llama a la función de emulación correspondiente.

- Los números de interrupciones emulados por esta función son INT 13H, INT 21H e INT 20H, (servicios de disco del BIOS, del núcleo de DOS y terminación del programa en ejecución, respectivamente).

Fueron emulados siete servicios de la interrupción 21H, los básicos del sistema de archivos: creación, apertura, lectura y escritura de archivos, búsqueda de la primera y siguiente ocurrencia de un archivo en el directorio. La implantación de tales servicios fue explicada con detalle en el capítulo anterior.

6.5 Resumen

Para realizar sus tareas, los programas DOS pueden hacer uso del BIOS ROM que es un conjunto de rutinas para el manejo de dispositivos específicos y el núcleo de DOS, que localizado en la RAM, proporciona una interfaz general de sistema operativo usando en ocasiones las rutinas del BIOS para implementar muchas de sus operaciones. Los programas DOS realizan los llamados al sistema cargando los parámetros a los registros de la máquina y generando interrupciones.

Por otro lado, recordemos el modo de operación virtual 8086 del procesador (**V86**), en el cual la memoria se interpreta y las instrucciones se ejecutan como si fuera una máquina nativa 8086, además de que un conjunto de instrucciones consideradas privilegiadas generan fallas al tratar de ejecutarse.

La emulación de DOS sobre Mach crea dos hilos. Un hilo opera en modo V86 y corre el programa DOS del usuario, generando fallas de Protección General por aquellas instrucciones que manipulan el estado de la máquina, generan o regresan de las interrupciones y acceden puertos privilegiados de E/S. El otro hilo, llamado monitor, corre en modo usuario y se encarga de procesar las fallas generadas por el hilo corriendo en modo V86. Cuando ocurre una falla, el registro IP se queda apuntando a la instrucción que generó la excepción, por lo que se hace necesario consultar el estado del hilo para saber el valor de tal registro. Una vez obtenida la dirección donde se encuentra la instrucción, se analiza el código de ésta y se determina de qué manera deberá atenderse. En caso de tratarse de una interrupción 21H, por ejemplo, se analiza el contenido de los registros para saber de qué servicio se trata, consultando el registro AH, y se invoca a la función que simula tal servicio.

Capítulo 7

Resultados

El monitor y la emulación prototipo de archivos DOS descritos en los dos capítulos anteriores, fueron probados con programas DOS que realizan varias operaciones de archivos. En este capítulo presentamos la manera en que se prepara el emulador de DOS y las aplicaciones que han de correr en el.

7.1 Compilación del emulador

Los archivos fuentes del monitor son los siguientes:

type.h: contiene la definición de estructuras de datos del disco virtual DOS,

base.h: contiene la definición de macros para el manejo de la memoria virtual (asignación y liberación), así también como manipulación de los registros (asignación y consulta de valores),

bios.h: contiene definiciones de estructuras de dispositivos, información de la partición del disco duro, conversión de direcciones segmentadas a lineales y lineales a segmentadas, etcétera.

def.h: contiene las definiciones del código del conjunto de instrucciones del procesador,

dos.h: contiene códigos de error y estructuras de datos del sistema de archivos (por ejemplo formato de los nombres de archivos)

dos_fs.c: en este módulo se encuentran las funciones del sistema de archivos (servicios de la interrupción 21h) y auxiliares (validación de FCB, recorrido de la FAT y directorio, etc.)

dos_disk.c: en este módulo se incluyen también funciones auxiliares para el manejo del sistema de archivos,

Monitor.c: incluye la implantación del monitor que pone al procesador en modo virtual (V86) y maneja las excepciones generadas por la ejecución de instrucciones privilegiadas.

Todos estos archivos deben estar en el mismo directorio. Para realizar la compilación deben seguirse los pasos:

1. Compilar el módulo **dos_fs.c** con el siguiente comando:
   cc -c dos_fs.c
   de donde el compilador genera el archivo en código objeto: **dos_fs.o**

2. Compilar el módulo **dos_disk.c** con el siguiente comando:
   cc -c dos_disk.c
   de donde el compilador genera el archivo en código objeto: **dos_disk.o**

3. Compilar el módulo **Monitor.c** con el siguiente comando:
   cc -c Monitor.c
   de donde el compilador genera el archivo en código objeto: **Monitor.o**

4. Ligar los módulos anteriormente compilados mediante el siguiente comando:
   cc -o Monitor Monitor.o dos_fs.o dos_disk.o
   este último paso genera el programa ejecutable **Monitor** que será el nombre a invocar para ejecutar nuestro sistema.

7.2 Creación de aplicaciones DOS

Debido a que el emulador carece de un relocalizador de programas, las aplicaciones a ejecutarse deberán ser del tipo *.com*, estas aplicaciones deben ser creadas y compiladas en sistemas DOS, recordando que sus funciones deberán limitarse al uso de servicios de entrada y salida estándar de cadenas y los servicios del sistema de archivos emulados, es decir: creación, apertura, lectura, escritura y búsqueda de archivos en el directorio.

En caso de que alguna aplicación ejecutada incluya llamados no implementados, el resultado será solamente la salida de una cadena desplegando el nombre de la función invocada.

7.3 Pruebas

Con el fin de probar el sistema se implementaron los siguientes comandos del shell de DOS: **dir**, **type** y **copy** y los programas **crea** que agrega un archivo al directorio y **escribe** que escribe información al archivo ya existente que se indique. Una sesión del sistema usando tales aplicaciones tendría los siguiéntes resultados:

```
Next$>Monitor
C>dir
Directory of C:\
0 File(s)

C>crea
Nombre: Prueba
Extensin: txt

C>dir
Directory of C:\
PRUEBA   TXT
1 File(s)

C>escribe

C>type
Nombre: Prueba
Extension: txt
hola mundo hola mundo hola mundo hola mundo hola mundo hola mundo hola mundo hola

C>copy
Fuente: Prueba.txt
Destino: Nuevo
```

```
C>type
Nombre: Nuevo
Extension:
hola mundo hola mundo hola mundo hola mundo hola mundo hola mundo hola mundo hola

C>dir
Directory of C:\
PRUEBA   TXT
NUEVO
2 File(s)
```

7.4 Discusión

El monitor está completo para la total funcionalidad de la emulación de aplicaciones DOS en NeXTSTEP. Solo se requiere completar el conjunto de funciones que implementan los servicios del sistema de archivos y otros tales como el manejo del video y demás dispositivos.

Nuestro emulador permite la ejecución de ciertas aplicaciones DOS en una plataforma totalmente diferente como lo es NeXTSTEP. Y su organización e implantación son relativamente sencillos debido a la utilización de los servicios que Mach y el procesador i486 ofrecen para emular otros sistemas.

Por otra parte, no es portable a sistemas con otros procesadores, el tamaño del disco virtual (3 Megabytes) es limitado, y para hacerlo portable a otros procesadores, el emulador tendría que rediseñarse para no hacer uso de las características del hardware considerado, enfocándose unicamente al uso de las funciones del micronúcleo de Mach para lograr la emulación, para esto, habría que extender la máquina virtual para que fuera capaz de interpretar todas y cada una de las instrucciones de código 8086.

Respecto al tamaño del disco virtual, esto se soluciona definiendo simplemente un archivo de mayor longitud, o dejando esta decisión a criterio del usuario en el momento de inicialización del sistema.

Para hacer posible la ejecución concurrente de varios programas 8086, es necesario cambiar la interfaz del emulador permitendo al usuario el manejo de diversas

ventanas de ejecución para cada una de las aplicaciones y agregando mecanismos de sincronización para el uso de funciones reentrantes y recursos no compartibles.

A pesar de haber concluído la parte referente al monitor podemos sugerir algunas extensiones al trabajo, por ejemplo:

- Completar el conjunto de funciones DOS.
- Implementar un relocalizador de código que permita la ejecución de programas .exe.

Capítulo 8

Conclusiones

El propósito inicial de las máquinas virtuales era el permitir a los diseñadores de sistemas operativos experimentar con nuevas ideas sin interferir con la comunidad de usuarios, mediante el aislamiento de una máquina virtual de otra. Actualmente, el beneficio principal es el de resolver problemas de compatibilidad entre sistemas operativos, pues existe una gran cantidad de aplicaciones disponibles para algún sistema específico que sería deseable poder seguir utilizando a pesar de la migración a un sistema operativo diferente. El diseño e implantación de una máquina virtual no es una tarea fácil de realizar, y esta tesis ha presentado un prototipo de máquina virtual, ejemplificando todos los pasos requeridos para el desarrollo de la misma. El sistema emulado es DOS/8086 dentro del sistema operativo NeXTSTEP/80486.

Para la implementación del prototipo se desarrollaron las siguientes componentes principales:

1. Un hilo que opera en modo V86 y que contiene el código de una aplicación DOS. En este modo de operación, las instrucciones de la aplicación que intentan manipular el estado de la máquina, llamar o regresar de las interrupciones y acceder puertos privilegiados de E/S generan fallas de Protección General que son atendidas por el hilo monitor.

2. Monitor: es el programa que corre en modo usuario y se encarga de procesar las fallas generadas por el hilo corriendo en modo V86. Cuando ocurre una falla, se analiza el código de la instrucción que la generó y se atiende de la manera correspondiente, siempre y cuando se trate de alguno de los servicios emulados por el sistema de archivos.

3. Sistema de archivos: consiste de las estructuras del sistema de archivos DOS implementadas sobre un archivo UNIX (registro del *boot*, FAT, directorio y área de datos del usuario dividida en sectores de 512 bytes cada uno). También se tiene la emulación de los servicios de la interrupción 21h de creación, apertura, lectura, escritura y búsqueda de ocurrencias en el directorio.

El concepto de micronúcleo fue muy útil para la implementación del prototipo, pues los micronúcleos estan diseñados para ofrecer soporte para emular otros sistemas operativos.

La contribución principal de nuestro trabajo es la ejemplificación del desarrollo completo de una máquina virtual. Más algunas de las aplicaciones de nuestro trabajo son las siguientes:

- Nuestro trabajo es ideal para enriquecer la práctica de los cursos de sistemas operativos y arquitectura de computadoras
- Puede servir también como base y referencia para lograr la ejecución de sistemas operativos ya existentes como XINIX o MINIX como aplicaciones de otros sistemas operativos como Windows N'I', Linux u otros.
- A manera de tareas de estos cursos o incluso temas de tesis, es posible realizar extensiones a nuestro trabajo que logren la emulación completa de DOS en Mach y NeXTSTEP, y en otros sistemas con un poco de más trabajo.
- Otra aportación del trabajo es la utilización en Mach de sistemas y datos que ya han sido desarrollados en DOS.
- Comunicación entre aplicaciones DOS y Mach o NeXTSTEP a través del sistema de archivos emulado. Esto puede lograrse si tomamos en cuenta que las aplicaciones DOS se ejecutan de manera concurrente a las aplicaciones comunes NeXTSTEP y dentro de ellas pueden integrarse interfases que hagan uso de los mecanismos de comunicación entre procesos que proporciona Mach para hacer posible esta comunicación con el resto de los procesos.

Apéndice A

Código Fuente

```
/*****************************************************************
type.h  definicion de estructuras de datos del disco virtual DOS
*****************************************************************/
typedef  unsigned char BYTE;
typedef  unsigned short int WORD;
typedef  unsigned long int DWORD;

typedef struct BPB
{
        char    jump_code[3];
        WORD    ss; /*tamano del sector en bytes*/
        char    vendor_id[8];
        BYTE    au; /*tamano de la unidad de alojacion (sectores/cluster)*/
        WORD    rs; /*numero de sectores reservados*/
        BYTE    nf; /*numero de FATs en el disco*/
        WORD    ds; /*numero de entradas al directorio raiz*/
        WORD    ts; /*numero total de sectores*/
        BYTE    md; /*descriptor de medio*/
        WORD    fs; /*numero de sectores en la FAT*/
        WORD    st; /*numero de sectores por track*/
        WORD    nh; /*numero de cabezas*/
        DWORD   hs; /*numero de sectores ocultos*/
}BIOS_BP;

typedef struct ent_dir{
        char    name[8]; /*8*/
        BYTE    attrib;  /*1*/
        char    ext[3];  /*3*/
        char    reserved[10]; /*10*/
        WORD    time; /*2*/
        WORD    date; /*2*/
        WORD    cluster; /*2*/
        DWORD   size; /*4*/
}Ent_Dir;

typedef union Dir{
```

```
        Ent_Dir direc;
        char chardir[32];
}Entry;

typedef union   bpb{
        BIOS_BP  BIOSBlockP;
        char     bloque[44];
        }block;


/*****************************************************************
base.h  definicion de macros
*****************************************************************/
#ifndef _BASE_H
#define _BASE_H

#include <mach/message.h>
#include <mach/exception.h>

#define Debug_Level_0 0
#define Debug_Level_1 1
#define Debug_Level_2 2

extern int video_debug_level;
extern int key_debug_level;
extern int disk_debug_level;

/*
 * Space allocator.
 */
#define Malloc(size) malloc(size)
#define New(typ) (typ *)malloc(sizeof(typ))
#define NewArray(typ,cnt) (typ *)malloc(sizeof(typ)*(cnt))
#define NewStr(str) (char *)strcpy(malloc(strlen(str)+1),str)
#define ZeroNew(cnt,typ) (typ *)calloc(cnt,sizeof(typ))
#define Free(ptr) free(ptr)

/*
 * Array operations
 */
#define Count(arr) (sizeof(arr)/sizeof(arr[0]))
#define Lastof(arr) (Count(arr)-1)
#define Endof(arr) (&(arr)[Count(arr)])

#define Min(a,b) (((a)<(b))?(a):(b))
#define Max(a,b) (((a)>(b))?(a):(b))

#define MASK8(x) ((x) & 0xff)
#define MASK16(x) ((x) & 0xffff)
#define HIGH(x) MASK8((unsigned long)(x) >> 8)
#define LOW(x) MASK8((unsigned long)(x))
#define WORD(x) MASK16((unsigned long)(x))
#define SETHIGH(x,y)  (*(x) = (*(x) & ~0xff00) | ((MASK8(y))<<8))
#define SETLOW(x,y)  (*(x) = (*(x) & ~0xff) | (MASK8(y)))
```

```
#define SETWORD(x,y) (*(x) = (*(x) & ~0xffff) | (MASK16(y)))

#define MACH_CALL(x,y) {int foo;if((foo=(x))!=KERN_SUCCESS){\
                        mach_error(y,foo);exit_dos();}}

typedef int onoff_t;

#define OFF 0
#define ON 1
#define MAYBE 2
#define UNCHANGED 2
#define REDIRECT 3

#define OFFSDIR 12800
#define OFFSFAT 512

extern char *malloc();

#endif _BASE_H


/*****************************************************************
bios.h  definicion de macros
*****************************************************************/
#define DEVICE_NOT_PRESENT 0
#define DEVICE_NOT_INITIALIZED 1
#define DEVICE_INITIALIZED 2
#define DEVICE_INITIALIZATION_FAILED 3

#define MAX_DEVICES 256

#define EFL_SAFE (EFL_CF|EFL_PF|EFL_AF|EFL_ZF|EFL_SF|EFL_DF|EFL_OF|EFL_RF|EFL_IF)
#define EFL_TSAFE (EFL_TF|EFL_CF|EFL_PF|EFL_AF|EFL_ZF|EFL_SF|EFL_DF|EFL_OF|EFL_RF|EFL_IF)

typedef struct device {
u_int d_state;
char * d_path;
int d_fd;

boolean_t d_absability;

/* info from the boot block */
u_char media; /* media type */
u_char bps[2]; /* blocks per sector */
u_char spa; /* sectors per allocation */
u_short sectrk; /* sectors per track */
u_short heads; /* heads per track */
u_char drive; /* drive number */
u_short sects; /* total sectors */
u_long SECTS; /* total number of secotrs */

/* hard disk partition info */
u_char start_head;
u_short start_cylsec; /* starting cylinder and sec */
u_char partition_type;
```

```
u_char end_head;
u_short end_cylsec; /* ending cylinder and sec */
u_long start_sec_rel; /* starting sec w/r to beginnng of disk */
u_long partition_len; /* partition length */
} * device_t;

typedef struct {
u_short offset;
u_short selector;
} * idt_t;

 idt_t idt;
 struct device devices[MAX_DEVICES];

typedef  i386_thread_state_t state_t;

#define Addr_8086(x,y) (( (x) & 0xffff) << 4) + ((y) & 0xffff))
#define Addr(s,x,y) Addr_8086(((s)->x), ((s)->y))

#define Segment(x) (((x) & 0xff000)>>4)
#define Offset(x) ((x) & 0xfff)
#define Abs2Segoff(x) ( (((x) & 0xffff0)<<12) | ((x) & 0xf) )

#define Fprintf(args) fprintf args

#define MAX_IO_PORTS 128
extern u_char io_ports[MAX_IO_PORTS];
#define PORT_ENABLE(x) io_ports[(x)/8] |= (u_char) (1 << ((x) % 8))
#define PORT_DISABLE(x) io_ports[(x)/8] &= (u_char)!(1 << ((x) % 8))
#define PORT_OK(x) (io_ports[(x)/8] & (1 << ((x) % 8)))

 FILE * dbg_fd;
 int dd_fd;
 FILE * dd_stream;
 int iopl_fd;
 boolean_t interrupt_bit;
 boolean_t cs_switch_needed;
 int cs_switch_turns;
 int cs_switch_count;
 int mon_space;
int disk;
char *memfile;



/*****************************************************************
def.h  definiciones de funciones y codigo de instrucciones
*****************************************************************/
#import <mach/thread_status.h>
#import <mach/mach_types.h>

#define DO_NOPS TRUE
#define NOP_CLI_STI TRUE

#define INT_01 0x01
```

```
#define BIOS_VIDEO 0x10
#define BIOS_CONFIG 0x11
#define BIOS_MEMORY 0x12
#define BIOS_DISK 0x13
#define BIOS_SERIAL 0x14
#define BIOS_IO_SUB 0x15
#define BIOS_KBD 0x16
#define BIOS_PRN 0x17
#define BIOS_BASIC 0x18
#define BIOS_REBOOT 0x19
#define BIOS_CLOCK 0x1a
#define TERMINATE 0x20
#define DOS_GENERAL 0x21
#define DOS_TTY 0x29
#define BIOS_MOUSE 0x33
#define BIOS_EMM 0x67
#define DOS_SAFE_TO_USE 0x28
#define DOS_NETWORK_INF 0x2a
#define DOS_EXTRA 0x2f
#define MACH_EXIT 0xfa
#define MACH_FMD 0xfb
#define MACH_XMS 0xfc
#define MACH_MOUSE 0xfd
#define MACH_FS_DEV 0xfe
#define BIOS_KBD_REDIRECT 0xff

#define DOS_FS_REDIRECT 0x11

#define I386_NOP 0x90
#define I386_INT 0xcd
#define I386_HLT 0xf4
#define I386_CLI 0xfa
#define I386_STI 0xfb
#define I386_PUSHF 0x9c
#define I386_POPF 0x9d
#define I386_LOCK 0xf0
#define I386_IRET 0xcf
#define I386_IN1 0xe4
#define I386_IN2 0xe5
#define I386_IN3 0xec
#define I386_IN4 0xed
#define I386_OUT1 0xe6
#define I386_OUT2 0xe7
#define I386_OUT3 0xee
#define I386_OUT4 0xef
#define I386_INS1 0x6c
#define I386_INS2 0x6d
#define I386_OUTS1 0x6e
#define I386_OUTS2 0x6f
#define I386_REPZ 0xf3
#define I386_OP_SIZE 0x66
```

```
/*************************************************************
dos.h  definicion del disco virtual para DOS
*************************************************************/
#include <sys/types.h>

/*
 * Dos error codes
 */
/* MS-DOS version 2 error codes */
#define FUNC_NUM_IVALID 0x01
#define FILE_NOT_FOUND 0x02
#define PATH_NOT_FOUND 0x03
#define TOO_MANY_OPEN_FILES 0x04
#define ACCESS_DENIED 0x05
#define HANDLE_INVALID 0x06
#define MEM_CB_DEST 0x07
#define INSUF_MEM 0x08
#define MEM_BLK_ADDR_IVALID 0x09
#define ENV_INVALID 0x0a
#define FORMAT_INVALID 0x0b
#define ACCESS_CODE_INVALID 0x0c
#define DATA_INVALID 0x0d
#define UNKNOWN_UNIT 0x0e
#define DISK_DRIVE_INVALID 0x0f
#define ATT_REM_CUR_DIR 0x10
#define NOT_SAME_DEV 0x11
#define NO_MORE_FILES 0x12

/* mappings to critical-error codes */
#define WRITE_PROT_DISK 0x13
#define UNKNOWN_UNIT_CERR 0x14
#define DRIVE_NOT_READY 0x15
#define UNKNOWN_COMMAND 0x16
#define DATA_ERROR_CRC 0x17
#define BAD_REQ_STRUCT_LEN 0x18
#define SEEK_ERROR 0x19
#define UNKNOWN_MEDIA_TYPE 0x1a
#define SECTOR_NOT_FOUND 0x1b
#define PRINTER_OUT_OF_PAPER 0x1c
#define WRITE_FAULT 0x1d
#define READ_FAULT 0x1e
#define GENERAL_FAILURE 0x1f
#define SHARING_VIOLATION 0x20
#define FILE_LOCK_VIOLATION 0x21
#define DISK_CHANGE_INVALID 0x22
#define FCB_UNAVAILABLE 0x23
#define SHARING_BUF_EXCEEDED 0x24
#define FILE_ALREADY_EXISTS 0x50

struct dir_ent {
char name[8]; /* dos name and ext */
char ext[3];
u_short mode; /* unix st_mode value */
long size; /* size of file */
time_t time; /* st_mtime */
```

```
struct dir_ent * next;
};

struct dos_name {
char name[8];
char ext[3];
};

typedef struct far_record {
u_short offset;
u_short segment;
} far_t;


typedef u_char * cds_t;
#define cds_current_path(cds) ((char    *)&cds[cds_current_path_off])
#define cds_flags(cds) (*(u_short *)&cds[cds_flags_off])

#define FAR(x) (Addr_8086(x.segment, x.offset))
#define FARPTR(x) (Addr_8086((x)->segment, (x)->offset))

typedef u_short * psp_t;

#define PSPPTR(x) (Addr_8086(x, 0))

typedef u_char * sda_t;

#define sda_current_dta(sda) ((char *)(FARPTR((far_t *)&sda[sda_current_dta_off])))
#define sda_cur_psp(sda) (*(u_short *)&sda[sda_cur_psp_off])
#define sda_filename1(sda) ((char  *)&sda[sda_filename1_off])
#define sda_filename2(sda) ((char  *)&sda[sda_filename2_off])
#define sda_sdb(sda) ((sdb_t    )&sda[sda_sdb_off])
#define sda_cds(sda) ((cds_t)(FARPTR((far_t *)&sda[sda_cds_off])))
#define sda_search_attribute(sda) (*(u_char *)&sda[sda_search_attribute_off])
#define sda_open_mode(sda) (*(u_char *)&sda[sda_open_mode_off])
#define sda_rename_source(sda) ((sdb_t    )&sda[sda_rename_source_off])

/*
 *  Data for extended open/create operations, DOS 4 or greater:
 */
#define sda_ext_act(sda) (*(u_short *)&sda[sda_ext_act_off])
#define sda_ext_attr(sda) (*(u_short *)&sda[sda_ext_attr_off])
#define sda_ext_mode(sda) (*(u_short *)&sda[sda_ext_mode_off])

#define psp_parent_psp(psp) (*(u_short *)&psp[0x16])
#define psp_handles(psp) ((char *)(FARPTR((far_t *)&psp[0x34])))

#define lol_cdsfarptr(lol) (*(far_t *)&lol[lol_cdsfarptr_off])
#define lol_last_drive(lol) (*(u_char *)&lol[lol_last_drive_off])

typedef u_char * lol_t;

#ifdef OLD_OBSOLETE
typedef struct lol_record {
u_char filler1[22];
```

```
far_t  cdsfarptr;
u_char filler2[6];
u_char last_drive;
} * lol_t;
#endif

/* dos attribute byte flags */
#define REGULAR_FILE  0x00
#define READ_ONLY_FILE 0x01
#define HIDDEN_FILE 0x02
#define SYSTEM_FILE 0x04
#define VOLUME_LABEL 0x08
#define DIRECTORY 0x10
#define ARCHIVE_NEEDED 0x20

/* dos access mode constants */
#define READ_ACC 0x00
#define WRITE_ACC 0x01
#define READ_WRITE_ACC 0x02

#define COMPAT_MODE 0x00
#define DENY_ALL 0x01
#define DENY_WRITE 0x02
#define DENY_READ 0x03
#define DENY_ANY 0x40

#define CHILD_INHERIT 0x00
#define NO_INHERIT 0x01

#define A_DRIVE 0x01
#define B_DRIVE 0x02
#define C_DRIVE 0x03
#define D_DRIVE 0x04

int sda_cur_psp_off;
boolean_t mach_fs_enabled;
char u_lower;
char u_upper;
int u_drive;


/*****************************************************************
virtual.c  definicion del disco virtual para DOS
*****************************************************************/
#include "type.h"
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#import <mach/mach.h>

#define PSIZE 8192

#define OFFBOOT 0
#define OFFFAT  512
```

```
#define OFFDIR  12800
#define OFFDATA 29184

block Boot_rec;

block *Boot;

WORD    FAT[6144];       /*Tabla FAT*/
Entry  dir[512];        /*512 entradas al directorio raiz*/
BYTE  data[3145728]; /*datos del usuario*/

void init(int fd)
{
    unsigned long int i;
    char buf[468];

    strcpy(Boot_rec.BIOSBlockP.jump_code, "JMP");
    strcpy(Boot_rec.BIOSBlockP.vendor_id,"MSDOS5.0");
    Boot_rec.BIOSBlockP.ss=(WORD)512;
    Boot_rec.BIOSBlockP.au=(BYTE)1;
    Boot_rec.BIOSBlockP.rs=(WORD)1;
    Boot_rec.BIOSBlockP.nf=(BYTE)1;
    Boot_rec.BIOSBlockP.ds=(WORD)512;
    Boot_rec.BIOSBlockP.ts=(WORD)6201;
    Boot_rec.BIOSBlockP.md=(BYTE)254;
    Boot_rec.BIOSBlockP.fs=(WORD)24;
    Boot_rec.BIOSBlockP.st=(WORD)0;
    Boot_rec.BIOSBlockP.nh=(WORD)0;
    Boot_rec.BIOSBlockP.hs=(DWORD)0;
    for (i=0;i<512;i++)/*Directorio*/
     {
       strcpy(dir[i].direc.name,"        ");
       strcpy(dir[i].direc.ext,"   ");
       dir[i].direc.attrib=i;
       strcpy(dir[i].direc.reserved,"          ");
       dir[i].direc.time=i;
       dir[i].direc.date=i;
       dir[i].direc.cluster=i;
       dir[i].direc.size=i;
     }
   /*6144 entradas de la FAT, dos bytes c/entrada*/
    FAT[0] = (WORD)0xfff7;
    for (i=1;i<6144;i++)/*FAT*/
    FAT[i]=(WORD) 0;

    for (i=0;i<468;i++)
      buf[i] = 0;
    /*se llena el boot record*/
    i   write(fd,(char *)&Boot_rec.bloque, sizeof(Boot_rec.bloque));
   /*Se escribieron i bytes agregar el resto a completar 1 sector*/
    write(fd, buf,468);

    /*se llena la FAT*/

    i = write(fd,(char *)FAT, 12288);
```

```
    printf("Se escribieron: %d bytes\n",i);

    /*se llena el directorio*/
    i = write(fd,(char *)&dir, 512*sizeof(Entry));
    printf("Se escribieron: %d bytes\n",i);

    /*se llena el espacio de datos del usuario*/
    bzero(data, 3145728);
    i = write(fd,(char *)&data, 3145728);
    printf("Se escribieron: %d bytes\n",i);
    return;
}


void main ()
{
 int fd;
 int i;
 char *memfile, vdisk[]="HARDDISK.HDF";
 kern_return_t r;
 size_t size,nitems;

/*CREACION DEL DISCO VIRTUAL*/

  if ((fd=open(vdisk,O_RDONLY))!=0)
   {
    fd = creat(vdisk,0xffff);
    if (fd)
      {
        printf("El archivo se ha creado exitosamente\n");
        init(fd);

        close(fd);
      }
    }

  strcpy(Boot->BIOSBlockP.jump_code, "   ");
  strcpy(Boot->BIOSBlockP.vendor_id,"        ");
  Boot->BIOSBlockP.ss=0;
  Boot->BIOSBlockP.au=0;
  Boot->BIOSBlockP.rs=0;
  Boot->BIOSBlockP.nf=0;
  Boot->BIOSBlockP.ds=0;
  Boot->BIOSBlockP.ts=0;
  Boot->BIOSBlockP.md=0;
  Boot->BIOSBlockP.fs=0;
  Boot->BIOSBlockP.st=0;
  Boot->BIOSBlockP.nh=0;
  Boot->BIOSBlockP.hs=0;

/*MAPEO DEL DISCO A MEMORIA VIRTUAL*/
  fd = open(vdisk,O_RDONLY);

  /* Map part of it into memory. */
  r = map_fd(fd, (vm_offset_t)0,(vm_offset_t *)&memfile,
```

```
                                TRUE, (vm_size_t)32768);
  //Se mapean 4 paginas de memoria virtual, 32k
  if (r != KERN_SUCCESS)
     mach_error("Error calling map_fd()", r);
     else{
        Boot = (block *) memfile;
        printf("Codigo : %s \n", Boot->BIOSBlockP.jump_code);
        printf("Sector size: %d\n",Boot->BIOSBlockP.ss);
        printf("Vendor id: %s\n",Boot->BIOSBlockP.vendor_id);
        printf("au size: %d\n",Boot->BIOSBlockP.au);
        printf("Reserved sectors: %d\n",Boot->BIOSBlockP.rs);
        printf("num. of FAT's:  %d\n",Boot->BIOSBlockP.nf);
        printf("Num. of root entries %d\n",Boot->BIOSBlockP.ds);
        printf("Total sectors %d\n",Boot->BIOSBlockP.ts);
        close(fd);
     }
}

/*****************************************************************
dos_fs.c  Funciones del sistema de archivos, Int 21h
*****************************************************************

#import "base.h"
#import "bios.h"
#import "type.h"
#import "dos.h"

#import <stdio.h>
#import <string.h>
#import <sys/file.h>
#import <sys/ioctl.h>
#import <sys/stat.h>
#import <sys/types.h>
#import <sys/time.h>
#import <ctype.h>
#import <sys/errno.h>

#import <mach/message.h>
#import <mach/exception.h>
#import <mach/mach_init.h>
#import <mach/mach.h>

boolean_t mach_fs_enabled   FALSE;

#define WRITESTD 0x09
#define READSTD 0x0A
#define RESET_DISK 0x0D
#define SET_DEFAULT_DRIVE 0x0E
#define OPEN_FILE_FCB 0x0F
#define CLOSE_FILE_FCB 0x10
#define FIND_FIRST 0x11
#define FIND_NEXT 0x12
#define DELETE_FILE_FCB 0x13
#define READ_FILE_FCB 0x14
#define WRITE_FILE_FCB 0x15
```

```
#define CREATE_FILE_FCB 0x16
#define RENAME_FILE_FCB 0x17
#define SET_DTA 0x1a
#define REMOVE_DIRECTORY 0x3a
#define MAKE_DIRECTORY 0x39
#define SET_CURRENT_DIRECTORY 0x3b
#define CLOSE_FILE 0x3e
#define WRITE_FILE 0x40
#define GET_DISK_SPACE 0x36
#define SET_FILE_ATTRIBUTES 0x43
#define GET_FILE_ATTRIBUTES 0x43
#define RENAME_FILE 0x56
#define DELETE_FILE 0x41
#define OPEN_EXISTING_FILE 0x3d
#define UNDOCUMENTED_FUNCTION_2 0xff

int  curr_rec;
extern char  * dos_root;
extern int  dos_root_len;
extern vm_address_t page;
extern vm_address_t  addr;
extern int      np1;
extern int nfo;
extern block  Boot_rec;
extern block  *Boot;
Ent_Dir  *entr;
char  *dta;

void time_to_dos(clock, date, time)
time_t * clock;
u_short * date;
u_short * time;
{
struct tm * tm;

tm = localtime(clock);

*date = ((((tm->tm_year - 80)&0x1f) << 9) |
 (((tm->tm_mon + 1) & 0xf) << 5) |
 (tm->tm_mday & 0x1f));

*time = (((tm->tm_hour & 0x1f) << 0xb) |
 ((tm->tm_min & 0x3f) << 5));
}



/*
 * function: file_file
 *
 * Finds file fpath using stat.
 * If file exists under given name, it returns true.
 * This routine also checks for uppercase and lowercase
 * versions of last component of filename.  It returns
 * the stat structure modified appropriately and converts
```

```
 * the string to the matching case or to uppercase if there
 * is no match.
 *
 */

boolean_t compare(fname, fext, mname, mext)
char * fname;
char * fext;
char * mname;
char * mext;
{
int i;

        printf("dos_gen: compare '%.8s'.'%.3s' to '%.8s'.'%.3s'\n",
mname, mext, fname, fext);
/* match name first */
for(i=0;i<8;i++) {
if(mname[i] == '?') {
i++;
continue;
}
if(mname[i] == ' ') {
if (fname[i] == ' ') {
break;
}else{
return(FALSE);
}
}
if(mname[i] == '*') {
break;
}
if (isalpha(mname[i]) && isalpha(fname[i])) {
char x = isupper(mname[i]) ?
mname[i]   toupper(mname[i]);
char y   isupper(fname[i]) ?
fname[i] : toupper(fname[i]);
if (x != y) return(FALSE);
} else if(mname[i] != fname[i]) {
return(FALSE);
}
}
/* if got here then name matches */
/* match ext next */
for(i=0;i<3;i++) {
if(mext[i] == '?') {
i++;
continue;
}
if(mext[i] == ' ') {
if (fext[i] == ' ') {
break;
}else{
return(FALSE);
}
}
```

```
if(mext[i] == '*') {
break;
}
if (isalpha(mext[i]) && isalpha(fext[i])) {
char x = isupper(mext[i]) ?
mext[i] : toupper(mext[i]);
char y = isupper(fext[i]) ?
fext[i] : toupper(fext[i]);
if (x != y) return(FALSE);
} else if(mext[i] != fext[i]) {
return(FALSE);
}
}
return(TRUE);
}


int
validate_fcb(char *fcb)
{
int a;
char *c;
 c = fcb;
 if (*c==0)
    {
c++;
if (isalpha(*c))/*PRIMER CARACTER DEL NOMBRE*/
{
  c++;
  for (c=c; c<fcb+11; c++)/*EL RESTO, INCLUYE EXTENSION*/
    if (!(isalpha(*c ))&& !(*c>29 && *c<40))
return(-1);
}
else return(-1);
     }
     else return(-1);
 return(1);
}

int
validate_fcbd(char *fcb)
{
int a;
char *c;
 c   fcb;
 if (*c==0)
    {
c++;
for (a=0;a<8;a++) {
 if ((*c=='?')||(isalpha(*c)))
   {
   c++;
   continue;
          }
        if ((*c=='*')||(*c==' ')
```

```
   break;
          else return(-1);
       }
c=fcb+8;
for (a=0;a<3;a++) {
 if ((*c=='?')||(isalpha(*c)))
   {
   c++;
   continue;
          }
        if ((*c=='*')||(*c==' ')
   break;
          else return(-1);
       }
      return(1);
    }
   else return(-1);
}



int findFATentry()
{
 WORD *ent;
 int i;

 ent = (WORD *)(memfile+OFFSFAT);
 for (i = 0; i<6144; i++)
   if (*ent == 0) break;
else ent++;
 if (i==6144)
   return(-1);
   else *ent=0xfff8;
 return(i);
}

void delContent(int ent)
{
 WORD *w;
 WORD d;

 entr = (Ent_Dir *) (memfile+OFFSDIR+ent*32);
 d = entr->cluster;
 do
{
   w = (WORD *) (memfile+OFFSFAT)+d;
   d   *w;
   *w = 0;
 }while(d!=0xfff8);
}

int find_file(char *nam, char *ex)
{
 int ent,j;
 ent = 0;
```

```
 for (ent = 0; ent<512; ent++)
  {
   entr = (Ent_Dir *) (memfile+OFFSDIR+ent*32);
   for (j=0; j<8; j++)
      if (entr->name[j]!=nam[j]) break;
      if (j<8) continue;
else for(j=0; j<3; j++)
if(entr->ext[j]!=ex[j]) break;
      if (j<3) continue;
         else break;
   }
  if (ent<512) return(ent);
else return(-1);
}

int find_free()
{
 int ent,j;
 ent = 0;
 for (ent = 0; ent<512; ent++)
  {
   entr = (Ent_Dir *) (memfile+OFFSDIR+ent*32);
   if (entr->name[0]==' ')
        break;
  }
  if (ent<512) return(ent);
        else return(-1);
}

unsigned map(int dir)
{
  int ofs;
  int limits;

  ofs = dir*DOS_BLOCK_SIZE;
  if (ofs>=REGION_SIZE)
    return 0;
    else return ofs;
}

/*Esta rutina es invocada cuando la instruccion que genero la excepcion
es algun servicio de la int 21h*/

boolean_t dos_general_fn(state, task)
state_t *state;
task_t task;
{
int fd;
int ret = REDIRECT;
sft_t   sft,algo;
     int i,np,index,k,j;
     char fname[9];
     char fext[4];
     char fpath[256];
     char buf[256];
```

```
     struct dir_ent * tmp;
struct stat st;
        vm_address_t ip2;
unsigned int cnt, ent;
struct dir_ent *file;
        char *ap,*car,*text;
kern_return_t error;
int p,f,d,tam;

sft = (u_char *) Addr(state, es, edi);

algo = (u_char *) Addr(state, ds, edx);
        np = (int) ((int)algo / 8192);
        if (np!=np1)
        {
                ip2 = 8192 * np;
                error = vm_read(task, (vm_address_t)(addr+ip2), vm_page_size,
(pointer_t *)&page, &cnt);
                if (error != KERN_SUCCESS)
                        mach_error("Call to vm_read() failed", error);
                }

        ap   (char *)page;
        index = (int)((int)algo % 8192);


        np1 = np;

switch(HIGH(state->eax)) {
case WRITESTD: /*0x0A*/
   car   ap+index;
   while (*(car)!='$')
    { printf("%c",*car);
      car++;
    }
  break;
case READSTD: /*0x09*/
  d=*(ap+index);
  text = malloc(d);
  scanf("%s",text);
  tam=strlen(text);
  if (tam>d) tam=d;
  car = ap+index+1;
  car++;
  strncpy(car,text,tam);
  free(text);
  break;
case RESET_DISK:        /*0x0D*/
                  printf("Reset Disk\n");
  break;
                case OPEN_FILE_FCB:     /*0x0F*/
                  printf("Open File Using FCB\n");
                  printf("Drive= %x \n",*(ap+index));
 //nombre:
                  for(i=index+1; i<index+9; i++)
```

```
                    fname[i-index-1] = *(ap+i);
                  fname[8]='\0';
 //extension:
                  for(i=index+9; i<index+12; i++)
                   fext[i-index-9] = *(ap+i);
                  fext[3]='\0';
                  printf("Nombre = %s\n", fname);
                  printf("Ext = %s\n", fext);
                  if (validate_fcb(ap+index)==1)
                    { printf("FCB ok\n");
 //  Buscar una entrada para el archivo en el directorio:
                      p = find_file(fname, fext);
                      if (p>-1)
                        {
  nfo++;
                          entr= (Ent_Dir *) (memfile+OFFSDIR+p*32);
                          curr_rec = entr->cluster; /*Bloque inicial del archivo*/
}
                       else {
printf("File not found");
SETLOW(state->eax,0xff);
  }
    }
                  break;
                case CLOSE_FILE_FCB:    /*0x10*/
                  printf("Close File Using FCB\n");
  nfo = -1;
                  break;
                case FIND_FIRST:
 //nombre:
                  for(i=index+1; i<index+9; i++)
                    ffname[i-index-1]   *(ap+i);
                  ffname[8]='\0';
 //extension:
                  for(i=index+9; i<index+12; i++)
                   ffext[i-index-9] = *(ap+i);
                  ffext[3]='\0';

                  if (validate_fcbd(ap+index)==1)
                    {
        entr = (Ent_Dir *) (memfile+OFFSDIR+ent*32);
//Buscar la primera coincidencia:
      for(sig=0; sig<512; sig++)
      {
            if (compare(entr->name, entr->ext, ffname, ffext))
break;
      entr=(Ent_Dir *) (memfile+OFFSDIR+ent*32);
      }
     if (sig<512){
               for (i=0; i<8; i++)
                   {
                       *dta = entr->name[i];
                       dta++;
               }
               for (i=0; i<3; i++)
```

```
                    {
                        *dta = entr->name[i];
                        dta++;
                    }
        }
else SETLOW(state->eax,0xff);
    }
                  else SETLOW(state->eax,0xff);

                  break;
                case FIND_NEXT:
//Buscar la siguiente coincidencia:
    if (sig<512){
      for(j=sig; j<512; j++)
      {
             if (compare(entr->name, entr->ext, ffname, ffext))
break;
      entr=(Ent_Dir *) (memfile+OFFSDIR+ent*32);
      }
      if (j<512){
                for (i=0; i<8; i++)
                    {
                        *dta = entr->name[i];
                        dta++;
                }
                for (i=0; i<3; i++)
                    {
                        *dta = entr->name[i];
                        dta++;
                }
       }
      sig=j++;
    }
                  else SETLOW(state->eax,0xff);

                  break;
                case DELETE_FILE_FCB:  /*0x13*/
                  printf("Delete File Using FCB\n");
  printf("Not implemented\n");
                  break;
                case READ_FILE_FCB:     /*0x14*/
                  printf("Sequential Read Using FCB\n");
                  if (nfo==-1) break;
                  ent = map(curr_rec);
                  if (ent>0)
                  for (i=0; i<DOS_BLOCK_SIZE; i++)
                   {
                      *dta   *(memfile+ent);
                      ent++;
                      dta++;
                   }

                  break;
                case WRITE_FILE_FCB:    /*0x15*/
/* Se supone recien abierto o recien creado el archivo*/
```

```
                printf("Sequential Write Using FCB\n");
                if (nfo==-1) break;
  ent = map(curr_rec);
  if (ent>0)
  for (i=0; i<DOS_BLOCK_SIZE; i++)
    {
*(memfile+ent) = *dta;
ent++;
dta++;
    }

  if (nfo==-1) break;
  /*copiar el contenido de dta al cluster indicado en el
desplazamiento correspondiente al campo registro actual ap+index*/

                break;
               case CREATE_FILE_FCB:  /*0x16*/
/*HAY QUE CREAR UNA ENTRADA DE DIRECTORIO PARA EL ARCHIVO, SACAR
LOS DATOS DEL FCB, INVESTIGAR FECHA Y HORA DEL SISTEMA, COLOCAR
  LA LIGA EN LA FAT*/
                printf("Create File Using FCB\n");
 printf("Drive= %x \n",*(ap+index));
//nombre:
   for(i=index+1; i<index+9; i++)
   fname[i-index-1] = *(ap+i);
 fname[8]='\0';
//extension:
 for(i=index+9; i<index+12; i++)
                fext[i-index-9] = *(ap+i);
        fext[3]='\0';
               printf("Nombre = %s\n", fname);
               printf("Ext = %s\n", fext);
 if (validate_fcb(ap+index)==1)
                 { printf("FCB ok\n");
//  Buscar una entrada para el archivo en el directorio:
     p = find_file(fname, fext);
     if (p>-1)
        {
    printf("Se encontro el archivo en la %da entrada\n",p);
 delContent(p);
       }
       else if (p<0) p=find_free(); /*se busca una entrada libre  de dir.*/
     if (p>-1)
       {
 nfo = p;
                      /* Se construye el FCB*/
                   *(ap+index) = 00; //Drive
                      *(ap+index+14) = 0x20; //Atributo
                      *(ap+index+20) = 00; //fecha
                      *(ap+index+21) = 0xf0; //fecha
                      *(ap+index+22) = 0x26; //hora
                      *(ap+index+23) = 0xd6; //hora
//  Llenar una estructura entrada de directorio
printf("Se encontro espacio libre para el archivo\n");
              entr= (Ent_Dir *) (memfile+OFFSDIR+p*32);
```

```
strcpy(entr->name, fname);
strcpy(entr->ext, fext);
entr->attrib   (BYTE) 20;
entr->size = (DWORD) 0;
/*falta llenar fecha y hora*/
/*  Hallar espacio libre en la FAT*/
f = findFATentry();
if (f>0) {
   entr->cluster = f; /*Bloque inicial del archivo, esto  debe colocarse en el FCB*/
   curr_rec   f;
   nfo ++;
   }
   else printf("disk full");

//escribir el sector......
}
else printf("No more entry at root directory\n");
     }
else printf("FCB not ok\n");
                break;
case SET_DTA: /* 0x1a */
dta = ap+index;
 break;
case REMOVE_DIRECTORY: /* 0x3a */
printf("Remove Directory\n");
break;
case MAKE_DIRECTORY: /* 0x39 */
printf("Make Directory\n");
break;
case SET_CURRENT_DIRECTORY: /* 0x3b */
return(TRUE);
break;
case CLOSE_FILE: /* 0x3e */
printf("Close file %x\n",fd);
  return(TRUE);
break;
case DELETE_FILE: /* 0x41 */
printf("Delete file\n");
return(TRUE);
case UNDOCUMENTED_FUNCTION_2:
printf("Undocumented function: %x\n",
LOW(state->eax));
return(TRUE);
     default:
printf("Undocumented function: %x\n",
HIGH(state->eax));
return(REDIRECT);
}
return(ret);
}
```

```
/*************************************************************
Monitor.c  Programa monitor V86
*************************************************************/
#import "bios_profile.h"
#import "type.h"
#import "base.h"
#import "def.h"
#import "bios.h"
#import "int21_names.h"
#import <string.h>
#import <stdio.h>
#import <fcntl.h>
#import <mach/mach_init.h>
#import <mach/mach.h>

#define NUM_PAGES 3
#define REGION_SIZE (NUM_PAGES * vm_page_size)

#define DOS_BLOCK_SIZE 0x200
#define BIOS_ADDR_MAX 0x100000
#define USER_ADDR_MAX 0x0a0000
#define vm_page_size  8192
#define EXC_TYPES 7

char *exception_names[6] = {
        "EXC_BAD_ACCESS",
        "EXC_BAD_INSTRUCTION",
        "EXC_ARITHMETIC",
        "EXC_EMULATION",
        "EXC_SOFTWARE",
        "EXC_BREAKPOINT" };


#define EXC_MSG_SIZE_MAX 1024

vm_address_t addr;
thread_t thread86;
int total_exceptions_processed = 0;
vm_address_t page;
int np1=-1;
int    terminate = 0;
int    nfo = -1; /*entrada del ultimo archivo abierto*/
char * dos_root = "/Users/ncasas/droot/";
int dos_root_len;

block Boot_rec;
block *Boot;


simulate_interrupt(state, interrupt, incr_eip)
state_t *state;
int interrupt;
int incr_eip;
{
```

```
        u_short * sp;
        u_short cs = idt[interrupt].selector;
        u_short eip = idt[interrupt].offset;

        if (Addr_8086(cs,eip) < 0x400) {
                printf("Simulate interrupt bad: %x:%x\n", cs, eip);
                return;
        }

        printf("Simulate interrupt: 0x%x\n", interrupt);
        state->eip += incr_eip;
        sp   (u_short *)Addr(state, ss, esp);
        *--sp   (state->eflags & EFL_SAFE);
        *--sp = state->cs;
        *--sp   state->eip;
        state->esp -= 6;
        state->cs = idt[interrupt].selector;
        state->eip = idt[interrupt].offset;
        state->eflags &= ~(EFL_IF | EFL_TF);
        printf("mon: CALLING 0x%x\n",Addr(state,cs,eip));
}

void do_int(eip, state,inst, task )
u_char *eip;
state_t *state;
unsigned char *inst;
task_t task;

{
 boolean_t ret;
 int interrupt_number;
 Ent_Dir *endir;

 state->eip++;
 inst++;
 interrupt_number = *inst;
 if ((interrupt_number == DOS_GENERAL) && (HIGH(state->eax) < DOS21_NAMES ))
   { printf("Interrupt--21, ax = %x, bx = %x, cx = %x, dx = %x, ds   %x,
     eip   0x%x:%x '%s'\n",WORD(state->eax), WORD(state->ebx),
     WORD(state->ecx),WORD(state->edx), WORD(state->ds), state->cs,
                 (state->eip -1), names21[HIGH(state->eax)]);
   }
   else if ((interrupt_number == 0x13) && (HIGH(state->eax) < 0x1b)) {
           printf("Interrupt++13, ax   %x, bx = %x, cx = %x, dx = %x,
                  eip = 0x%x:%x '%s'\n", WORD(state->eax), WORD(state->ebx),
                  WORD(state->ecx), WORD(state->edx), state->cs,
                  (state->eip -1), bios_13_names[HIGH(state->eax)]);
           }
       else{
            printf("Interrupt--%x, ax = %x, bx   %x, cx = %x, dx = %x, ds = %x
              es = %x eip = 0x%x:%x\n", interrupt_number, WORD(state->eax),
              WORD(state->ebx), WORD(state->ecx), WORD(state->edx),RD(state->ds),
              WORD(state->es), state->cs, (state->eip -1));
           }
 switch (*inst) {
```

```
   case INT_01:
              printf("mon: int 01 occurred.\n");
              printf("void)do_breakpoint(eip, state)");
              break;
   case TERMINATE:
              endir = (Ent_Dir *) (memfile+OFFSDIR);
              write(disk,memfile,32768);
              printf("Programa terminated normally\n");
              thread_terminate(thread86);
              close(disk);
              terminate = 1;
              break;
              case (char)MACH_EXIT:
              break;
   case (char)MACH_FMD:
              exit(0);
              break;
   case (char)MACH_MOUSE:
              printf("mouse interrupt");
              break;
   case (char)BIOS_VIDEO:
              printf("bios video fn");
               break;
   case BIOS_DISK:
              printf("bios disk fn");
              break;
   case DOS_GENERAL:
              ret = dos_general_fn(state,task);
              break;
   default: {
              redirect:
              printf("simulate_interrupt(state, *inst, 1)");
              return;
             }
  }
 if (ret == LEAVE_EIP_ALONE)
   return;
 if (ret != UNCHANGED)
    state->eflags = (ret ? (state->eflags&~EFL_CF) : (state->eflags|EFL_CF));

}


void do_bad_instruction (eip, state,inst,task)
  u_char *eip;
  state_t *state;
  char *inst;
  task_t task;
{
   boolean_t operand_size_override = FALSE;
   restart:
   switch (*inst) {
    case (char)I386_INT:
         do_int(eip, state,inst,task);
         if (terminate==1) ;/*return;*/
```

```
        break;
    case (char)I386_CLI:
        printf("mon: CLI\n");
        printf( "cs 0x%x eip 0x%x fl 0x%x\n",
        state->cs, state->eip, state->eflags);
        #ifdef NOP_CLI_STI
        /* make cli a nop */
        if (eip <= (u_char *) USER_ADDR_MAX)
        *inst = I386_NOP;
        #endif NOP_CLI_STI
        /* step over it */
        state->eip++;
        break;
    case (char)I386_HLT:
        printf("mon: HLT\n");
        /* step over it */
        state->eip++;
        break;
    case (char)I386_STI:
        printf( "mon: STI\n");
        printf( "cs 0x%x eip 0x%x fl 0x%x\n",
        state->cs, state->eip, state->eflags);
        #ifdef NOP_CLI_STI
        /* make sti a nop */
        if (eip <= (u_char *) USER_ADDR_MAX)
        *inst = I386_NOP;
        #endif NOP_CLI_STI
        /* step over it */
        state->eip++;
        break;
    case (char)I386_OP_SIZE:
        operand_size_override = TRUE;
        state->eip++;
        inst++;
        goto restart;
    case (char)I386_PUSHF:
        printf( "pushf cs 0x%x eip 0x%x fl 0x%x\n",
               state->cs, state->eip, state->eflags);
        if (operand_size_override)
          printf("do_pushf_32(eip, state)\n");
          else printf("pushf (eip, state)\n");
        break;
    case (char)I386_POPF:
        printf( "popf cs 0x%x eip 0x%x fl 0x%x\n",
        state->cs, state->eip, state->eflags);
        if (operand_size_override)
         printf("popf_32(eip, state)\n");
         else printf("popf(eip, state)\n");
        break;
    case (char)I386_LOCK:
        printf("\nmon: LOCK\n");
        state->eip++;
        break;
    case (char)I386_IRET:
        printf("iret (eip, state)\n");
```

```
        break;
    case (char)I386_IN1:
    case (char)I386_IN2:
    case (char)I386_IN3:
    case (char)I386_IN4:
    case (char)I386_INS1:
    case (char)I386_INS2:
        printf("in(eip, state)\n");
        break;
    case (char)I386_OUT1:
    case (char)I386_OUT2:
    case (char)I386_OUT3:
    case (char)I386_OUT4:
    case (char)I386_OUTS1:
    case (char)I386_OUTS2:
        printf("out(eip, state)\n");
        break;
    case (char)I386_REPZ:
        printf("repz(eip, state)\n");
        break;
    default: {
    char outbuf[1024];
    int stepover;
    printf("\nmon: REAL BAD instruction=0x%x %x %x\n",
    *inst, *(inst+1), *(inst+2));
    state->eip += stepover;
    printf("mon: pausing before exiting do_bad_instruction.");
    break;
    }
  }
}

kern_return_t catch_exception_raise (port, thread, task,
     exception, code, subcode)
    port_t port;
    thread_t thread;
    task_t task;
    int exception, code, subcode;
{
  char *ap;
  int state_count;
  state_t state;
  unsigned char *eip;
  int np,index,i;
  vm_address_t ip2;
  unsigned int cnt;
  kern_return_t error;

 state_count = i386_THREAD_STATE_COUNT;
 thread_get_state (thread, i386_THREAD_STATE,
                   (thread_state_t)(&state), &state_count);

 eip = (unsigned char *)Addr(&state, cs, eip);

/*VAMOS  A LEER LA PAGINA DONDE SE ENCUENTRA LA
```

```
 INSTRUCCION QUE GENERO LA EXCEPCION*/

np = (int) ((int)eip / 8192);
if (np!=np1)
{
  ip2 = 8192 * np;
  error = vm_read(task, (vm_address_t)(addr+ip2), vm_page_size,
                  (pointer_t *)&page, &cnt);
  if (error != KERN_SUCCESS)
    mach_error("Call to vm_read() failed", error);
    else printf("Bytes leidos: %d\n",cnt);
 }
 ap = (char *)page;
 index = (int)((int)eip % 8192);

 np1 = np;


printf ("instruccion: %x\n",*(ap+index));
switch (exception) {
 case EXC_BAD_INSTRUCTION:
          do_bad_instruction(eip,&state,(ap+index),task);
          if (terminate == 1) break;
            state_eip++;
          error   thread_set_state ( thread, i386_THREAD_STATE,
   thread_state_t)&state, state_count);
          break;
 case EXC_BREAKPOINT:
        printf("do_breakpoint IP=%x\n",eip);
        thread_set_state ( thread, i386_THREAD_STATE,
        (thread_state_t)&state, state_count);
        break;
 case EXC_BAD_ACCESS:
        printf("do_bad_access IP=%x\n",eip);
        thread_set_state ( thread, i386_THREAD_STATE,
        (thread_state_t)&state, state_count);
        break;
 case EXC_ARITHMETIC:
        printf("do_arithmetic (eip, &state, code, subcode)\n");
        thread_set_state ( thread, i386_THREAD_STATE,
        (thread_state_t)&state, state_count);
        break;
 default:
        printf("mon: pausing before exiting catch_exception_raise");
        if (exception < EXC_TYPES)
          printf("\nmon: %s code   0x%x subcode = 0x%x\n",
              exception_names[exception-1], code, subcode);
  else
           printf("\nmon: exception = 0x%x code = 0x%x subcode = 0x%x\n",
                    exception, code, subcode);
         printf("mon: instruction - 0x%8.8x %x %x\n",*eip, *(eip+1),
                *(eip+2));
         break;
 }
 return(KERN_SUCCESS);
```

```
}

/*
 *  A bad instruction GP fault occurred.
 *  Determine if it should be emulated.
 */

void process_exceptions (exc_port )   /*TH_EXC_HANDLER DE EXC3.C*/
port_t exc_port;

{
  kern_return_t ret;
  msg_return_t  mret;

  struct msg_rcv {
  msg_header_t    rcv_hdr;
  int             data[EXC_MSG_SIZE_MAX - sizeof (msg_header_t)];
} exc_msg;


 struct mgrep {
                msg_header_t head;
                int msg_data[1*2]; /*1 type, item pair*/
        } rep_msg;

        exc_msg.rcv_hdr.msg_local_port = exc_port;
        exc_msg.rcv_hdr.msg_size = sizeof(exc_msg);

        mret = msg_receive(&exc_msg.rcv_hdr, MSG_OPTION_NONE, 0);
if (mret != KERN_SUCCESS) {
        mach_error("process_exception: msg_receive failed", mret);
        exit(1);
}

  while (TRUE) {
  (void) exc_server (&exc_msg, &rep_msg);
   mret = msg_send(&rep_msg.head, MSG_OPTION_NONE, 0);
   if (terminate == 1) break;/*return*/
   mret = msg_receive(&exc_msg.rcv_hdr, MSG_OPTION_NONE, 0);

   total_exceptions_processed++;
   printf("Total exception processed: %d\n",total_exceptions_processed);
  }
}


virtual_thread86(vm_address_t entry_point,thread_t *hilo,
                 port_t *except_port, task_t *child_t)
{
  int                 vm_state_count;
  i386_thread_state_t  vm_thread_state;
  kern_return_t        error;
  thread_t  thread86;

  error =  thread_create(*child_t, &thread86);
```

```
 if (error != KERN_SUCCESS)
   mach_error("thread_create del hilo ",error);

 error   port_allocate(task_self(), &(* except_port));
 if (error != KERN_SUCCESS) {
   mach_error("exc3: port_allocate failed",error);
   exit(1);
   }

 error = thread_set_exception_port(thread86,*except_port);
 if (error != KERN_SUCCESS) {
     mach_error("exc3: port_allocate failed",error);
     exit(1);
    }

 vm_state_count   i386_THREAD_STATE_COUNT;

 error = thread_get_state(thread86, i386_THREAD_STATE,
          (thread_state_t)(&vm_thread_state), &vm_state_count);
 if (error!=KERN_SUCCESS)
   mach_error("Error en thread_get_state", error);

//poner al procesador en modo virtual

 vm_thread_state.eflags  =(vm_thread_state.eflags | EFL_VM) ;

 vm_thread_state.cs  = 0x00;
 vm_thread_state.ds  = 0x00;
 vm_thread_state.ss  = 0x00;
 vm_thread_state.gs  = 0x00;
 vm_thread_state.es  = 0x00;
 vm_thread_state.ebp = 0x00;
 vm_thread_state.eax = 0x00;
 vm_thread_state.ebx = 0x00;
 vm_thread_state.ecx = 0x00;
 vm_thread_state.edx = 0x00;
 vm_thread_state.edi = 0x00;
 vm_thread_state.esi = 0x00;
 vm_thread_state.esp = 0x00;

 vm_thread_state.eip = entry_point;

 vm_thread_state.esp = (unsigned int)(8000);

//lo dejamos trabajando en modo virtual
 error  = thread_set_state(thread86, i386_THREAD_STATE,
          (thread_state_t)(&vm_thread_state), vm_state_count);
 if(error!=KERN_SUCCESS)
   mach_error("Fallo el llamado a thread_set_state", error);

 error = thread_get_state(thread86, i386_THREAD_STATE,
          (thread_state_t)(&vm_thread_state), &vm_state_count );
 if (error!=KERN_SUCCESS)
    mach_error("Error en  thread_get_state", error);
```

```
  printf("\n CS=%x EIP=%x ESP=%x STACK=%x Hilo=%x\n",
         vm_thread_state.cs, vm_thread_state.eip, vm_thread_state.esp,
         vm_thread_state.ss, thread86);

  *hilo = thread86 ;
 }


main()
{
   char c;
port_t   except_port;
char  filename[11];
  task_t child_task;
        vm_address_t buffer,load_address;
        kern_return_t  error,r;             /* tipo de error */
        int i;
char *ap;
        vm_address_t entry_point;
        struct task_basic_info    info;
        char vdisk[]="HARDDISK.HDF";
        unsigned int info_count=TASK_BASIC_INFO_COUNT;
int fd;
        unsigned int  data_cnt;


        dos_root_len = strlen(dos_root);

//1: Se crea una tarea donde poner el hilo en modo virtual
 ejecutando el programa .com
        error=task_create(task_self(), FALSE, &child_task);
        if(error!=KERN_SUCCESS)
            mach_error("Call to task_create() failed", error);

//2: Se asigna un espacio de direcciones de memoria a la tarea
 hija, donde cargar el .com
addr = 0;
        error = vm_allocate(child_task, &addr, BIOS_ADDR_MAX, TRUE);

        if(error!=KERN_SUCCESS)
             mach_error("Call to vm_allocate() failed", error);

        error   vm_allocate(task_self(), &buffer, vm_page_size, TRUE);
        if(error!=KERN_SUCCESS)
             mach_error("Call to vm_allocate() failed". error);

//3: Abrimos el archivo y lo cargamos en el espacio de la tarea hija
fd = -1;
while(fd<0)
{
printf("C>");
scanf("%s",&filename);
strcpy(filename,strcat(filename, ".com"));
        if ((fd = open(filename, O_RDONLY)) < 0)
            printf("Command not found:'%s'\n",filename);
```

```
}
  //Se lee la primera pagina del archivo .com a cargar en memoria
   i = read(fd,(char *)(buffer + 256), 7936);
  //se escribe a la memoria
   error   vm_write(child_task, addr, buffer, vm_page_size);
   if (error != KERN_SUCCESS)
           mach_error("Call to vm_write() failed", error);

load_address =addr +8192;

//se carga el resto del archivo
        while (i>=7936){
          i    read(fd,(char *)buffer, 8192);
  error = vm_write(child_task, load_address, buffer, vm_page_size);
  if (error != KERN_SUCCESS)
mach_error("Call to vm_write() failed", error);
          load_address = +8192;
         }

//se cierra el archivo
        if (close (fd) < 0) {
            printf("\rmon: disk boot close error, exiting.");
            exit(1);
           }

//el IP
  entry_point = (vm_address_t) 0x100 ;

 //se crea el hilo virtual
  virtual_thread86(entry_point, &thread86, &except_port, &child_task);

//Mapeamos 32k del disco virtual en memoria
  disk = open(vdisk,O_RDWR);

  /* Map part of it into memory. */
  r = map_fd(disk, (vm_offset_t)0,(vm_offset_t *)&memfile,
             TRUE, (vm_size_t)32768);
//Estas cuatro paginas contienen al directorio en su totalidad

  //Se mapean 4 paginas de memoria virtual, 32k
  if (r != KERN_SUCCESS)
      mach_error("Error calling map_fd()", r);

//4: Ejecutar  el hilo

  thread_resume(thread86);

  process_exceptions(except_port);

  exit(0);
 }
```

Apéndice B

Programas Ejemplo

```
;CREA.ASM: Este programa demuestra la creación de archivos mediante
;el uso de las siguientes funciones de la interrupcion 21h:
;servicio 09h, para escribir cadenas desde la salida estandar
;servicio 0ah, para leer cadenas desde la entrada estandar
;servicio 16h, para crear un archivo usando FCB.

jmp ini
let1 db "Nombre: $"
let2 db "Extensin: $"
nn   db 9
nam  db 0
     db 8 DUP(0)
ee   db 4
ext  db 0
     db 4 DUP(0)

fcb db 0
n   db 8 DUP(0)
e   db 3 DUP(0)
    db 31 DUP(0)

ini:
push cs
pop ds
```

```
mov dx,offset let1
mov ah,09h
int 21h

mov dx,offset nn
mov ah,0ah
mov al,8
int 21h

mov di,offset n
mov si,offset nam
inc si
mov ch,8
ciclo:
mov [di],[si]
inc si
inc di
dec ch
jnz ciclo

mov dx,offset let2
mov ah,09h
int 21h

mov dx,offset ee
mov ah,0ah
mov al,3
int 21h

mov di,offset e
mov si,offset ext
inc si
mov ch,3
ciclo2:
mov [di],[si]
inc si
```

```
inc di
dec ch
jnz ciclo2

mov dx,offset fcb
mov ah,16h
int 21h
int 20h


;ESCRIBE.ASM: Este programa demuestra la apertura y escritura a un archivo
;existente, haciendo uso de las siguientes funciones de la interrupcion 21h:
;servicio 09h, para escribir cadenas desde la salida estandar
;servicio 0ah, para leer cadenas desde la entrada estandar
;servicio 0Fh, para abrir un archivo
;servicio 15h, para escribir a un archivo usando FCB.

jmp ini
fcb db 0
    db "nelida  "
    db "pru"
    db 31 DUP(0)
dta db " hola mundo hola mundo hola mundo hola mundo hola mundo hola mundo hola mu

ini:
        push cs
        pop ds
        mov dx,offset fcb
        mov ah,0fh
        int 21h    ;se abre
        mov ah,1ah
        mov dx,offset dta
        int 21h     ;se fija el area de dta
        mov dx,offset fcb
        mov ah,15h
        int 21h     ;se escribe lo que esta en la dta
        int 20h
```

```
;TYPE.ASM: Este programa demuestra las operaciones de apertura y lectura a un
;archivo existente, haciendo uso de las siguientes funciones de la interrup-
;cion 21h:
;servicio 09h, para escribir cadenas desde la salida estandar
;servicio 0ah, para leer cadenas desde la entrada estandar
;servicio 16h, para crear un archivo usando FCB.

jmp ini
fcb db 0
    db "nelida  "
    db "pru"
    db 31 DUP(0)
dta db 512 DUP(0)

ini:
        push cs
        pop ds
        mov dx,offset fcb
        mov ah,0fh
        int 21h     ;se abre
        mov ah,1ah
        mov dx,offset dta
        int 21h      ;se fija el area de dta
        mov dx,offset fcb
        mov ah,14h
        int 21h      ;se lee un registro del archivo
        mov si,80
        mov dta[si],'$'
        mov dx,offset dta
        mov ah,09
        int 21h
        int 20h


;DIR.ASM: Este programa realiza el despliegue del directorio mediante
;el uso de las siguientes funciones de la interrupcion 21h:
```

```
;servicio 09h, para escribir cadenas a la salida estandar
;servicio 1ah, para fijar el area de DTA
;servicio 11h, para buscar la primera ocurrencia en el directorio.
;servicio 12h, para buscar la siguiente ocurrencia en el directorio.

jmp ini

fcb db 0
    db 8 DUP('?')
    db 3 DUP('?')
    db 31 DUP(0)
dta db 512 DUP(0)

head db " Directory of C:\"
     db 1 DUP(0ah)
     db 1 DUP(0dh)
     db 1 DUP('$')

totf1 db 8 DUP(0)
totf2 db "     File(s) $"

dat  db 1 DUP(0)


nfiles:
     push si
     push ax
     push bx

;convierte el numero a hexadecimal:
     xor ax,ax
     mov al,ch
     mov bl,0ah
     div bl
     mov ah,0
     mov bl,6
     mul bl
```

```
     add ch,al

;guarda el primer digito en la cadena:
     mov ah,ch
     and ah,0f0h
     shr ah,4
     add ah,30h
     mov si,0
     mov totf2[si],ah

;guarda el segundo digito en la cadena:
     mov ah,ch
     and ah,0fh
     add ah,30h
     inc si
     mov totf2[si],ah

     pop bx
     pop ax
     pop si
     ret

ini:

   mov ah,1ah
   mov dx,offset dta
   int 21h      ;se fija el area de dta

   push cs
   pop ds
   mov dx,offset fcb
   mov ah,11h
   int 21h     ;se busca la primera ocurrencia

   mov dx,offset head
   mov ah,09
   int 21h    ;se imprime el encabezado
```

```
   mov ch,0

ciclo:
   inc ch
   push cx
   mov si,12
   mov dta[si],0dh
   inc si
   mov dta[si],0ah
   inc si
   mov dta[si],'$'
   mov dx,offset dta+1
   mov ah,09
   int 21h     ;se imprime la informacion de la dta

   push cs
   pop ds
   mov dx,offset fcb
   mov ah,12h
   int 21h     ;se busca la siguiente ocurrencia
   cmp al,0ffh
   pop cx
   jnz ciclo


   call nfiles  ;se convierte a hex. el numero de archivos
   mov dx,offset totf1
   mov ah,09
   int 21h      ;se imprime el no. de archivos

   int 20h

;COPY.ASM: Este programa realiza la copia de un archivo fuente a un destino
;usando las siguientes funciones de la interrupcion 21h:
;servicio 09h, para escribir cadenas a la salida estandar
;servicio 16h, para crear
```

```
;servicio 10h, para leer
;servicio 0Fh, para abrir un archivo ya existente
;servicio 15h, para escribir a un archivo
;servicio 1ah, para fijar el area de DTA
;servicio 11h, para buscar la primera ocurrencia en el directorio.
;servicio 12h, para buscar la siguiente ocurrencia en el directorio.

jmp ini
let1 db "Nombre $"
let2 db "Extensin $"
let3 db "Fuente:"
     db 10
     db 13
     db "$"
let4 db "Destino:"
     db 10
     db 13
     db "$"

nn   db 9
nam  db 0
     db 8 DUP(0)
ee   db 4
ext  db 0
     db 3 DUP(0)

fcb db 0
n   db 8 DUP(0)
e   db 3 DUP(0)
    db 31 DUP(0)

fcbd db 0
nd  db 8 DUP(0)
ed  db 3 DUP(0)
    db 31 DUP(0)

dta db 512 DUP(0)
```

```
ini:
mov dx,offset let3  ;Fuente
mov ah,09h
int 21h

mov dx,offset let1  ;pide el nombre
mov ah,09h
int 21h

mov dx,offset nn    ;lee el nombre
mov ah,0ah
mov al,8
int 21h

mov di,offset n    ;lo copia al fcb
mov si,offset nam
mov ch,nam
inc si
dec ch
ciclo:
mov [di],[si]
inc si
inc di
dec ch
jnz ciclo

mov dx,offset let2  ;pide extension
mov ah,09h
int 21h

mov dx,offset ee    ;lee extension
mov ah,0ah
mov al,3
int 21h

mov di,offset e    ;la copia al fcb
```

```
mov si,offset ext
mov ch,ext
dec ch
inc si
ciclo2:
mov [di],[si]
inc si
inc di
dec ch
jnz ciclo2

mov dx,offset let4  ;Destino
mov ah,09h
int 21h

mov dx,offset let1  ;pide el nombre
mov ah,09h
int 21h

mov dx,offset nn    ;lee el nombre
mov ah,0ah
mov al,8
int 21h

mov di,offset nd    ;lo copia al fcb
mov si,offset nam
mov ch,nam
dec ch
inc si
ciclo3:
mov [di],[si]
inc si
inc di
dec ch
jnz ciclo3

mov dx,offset let2  ;pide extension
```

```
mov ah,09h
int 21h

mov dx,offset ee     ;lee extension
mov ah,0ah
mov al,3
int 21h

mov di,offset ed   ;la copia al fcb
mov si,offset ext
mov ch,ext
inc si
dec ch
ciclo4:
mov [di],[si]
inc si
inc di
dec ch
jnz ciclo4
mov di,offset fcb
mov ah,0
mov [di],ah

push cs
pop ds
mov dx,offset fcb
mov ah,0fh
int 21h    ;se abre
mov ah,1ah
mov dx,offset dta
int 21h     ;se fija el area de dta

mov dx,offset fcb
mov ah,14h
int 21h     ;se lee un registro del archivo

mov dx,offset fcbd  ;se crea el archivo destino
```

```
mov ah,16h
int 21h
mov ah,1ah
mov dx,offset dta
int 21h      ;se fija el area de dta
mov dx,offset fcbd
mov ah,15h
int 21h      ;se escribe lo que esta en la dta
int 20h
```

Bibliografía

[APPLE 01] *Apple Computer, Inc.* http://www.apple.com/enterprise/

[BOYKIN 93] J. Boykin, David Kirschen, Alan Langerman, Susan Loverso. *Programming Under Mach.* Addison-Wesley Publishing Company. 1993. Pag. 5-6, 8-9, 14-33, 63-70, 99-143

[BUENABAD 89] Buenabad Chávez, Jorge. *XINIX: Sistema Operativo para Computadora Personal.* Tesis M.C. Centro de Investigación y Estudios Avanzados del IPN. Departamento de Ingeniería Eléctrica, Sección de Computación. México, 1989.

[COULOURIS 94] Coulouris, George., Follimore, Jean and Kindberg, Tim. *Distributed Systems. Concepts and Design.* Addison-Wesley Publishing Company. 1994.

[FINKEL 86] Raphael A. Finkel. *An Operating Systems Vade Mecum.* Prentice Hall. 1986. Pag. 20-23.

[FORIN 93] Alessandro Forin, Gerald R. Malan. *An MS-DOS File System for UNIX.* Proceedings of USENIX Conference, January 17-21, 1994.

[IEEE 96] *IEEE Standards Interpretations for IEEE Std 1003.1-1990 by the Institute of Electrical and Electronics Engineers, Inc.* 345 East 47th Street New York, New York 10017 USA, 1996. http://standards.ieee.org/reading/ieee/interp/1003-1-90_int/index.html

[INTEL 90] Intel Corporation *i486 Microprocessor Programmer's Referencce Manual* Osborne McGraw-Hill. 1990.

[JAMSA 94] Kris Jamsa. *Programación en DOS. Manual de Referencia.* Osborne McGraw-Hill. 1994.

[LOYND 92] Kerry Loynd. *Mixing Real- and Protected-Mode Code.* Dr. Dobb's Journal, February 1992.

[MACKINNON 79] R. A. MacKinnon. *The changing virtual machine environment: Interfaces to real hardware, virtual hardware, and other virtual machines.* IBM System Journal, Vol. 18, No. 1, 1979.

[MALAN 91] Gerald Malan, Richard Rashid, David Golub, Robert Baron. *DOS as a Mach 3.0 Application.* Proceedings of the Usenix Mach Symposium, November 1991.

[MENDEL 94] Mendel Rosenblum, Mani Varadarajan. *SimOS: A Fast Operating System Simulation Enviroment.* Technical Report: CSL-TR-94-631, July 1994.

[PADOVANO 93] Michael Padovano. *Networking Applications on UNIX. System V.* Prentice-Hall, Inc. 1993.

[RASHID 87] Richard Rashid et. al. *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.* Proceedings of the second International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS-II. ACM Press, October 1987. Pag. 31-39

[REDHAT 99] *Redhat support.* http://www.redhat.com/support/docs/samba.html

[SILBERSCHATZ 98] Abraham Silberschatz y Peter Galvin. *Operating Systems Concepts.* 5th Edition. Addison Wesley. 1998. Pag. 74-78, 745, 747-748, 763-764.

[SIMSON 93] Simson L. Garfinkel, Michael K. Mahoney. *NeXSTEP Programming STEP ONE: Object-Oriented Applications.* Springer-Verlag Inc. 1993.

[STEVENS 95] W. Richard Stevens, *UNIX Network Programming.* Prentice-Hall, Inc. 1995.

[SUN 01] *Sun Microsystems, Inc.* 901 San Antonio Road, Palo Alto, CA 94303 USA. http://www.sun.com/staroffice/

[STUART 74] Stuart E. Madnick and John J. Donovan. *Operating Systems.* McGraw-Hill Book Company. 1974. Pag. 549, 552-553.

[TANENBAUM 92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall. 1992.

[TANENBAUM 94] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, 1988.

[TANENBAUM 95] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice-Hall. 1995. Pag. 501-502, 517-518.

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará la **Lic**. **Nélida Alicia Casas Reyes**, declaramos que hemos revisado la tesis titulada:

"Emulación Prototipo de Sistemas DOS/8086"

y consideramos que cumple con los requisitos para obtener el grado de Maestra en Ciencias, con especialidad en Ingeniería Eléctrica.

Atentamente,

Dr. Jorge Buenabad Chávez

Dr. Arturo Díaz Pérez

Dr. José Oscar Olmedo Aguírre