DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Prototipo de Sistema Operativo

Informe

Organización del Computador II - Trabajo Práctico Final
Primer Cuatriméstre de 2013

Integrante	LU	Correo electrónico
Juan Pablo Darago	272/10	jpdarago@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

> I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones.
>
> Linus Torvalds, anunciando Linux por `news:comp.os.minix`

El siguiente trabajo se presenta como complemento y documentación del trabajo práctico final realizado para la materia Organización del Computador II.

El trabajo práctico final fue realizado sobre el tema de programación de sistemas operativos sobre la arquitectura Intel IA-32, correspondiente a la segunda parte de la cursada de la materia en el segundo cuatrimestre de 2011.

El propósito de este informe es detallar las decisiones de diseño y los problemas y soluciones encontrados durante la implementación del prototipo de Sistema Operativo que se incluye con este informe.

2. Alcance y propósito del trabajo

Se buscó implementar un Sistema Operativo monousuario y multitarea, utilizando para ello las facilidades para el manejo de tareas en nivel de usuario y demás recursos que se hacen disponibles mediante la arquitectura IA - 32.

Se buscó por sobre todo implementar la infraestructura necesaria para poder ejecutar tareas de nivel de usuario leidas desde un disco duro con sistema de archivos. Por lo tanto se priorizó sobre todo la simplicidad de los algoritmos y técnicas utilizados. Los principales objetivos del trabajo se detallan a continuación:

- Lograr que cada tarea corra en un espacio de memoria virtual, bajo la ilusión de que dispone de la memoria de la computadora para ella sola.
- Lograr que el Sistema Operativo corra en un nivel de privilegio distinto al de las tareas, y que estas puedan acceder al Sistema Operativo solamente mediante una interfaz clara.
- Lograr que el desarrollador a nivel de usuario (en este caso, el autor mismo) pudiera programar utilizando un conjunto de librerías acorde a la experiencia de desarrollo en nivel de usuario en Linux o Windows. Por ello, se priorizó lograr una compilación limpia con GCC, para poder programar en C, y la interpretación de binarios ejecutables ELF tan trasparente como fuese posible, para permitir la programación y testeo de las librerías por separado al código de tarea (y su integración mediante linkeo estático con un linker).
- Lograr una infraestructura de sistema de archivos que permita no solo archivos en disco duro sino que también permita manejar como archivos drivers como por ejemplo el driver de terminal.
- Lograr que se puedan disparar tareas mediante el uso de una consola de comandos. Esta debe correr con privilegio de usuario (no de Sistema Operativo).

En particular, se tomaron algunas decisiones pragmáticas para permitir cumplir estos objetivos en un tiempo de desarrollo razonable.

- Se decidió utilizar un kernel no preempteable, para evitar la aparición de posibles problemas de sincronízación en el manejo de estructuras claves de sistema operativo. Sin embargo, se

implementó una manera de que el sistema operativo libere el uso del procesador a la siguiente tarea en casos donde esto es relevante (por ejemplo, a la espera de una interrupcción de teclado).

- Se decidió no utilizar métodos de I/O no bloqueantes, utilizandose polling para acceso a disco (esto se detalla más adelante en la sección sobre ATA PIO (Sección 8)). Para evitar problemas de sincronización con respecto al uso de los inodos de disco duro, se realiza busy-waiting: El Kernel espera a que el disco duro termine la operación que se esta realizando. Más alla del pragmatismo de implementación, existe también otra motivación que consiste en que el uso de ATA PIO requiere el uso de ciclos de CPU para leer los puertos de entrada. Por lo tanto, se considero que liberar el CPU por el tiempo de espera de movimiento del disco duro es insignificante al lado del costo computacional que insume el uso de Programmed I/O para leer o escribir los datos al disco.
- No se consideró la configuración de los distintos hardwares de la computadora, asumiéndose defaults razonables que permitieran verificar la correctitud de los algoritmos. Si se verifico la existencia del hardware asumido y se realizan etapas de verificación para asegurar el correcto funcionamiento de los algoritmos posteriores. Por ejemplo se asume un solo disco ATA Master y no se verifica la existencia de otros.
- Muchos de los algoritmos implementados corresponden a implementaciones de algoritmos posiblemente subóptimos. Sin embargo, se buscó una clara separación de los módulos del Sistema Operativo para permitir el reemplazo de estas versiones por versiones más óptimas (Véase la sección 12).

3. Alcance y propósito de este informe

En este informe se detallaran aquellos aspectos del trabajo práctico que exceden los temas vistos en la materia. En particular, se asumirá que el lector esta familiarizado con los aspectos fundamentales de la arquitectura IA 32 como son detallados por el programa de la materia Organización del Computador II dictada en la Universidad de Buenos Aires, conoce los lenguajes de programación C y ensamblador, y esta familiarizado con las herramientas de desarrollo disponibles en sistemas operativos basados en UNIX, en particular con los conceptos de linker, script de linker, compilador, sistemas jerárquicos de archivos (su interfaz de usuario, no su implementación), lenguajes de scripting y herramientas de montado de sistemas de archivos.

4. Consideraciones generales

El Sistema Operativo implementado consiste de un conjunto de módulos, que se explicarán por separado en una sección para cada uno. En esta sección se detalla en general cada uno de los módulos, y se señalan que módulos no se explican y el motivo de ello.

- Bootloader: Para este trabajo, se decidió utilizar el bootloader GRUB, y por lo tanto se implementó la especificación *multiboot*. GRUB en particular nos permite varias cosas: cargar modulos dinámicos (por ejemplo el proceso init que es el que forkea al shell) y además se ocupa de dejar al procesador en modo protegido con la línea A20 habilitada y en un estado coherente, permitiendonos continuar el desarrollo directamente en C. Los detalles de como se integró GRUB se detallan en la sección 5

- Mapa de memoria: Se utilizan dos tipos de asignadores de memoria: asignador de marcos de página y asignador de espacio de memoria virtual de kernel. También se emplea un esquema de *copy on write* para manejar las páginas compartidas. Adicionalmente se maneja una pila de usuario y una pila de kernel para las tareas. Los detalles del mapa de memoria se encuentran en la sección 6.

- Multitarea: El sistema maneja tareas en nivel de privilegio 3 (correspondiente al nivel de privilegio de usuario en la arquitectura IA-32). El salto de tarea se realiza por hardware mediante el uso del registro TR y la estructura TSS (Task State Segment) que nos provee la arquitectura. Adicionalmente, se maneja un mapa de memoria para cada proceso, una serie de funciones para el manejo de señales e información sobre la estructura del arbol de procesos, usandose listas intrusivas. Los detalles de este módulo se incluyen en la sección 7.

- Driver de disco: Se dispone de un primitivo driver de disco IDE mediante ATA PIO, que maneja sectores de 512 bytes. Este driver utiliza busy waiting para realizar las lecturas y escrituras de y hacia (segun corresponda) buffers de memoria. Esta capa se mantuvo lo más primitiva posible para poderse implementar y testear el sistema de archivos simulando los accesos a disco mediante la memoria RAM. Los detalles de este driver se incluyen en la sección 8.

- Sistema de archivos: Se implemento un subconjunto de funcionalidad del sistema de archivos MINIX 1. Se eligió este sistema de archivos por su simplicidad y por soportar el concepto de inodos, lo cual nos permitió definir una capa de Virtual Filesystem (VFS) muy similar a la disponible en Linux 2.6, y manejar la abstracción de que los drivers son también archivos jerarquicos. Se consideró implementar VFAT 16 pero motivos de implementación motivaron el cambio. Adicionalmente se implemento un esquema de cache de buffers en memoria RAM para mantener la consistencia del sistema de archivos y permitir abstraer la escritura al mismo como secuencial.

  Ambos módulos se detallan en la sección 9.

- Interfaz de llamadas a sistema, interrupcciones y excepciones: Se implementó una interfaz de acceso al Sistema Operativo similar al que utiliza Linux, empleandose un Interrupt Gate asociada a la interrupcción 0x80. Asimismo, los parámetros se pasan por los registros. Se implementaron además un conjunto de llamadas a sistema inspiradas en las llamadas POSIX usuales. Las mismas no se detallaran en una sección especial sino que se explicarán según el caso en cada sección. Adicionalmente, puesto que concierne directamente a la materia, no se explicará como se implementó el sistema de interrupcciones (consiste primariamente en configurar correctamente una IDT para el procesador siguiendo los lineamientos de la arquitectura, lo cual se cubre en Organización del Computador II, y mantener un handler de excepcion general que despacha una función según la interrupcción recibida).

- Driver de teclado, video y reloj: Se programó un primitivo driver de teclado mediante interrupcciones que nos permite bufferear teclas. Asimismo se implemento un driver sencillo de terminal

usandose para ello la memoria de video en modo texto. Ambos se combinaron en un driver con interfaz al sistema de archivos de terminal, utilizado por el programa shell para controlar la entrada salida. Por último, se implementaron funciones básicas de uso del reloj CMOS de sistema para obtener la fecha y hora. No se implementó un driver para este dispositivo por simplicidad. Los detalles se incluyen en la sección 10.

- Shell y tareas a nivel de usuario: Se implementaron un subconjunto de tareas que permitiera evidenciar las facilidades del sistema operativo. Con el proposito de simplificar esto todo lo posible, se busco que se pudiera linkear estáticamente un conjunto de librerías que usaran la interfaz del Sistema Operativo junto con código de lógica algoritmica espefico a la tarea. También se buscó permitir el uso de buffers de memoria estática (las conocidas `section .rodata` y `section .bss`). Con esto en mente, se buscó que el kernel soportara la lectura de tareas en formato ELF estático de 32 bits ejecutable. Los detalles de la implementación de estos módulos dentro del kernel y de los programas de usuario en si (en especial el shell de sistema operativo) se incluyen en la sección 11.

5. Bootloader : GRUB

Para este trabajo se decidió utilizar el bootloader GRUB (GRand Unified Bootloader), en su version Legacy. La motivación principal de utilizar este bootloader por sobre un bootloader propio o el utilizado en la materia Organización del Computador II fue la robustez y facilidad de uso de GRUB, que incluye funciones que resultaron particularmente utiles como por ejemplo detección de memoria RAM disponible y carga de archivos en filesystem como módulos dinámicos (lo cual se utiliza para, en el booteo de sistema operativo, obtener el proceso init directamente de la imágen), y el hecho de que GRUB realiza cierto trabajo antes de entregar control al código de Sistema Operativo, como por ejemplo activar la línea A20 para disponer de más del primer megabyte de memoria, y setear el procesador en modo protegido con un estado conocido para evitar tener que realizar este proceso (que involucra utilizar assembly en modo real).

GRUB Legacy puede bootear cualquier sistema operativo que cumpla con lo que se conoce como especificación Multiboot. La misma es muy sencilla y requiere únicamente de ciertos valores mágicos en específicas posiciones de memoria dentro del binario que vamos a cargar (en este caso usando GRUB).

En detalle, esto consiste de que el binario de kernel a utilizar debe empezar con un header que contenga:

- Un número mágico de identificación: El valor 0xBADB002.
- Flags para GRUB que indican el tipo de alineación del kernel (por ejemplo, se puede especificar un offset de página, en nuestro caso le indicamos a GRUB que las estructuras son alineadas a página).
- Un valor de checksum calculado con los flags y el número mágico de GRUB.

La necesidad de mantener esta estructura alineada en el kernel nos motivó a utilizar una sección ELF especial que denominamos `.__mbHeader` y que utilizamos en un script de linker para que el binario ELF resultante empiece siempre con el header *multiboot* que queremos alineado a 32 bits (como pide la especificación).

Con esto, el binario resultante de linkear con ld y este script los distintos archivos objeto obtenidos mediante el ensamblador NASM y el compilador GCC sobre el código fuente del kernel, se encuentra listo para que GRUB lo pueda usar.

Para crear la imágen de diskette que utilizamos para bootear el Sistema Operativo, lo que hicimos fue (mediante el proceso descripto en `http://www.osdever.net/tutorials/view/using-grub` en el apartado `Installing GRUB on a floppy with a filesystem`) crear una imágen formateada con ext2 lista para bootear con GRUB, y luego agregamos un archivo `menu.lst` que describe a GRUB el nombre del sistema operativo, el archivo que tiene que utilizar como imágen multiboot para iniciar el Sistema Operativo, y también le indicamos que módulos debe cargar. Puesto que GRUB entiende el sistema de archivos ext2, lo único que es necesario por lo tanto hacer es copiar los archivos a los lugares correctos (copiando para ello la imagen *raw* del diskette booteable y usando `e2cp` para copiar los archivos a la imágen) y referirse a ellos por nombre en el listado del archivo *menu.lst*.

Con esto entonces para cuando llamamos a la función `kmain` del Sistema Operativo disponemos de una estructura de memoria detectada por GRUB (lo cual potencialmente usa la BIOS si esta disponible u otro método si no es así) y de un puntero a estructuras de módulos para los archivos que hayamos deseado cargar.

Para más información consúltese [2], [9], [10], [11].

6. Mapa de memoria

Luego de realizar la configuración de la GDT e IDT (que omitimos explicar en este trabajo puesto que lo consideramos cubierto por el contenido de la materia), el siguiente paso consiste en configurar la administración de la memoria RAM de la computadora, en base al esquema de memoria que se obtuvo de GRUB.

La administración de la memoria física per se (es decir, la RAM realmente en la máquina) se hace en marcos de página, unidades contiguas de 4 KB. Esto es porque se utiliza paginación de a 4 KB mediante la MMU (Memory Management Unit) de procesador, correspondiente a uno de los esquemas de paginación que disponemos por la arquitectura IA-32.

Para administrar estos marcos de página se utiliza una estructura denominada *bitmap* o mapa de bits. El mismo consiste en mantener una larga tira de bits, uno para cada marco de página disponible para el procesador. El estado del bit correspondiente a un marco indica si esta en uso o no. También se mantiene un contador entero para cada marco de página que indica cuantos esquemas de paginación (es decir, cuantos directorios y tablas de página) lo estan usando (esto se usa, como veremos posteriormente para implementar una política de *copy-on-write*).

La razon de tener un bitmap separado (se podría usar directamente el contador para determinar si un frame esta o no en uso) es eficiencia: al mantenerlo separado puede escanearse muy rápidamente por un frame libre utilizando la siguiente estrategia: Vamos levantando de a 32 bits (correspondiente al tamaño de un registro de propósito general en IA-32) y apenas encontramos un valor diferente a $2^{32} - 1$ (una tira de 32 unos) escaneamos el valor del registro para determinar el índice. De esta manera el escaneo se realiza 32 veces más rápido que escanear de a un entero por vez. Adicionalmente, existe una instrucción dedicada del procesador que se puede prefijar para realizar este escaneo: `repnz scasd`, y una instrucción, `bsr`, que permiten realizar los dos pasos mencionados y que en nuestros experimentos optimizó apreciablemente el tiempo de asignación de marcos de página por sobre una función programada en C. Los detalles de funcionamiento de estas funciones no son de interés para este informe, referimos el lector al manual 2 de la arquitectura [4] y al código fuente en `bitset_search.asm`.

Algunos frames se desperdician para estas estructuras, y también se pierden algunos frames para que la memoria administrada este siempre alineada a página (como sería deseable). La perdida de memoria incurrida es mínima (Con 4 Kb de un frame se pueden administrar 512 megabytes de memoria, por lo tanto usando paginación estandar gastaríamos unos 32 Kb en frames lo cual consideramos despreciable) por lo tanto se utilizó esta opción por su simplicidad de implementación y su buena performance.

Con esta función disponible, podemos mantener la administración de los frames encapsulada mediante las funciones `frame_alloc`, `frame_free`. También incluimos una función `frame_add_alias` cuyo propósito explicaremos posteriormente.

Para mantener una abstracción sobre los recursos de la computadora, en este caso la memoria RAM, queremos darle la ilusión a los procesos de usuario de que disponen de todo el espacio de direcciones de memoria. Para ello es que usamos memoria virtual mediante paginación. Sin embargo, nos interesa mantener una visión unificada y cohesiva del Sistema Operativo de manera que los procesos puedan interactuar con el mismo.

Para resolver este problema, se resolvió utilizar un esquema de memoria para los procesos en el que, además de sus secciones de código y datos pertinentes, los procesos tienen:

- El código y estructura iniciales del kernel mapeadas con *identity mapping* a las ubicaciones en memoria física, de manera que todos saben donde esta el código de kernel. Se utiliza el esquema de protección propio de paginación de manera que solo se pueda acceder a estas páginas en anillo 0, es decir en modo kernel.

- Se mantiene una sección especial de memoria de datos del kernel para estructuras necesarias (como puede ser, estructuras de procesos, objetos para representar archivos, etc.) mediante el uso de memoria dinámica. Esta sección también es común a todos los procesos puesto que por ejemplo en cambios de contexto es necesario que, cuando el procesador cambie de modo usuario a modo kernel, el esquema de memoria con el que venia corriendo el proceso tenga acceso a la lista de procesos a schedulear. Este pedazo de la memoria virtual se denominará heap de kernel.
- Una sección de pila de usuario, con una cantidad de páginas estática.
- Una sección de pila de kernel, con una cantidad también estática de páginas asignadas.

La heap de kernel, si bien se corresponde directamente con un espacio de memoria físico, se administra a nivel virtual, es decir considerando paginación. La administración se realiza mediante el uso de bloques de listas enlazadas: Cada sección de la memoria mantiene su tamaño y el puntero a la siguiente posición libre de memoria. Si bien esto desperdicia algo de espacio, nuevamente, este es mínimo, y el algoritmo en si es relativamente sencillo. La asignación de memoria sigue la implementación de [7] con algunas modificaciones menores para mejorar la detección de errores.

También implementamos una función que obtiene memoria alineada a página (lo cual se usa para por ejemplo para obtener directorios de página para los procesos, ya que la arquitectura requiere que los directorios de página esten alineados a página). Dado que el inicio de la heap de kernel es en los 3 GB, la dirección virtual y la física están alineadas a página. Lo que se hace para lograr esto es simplemente pedir 4095 bytes (4 KB - 1) más de espacio. De esta manera, nos aseguramos (por aritmética modular) que hay una sección continua del tamaño que queremos y que esta alineada (tiene resto 0) a 4 K. Para evitar desperdiciar la memoria que queda a los extremos de esta sección asignada, la devolvemos a la heap de kernel. Si bien esto puede llegar a contribuir a la fragmentación externa de la memoria, no consideramos esto un requisito y por lo tanto decidimos utilizar esta solución.

Para mantener la heap de kernel siempre mapeada, preasignamos una cantidad adecuada de tablas de páginas de manera que tengamos el espacio inicial de heap asignado (ya que necesitariamos de otra manera conseguir memoria para las tablas de páginas para poder administrar memoria, teniendo una dependencia circular donde pediríamos memoria eternamente).

Por último, en esta sección describimos el esquema de `copy-on-write` utilizado. Cuando una tarea usa la llamada de sistema `fork`, no se duplica el espacio de memoria directamente. En particular, por ejemplo, las paginas de kernel se linkean directamente, no se obtienen marcos de página para duplicar. En el caso de las páginas de usuario, tampoco se obtiene directamente memoria. Lo que se hace es utilizar uno de los bits disponibles de las entradas de página para marcar la página como `copy-on-write`, y se mantiene el identificador de marco de página. La página se marca además como de solo lectura. De esta manera, no se incurre en ningún costo al acceder a la página para leerla, y no se hace la copia. En cambio, si se trata de escribir, la unidad de memoria detecta que la página esta como solo lectura y se dispara una excepción 14 (Page Fault). En ese momento toma el control el manejador de esta excepción, que usa el bit de `copy-on-write` para determinar que se tiene que copiar el marco de página, asigna un nuevo marco, lo copia, desmarca el marco original y luego le permite al proceso retornar su ejecución. Esto no solo simplifica el código necesario para que las tareas realicen `fork` sino que también mejora mucho la performance (es una optimización implementada en Linux).

Sin embargo, las páginas de la pila de kernel no se manejan con este esquema: Recordemos que el manejador de la excepción necesita una pila válida para trabajar, y que si la propia pila de kernel necesita del manejador de page fault entonces se produce una dependencia circular que deriva en Triple Fault de procesador. Esta motivación lleva a que la pila de kernel de cada proceso se copie en cada `fork` por separado: usar una pila común es una solución que es imposible hacer escalar (en una posible continuación de este TP) a un kernel preempteable.

7. Mecanismos de multitarea

> ...nobody really uses an operating system; people use programs on their computer. And the only mission in life of an operating system is to help those programs run.
>
> Linus Torvalds - *Revolution OS*

Como bien dice Linus, el propósito del Sistema Operativo en si es servir como capa de abstración de la computadora para los procesos de nivel de usuario, y permitir además la multiplexación de estos recursos entre multiples instancias de programas de usuario en ejecución. Lo que buscamos en este Trabajo Práctico fue proveer una capa básica de interfaz para las tareas de usuario. En este apartado describimos el mecanismo de multitarea provisto.

La implementación sigue la línea estandar de cambios de contexto por hardware. Para esto utilizamos fuertemente la estructura de TSS (Task State Segment) del procesador, lo cual simplifica mantener al procesador en un estado consistente según la tarea en ejecución. No incluimos una explicación detallada del uso de esta estructura pues es parte del programa de la materia.

El tiempo de uso de CPU de cada proceso es un *quantum* fijo. El scheduler entonces provee una serie de funciones para manejar las tareas en modo *round-robin*: El orden de ejecución de las tareas es circular y no existen diferencias de prioridad de tareas. Para tener una noción de tiempo asociada a cada uno de estos procesos, utilizamos el PIT (Programmable Interrupt Timer).

El PIT es un reloj de cuarzo que envía cada cierto intervalo de tiempo una interrupcción por la línea 1 del chip del Programmable Interrupt Controller. Esta interrupcción nos sirve entonces como medida de paso del tiempo. Para ajustarla a una cantidad determinada de milisegundos, lo configuramos mediante el uso del puerto de comando 0x43 y enviando la nueva frecuencia en dos bytes al puerto 0x40 como esta detallado en `timer.c`. De esta manera tenemos manejo de cuanto tiempo permitimos correr a cada proceso.

Si bien no existen prioridades para los procesos, un proceso puede decidir dejar el procesador voluntariamente mediante la llamada a sistema `sleep`. Esta es también la única manera que el procesador puede liberar la computadora si se encuentra en modo kernel. Dado que el scheduler no puede retirar una tarea si esta se encuentra en modo kernel, este es no preemteable. Sin embargo, si es posible interrumpir al kernel, como veremos posteriormente cuando se explique la implementación del driver de terminal.

Para crear y manejar tareas se utiliza un esquema similar a las llamadas de Sistema Operativo clásicas de UNIX y POSIX, `fork`, `exec`, `wait` y `exit`.

- `fork` crea un duplicado del proceso actual, con el mismo mapa de memoria (marcado acordemente como se explicó anteriormente en la sección de *copy-on-write* para que apenas halla una escritura se reasignen las paginas), archivos abiertos y directorio de trabajo. La relación entre el nuevo proceso y el proceso del cual se obtuvo es de padre e hijo: en particular como veremos posteriormente un proceso puede esperar por uno de sus hijos dado su número identificador de proceso (PID). Para ello, *fork* devuelve dos valores distintos al proceso hijo y al proceso padre: al padre le devuelve el PID del hijo, y al hijo le devuelve 0. Si ocurre un error al intentar ejecutar esta llamada a sistema, se devuelve un valor negativo que sirve para identificar el tipo de error.
- `exec` sobreescribe la imágen del proceso con la imágen de proceso obtenida del archivo en disco duro pasado por parámetro, efectivamente entonces poniendo a ejecutar un proceso con una nueva sección de texto y datos. `exec` cierra todos los archivos abiertos por esta nueva imagen exceptuando entrada y salida estandar. La relación padre e hijo se mantiene.

- `wait` bloquea un proceso hasta que uno de sus hijos (dado por el número PID pasado por parámetro) llama a la system call `exit`. Esto se usa por ejemplo en el shell para permitir la ejecución de una tarea y luego volver el control a la consola.
- `exit` Termina la ejecución del proceso y libera sus recursos. Adicionalmente, se desbloquea su padre si lo estaba esperando. Cuando un proceso hace `exit` sus hijos son reasignados a su padre en caso de tenerlo (se asume de todos modos que el proceso inicial `init` nunca hace exit).

La implementación, si bien en funcionalidad es cruda, demuestra como se podría estructurar este mecanismo para permitir multitarea.

Adicionalmente, se implementó como experimento un sistema básico de manejo de señales entre procesos. Las mismas se manejan solamente en modo usuario y se dispone de un subconjunto de señales: SIGSTOP, SIGCONT, SIGKILL y SIGSTOP. El kernel anota si las señales fueron recibidas en modo kernel y registra esto para manejarlas apenas retorna a modo usuario. Se permite además registrar handlers particulares para señales excepto para SIGSTOP SIGCONT y SIGKILL. Para que todos las funciones manejadoras de señales se ejecuten en modo usuario, los handlers por default para cada una de estas señales se mantienen en una página especial de kernel que se mapea con permisos de usuario. Para bloquear un proceso ante la recepción de SIGSTOP, se implementa una syscall especial denominada `do_coma` que bloquea permanentemente a un proceso. Esto solo se deshace ante la recepción de la señal SIGCONT, cuyo handler no realiza acción alguna.

Cada proceso tiene asociada una estructura, además de la TSS, que controla que archivos tiene abiertos, su estado (si esta bloqueado, corriendo actualmente o listo para correr, de manera de tomar decisiones acordes cuando se realiza el *scheduling*) su directorio de trabajo actual, una lista enlazada (que casualmente es la implementación de lista enlazada intrusiva que utiliza el kernel de Linux) de sus procesos hijos, su número de identificación, un puntero a la estructura de proceso de su padre (o NULL si es el proceso init). Esta estructura sirve para mantener el árbol de procesos organizado: Un ejemplo es que cuando un proceso realiza *exit*, sus hijos son reasignados a su padre y además se necesita saber cual es su padre para desbloquearlo.

8. Disco duro

Se implementó un sencillo driver de disco duro con el propósito de dar un soporte físico al sistema de archivos que describiremos posteriormente.

El driver de disco implementado maneja un solo disco duro IDE, el disco duro maestro. La implementación del driver utiliza ATA PIO, o Programmed I/O. Los sectores en el disco se direccionan utilizando para ello LBA 28 (Logical Block Addressing) lo cual nos permite abstraernos sobre la estructura del disco en términos de Cylinders, Heads y Sectors (*CHS Addressing*) y pensarlo como un arreglo secuencial de sectores. Se asume adicionalmente que el disco duro tiene un tamaño de sector de 512 bytes (lo estandar para los discos duros ATA).

La implementación realizada interactua con el controlador de disco duro ATA Master mediante una serie de puertos, que detallamos a continuación:

- Los puertos 0x1F0 a 0x1F7 son puertos de control para las operaciones de lectura y escritura. A continuación detallamos el uso de cada uno de los registros
  - Puerto 0x1F0: Este puerto es el puerto de datos, el cual nos permite enviar y recibir datos del controlador en grupos de *words* (2 bytes o 16 bits).
  - Puerto 0x1F1: Puerto de error. En la implementación realizada, por simplicidad, no se empleo este puerto. Se utiliza para verificación de errores entre transferencias luego de leer el registro de control.
  - Puerto 0x1F2: Conteo de sectores. Se emplea para multitrasferencias de más de un sector contiguo, lo cual prepara el controlador de disco con los datos.
  - Puerto 0x1F3: Se usa para indicar el primer byte de la dirección LBA donde empieza la transferencia de sectores indicados.
  - Puertos 0x1F4,0x1F5: Se usan con similar motivación al puerto anteriormente explicado, pero para el segundo y tercer byte respectivamente.
  - Puerto 0x1F6: Se utiliza para seleccionar que disco duro se va a usar (cuando se dispone de slaves además de un master). Asimismo se utiliza para indicar los bits restantes de la dirección LBA.
  - Puerto 0x1F7: Este puerto corresponde al puerto de control del controlador de disco. Se lee para obtener el registro de estado del controlador como flag, y si se escribe es para indicar un comando al controlador. Los bits del byte de estado devueltos que son importantes a la implementación son:
    - Bit 0 (ERR): Indica que ocurrió un error.
    - Bit 3 (DRQ): Indica que el controlador esta listo para recibir o enviar datos.
    - Bit 5 (DF): Indica una falla en el disco
    - Bit 6 (RDY): Indica que termino la operación o que ocurrió un error.
    - Bit 7 (BSY): Indica que el disco esta ocupado atendiendo un comando.
- El puerto 0x3F6 se utiliza para controlar el disco primario. En particular se utiliza para detectar e inicializar el disco duro maestro.

Los algoritmos utilizados en si son sencillos: Las operaciones son bloqueantes en el sentido de que utilizan ciclos de CPU para ejecutarse y no se desaloja al proceso que las esta realizando (porque esta en modo kernel). Si bien esto es desventajoso, la velocidad de transferencia supera a la que se puede obtener utilizando por ejemplo ISA DMA por el chip 8237.

En primer lugar se detecta y se realiza un reset por software del controlador de disco (función `ata_reset` y `hdd_init`). El reset se realiza enviando el valor 4 por el puerto 0x3F6, valor que corresponde al comando de reseteo. La función `ata_read_stable` tiene como propósito proveer un tiempo

de espera de 400 ns (lo cual es requerido por la especificación) para que se estabilice la circuitería interna del controlador. El valor del registro corresponde al último leído. Una vez que el controlador resetea todos los discos ATA en el bus correspondiente al registro de control usado, liberamos el bit prendido anteriormente con el comando enviado.

Para detectar el disco, simplemente enviamos un comando al puerto de dirección LBA con un valor y esperamos volver a leer ese valor.

Por último, veamos el algoritmo para lectura y escritura. Lo primero que hacemos es, usando los registros de control, ubicar al controlador de disco en el sector de inicio y pasarle la cantidad de sectores que vamos a leer o escribir. Posteriormente esperamos a que el disco duro este listo para continuar haciendo *busy-waiting* en el registro de control hasta obtener que se libere el bit BSY, indicando que el controlador esta listo para hacer la transferencia.

La transferencia se realiza de a un sector por vez, enviando 256 words por el puerto de datos. No utilizamos las instrucciones del procesador `rep outsw` y `rep insw` puesto que el controlador de disco requiere de un tiempo entre transferencias para acomodar los datos y el procesador puede transferir demasiado rápido para él. Adicionalmente, después de la escritura de un sector enviamos un comando de flush de caches del controlador para evitar que siguientes escrituras fallen silenciosamente. Y luego de una escritura le damos un tiempo de 400ns para que se estabilice el controlador (en el caso de la escritura eso ya lo realiza de por si el flush de caches).

Para más información se puede consultar [1], [9].

9. Sistema de archivos

Para permitir cargar tareas desde disco duro, se hace necesario contar con una manera de abstraer una jerarquía de archivos por sobre la organización de bits en sectores que corresponde al nivel de abstracción de un controlador de disco duro. Para ello se implementó un sistema de archivos.

Nos debatimos entre dos sistemas de archivos: El sistema de archivos VFAT 16, que consiste en FAT 16 pero con *hack* que permite nombres más largos que 8 caracteres (más los 3 de extensión), y el sistema de archivos de Minix 1. Luego de implementar ambos sistemas de archivos, preferimos utilizar el sistema de archivos de Minix puesto que es más sencillo y permite una mejor abstracción de recursos que no necesariamente son archivos físicos en disco (como por ejemplo drivers de dispositivo).

A continuación describimos entonces el sistema de archivos de Minix 1. El sistema de archivos VFAT 16 lo dejamos para otro trabajo.

El concepto prevaleciente de Minix 1 es el concepto de inodo: Todo archivo en el sistema tiene un inodo correspondiente que guarda no solo su organización en disco (dependiendo si el archivo es un archivo fisico, un directorio, un dispositivo, etc.) sino sus metadatos: tamaño, estado en disco (si la imagen en memoria esta modificada y debe hacerse una modificación al disco para mantener la coherencia), etc. Todo inodo que tiene datos físicos asociados tiene un cierto número de zonas asignadas. En Minix una zona o bloque corresponde a una sección contigua de 1024 bytes (2 sectores según el driver de disco detallado en la sección 8).

Las zonas asignadas a un inodo se organizan en 3 partes, identificadas mediante números de 16 bits (lo cual implica un límite de 65536 zonas que Minix puede direccionar en un disco duro).

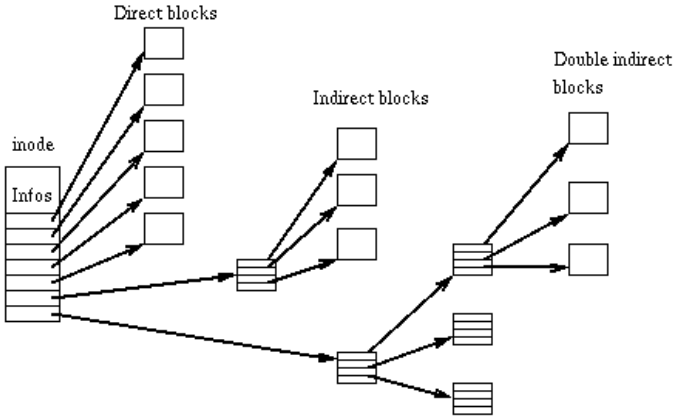
- De direccionamiento directo: Hasta 7 zonas se guardan en la representación misma del inodo en disco duro, permitiendo así acceso rápido a las primeras partes del archivo.
- Bloque indirecto: Además de estas 7 zonas se guarda un puntero a una zona de disco que contiene asimismo punteros a zonas directas de datos. Esto nos permite tener

  $$\frac{1024 \cdot 1024}{16} = 65536$$

  o 64 Kb de datos accesibles en zonas que requieren dos lecturas.
- Bloque doble indirecto: Por último, el inodo en memoria contiene un puntero a una zona de punteros a zonas de punteros (por eso doble indirecto) a zonas de datos.

Un esquema de la organización de los inodos en disco se detalla en la Figura 1

Figura 1: Ejemplo esquematico de la organización de un inodo en disco. Si bien esta imágen corresponde a la organización en el sistema de archivos ext2, para el sistema Minix 1 la organización es similar exceptuando la cantidad de zonas directas. Imagen tomada de http://e2fsprogs.sourceforge.net/ext2-inode.gif



Cada inodo en el sistema de archivos Minix 1 se identifica por un número entero positivo entre 1 y 65536. Esto permite por ejemplo que la organización de los directorios siga un esquema de árbol (aunque existe la noción de *hard* y *soft* links en Minix 1, por simplicidad no fueron implementados): Cada inodo de directorio almacena en sus zonas de datos entradas de 32 bytes que consisten en:

- 2 bytes para el numero de inodo correspondiente al hijo.
- 30 nombres para el pedazo del nombre del hijo.

Por ejemplo, si tuvieramos el archivo `/docs/docs/archivo.txt`, `archivo.txt` sería el nombre con el que identificaríamos en el inodo del segundo `docs` al inodo del archivo final. La organización es entonces jerárquica separada por caracteres de barra como es usual en los sistemas UNIX.

Por último es necesario considerar los archivos correspondientes a drivers. Los drivers en Minix se identifican con dos números, el número major (que indica el tipo de dispositivo) y el número minor (que indica una instancia del tipo de dispositivo dado por el otro número). Estos dos valores ocupan 1 byte y por lo tanto se almacenan en la primer entrada de zonas del inodo en disco duro. Como esta entrada tiene tamaño de 2 bytes, en el byte menos significativo se almacena el major number y en el más significativo el minor number.

La distinción entre los distintos tipos de inodo es posible gracias a un flag de identificación que es parte del inodo. Además de esto el inodo en disco mantiene el tamaño en bytes del archivo.

Finalmente, es necesario mantener ciertos datos sobre la partición utilizada: por ejemplo que inodos están usados o no, la información de cada inodo en una sección encontrable a priori de iniciar la partición, y que zonas en disco están o no libres.

Para esto, toda partición Minix 1 de disco inicia con el bootblock (que contiene la información para bootear el sistema operativo en disco duro, que en nuestro caso no utilizamos puesto que se bootea directamente mediante el diskette) al cual le sigue el denominado *super block*. El super bloque inicia entonces en la segunda zona (Sector número 3) y contiene la siguiente información:

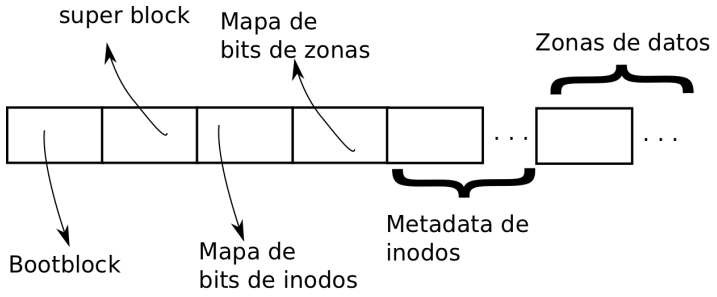
- La cantidad de inodos en la partición actual.
- La cantidad de zonas de datos disponibles en la partición actual.

- Cantidad de bloques correspondientes al mapa de bits de inodos.
- Cantidad de bloques correspondientes al mapa de bits de zonas de datos libres.
- Zona de inicio de las zonas de datos.
- Valor c tal que 2^c corresponde al tamaño en zonas del espacio de datos del disco.
- Tamaño máximo de archivo.
- Valor mágico que identifica esta partición. En el caso de MINIX 1 este valor es 0x138F

Posteriormente a este super bloque, viene un mapa de bits que codifica que inodos están o no libres, y luego un mapa de bits que codifica que zonas de datos están o no libres. Ambos mapas de bits son cargados a memoria cuando se inicializa el super bloque en memoria con el próposito de utilizar la estructura `bitset` descripta en la sección 6 para administrar estos espacios. Finalmente, lo que sigue es la zona de inodos: un arreglo contiguo de zonas que almacenan la metainformación de cada inodo del sistema, o espacio libre si ese inodo no esta asignado. Puesto que la estructura en disco de un inodo consiste de 32 bytes, la cantidad de estas que entra en 1024 bytes es entera y nunca cruza un borde de zona. Finalmente, a esto sigue las zonas de datos.

Un esquema de esta organización de las primeras zonas del disco se puede ver en la Figura 2.

Figura 2: Organización de las primeras zonas de una partición de Minix.



Otro motivo para utilizar este sistema de archivos es que en el entorno de desarrollo utilizado se disponen de herramientas para crear archivos que contengan una partición de Minix 1: En particular estamos hablando del comando `mkfs.minix`. De esta manera, podemos utilizar este comando y comandos específicos para armar imágenes de disco para el simulador Bochs para crear una imágen de disco con el sistema de archivos valido y que contenga los valores que deseemos. Esto en particular nos fue muy útil para resolver problemas no documentados sobre el sistema de archivos y para además poder testear el código de sistema de archivos por separado al código del resto del Sistema Operativo (mejorando así la modularidad).

Por último ahondaremos en dos detalles implementativos: La noción de Virtual Filesystem y la noción de cache de buffers y cache de inodos.

9.1. Virtual Filesystem

La idea de un sistema de archivos virtual es proveer una abstracción por sobre la implementación particular del sistema de archivos de manera de poder trabajar de manera general sobre ellos. En nuestro caso, la capa de abstracción se provee mediante las nociones de inodo, superbloque y file object. Las dos primeras nociones son idénticas a las de Minix, con modificaciones que veremos posteriormente. La noción de file object es la de una instancia de un inodo abierto. Esto sirve por ejemplo para mantener la idea de que estamos leyendo un cierto offset dentro de un inodo y que esto es independiente del inodo en si (que representa un archivo).

La modificación más importante es que cada uno de estos tres valores provee una abstracción en la forma de una serie de funciones:

- El super bloque debe ser capaz de crear, leer de disco, escribir a disco y mantener la consistencia de los inodos en una partición, sea esta la que sea. En el caso de Minix por ejemplo se ocupa de mantener la consistencia de los mapas de bits con los inodos en uso y las zonas de datos disponibles, y de mantener los metadatos de cada inodo coherentes con su representación en los caches de memoria RAM.
- El inodo debe ser capaz de operar sobre la noción de archivo. Por ejemplo, un inodo de directorio debe proveer funciones para buscar el inodo de un subarchivo directo en la jerarquía, crear subdirectorios y subarchivos, borrar subarchivos, etc. Debe proveer además una abstracción con respecto a que operaciones de acción se pueden realizar sobre él: que quiere decir leerlo, escribirlo, etc.
- Un file object no provee más abstracción que la de ser una instancia de un inodo abierto. Mantiene índices dentro del mismo y conoce las operaciones que se pueden realizar. Su propósito por sobre todo es ser el objeto interfaz entre los procesos (mediante las llamadas a sistema) y el sistema de archivos en si.

De esta manera, se tiene una abstracción general que permite por ejemplo el manejo genérico de archivos con un mismo conjunto de llamadas a sistema (que describiremos en 9.3).

9.2. Caches de disco y de inodos

Para evitar posibles problemas de fragmentación de la heap de kernel, se mantiene un cache de inodos. Esto quiere decir que cuando el sistema operativo necesita un inodo, busca si uno ya preasignado fue liberado. Si fue liberado, lo utiliza, sino, crea uno nuevo y lo agrega a una lista. Esta lista es responsabilidad del superbloque de la partición, que la mantiene. De esta manera, no se fragmenta la memoria producto de crear y liberar muchas veces un inodo, y se mejora la performance y se mantiene una interfaz más simple. Cada inodo lleva para este propósito un contador de instancias en uso. Si bien el sistema de archivos es *single-threaded* como esta implementado (es decir, se producirían condiciones de carrera si dos instancias de un proceso por ejemplo actuaran sobre el mismo archivo), este contador de instancias de uso se podría utilizar posteriormente para mantener una sola instancia sincronizada en memoria del inodo, manteniendose entonces la consistencia del sistema de archivos (usandose para ello por ejemplo un esquema de sincronización por semáforos).

Para abstraer los accesos a disco, que se hacen por sectores, se implementó además un cache de buffers, muy similar al disponible por ejemplo en Unix System V (ver [3]). Este cache consiste en una serie de buffers de tamaño de una zona, linkeados entre si con una lista enlazada. Cada buffer sabe a que zona y con que desplazamiento esta actuando, y además sabe si los datos están o no sincronizados con el disco duro (ya que sabe que operación se esta realizando sobre él). Cuando se desea escribir a disco solamente es necesario indicar el bloque y offset deseados, y pasar los datos. El cache de buffers se ocupa de buscar el buffer correspondiente si ya esta abierto, y si no es así se ocupa de encontrar un buffer libre y desalojarlo como corresponda (por ejemplo escribiendolo a disco si estaba sucio). En última instancia si no encuentra ninguno lo que hace es agregar un nuevo buffer a la lista enlazada de buffers libres.

Adicionalmente, se mantiene una lista de buffers sucios y a que inodo corresponde cada uno, de manera de poder, en caso de ser pertinente, bajar los datos a disco de los buffers sucios correspondientes a un inodo (por ejemplo, cuando se realiza un `close` o un `flush`).

9.3. Llamadas de sistema operativo

La interfaz con la que cuentan los procesos es la interfaz de llamadas de sistema de archivos. Estas llamadas son similares a las del estandar POSIX: Cada proceso guarda una estructura de hasta cierto número de file objects abiertos. Estos file objects se heredan entre llamadas de fork y se cierran al terminar el proceso (con excepcion de la entrada y salida estandares que se comparten con el shell como veremos posteriormente). Al usuario de estas llamadas no se le devuelve un file object directamente sino que se le devuelve un identificador dentro del arreglo del proceso, mas conocido en la jerga como un *file descriptor*. Este se utiliza para el resto de las llamadas.

A continuación describimos las llamadas de sistema implementadas.

- open: Esta llamada a sistema recibe un directorio y una serie de flags que indican que se desea hacer con el archivo a abrir, y el sistema entonces accede al archivo correspondiente, lo prepara para utilizarse, y devuelve un *file descriptor* identificador para llamadas posteriores. Si se produjo un error en este proceso, se devuelve el mismo en vez de un descriptor de archivo. Para diferenciar identificadores de códigos de error, los códigos de error tienen valores negativos (el mismo sistema se usa para las demás llamadas a sistema).
- read: Esta llamada a sistema toma un archivo abierto para lectura y lee una cantidad determinada de bytes a un buffer. Para evitar indicar el índice cada vez, este se mantiene en el *file object* correspondiente, y se actualiza de manera coherente. Si ocurre un error en la lectura se retorna un código de error. Sino, se retorna la cantidad efectiva de bytes leidos (el sistema de archivos se ocupa de evitar que se pase de largo del archivo, lo cual facilita el uso de la interfaz mediante el uso de un buffer).
- write: La llamada es similar a read, pero para escribir a un archivo. A diferencia de read, esta llamada utiliza un flag (que se determina a tiempo de apertura del archivo) para indicar si se desea sobreescribir un archivo o agregarle información . También esta llamada utiliza otro flag para indicar si al intentar escribir un archivo inexistente el mismo debe ser creado (FS_TRUNC y FS_CREAT especificamente). También se retorna la cantidad de bytes escritos o un error en caso correspondiente.
- readdir: Esta llamada es específica para directorios y se provee como interfaz por sobre la llamada a read. Lo que hace es leer la siguiente entrada de directorio en el archivo a un buffer. Regresa el tamaño de una de estas entradas o un error como código negativo.
- close: Indica que el proceso no va a usar más este archivo y que por lo tanto el sistema operativo debe liberar los recursos asignados a este.

Para más información puede consultarse [13]. Dada la falta de documentación del sistema de archivos Minix, dado que el autor no pudo conseguir una copia de *Operating Systems Design and Implementation* correspondiente a las primeras ediciones, parte de los detalles fueron descubiertos por el mismo autor mediante el exámen minuscioso de imágenes de disco construidas con el comando `mkfs.minix` y examinadas byte por byte con el uso del comando `hdd`.

10. Drivers implementados

Además de proveer una interfaz al disco duro y un sistema de archivos para soportar la abstracción de jerarquía de archivos, implementamos también una serie de funciones para manejar dispositivos de la computadora. En particular, los drivers de video y teclado se unieron en un driver (representado en el sistema de archivos como `/dev/tty`) para terminal de manera que los procesos puedan acceder a estos recursos de manera transparente. También se implementó funciones para acceder al registro de reloj CMOS del procesador de manera de poder proveer al usuario fecha y hora.

10.1. Driver de teclado

El driver de teclado es muy sencillo: Consiste en una interrupcción asociada al pin IRQ 0x1 que se mapea al número de interrupcción 0x21 (este valor se obtiene posteriormente a remapear el Programmable Interrupt Controller de manera que las interrupcciones por PIN no colisionen con las interrupcciones como se explica en [2]). Se instala entonces una función particular para manejar esta interrupcción de manera que el manejador general de interrupcciones pueda llamarla apenás reciba una interrupcción de teclado.

Esta función *handler* de la interrupcción de teclado simplemente lee la tecla presionada usando el puerto 0x60 de datos. Un evento de teclado consiste en el apretado de una tecla o la liberación de la misma. Para manejar esto de manera independiente al resto del sistema, se mantiene un buffer circular de teclas tal que la función de manejo de la interrupcción solo tiene que tomar esta tecla de este puerto y agregarla al buffer circular. Se provee una función además para que los usuarios de este buffer puedan obtener las teclas presionadas. Esto se utiliza en el driver de consola (que el shell en particular emplea).

10.2. Driver de pantalla

El driver de pantalla es sencillo. Se utiliza modo texto en VGA. Las funciones de pantalla entonces actuan directamente sobre el *framebuffer* de modo texto: Un arreglo contiguo de 25 filas, 80 columnas que empieza en la dirección 0xb8000. Este arreglo codifica cada pixel usando 2 bytes: un byte para el caracter a mostrar, y otro byte de formato que almacena en su nibble alto el color del fondo y en su nibble inferior el color del carácter en esa posición. En particular en este trabajo usamos verde sobre negro como ciertos terminales de antaño.

Adicionalmente se mantiene el cursor de posición actual. Este cursor se controla mediante la interfaz por puertos de VGA: Se indica la posición en dos comandos separados. El registro de comandos de VGA es el puerto 0x3D4 y el de datos es el 0x3D5. El proceso entonces para actualizar la posición del cursor a una nueva posición (dada por su offset lineal en el *framebuffer* como word de 16 bits) es:

- Enviar el comando 14 y luego el byte más significativo de la posición al puerto de datos.
- Enviar el comando 15 y luego el byte menos significativo de la posición al puerto de datos.

Lo cual actualiza el indicador de cursor de VGA. Las demás funciones de manejo de la pantalla son relativamente triviales (puesto que escribir un caracter a pantalla consiste en manipular el buffer de pantalla) y su implementación es sencilla de entender del código.

10.3. Driver de reloj CMOS

El reloj CMOS es un reloj de batería que mantiene la fecha y hora del CPU. Este reloj se mantiene actualizado incluso cuando la máquina esta apagada. Almacena las horas en un sencillo formato que corresponde directamente con el que uno esperaría: el tiempo en segundos,minutos,horas y la fecha en día, mes, año y centuria. El formato es un poco distinto al presentado pero es en espíritu similar.

Para leer este reloj se utiliza el puerto 0x70 para comandos y el puerto 0x71 para datos. Los comandos son bytes correspondientes a que registro se desea leer, de los que el reloj CMOS almacena y hemos detallado en el parrafo anterior.

Un cuidado necesario en este driver es deshabilitar interrupcciones para que la lectura no sea influenciada por el scheduler, y la otra es tener cuidado en que el estado del reloj es inconsistente entre updates del mismo: Un update se realiza incrementando el contador de segundos y luego modificando acordemente los otros valores de los registros. Para asegurar que no caemos en el borde de un update, se leen los registros del reloj hasta que se obtienen los mismos dos valores en lecturas consecutivas. Ahi se asume que el estado del mismo esta estabilizado. Para más detalles consultar [9].

Otro cuidado es realizar la conversión de valores según el formato. Los detalles de formato son los siguientes:

- Normal o BCD (Binary Coded Decimal). BCD requiere simplemente realizar una serie de operaciones aritméticas para convertir los valores. Los detalles se pueden consultar en [9] y en el código de conversión de formato.
- Horas en formato 12 o 24.

De ambos detalles se ocupan las funciones programadas.

Un detalle a considerar es que no se implementó, por simplicidad, un driver con representación en disco de este driver. Se prefirió mantener una sola llamada a sistema por simplicidad (ya que lo único que puede hacer un proceso con la fecha y hora es leerla, no la puede modificar ni usar ninguna otra opción).

10.4. Terminal

Combinando las funciones de pantalla y driver implementamos entrada y salida estandares por terminal, para uso del shell y otros programas. Este driver se identifica por el *major number* 0 y el *minor number* 0. La interfaz es idéntica a las llamadas de sistema para archivos descriptas en la sección 9.3. Al abrirse el terminal, se limpia la pantalla y se establecen los offsets necesarios. Leer el terminal consiste en leer de teclado. Este procedimiento es bloqueante, pero sin embargo si el proceso que realizo la lectura no encuentra ningun caracter, simplemente libera el procesador mediante la llamada a sistema `do_sleep` al siguiente proceso en la lista de scheduling. De esta manera no se bloquea el sistema. Escribir consiste en pasar el buffer de caracteres al *framebuffer* del terminal.

El terminal es asociado a los procesos con dos descriptores de archivo: el descriptor 0 para entrada estandar y el descriptor 1 para salida estandar.

11. Tareas de usuario

Por último describiremos las tareas de usuario programas para demostrar el uso de las facilidades de Sistema Operativo que hemos implementado. En primera instancia describiremos el formato ELF usado para las tareas y luego describiremos las tareas mismas que hemos programado.

11.1. Formato ELF

Para facilitar la implementación de las tareas, se decidió utilizar C como lenguaje de implementación. También era nuestro interés poder disponer de una librería separable de funciones y wrappers del código específico para el Sistema Operativo, de manera de poder tener una separación de módulos y así favorecer la simplicidad de desarrollo de las tareas. Por ejemplo, nos pareció importante poder mantener una sección de datos (`.data`), secciones reservadas de espacio de memoria (`.bss`) y secciones de constantes (`.rodata`) de manera de poder emplear buffers estáticos de memoria (para superar temporalmente la limitación de falta de memoria de *heap* en las tareas).

Ambos objetivos hicieron entonces que fuera necesario que el Sistema Operativo pudiese, como mínimo, soportar la carga de ELFs estáticos. Esto era una solución a todos los problemas planteados anteriormente: es muy sencillo compilar código en C a archivo objeto y linkear todos estos en un ejecutable ELF estático final, mediante el uso de un compilador como GCC y un linker como LD.

Para ello primero describiremos el formato ELF estático ejecutable. El mismo es muy sencillo y consiste de las siguientes partes:

- Un *header* identificador con constantes para comprender y parsear el resto del archivo. Los que resultaron importantes para nuestros propósitos son los siguientes:
  - 16 bytes mágicos: el valor 0x7F y los caracteres ASCII E, L y F y otros valores mágicos que permiten identificar y diferenciar el header de este tipo de ejecutables. En particular dos de estos números mágicos que nos interesan son el 4 (contando desde 0), que corresponde a si es un binario de 32 bits, y el 5 que corresponde a si el binario tiene sus valores multibyte encondeados mediante Little Endian y usando complemento a dos.
  - El tipo de binario ELF: En este caso nos interesa el tipo 2: Executable Files.
  - La version del binario ELF. No nos interesa en particular este valor.
  - Punto de entrada: Dirección de memoria virtual donde inicia el código de este ejecutable (puesto que estamos considerando solamente el caso de binario ELF ejecutable).
  - Offset al comienzo de la tabla de headers de programa. Estos son los segmentos de memoria que necesitamos crear y copiar en memoria virtual para poder tener la imágen de este proceso corriendo.
  - Contador de cantidad de secciones de program headers.
  - Tamaño de una sección de program header (para poder cargarlo e interpretarlo).

  Los demás valores de este header no son utilizados, ya que corresponden a por ejemplo la tabla de secciones y símbolos para un archivo objeto linkeable y demás valores.
- Un header de programa para cada sección relevante. Cada uno de estos consiste de:
  - Un flag de tipo. En nuestro caso particular solo nos interesan los que son del tipo ELF_LOAD, valor 0x1, correspondiente a un segmento cargable. En particular esto implica que el kernel no soporta cargar binarios linkeados dinámicamente. Esto es una limitación aunque si le interesa al lector hay una discusión sobre si realmente linkear dinámicamente con librerías tiene beneficios [1].

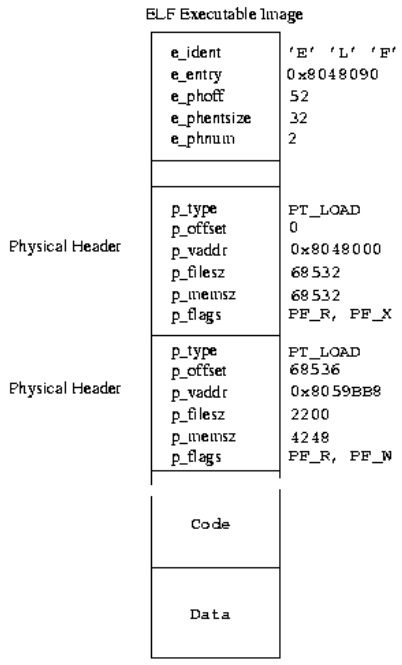
[1] `http://harmful.cat-v.org/software/dynamic-linking/`

- Un offset al inicio de esta sección en la imágen de archivo cargada a memoria. De ahi obtenemos los datos correspondientes a este segmento.
- La dirección virtual donde debemos cargar este módulo.
- La dirección física donde debemos cargar este módulo si es pertinente. Dado que el entorno de desarrollo empleado es GNU Toolchain en Linux, este valor no es relevante a nuestros propósitos.
- Tamaño en archivo en bytes de la sección.
- Tamaño en memoria en bytes de la sección. Estos dos valores pueden diferir ya que una sección puede consistir en una indicación al sistema de que debe reservar una cantidad determinada de memoria. Un ejemplo de este tipo de secciones son las secciones `.bss` que surgen de reservar arreglos estáticos en por ejemplo un programa en C.
- Flags. Estos flags indican por ejemplo si la sección es una sección ejecutable, de lectura y de escritura. Se usa por ejemplo para reservar páginas con permisos acordes.
- Requerimientos de alineación. Generalmente se requiere que los datos esten alineados a páginas de 4 KB.

Con estos datos podemos entonces parsear el archivo elf y generar el mapeo de páginas acorde de manera muy sencilla, que es exactamente lo que realizan los algoritmos. En particular `elf_overlay_image` toma un archivo ELF válido (según nuestras precondiciones) y crea el mapa de páginas acorde en el directorio de páginas actual. Esto se usa para inicializar la tarea `init` y luego se usa en la llamada a sistema `exec`.

Un esquema de un ejecutable ELF se encuentra en la Figura 3.

Figura 3: Diagrama de un ejecutable ELF, mostrando la estructura a alto nivel. Tomado de http://tldp.org/LDP/tlk/kernel/elf.gif



Más detalles sobre el formato ELF se pueden encontrar en [8].

11.2. Consola

La principal tarea de usuario que implementamos es la consola o Shell. Esta se ocupa de interpretar comandos (tomados por teclado) y responder a los mismos de manera acorde.

Es la consola quien en primera instancia abre el terminal para lectura y para escritura (los procesos luego heredan los descriptores de archivos asignados). El interprete de comandos acepta una gramática de la forma:

```
comando := "" | programa lista_argumentos "\n"
lista_argumentos := "\n" | [caracteres] (' ' | '\t')
```

es decir, una lista de un comando y argumentos correspondientes a lo que uno esperaría viniendo de un terminal de consola como los de Unix o DOS.

El Shell parsea entonces comandos como líneas y luego realiza el siguiente proceso:

- Si el comando corresponde a una función de consola (por ejemplo el comando `cd` para cambiar de directorio), se ejecuta una función manejadora de este comando.
- Si no es así, La consola forkea un proceso hijo.
- El subproceso recien creado realiza exec para "convertirse" en el comando que queremos ejecutar. Los argumentos son pusheados a la pila de este proceso mientras se realiza la llamada a sistema `exec`, de manera que al ejecutarse la tarea dispone de los mismos mediante `argc` y `argv` como uno esperaría que ocurre en la mayoría de los Sistemas Operativos basados en UNIX o en DOS.
- La consola espera a que termine este proceso hijo de ejecutarse.
- El proceso hijo realiza su trabajo y luego ejecuta la llamada a sistema `exit` para liberar el procesador.
- La consola retoma entonces el control.

Como podemos ver la implementación sigue los lineamientos que uno esperaría de una consola de sistema muy básica, como esta explicado en [3].

11.3. Tareas implementadas

Para demostrar un subconjunto de funcionalidad, implementamos las siguientes tareas y comandos específicos de consola

- echo, devuelve por consola todos los parámetros que le son dados.
- date, devuelve por consola la fecha y hora
- ls, devuelve el contenido del directorio indicado si es llamado con argumentos y el del directorio actual si no se le pasan argumentos.
- cat, devuelve por consola el contenido del archivo pasado por parámetro.
- cp, copia (creandolo si es necesario) un archivo con nombre pasado por parámetro a otro lugar pasado también por parámetro.

12. Posibilidades de expansión

> Regression testing? What's that? If it compiles, it is good; if it boots up, it is perfect.
>
> Linus Torvalds

Considerando las limitaciones señaladas en este trabajo, proponemos que siguientes trabajos podrían expandir lo realizado en este en las siguientes áreas:

- Quitar la no preemptabilidad del kernel. Esto requiere la implementación de un sistema de exclusión mútua para proteger las áreas compartidas del sistema operativo de acceso concurrente (como por ejemplo el asignador de memoria o la lista de procesos). Adicionalmente, requiere tomar una solución con respecto al problema de PC Losering, es decir como debe manejar el Sistema Operativo una señal o interrupcción cuando esta realizando el servicio de una llamada de Sistema. Posibles soluciones a este problema varía y recomendamos `http://www.jwz.org/doc/worse-is-better.html` donde se introduce y discute la temática.
- Implementar más características de consola, como por ejemplo redirección de entrada salida y unión de comandos mediante `pipes` clásicos de Unix. También se haría necesario implementar *job control groups* para dar una interfaz esperable sobre la entrada para que los procesos puedan leer de entrada salida estandar.
- Agregar soporte de concurrencia al sistema de archivos, mediante un sistema de exclusión mútua. Esto requeriría por ejemplo implementar semáforos y agregarselos a los inodos y al caché de buffers (usando patrones como por ejemplo productor consumidor, para más información véase [12]). Esto se hace imperante si se desea además agregar acceso a disco no bloqueante o mediante el uso de DMA (Direct Memory Access).
- Agregar características al sistema de archivos, como por ejemplo particiones multiples, *symlinks*, filesystems multiples (usando la capa de Virtual Filesystem implementada actualmente), soporte para filesystems en block devices no basados en disco
- Implementar un driver en modo usuario de VGA o VESA para permitir mayores resoluciones. También podría implementarse un driver de sonido para placas Sound Blaster.
- Agregar una etapa de detección de hardware y modificar el driver de disco duro para que utilice PCI Busmastering DMA y permita multiples discos duros ATA (no solo un master).
- Implementar más programas a nivel de usuario.
- Implementar librerías compartidas y agregar soporte para linkeo dinámico.
- Implementar un esquema de `swapping` de páginas a disco.
- Optimizar los algoritmos: Un lugar posible de optimización es reducir la cantidad de búsquedas secuenciales mediante el uso de tablas de Hashing o diccionarios similares.
- Implementar más llamadas a sistema y posiblemente convertir las existentes para que sean POSIX compatibles.
- Usar esquemas de paginación más modernos en caso de detección para soportar más de 4 Gb de memoria (por ejemplo PAE Paging).
- Portear el código a IA 64.

Todos estas posibles mejoras se han tenido en cuenta en el diseño actual (si bien no están implementadas) de manera que no debiera ser necesario un rediseño completo para implementarlas (aunque si posiblemente se debe tener cuidado en especial con las que involucran acceso concurrente). Dejamos estas áreas como posibilidades de expansión.

13. Conclusión

El Sistema Operativo implementado, si bien es crudo en funcionalidad y tiene varias áreas posibles de optimización, sirvió para organizar y experimentar con distintos aspectos del diseño de un sistema de estas características, en especial que módulos es necesario tener y como se pueden abstraer detalles de la arquitectura pero usarlos de manera de implementar funcionalidad. El trabajo también sirve como base y punto de expansión para trabajar con otros conceptos de Sistemas Operativos y hace uso de la arquitectura IA 32 de manera de mostrar como se pueden emplear los recursos que esta provee.

14. Agradecimientos

Se agradece a Christian Heitman por sus útiles links y comentarios, sin los cuales este trabajo hubiese probablemente sido inatacable, y a Manuel Ferreria por su ayuda en la verificación de la imágen entregable.

A. Como correr el Sistema Operativo

Para correr el sistema operativo se necesita de una distribución de Linux que cuente con `gnuutils`, en particular GCC y LD, *nasm* y `bash`. Además, en particular se necesita

- Bochs 2.4.6 o más moderno.
- La suite de utilidades `e2tools` para copiar archivos a la imagen de diskette. Se puede conseguir usando el *package manager* de la distribución de Linux utilizada. En el caso de Ubuntu, se puede usar el comando `sudo apt-get install e2tools`.
- La suite de utilidades de particiones `mkfs`, en particular `mkfs.minix`. Para instalarlo puede utilizarse el comando `sudo apt-get install util-linux`.

También, dado que uno de los pasos requiere configurar y montar un *loopback device*, se necesitan privilegios de sudo para ejecutar ese script. Si bien se ha tenido cuidado de no conflictuar ningun nombre, se avisa que es posible que se produzcan problemas en este paso (por lo cual se recomienda leer el script `linkage/build_image.sh` antes de ejecutar el siguiente comando).

Correr la simulación dadas las dependencias citadas anteriormente consiste en situarse en el directorio más arriba y ejecutar:

```
make run
```

El Makefile asume que los binarios de Bochs (en particular `bximage` y `bochs`) se encuentra en la carpeta `~/bochs/bin` siendo `~` la carpeta home del usuario actual. Esto sin embargo se puede modificar utilizando el prefijo `BOCHSDIR` al ejecutar `make` de la siguiente manera:

```
make BOCHSDIR=path/a/bochs/bin run
```

Referencias

[1] Especificación ATA, disponible en `http://hddguru.com/content/en/documentation` y código ejemplo en `http://www.ata-atapi.com/`

[2] James Molloy, *Roll your own toy UNIX-clone OS*, tutorial disponible en `http://www.jamesmolloy.co.uk/tutorial_html/index.html`

[3] Maurice Bach, *The Design of the Unix Operating System* Prentice Hall Software Series, 1986

[4] Intel 64 and IA-32 Architectures Software Developer's Manual, *Volume 2 (2A & 2B): Instruction Set Reference, A-Z*

[5] Intel 64 and IA-32 Architectures Software Developer's Manual, *Volume 3A: System Programming Guide, Part 1*

[6] Bovet, Daniel P. y Cesati, Marco *Understanding the Linux Kernel*, O'Reilly Media, 2000

[7] Kernighan, Brian y Ritchie, Dennis *The C Programming Language, 2nd Edition*, Pearson Education, 1991

[8] System V ABI Especification, `http://www.sco.com/developers/devspecs/gabi41.pdf`

[9] Wiki de OSDEV, `wiki.osdev.org`, en particular los artículos:

- ATA PIO
- CMOS clock
- ELF
- Bare Bones Tutorial

[10] Bona Fide OS Developer Tutorials, en particular:

- Memory Management 1 y 2 por Tim Robbins.
- LBA HDD Access via PIO por Dragoniz3r.
- Bran's Kernel Development Tutorial por Brandon F.
- Using GRUB por Chris Giese.

[11] Especificación Multiboot por la GNU Free Software Foundation, `www.gnu.org/software/grub/manual/multiboot/multiboot.html`

[12] Downey, Allen B. *The Little Book of Semaphores*, Green Tea Press, 2008, `http://www.greenteapress.com/semaphores/downey08semaphores.pdf`

[13] Heavner, Scott D. *Introduction to the Minix File System*, `http://ohm.hgesser.de/sp-ss2012/Intro-MinixFS.pdf`