Organización del Computador II

Final

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Sistema de Archivos

Integrante	LU	Correo electrónico
Bottaro, Juan Pablo	552/09	jpbottaro@gmail.com

Contents

1 Introducción

Los sistemas operativos han evolucionado mucho desde los primeros momentos en que fueron concevidos. Fue esta evolución y crecimiento que hizo necesario modularizar y separar los componentes principales que lo conforman. Uno de los componentes más importantes de los sistemas operativos modernos es el denominado Sistema de Archivos (o más conocido como File System o FS), que es el encargado de simular todo el paradigma de archivos en el que basamos la computación. Distintos modelos se han creado con el paso de los años para manipular el directorio de archivos, con ejemplos como FAT/FAT32/NTFS en el mundo Microsoft, ext2/ext3/ext4/xfs etc. en el mundo *nix/Linux.

En esta ocasión me propongo a estudiar, analizar e implementar a grandes razgos un sistema de archivos simple denominado MinixFS, creado por Dr. Andrew S. Tanenbaum para su sistema operativo educacional Minix. La razón por la que elijo este FS es que en el transcurso de la materia ya se estudio el sistema FAT, que es considerado muy viejo y, si bien es muy simple de implementar, las ideas que utiliza fueron ya deprecadas hace mucho tiempo. La diferecia con MinixFS es que éste utiliza inodos, una idea que se explicará con detenimiento más adelante, que hasta el dia de hoy sigue siendo la base de la mayoría de los FS modernos.

Para basar la implementación del sistema de archivos utilizaré una implementación muy arcaica de un núcleo, desarrollado durante el transcurso del cuatrimestre, que inicializa la computadora y contiene manipulación muy básica de scheduler/interrupciones/mmu como para realizar algunas operaciones en la misma. En cuanto a la interfaz del SO con el resto de las aplicaciones, se utilizará un modelo similar al de Linux (es decir utilizaremos el formato POSIX con llamadas a sistema por interrupción 0x80)

Por último, toda este desarrollo no puede ser apreciado si no se cuenta con algunas aplicaciones para utilizarlo y manipularlo, por lo que se introduciran además algunos programas como una consola y una serie de utilidades comunes (ls, rm, touch, cat) para interactuar con el SO y el FS para testear y probar su funcionalidad.

2 Alcance y explicación

2.1 Kernel

El kernel o núcleo con el que contamos es muy limitado, únicamente se encarga de inicializar un sistema x86, setea ciertas estructuras necesarias para iniciar la ejecución de aplicaciones, y finaliza su ejecución.

Las partes más importantes y desarrolladas de este kernel son el scheduler, la atención de algunas interrupciones de usuario que conforman la interfaz del núcleo con el resto del sistema, y el manejo del teclado/pantalla.

La interfaz esta basada en el estandar de POSIX y implementa llamadas como open/close/read/write etc., y de esta manera provee a las aplicaciones de herramientas para crear nuevos procesos, leer y modificar archivos, terminar la ejecución y demás. Estas rutinas estan implementadas en distintos lugares del kernel, ya que cada una toca un sistema distinto. Por ejemplo open/close se encuentran implementadas en el FS, mientras que la familia execve están en el scheduler.

En cuanto a la implementación del scheduler, éste provee mecanismos para agregar y remover procesos de la lista de ejecución, y se encarga de alternar los mismos en el procesador para generar la ilución de multiprogramación. El algoritmo que elegí para desarrollarlo fue el round-robin, que tiene como ventajas su simpleza y escases de problemas graves como starvation.

Por supuesto el sistema no cuenta con una biblioteca estandar de C (implementaciones de la misma como glibc hacen _extensivo_ uso de todas las llamadas de sistema disponibles en POSIX y todos sus flags/agregados/etc, por lo que es imposible soportarla). Por lo tanto las aplicaciones apuntadas a este SO deverán interactuar directamente con el núcleo mediante las llamadas al sistema.

2.2 Minix FS

La parte más importante del trabajo consiste en analizar e implementar el sistema de archivos. Es importante aclarar que la imagen del sistema de archivos se cargará completa en memoria al inicio del sistema, la razón siendo simplemente que no buscamos estudiar ni desarrollar el manejo de disco, mucho menos implementar toda la lógica de un disco IDE/ATA/SATA en un driver que puede ocupar todo un libro.

Utilizando este camino, logramos no solo evitarnos problemas con disco y drivers, sino que todos los cambios y modificaciones a la imagen no necesitan ser pasadas por una cache, simplemente interactuamos con la imagen completa en la memoria. Esto es lo que se logra en los SO modernos al utilizar un ramdisk (aunque los fs en general desconocen esto y interactuan con el ramdisk como si fuera un disco común).

Con respecto a las rutinas de manejo de MinixFS, implementaré todo lo necesario para interactuar normalmente con un sistema de archivos en un SO POSIX, es decir open/close/read/write/lseek/unlink/rename, incluyendo manejo de directorios con chdir/mkdir/rmdir/getdents, entre otras, considerando las flags más importantes de cada uno (por ejemplo O_CREAT/O_APPEND/O_TRUNC en open).

Por último, hay 3 versiones de MinixFS, en este caso implementamos la versión 2. Si bien comenzé con su última versión, me dí cuenta que no existe mucho soporte para la misma en linux (en especial mkfs.minix no sabe crear imagenes versión 3) por lo que decidí adaptar las partes afectadas y remitirme a la versión 2. De todas maneras la diferencia entre ambas es muy pequeña, solo se aumenta el tamaño de algunos campos para soportar mayores discos, y se agregan algunas utilidades no muy útiles para el trabajo.

2.3 Consola y utilidades varias

Todo el desarrollo descripto hasta ahora genera un SO primitivo que bootea la computadora y provee funciones para manipularla. Para poder verlo en acción introduzco algunas aplicaciones que manejan el sistema mediante

llamadas al sistema.

La aplicación más importante es la consola, es la única aplicación que es llamada directamente desde el kernel, y es lo que marca la finalización del booteo y el inicio de las aplicaciones de usuario. La consola permitirá recorrer el sistema de archivos y a su vez lanzar otras aplicaciones.

También se proveen una batería de programas como ls, rm, cat, etc. que simulan a los programas con los mismos nombres que se encuentran en sistemas *nix.

3 Build system

El proyecto del nucleo que se nos presentó en la cursada contiene todo el código y el build system en la misma carpeta. Se basa en un único makefile que se encarga de compilar todo el código, crear la imagen y armar un diskette booteable. Este diskette contiene un boot loader que simplemente carga toda la imagen en la memoria y comienza la ejecución del mismo.

En un principio me propuse a armar un prototipo del FS directamente en la carpeta de nuestro trabajo mini-kernel. Manteniendome fiel a la filosofía Unix/C, comenzé a separar unidades lógicas en distintos archivos para organizar el código.

El problema de esta organización es que no escala bien. Mientras más crecía el código, mayores eran los archivos que se generaban y manejarlo no era práctico. Por esto es que se me ocurrió jugar un poco con Make y generar un simple build system para poder imitar el manejo de directorios de projectos como linux y minix.

En resumen, organizé el codigo en los siguientes directorios:

- apps: algunas aplicaciones para utilizar en el entorno de Mini-Kernel 0.01

- bin: elementos para compilar/testear kernel

- doc: documentacion del trabajo

- fs: codigo del file system

- include: algunos archivos .h utiles

- kernel: el codigo del kernel con funcionalidad super-básica (incluye mmu, interrupciones y hasta el scheduler)

- lib: funciones varias de uso en general (ej mystrncpy, mystrncmp, etc)

Cada directorio con algo de código contiene su propio makefile que se encarga de compilar y armar todo lo que le corresponde. Luego en la carpeta 'bin' se encuentra el makefile central que se encarga de llamar a todos los makefiles de las subcarpetas y linkear el kernel, finalizando con el armado del diskette booteable.

Como el makefile central no sabe que es lo que hace cada sub-makefile (a propósito), y más importante no sabe que archivos objeto crean, cada sub-makefile se encarga de crear un archivo especial que especifica que es lo que crea (que denomine 'depend'). El central simplemente lee este archivo por cada sub-makefile y agrupa todos los archivos objeto para enviarselos al linker.

De esta manera agregar algún nuevo componente al sistema es mucho más cómodo, solo es necesario avisarle al makefile central de la nueva carpeta, y luego ocuparse de crear el sub-makefile únicamente para el nuevo código, sin tener que modificar nada del resto del sistema. Es una manera modular de construir el kernel, evitandose tener un mega-makefile complicado que este constantemente cambiando, siendo difícil entenderlo y editarlo.

Como es costumbre en la mayoría de los proyectos de SO, se expone la carpeta include/ con muchos de los headers de los distintos componentes del núcleo. En particular se tiene include/minikernel/, que contiene las definiciones de estructuras, signaturas y algunos macros que deben ser accesibles desde más de 1 componente (en nuestro caso es más bien para la interoperabilidad entre el kernel y el fs).

Otro detalle del código es el idioma. En la actualidad el idioma predominante para desarrollar es obviamente el inglés, y cualquier aplicación que uno quiere compartir con el mundo por convensión se implementa en inglés, incluyendo nombre de rutinas/variables/comentarios/etc. Es por ésto que antes de comenzar a codificar nada nuevo me aseguré de modificar todo lo que se encuentra en castellano al inglés, para mantener una consistencia con todo el projecto.

Por último, si el lector se encuentra interesado en el ambiente de programación que se utilizo, la edición fue hecha exclusivamente con gvim/vim, la compilación con nasm/gcc/ld, como se explicó recien todo se pega con

make, y para testear el sistema se utilizó la máquina virtual bochs. Si bien al principio el desarrollo fue todo local, cuando la base del código ya estaba ordenada la manejé con git, manteniendo una copia del repositorio en assembla.com (privado ya que el código del kernel utilizado en la cursada no tiene licencia y no sabía si podía o no publicarlo en una página pública como github, que maneja mucho mejor a git). Para la programación del kernel en especial se hizo mucho uso de el programa objdump, para decompilar los ejecutables y ver exactamente el código producido.

Para probar el kernel, simplemente ir al directorio 'bin', armar el diskette booteable con 'make', y ejecutarlo en una máquina virtual con 'bochs'.

4 Mini-Kernel

El orden en que escribí el código fue primero el FS, luego el kernel y por último las utilidades. Sin embargo para hacerlo más consistente se presentará primero el kernel, que pega todos los componentes y facilita la explicación del FS y las utilidades.

El kernel está basado en el utilizado en la cursada de Organización del Computador 2, 2do cuatrimestre del 2010. En este sistema armamos las estructuras necesarias para bootear un sistema en modo protegido con paginación. Esto incluye un bootloader que carge la imagen, el manejo del A20, y llenado de GDT/IDT/TSS etc. Esto permitía manejo de interrupciones y dio lugar a un simple scheduler que daba una forma primitiva de multiprogramación.

Si bien partí de esta base de código, muchos de los componentes fueron modificados o reescritos totalmente. Los cambios más importantes fueron en el scheduler (conjunto con el manjeo de tss), el manejo de procesos, el mmu o manejo de la memoria y el manejo del teclado/pantalla.

Ya teniendo un núcleo booteable con el que trabajar me hizo pensar que los cambios no iban a requerir demasiado trabajo. Estaba seriamente equivocado. Si bien grandes partes del código las implementaba rápido y bien en general en la primera pasada, eran los pequeños errores, un bit incorrecto en el lugar menos esperado, que me robaba horas (o hasta días algunos casos) intentando de arreglar. Esto me hizo recordar la ley de Hofstadter: "It always takes longer than you expect, even when you take into account Hofstadter's Law"

Antes de comenzar a hablar sobre cada componente en detalle, voy a dar algunas características generales del kernel. Este núcleo utiliza la idea de "Lower Half Kernel", o kernel en la parte baja de la memoria. Esto significa que la imagen del kernel se copia y mapea virtualmente en los primeros 4mb de la memoria. En un principio pensé utilizar la idea de "Higher Half Kernel" en donde el mapeo se hace al final de la memoria (en general al rededor de los 3gb), que es la que se utiliza en las kernels modernas (linux, windows, bsd, etc), haciendo la carga de los programas más simple y por sobretodo manteniendome en el standar. Sin embargo debido a no tener control sobre el bootloader, y no siendo este compatible con multiboot, me fue imposible controlar la carga de la imagen y por ende abandoné la idea.

Las implicaciónes de esta decisión fueron que al compilar las aplicaciónes de usuario, estas deben empezar con el offset de 4mb, o 0x400000 en hexa, para que tengan sentido todos los punteros. Para ver más sobre esta explicación, dirigirse a la sección "Manejo de Procesos" y "Utilidades" al final del informe.

Desde un primer momento me propuse implementar una interfaz con el usuario del tipo POSIX, es decir utilizar al igual que linux/bsd la interrupción 0x80 para manejar llamadas al sistema. Respeté todos las signaturas de las llamadas POSIX (exit/open/close/write/etc.), sus números de interrupción, logrando de esta manera tener un sistema "compatible" con linux. Si bien obviamente esta compatibilidad es muy reducida, en la sección de utilidades se mostrará como todas las aplicaciónes programadas para esta kernel funcionan sin ninguna modificación en un sistema linux.

Otro punto importante es el uso de sys/queue.h. Este header extraido de minix (que a su vez fue extraido de NetBSD) contienen una implementación estándar varios tipos de listas (comunes, doble enlazadas, circulares, etc). Esto es utilizado en varias partes, incluyendo el scheduler, el manejo de memoria, y en el fs. Para entender más sobre esto dirigirse a sus respectivas páginas en el manual ("man queue").

4.1 Biblioteca General

Al codificar los distintos componentes del núcleo, me di cuenta rápidamente que existía un set de funciones básicas que eran útiles en muchas parte del código. En su gran mayoría estas funciones eran imitaciones de aquellas en la biblioteca estándar de C.

Es por esto que agregé al build la carpeta lib/, que contiene los archivos lib/misc.c y lib/misc.h. Aquí

implementé algunas de las funciones de libc que necesitaba en más de un lugar, y eran lo suficientemente genéricas como para que se puedan reutilizar. Decidí mantenerles el nombre que tienen en libc, agregandole el prefijo "my". Si bien esto no corresponde extrictamente a la explicación del kernel, me pareció correcto nombrarlas en este momento ya que se verán a través de todo el código del kernel/fs/utilidades.

Así es como nacieron mystrlen(), mystrncmp(), mystrncpy(), mymemcpy(), mymemset(), MIN() y space(). Todas estas funciones se comportan de la manera que uno espera que funcionen, son compatibles con sus clones.

4.2 Manejo de Procesos

El manejo de procesos en el kernel se encarga de mantener información sobre todos los procesos actualmente en el sistema, información relevante para el scheduler/tty/fs etc. La implementación del mismo se encuentra en los archivos kernel/sched.c y kernel/sched.h. Es importante notar que en el código, no hay distinción entre manejo de procesos y scheduler, ambos se encuentran en el mismo archivo.

Cada proceso en el sistema ocupa una pocisión en el arreglo "ps", de tipo "struct process_state_s" definido en include/minikernel/sched.h. Esta estructura contiene información como el pid/uid/gid del proceso, un puntero al padre, la información del fs (directorio actual, descriptores de archivo abiertos, etc), una serie de entradas de manejo de listas, y una lista de páginas utilizadas. Toda esta información, más el estado del proceso guardado en su respectivo TSS, define el estado de cualquier proceso en el sistema:

```
struct process_state_s {
    /* number */
    int i;

    /* process id */
    pid_t pid;

    /* parent */
    struct process_state_s *parent;

    /* process owner's uid and gid */
    pid_t uid;
    pid_t gid;

    /* data to keep track of waitpid sys call */
    struct process_state_s *waiting;
    int *status;
    pid_t child_pid;

    /* fs data */
    struct unused_fd_t unused_fd;
    struct file_s files[MAX_FILES];
    unsigned int curr_dir;

    /* dev io data */
    unsigned int dev;

    /* scheduler ready list pointers */
    CIRCLEQ_ENTRY(process_state_s) ready;
```

```
    /* scheduler waiting list pointers */
    LIST_ENTRY(process_state_s) wait;

    /* unused list pointers */
    LIST_ENTRY(process_state_s) unused;

    /* list of used pages */
    LIST_HEAD(pages_list_t, page_s) pages_list;
} __attribute__((__packed__)) ;
```

En este archivo son definidas los handlers de todas las llamadas al sistema que manejan procesos. Estas incluyen exit()/waitpid()/getpid() entre otras.

La llamada exit() se encarga de liberar todas las páginas pedidas por el proceso, removerlo de la lista de procesos listos del scheduler, despertar al padre si este estaba dormido esperandolo (con waitpid()), y programar el siguiente proceso a ser ejecutado.

waitpid() recibe un pid de un proceso hijo, y bloquea al proceso que ejecuto la llamada hasta que no termine el proceso hijo. Es utilizado por ejemplo en la consola para esperar a que un comando enviado termine de ejecutarse. Logra esto removiendo al proceso de la lista del scheduler, y apuntando en su estructura al proceso que espera que termine. Lo despierta exit(). La forma de encontrar al procesos hijo por su pid es mediante la función find_pid(), que recorre la lista de procesos buscando el deseado (recordando a Ken Thompson, uno de los padres de Unix, When in doubt...).

getpid() simplemente devuelve el pid de la estructura "ps".

Obvié intencionalmente a las dos llamadas más importantes, fork() y execve(). En un principio comencé a utilizar este modelo para crear nuevos procesos, pero resultó tener varias complicaciónes que no esperaba (en especial cuando se realiza el fork() y no se sigue con el execv(), copiar el estado de un proceso no era tan simple), por lo que decidí unir ambas llamadas y crear una nueva, denominada newprocess(). Esta recibe los mismos parametros que execv(), el path del proceso y el arreglo de argumentos argv, y realiza todo sola, crea el proceso y lo prepara para ser ejecutado. Esta es la única razón que hace que una aplicación ("cash", la consola) no compile directamente para un sistema linux, es necesario reemplazar la llamada a newprocess() por un fork() y un execv() y se logra la compatibilidad.

newprocess() es la llamada más complicada del manejo de procesos, y consiste en los siguientes pasos:

- consiguir una entrada libre en el arreglo "ps"

- buscar en el sistema de archivos el inodo correspondiente al path pedido por el usuario

- inicializar la estructura process_state_s con datos como pid/uid/gid, parent, se inicializan los file descriptors del proceso y la lista de páginas utilizadas (todavía ninguna).

- armar el nuevo directorio de páginas o PDT del proceso. Esto implica armar un PDT con los primeros 4mb con identity mapping (para el kernel). Además se arman todas las páginas de código/stack/argumentos necesarios, ver más abajo para la explicación.

- llenar la entrada correspondiente en la tss para el proceso con esta nueva PDT

- agregar el proceso a la lista ready del scheduler

Esto concluye la carga de un programa y este se encuentra listo para ejecución.

El armado del PDT tiene un par de etapas, y es necesario saber algunas reglas que decidí para construirlo. Primero en principal, todo el código se carga a partir de la dirección 0x400000, es decir se copia el codigo en una serie de páginas que pueden estar físicamente en cualquier lado, pero que deben mapearse en orden a partir

de esta dirección. Esto evita tener que responder Page Faults (todo se copia de antemano). Luego la stack esta mapeada en la última posición posible, es decir la 0xFFFFF000, y le sigue la stack del kernel (necesaria para atender interrupciones) en 0xFFFFE000. De nuevo, estas páginas son pedidas al MMU y pueden estar en cualquier lado, pero la PDT debe encargarse de mapearlas en estas direcciones.

Por último se agregó soporte para argumentos, que en C corresponden a los parametros argc y argv del main(). No sabía muy bien como era que se les entregaban argumentos a un programa de C por lo que recurrí a una herramienta que resulto indispensable en el proyecto, objdump. Decompilando un ejecutable común de C pude ver que main() espera estos argumentos como cualquier otra función, en la pila antes de la dirección de retorno. Esto implica que al crear un proceso, debía armar su pila de manera que main() recibiera bien estos parametros.

La otra complicación que surgió fue que, al crear un proceso, uno provee el argv ya armado. Pero este arreglo es de punteros a cadenas y las cadenas mismas son datos del proceso que realiza la system call, pero el nuevo proceso no comparte el espacio de datos del viejo, por lo que no puede accederlas. La solución fue crear una nueva página en newproces() que mapee en 0xFFFFD000, y copiar todas las strings de los argumentos ahí. Luego armar en esta misma página un nuevo argv con los punteros a estas nuevas strings. Por último puse el puntero a este argv en el stack del proceso, logrando así la compatibilidad con C.

4.3 Scheduler y TSS

El scheduler también se encuentra en los archivos kernel/sched.c y kernel/sched.h. Este comprende las funciones init_scheduler()/schedule()/block_process()/unblock_process().

La función init_scheduler() debe ser ejecutada antes de que se activen las iterrupciones en el sistema. Esta se encarga de inicializar las listas necesarias para la programación de tareas (ready/waiting), inicializa los tss, y crea la tarea idle, que será ejecutada siempre que la lista ready este vacía.

schedule() es la función más importante en el scheduler, es llamada siempre que ocurre la interrupción del clock y se encarga de decidir cuál es el siguiente proceso a ser ejecutado. Para simplificar la lógica de la misma decidí implementar la lista de procesos listos como una circular, eligiendo efectivamente el método round-robin para el scheduler. Esta función luego se fija si hay procesos listos, y en caso de haberlos los ejecuta uno por uno. En caso contrario vuelve al estado idle. Todo es logrado con sys/queue.h. Básicamente hay 3 posibilidades, que se pida un cambio de contexto (que en nuestro sistema es inferido por current_process siendo NULL), que no halla procesos para ser ejecutados, o que haya más de uno (current_process denota el proceso que se esta ejecutando actualmente, el que fue interrumpido):

```
/* no process running */
if (current_process == NULL) {
    /* if any process ready then execute, otherwise go idle */
    if (!CIRCLEQ_EMPTY(&ready_list)) {
        process = CIRCLEQ_FIRST(&ready_list);
        current_process = process;
        load_process(process->i);
    } else {
        current_process = IDLE;
        load_process(1);
    }
/* if we are idle, check for new processes */
} else if (current_process == IDLE) {
    if (!CIRCLEQ_EMPTY(&ready_list)) {
        process = CIRCLEQ_FIRST(&ready_list);
```

```
        current_process = process;
        load_process(process->i);
    }
/* if there are more than 1 process ready */
} else if (CIRCLEQ_NEXT(current_process, ready) !=
           CIRCLEQ_PREV(current_process, ready)) {
    current_process = CIRCLEQ_NEXT(current_process, ready);
    load_process(current_process->i);
}
```

Por último las funciones block/unblock_process() fueron creadas para bloquear procesos en espera de dispositivos. Como ya fue explicado no se hace uso de discos ni ningun tipo de dispositivo que requiera bloquear procesos, por lo que en un principio no fueron necesarios. Sin embargo al progamar el manejo de teclado y pantalla me di cuenta que al realizar un "read()" al stdin, era necesario justamente bloquear al proceso y esperar una respuesta del teclado, por lo que estas funciones entraron en juego. Simplemente sacan de la lista ready a un proceso y lo mandan a wait, y cuando el dispositivo que lo bloqueo avise que volvió, vuelven a agregar el proceso en ready. Más sobre este comportamiento en la sección Teclado y Pantalla.

4.4 MMU

La implementación del manejo de memoria se encuentra en los archivos kernel/mmu.c y kernel/mmu.h.

Por casi toda la vida del núcleo el mmu fue el más simple posible. Preferí no agregar complicaciones al resto del kernel, que ya era suficientemente frágil. Luego la función new_page(), que devolvía una página libre, nunca liberaba páginas, siempre daba nuevas. El problema de esto era obvio, era un enorme memory leak en el sistema.

Sin embargo cuando ya tenía un sistema funcionando me propuse a implementar una forma simple de manejo de memoria, donde los procesos manienen una lista de las páginas que utilizan (ya sea para codigo/datos/stack o para las PDT/PTT), y estas son liberadas cuando el proceso termina. Nuevamente se utilizo sys/queue.h para armar estas listas. La función new_page() recibe el número de proceso pidiendo la página, y la agrega a su lista de páginas usadas (en el arreglo de process_state_s).

Además se proveen las funciónes init_directory(), que crea un PDT inicial que solo tiene identity mapping en los primeros 4mb (la parte del kernel), y dos funciones claves: map_page() agrega en un PDT la transformación de una dirección virtual en una física, y unmap_page() remueve esta transformación. Estas son las funciones utilizadas para armar el directorio de páginas de un proceso.

4.5 Teclado y Pantalla, o el driver tty

El problema de implementar un file system es que este no se puede ver. En otras palabras, su funcionamiento se ve através de las aplicaciónes que lo usan. Y de la misma manera, no podemos ver estas aplicaciones sin una manera de interactuar con la pantalla. Es por esto que programe un driver tty para manejo de pantalla y teclado.

El código de este driver se encuentra en kernel/keyboardscreen.c y kernel/keyboardscreen.h.

Para el manejo de la pantalla se utilizó el modo SVGA estándar de todas las computadoras x86. Escibiendo bytes de caracteres ASCII y colores en posiciones de la RAM de la placa de video mapeadas en la memoria principal (en la dirección 0xB8000), se logra imprimir caracteres en la pantalla. De esta manera cree la función print_key() que recibe un caracter ascii y lo imprime en la pantalla.

Para lograr esto es necesario mantener un cursor a la última posición escrita, de manera que el texto sea continuo. Además hay que tratar a algunos caracteres de manera especial, por ejemplo si recibimos el caracter '\n', actualizamos el puntero para que apunte a la siguiente linea. En caso de que nos quedemos sin lineas, se copian todas un linea hacia arriba para hacer lugar a una linea nueva. Todo esto es manejado por print_key().

Además, para agraciar un poco más la pantalla, agregé el manejo del cursor. Esto requiere actualizar unos registros de I/O (0x3D4 y 0x3D5) cada vez que se quiere cambiar la posición del mismo. Esto está implementado en la función move_cursor(). Uniendo todo esto tenemos (vram apunta a 0xB8000):

```
void print_key(char key)
{
    switch (key) {
        case '\n':
            x = xlimit = 0;
            if (y == 24)
                scroll_up_vram();
            else
                y++;
            break;
        case '\b':
            if (x > xlimit) {
                x--;
                (*vram)[y][x].letter = 0;
            }
            break;
        case '\t':
            key = ' ';
        default:
            (*vram)[y][x].letter = key;
            (*vram)[y][x].color = WHITE;
            x++;
            if (x == 80) {
                x = xlimit = 0;
                if (y == 24)
                    scroll_up_vram();
                else
                    y++;
            }
    }
    move_cursor(x, y);
}
```

En el código, x e y dan la posicion actual del puntero, y xlimit es una manera de marcar el último lugar en la linea actual que puede ser borrado. Esto es para evitar por ejemplo en la consola que el usuario borre el PS1, solo puede borrarse lo que se ingresó por el teclado.

Luego se provee una función print() que recibe una cadena de caracteres y los imprime en la pantalla, uno por uno utilizando print_key().

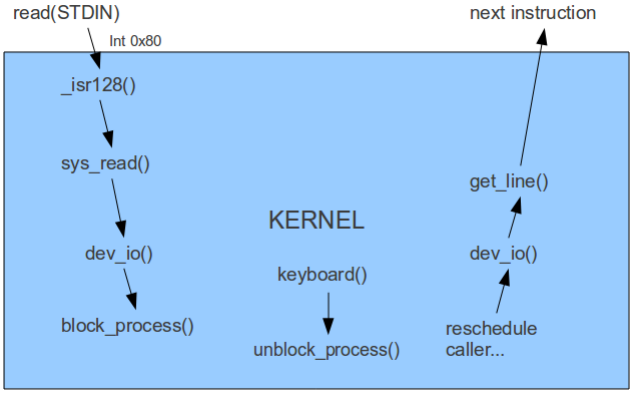
La forma en que trabajan las consolas es que toda tecla presionada en el teclado debe reproducirse en la pantalla. Recibimos las teclas como interrupciones, y las atendemos con la función keyboard(). El problema es que el estándar de IBM hace que recibamos teclas en modo scancode y no ASCII. Para esto cuento con un

arreglo que sirve de traductor, para cada posición devuelve el ASCII correspondiente a un scancode. Luego keyboard() simplemente hace esta conversión y se la envía a print_key() que la replica en la pantalla.

Así logramos el manejo del texto en la pantalla, pero además necesitamos una manera de recordar las teclas ingresadas por el usuario para luego darselas a la aplicación que intente leer STDIN. De esta manera keyboard() guarda todas las teclas ingresadas en un arreglo que hace las veces de buffer. En este caso opté por un buffer circular para ahorrarme los problemas de quedarme sin espacio (aunque con el tamaño elegido no parece ocurrir nunca). Utilizo como separador la tecla '\n', y cada ves que se pide leer STDIN, se lee una linea de este buffer. Esto se logra llamando a la función get_line().

4.6 Resultado

El 'desafío' en esta sección es responder a una llamada del tipo "read(STDIN_FILENO, buf, len)" satisfactoriamente. De esta manera se podrán ver muchos de los mecanismos que se describieron anteriormente. A continuación detallo los pasos que sigue el kernel para responder esta syscall (de paso se ve el proceso de atención de cualquier system call):



- El proceso ejecuta "read(STDIN_FILENO, buf, len)", que consiste en poner los valores correspondientes en ebx, ecx y edx, poner el número de la system call en eax, y realizar un "int 0x80". Más sobre esto en la sección "Utilidades".

- Se atiende la interrupción en kernel/isr.asm, en la etiqueda _isr128 (128 = 0x80). Esta función se encarga de guardar el valor de los registros, armar la pila con los parametros y llamar a la rutina de atención correspondiente al número ingresado en eax.

- En este caso caemos en sys_read(), que se encuentra en fs/fs.c. Esta función recobra el inodo correspondiente al descriptor de archivo ingresado (STDIN_FILENO = 0, que es un fd inicializado cuando se creo el proceso, apuntando al inodo /dev/stdin). Se pregunta si el inodo corresponde a un dispositivo de caracteres, que resulta ser verdadero en este caso, y en consecuencia llama a dev_io(), la función que hace I/O de dispositivos.

- dev_io() se encuentra en kernel/dev.c. Actualmente esta función solo atiende peticiones para STDIN, STDOUT ó STDERR (aunque debería ser genérica y atender cualquier dispositivo de entrada/salida). En este caso el dispositivo es STDIN, y como se pide leer del mismo, se llama a la función block_process().

- block_process() se encuentra en kernel/sched.c, proceso se bloquea en este punto, esperando la respuesta del dispositivo en cuestion (el teclado). Esto simplemente significa remover al processo de la lista ready, agregarlo a la lista wait, y llamar a schedule() para dar lugar a otro proceso. En nuestro sistema esto probablemente significa que pasamos al estado idle.

- El proceso se mantiene bloqueado, hasta que por el teclado es enviada la tecla '\n' (el enter). Esta es nuestra señal para enviar los datos al proceso que los pidio. Por ende la función keyboard() (que es la que recibió la interrupcion con el '\n' del teclado) llama a unblock_process(), que desbloquea al primer proceso que estuviese esperando un dispositivo, en este caso el teclado.

- unblock_process() tambien se encuentra en kernel/sched.c, busca en la lista de procesos bloqueados el primero correspondiente al dispositivo en cuestion, y lo agrega nuevamente a la lista ready. Luego sigue la ejecución del proceso que fue interrumpido por el teclado.

- Llega el momento en que el scheduler programa nuevamente al proceso que inició la llamada read(). Este vuelve al punto en donde se bloqueo, es decir en dev_io(). La siguiente linea luego de la llamada block_process() es una llamada a get_line(), que esta garantizada que funcionará ya que fuimos despertados por el teclado por esta razón. Luego se llena el buffer con lo recibido por la consola, se recobra el estado de los registros y se realiza el anticipado iret de _isr128().

Esto concluye la atención a esta llamada al sistema.

Resumiendo, el kernel realiza en el booteo la inicialización de los siguiente componentes en orden (ver kernel/kernel.asm): A20, GDT, IDT, Modo Protegido, System calls, MMU, Paginación, Clock, File System, Scheduler y habilita interrupciones. Antes de habilitar interrupciones se agrega a la lista ready un proceso "cash" (crappy *** shell), la shell que termina manejando el hilo de ejecución. Luego el sistema esta en manos del usuario.

5 Minix FS

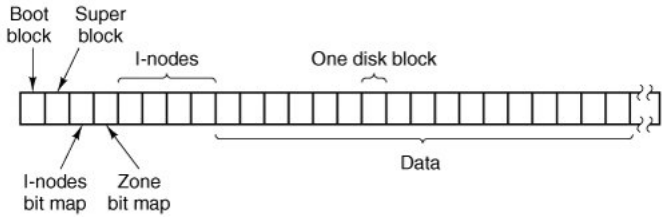
El sistema de archivos conforma el componente más trabajado de este núcleo. Como ya fue explicado, elegí implementar el sistema de archivos MinixFS V2. A continuación daré una explicación a grandes razgos de todos las características de este FS (la mayor parte de este explicación esta basada en el libro de Tanenbaum "Operating Systems: Design and Implementation", capítulo 5)

Comenzaré contando un poco la organización y distribución de la información en minix, el superblock y manejo de blockes, poniendo la mayor atención en la idea de inodos. A partir de ahora llamaré a el sistema de archivos MinixFS v2 como mfs.

Doy por sabido que el lector maneja conceptos básicos de computación a bajo nivel, por ejemplo saber que es un bloque o un mapa de bits.

5.1 Organización en MinixFS

Todos los sistemas de archivos modernos presentan una idea de organización de la información, acompañada con la forma en que se guardan en el disco las estructuras y datos necesarios para mantener esta idea. En el caso de mfs se cuentan con una serie de estructuras que dan atributos y parametros del mismo, seguido por el espacio vacio donde se guardan los datos en sí. La estructura esta compuesta por: Boot Sector, Super Block, Inode Map, Zone Map, Inodes y Empty Blocks. Esta distribución se puede apreciar en la siguiente imagen:



El Boot Sector o bs es un bloque de tamaño fijo (1024 bytes) que se encuentra obligatoriamente al comienzo de la imagen de mfs. El uso de este bloque es una convensión en la computación para comenzar el booteo del sistema, es el punto de partida de la ejecución luego de que la BIOS termina la inicialización. Es por esto que este bloque no es de mayor importancia para nosotros, simplemente hay que tenerlo en cuenta a la hora de calcular offsets para el resto de los bloques.

El Super Block o sb, que al igual que el bs ocupa exactamente un bloque de tamaño fijo (1024 bytes). Mfs, al igual que todo el resto de los sistemas de archivos, tiene una serie de atributos que pueden variar dependiendo de las necesidades del administrador del sistema (por ejemplo el tamaño de los bloques). Todos estos valores se encuentran en el sb, incluyendo información necesaria como la cantidad de bloques o inodos, el tamaño máximo de un archivo, etc. Se verá con más detenimiento es la sección correspondiente.

El Inode Map o im y Zone Map o zm. Estos dos mapas de bits son usados por mfs para marcar cuales de las posiciones de memoria reservadas para inodos y para datos estan usadas, im para inodos y zm para datos. En MinixFS se hace una distinción entre Zone o zona y Block o bloque, donde una zona es un conjunto de bloques (una potencia de dos, por ejemplo cada zona puede tener 4 bloques). La razón por la que utilizaron esta diferenciación será explicada más adelante, pero para el trabajo que estamos haciendo esta diferencia fue evitada asegurandose de que las zonas contengan un solo bloque.

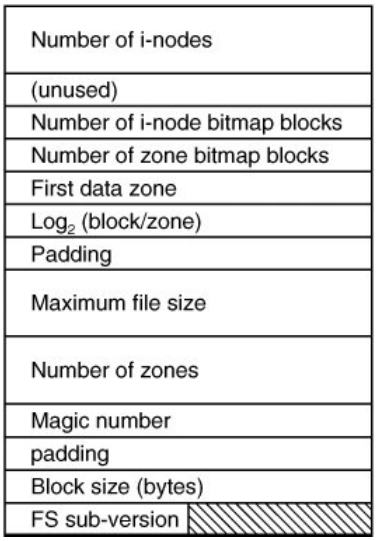
La parte siguiente es la más importante del sistema, y corresponde al espacio de memoria reservado para los inodos. En la sección correspondiente se explicará su funcionamiento, pero anticipando, todo archivo esta

descripto por un inodo (aunque puede ser más de uno) que da algunas características del mismo, y una forma de ubicar los bloques con sus datos.

Por último se encuentra la parte más grande de la imagen mfs que corresponde a todos los bloques restantes, libres para guardar información. Por lo general conforman el contenido de los archivos en varios bloques distribuidos por el espacio libre, aunque en algunos casos particulares tienen información de mfs (ver el funcionamiento de los directorios).

5.2 Super Block

El Supero Block o sb contiene toda la información sobre las características de una imagen mfs en un bloque de tamaño 1024 bytes. En la siguiente imagen vemos la distribución de la información en el bloque:



La mayoría de los campos son suficientemente descriptivos como para explicarlos, basta con afirmar que los únicos que no utilizamos son "FS sub-version" que no juega ningun papel en la versión 2 de mfs, y "$Log_2(block/zone)$" ya que no hacemos distinciones entre zonas y bloques (este valor debe ser 0).

Todo el código de manejo del sb se encuentra en los archivos fs/super.c y fs/super.h.

La primera función que debe ser llamada antes de poder interactuar con el sb es read_super(), que se encarga de leer el Super Block. En nuestro caso nos basta únicamente con apuntar al inicio del sb en la memoria, ya que como explicamos anteriormente toda la imagen mfs ya esta cargada en memoria. La estructura superblock_s representa al sb con los tamaños exactos como aparece en la imagen, por lo que un puntero de este tipo nos ahorra toda necesidad de tener que reservar espacio para la estructura y copiar los datos en la misma.

Luego la manera de interactuar con el sb es mediante defines. Una serie de definiciones en el super.h acceden a la estructura superblock_s para devolver los datos. Una forma más "correcta" de hacer esto hubiese sido utilizar getters y setters, asegurandonos en cada uno que el puntero al sb ya se hubiese inicializado. La razón por la que no elegí este método es que read_super() es un método obligatorio para cualquier implementación de mfs y en caso de error la implementación no podría seguir, por lo que es seguro asumir que no existe este error y obviarlo. De esta manera nos ahorramos el overhead de las llamadas a los getters y setters y trabajamos directamente con los punteros. Un ejemplo es:

```
#define IMAP_BLOCKS (sb->s_imap_blocks)
```

Con 'sb' el putero a estructura superblock_s. Este define recupera la cantidad de bloques que ocupa el mapa de inodos.

5.3 Inode y Zone Maps

El mapa de inodos se encarga de mantener un registro de cuales de los espacios reservados para inodos se encuentran ocupados. El mapa de zonas provee la misma funcionalidad pero a nivel de bloques. Ambos son bitmaps en los cuales cada posición posible de guardar un inodo/bloque está representado por un bit, 0 si se encuentra libre y 1 en caso contrario.

El código para el manejo de estos mapas se encuentra también en los archivos fs/super.c y fs/super.h.

En el caso del mapa de inodos hay 2 funciones lo manejan: rm_inode() y empty_inode(). La primera se encarga de liberar la posición del inodo asigandole un 0 al bit correspondiente. La segunda devuelve una posición libre para crear un nuevo inodo. Logra esto manteniendo siempre un puntero a la última posición libre del bitmap, y luego recorriendo el mapa hasta encontrar un bit vacío.

Para el mapa de bloques existen 2 funciones similares: rm_block() y empty_block(). Estas funciones se comportan de manera similar a las de inodos, por lo que en realidad ambas son wrappers que llaman a un par de funciones genericas free_bit() y alloc_bit() que son las que realizan el trabajo descripto.

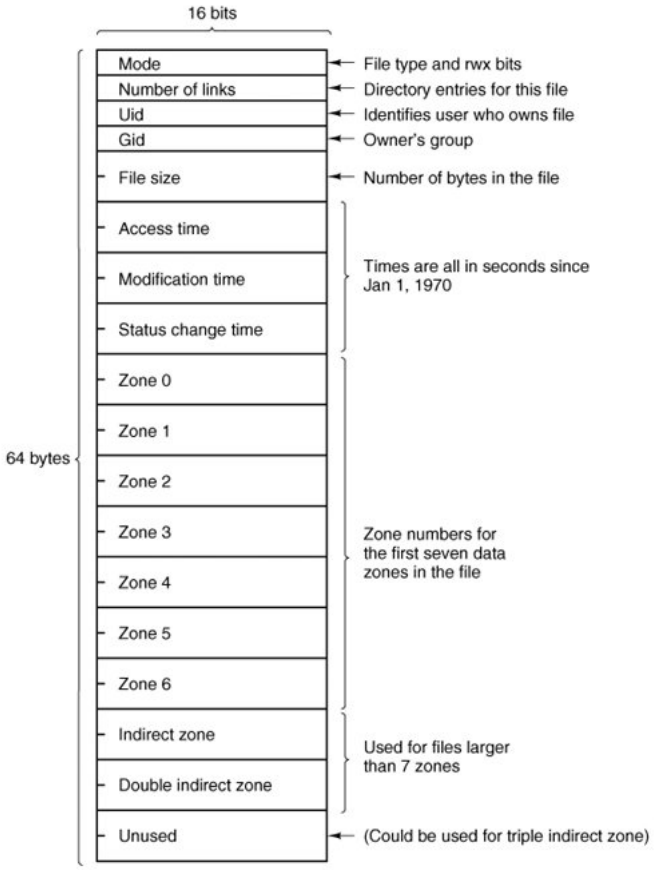
La función alloc_bit() que encuentra un bit vacío en el bitmap es parecida a una en el código de minix con el mismo nombre. Ambas usan una técnica simple para rastrear ese 0 en el mapa. Recorren el bitmap de a palabras, y las comparan con la palabra de todos 1's, logrando leer de a 32 bits por repetición del ciclo. Al encontrar una palabra que es distinta a la completa por 1 solo hace falta hubicar el 0 que sabemos existe.

5.4 Inodes

La parte más interesante de un sistema de archivos tipo Unix como lo es mfs es la idea de inodos. Básicamente todo archivo (y como archivo contemplamos también a directorios/pipes/links/etc.) es representado por un inodo. Un inodo esta compuesto por una serie de campos que definen el archivo, y luego referencias a los bloques en donde se encuentran sus datos, en el caso de tenerlos.

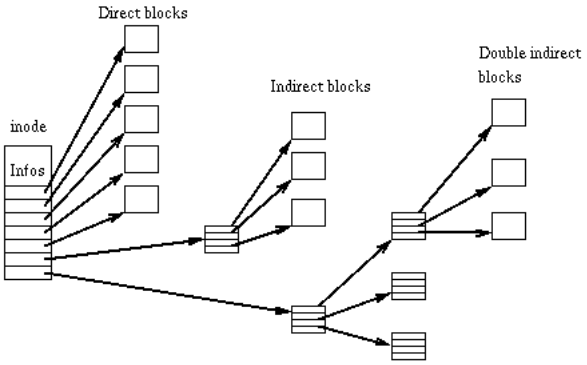
La función principal de un inodo es referenciar todos los bloques que conforman un archivo. Lo bueno de este método es que al abrir y leer un archivo, solo es necesario mirar a su inodo, éste tiene toda la información necesaria para leerlo. Esto es una mejora substancial comparado con sistemas del tipo FAT, que requieren de una tabla aparte con la información de todos los archivos. En este último caso una tabla de este estilo sería enorme para los tamaños de los discos en la actualidad, y ocuparía varios Mbyes en memoria constantemente. En cambio con inodos solo es necesario mantener en memoria aquellos que se encuentran actualmente en uso, aprovechando mejor los recursos.

A continuación se muestra un gráfico con la estructura general de un inodo en mfs (similar a los inodos en ext2 o cualquier otro fs tipo unix):



Esta imagen (al igual que el resto de las utilizadas en esta sección MinixFS) fue tomada del libro de Tanembaum, y contiene descripciones de los campos que conforman el inodo: 'mode' que nos da el tipo de archivo y sus atributos de lectura/escritura/ejecución; 'links' nos dice cuantos directorios contienen al archivo (es decir cuantos 'hard links' existen); 'uid/gid' el dueño y su grupo; 'size' el tamaño; 'access/mod/status time' tiempos varios; 'zone 0-6' los punteros a los primeros 7 bloques del archivo.

Paramos aquí para comentar una característica de los inodos. Para poder contar con archivos suficientemente grandes, se implemento la idea de bloques indirectos/doble indirectos/triple indirectos. En caso de que la información del archivo supere el espacio que proveen los primeros 7 bloques referenciados en el inodo, se pasa al bloque de indirectos, que consta de un bloque completo con punteros a otros bloques donde sigue el archivo. En caso de que éste último no alcanze, se repite la idea y se utiliza el doble indirecto, que apunta a un bloque con punteros a otros bloques, que a su vez contienen punteros a bloques con la información. Para entender mejor el funcionamiento ver el siguiente gráfico:



De esta manera se logra expandir el límite del tamaño de un archivo los suficiente como para que no sea un problema. En el caso de mfs los bloques triple indirectos no son utilizados, pero el espacio en el inodo existe por si fueran a ser necesarios en el futuro.

Veamos ahora como se implementan en nuestro sistema. El código de manejo de inodos se encuentra en los archivos fs/inode.c y fs/inode.h, con partes menores en fs/fs.c.

La función más interesante en el manejo de inodos es find_inode(), que dado un directorio de inicio y un path tipo unix (ej. jp/orga2/informe.pdf), encuentra y devuelve el inodo buscado en caso de existir (en nuestro ejemplo sería el inodo correspondiente al archivo informe.pdf). La implementación permite que se pueda borrar el archivo en cuestion, se cree, o simplemente que se devuelva. Por ejemplo, para conseguir el inodo correspondiente al archivo '/home/jp/hola.txt', bastaría con llamar a la la función find_inode(NULL, "/home/jp/hola.txt", FS_SEARCH_GET).

Luego de decidir por donde empezar la busqueda (si el path empieza con '/' empezamos en el root directory), esta función se encarga de ir componente por componente navegando los directorios del path y avanzando hasta el último subdirectorio. Logra esto utilizando la función search_inode(), que dado un directorio y un nombre de archivo, devuelve el numero inodo correspondiente, en caso de que exista.

Analicemos más de cerca esta última función. Comienza recorriendo un directorio entrada por entrada buscando aquella que coincide con el nombre de archivo/carpeta buscado. Hace uso de varias funciones auxiliares que detallaremos a continuación. El siguiente extracto de código de search_inode() muestra el loop principal (simplificado):

```
pos = 0;
entries = dir->i_size / DIRENTRY_SIZE;
for (i = 0; i < entries; ++i) {
    if ( (dentry = next_entry(dir, &pos)) == NULL)
        break;

    if (dentry->num != 0)
        if (mystrncmp(name, dentry->name, MAX_NAME) == 0)
            return dentry;
}
```

La manera en que el código recorre las entradas del directorio es mediante la función read_map(), que dado una posición de un archivo/carpeta, devuelve el bloque en que se encuentra (es decir, resuelve bloques directos/indirectos/etc.); esta es utilizada por next_entry() para recorrer los bloques del directorio. Así vamos bloque por bloque recorriendo todas las entradas, comparando una por una con la función mystrncmp (una

imitación de strncmp de la biblioteca estandar, recordar que no contamos con ninguna de estas funciones). En el caso de que no se encuentre la entrada que se buscaba, search_inode() devuelve NULL. En el caso de que necesite una entrada vacía, se provee la función empty_entry().

Es importante aclarar que hay una diferencia clave con la manera en que está implementado mfs en minix. Funciones como get_block(), que dado un número de bloque, devuelve el puntero al mismo, requieren llamar al disco, recobrar el bloque en cuestión y guardarlo en memoria para poder utilizarlo. En cambio nosotros simplemente apuntamos a la posición de dicho bloque, ya que todo se encuentra pre-cargado. Lo mismo sucede con los inodos y la función get_inode(), en la implementación real del fs se mantiene una cache de inodos que contiene todos aquellos que se están utilizando en ese momento, lo cual no es necesario en nuestro caso por el mismo motivo.

5.5 System Calls del FS

Los sistemas tipo unix utilizan la idea de descriptor de archivo o 'fd' para identificar archivos abiertos. Cuando un programa quiere abrir un archivo, se le es entregado este fd, que no es más que un número identificador. Luego basta utilizar este número para realizar cualquier acción sobre el archivo (leer/escribir/etc.).

Las funciones de manipulación de fds se encuentran en los archivos fs/file.c y fs/file.h. Cada proceso en su tabla contiene un arreglo con los fds y sus respectivos inodos. Mediante las funciones get_fd() y release_fd(), uno reserva y devuelve fds según las necesidades del proceso.

La implementación de estas funciones hace uso del archivo sys/queue.h, una implementación estándar de listas para los sistemas unix. Declarando un par de punteros en el arreglo de fds, simulamos una lista tipo fifo con la que nos aseguramos que pedir y devolver fds sea siempre $O(1)$. En el caso de get_fd() tenemos:

```
int get_fd(ino_t ino_num, unsigned int pos)
{
    struct file_s *file = LIST_FIRST(&unused_fd);

    if (file != NULL) {
        LIST_REMOVE(file, unused);
        file->ino = ino_num;
        file->pos = pos;
        return file->fd;
    }

    return ERROR;
}
```

De esta manera podemos utilizar la idea de listas sin tener que agregar complicaciones al código, y más que nada no tenemos que introducir memoria dinámica, ya que los arreglos que simulan la lista son estáticos y por ende fijos.

Con el manejo de fds completo, comienza el código que rearma todo lo ya explicado para implementar las llamadas al sistema. Para cada system call unix, la función que la recibe tiene el prefijo "sys_" (por ejemplo, la función sys_open() atiende la llamada open()). Se implementaron las llamadas open(), close(), write(), read(), lseek(), unlink() y rename() para manejo de archivos, chdir(), mkdir(), rmdir() y getdents() para manejo de carpetas.

Las implementación de estas funciones se encuentra en los archivos fs/fs.c y fs/fs.h. Veamos como se hicieron algunas de estas funciones.

En el caso de read() y write(), comenzaron siendo funciones distintas, pero rápidamente me di cuenta que el código entre ellas era muy similar, por lo que decidí separarlo en la función fs_readwrite(). Esta se encarga de leer o escribir un archivo, dependiendo de la flag que se le provee. Hace algunos checkeos importantes (como por ejemplo fijarse si el archivo no es un dispositivo de caracteres), y luego llama a copy_file() que realiza el trabajo duro:

```
while (n > 0) {
    if ( (blocknr = read_map(ino, pos)) == NO_BLOCK)
        return ERROR;
    block = (char *) get_block(blocknr);

    off = pos % BLOCK_SIZE;
    size = MIN(n, BLOCK_SIZE - off);

    if (flag == FS_WRITE) mymemcpy(block + off, buf, size);
    else                  mymemcpy(buf, block + off, size);

    n -= size;
    pos += size;
    buf += size;
}
```

El loop principal de copy_file() va navegando los bloques del archivo con read_map() y, dependiendo si se busca leer o escribir, utiliza la función mymemcpy para realizar el trabajo (que es un clon de la función de mismo nombre de la biblioteca estándar).

El resto de las llamadas de archivo son simples, open() y close() solo manejan fds con las funciones antes descriptas; unlink() llama a find_inode() pidiendo que se borre el archivo; lseek() modifica el puntero al archivo en la lista de fds.

En un principio las funciones search_inode() y find_inode() eran muy simples, uno le daba un path y devolvian el inodo correspondiente. Cuando comenzé a implementar las llamadas de sistema de directorios me di cuenta que este método no iba a funcionar, y tenía dos opciones: o bien crear nuevas funciones de manejo de inodos para directorios, o cambiar estas 2 y reordenar el código. El problema de la primer opción es que mucho código se duplica, lo que ensucia la implementación y por lo tanto no lo consideré. Luego tomé el segundo camino, cambie la signatura de las funciones para que search_inode() devuelva la entrada misma en el directorio.

Utilizando esta modificación, rmdir() remueve un directorio asegurandose primero que este vacío. Esto requiere de conseguir el inodo del padre y del hijo, checker que este vacío y limpiar la entrada en el padre.

La llamada mkdir() no solo requiere crear el nuevo inodo de directorio, sino que también hay que colocar los links a '.' y '..'; también rename() hace uso de este cambio, toma las entradas del directorio viejo y del nuevo, y copia los contenidos de una a la otra, además debe fijarse si se trata de un directorio, para actualizar la referencia '..'; chdir() que cambia el directorio actual (el inodo del directorio actual se guarda en la tabla del proceso).

Por último esta getdents(), que se utiliza para recorrer un directorio. Esta llamada es media extraña (por esta razón se utiliza una mucho más amistosa en POSIX, readdir(), que a su vez llama a getdents()), dado un buffer y un tamaño, copia en este buffer la mayor cantidad de estructuras de entrada de directorio posibles. El problema es que estas estructuras no tienen un tamaño fijo, contienen el nombre del archivo que es variable. Por si esto no fuera poco, esta estructura evolucionó con el tiempo en linux, y se le agregó al final de todo, despues del nombre, un campo d_type que dice si la entrada es un archivo regular/directorio/dispositivo etc. Para recobrar esta información es necesario calcular el tamaño de la estructura, sumarlo al inicio de la misma

y castear el valor de d_type. Luego recorrer el buffer para el usuario puede ser un poco trabajoso (ver la implementación del programa ls).

6 Cash y otras utilidades

Hasta ahora todo el desarrollo fue concentrado en el núcleo del sistema. Si bien esta es la pieza más importante, la única manera de poder verlo en acción es correr aplicaciones en el sistema.

Tome todas las medidas necesarias para que este kernel sea compatible con C, con la interfaz POSIX y GCC. Más aún mi objetivo desde un principio fue que las aplicaciones programadas para este núcleo funcionen sin cambios en linux. Salvando las distancias, y reduciendose a utilizar únicamente el set pequeño de funciones que nos da el sistema, proveo una serie aplicaciones que cualquier usuario de un sistema GNU/Linux va a conocer de inmediato (todas se encuentran en el directorio apps):

- cash: un "clon" de sh, una consola que recibe comandos y los ejecuta.
- cp: copia un archivo
- echo: imprime una linea en stdout
- ls: lista las entradas de un directorio
- rm: remueve un archivo
- mv: mueve un archivo
- mkdir: crea un directorio
- rmdir: remueve un directorio
- cat: imprime en stdout los contenidos de un archivo
- argc: imprime en stdout la cantidad de argumentos que recibió
- lineecho: recibe lineas en stdin y las reproduce en stdout

Estas aplicaciones demuestran todo el funcionamiento del núcleo, y dejan al usuario en un sistema que simula las consolas modernas de hoy en día. Para poder implementar estos programas fue necesario crear un archivo aparte, apps/scall.asm, que es el puente entre las llamadas al sistema en C, que llaman a una función con los parametros en la pila, y la verdadera llamada al sistema, que pone los parametros en los registros de la CPU y genera la interrupción 0x80.

Luego el archivo apps/scall.asm es simplemente una colección de etiquetas de todas las llamadas soportadas, que elige para cada una su respectivo número de system call y lo coloca en eax. Luego se completan ebx, ecx y edx y se lanza la interrupción.

En general todas las aplicaciones son bastante simples, como no les agregé soporte a flags ni parametros raros por consola, solo realizan un trabajo concreto (por ejemplo cp copia _un_ archivo de un lugar a otro). Entre todas las aplicaciones, las más interesante es cash, en apps/cash.c.

Cash intenta de emular una consola moderna de un sistema tipo *nix. Presenta información como el usuario y la carpeta actuales, y luego se queda esperando a que se ingrese un comando. Luego pueden pasar una de dos cosas, este comando puede ser manejado directamente por cash, o cash crea un nuevo proceso con la aplicación deseada. La primera opción por ejemplo ocurre cuando se ejecuta el comando "cd". Esto implica realizar la llamada al sistema chdir() para cambiar el directorio actual, pero también requiere que se actualize el texto del directorio actual que se le presenta al usuario (esto se encuentra en la función updatepwd()). Todo este manejo de comandos es realizado en la función execute().

La segunda opción es más simple, solo requiere que se ejecute la llamada newprocess() con el programa y sus argumentos, y luego se bloquee esperando la terminación del proceso hijo con waitpid().

Cash luego entra en un ciclo infinito en donde espera comandos del usuario, los atiende y vuelve nuevamente al principio.

Como fue anticipado, todas las aplicaciones compilan sin modificaciones en linux, ejecutando por ejemplo "gcc cp.c" para obtener cp. Hay 2 excepciones, cash y ls. El problema con ls es que la llamada getdents() de linux no tiene un wrapper en glibc (proveen readdir()), luego no puede ser llamada directamente desde C de esta manera. La forma de solucionar esto es utilizar la función syscall() con el número de getdents(), llamando indirectamente al kernel.

El problema con cash es de diseño, y ya fue explicado un poco anterioremente. Para simplificar la implementación del kernel decidi reemplazar las llamadas fork() y execv() por una unión de ambas, newprocess(). Por supuesto esto no existe en linux ni en ningún sistema *nix, luego para que cash pueda ser compilado por gcc para linux es necesario reemplazar esta llamada por un fork() y seguido por un execv() inmediatamente, logrando el mismo resultado.

7 Conclusión y últimas palabras

Es muy satisfactorio ver por primera vez la consola funcionar, enviarle mensajes y recibir las respuestas esperadas. Más aún cuando se conocen todas las complicaciones y ramificaciones que se disparan en cada comando.

La primer conclusión que me gustaría compartir es más bien una crítica. Habiendo trabajado extensivamente con la arquitectura x86, es muy difícil que uno quede conforme con la misma. Creo que esta necesidad que se ha impuesto de mantener la compatibilidad hacia atrás ha perjudicado mucho el diseño de los sitemas modernos, y x86 es un ejemplo de esto. Cosas como el modo real/protegido no tienen más sentido y simplemente agregan complejidad y dificultades al programador de sistema. Lo mismo puede decirse de la segmentación, un modelo que no es utilizado por ninguno de los SO en la actualidad, pero aún hay que cargarlo y mantenerlo constantemente. Es una lástima que proyectos como Itanium que justamente intentaron curar estos problemas no hayan sido exitosos.

A la hora de tener que programar en tan bajo nivel, es claro que es necesario tener mucha más atención al detalle. Tener un bit en 1 ó en 0 puede significar que el sistema bootee perfectamente o que no haga nada. Esto no es un problema tan grande en aplicaciones de usuario ya que estas pueden ser debuggeadas y estos errores corregidos con cierta eficacia. Sin embargo es muy difícil diagnosticar problemas en el sistema operativo, no contamos con tantas herramientas como para darnos cuenta cual es el inconveniente y muchas veces el proceso se reduce a colocar mensajes en lugares claves, y pensar mucho. El procesador a veces puede no ser nada descriptivo sobre el error, o podemos estar en una etapa del booteo donde no podemos ni escribir información en la pantalla para intentar entender la situación.

En un principio, cuando todavía no pretendía bootear el kernel y solo me preocupaba por que el código compile y sea limpio, el desarrollo fue muy ágil. Esto me engaño en pensar que el proyecto no iba a ser demasiado extenso. Sin embargo, como en todo sistema de software, a la hora de pegar todos los distintos componentes y ver que nada funcionaba al l00%, fue obvio que me esperaba la parte más dificil. Le dio un nuevo significado a la frase de Tom Cargill: "The first 90 percent of the code accounts for the first 90 percent of the development time... The remaining 10 percent of the code accounts for the other 90 percent of the development time." Creo que esto es más que cierto para programadores de sistemas en bajo nivel.

Los manuales de intel fueron indispensables en todo el proceso. Si bien se les puede criticar seriamente muchas de las desiciones sobre la arquitectura o manejo de sus procesadores, la documentación de los mismos es excepcional, estan muy bien organizados y logran responder a casi todas las dudas que pueden surgir. Esto es muy meritorio considerando que la docuementación suele no ser el punto fuerte de la mayoría de los proyectos de software.

Como último punto, quiero remarcar a algunas de las fuentes en las que me apoyé para realizar todo este desarrollo. La página osdev.org que contiene lindas wikis y un foro muy activo con gente dispuesta a responder cualquier pregunta. A los libros "Operating Systems: Desing and Implementation" de Tanenbaum y "Operating Systems Concepts" de Silberschatz, Galvin y Gagne, que utilicé para aprender y construir este mini SO. El código de linux, en donde conseguí autoresponderme preguntas sobre las estructuras en su implementación del fs Minix. Pero por sobre todo al SO Minix, que me sirvió como referencia a la hora de tomar decisiones del núcleo y fs, y aunque por motivos de simplicidad esta kernel terminó siendo monolítica, lo tuve siempre como objetivo a alcanzar (en especial en la implementacion del MinixFS).