DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Marzo de 2010 — Organización del Computador II

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	marianobianchi08@gmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Di Pietro, Carlos Augusto Lyon	126/08	cdipietro@dc.uba.ar

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

El presente trabajo final surge como una continuación del tercer trabájo práctico de la materia en el segundo cuatrimestre de 2009. Aquél trabajo consistía en implementar un sistema minimal que permitiese correr concurrentemente dos tareas. Concretamente, dicho sistema consitía en un bootloader que se encargaba de cargar a memoria un kernel simplificado que incluía los binarios de las tareas en cuestión, junto con todas las estructuras necesarias para que dichas tareas pudieran ser ejecutadas (GDT, Directorio de Tablas de Páginas, una Tabla de Páginas para cada tarea, etc.). Luego, el kernel simplemente debía encargarse de activar el *Gate A20*, pasar el procesador a modo protegido, activar el sistema de paginación, y poner a correr las tareas llamadas "Pintor" y "Traductor", las cuales alternaba mediante una interrupción del timer.

Por el contrario, el trabajo aquí presentado, si bien se basa en el anterior, posee algunas diferencias. A pesar que las tareas de usuario se encuentran cargadas en memoria de manera estática, el principal aspecto que diferencia al presente trabajo, es que estas se van inicializando a medida que el usuario así lo desee, a través del manejo de una consola diseñada para dicho fin. A la hora de inicializar una tarea, esta es puesta en ejecución a través de un scheduler que va asignando tiempos de CPU a cada uno de los procesos que se ejecutan de forma concurrente en el sistema. Naturalmente, esta diferencia en cuanto al otro kernel conlleva un cambio en lo que respecta a la administracíón de memoria, ya que estructuras como entradas de la GDT o TSS's deberán ser creados e inicializados para cada nueva tarea conforme estas van siendo lanzadas.
En consecuencia, el resultado final es un *kernel multitarea* (es decir capaz de ejecutar varias tareas alternadamente dando la ilusión de simultaneidad) que puede lanzar procesos de forma dinámica con tan solo cargar tareas de memoria y creando las estructuras necesarias para que estas puedan ejecutarse en un procesador de arquitectura Intel-x86.

2. Instrucciones de uso

Para ejecutar el kernel basta con compilar el kernel utilizando el MakeFile. Luego utilizando Bochs 2.4 se iniciara el proceso. Una vez cargado, el kernel mostrará en pantalla una consola mediante la cual se podrá cargar y poner a ejecutar cada una de las tareas. A continuación se detallan la lista de comandos que pueden ser interpretados por la consola:

- "h": ayuda (lista los comandos).
- "l": muestra todas las tareas disponibles.
- "p": muestra todas las tareas en ejecución junto con su pid.
- "v {x}": ejecuta y muestra la tarea {x}.
- "d {pid}": muestra la tarea con pid {pid}.
- "k {pid}": termina la tarea con pid {pid}.

3. Implementación

3.1. Descripción General

El código fuente que implementa el kernel se entrega junto con este informe en un sorpote digital y se ubica en la carpeta `codigo`. Dentro de la misma los archivos se organizan de la siguiente manera:

- Memoria
- GDT - Global Descriptor Table
- Consola
- TSS - Task State
- Interrupciones
- Scheduler
- Paginación
- Pantalla
- BCP - Block Control Process
- Kernel

Esta distribución no es arbitraria, sino que responde a la modularización con la cual se encaró el diseño y desarrollo del kernel aquí presentado. Así, cada directorio contiene el código fuente de uno o más de los módulos que integran al kernel, cada uno de los cuales fue desarrollado de forma incremental y testeado individualmente.

En la sección 3.3, se procederá a explicar en detalle cada uno de estos módulos a fin de poder comprender con claridad todas las partes que componen al kernel elaborado. Una vez concluída esa explicación, en la sección siguiente (3.4), se detallará de qué forma el kernel agrupa y hace uso de todos estos módulos a fin de dar como resultado un sistema *multitasking* con un *scheduler* dinámico capaz de levantar tareas de memoria y alternarlas por medio de una política de reemplazo *Round Robin*.

Seguidamente, se expone cómo está constituído el *Mapa de memoria* del sistema, así como también el porqué de su elección.

3.2. Mapa de Memoria

Previo a la escritura del código de los módulos mencionados en la sección anterior, a sabiendas de que el sistema utilizaría la paginación como forma de direccionar a memoria, se procedió a establecer un mapa de memoria que permitiese ubicar las estructuras críticas e indispensables del sistema de manera inequívoca. La opción elegida fue realizar un *identity mapping* de las páginas (todas ellas de 4 KB) a los frames de memoria, fijando algunos de ellos para uso exclusivo del kernel o de otras estructuras como la *GDT*, las *TSS* o el *Bitmap*.

La imagen que sigue muestrá como estan mapeadas y ocupadas las páginas de memoria en el sistema:

1	0x00000000	Boot Sector
2	0x00001000 0x000011FF	
	0x00001200 0x00001FFF	Kernel
3 a 12	0x00002000	Código y datos de las tareas
	0x0000CFFF	
…	253 páginas	Kernel
256	0x000FF000 0x000FFFFF	
257	0x00100000	Directorio de Tablas de Páginas, Tablas de Páginas y Bitmap
…	8 páginas	
265	0x00109000 0x001093FF	
	0x00109400 0x00109FFF	Pila del Kernel
266	0x0010A000	
…	245 páginas	
511	0x001FF000 0x001FFFF	
512	0x00200000	Memoria libre
…	7678 páginas	
1048575	0x01FFF000 0x01FFFFFF	

3.3. Módulos

3.3.1. Memoria

El módulo de **Memoria** es el encargado de brindarle al kernel las herramientas para contabilizar y administrar la memoria del sistema.

Básicamente, la idea empleada para administrar consiste en contabilizar la cantidad de bytes con los que cuenta la memoria, luego determinar cuántas páginas de 4 KB hay en la memoria del sistema y luego plasmar esta información en un *Bitmap*. El *Bitmap* no es más que una porción de memoria en donde hay tantos bits como páginas haya en el sistema. Luego, cada página ocupada se representa en el *Bitmap* poniendo un "1" en el bit asociado a dicha página, mientras que las páginas libres se representan con un "0".

En líneas generales la interfaz de este módulo puede ser caracterizada de la siguiente manera:

- Variables Globales:
  - `memoria_total:` Variable global en la cual se almacena la cantidad de Megabytes de memoria con la que cuenta el sistema.
  - `paginas_libres:` Variable global que contabiliza el número de páginas libres de memoria en el sistema.
  - `dir_init_bitmap:` Puntero a la dirección 0x???, que es la posición donde se inicia el *Bitmap* para administrar las páginas de memoria.
  - `dir_end_bitmap:` Puntero a la dirección 0x??, que es la última dirección válida del *Bitmap*.
- Funciones:
  - `contarMemoria:` Función que se encarga de contar cuántos MB de memoria hay en el sistema guardando dicho valor en la variable `memoria_total`. Asímismo, determina el número de frames de 4 KB que puede haber en la memoria del sistema y almacena dicho valor en la variable `paginas_libres`.
  - `llenarBitmap:` Función que a partir de la posición apuntada por `dir_init_bitmap` marca poniendo en "1" todos los bits correspondientes a las páginas de memoria utilizadas por el kernel y luego pone en "0" a las restantes.
  - `pidoPagina:` Función que devuelve un puntero a la primer posición de memoria de una pagina libre, y previo a ello, la marca como ocupada en el Bitmap poniendo en "1" el bit correspondiente a dicha página.
  - `liberoPagina:` Función que dado un puntero a una posicion de memoria, determina en qué página de memoria se alberga dicha posición y luego pone en "0" el bit correspondiente a esa página para así marcarla como libre.
  - `setmem:` Función que dado un puntero a una posición de memoria y dos valores enteros *set* y *cant*, pone el valor de *set* en *cant* bytes desde la posición de memoria apuntada por el puntero en adelante.

- `cpmem:` Función que dados dos punteros a posiciones de memoria y un valor entero *cant*, copia el valor de *cant* bytes desde la posición de memoria apuntada por el primer puntero en adelante, hacia los *cant* bytes de memoria desde la posición de memoria apuntada por el por el segundo puntero en adelante.

3.3.2. Global Descriptor Table (GDT)

El módulo de **GDT** es aquel mediante el cual se introduce la estrutura de datos *GDT* que modela la estructura *Global Descriptor Table* de la arquitectura IA-32 de Intel[1]. Asímismo, implementa también las funcionalidades necesarias para que el kernel pueda operar y hacer uso de esta estructura.

Concretamente, la implementación de la estructura *GDT* no es más que un arreglo de otra estructura más pequeña definida en este mismo módulo: la *GDT_entry*. Dicha estructura constituye un bloque de memoria de 4 bytes que modela las entradas en la *GDT*, las cuales reciben el nombre de descriptores. Los descriptores pueden ser de varios tipos (descriptor de segmento, descriptor de TSS, etc) y se separan en diversos campos. Por tal motivo, la estructura *GDT_entry* también está subdividida en idénticos campos los cuales, segun los valores que se le fijen, indican entre otras cosas a qué tipo de descriptor corresponde cada entrada.

Finalmente, nuestra GDT queda organizada de la siguiente manera:

- entrada nula
- codigo kernel
- datos kernel
- codigo usuario
- datos usuario
- descriptor_tss1
- descriptor_tss2
- ...
- descriptor_tssx

En el módulo implementado podemos encontrar las siguientes funciones:

- `make_descriptor`: función que crea una entrada para la gdt tomando como parametros la base, el limite y los atributos.
- `buscar_entradaGDT_vacia`: Devuelve una posicion de la gdt que este vacia.
- `borrar_gdt_entry`: Crea un descriptor vacio en la posición a borrar.

[1] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1 - Sección 2.1.1, página 61

3.3.3. Paginación

En este módulo, podemos encontrar todas las funcionalidades necesarias para el manejo de la paginación, las cuales son:

- mapear_tabla
- mapear_pagina
- obtener_mapeo
- iniciar_paginacion_kernel
- liberar_directorio

Cuando se hizo necesario implementar el módulo, nos dimos a la tarea de imaginar qué necesidades íbamos a tener respecto a la paginación. En un comienzo, las rutinas *mapear_tabla* y *mapear_página* eran precarias. Necesitaban recibir como parámetro la dirección de la entrada de directorio y la de tabla de páginas respectivamente. Esto resultaba molesto, ya que primero habia que hacer un cálculo para conocer el offset dentro de cada tabla antes de poder realizar el mapeo. Además, en el caso del mapeo de una página, también habia que verificar que la tabla de páginas asociada a la página que se intentaba mapear estuviera presente.

Teniendo en cuenta estas desventajas, se concluyó que era necesario que todos estos cálculos y verificaciones se hicieran dentro de las mismas funciones. Seguidamente, se modificó el código de *mapear_página* para que sólo reciba la dirección fisica del directorio de páginas, además de las necesarias como dirección virtual (o lógica), dirección física a la cual se debía mapear la dirección virtual y los atributos de dicha página (escritura/lectura, supervisor/usuario, presente/no presente). Una de las principales características de la función es que allí se hace la verificación de si está o no presente la tabla de páginas que corresponde, y en caso de no estarlo, se pide una página para su estructura, se la inicia usando *mapear_tabla* y luego se continúa con el mapeo deseado. Estas características hicieron mucho más simple su uso y nos facilitó la tarea a la hora de programar otras rutinas que dependían de esta implementación.

Por otra parte, está la función *iniciar_paginacion_kernel*, que simplemente inicializa el Directorio de Tablas de Páginas (DTP) y todas las Tablas de Páginas (TP) asociadas. Para ello, se reservó una dirección dentro del mapa de memoria (la dirección es 0x100000), en la cuál se alojan todas estas estructuras.

La última función crucial de este módulo es *liberar_directorio*. Esta función se encarga de, dada la dirección de un directorio de tablas de páginas, borrar todas las entradas del directorio y de las tablas de páginas y limpiar y liberar todas aquellas páginas que hayan sido utilizadas para contener estas estructuras, las páginas de las pilas de la tarea y del buffer de video. Esta función es utilizada a la hora de eliminar una tarea de memoria.

3.3.4. TSS

La TSS (Task State Segment) es una estructura de datos que contiene información acerca de una tarea.

Para setear la TSS se necesita una entrada en la GDT especificando la base, limite y atributos.

Luego, para acceder al código de la tarea, deberemos, entre otras cosas, hacer un jump al descriptor de TSS en la gdt.

En nuestro kernel, las funciones que poseemos en este módulo son:

- `crear_TSS`: Crea una TSS (y su descriptor en la GDT) utilizando como parametros: dirección, CR3, EIP, EFLAGS, pila y el ESP0.
- `buscar_TSS_vacia`: Busca una entrada de TSS vacia (toma como TSS vacia aquella cuyo cr3 sea igual a 0, devuelve un número igual o mayor que CANT_TAREAS si no hay ningun lugar disponible)
- `vaciar_TSS` Setea el cr3 de una tss en 0 para ser reutilizada por otras tareas.

3.3.5. Process Control Block (BCP)

El Process Control Block, es una estructructura de datos en el Kernel que contiene la información necesaria para manejar cada uno de los procesos. En nuestro caso, esta estructura contiene la siguiente información:

- **pid**: indice de la tarea en el gdt_vector
- **estado**: indica el estado de la tarea
- **entrada_directorio**: direccion del directorio de la tarea
- **sig**: siguiente tarea para el round robin scheduler
- **ant**: anterior tarea para el round robin scheduler
- **pantalla**: puntero a la pagina destinada al video de la tarea
- **nombre**
- **dir_fisica**: puntero a la/s página/s a donde fue copiada la tarea para ejecutarse

Ya que el BCP contiene informacion critica de los procesos, esta almacenada en un area protegida de los usuarios. En nuestro caso la protección que tiene es la misma que la de todo el kernel. El usuario tiene permisos de lectura, pero no de escritura.

Las principales funciones del módulo BCP son las siguientes:

- `iniciar_BCP`: llena el BPC[0] con los datos del kernel, y inicializa variables globales
- `buscar_entradaBCP_vacia` : busca entrada libre en el BCP (libre significa estado muerto)

- `crear_entradaBCP` : llena la entrada con los datos de la tarea y la agrega al final de la cola de tareas activas
- `cambiar_estado`: cambia el estado de una tarea, y si el estado es MUERTO la quita de la cola de tareas activas
- `buscar_entradaBCP`: devuelve la posicion en la BCP de una tarea pasada como parametro.
- `buscar_entradaBCP_matar`: devuelve la posicion en la BCP de alguna tarea con estado "MUERTA". Si no hay ninguna, devuelve CANT_TAREAS
- `cargarTarea`: carga una tarea y todo sus datos y contexto en memoria y la agrega en la BCP para incluirla en el scheduling
- `matarTarea`: Marcar tarea como "MATAR"para que luego el KERNEL se encargue de eliminarla.
- `exit`: esta es llamada cuando la tarea actual quiere terminar y llama a la interrupcion 80
- `desaparecerTarea`: Esta funcion se va a llamar cada vez que se ejecute el kernel. La idea es que si hay alguna tarea en la BCP marcada como "MATAR"(ya va a estar fuera del scheduler), esta funcion se encargue de eliminar y liberar todas las estructuras utilizadas por la tarea (BCP, TSS, directorio y tablas de páginas, paginas de video y de pila y gdt).

3.3.6. Interrupciones

Las estructuras y funciones necesarias para el manejo de interrupciones en el kernel aquí presentado se encuentran implmentadas en dos módulos:

- El primero se encarga de la creación y el manejo de las estructuras necesarias para el funcionamiento de las interrupciones: *IDT, descritores de IDT, registro de IDT, etc.*
- El segundo se encarga de definir cada una de las funciones que constituyen las rutinas de atencion de interrupciones.

El primer módulo está compuesto por los archivos *idt.c* e *idt.h*. Allí se especifican las estructuras *IDT* e *IDT_Descriptor* que modelan respectivamente la *Interruption Descriptor Table (IDT)* y el *registro de GDT (GDTR)* de la arquitectura IA-32 de Intel[2].
Al igual que la estructura *GDT* descripta en la sección 3.3.2, la *IDT* es un arreglo de otra estructura menor que es la *IDT_entry*, la cual a su vez modela las entradas de la *IDT* que reciben el nombre de *descriptores de IDT*[3].

[2]Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1 - Sección 6.10, página 226

[3]Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1 - Sección 6.11, página 228

A continuación presentamos la interfaz de este módulo explicando cada una de las funciones que brinda al kernel para el manejo de las estructuras antes explicadas:

- `set_interrupt:` Setea una *IDT_entry* con los valores necesarios para que sea un descriptor de interrupción.
- `set_trap:` Setea una *IDT_entry* con los valores necesarios para que sea una trap gate.
- `set_task_gate:` Setea una *IDT_entry* con los valores necesarios para que sea una puerta de tarea.
- `idtFill:` Setea las primeras 20 entradas de la *IDT* como descriptores de interrupciones apuntando cada uno a su correpondiente rutina de atención, cada una de las cuales llama a su vez a la función *handler_de_interrupciones*. Luego, setea la entrada 32 y 33 de la *IDT* como descriptores de interrupciones para el timertick y teclado respectivamente. Finalmente, setea la entrada 80 de la *IDT* como un descriptor de interrupción que implementa la syscall *exit* para matar al proceso que se encuentra en ejecución.
- `void handler_de_interrupciones:` Esta función es llamada por las primeras 20 rutinas de atención de interrupciones para imprimir por pantalla el número de interrupción producido y colgar la ejecución del sistema.

El segundo módulo...

3.3.7. Scheduler

El algoritmo de scheduling es el que determina cuanto tiempo de CPU sera dedicado a cada proceso dependiendo cumpliendo con ciertos criterios

- todas las tareas deben poseer tiempo en la CPU.
- si se usan priordades, una tarea con menor prioridad no debería trabar a una de mayor prioridad.
- el scheduller debe escalar bien a medida que se agregan tareas en el kernel.

Para este trabajo elegimos el algoritmo de Round Robin.

Este algorimo es simple ya que se utiliza una única cola de procesos. Cuando el tiempo de cada uno de estos pasa, se efectua un cambio hacia la ejecución del siguiente y el anterior es puesto nuevamente en la cola.

Cada proceso posee un quantum (cierto numero de ticks del reloj) que es el tiempo en el que se estara ejecutando. En nuestro caso, cada quantum equivale a 1 tick de reloj.

En nuestro kernel, el scheduller esta compuesto por la función switch_task, cuyo funcionamiento es el siguiente

1. comprueba que haya mas de una tarea, sino no hace nada.
2. cambia el estado de la tarea actual de CORRIENDO a ACTIVO (o lo deja en MATAR si es que estaba asi)
3. cambia el selector de 'salto' a donde se marco el 'pid' de la tarea_actual, teniendo en cuenta qué RPL (Requested Privileged Level) elegir dependiendo de qué tarea se va a pasar a ejecutar
4. se hace el cambio de tarea

3.3.8. Consola

La consola es una función del kernel que permite una interfaz hacia los usuarios, en este caso es una linea de comandos que permite listar, correr, matar, etc. una lista de tareas disponibles en el sistema.

A continuación se listan las principales funciones necesarias para hacer posibles la linea de comandos:

- **cargar_tarea**: recibe la posicion de la tarea a cargar dentro de "tareas_en_memoria"
- **console**: Es la función llamada cada vez que se preciona una tecla en la linea de comandos. En caso que esta tecla sea un 'enter' se llama a la función 'run' del comando almacenado, en caso contrario se van almacenando las teclas.
- **run**: Es la encargada de obtener y decodificar el comando enviado por console de manera de llamar a la función correcta.

3.3.9. Pantalla

El módulo pantalla se ocupa de lo necesario para poder imprimir información en la pantalla para el uso del kernel y las tareas.

Está planeado de la siguiente manera:

- 23 filas centrales que podras usar las tareas y el kernel para escribir
- En la última fila se vera la linea de comandos (manejada por console.c)

Las funciones que provee son:

- **avanzar_puntero**: Avanza el puntero de la pantalla un caracter.
- **retroceder_puntero**: Retrocede el puntero de la pantalla un caracter.
- **mover_puntero**: Mueve el puntero de la pantalla hasta una fila y columna indicadas.

- **salto_de_linea**: Avanza el puntero de la pantalla una fila.
- **printf**: Es utilizadad para imprimir caracteres.
- **printdword**: Se usa para ver valores de variables. Recibe como parámetros la dword a imprimir y los atributos de escritura.
- **putc**: Escribe un caracter en la pantalla
- **borrarc**: Borra el ultimo caracter de la pantalla
- **num2char**: Deja en buffer un string que representa al numero pasado escrito en una base indicada.
- **clear_screen**: Limpia la pantalla.
- **mostrar_pantalla_entera**: La idea es copiar 23 renglones enteros de la tarea 'pid'. (Recordar que la pantalla es de 80*23, de manera de dejarle el ultimo 'renglon' para el kernel y la consola y el primer renglón para saber qué estamos visualizando.
- **cambiar_pantalla**: Setea la pantalla a mostrar y llama a mostrar pantalla entera.
- **clear_command_line**: limpia la linea de comandos (fila 24 de la pantalla) y deja el prompt y el puntero a dicha linea inicializado
- **clear_info_line**: Limpia la linea de informacion (fila 0 de la pantalla) y deja el puntero a dicha linea inicializado
- **removerc**: Borra el ultimo caracter de la linea de comandos
- **agregarc**: Escribe un caracter en la linea de comandos

3.4. Ensamblando el Kernel

- Habilitar Gate A20
- Inicializar la GDT y el GDT_desc
- Copiar GDT_desc a lgdt
- Habilitar bit PE de CR0
- Jmp 0x08:modo_protegido
- Pasaje a modo protegido
- Actualizar selectores
- Inicializar la pila
- Contar memoria disponible
- Crear estructuras de paginación
- Cargar en CR3 la direccion del Directorio de Tablas de Páginas

- Habilitar bit PG de CR0
- Inicializar Bitmap
- Pasaje a paginación