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.
- `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 es aque en el que se implementan las funciones *_INTX*, donde `X` el número de la interrupcion que al producirse invoca a su correspondiente función *_INTX*.

Seguidamente se expone la lista de rutinas de interrupción implementadas junto con una breve descripión de su funcionamiento:

- `_INT0 a _INT19:` Llama a la función `handler_de_interrupciones` pasándole como parámetro el número de la interrupción producida para que ésta imprima por pantalla un mensaje asociado a dicha interrupción y luego ejecute un loop infinito a fin de colgar la ejecución del sistema.
- `_INT32:` Llama a la función `switch_task` del módulo **Scheduler** para que se produzca el cambio de la tarea que se esta ejecutando por la tarea elegida por el *scheduler* segun su politica de reemplazo.
- `_INT33:` Copia el valor de la tecla pulsada en el registro *AX* y llama a la función `console` (implementada en el módulo **Consola**) pasandole el valor de ese registro como parámetro.
- `_INT80:` Llama a la función `exit` del módulo **BCP** que se encarga de imprimir en pantalla el mensaje *"TERMINE!!"*, matar a la tarea que se está ejecutando y llamar a la función `switch_task` para pasar a la proxima tarea segun la política de scheduling.

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. EL Kernel

Hasta aquí se explico el funcionamiento de todos los módulos que componen el kernel. En esta sección, se explicará cómo se integran todos estos componentes para conformar el kernel propiamente dicho. Veamos entonces cómo se conforma el módulo **Kernel**.

Los archivos del módulo **Kernel** son tres: *kernel.c*, *kernel.h* y *kernel.asm*. Los dos primeros, son archivos que contienen la declaración e implementación de funciones que serán utilizadas por el kernel durante su ejecución. Estas son:

- `kernel_infinito:` Esta función consistuye el loop infinito sobre el cual se basa todo kernel. Durante este ciclo, la única acción del kernel consiste en la búsqueda de algún proceso al que se le haya marcado en su entrada de *BCP* como listo para matar y se procede a la liberación de todos los recursos asociados a ese procesos (páginas de memoria, TSS, pila, entrada en la *BCP*, entrada en la *GDT*).
- `update_cursor:` Esta función únicamente se encarga de mover el cursor que está en la pantalla.

Finalmente, el archivo *kernel.asm* contiene el código de inicialización del kernel, que es en donde finalmente se hace uso de todas las estructuras y funcionalidades aportadas por los módulos vistos anteriormente. A continuación, se detallará todos los pasos que realiza el kernel a fin de preparar el sistema para que sea capaz de ejecutar tareas de modo concurrente, para luego dar paso a la ejecución de las mismas por medio de la invocación a la función `kernel_infinito`:

1. Como primer paso, se habilita el Gate A20 para que se pueda direcciónar a memoria más alla de la dirección 0x100000; es decir, más allá de 1 Megabyte.
2. Luego, se carga la estructura *GDT_desc* del módulo **GDT** en el registro *gdtr* del procesador, el cual describe la direción base y el límite de la *GDT*. A su vez, la *GDT*, que tal como se explicó oportunamente en la sección 3.3.2 es un arreglo de descriptores, también es inicializada seteando la memoria contigua al código del kernel (hacia el final del archivo *kernel.asm*) de forma tal que allí queden alojados los 128 descriptores detallados en la sección 3.3.2.
3. Seguidamente se setea el bit *PE* de registro *CR0* del procesador a fin de poder realizar el pasaje a *modo protegido*.
4. Luego se ejecuta la instrucción `Jmp 0x08:modo_protegido` a fin de efectivizar le pasaje al *modo protegido* del procesador. Segudiamente, se actualiza el valor de los selectores para que apunten al descriptor de segemento de datos en la *GDT*.
5. A continuación, se inicializar la pila en la posición de memoria 0x200000 (2 MB). Se asume que el sistema cuenta con esa mínima cantidad de memoria, ya que es requisito por el tamaño del kernel.
6. Una vez hecho esto se realiza un llamado a la función `contarMemoria` del módulo **Memoria** a fin de poder inicializar las variables `memoria_tota, paginas_libres` con valores acordes a la memoria del sistema sobre el cual se está ejecutando el kernel.

7. Luego, se hace una invocación a la función `iniciar_paginacion_kernel` del módulo **Paginación** (sección 3.3.3) para setear e inicializar las estructuras necesarias (directorios, tablas de páginas, páginas)a fin de que el sistema pueda trabajar con este método de direccionamiento a memoria.

8. Inmediatamente después se procede a cargar el registro *CR3* del procesador con la direccion del Directorio de Tablas de Páginas a fin de que el procesador pueda ubicar esta estructura en memoria.

9. Seguidamente, se habilita la unidad de paginación en el procesador. Para ello, se setea el bit *PG* del registro *CR0*.

10. Luego, una vez habilitada la paginación, se invoca a la función `llenarBitmap` del módulo **Memoria** para marcar como ocupadas a todas las páginas utilizadas por el kernel y como libres a las restantes.

11.

4. Cómo funciona todo

4.1. Introducción

En las secciones anteriores de este informe se realizó una explicación a grandes rasgos de todas las estructuras y métodos utilizados para el funcionamiento del kernel y de las tareas. No obstante, para lograr entender mejor cómo es que fue implementado todo y cómo es que funcionan las cosas, creemos que sería de mucha ayuda hacer una descripción paso a paso de algún posible escenario. En esta sección entonces, trataremos de clarificar el funcionamiento global del trabajo práctico.

4.2. Escenario

El escenario a describir es el siguiente:

- Se inicia el kernel
- Se carga la tarea 1
- Se elimina la tarea 1

Se inicia el kernel

Se carga la tarea 1

Se elimina la tarea 1

5. Nosotros también nos equivocamos

A lo largo de la realización del trabajo práctico fuimos encontrándonos con infinidad de problemas. Muchos fueron fáciles de descubrir y resolver, pero con otros no se contó con la misma suerte. Muchas veces estuvimos trabados mucho tiempo hasta poder continuar programando ya que había errores que no nos dejaban seguir adelante. Por esta razón es que creemos que en este informe no podía faltar una sección comentando alguno de los errores que nos causaron mayores dolores de cabeza.

Lo primero que se hizo a la hora de comenzar este trabajo práctico, fue limpiar el trabajo hecho en clase y sólo tomar las cosas básicas y primordiales, como la activación del GATE A20, la estructura de la GDT para el kernel, el paso a modo protegido y la actualización de los registros de segmento para que utilizaran los segmentos de la GDT correspondientes. Luego, comenzamos a trabajar en el resto de las funcionalidades, como la paginación, el manejo de memoria, etc. La idea era utilizar la menor cantidad de código assembler posible, para que el código resulte más legible y amigable, por lo tanto todas estas rutinas fueron implementadas en lenguaje C. Gracias a esta decisión, nos topamos con el primer problema. Intentando probar un código de C, durante el llamado a la función ocurría una excepción y la maquina virtual Bochs se colgaba. Intentando buscar donde estaba el error, fuimos comentando lineas de código de la función, hasta que finalmente se había comentado todo el código y sin embargo seguía ocurriendo la misma excepción. Lo que restaba entonces era que había un problema cuando se realizaba el llamado a dicha función. En ese momento, fue cuando caímos en la cuenta que nunca se había inicializado la pila del kernel. Asombrosamente, este error fue arrastrado desde el trabajo práctico realizado durante el cuatrimestre, pero en ese momento no había surgido este problema. La solución entonces fue cargar los registros *EBP* y *ESP* con la dirección destinada a la pila del kernel.

Más adelante, una vez que la cantidad de código había aumentado, nos topamos con otro error. Pero este error era distinto por una simple razón: el error se daba al compilar el kernel. En un comienzo no se entendía el por qué del problema. Luego de insistir un poco, pudimos entender a qué se debía. Dentro del código de error que nos daba al compilar, logramos ver que había algún problema con una variable llamada *GDT_DESC*. Esta variable pertenecía al código C de la GDT, y era la que contenía el tamaño y la dirección de la GDT. Notamos que este error dejaba de ocurrir si borrabamos o comentabamos líneas de código o eliminabamos alguna estructura de datos grande. Hecho esto, pudimos compilar y correr el código para poder debuggear. Lo primero que chequeamos fue la dirección en la que se alojaba dicha variable, la cuál rondaba por 0xFFxx (no registramos exactamente el valor para este apartado, pero estaba muy cerca de 0xFFFF). Revisando en qué parte del código se utilizaba esta variable, se notó que la misma sólo se usaba en las primeras líneas de código para cargar la GDT y poder así pasar luego a modo protegido. En este momento, recordamos que esas líneas de código utilizan el procesador en modo real, por lo que el código era procesado en 16 bits. Lo que ocurría entonces es que la dirección de la variable *GDT_DESC*, había superado la 0xFFFF y dicha dirección no era alcanzable por el código de 16 bits. Como no logramos encontrar la forma de indicarle al compilador GCC dónde queríamos que compilara cierto código, la solución que adoptamos fue implementar el arreglo de la GDT en assembler, dentro del archivo *kernel.asm*. De esta manera nos aseguramos que el mismo quede en posiciones

bajas de la memoria para que este error no volviera a ocurrir.

Una vez adentrados en la construcción del kernel, se optó por probar si se podía correr alguna tarea junto con el kernel utilizando todo lo que se había implementado hasta el momento, que si bien había sido probado por separado nunca se había puesto a prueba en una situación un poco más real. Lo primero que se intentó fue correr una tarea que se ejecutara en modo kernel es decir, igual a lo que se había realizado durante la cursada. En este caso no hubo inconveniente. Sin embargo, sí lo hubo al intentar correr una tarea en modo usuario. En este caso, cuando se intentaba realizar el switch de tareas, se producía una excepción de protección general (#GP). Luego de ahondar un poco en el tema y de realizar algunas consultas a Alejandro Furfaro y a Federico Raimondo, pudimos resolver el problema modificando el RPL del selector con el cuál se hacían los cambios de contexto, es decir cuando se ejecutaba la instrucción *jmp selector:offset*.

Sin embargo, este problema no terminó allí. Al momento de pasar a ejecutarse una tarea, sucedió que se producía un *page fault*. Luego de hacer algunos debuggeos, notamos que este error se producía cuando la tarea intentaba escribir un dato en la pantalla. Estabamos seguros que el problema era de permisos, pero no lográbamos encontrar en dónde estaba. Luego de una nueva consulta, caímos en la cuenta que el error estaba en el permiso de la tabla de páginas. Cuando se había implementado la función mapear página, una de las verificaciones que se realizaba era para ver si la tabla de páginas estaba presente o no. Si no estaba, se alojaba una nueva página para la misma y se la mapeaba dentro del directorio de páginas en la entrada correspondiente, utilizando los mismos atributos/permisos que se usaban para la página que se estaba mapeando. El problema fue que el primer mapeo que se realizaba tenía permisos de lectura únicamente. Por lo que para la tabla de páginas que se utilizaba, sólo podían realizarse lecturas. Para corregir este error entonces, se procedió a modificar levemente el código de la función *mapear_pagina* verificando si el permiso de la página es de escritura y el de la tabla no, en cuyo caso se modificaban los permisos de la tabla de páginas. De esta manera, siempre prevalece el permiso más alto.

Uno de los problemas graves que nos surgieron a la hora del manejo de tareas fue cuando intentamos eliminar una tarea del scheduling. La idea principal era poder hacerlo a través de una llamada al sistema, pero sin embargo el primer objetivo fue lograr hacerlo desde alguna función que se pudiese llamar utilizando la consola. Algo así como el "kill" del sistema GNU/Linux (un poco más humilde en realidad). El primer problema que surgió fue que en algunas ocasiones funcionaba bien y en otras mal. La excepción que se producía era un *page fault*. Luego de varios debuggeos, observamos que este error se producía cuando se estaba ejecutando la tarea a matar. Claramente, esto sucedía porque dentro de todas las acciones a realizar para eliminar a la tarea, una de ellas era vaciar (dejar en 0) el directorio de tablas de páginas. Claramente, estas posiciones de memoria no estaban dentro del mapeo de la tarea, por cuestiones de seguridad, además de no ser necesario en absoluto. Había varias soluciones distintas posibles de implementar, pero elegimos una que a los fines prácticos era más sencilla y que al igual que las otras, cumplía con los requisitos necesarios. Lo que se hizo entonces fue que el "matar" de una tarea fuese simplemente quitar a la tarea del scheduling modificando los punteros de las tareas que estaban alrededor de la tarea a matar y marcarla dentro de la BCP como tarea a matar. Luego, se agregó dentro del código del kernel que se buscara

a las tareas a matar y que en caso de haberlas, ejecute ahora sí el código que se encargaba de limpiar y desocupar todas las estructuras utilizadas para dicha tarea (pilas, directorio y tablas, BCP, TSS, GDT, etc). Como esto se ejecutaba en el contexto del kernel, al tener el kernel toda la memoria mapeada en el directorio, quitamos el problema de los permisos a la hora de la limpieza.

Luego de esta modificación, pudimos encontrar un problema que estábamos acarreando en una rutina de atención de interrupciones. Cuando corriamos el kernel, luego de un tiempo de inactividad, recibíamos una excepción #GP. Luego de mucho debuggear, notamos que lo que ocurría es que se estaba realizando una llamada a la función que realizaba la limpieza de una tarea a pesar que en ningún momento se le había indicado. Más aún, ni siquiera se había cargado una tarea. Lo que ocurría es lo siguiente: en la ejecución del kernel, se realizaban 2 pasos:

- Se llama a una función que busca una tarea marcada como "a matar" dentro de la BCP
- Si el resultado de esta búsqueda es positivo (es decir, si hay alguna tarea así marcada), se llama a la función que limpia el contexto de una tarea.

El problema fue que al parecer, esta función estaba retornando un valor erróneo. Sin necesidad de realizar un debbugeo, se descartó esta posibilidad porque el código de la esta función era muy simple y fácil de revisar y no se encontraron errores. Luego de analizarlo un tiempo, supimos que el único código que se ejecutaba para poder reproducir el error era el código del kernel (el loop infinito que realiza los 2 pasos antes mencionado) y el handler de atención de interrupción del timertick (incluído el scheduler). Fue entonces cuando observamos que en la atención de la rutina, no se estaban guardando los registros de interés general. El problema era entonces que justo luego de ejecutarse todo el código de la función que buscaba la tarea a matar y antes que devolviera el resultado, se producía una interrupción del timertick que pisaba entre otras cosas, el valor del registro eax, registro en el que se devuelve el resultado de una función en C, es decir, se estaba pisando el resultado de la función que buscaba una tarea a matar. Esto hacía que se ejecutara la "función de limpieza" en un momento equivocado y que se produzca el error ya mencionado. Este problema se solucionó simplemente pusheando los registros generales ni bien se entraba al handler del timertick y popeandolos justo antes de salir.

Todos los errores antes mencionados fueron los que más dolores de cabeza e insomnio nos produjeron. Sin embargo, los errores que tuvimos en la realización de este TP son incontables y los problemas acarreados por los mismos, muchos más. Nombrarlos y explicarlos todos nos llevaría mucho tiempo y sería poco práctico. Creemos que con esta compilación de errores podemos dar una idea de qué tan complicado puede ser encarar un proyecto de este tipo y de cómo a veces el código más insignificante puede hacer que las cosas cambien drásticamente.

6. Algunas conclusiones

Nuestra decisión de hacer un trabajo práctico sobre scheduling dinámico fue entre muchas cosas, por el hecho de poder entender un poco mejor cómo funcionan por detrás muchas

cosas de un sistema operativo. En este sentido, creemos que nuestro objetivo fue cumplido. Gracias a este proyecto, no sólo aplicamos conocimientos adquiridos durante la cursada, sino que también pudimos reforzarlos y aprender muchas cosas más sobre el tema de system programming.

Creemos que hay muchas cosas más que se podrían agregar a nuestro kernel, para hacerlo más robusto y completo, pero las mismas exceden los fines de la cursada, además de estirar por más tiempo la finalización de este trabajo práctico. Sin embargo, esto no significa el fin de este viaje, sino que al contrario, nos abre otras puertas. El haber recorrido este camino, nos sirvió para conocer un poco el mundo de la programación de sistemas operativos, despertando en alguno de nuestros integrantes ganas por continuar creciendo dentro de esta rama.

PONER COMO BIBLIOGRAFIA CONSULTADA LOS MANUALES DE INTEL DE SYSTEM PROGRAMMING Y DEL SET DE INSTRUCCIONES, LA PAGINA OSDEV.ORG, EL TRABAJO PRACTICO QUE HICIMOS DURANTE EL CUATRIMESTRE, LAS TEORICAS VISTAS EN CLASE Y LAS SLIDES DE LAS CLASES PRACTICAS DE SYSTEM PROGRAMMING. SE PUEDE PONER ALGUNA “CONCLUSION” CHAMUYANDO UN POCO DE POR QUE ELEGIMOS EL TEMA