

Organización del Computador II

Trabajo Práctico 3

Emilio Almansi
ealmansi@gmail.com

Álvaro Machicado
rednaxela.007@hotmail.com

Miguel Duarte
miguelfeliped@gmail.com

2^{do} cuatrimestre de 2013

Índice

1. Introducción	2
2. Modo real y modo protegido	3
3. Global Descriptor Table (GDT)	4
4. Paginación	5
5. Interrupciones	6
6. Manejo de tareas	7
7. Protección	8
8. Scheduler	9
9. Conclusiones	10

1. Introducción

Se desarrolló un kernel elemental para la arquitectura Intel x86 con soporte básico para las unidades de segmentación y paginación, manejo de interrupciones y cambio automático de tareas por hardware.

El alcance del trabajo incluye múltiples etapas del proceso de arranque de todo sistema operativo, permitiendo apreciar las herramientas que provee el procesador para la administración del sistema. Durante la etapa de inicialización, es necesario llevar a cabo distintos pasos como el traspaso de modo real a modo protegido, la habilitación de la unidad de paginación, o el armado de múltiples estructuras requeridas por el procesador para el manejo de interrupciones o de tareas.

Adicionalmente, se implementó un scheduler rudimentario permitiendo realizar la ejecución por tiempo compartido de múltiples tareas definidas al tiempo de compilación, realizando un ciclo de ejecución no lineal guiado por tics del reloj.

2. Modo real y modo protegido

En la GDT hay que poner los descriptores de segmento y los descriptores de TSS para cada tarea y para cada bandera.

La misma está representada como un arreglo “*gdt_entry*” declarado de manera global en C. Las *gdt_entry* son structs de 4 bytes que poseen un campo para cada atributo de una entrada de gdt.

Los descriptores de segmento fueron cargados de manera estática en tiempo de compilación. Lo mismo con el descriptor de la IDT. Esto fue posible porque se conocen de antemano todos los valores necesarios para completar los descriptores.

A la hora de cargar los descriptores de TSS nos encontramos con la siguiente dificultad: Los descriptores de TSS fueron declarados como una variable global en código C. Por lo tanto en tiempo de compilación no se sabe en qué dirección van a ser cargados. Por este motivo se cargan de manera dinámica mediante una función que se llama desde *kernel.asm*. La función sencillamente crea una entrada más en el arreglo que representa de *gdt_entry* con los atributos adecuados.

3. Global Descriptor Table (GDT)

En la GDT hay que poner los descriptores de segmento y los descriptores de TSS para cada tarea y para cada bandera.

La misma está representada como un arreglo “*gdt_entry*” declarado de manera global en C. Las *gdt_entry* son structs de 4 bytes que poseen un campo para cada atributo de una entrada de gdt.

Los descriptores de segmento fueron cargados de manera estática en tiempo de compilación. Lo mismo con el descriptor de la IDT. Esto fue posible porque se conocen de antemano todos los valores necesarios para completar los descriptores.

A la hora de cargar los descriptores de TSS nos encontramos con la siguiente dificultad: Los descriptores de TSS fueron declarados como una variable global en código C. Por lo tanto en tiempo de compilación no se sabe en que dirección van a ser cargados. Por este motivo se cargan de manera dinámica mediante una función que se llama desde kernel.asm. La función sencillamente crea una entrada más en el arreglo que representa de *gdt_entry* con los atributos adecuados.

4. Paginación

La parte de paginación se resolvió de manera bastante intuitiva.

Se crearon funciones en C que se encargan de inicializar los directorios de páginas del kernel y de las tareas. Un detalle importante de la implementación es que tanto el kernel como las tareas comparten las primeras 2 tablas de páginas de sus directores de páginas.

Estas dos tablas son las que se hacen con identity mapping. El identity mapping está en todos los mapas de memoria de la misma manera y con los mismos atributos. Además nunca debe ser cambiado a lo largo de la ejecución de todo el programa en ninguno de los mapas. Por eso decidimos crear sólo 2 tablas de páginas y hacer que todas las tareas lo compartan.

En un principio a cada tarea se le asigna una tabla extra inicializada en cero. Luego mediante las funciones para mapear páginas se completan estas tablas de manera adecuada para que se efectivicen los mapeos.

Es importante notar que el resultado final de esto es que cada tarea tiene mapeadas 2 tablas con identity mapping y con privilegios restringidos (sólo para supervisor) y luego un par de páginas mas con permisos de usuario, que son donde efectivamente va a trabajar.

Todo eso se englobó en las siguientes funciones de C:

```
void mmu_inicializar_dir_kernel();  
void mmu_inicializar_paginas_kernel();  
void mmu_inicializar_tareas();
```

mmu_inicializar_tareas no solo inicializa los directorios de páginas sino que además hace los mapeos de páginas correspondientes.

Finalmente esas funciones se llaman dentro de kernel.asm. Una vez terminado eso se habilita la paginación.

5. Interrupciones

Las interrupciones se manejaron de 2 maneras diferentes:

Por un lado están las excepciones de sistema. De esas sólo se busca que si una tarea las produce esta sea desalojada. De esta manera se pudieron implementar mediante un macro sin mayores inconvenientes.

Por otro lado están las otras interrupciones: La de reloj, la de teclado y las syscalls. Estas si se tuvieron que elaborarse de manera mas detenida.

La interrupción de teclado basicamente es un gran *switch* que verifica si el contenido del código obtenido. Si este coincide con alguna de las teclas esperadas ("e", "m", "0-9") entonces realiza la acción debida en pantalla.

En las syscalls se tiene el cuidado de verificar quién las llama y como. Muchas posibles llamadas a syscalls son ilegales. Por ejemplo si la llamada a una `int0x50` la realiza una bandera esta debe ser desalojada. Lo mismo si esa interrupción es llamada con parámetros ilegales. Toda esta lógica se implementó mediante una función escrita en C y llamada desde assembler.

La interrupción de reloj terminó siendo la mas compleja. Para implementarla lo que se hizo, entonces, fue realizar una función en c que se llama desde la interrupción. Esta función verifica si quién se estaba ejecutando al momento de la interrupción de reloj era una bandera. En ese caso la desaloja. Luego le pide al scheduler el índice de la GDT que necesario para realizar un JUMP far al contexto de la otra tarea. A continuación se fija si la próxima tarea a ejecutarse es una bandera. Si es una bandera entonces primero reinicia su tss y luego, si, realiza el JMP FAR.

6. Manejo de tareas

El manejo de tareas tiene varias aristas. Por un lado está la gestión “física” de las tareas. Es decir como se cambia entre el contexto de una y el contexto de otra. Este punto se maneja mediante el mecanismo automático que provee la arquitectura intel.

Para esto hubo que crear tss para cada tarea. Además por cuestiones de facilitar la gestión de las tareas se decidió crear una tss extra para cada bandera. De esta forma cada navio tiene 2 tss, una para correr su código de acción y otra para correr su código de bandera. Además hay 2 tss extra, una para la tarea idle y otra auxiliar, para poder realizar el primer salto sin inconvenientes.

Para poder hacer el primer cambio de contexto (JMP FAR) sobre esta tss auxiliar previamente se setea el TR. Este seteo está hardcodeado porque el descriptor de la tss auxiliar siempre se guarda exactamente en el mismo índice de la gdt.

Todos los cambios de contexto se hacen mediante un JMP FAR. En ningún momento se usa ningún otro mecanismo. El problema que surge con esto es que las banderas no tienen un código cíclico. Es decir, las banderas ejecutan una porción de código que termina, al terminar su contexto queda estático en la posición en la que terminó, por lo tanto se llama a esa bandera otra vez en ese mismo contexto sin hacer nada la bandera va a ejecutar cosas indebidas. Para subsanar ese inconveniente siempre antes de saltar una bandera se vuelve a inicializar su tss, de esta forma nos aseguramos de que el contexto siempre sea el contexto inicial.

7. Protección

Las tareas corren en el anillo de seguridad 3. El kernel en anillo 0. A nivel paginación las tareas son usuario y el kernel es supervisor.

Las tareas no pueden acceder a memoria del kernel bajo ninguna circunstancia. Esto está asegurado en el mapa de memoria. El mismo si bien incluye las tablas de memoria del kernel las incluye con un privilegio mayor, por lo tanto si una tarea intenta acceder a esas páginas cae un page fault que termina en el desalojo de esa tarea.

El kernel está programado para no tener que incurrir en ninguna excepción especial, lo tanto el código que se ejecuta cuando ocurre una excepción es siempre el mismo. Desalojar a esa tarea y saltar a la tarea iddle. Esto asegura que toda tarea que realice una acción ilegal que produzca una excepción (dividir entre cero, seg fault, page fault, etc) será desalojada.

Además hay un par de cuestiones extra. Por ejemplo, una bandera no puede llamar a una `int0x50`. Si hace esto inmediatamente es desalojada.

8. Scheduler

Dadas las dieciséis tareas del sistema -ocho navíos, ocho banderas-, el scheduler desarrollado realiza un ciclo de ejecución sobre todas las tareas alternando de la siguiente forma: se dedican tres tics de reloj a ejecutar navíos, se procesan todas las funciones bandera pertenecientes a navíos activos, y luego se retoma la ejecución de los navíos. Los navíos se ejecutan secuencialmente por lo que todos reciben la misma cantidad de tics mientras no sean desalojados.

Para implementar esta lógica, se realizaron colas circulares basadas en arreglos para los índices de la tabla GDT de los navíos y las banderas¹. Como no se ingresan nuevos elementos dinámicamente, el mecanismo de eliminación de una tarea se puede realizar sencillamente marcando su entrada con un valor inválido, como el 0 ya que nunca es un índice válido en la tabla.

Para mantener el orden de ejecución descripto, se guardaron adicionalmente contadores para la cantidad de navíos activos en la cola, la cantidad de navíos pendientes hasta alternar con la ejecución de banderas, e inversamente la cantidad de banderas pendientes.

De esta forma, se mantiene el invariante de que al menos uno de los dos contadores de tareas pendientes es nulo, y se consumen elementos de la cola opuesta hasta terminar la ronda. Es decir, inicializando la cantidad de navíos pendientes a 3 y la cantidad de banderas pendientes a 0, se dedicarán 3 tics a ejecutar navíos, y luego se asigna la cantidad de tareas activas al contador de banderas pendientes, asegurando que se llamen todas las banderas disponibles.

Como ya se dijo, cuando una tarea necesita ser desalojada, su entrada correspondiente y la de su navío o bandera asociada se anulan, permitiendo evitar su ejecución simplemente saltando los ceros que se encuentren en la cola.

¹De los descriptores de sus tablas TSS, para ser preciso.

9. Conclusiones

A lo largo de todo el trabajo se presentó la enorme cantidad de inconvenientes que conlleva trabajar sobre la nada. Para realizar el trabajo fue imprescindible entender lo que estaba sucediendo en el kernel a muy bajo nivel, pues constanemente surgen inconvenientes cuya explicación no suele ser evidente.

También una cosa que se presentó es que la dificultad de la implementación asciende al infinito si uno lo permite. En todo momento se presentan ecrucijadas sobre como realizar algo específico teniendo una libertad casi absoluta al tener permisos de kernel. En general en esos momentos intentamos optar por la implementación mas clara con una lógica mas sólida perdiendo en algunos casos algo de performance. Sin embargo lo que se ganó con eso fue un código mucho mas manejable.