Trabajo Práctico 1 ∼ PSO

Liptak, Leandro
Reboratti, Patricio
Silvani, Damián

May 6, 2011

Documentación

Kernel Básico

`kernel` - Inicialización

El proceso de inicialización de nuestro kernel consiste en hacer que los demás módulos se inicialicen en un orden consistente (primero los módulos críticos como `vga`, `idt` y `gdt`), dotar luego al kernel de un esquema de paginación y activar paginación, y finalmente cargar el registro de tarea actual, colocar el segmento de datos de anillo 0 como segmento de pila de anillo 0 en la TSS (que posteriormente jamás será cambiado), activar las interrupciones externas y convertirse en la tarea inactiva.

Cabe destacar que como parte de la inicialización de los demás módulos se registran las interrupciones y excepciones pertinentes.

`gdt` - Global Descriptor Table

La inicialización de la GDT fue sencilla: a los dos descriptores existentes (código y datos de nivel 0) adicionamos tres descriptores más, uno de código de nivel 3, otro de datos de nivel 3 (ambos flat), y el descriptor del *Task State Segment*. El tamaño de este último fue el mínimo posible pues no adicionamos el mapa de puertos de E/S.

`vga` - Pantalla

...

`idt` - Administrador de interrupciones

Como creímos (equivocadamente) que iba a resultar en código mantenible y facil de leer, definimos tres *wrappers* comunes para los *handlers* de interrupción definidos en la IDT: `ISR_NOERRCODE`, `ISR_ERRCODE` y `IRQ`.

Los primeros dos corresponden a un handler de excepción del procesador, y el tercero a un IRQ proveniente del PIC. La diferencia entre los dos primeros está dada en que hay ciertas excepciones en las que el procesador pushea un código de error a la pila antes de saltar al handler. Ambos finalmente hacen `jmp` a un handler común llamado `isr_common`.

Por otro lado `IRQ` salta a `irq_common`, previamente pusheando el número de IRQ y de la interrupción tal como está registrada en la IDT. Finalmente salta a `irq_common`.

Luego del salto, ambos pushean los registros de uso general y llaman a una función definida en C. Luego del `call` se acomoda la pila y se ejecuta finalmente `iret`.

Una vez en el mundo C, llamamos al handler definitivo registrado en un arreglo `interrupt_handlers` utilizando en número de interrupción obtenido de la pila. Aprovechando la pila que nos deja el procesador al principio del proceso de manejo de interrupción, y los otros valores que pusheamos antes, definimos una estructura en C llamada `registers_t` que contiene toda esta información, necesaria para cualquier handler que sea necesario registrar.

En el caso particular de las IRQs, antes de llamra al handler definitivo, se envía una serie de comandos a la PIC maestra y esclava para señalizar que la interrupción está siendo atendida.

El método `idt_register`, además de registrar en el arreglo `interrupt_handlers` el puntero al handler en C de la interrupción pedida, debe definir la entrada correspondiente en la IDT, con el puntero al handler en ASM definido por las macros mencionadas al principio. Es por eso que fue necesario un segundo arreglo, esta vez de punteros a los handlers en ASM.

Lo que al principio pareció ser una buena idea, terminó complicando la implementación, además de ser menos eficiente en espacio y tiempo. Para los dos arreglos de punteros de 256 entradas, se necesitan 2KB de espacio de kernel. En cuanto a la eficiencia, se realiza un `call` extra para todo handler de interrupción.

Surgió otro problema en la etapa de desarollo del módulo `loader`, dónde necesitamos que el handler del PIT sea rápido y tuviera una pila lo más chica posible, y el wrapper común nos dificultaba. Terminamos definiendo un `idt_register_asm` alternativo, tal como está especificado en la consigna del TP, que permitiera definir un handler en ASM, la cual es registrada en la IDT.

`debug` - Debug

El módulo de depuración provee una función (`debug_kernelpanic`) para mostrar el estado del procesador (dirección dónde ocurrió la excepción, registros al momento de la misma, código de error si lo hubiese) y la memoria (volcado de la pila, backtrace desde la rutina que estaba ejecutando) frente a una excepción. Como parte de su inicialización registra todas las excepciones para ser manejadas por la rutina antes mencionada, y la interrupción espúrea (IRQ7) para ser manejada por una rutina que muestra por pantalla el por qué del nombre de nuestro pequeño sistema operativo.

Adicionalmente se provee una rutina para mostrar mensajes de log, los cuales son impresos por pantalla con la información del TSC (*Timestamp Counter*) actual del procesador.

Kernel funcional

`mm` - Manejador de memoria

La principal decisión de diseño que tuvimos que tomar sobre el manejador de memoria fue cómo íbamos a representar la memoria libre. En un principio listas enlazadas parecieron una buena opción si implementábamos una función para otorgar una cantidad arbitraria de memoria pues podíamos tener un solo nodo representando una cierta cantidad de memoria libre contigua. Sin embargo esto se tornaría difícil de implementar y nos volcamos hacia las funciones sugeridas que otorgaran y liberaran páginas. En este caso, una lista enlazada no presentaba ventaja alguna frente a un mapa de bits, motivo por el cuál optamos por este último.

Para obtener la cantidad de memoria disponible hacemos una llamada a una interrupción del BIOS (aún estando en modo real) que nos provee un mapa de memoria, el cual pasamos como argumento a la función `kernel_init`.

Respecto a los directorios y tablas de páginas, creamos una tabla de páginas que tiene mapeados los frames desde 4KB a 4MB - 4KB y almacenamos su dirección en una variable global pues es utilizada al crear nuevos directorios de páginas (se enlaza dicha tabla a la primer entrada en el directorio de páginas de modo de tener mapeado con *identity mapping* toda la memoria en el rango 4KB a 4MB - 1).

`sched` - Scheduler

`loader` - Loader y administrador de tareas

La inicialización del módulo consiste en registrar el handler de la IRQ0 (*System Timer*), inicializar el módulo del timer con la frecuencia adecuada para el

mismo y dar de alta el PCB con la información del proceso inactivo del sistema (IDLE). A su vez se inicializan los diversos arreglos de procesos para indicar la disponibilidad correspondiente.

El diseño del PCB (*Process Control Block*) fue variando a medida que el cambio de contexto iba tomando una forma definida. En principio no teníamos del todo claro cómo íbaoms a hacerlo, así que manteníamos en el PCB todo el estado de la arquitectura (todos los registros). Posteriormente entendimos que nuestro cambio de contexto iba a salvar la mayoría de los registros en la pila de la propia tarea de forma implícita, así que resultó inútil seguir manteniéndolos en el PCB y los eliminamos. Sólo dejamos allí aquellos registros que se dificultaba guardar en la pila, estos son el CR3 (que define el esquema de paginación) y el ESP (que define el tope de la pila). Adicionalmente guardamos el propio ID de proceso (que resulta redundante pues coincide con el índice dentro del arreglo de procesos `processes` pero sirve además para indicar si dicha entrada del arreglo se encuentra libre) y su nivel de privilegio. Finalmente al diseñar el esquema de colas se adicionaron los dos últimos campos `prev` y `next` para que el mismo PCB actúe como nodo de la cola en la que se encuentra bloqueado.

Respecto a la creación de una nueva tarea, las decisiones de diseño tomadas fueron las siguientes:

- Toda tarea sin importar su nivel de privilegio utiliza páginas del área de usuario (desde los 4 MB en adelante) para su pila, código y datos
- Los directorios y tablas de páginas utilizan actualmente páginas del área de kernel por una cuestión de comodidad: para evitarnos mapeos temporales al momento de tener que inicializarlas, ya que en ese instante está ejecutando otra tarea con otro esquema de paginación en el cual esos frames no se encuentran mapeados, en cambio el espacio de memoria del kernel siempre se encuentra mapeado, sin importar la tarea.
- Solo se efectúa un mapeo temporal, utilizando siempre la misma dirección virtual para copiar los datos a el/los frames necesarios.

En lo que respecta al cambio de contexto, como se ha mencionado el esquema sufrió variaciones a lo largo de la implementación. Finalmente resultó en una rutina en lenguaje ensamblador que esperaba en una variable global `tmp_pid` el *process id* destino al cuál hacer el cambio. Dicha rutina salva en la pila todos los registros mediante una instrucción `pushad` y sólo salva en el PCB el CR3 y ESP como se ha mencionado. Al momento de crear una tarea se arma para la misma una pila de anillo 0 con una estructura consistente con la que esta rutina espera encontrar luego de hacer el cambio esquema de paginación y pila. Dicha pila contiene los registros en el orden que son apilados por la instrucción `pushad` y una dirección de retorno de la rutina. Inicialmente esta dirección de retorno contine la dirección de una rutina que sólo efectúa una instrucción de retorno de interrupción, con lo cual también inicializamos la pila de forma consistente

a lo que dicha instrucción espera encontrar, dependiendo el nivel de privilegio de la tarea para así poder indicar el cambio a una pila de anillo 3 o no, según corresponda. Finalmente cabe destacar que la rutina de cambio de contexto es llamada no sólo desde la interrupción del reloj sino también cuando una tarea se bloquea frente a algún evento o termina.

`sem` - Semáforos de kernel