Programación de Sistemas Operativos

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

Trabajo Práctico 1

Desarrollo de un Kernel básico

Integrante	LU	Correo electrónico
Alejandro Mataloni		amataloni@gmail.com
Emiliano Mancuso		emiliano.mancuso@gmail.com
Martin Miguel		m2.march@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

El presente trabajo práctico comienza con el desarrollo de un sistema operativo. Este práctico conforma una parte de una serie de trabajos. El acumulado de estos trabajos llevaran a la consolidación de un kernel pequeño pero funcional. El objetivo de esta primera entrega es crear las unidades básicas de funcionamiento del kernel que sirvan como cimientos para los próximos avances. Entre estas unidades básicas se encuentran:

- una biblioteca para el manejo de la pantalla
- un sistema de debug basado en interrupciones
- un módulo encargado del manejo de la memoria mediante paginación
- un módulo capaz de cargar y descargar tareas del procesador así como manejar colas de espera entre ellas
- un módulo que implemente un algoritmo de scheduling para manejar el orden en que se ejecutarán las tareas en el procesador y la implementación de semáforos para operaciones de kernel

La red conformada por la relación de todos estos sistemas serán luego la base para la construcción de nuevas estructuras y funcionalidades en el kernel en los próximos trabajo prácticos.

2. Módulos

En esta sección se explica la funcionalidad exportada de cada uno de los módulos desarrollados, así como detalles de implementación importantes para comprender el funcionamiento del módulo o para cuidados necesarios en su uso o futuros desarrollos.

2.1. Inicialización del kernel

Este módulo se conforma por dos archivos encargados de toda la inicialización del kernel. El principio de la ejecución de nuestro código sucede en *kinit.asm*, que activa la línea a20 y pasa a modo protegido. Para ello utiliza la gdt estática declarada en *gdt.c*. Esto luego desemboca en la función `kernel_init()`. Esta función es la encargada de llamar a los inicializadores de los módulos de forma que todos los aparatos del kernel estén listos para trabajar.

2.2. GDT

El módulo GDT, conformado por los archivos *gdt.c* y *gdt.h*, es la definición en C de la gdt. En esta se definen estáticamente los 4 segmentos flat de código y de datos, dos para anillo 0 y dos para anillo 3. Además se define una entrada para la única TSS que utilizamos. Esta debe definirse dinámicamente y lo hace la función `gdt_init()`. Esta GDT representa un modelo de segmentación flat, la forma de eludir el sistema de segmentación del procesador. La idea detrás de esto es usar el sistema de paginación para el manejo estructurado de la memoria física y como mecanismo de memoria virtual.

2.3. VGA

Este es el módulo que provee la funcionalidad para el acceso a la pantalla. En esta etapa la pantalla solo se accede en modo texto, escribiendo caracteres es un espacio de 25 filas por 80 columnas, a 2 bytes por celda. Para trabajar en la pantalla se utilizan 3 funciones: `vga_write(...)`, `vga_printf(...)` y `printf (...)`. La primera permite escribir un string en algún lugar de la pantalla especificando fila, columna y formato del texto. La segunda permite escribir un string formateado definiendo también posición en pantalla y color del mismo. Un string formateado es un string que declara internamente lugares que son rellenados con el valor en string de la variable. Esto lo hace de forma análoga al string *fmt* de la función *printf()* de la biblioteca standard de C, aunque con funcionalidades limitadas. Para hacer esto se basa en una biblioteca creada para este sistema, *lib_str.c*, que contiene funciones necesarias para la generación del string final. Por último, la función `printf(...)`, hace una réplica más fuerte a la función de C ya que trata a la pantalla como una consola. Esta función imprime al principio de la siguiente fila que ya fue escrita por la función, y de llegar al final de la pantalla mueve todo lo escrito una fila hacia arriba. Para mantener la fila actual se usa una variable global que solo es modificada por esta función. Esta función resultó sumamente útil para imprimir resultados de funciones de evaluación de nuestros módulos.

2.4. Interrupciones

Para trabajar con interrupciones se armaron los siguientes 3 archivos con distintas funciones relacionadas: *idt.c*, *isr.asm* y *interrupt.mac*. En *idt.c* está declarada la idt propiamente dicha y se cuenta con `idt_init()`, lugar para inicializaciones varias relacionadas a la idt, y `idt_register(...)` que permite registrar una función como respuesta a un número de interrupción dado. La `idt_init()` además remapea los PICs a la posiciones 0x20 y 0x28, de forma que las interrupciones de el mismo sean consecutivas a las del procesador. Para ello usamos `pic_reset(...)` y `pic_enable()`) del archivo *pic.c*.

Por distintos motivos, es necesario que las funciones que son accedidas mediante una interrupción estén en assembler. Estas definiciones se encuentran en distintos archivos .asm. En particular *isr.asm* es

un lugar para declarar rutinas de atención a interrupción varias. Este archivo funciona que conjunto con *interrupt.mac*, donde están declarados varios macros que facilitan el trabajo. El macro `isr_define_ep` sirve para declarar una isr standard que solo llama a una función C, donde no hubo código de error por culpa de la interrupción. El primer parámetro del macro define el nombre de la isr y el segundo la función de C a llamar. El macro `isr_dkp_e` sirve para definir una isr que llama a `debug_kernelpanic` (ver módulo Debug). Recibe como parámetros una etiqueta para la isr y un número de interrupción. La otra versión del macro, `isr_dkp_E` es idéntica a la misma pero asume que el procesador dejó en la pila un código de error y se encarga de trabajarla bajo esas condiciones. Estos macros se encargan de preparar en pila los parámetros para llamar a `debug_kernelpanic(...)`.

2.5. Debug

En este nivel de trabajo, nuestra mayor herramienta de debug (además de las provistas por la máquina virtual) son las mismas interrupciones del procesador. El objetivo del módulo de debug es tratar de salvaguardar alguna información del estado de la máquina cuando se produce una de estas interrupciones. La función `debug_kernelpanic(...)` recibe información del estado de la pc (valor de distintos registros de propósito general y de control) al momento del la interrupción inesperada y la imprime en formato de una pantalla de *kernel panic*. La función `debug_init()` registra todas las interrupciones a la función de impresión utilizando los macros `isr_dkp_*` en *interrupt.mac*. En debug también se declara y registra en idt una función particular para la interrupción del timer tick de forma que esta no frene el funcionamiento de la pc. La función (`isr_timerTick_c`) imprime un relojito en la esquina inferior derecha de la pantalla para mostrar que el mismo está interrumpiendo.

2.6. Memory Management

El manejo de la memoria física y virtual se hace utilizando los sistemas de paginación del procesador. *mm.c* contiene todos los métodos para trabajar con estos sistemas. Internamente es necesario mantener información sobre los marcos de páginas. Considerando que se utilizan separadamente algunos marcos para kernel y otros para usuario, estos se tratan de forma se parada. El sistema de directorios del procesador contiene numerosa información sobre el uso de las páginas, en su sentido virtual, es por esto que nosotros debemos guardar aparte información sobre los marcos de página. En particular nos interesa saber cuales están libres y cuales no. La estructura utilizada para guardar esta información es un arreglo de enteros, donde cada bit se corresponde con un marco de pagina. El bit en 1 indica que la página está ocupada, en 0 que está libre. Estas estructuras se acceden mediante las funciones `void* mm_mem_alloc()`, `void* mm_mem_kalloc()` y `void mm_mem_free(void* page)`. A fines de mantener modularidad, estas funciones trabajan solo con las estructuras de marcos de página, y no con las tablas de páginas y directorios. Para lograr verdaderos efectos en el sistema de paginación estas funciones deben utilizarse en conjunto con otras.

Para trabajar particularmente con un directorio de páginas se desarrollaron las funciones `mm_page_map`, `mm_page_free`, `mm_dir_free`, `mm_table_free` y `mm_dir_unmap`. Todas estas funciones reciben o devuelven direcciones a distintas secciones del sistema de directorios y se encargan de mapear y desmapear direcciones virtuales a reales en estas estructuras. A excepción de `mm_dir_new` y `mm_dir_free`, estas funciones no se preocupan si las páginas físicas a las que se están refiriendo están ocupadas o no, o si están marcadas como ocupadas en nuestras propias estructuras. Es responsabilidad de quién las utilice de asegurarse que las direcciones provistas sean correctas. `mm_dir_new` y `mm_dir_free` son funciones más autocontenidas que arman y desarman directorios de páginas, pidiendo y devolviendo nuevos marcos de páginas. `mm_page_map` también pide marcos al kernel si requiere una nueva página para una tabla de páginas.

Las estructuras del **mm** se inicializan con la función `mm_init()`. Esta función llena de 0s la estructuras que mantienen la información de los marcos de páginas. En este sistema se asume que contamos con al menos 4mb de memoria ram, ya que se asume que el mapa de memoria física para procesos de usuario

empieza luego de eso y necesitamos memoria de usuario. De esta forma, la cantidad de marcos de páginas de kernel es fija. En cambio, dado que cuanta memoria tiene el sistema no es un valor constante, el proceso de inicialización debe contabilizar la cantidad de memoria disponible y guardar esta información. Esto se realiza con el método `uint_32* memory_detect()` y la información se guarda en `uint_32 usr_pf_limit` como la cantidad de ints que pueden verificarse en el arreglo que mantiene la información de los marcos de página de usuario. Marcos más allá de ese límite no pueden ser asignados. Para contar memoria se testean secciones de 4kb. La verificación consiste en escribir en la última dirección de la sección y leer. Si el valor es igual al escrito se considera que la dirección es válida y sus 4kb superiores también son memoria accesible. Como solo nos interesa un bloque de memoria contigua, la primera vez que falla la verificación se frena el proceso y la última dirección escrita y leída satisfactoriamente pasa a ser el límite de la memoria útil.

La inicialización también activa **PSE**, genera un directorio de paginas básico, actualiza el cr3 y activa paginación. Al activarse paginación, el proceso empieza a utilizar el directorio de páginas apuntado por el cr3 para convertir direcciones virtuales en físicas. Es por esto que el cr3 debe tener un directorio de páginas que posea correctamente mapeada la siguiente instrucción a ejectuar luego de activar paginación. Un criterio general es que todos los directorios deben tener mapeada la memoria de kernel (los primeros 4mb), ya que las estructuras definidas allí deben estar disponibles siempre. Para mapear los 4mb de manera eficiente se utilizan páginas de 4mb permitidas por la función **PSE** del procesador. En este momento no se está verificando que esta función esté soportada por el procesador, simplemente se asume. Además, por simplicidad, se considera que este primer mapeo es el único de 4mb para cada directorio.

Finalmente, este módulo posee la función `void* sys_palloc()` que es llamada por la syscall `palloc`. Esta syscall es registrada por `mm_init()` como la syscall 0x30, considerando 0x3* el prefijo de las syscalls de memoria. La llamada a sistema permite a una aplicación de usuario pedir una página de memoria y que esta sea mapeada a su espacio de direcciones virtuales. La syscall `palloc` devuelve la dirección virtual de la página habilitada para el usuario.

En `mm_init()` se realizan testeos de todas las funciones nombradas anteriormente. Todas verificaciones imprimen en pantalla. La verificaciones se encuentran comentadas.

2.7. Scheduler

El módulo de scheduler es el algoritmo encargado de decidir cual será la próxima tarea a ejecutarse. Funciona muy relacionado con el módulo **loader**. El **scheduler** mantiene un *array* de **sched-task**. Ésta estructura contiene el estado y dos enteros (next y prev) que sirven para mantener una cola de tareas sobre un *array*. En esta cola, sólo mantenemos enlazadas a las tareas que están en estado *RUNNING*. Además de la cola, el módulo lleva una cuenta del quantum consumido por la tarea actual, el **pid** de la misma y el indice del ultimo elemento de la cola.

2.8. Loader

El módulo del loader se encarga de mantener el **Process Control block**, cambiar el contexto de las tareas, e interactuar con el **scheduler**. Para el **Process Control block**, creamos una estructura que solo guarda el **CR3** y el **ESP0** de la tarea, junto con dos indices (prev y next) para armar las colas donde las tareas se bloquearan. Para administrar los **pids** tenemos una cola de los libres, asi el próximo lo obtenemos en $\mathbf{O}(1)$

2.9. Syscalls

Los archivos *syscalls.h* y *syscalls.asm* proveen las herramientas para el registro de las llamadas al sistema provistas por el kernel. Las syscalls suelen hacerse mediante interrupciones ya que es un método que nos provee el procesador para cambiar de nivel de privilegio. Nosotros decidimos mantener esta idea

y además, para independizarnos de la cantidad de interrupciones posibles en el procesador, las syscalls fueron vectorizadas a dentro de la interrupción `0x30`. Esto es, acceder a una syscall particular significa hacer por software la interrupción `0x30` teniendo en el registro `eax` el número de la syscall a acceder. Este proceso se envuelve en una serie de funciones declaradas en *syscalls.h*, cuya verdadera implementación se encuentra en *syscalls.asm* y realizan las acciones descriptas anteriormente. La interrupción `0x30` está conectada a la isr `isr_syscall` (en *isr.asm*), encargada del proceso de vectorización. El vector que relaciona el número de syscall con una función se encuentra en *syscalls.h*. Este vector debe ser llenado por la inicialización de los distintos módulos que exporten syscalls en la posición correspondiente con la dirección de la función llamada.

2.10. Semáforos de Kernel

Este módulo permite el manejo de semáforos de kernel. Funciona muy relacionado con el módulo **loader**. Los semáforos se conforman de dos enteros sin signo. El primero es el valor del semáforo. Si vale 0 significa que hay uno o más tareas esperando el evento del semáforo. Si es mayor a 0 significa que pueden hacerse tantos `sem_wait(...)` como el valor de la variable sin bloquear el proceso. El segundo entero es una cola, y funciona de la misma forma que las colas en **loader**. La función `sem_wait(...)`, si el valor del semáforo parámetro es 0, encola al proceso actual en la cola del semáforo mediante `loader_enqueue(...)`. La función `sem_signaln(...)` incrementa el valor del semáforo y, si era 0, le avisa al loader que desencole a alguién mediante `loader_unqueue(...)`. La función `sem_broadcast(...)` desencola a todos los miembros de la cola del semáforo y establece su valor en 1, de forma que al menos un proceso pueda pedir el semáforo.

El módulo no maneja otras estrucutras. Los módulos de kernel son libres de registrar semáforos declarando una nueva variable de tipo `sem_t` e inicializándola con el macro `SEM_NEW(uint_32 val)`, que asigna al semáforo el valor `val` y establece la cola como vacía.