TP Final de Orga-2

HISTORIAL DE REVISIONES			
NÚMERO	FECHA	MODIFICACIONES	NOMBRE
1.0	2011-07	Versión inicial	AP

Índice

Resumen

Este trabajo práctico consiste en modificaciones al sistema operativo desarrollado en Orga-2 en el segundo cuatrimestre de 2010. El mismo era bastante elemental, cargándose de un floppy pero manejando la segmentación, paginación y niveles de privilegio del procesador IA-32. Había cuatro tareas compiladas como binarias y cargadas desde posiciones fijas en la memoria. Un scheduler Round-Robin asignaba tiempos en cada interrupción del reloj. Para más detalles, ver el enunciado.

La funcionalidad agregada consiste en: interfaz general para system calls, ejecución dinámica de tareas (independiente de su posición en el ejecutable y de los procesos existentes), manejo del teclado, una interfaz a la consola, varias consolas virtuales (sin que los procesos se enteren), manejo de fallas y excepciones para tareas de usuario (cerrándolas con un aviso), y una librería standard (en forma estática, pero puede usarse tanto en el kernel como en procesos) para programas.

Este informe fue escrito en *AsciiDoc*, un formato liviano que genera *DocBook* (el más usado para documentación en el área no matemática).

1. Introducción

1.1. Observaciones

Primero describiremos brevemente las decisiones (arbitrarias) sobre la interfaz de usuario para evitar sorpresas. El diseño fue inspirado en las consolas virtuales de *FreeBSD* (o *Linux*), pero hay diferencias importantes.

El sistema posee 12 consolas virtuales, que se eligen con las teclas **F1** a **F12**. Cada una puede tener un proceso adjunto, o estar libre. En el primer caso, será utilizada para la entrada y salida del mismo.

---

nota
Cuando la consola está libre, no se ejecuta un shell ni programa de login. Los procesos se lanzan siempre desde el shell en la primer consola.

Cuando un proceso solicita entrada, el cursor se colocará en la última línea de la pantalla (fuera del área de salida del proceso), en donde aparecerán los caracteres a medida que se pulsan las teclas. Se puede borrar con **BACKSPACE** o terminar con **ENTER**.

Al pulsar **PGUP** o **PGDOWN** la consola entrará en modo *scrolling*, y dejará de desplazar el texto anterior para dejar lugar al nuevo. Pueden pulsarse las teclas mencionadas anteriormente varias veces para observar la historia, y para volver a la última posición hay que usar **ESC** (que también sale del modo *scrolling*).

1.2. Manual

Luego del booteo se ejecuta un shell en la primer consola, que responde a los siguientes comandos.

`help`
: Imprime la lista comandos, detallando lo que hace cada uno.

`ls`
: Imprime la lista de tareas que se pueden ejecutar.

`run <T>`
: Ejecuta la tarea “t_<T>.tsk” en alguna consola libre.

`ps`
: Imprime la lista de procesos del sistema.

`kill <PID>`
: Termina el proceso de *PID* igual al indicado.

`divz`
: Divide por cero, lo que causa una excepción y se cierra (esto es para mostrar que el shell vuelve a lanzarse solo; otra manera es haciendo `kill 1`).

1.3. Organización del código

A continuación se mencionarán los archivos fuente junto con su propósito.

`consola.[ch]`	Consolas virtuales, entrada y salida para procesos.
`gdt.[ch]`	Segmentación básica (todos los segmentos cubren el total de la memoria).
`i386.h`	Funciones *inline* para usar instrucciones del CPU.
`idt.[ch]`	Inicialización de interrupciones y manejo de fallas.

`isr.asm`	Wrappers de interrupciones, y manejo de system calls.
`kbd.[ch]`	Manejo a bajo nivel del teclado (interrupciones y tabla de caracteres).
`klib.[ch]`	System calls y utilidades varias de kernel.
`kmain.c`	Rutina principal, y tabla de system calls.
`loader.[ch]`	Módulo de carga de procesos y scheduler.
`mmu.[ch]`	Mapeo y administración de memoria.
`pak.[ch]`	Interfaz para leer el filesystem dentro del kernel.
`pak_create.cpp`	Programa para crear el filesystem.
`pic.[ch]`	Programación del **PIC**.
`stdlib.[ch]`	Librería standard tanto para kernel como procesos.
`tss.[ch]`	Definicion de la **TSS** y sus campos.

Las siguientes tareas se proveen para mostrar las características del sistema operativo.

fib
: Imprime el elemento de la secuencia de Fibonacci que se pide (por su índice). Usa recursión, por lo que se cerrará al producirse un "stack overflow" dado un valor suficientemente grande.

guess
: Juego que consiste en adivinar un valor elegido al azar entre 0 y *MAX* (elegido por el usuario). Dice si el valor ingresado es mayor o menor al real (en caso de no adivinar). Originalmente era un ejemplo para enseñar cómo funciona la búsqueda binaria.

idle
: La tarea inactiva del sistema, que puede ejecutarse explícitamente en una consola. Aunque por supuesto, no hace nada. El proceso usado por el sistema no aparece en la lista (aunque tiene *PID* 0)y no puede cerrarse.

print
: Imprime todos los numeros que entran en un *int* una y otra vez comenzando desde cero, en bases 2, 10 y 16.

randgen
: Genera *N* números aleatorios, con *N* elegido por el usuario.

shell
: El shell del sistema. [1]

2. Implementación

2.1. Teclado

Utilizamos una tabla para traducir el "scancode" a su valor *ASCII* correspondiente. Como no tenemos en cuenta las extensiones de codificación [2] , nos quedan 128 valores adicionales para identificar teclas especiales como **ESC**, **ENTER**, etc.

También tenemos en cuenta modificadores como **SHIFT** (usando una segunda tabla de códigos) y estados como **CAPSLOCK**. El teclado numérico funciona si **NUMLOCK** está activo.

[1] Si bien puede ejecutarse otro, el único proceso capaz de acceder a las llamadas al sistema restringidas (crear procesos, etc) es el primero.
[2] Como por ejemplo *ISO-8859-1*, el más conocido que contiene acentos.

aviso
Debido a una limitación en el programa no podemos diferenciar las teclas correspondientes a 1, 3, 7 y 9 (del teclado numérico) de sus equivalentes cuando **NUMLOCK** está desactivado (es decir, al presionar **HOME** se ingresa un 7 si está encendido, y 3 hace scrolling si está apagado).

La tecla pulsada se procesa al recibir la interrupción apropiada, pero no se interpreta de ninguna manera sino que pasa al módulo de consola.

2.2. Consola

La consola es uno de los módulos más complejos del sistema operativo, ya que debe interactuar con el teclado y el scheduler (para bloquear las tareas que esperan entrada, y desbloquearlas al recibirla).

Los procesos hacen entrada/salida sin necesidad de saber cuál les fue asignada. Además no tienen permitido elegir la posición a escribir, sino que la consola permite al usuario hacer *scrolling* para ver los mensajes anteriores.

La pantalla se actualiza sólo si hay cambios, y no más de una vez cada 6 ticks (teniendo el reloj configurado en default, equivale aproximadamente a dos veces por segundo).

2.3. Interrupciones

Para las primeras (y obligatorias), se utilizó lo mismo que en *Bran's Kernel Development tutorial*: un handler común para todas, y cada función de atención lo único que hace es llamarla con los parámetros adecuados.

Si la CPU estaba ejecutando en modo usuario, se cierra el proceso y si estaba en el kernel se detiene el sistema. En ambos casos, se muestra un mensaje de error y los valores de los registros.

Las interrupciones de teclado, reloj y de software (system calls) se atienden en las (distintas) rutinas apropiadas.

2.4. System calls

Las realizamos mediante la interrupción de software 80 (en base 10, no 16 como en *Linux*), y una tabla global `syscalls_tabla` en `kmain.c` que indica las funciones disponibles y la cantidad de parámetros de cada una (que se pasan por pila).

La función a ejecutar se indica en el registro **EAX**, que será el índice en la tabla mencionada anteriormente. Los parámetros se copian antes de ejecutar la función indicada por el puntero de la tabla.

Si bien no hay una estructura de privilegios para los procesos de usuario, sólo el shell principal puede ejecutar las llamadas `sys_ls()`, `sys_ps()`, `sys_exec()` y `sys_kill()`.

2.5. Filesystem

Se desarrolló un filesystem de sólo lectura bastante sencillo basado en el formato **PAK** de *idSoftware* (la empresa que hizo el Doom y el Quake, entre otros). La idea es similar al **TAR** de *UNIX* pues concatena varios archivos en uno, pero además contiene un índice al comienzo (lo que facilita las búsquedas).

La única parte del TP hecha en C++ es la utilidad `pak_create`, que genera el archivo para incluírlo en el kernel. No hay una utilidad de lectura o extracción, ya que se lee dentro del sistema operativo.

2.6. Librería standard

En `stdlib.c` hay un conjunto de rutinas que pueden usarse tanto en procesos como en el kernel (incluso algunas de ellas usan llamadas a sistema; esto funciona porque se linkea de distinta forma según si está en una tarea o el kernel).

Además provee un punto de entrada para las tareas, que llama a `main()` y luego realiza `sys_exit()` con el valor retornado por la anterior. Algunas rutinas fueron inspiradas por las existentes en la librería standard de C, a veces con diferencias menores. Una interfaz que vale la pena mencionar es la de `man 3 stdarg`, para leer parámetros adicionales en `printf()`.

2.7. Memoria

Utilizamos dos zonas de memoria, que se asignan y liberan mediante un mapa de bits. Una de ellas se mapea a la misma dirección física y se comparte entre todos los procesos, esta es el área de páginas de kernel a la que sólo pueden acceder en modo supervisor. La otra (de usuario) se utiliza para ubicar datos que sólo sirven para determinados procesos, y pueden mapearse a una dirección física distinta (por ejemplo, todas las tareas creen que están ejecutándose en la misma posición de memoria).

Así funcionan las llamadas al sistema, ya que todos los procesos pueden acceder a los datos del kernel en el mismo lugar dentro de las mismas. Y también acceder a los parámetros (por ejemplo un string) sin necesidad de cambiar el mapa de memoria.

2.8. Scheduler

Es de tipo *Round-Robin* con quantum variable por proceso, y utiliza una tarea especial (oculta en la salida del comando `ps`) para cuando no hay nada para correr (la "idle task").

El bloqueo y desbloqueo de tareas se realiza conjuntamente con el módulo de consola.

3. Limitaciones e ideas

3.1. Relocación de código

Para poder implementar una librería compartida (tarea que intenté llevar a cabo, pero no logré progresar con los scripts de `ld`), habría que hacer que ciertas secciones en el archivo binario resultante esperen ser cargadas en otra posición distinta al ejecutarse.

En ese caso se podrían compartir las páginas de código para todos los procesos que la usen, y también el del mismo programa cargado varias veces.

Recordemos las secciones más comunes del formato *ELF* (que también pertenecían a los formatos anteriores), y su propósito en la compilación de programas en C.

text
: El código del programa, que probablemente hace referencias a alguna posición absoluta de memoria (con esto hay dos alternativas: hacer una tabla indicando las posiciones a corregir según dónde se cargue en la memoria, o establecerlas fijas al momento de linkear). Si se ejecutan en la misma dirección, varios procesos pueden compartir esta sección sin problema (y hasta suele protegerse contra escritura).

data
: Los datos inicializados (variables globales, y las estáticas en funciones) no pueden ser compartidos por varios procesos (a menos que requieran memoria compartida explícitamente, como threads).

rodata
: Son similares a los anteriores, pero se separaron para poder compartirse ya que están protegidos cotra escrituras. Son constantes globales, aunque es probable que el compilador las elimine y reemplace donde haga falta (a menos que su dirección se use en un puntero, u otros casos).

bss
: Los datos no inicializados tampoco pueden ser compartidos, pero a diferencia de *data* y *rodata* sería un desperdicio guardarlos dentro del ejecutable. Por ejemplo, para el caso de un array global que no tiene expresión de inicialización, lo que se hace es: en vez de llenar el espacio en el binario con ceros, se deja una indicación del tamaño requerido y sólo se reserva al cargarlo en la memoria.

3.2. Asignación de memoria a procesos

Si bien sería sencillo asignarle memoria a un proceso de forma contigua (al menos aparentemente), faltaría una rutinas de usuario como `malloc()` y `free()` que operen por bloques de tamaño arbitrario y no sólo páginas.