Trabajo Práctico Especial

May 30, 2011

Objetivo: Realizar un programa que muestre algunas de las características del Modo protegido de los microprocesadores de Intel.

1 Implementacion

En esta sección se explicará el funcionamiento interno del sistema operativo desarollado y porque se eligio tal implementacion.

Para una buena organizacion del codigo, se optó por realizar varios archivos que se encargaran de manejar tareas especificas (un estilo "*objetoso*") y de este modo poder lograr un programa modularizado y por sobre todo implementable.

A continuación se listan los archivos que se han considerado como los mas importantes:

- Interrupt: Cada vez que llega una iterrupcion al micro (ya sea por software o por harware), se ejecuta el handler correspondiente (programado en ASM) que se ocupa de llamar a la funcion handler en C que la maneja. Aqui es donde entra en juego el archivo Interrupt.c. Este contiene la logica necesaria para cada interrupcion.

  Actualmente solo se manejan las interrupciones $80h$, $9(teclado)$, y la $8(timer\ tick)$.

- Keyboard: Luego de leido el scanCode de la tecla presionada (desde el manejador de la interrupcion correspondiente), se "avisa" a este archivo el scanCode y este se encarga de guardar el caracter que corresponda en el buffer del teclado.

  Para el buffer de teclado de decidió implementar un buffer circular, en donde cada vez que llega un caracter, se agrega al a continuacion del anterior (se ignora si se llego o no al final del arreglo). Esto tiene la ventaja que nunca hace falta limpiar el buffer ya que aunque se llene, solo hay que "tomar" las caracteres no leidos del buffer para que automaticamente se haga lugar para las nuevas.

- Video: Funciona como un controlador de video. Es el encargado de administrar la porcion de memoria asigada a memoria. Ya que por cada caracter que se quiera mostrar, hay que decidir en que columna hay que ponerlo, en que fila y con que colores; se decidió que seria muy útil tener este controlador.

  Siempre que por *io* se indica que se quiere escribir un buffer por la *salida estadard*, se llama a este controlador para que se graben adecuadamente.

  Si bien en memoria el sector destinado para la pantalla es contínuo, se implemto todo como si se estuviese trabajando sobre una matriz ($getCurrRow()$ y $getCurrColumn()$ se encargan de realizar las transormaciones matematicas correspondientes).

- Shell: Se encarga de tomar los caracteres del buffer del teclado (siempre que los halla) y decidir si se muestran o no en pantalla. En la actual implementacion, se tiene una shell con un buffer de tamaño fijo en donde se van guardando los caracteres tomados del buffer del teclado. Cuando el usuario indica que quiere ejecutar el comando ingresado, se busca si lo escrito coincide con el nombre de algún comando que tiene guardado internamente en el vector de comandos.

  - Vector de comandos: Es un vector de estructuras en donde cada elemento es una estructua que contiene: nombre de comando, puntero a la funcion que ejecuta dicho comando y un char* que es la ayuda del comando.

  Actualmente la shell es capaz de aceptar argumentos por linea de comandos. Esto fue logrado con relativa facilidad gracias a que se convino que todos los comandos recibirian los mismos argumentos, por lo que, visto desde esta perspectiva, invocar a uno u otro comando con los argumentos tomados es indiferente.

- Commands: Aqui se encuentra la rutina de cada comando mencionado en el punto anterior. Para ver una lista de comandos que se ofrecen basta ejecutar "*help*".

- Kernel: Esta es la parte mas imprtante de la implentacion ya que, siempre que se desea realiza un *read* o un *write*, es realizado siempre a traves de este.

  Tal como sugeria el archivo *kernel.h* suministrado por la cetdra, se ofrecen dos metodos, $\_\_read$ y $\_\_write$. Uno de los parametros requeridos, es el *file descriptor* que indica a donde que quiere acceder. Actualmente se tienen implementados *tres* tipos de *file descriptors*:

  - Si se quiere escribir en *pantalla*, se realiza un $\_\_write$ con parámetro STD_OUT o bien STD_ERR.

– Si se quiere leer del *teclado*, se realiza un $\_\_read$ con parámetro STD_IN.

Cada vez que alguna de estas funciones es invocada, se ejectua un $\_SystemCall$ (llamda a ASM) quien se ocupa de invocar a la interupcion $int80h$ y esta es majedada (como ya se menciono) por el $int80$ *handler*.

2 Manual de uso

Para ver cuales son los comandos disponibles, basta con ejecutar el comando *help*.

Para ejecutar cualquier comando valido, basta escribir "*nombreComando argumentos separados por espacios*" por ej, si se invoca al comando *help random*, se desplegara un menu con una breve explicacion de como funciona el comando random.

2.1 Comandos actualmente disponibles:

- help
- echo
- restart
- clear
- getCPUspeed
- countDown
- setPit
- resetPit
- random
- setAppareance

Observaciones:

- Notar que se considera a cada espacio como un nuevo argumento. Por lo que la invoacion a una funcion con varios espacios en entre los argumentos, es lo mismo que mandar un argumento "" (char * apuntando a un '\0') en el medio.
- Si se colocan espacios al principio de cada comando, el resultado final no debe esperarse igual a que poner el comando sin los espacios. Es decir: " echo" no ejecutará el comando llamado "echo" (sin espacio al principio)

3 CPU Speed

Para el calculo de la velocidad de la CPU se decidió calcular cuantas veces se llegaba a incrementar una variable entre dos timerTicks. Es decir, calculamos la cantidad de *iteracciones* que la CPU es capaz de realizar en este intervalo tiempo. Sin embargo, como este valor puede llegar a variar en cada mdicion ya que la CPU puede estar mas o menos ocupada, se decidió que lo mejor era realizar N mediciones y luego calcular el promedio de las mismas.