Arquitecturas de Computadoras

TP Especial

Mini Sistema operativo

HAL 9000

Integrantes

MAGNORSKY, Alejandro
MATA SUÁREZ, Andrés
MERCHANTE, Mariano

Cátedra

VALLÉS, Santiago Raúl
PUNCI, Román

Fecha de entrega

Viernes 18 de junio de 2010

Índice

1. Excepciones implementadas

Overflow exception:

en `mkexec.c`:

```
. . .
int mkexc(int argc, char * argv[]){
        . . .
                case 4:
                __asm__("movb   $127, %dl\n\t"
                        "addb   $127, %dl\n\t"
                        "into");
                break;
        . . .
}
. . .
```

Cuando se ejecutan operaciones con operandos con signo, el flag `OF` es seteado cuando se detecta un overflow (el tamaño del resultado no entra en el registro indicado). La instrucción `into` verifica si el flag `OF` está activado, en cuyo caso dispara la excepción.

Bounds Check exception:

en `mkexec.c`:

```
. . .
int mkexc(int argc, char * argv[]){
        . . .
                case 5:
                __asm__("movl   $0, -8(%ebp)\n\t"
                        "movl   $2, -4(%ebp)\n\t"
                        "movl   $3, %eax\n\t"
                        "bound  %eax, -8(%ebp)");
                break;
        . . .
}
. . .
```

La instrucción `bound` sirve para verificar si un índice no supera ciertos límites. Si no se cumple, se dispara la excepción.
En el código de generación de la excepción descripto más arriba el registro `eax` estaría haciendo las veces de índice, y `[ebp-8]` y `[ebp-4]` cumplirían los roles de límite inferior y superior, respectivamente.

Luego de invocar a la rutina que imprime en pantalla el mensaje "Bounds check", en `libasm.asm`, se vuelve a setear el registro `eax` de manera que no se vuelva a disparar la excepción.

en `libasm.asm`:

```
. . .
        call    int_05
        mov     eax, 1
. . .
```

2. Inconvenientes surgidos

Dada la imposibilidad del uso de las librerías de C oficiales, en determinadas situaciones nos vimos obligados a ajustar el diseño de nuestro sistema de acuerdo con la implementación de nuestra propia libreria de C. Tales percances y sus resoluciones son expuestos a continuación:

Alocación en memoria

Quizás este fue el problema con el que más frecuentemente tuvimos que lidiar. En otras circunstancias, el uso de la función de reserva de espacio en memoria `malloc` de C nos hubiera permitido no muy dificilmente solucionar la cuestión. Sin embargo, al no tener lal bases suficientes como para implementar una función similar a la mencionada en nuestra librería, en los casos que así lo dispusieron optamos por reservar cierta cantidad fija en memoria y sacrificar la manipulación dinámica de ese espacio, mediante la declaración de vectores.

Máxima cantidad de caracteres ingresados por el usuario

La función `scanf` de `lib.c`, en un principio, solicita al usuario ingrese una string, invocando a la función `getString` de la misma librería. Esta última función recibe un parámetro de tipo `char *` donde se guardará cada caracter leído con `getchar`, dejando un último lugar para el caracter nulo (`'\0'`). No devuelve nada; el parámetro recibido es también de retorno.

Ahora bien, es tarea de la función invocadora (en este caso, `scanf`) pasarle a `getString` un `char *` con suficiente espacio reservado para que la función se encargue de rellenarlo. Si el parámetro no tiene suficiente espacio para guardar todo lo introducido por el usuario, se producirá un error de segmentación en memoria y el sistema se reiniciará.

Y es aquí donde se toma la decisión. Con las constantes globales, `MAX_ARGUMENTS` y `MAX_ARGUMENT_LENGTH`, se limita la cantidad de argumentos y caracteres por argumento que puede introducir el usuario, respectivamente:

en `shell.h`:

```
. . .
#define MAX_ARGUMENT_LENGTH 64
#define MAX_ARGUMENTS 10
. . .
```

Dicho esto, dentro de `scanf` y antes de la llamada a `getString`, se declara la variable que posteriormente será utilizada como el parámetro de retorno que esta última función requiere de la siguiente manera:

en `libc.c`:

```
. . .
#define MAX_USER_LENGTH MAX_ARGUMENTS*MAX_ARGUMENT_LENGTH
. . .
int scanf(const char * str, ...){
    . . .
    char strIn[MAX_USER_LENGTH+1];
    . . .
}
. . .
```

Así, se reserva el suficiente espacio para que el usuario pueda meter `MAX_USER_LENGTH` caracteres, más uno para hacer la string `null-terminated`.

Máxima cantidad de comandos en el historial, de terminales y de caracteres en el símbolo de sistema

De la misma manera que ocurre con el ítem anterior, se vuelve necesario definir una constante que limite la cantidad máxima de argumentos que se van a tener en cuenta en el historial. La constante `MAX_HISTORY` en `shell.h` se encarga de esta definición.

en `shell.h`:

```
. . .
#define MAX_HISTORY 20
. . .
```

en `console.h`:

Para el máximo de terminales:

```
. . .
#define __MAX_TERMINALS 10
. . .
```

Para el máximo de caracteres en el símbolo de sistema:

```
. . .
#define __MAX_SS 15
. . .
```

Scancodes

Teclas '`Pausa|Inter`', 'F11'y 'F12'

El scancode de la tecla '`Pausa|Inter`'comprende 4 bytes de `make code` y 4 de `break code`. Debido a eso, se hacía muy engorroso extender la condición para levantarla en `getAscii`, de la librería de teclado `keyboard.c`, ya que uno de sus `make codes` coincide con el de

`bloqNum`, entonces ambas teclas activarán el `numPad`. A su vez, las teclas 'F11'y 'F12'tienen un scancode que no fue considerado y preferimos hacer algo más productivo.

Exceptions

Zero Divide Exception

En un primer momento, intentamos implementar una función que genere una `Zero Divide Exception` a modo de testeo. Sin embargo, nos encontramos con que el sistema se sumergía en un loop infinito y la única manera de seguir, era reiniciando la máquina.

Esto se debía a que esta excepción, una vez ejecutada su rutina de atención, vuelve a ejecutar la misma instrucción que la generó (el `eip` no se incrementa al ser pusheado en el stack), entonces, esa instrucción volvía a generar la misma excepción, y así sucesivamente.
Por lo tanto decidimos programar testeos para las excepciones `Overflow` y `Bounds Check`.

3. Bibliografía consultada

- **OSDev.org**
  http://wiki.osdev.org/Main_Page

- **Programación del PIC**
  http://www.beyondlogic.org/interrupts/interupt.htm#6
  http://www.osdever.net/tutorials/pdf/pic.pdf

- **ASCIIs y ScanCodes**
  http://www.win.tue.nl/ aeb/linux/kbd/scancodes-1.html
  http://www.cdrummond.qc.ca/cegep/informat/Professeurs/Alain/files/ascii.htm
  http://www.ee.bgu.ac.il/ microlab/MicroLab/Labs/ScanCodes.htm

- **Manejo de excepciones**
  http://pdos.csail.mit.edu/6.828/2008/readings/i386/s09_08.htm
  http://internals.com/articles/protmode/interrupts.htm

- **Command bytes del teclado**
  Command bytes
  http://www.arl.wustl.edu/ lockwood/class/cs306/books/artofasm/Chapter_20/CH20-2.html
  LEDs http://wiki.osdev.org/PS2_Keyboard#Keyboard_LEDs
  Reboot http://wiki.osdev.org/Reboot

- **CPUID** http://forum.osdev.org/viewtopic.php?t=11998

- **VGA**
  VGA Texto http://www.osdever.net/FreeVGA/vga/vgatext.htm
  VGA General http://www.osdever.net/FreeVGA/vga/vga.htm

4. Código fuente