SISTEMAS OPERATIVOS

Ensamblador en LINUX

Elaborado por: Ukranio Coronilla

Uso del depurador gdb

En Linux existe una herramienta para depurar programas conocida como gdb. Con gdb es posible ver la ejecución de un programa instrucción por instrucción, de este modo podemos observar cómo cambian los valores de las variables o los valores de los registros dentro del CPU. Así podemos detectar errores de programación o simplemente tener una mejor comprensión de cómo opera el código.

Para utilizar gdb no solo es necesario tener el código ejecutable, se requiere cierta información adicional relacionada con direcciones de memoria, variables, tipos de datos, funciones etc… Para incluir esta información es necesario utilizar la opción -g a la hora de la compilación.

Salve el siguiente programa como `ejercicio2_1.c`, compile y ejecute (Para detener el programa pausado presione CTRL-C). Este programa no funciona como debería, ¿Cuál cree que es el valor correcto que debería imprimir la variable `resultado`?
Supongamos que no podemos entender dónde está el error y acudimos al depurador. Para ello es necesario compilar de nuevo, pero ahora añadimos la opción -g antes de la opción
-o. Observe que esta opción incrementa el tamaño del archivo ejecutable.

```
#include <stdio.h>
int main(void){
   int primero;
   int intermedio;
   int resultado;

   printf("Hola gdb\n");
   intermedio = primero + 4 * 5 - 2;
   resultado = intermedio * 4;
   printf("%d\n",resultado);
   pause();
}
```

El programa depurador se llama gdb y recibe como parámetro el nombre del programa ejecutable, en este caso:

```
gdb ejercicio2_1
```

A continuación veremos los comandos más utilizados con gdb. Vaya haciendo uso de todos y cada uno de ellos hasta que quede completamente clara su funcionalidad, pregunte al profesor cualquier duda.

El depurador gdb va a imprimir alguna información y posteriormente podremos ejecutar comandos de gdb. Pruebe a usar repetidamente el comando l el cual nos devuelve un listado del código con sus correspondientes números de línea.

Se puede obtener ayuda sobre cualquier comando interno a gdb con help, por ejemplo para el comando l se escribe:

```
(gdb) help l
```

Para iniciar el proceso de depuración es necesario ubicar las líneas de código donde la ejecución va a detenerse, estas ubicaciones se conocen como breackpoints. Para poner un breackpoint en la línea 9 del código fuente escribimos:

```
(gdb) b 9
```

Aparece entonces un identificador de la interrupción y la dirección de memoria donde se ubica dicha instrucción, en mi caso apareció:

```
Punto de interrupción 1 at 0x804844d: file ejercicio2_1.c, line 9.
```

Se puede ver una lista de los breackpoints con:

```
(gdb) info break
```

Se puede borrar un breackpoint con:

```
(gdb) del <número del breakpoint>
```

Teniendo al menos un breackpoint podemos correr el programa con:

```
(gdb) run
```

Posteriormente podemos ejecutar paso a paso instrucción por instrucción de C con:

```
(gdb) step
```

O paso a paso de instrucciones en lenguaje maquina (más adecuado para programas en ensamblador debido a que una instrucción en lenguaje ensamblador le corresponde solo una instrucción en lenguaje máquina):

```
(gdb) stepi
```

Se puede depurar nuevamente desde el inicio con:

```
(gdb) start
```

Para ver los valores de las variables correspondientes a la función en la que se encuentra ejecutando en ese momento utilizamos:

```
(gdb) info locals
```

Aunque también puedo imprimir el contenido de cualquier variable:

```
(gdb) p primero
```

O su dirección de memoria (en este caso se trata de una dirección virtual):

```
(gdb) p &primero
```

Se pueden desplegar 5 instrucciones a partir de una dirección de memoria específica con:
```
(gdb) x/5i 0x804844d
```

O se puede desplegar el contenido de la memoria a partir de dicha dirección:
```
(gdb) x/10xb 0x804844d
```

En este caso la letra b después de 10x es para especificar que se muestre byte por byte pero podemos utilizar h para 2 bytes, w para 4 bytes y g para 8 bytes.
Se pueden ver los valores almacenados en los registros del procesador mediante:
```
(gdb) info register
```

Uno de los registros más importantes es el apuntador de instrucción o contador de programa el cual almacena la dirección de memoria de la siguiente instrucción que va a ejecutar el CPU. (Véase https://es.wikipedia.org/wiki/Contador_de_programa).

Para salir de gdb
```
(gdb) quit
```

Se tiene una gran cantidad de herramientas de depuración para lo cual es recomendable ejecutar help junto con la clase de comandos a los que se desea acceder.
```
(gdb) help running
```

Ejercicio 1
Analice con gdb el siguiente código y entienda de manera exacta cómo funciona la conversión de un número decimal a binario:

```
#include <stdio.h>
int main(void)
{
   unsigned char var = 42;
   unsigned int contador, inicio = 128; // 2^(8-1) = 128
   printf("En decimal el valor %d equivale en binario a:\n", var);
   for(contador = inicio; contador > 0; contador >>= 1)
      if(contador & var)
         printf("1");
      else
         printf("0");
   printf("\n");
   return 0;
}
```

Principios de lenguaje ensamblador en LINUX

Veremos a continuación como se crea un programa ejecutable a partir de código en lenguaje ensamblador, así como algunas características del lenguaje. Cabe destacar que cada arquitectura de CPU tiene su propio lenguaje de máquina. En este caso estudiaremos solo la arquitectura Intel en modo de 32 bits con uso de memoria virtual y páginas de 4 kilobytes.

Todos los programas de ensamblador en LINUX mantienen el siguiente cuerpo de instrucciones (exceptuando la línea que inicia con el signo ";", el cual se utiliza para comentarios):

```
section .data
section .bss
section .text

     global _start

_start:
     nop
; Aquí se introduce el código fuente...
     nop
section .bss
```

En Linux abra su editor de texto y teclee el programa 2.2 al cual llamaremos `prog2_2.asm` (Todos los programas en ensamblador tienen la extensión .asm).

```
section .data
flda        dd    250
fldb        dd    1fH
fldc        dd    0
D1          db    0
D2          dw    1000
D3          db    110101b
D4          db    12h
D5          db    17o
D6          dd    1A92h
D7          dd    2011
D8          db    "A"
D9          db    0, 1, 2, 3
D10         db    "H", "O", "L", 'A', 0
D11         db    'hola', 0
D12         times 32 db 16

section .text
      global _start

_start:
   mov ebx, 0x69abcdef
   mov eax, [flda]
   add eax, [fldb]
   mov [fldc], eax
   mov eax,1 ; Regresar al Sistema Operativo
   mov ebx,0
   int 0x80
```

Para compilar utilizamos el programa nasm como sigue:

```
nasm -f elf -g -F stabs prog2_2.asm
```

-f elf: especifica que el archivo objeto se va a generar en el formato elf (Executable and Linkable Format).

-g : Especifica que se va a incluir información de depuración en el archivo objeto.

-F stabs: genera un formato stabs de información para depuración.

En el caso de que el sistema operativo sea de 64 bits, debe compilarse con:

```
nasm -f elf64 -g -F stabs prog2_2.asm
```

El compilador nasm crea el programa objeto:

```
prog2_2.o
```

Posteriormente se enlaza con el programa ld:

```
ld prog2_2.o -o prog2_2
```

Obteniéndose finalmente el ejecutable `prog2_2`

Ahora si ejecuta este programa en la línea de comandos, observará que no se imprime nada en pantalla, sin embargo sí se estarán ejecutando las instrucciones del código.

En este programa las últimas tres líneas son necesarias para terminar correctamente el programa.

Ejercicio 2

Dentro de la sección de datos se han definido tres variables, del mismo tipo (dd). Utilizando gdb determine cuantos bytes ocupa el tipo de dato dd (vea la dirección donde inicia cada variable y tenga en consideración que cada variable se almacena en direcciones consecutivas de memoria). Posteriormente despliegue el contenido de la memoria a partir de la dirección donde se encuentra ubicada la variable flda (use en gdb x/16xb), y cerciórese de que los valores almacenados corresponden con los valores con los que se inicializaron las variables. Verifique como están almacenados también los demás datos `D1` a `D12`. Recuerde poner un breackpoint por lo menos en la primera línea de código.

Dentro de la CPU existen varios registros, podemos visualizar su contenido en un momento dado mediante el comando de gdb “info register”. Existen cuatro registros de propósito general EAX, ECX, EDX, y EBX, todos ellos con tamaño de 4 bytes. También se puede hacer referencia al byte menos significativo de EAX como AL, al siguiente byte adyacente a AL como AH, y a los dos bytes menos significativos como AX (en arquitecturas de 64 bits en lugar de EAX se llama RAX).

La instrucción MOV transfiere los datos referenciados por la dirección del segundo operando a la dirección del primer operando. Es importante que ambos operandos sean del mismo tamaño. Por ejemplo la primera instrucción:

```
mov ebx, 0x69abcdef
```

Mueve el valor hexadecimal 69abcdef al registro del CPU EBX.

Ejercicio 3

Intente compilar el programa cuando se intenta mover un valor mayor de 4 bytes al registro ebx. ¿Qué sucede? ¿Por qué? Si alguien del equipo tiene un sistema operativo de 64 bits cambien la instrucción anterior por:

```
mov rbx, 0x69abcdeffff
```

¿Cuál es el tamaño máximo en bytes que se puede mover al registro rbx? Este tamaño en bits indica la capacidad del CPU ¡

Compruebe que el valor se mueve al registro del cpu mediante el uso alternado de las instrucciones “info register” y “stepi”.

Ejercicio 4

Con ayuda de gdb describa que operaciones realizan las primeras cuatro instrucciones de lenguaje ensamblador en el programa 2.2. ¿Para qué sirven los corchetes? Determine también con ayuda de gdb que realizan las siguientes instrucciones:

```
mov ecx, ebx
sub ecx, 10
inc ecx
dec ecx
mov al, [D4]
mov eax, [D4]
mov [D1], ch
mov al, [D6]
```

La siguiente instrucción falla:

```
mov [D6], 96
```

Debido a que el ensamblador no sabe si almacenar el 96 como byte (byte), palabra (word) o palabra doble (dword). La manera correcta es:

```
mov dword [D6], 96
```

Como en este ejemplo, elabore una instrucción que ocupe byte y otra word para almacenar en la variable D6.

El registro eip (extended instruction pointer) almacena la dirección de inicio de la siguiente instrucción a ejecutarse. Observe como cambia su valor con cada línea de código ejecutada