Alberto Jesús Durán López – Estructura de computadores

Práctica 2: Preguntas de Autocomprobación

suma_01_S_cdec1

1)Realizamos un volcado de la pila. Usamos el comando x/8xw $esp y obtenemos:

```
(gdb) x/8xw $esp
0xffffd33c: 0x00000000  0x00000000  0x08048084  0x080490b2
0xffffd34c: 0x00000009  0x00000001  0xffffd4db  0x00000000
```

-NULL
-argv[0] = pname que es 0xffffd4db
-argc=1, por lo que tenemos 0x00000001
-0x00000009 es el contenido de longlista
-0x080490b2 es la dirección de lista
-0x08048084 es la dirección de retorno
-0x00000000 es el antiguo contenido de $esp
-0x00000000 es el antiguo contenido de $ebx

2)
-cuantos: 9
-qué: 0x080490b2

Ejecutamos en gdb:

```
(gdb) x /9xw 0x080490b2
0x80490b2: 0x00000001 0x00000002 0x0000000a 0x00000001
0x80490c2: 0x00000002 0x00000002 0x00000001 0x00000002
0x80490d2: 0x00000010
```

3)
El registro %ebx es salva-invocado (la función debe preservar su valor de tal forma que al final de la ejecución sea igual que al principio.

El registro %edx es salva-invocante (la función puede utilizarlos)

4)
-Registro base: %ebx
-Registro índice: %ebx
-Factor de escala: 4, sizeof(int)

5)
si, con el siguiente código:

```
bucle:
    add (%ebx, %edx), %eax
    add $4, %edx
    dec %ecx
    jne bucle
```

6)

```
suma:
    push %ebp
    mov %esp, %ebp

    push %ebx
    mov 8(%ebp), %edx
    mov 12(%ebp), %ecx
    mov $0, %eax
bucle:
    add (%edx), %eax
    add $4, %edx
    dec %ecx
    jne bucle

    pop %ebx
    pop %ebp
    ret
```

7)

-jnz, cuando no sea 0
-ja, cuando sea mayor que 0
-jnbe, cuando no sea menor o igual que 0

8)

```
suma:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    push %ebx
    mov 8(%ebp), %edx
    mov 12(%ebp), %ecx
    mov $0, %eax
bucle:
    add (%edx), %eax
    add $4, %edx
    dec %ecx
    ja bucle

    pop %ebx
    add $4, %esp
    pop %ebp
    ret
```

Suma_02_S_libC

1)

```
suma_02_S_libC.o: En la función ‘_start‘:
/home/alberto/EC/Practica2/suma_02_S_libC.s:26: referencia a ‘printf‘ s
/home/alberto/EC/Practica2/suma_02_S_libC.s:30: referencia a ‘exit‘ sin
```

2)

Error de ejecución:

Failed to execute process ’./suma_02_S_libC’. Reason:
The file ’./suma_02_S_libC’ does not exist or could not be executed.

3)

En la línea 18:
-En modo gráfico, Examine 5 hex words desde $esp
-En gdb, x/5xw $esp

5)

Para el volcado tenemos que hacer, en la línea 26:
-En modo gráfico, Examine 6 hex words desde $esp
-En gdb, x/6xw $esp

7)

Para el volcado tenemos que hacer, en la línea 30:
-En modo gráfico, Examine 4 hex words desde $esp
-En gdb, x/4xw $esp

suma_03_SC

1)
Ejecutamos el comando info files:

(gdb) info address _start
Symbol "_start" is at 0x8048207 in a file compiled without debugging.

Si no supieramos el punto de entrada no seríamos capaces de depurar en ensamblador.

2)
-next avanza la instrucción sin meterse en las subrutinas
-step avanza la instrucción introduciéndose en las subrutinas
-stepi avanza instrucción del código máquina introduciéndose en las subrutinas.
-next avanza instrucción sin introducirse en las subrutinas.

Call suma:
- next ejecuta toda la subrutina suma.
- step nos introduce en la función suma en C

(gdb) step
suma (lista=0x804933c, len=9) at suma_03_SC_c.c:1

int suma(int* lista, int len)

-stepi y next devuelven el mismo resultado que step y next

4)

Podemos obtener el código ensamblador a partir del módulo objeto con objdump:

```
0: mov  4(%esp),%ecx  # %ecx: lista
4: mov  8(%esp),%eax  # %eax: longlista
8: test  %eax,%eax        # Comprueba longlista > 0
a: jle  21 <suma+0x21>
c: mov  %ecx,%edx
e: lea  (%ecx,%eax,4),%ecx # %ecx: lista + 4*len
11: mov  $0x0,%eax     # %eax: 0

## Bucle principal
16: add  (%edx),%eax
18: add  $0x4,%edx       # lista++
1b: cmp  %ecx,%edx     # Comprueba que no hemos llegado al final
1d: jne  16 <suma+0x16> # Suma el siguiente
1f: repz ret
21: mov  $0x0,%eax      # Caso longlista = 0
26: ret
```

5)
Significa que la variable res no tiene código ensamblador.

Aparece en la tercera línea:
for(int i=0; i<len; i++)

6)
Las direcciones de las variables serían:

```
(gdb) print &res
$1 = (int *) 0xffffd348
(gdb) print &i
$2 = (int *) 0xffffd34c
```

suma_04_SC

1)

```
8048200:sub  $0xc,%esp # Reserva 3 variables locales
8048203:mov  0x8049388,%ecx # Mueve longlista a ecx
8048209:test  %ecx,%ecx # Si longlista es 0, salta al final
804820b:jle  8048227 <suma+0x27>
804820d:mov  $0x0,%edx # res
8048212:mov  $0x0,%eax # i

# Bucle principal
8048217:add  0x8049364(,%eax,4),%edx # Suma lista[i]
804821e:add  $0x1,%eax
8048221:cmp  %ecx,%eax # Si i != longlista, repite
8048223:jne  8048217 <suma+0x17>
8048225:jmp  804822c <suma+0x2c>
8048227:mov  $0x0,%edx
804822c:mov  %edx,0x804938c

# Printf y exit
8048232:push  %edx
8048233:push  %edx
8048234:push  $0x8049390
8048239:push  $0x1
804823b:call  80481f0 <__printf_chk@plt>
8048240:movl  $0x0,(%esp)
8048247:call  80481e0 <exit@plt>
```

2)
Si la lista tiene longitud 0 la version de suma.s puede fallar.

3)
Se reservan 3 variables locales y se llama a _ _printf _ clk:

```
080481f0 <__printf_chk@plt>:
80481f0:  ff 25 60 93 04 08    jmp  *0x8049360
80481f6:  68 08 00 00 00       push  $0x8
80481fb:  e9 d0 ff ff ff       jmp  80481d0 <exit@plt-0x10>
```

5)
Los símbolos sin definir son las llamadas al sistema **exit y printf.**
Dependen de librerías dinámicas y no sabemos donde se localizan hasta que no las ejecutemos.

7)
Cambian ya que el ejecutable se genra gracias al enlazador **ld** al que le pasamos la opción -lc -dynamic-linker /lib/ld-linux.so.2

8)
Realizando objdump sobre los ficheros objetos, observamos que las direcciones de memoria desconocidas aparecen como 0x0 en el código ensamblador
En el ejemplo de suma_04:

```
3:mov    0x0,%ecx #longlista, desconocido
  ...
d:mov    $0x0,%edx
12:mov  $0x0,%eax
17:add   0x0(,%eax,4),%edx #lista, desconocido
  …
```

suma_05_C

1)
-next: avanza por el main sin introducirse en funciones.
-step: se introduce en la función suma y printf.
-next: recorre el main de la versión en ensamblador.
-stepi: se introduce en la función suma y printf y avanza por sus instrucciones en ensamblador.

2)
Tras hacer **next** el marco de pila queda:

```
(gdb) x /8xw $esp
0xffffd854: 0x08048560  0x00000025  0x00000025  0xf7fb641c
0xffffd864:  0xffffd880    0x00000000  0xf7e1a71e   0x00000000
```

El valor del resultado es 0x00000025
La dirección de libc es 0xf7e1a71e

3)
```
(gdb) x /2xw $esp
0xffffd2a8: 0x0804a060  0x00000009
```

– 0x0804a060 es lista.
– 0x00000009 es el número de elementos de la lista

suma_07_Casm

1)
El main es exactamente igual. Pero en el código de **suma**, podemos ver que en **suma_07** el contenido de la sentencia **asm** que escribimos en C posee comentarios que indican el origen del código.

2)
Se usan los registros %eax, %ebx, %ecx y %edx.
-El registro %ebx se guarda en la pila y se restaura por lo que no hay problema si se modifica.
-El resto de registros se pueden usar para cualquier función, que se encarga de restaurar sus valores si llama a otra función.

3)
Cuando se usa \t, las líneas aparecen desplazadas por lo que es preferible no usarlo

4)
Se sabe ya que el código introducido se copia de forma literal en el código ensamblador y en que sólo se añade un salto de linea al final.

suma_08_Casm

1)
```
suma:
.LFB38
        pushl  %ebp
        movl   %esp, %ebp
        pushl  %ebx
        movl  8(%ebp), %ebx
        movl  12(%ebp), %ecx
        testl  %ecx, %ecx
        jle .L4
        movl  $0, %eax
        movl  $0, %edx
.L3:
#APP
# 13 "suma_08_Casm.c" 1
        add (%ebx,%edx,4),%eax
# 0 "" 2
#NO_APP
        addl  $1, %edx
        cmpl  %edx, %ecx
        jne .L3
        jmp .L2
.L4:
        movl  $0, %eax
.L2:
        popl  %ebx
        popl  %ebp
        ret
```

Como podemos comprobar, **suma_07** comprueba el caso en el que **longlista**=0. Además hace uso de la instrucción **addl $1, %edx** en lugar de usar inc

2)
Tenemos que compilar con las opciones -fno-omit-frame-pointer -m32 -01 -S en gcc
Con la herramienta **diff** vemos que los archivos suma_08.s y suma_08cc.s son identicos.

3)
Porque todos los registros que utilizamos los añade gcc en compilación.

4)
Necesitamos indicar que %eax es un registro de entrada salida para que efectue la operación:
mov1 $0, %eax

5)
En este caso, se prefiere acabar con \n por si se añaden nuevas instrucciones, ya que si se pone \n\t al añadir una instrucción nueva esta quedaría desalineada del resto; aunque cabe destacar que no deja de ser una preferencia estética

suma_09_Casm

1)
El código de **suma2** es igual al de **suma_08** mientras que el código de **suma3** es idéntico al de **suma_07.** Además, el código de **suma1** es igual al código de **suma_05.**

2)
El código generado es idéntico. Pero cabe destacar que al declarar el registro %ebx como clobber, el compilador gestiona el guardado del valor de este registro en la pila, así como su restauración, ahorrando código.

3)
Los escribimos con dos % para indicar a gcc que nos referimos a registros y no a los argumentos que esfecificamos en la lista de entradas y salidas.
Al especificar el nuevo clobber necesitamos utilizar los dos %% para evitar que gcc confunda los no,bre de varibles con los de registros.

4)
El array tiene 1 << 16 elementos, es decir, 2^16 = 65536 elementos.
Cada entero ocupa 4 bytes, es decir que en total el array ocupará 65536 $*$ 4 =262 kB.
La suma vale 2147450880. Se trata de una sucesión aritmética.

5)
Ambas fórmulas son correctas ya que en este caso SIZE=N+1.

6)
Ese comentario puede indicar que se produce un overflow.
La fórmula está escrita de forma que se evita realizar el producto SIZE*(SIZE-1). Esto se debe a que este producto puede producir overflow.
Para evitar esto podríamos haber usado long unsigned y printf podría escribirse como:
printf("N*(N+1)/2 = %lu\n", ((SIZE-1)*SIZE)/2);

7)
El primer tipo del argumento es un puntero a una función que devuelve un int.
El formato para imprimir el segundo argumento es %s: %91d us, es decir, se imprime la string y luego el tiempo con un espacio de al menos 9 caracteres.
La función gettimeofday modifica el struct apuntado por el puntero que le pasamos para añadir el tiempo.
El formato acaba en \t para introducir un espacio. Se puede utilizar %91d para imprimir un long.

Popcount.c

1)
Resolviendo la ecuación: 2^31 -1 = 32*n → n= 2^26
Si fuera unsigned tenemos: 2^32-1 = 32*n → n=2^27

3)
Se hace así para que en el caso de que hubiera bits de signo no se dupliquen al hacer el desplazamiento hacia la derecha. Si lo declaramos como int se podrían duplicar bits en el caso de que hubiera un número negativo.
No pasaría ya que únicamente trabajamos con positivos. Para notar la diferencia basta con poner algun bit negativo.

5)
Porque son arrays cuyo contenido no cabe en un sólo registro.

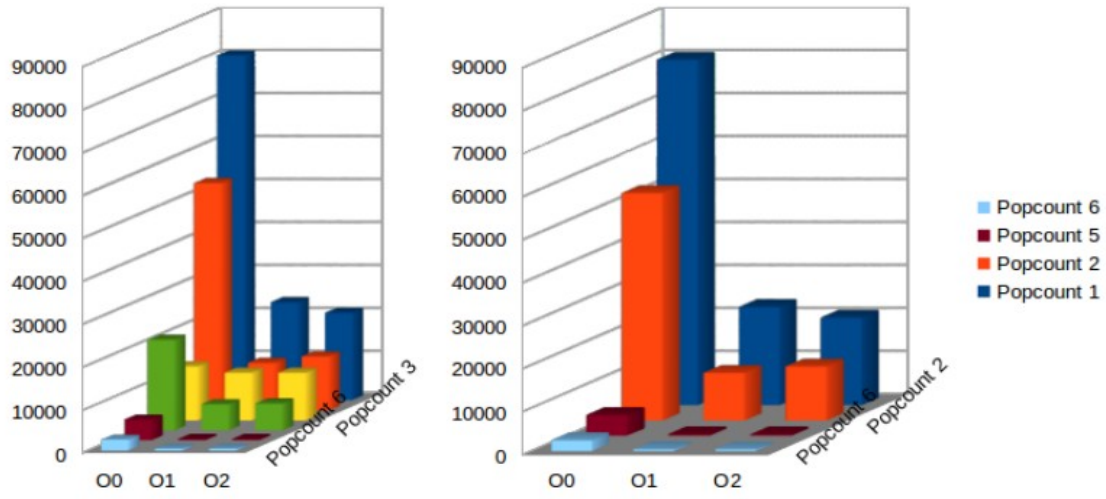
7)
Código de la versión 3:

```
.L24:
      movl  (%edx), %ecx
      addl  $4, %edx
bpcnt3:
      shrl %ecx
      adc $0, %eax
      cmp $0, %ecx
      jnz bpcnt3
.L13:
      movl  %edx, %ecx
      andl  $1, %ecx
      addl  %ecx, %eax
      shrl  %edx
      jne .L13
      cmpl  %ebx, %edx
      jne .L24
```

Y el generado para la versión 2:

```
.L14:
      movl  (%ebx), %edx
      testl  %edx, %edx
      je .L12
      .p2align 4,,10
      .p2align 3
.L13:
       movl  %edx, %ecx
       andl  $1, %ecx
       addl  %ecx, %eax
       shrl  %edx
       jne .L13
.L12:
       addl  $4, %ebx
       cmpl  %ebx, %esi
       jne .L14
```

8)
Gráficas:



Lo mejor ha sido mejorar la programación en C aunque asm también ha dado buenos resultados.

parity.c

1)
Para SIZE = 2^n elementos la fórmula se reduce a SIZE /2 ya que podemos ver que:
- Para los dos primeros números habrá exactamente 1 impar, para los dos segundos también.
- Siguiendo con este patrón, para los 4 números siguientes habrá exactamente 2 impares, para los 4 siguientes también. Podemos extender este patrón en general.
De esta forma podríamos agrupar en grupos de 2 m elementos con exactamente 2^(m−1) impares,

2)
Por si nos encontramos el caso en el que hubiera bits de signo estos no se dupliquen al hacer desplazamientos hacia la derecha. Que se podrían duplicar bits caso de que hibiera un número negativo cambiando la paridad.
No, ya que todos los números con los que trabajamos son positivos. Habría que poner un bit negativo.

3)
Hay una pequeña mejora para todos los niveles de optimización, siendo el más relevante cuando no se aplica ninguna optimización y una mejora menor en el resto de los casos.
En la tercera versión no hay código ASM así que no se pueden hacer restricciones a memoria.

4)
La cuerda versión tarde menos que la tercera versión.

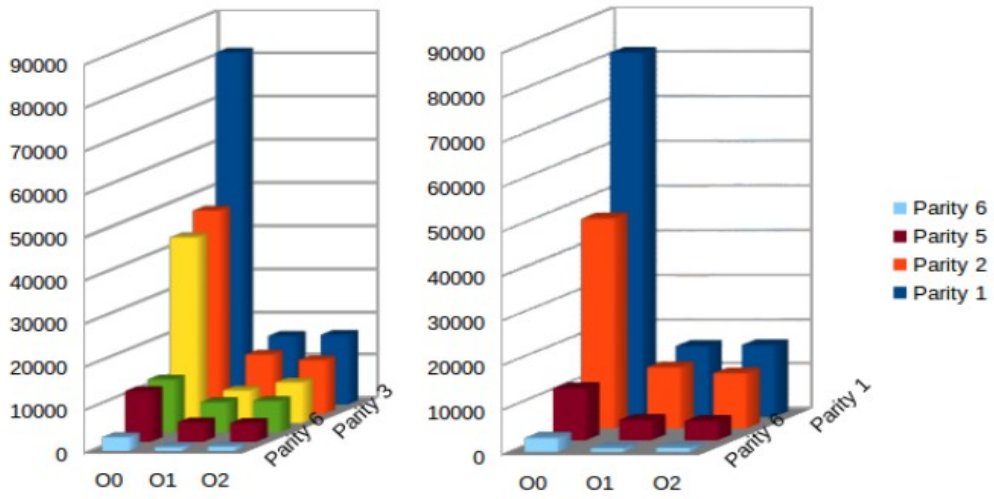
7)
El código final sería:

```
mov     %[x],   %%edx
shr     $16,    %%edx
xor     %[x],   %%edx
xor     %%dh, %%dl
setpo   %%dl
movzx  %%dl,  %[x]
```

Solo hay un desplazamiento, ya que aprovechamos el acceso a los LSB y MSB de edx para realizar el segundo desplazamiento.
El código es el más eficiente. g**cc** no es capaz de generar este código

8)
Con la opción -01 el resultado es 0. Esto se debe a que al no indicar edx como clobber se toma como registro para x obteniendo como reultado un código sin sentido.

9) Gráficas



Mejorando la programación en C se obtienen mejores resultados, aunque el uso de asm no ha dadado malos resultados.