Plancha 3: Ensamblador de x86_64

2018 – Arquitectura de las Computadoras
Licenciatura en Ciencias de la Computación
Entrega: fecha a determinar

Nota: los registros `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`, `r9`, `r10` y `r11` y sus subregistros *no* son preservados en llamadas a funciones de biblioteca ni en llamadas al sistema (servicios del núcleo de sistema operativo). Si se necesita preservar los valores de estos registros, lo mejor es guardarlos en la pila.

1. General

1)

Una forma de imprimir un valor entero es realizando una llamada a la función `printf`. Esta toma como primer argumento una cadena de C (las cuales se representan como un puntero a caracter) indicando el formato y luego una cantidad variable de argumentos que serán impresos. La signatura en C es la siguiente:

```
int printf(const char *format, ...);
```

La forma de llamarla en ensamblador es como sigue:

```
        .data
format: .asciz "%ld\n"
i:      .quad 0xdeadbeef
        .text
        .global main
main:
      movq $format, %rdi   # El primer argumento es el formato.
      movq $1234, %rsi     # El valor a imprimir.
      xorq %rax, %rax      # Cantidad de valores de punto flotante.
      call printf
      ret
```

Modifique el código para imprimir:

1. El valor del registro `rsp`.
2. La dirección de la cadena de formato.
3. La dirección de la cadena de formato en hexadecimal.
4. El *quad* en el tope de la pila.
5. El *quad* ubicado en la dirección `rsp` + 8.
6. El valor `i`.

7. La dirección de `i`.

2) Las instrucciones `rol` y `ror` rotan los bits de su operando a izquierda y derecha, dejando el bit izquierdo –respectivamente, el derecho– en la bandera de acarreo (*carry*) del registro de estado. Toman dos operandos:

```
rol cantidad_a_rotar, registro
ror cantidad_a_rotar, registro
```

Además, existe la instrucción `adc` (*add with carry*), que toma dos operandos:

```
adc op_origen, op_destino
```

y calcula *op_destino* ← *op_origen* + *op_destino* + *acarreo*.

Use esto para encontrar cuántos bits en uno tiene un entero de 64 bits (*quad*) almacenado en el registro `rax`.

3) Implemente en ensamblador funciones que realicen lo siguiente:

1. Busquen un caracter dentro de una cadena apuntada por `rdi`.
2. Comparen dos cadenas de longitud `rcx` apuntadas por `rdi` y `rsi`.

Luego, utilizando las funciones anteriores, implemente en ensamblador el algoritmo de "fuerza bruta":

```
int fuerza_bruta(const char *S, const char *s, unsigned lS, unsigned ls)
{
    unsigned i, j;
    for (i = 0; i < lS - ls + 1; i++)
        if (S[i] == s[0]) {
            for (j = 0; j < ls && S[i + j] == s[j]; j++)
                ;
            if (j == ls)
                return i;
        }
    return -1;
}
```

Escriba una función `main` en C que llame a la función `fuerzaBruta`. Deberá mantener dos archivos separados: uno para el código en ensamblador y otro para el código en C. Para generar el binario ejecutable, debe pasar los nombres de ambos archivos fuente como argumentos de `gcc`:

```
$ gcc fuerza_bruta.s main.c
```

Este ejercicio se debe realizar sin guardar variables locales en la pila.

4) En el programa que sigue, `funcs`, implementa `void (*funcs[])() = {f1, f2, f3}`. Complételo para que la línea con el comentario corresponda a `funcs[entero]()`. Use el código más eficiente.

```
        .data
fmt:    .string "%d"
entero: .long -100
funcs:  .quad f1
```

```
        .quad f2
        .quad f3

        .text
f1:     movl $0, %esi; movq $fmt, %rdi; call printf; jmp fin
f2:     movl $1, %esi; movq $fmt, %rdi; call printf; jmp fin
f3:     movl $2, %esi; movq $fmt, %rdi; call printf; jmp fin

        .global main
main:
        pushq %rbp
        movq %rsp, %rbp

        ## Leemos el entero.
        movq $entero, %rsi
        movq $fmt, %rdi
        xorq %rax, %rax
        call scanf

        xorq %rax, %rax

        ## ¡COMPLETE CON DOS INSTRUCCIONES!
        jmp *%rdx
fin:
        movq %rbp, %rsp
        popq %rbp
        ret
```

5) Las funciones `setjmp` y `longjmp` permiten hacer saltos no locales. Sus signaturas en C se encuentran en la cabecera `setjmp.h` y son las siguientes:

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

`setjmp` "guarda" el estado de la computadora en el argumento `env` y luego `longjmp` lo restaura. Implemente `setjmp` y `longjmp`. Llámelas `setjmp2` y `longjmp2`.

2. Punto flotante

6) Implemente en ensamblador de x86_64 la función:

```
int solve(float a, float b, float c, float d, float e, float f, float *x, float *y);
```

que resuelva el sistema de ecuaciones:

$$\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned} \tag{1}$$

y escriba el resultado en los punteros `x` e `y`. La función debe devolver 0 si encontró una única solución y -1 en caso contrario.

7) Implemente en ensamblador la siguiente función:

```
void sum(float *a, float *b, int len);
```

que suma dos arreglos de flotantes de longitud `len` dejando el resultado en `a`.

8) Reimplemente la función anterior utilizando instrucciones SSE. Llámela `sum_sse`. Utilizando la función `clock_gettime` compare el tiempo computacional de cada implementación para arreglos de distinto tamaño (de 1000 a 100.000.000 elementos).

3. Funciones

9)

1. Realice un diagrama de la pila utilizada por el siguiente programa C (`stack_usage.c` en el directorio `codigo`) cuando se está ejecutando f:

```
#include <stdio.h>

int f(char a, int b, char c, long d,
      char e, short f, int g, int h)
{
    printf("a: %p\n", &a);
    printf("b: %p\n", &b);
    printf("c: %p\n", &c);
    printf("d: %p\n", &d);
    printf("e: %p\n", &e);
    printf("f: %p\n", &f);
    printf("g: %p\n", &g);
    printf("h: %p\n", &h);
    return 0;
}

int main(void)
{
    return f('1', 2, '3', 4, '5', 6, 7, 8);
}
```

   Indique en el diagrama la ubicación y el espacio utilizado por cada argumento.

2. Indique la dirección dentro de la pila en donde está almacenada la dirección de retorno de f y si es posible verifique con GDB. Sugerencia: utilice el comando `si`, para avanzar de a una instrucción.

10)

1. Compile y ejecute el código de corrutinas:

```
gcc demo_corrutinas.c guindows.c -o demo_corrutinas
./demo_corrutinas
```

2. Agregue una nueva corrutina:

```
task t3;

void ft3(void)
{
    int i;
    for (i = 0; i < 5000; i++) {
            printf("t3: i=%d\n", i);
            TRANSFER(t3, t1);
        }
    TRANSFER(t3, taskmain);
}
```

Nota: se debe reservar espacio en la pila (`stack` también para `ft3`). Haga que `ft3` realice una iteración luego de que `ft2` lo haya hecho.

3. Modifique las tres corrutinas para que impriman la dirección de una variable local antes de comenzar a iterar. Compare las direcciones mostradas.