Universidad Carlos III

Curso Estructura de computadores 2022-23

Práctica 1

Curso 2022-23

Ingeniería Informática, Segundo Curso

Paula Subías Serrano(NIA: 100472119, e-mail: 100472119@alumnos.uc3m.es)
César López Mantecón (NIA: 100472092, e-mail: 100472092@alumnos.uc3m.es)
Prof. Felix García Caballeira
Grupo: 81

Índice

1. Ejercicio 1: string_compare:

1.1. Funcionalidad

La función recibe dos parámetros, dos direcciones de memoria donde empiezan dos cadenas de caracteres. La función devolverá 1 si las cadenas son iguales y 0 si son diferentes. En caso de error (dirección de memoria nula), la función devolverá -1.
La funcionalidad de *string_compare* queda resumida en la siguiente tabla:

Input	Output
A, B: A[i] == B[i],$\forall i \in [0, \|A\| - 1]$	1
A, B: A[i] != B[i]	0
A, B: A == 0	-1
A, B: B == 0	-1

1.2. Diseño y pseudocódigo

La función es terminal, por lo que no necesita conservar *ra* en pila. Carga 1 en *a0*, valor que será modificado en caso de encontrar un error o encontrar algún carácter en A distinto de su homólogo en B. En este último caso, la función deja de ejecutarse al momento.
Los caracteres son comparados en un bucle. Al comparar dos caracteres, si son iguales, comprobará si el carácter que ha sido comparado era el carácter '\0'. Si efectivamente era el carácter de fin de cadena, saldrá de la función. Sino, continuará en el bucle de comparación. No es necesario comprobar si solo una de las cadenas ha terminado, pues en tal caso se detectará que los caracteres son distintos.
Los bloques de etiquetas se han ordenado para tener el menor número de saltos posible.
El algoritmo queda descrito en el siguiente pseudocódigo:

```
int string_compare(char* A,  char* B) {
    if (A == 0 || B == 0) {
        return -1;
    }
    int a = 0;

    while (A[a] == B[a]) {
        if A(a) == 0 {
            return 1;
        } else {
            ++a;
        }
    }
    return 0;
}
```

1.3. Plan de Pruebas

Datos a introducir:	**Descripción de la prueba:**	**Resultado esperado:**	**Resultado obtenido**
`A = 0` `*B = “gato”`	Comportamiento de que la primera dirección de memoria sea nula	`-1`	`-1`
`*A = “gato”` `B = 0`	Comportamiento cuando la segunda dirección de memoria es nula	`-1`	`-1`
`A = 0` `B = 0`	Comportamiento cuando ambas direcciones son nulas	`-1`	`-1`
`*A = “sopa”` `*B = “gato”`	Direcciones válidas, cadenas diferentes	`0`	`0`
`*A = “gato”` `*B = “gato”`	Direcciones válidas, cadenas iguales	`1`	`1`
`*A = “gato”` `*B = “gatopeludo”`	Direcciones válidas, cadenas diferentes a partir del carácter de fin de cadena	`0`	`0`
`*A = “abc”` `*B = “”`	Direcciones válidas, cadenas diferentes, la segunda vacía.	`0`	`0`
`*A = “”` `*B = “abc”`	Direcciones válidas, cadenas diferentes, la primera vacía.	`0`	`0`
`*A = “”` `*B = “”`	Direcciones válidas, cadenas iguales, ambas vacías.	`1`	`1`

2. Ejercicio 2: study_energy

2.1. Funcionalidad

La función recibirá un parámetro, la dirección del inicio de una cadena de texto (*password*), y deberá imprimir el número de ciclos invertidos en comparar las cadenas a través de *string_compare* dicha cadena con otra de una sola letra. Debe imprimir una línea por cada letra del abecedario (a excepción de la *ñ*) con el siguiente formato:

```
a: n1
b: n2
c: n3
…
```

Donde *ni* es el número de ciclos invertidos en la comparación.

2.2. Diseño y pseudocódigo

Al tratarse de una función no terminal, reservamos espacio en pila para guardar 20 bytes. Lo que se corresponde con:

- cadena de un carácter: *character*[2]= {*current*, '\0' } (4 bytes)
- *password* (4 bytes): dirección de la cadena sobre la que se va a hacer el estudio de energía.
- ciclos anteriores (4 bytes)
- 123 (código ascii de la z +1) (4 bytes)
- ra (4 bytes)

Estos datos necesitan almacenarse para poder ser conservados entre llamadas.

En cuanto a la implementación, la función deja de ejecutarse cuando *current* sea mayor o igual que 123; es decir, cuando ha comparado *password* con todas las letras del alfabeto inglés en minúscula.

Una vez terminada la ejecución, se restaura el valor de ra y se devuelve el puntero de pila al valor previo a la llamada de la función.

El algoritmo queda descrito por el siguiente pseudocódigo:

```
void study_energy(char* password){
    char character[2];
    char current = 97; //ascii code for ‘a’
    character[1] = ‘\0’
    int result;
    //ascii code for ‘z’ is 122
    while (current < 123) {
        character[0] = current;
        rdcycle t0 //t0 <- # cycles so far
        string_compare(password, character);
        rdcylce t1
        result = t1 - t2;
        printf(“%c: %i”,current, result);
        ++current;
    }
}
```

2.3. Plan de pruebas

Una prueba completa contaría con al menos 26 casos, uno por cada letra del alfabeto. No obstante, teniendo en cuenta el espacio y tiempo requerido para esto, hemos considerado oportuno probar tan sólo unos pocos para comprobar el funcionamiento correcto de la función.

Datos a introducir:	**Descripción de la prueba:**	**Resultado esperado:**	**Resultado obtenido**

*A = “password”	Cadena válida que comienza por el carácter “p”.	Ciclos para todas las letras igual excepto para p, que será mayor.	`a: 13`
			`b: 13`
			…
			`p: 20`
			…
			`z: 13`

*A = “gatomojado”	Cadena válida que comienza por el carácter “g”.	Ciclos para todas las letras igual excepto para g, que será mayor.	`a: 13`
			`b: 13`
			…
			`g: 20`
			…
			`z: 13`

*A = “”	Cadena válida vacía.	Ciclos para todas las letras igual.	`a: 13`
			…
			`z: 13`

*A = 0	Dirección de memoria nula	Ciclos para todas las letras igual.	`a: 12`
			…
			`z: 12`

3. Ejercicio 3: attack

3.1. Funcionalidad

La función debe recibir dos parámetros: *password*, dirección de memoria de la cadena de texto a descubrir; y *dummy*, dirección de memoria de la nueva cadena de texto. No devolverá nada, simplemente almacenará en *dummy* la cadena descubierta. Para descubrir la cadena se utilizará el siguiente método: dado que conocemos que *string_compare* tarda más en procesar caracteres correctos que incorrectos, podemos probar cada una de las letras del alfabeto para cada carácter de la contraseña para evaluar cuál tardan más en ser procesadas y, así, descubrir todos los caracteres de la contraseña.

3.2. Diseño y pseudocódigo

La función no es terminal, por tanto debemos almacenar el valor de *ra* en pila. Además, debemos almacenar una serie de valores que necesitamos que se conserven entre llamadas. Con esto, en pila quedan almacenados:

- *dummy_len* (4 bytes): longitud de *dummy* hasta el momento.
- *dummy* (4 bytes): dirección de la cadena *dummy*.
- *password* (4 bytes): dirección de la cadena *password*.
- *ra* (4 bytes): valor de *ra* previo a entrar a la función

La función llamará a una función auxiliar llamada *study_energy_attack*, encargada de obtener el siguiente carácter de la contraseña a descubrir; y añadirá el carácter descubierto a *dummy*.
La función *study_energy_attack* recibe dos direcciones de memoria (*password* y *dummy*) y la longitud hasta el momento de *dummy*. Añadirá al final de la cadena el carácter vacío y comprobará si la contraseña es correcta. En caso de que sea correcta, dejará de ejecutarse y devolverá un 0, valor que indicará a *attack* que la contraseña ha sido descubierta. En caso contrario, probará a añadir cada una de las letras del alfabeto a *dummy* y devolverá el carácter con el que *string_compare* tarde más en comparar ambas cadenas (aplicando el principio del ejercicio 2). Para esto, necesitamos almacenar los siguientes valores en pila:

- *dummy_len*: tamaño de la cadena *dummy*.
- *password*: dirección de *password*.
- *123*: cota superior del código ascii del alfabeto.
- *dummy*: dirección de la cadena *dummy*.
- *prev_cycles*: ciclos antes de llamar a *string_compare*.
- *cycles_max*: mayor número de ciclos invertidos en una comparación.
- *char_out*: carácter asociado al mayor número de ciclos.
- *ra*: valor de ra previo a la llamada de la función.

El algoritmo queda descrito por el siguiente pseudocódigo.

```
void attack(char* password, char* dummy) {
    char new;
    while (new != ‘\0’) {
        int dummy_len = 0;
        char new = study_energy_attack(password, dummy, dummy_len);
        dummy[dummy_len] = new;
```

```
        ++dummy_len;
    }
}

char study_energy_attack(char* password, char* dummy, int dummy_len){

    int max_cycles = 0;
    int prev_cycles = 0;
    int prev_char = 97; // a
    int char_out;
    dummy[dummy_len] = '\0';

    if (sutudy_energy_attack(password, dummy) == 1) {
         return 0;
    }

    while (prev_char <= 123) {
        dummy[dummy_len] = prev_char;
        prev_cycles = rdcycles;
        string_compare(password, dummy);
        int current_cycles = rdcycles;

        int cycles = current_cycles - prev_cycles;
        if (cycles > max_cycles) {
            max_cycles = cycles;
            char_out = prev_char;
        }
        ++prev_char;
    }
}
```

3.3. Plan de pruebas

Datos a introducir:	**Descripción de la prueba:**	**Resultado esperado:**	**Resultado obtenido**
“password”	Cadena básica de longitud 8.	*dummy = “password”	*dummy = “password”
“gato”	Cadena de la contraseña de longitud 4(menor que 8), todas las letras distintas.	*dummy = “gato”	*dummy = “gato”
“”	Cadena vacía.	*dummy = “”	*dummy = “”
“aaaaaaaa”	Cadena de longitud 8 con todas las letras iguales.	*dummy = “aaaaaaaa”	*dummy = “aaaaaaaa”
“zzzzzzzz”	Cadena de longitud 8, último caracter.	*dummy = “zzzzzzzz”	*dummy = “zzzzzzzz”

4. Ejercicio 4:

String_compare, tal y cómo está implementada, deja de ejecutarse en cuanto detecta un carácter que no es igual en las cadenas de texto que obtiene como parámetros. Esto presenta una vulnerabilidad, ya que tardará más en procesar una cadena cuantos más caracteres correctos presente (idea utilizada en el ejercicio anterior para descubrir una contraseña).

Para que sea imposible atacar a *string_compare* de esta manera, las comparaciones siempre tienen que durar lo mismo, ya sea una cadena correcta o no. Una posible solución sería que *string_compare* siempre recorra toda la cadena de password, aunque no coincida algún carácter. Si hay alguno que no coincide, cambiará el valor de a0 a 0, pero seguirá comparando. De esta forma, las cadenas correctas e incorrectas tardarían lo mismo en ser procesadas, haciendo imposible descubrir la contraseña por este método. Además, en el caso de detectar un carácter correcto tendrá que realizar el mismo número de operaciones que si no lo fuera. Solo saldrá de este bucle al llegar al carácter vacío de la cadena A(password).

5. Horas invertidas

NOMBRE	INDIVIDUAL	EQUIPO	TOTAL
Paula Subías Serrano	1	5.5	**6.5**
César López	1.5	5.5	**7**
TOTAL	2.5	11	**13.5**

6. Conclusiones

La programación en ensamblador no es tarea sencilla. Es un entorno donde no existen funciones de alto nivel que durante el proyecto hemos echado en falta. Además, viniendo de un lenguaje de programación como es Python, el salto en abstracción es notorio; siendo Python un lenguaje mucho más cercano al lenguaje natural de lo que es ensamblador.

La gestión de los registros y memoria, especialmente la pila, fue una tarea que nos presentó más de una dificultad, pero que una vez entendido supimos emplear de manera correcta y nos fue de mucha utilidad. Respecto a las instrucciones, el problema más complicado fue acostumbrarnos a no contar con variables locales y al número limitado de registros, obligándonos a pensar nuestros algoritmos de otra manera gracias a las restricciones que la programación en ensamblador nos imponía.

Una lección bastante importante que hemos obtenido, es el uso del pseudocódigo y la planificación del programa antes de escribir. La nueva sintaxis y las restricciones de ensamblador nos ha hecho entender que planificar el algoritmo en papel **antes de programar** es una tarea, aunque no imprescindible para programas de este tamaño, muy útil y que facilita la programación y ahorra horas de trabajo tanto de corrección y depuración, como de reescritura de funciones en su totalidad.