



Organización de Computadoras

Segundo Cuatrimestre 2017

Trabajo Práctico 1

| Integrante | Padrón | Correo electrónico |
|------------------|--------|----------------------------|
| Rodrigo De Rosa | 97799 | rodrigoderosa@outlook.com |
| Marcos Schapira | 97934 | schapiramarcos@gmail.com |
| Facundo Guerrero | 97981 | facundoiguerrero@gmail.com |

Índice

| | |
|---|----------|
| 1. Diseño e Implementación | 1 |
| 1.1. Estructura del problema | 1 |
| 1.2. Entorno | 1 |
| 1.3. Complicaciones | 1 |
| 1.4. Desarrollo | 2 |
| 1.5. Manejo de errores | 2 |
| 1.5.1. Valores devueltos por la función <code>main</code> | 2 |
| 1.6. Documentación | 3 |
| 1.6.1. Funciones en assembly C | 3 |
| 1.6.2. Funciones en assembly MIPS | 4 |
| 2. Ejecución | 5 |
| 2.1. Instrucciones para la compilación | 5 |
| 2.2. Instrucciones para la ejecución | 5 |
| 2.3. Pruebas | 6 |
| 2.3.1. Prueba 1 | 6 |
| 2.3.2. Prueba 2 | 6 |
| 2.3.3. Prueba 3 | 6 |
| 2.3.4. Prueba 4 | 6 |
| 2.3.5. Prueba 5 | 7 |
| 2.3.6. Prueba 6 | 7 |
| 2.3.7. Prueba 7 | 7 |
| 2.3.8. Prueba 8 | 8 |
| 2.3.9. Prueba 9 | 8 |
| 2.3.10. Prueba 10 | 8 |
| 2.3.11. Prueba 11 | 8 |
| 2.3.12. Prueba 12 | 9 |
| 3. Conclusiones | 9 |

1. Diseño e Implementación

En este trabajo práctico, cuyo objetivo es familiarizarse con el conjunto de instrucciones MIPS y el concepto de ABI, se implementa el mismo programa que el del TP0 pero esta vez utilizando assembly MIPS. Dicho programa recibe una entrada de texto e identifica los palíndromos que se encuentran en ella.

1.1. Estructura del problema

La entrada de texto previamente mencionada es una cadena de caracteres *ASCII* sin ninguna restricción. Dentro de esta cadena son consideradas *palabras* aquellas que están compuestas por los caracteres:

- $a - z$
- $0 - 9$
- $_ y -$

Cualquier otro carácter *ASCII* es considerado un *espacio*. Es decir, indica el fin de una *palabra* y el comienzo de otra. Cabe destacar que una cadena con un sólo carácter es considerada *palabra*.

1.2. Entorno

El trabajo se realizó en una máquina virtual *NetBSD* (que simula tener un procesador *MIPS*) montada por el emulador *GXEmul* en *Ubuntu 17.04*.

1.3. Complicaciones

Durante el desarrollo de el trabajo practico que esta siendo presentado, se presentaron muchas dificultades. La primera de ellas fue al momento de usar funciones que se encontraban en módulos externos, es decir, cuando se quería llamar desde un módulo main a una función auxiliar. El problema en si, fue que al llamar solamente a una función externa el salto funcionaba, pero cuando desde un main se querían hacer 2 saltos a funciones externas, dicho programa arrojaba un *Segmentation Fault*. Por otro lado, al intentar debuggear el programa con GDB, no se obtenía ningún tipo de información ya que de GDB se obtenía la siguiente salida:

```
Starting program: /root/TPs/TP1/tests/buffer/buffer
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
warning: Warning: GDB can't find the start of the function at 0x27bdffc8.
```

```
GDB is unable to find the start of the function at 0x27bdffc8
and thus can't determine the size of that function's stack frame.
This means that GDB may be unable to access that stack frame, or
the frames below it.
```

```
This problem is most likely caused by an invalid program counter or
stack pointer.
```

```
However, if you think GDB should simply search farther back
from 0x27bdffc8 for code which looks like the beginning of a
function, you can increase the range of the search using the 'set
heuristic-fence-post' command. 0x27bdffc8 in ?? ()
```

Luego de continuar un arduo análisis del problema, y con ayuda del grupo de consultas, se pudo advertir que el error se encontraba al momento de salvar el *ra* al inicio de una función, y restaurarlo al final de la misma.

Una vez superada dicha complicación, se pudo integrar casi toda la funcionalidad del programa que se tenía hasta el momento. Como se menciono recientemente, lo primero que se hizo fue programar funciones por separado que realizaban las distintas tareas necesarias para el correcto funcionamiento del programa, las cuales leían una entrada con un *SYS.READ*, el cual era guardado en un buffer definido en la sección de *.data* del *.S*. En este caso, el parámetro del syscall pasado en *a2* (available space) también estaba hardcodeado y era el mismo tamaño del buffer. Una vez logrado esto, el siguiente paso fue introducir el buffer dinámico, que fue donde apareció la siguiente complicación. Entonces, lo que se quería hacer ahora es que el buffer fuera dinámico, es decir que recibiera por parámetro su tamaño y se llamara a *mymalloc* para obtener el puntero a el mismo. En este caso, el programa terminaba con un *Segmentation Fault*. Al intentar debuggear el programa con GDB, se obtuvo una salida muy similar a la presentada anteriormente, por lo que tampoco se pudo obtener información relevante sobre el error. Luego de realizar el análisis pertinente, pudimos arreglar el error que se encontraba al momento de realizar el pedido de memoria con la función *my_malloc*.

También se puede notar, que una complicación adicional fue la poca información que se obtenía de GDB cuando se corrían los programas que terminaban con *Segmentation Fault*.

1.4. Desarrollo

El programa fue implementado en lenguaje C y assembly MIPS.

ESPECIFICACIONES DEL TP.

A continuación se detalla la documentación explícita de las funciones implementadas.

1.5. Manejo de errores

A lo largo del desarrollo del programa se definen ciertos errores para manejar posibles fallas del programa y así lograr un funcionamiento controlado y acorde. Estas son:

■ ALLOC_ERROR

El error se puede dar al llamar a la función `malloc`. Junto a su mensaje específico se imprime a la vez el código generado por `strerror` en la anterior función.

Mensaje:

An error occurred while allocating memory!

■ REALLOC_ERROR

El error se puede dar al llamar a la función `realloc`. Junto a su mensaje específico se imprime a la vez el código generado por `strerror` en la anterior función.

Mensaje:

An error occurred while reallocating memory!

■ INPUT_OPEN_ERROR

El error se puede dar al llamar la función `fopen`. Junto a su mensaje específico se imprime a la vez el código generado por `strerror` en la anterior función.

Mensaje:

An error occurred while opening input file!

■ OUTPUT_OPEN_ERROR

El error se puede dar al llamar la función `fopen`. Junto a su mensaje específico se imprime a la vez el código generado por `strerror` en la anterior función.

Mensaje:

An error occurred while opening output file!

■ RESULT_WRITING_ERROR

El error se puede dar al llamar la función `fprintf` si no se logró escribir todo el mensaje o si algo falló. Junto a su mensaje específico se imprime a la vez el código generado por `strerror` en la anterior función.

Mensaje:

An error occurred while writing the result!

■ PALINDROME_ERROR_MESSAGE

El error se puede dar al llamar a la función interna `get_palindromes`. Esta devuelve `NULL` en caso de fallar (junto con su adecuado mensaje, explicado a continuación en el informe).

Mensaje:

An error occurred while checking for palindromes!

1.5.1. Valores devueltos por la función main

Los siguientes códigos son mensajes devueltos por la función `main` al utilizar las funciones internas del programa (documentadas en la próxima sección del informe). Algunos de estos valores, en especial `FAIL` y `SUCCESS` son utilizados en otras funciones como valores booleanos `False` y `True` respectivamente.

■ SUCCESS valor 0

valor booleano de éxito.

■ FAIL valor 1

valor booleano de falla.

- **PALINDROME_ERROR** valor mayor estricto a cero.
*ocurre cuando la función **palindrome** falla. Esto puede deberse a los siguientes errores:*
 - INPUT_MALLOC_ERROR** valor 1.
ocurre cuando la función **malloc** falla en el contexto de Input.
 - OUTPUT_MALLOC_ERROR** valor 2.
ocurre cuando la función **malloc** falla en el contexto de Output.
 - WRITE_ERROR** valor 3.
ocurre cuando hay un error de escritura en assembly MIPS.
 - READ_ERROR** valor 4.
ocurre cuando hay un error de lectura en assembly MIPS.
- **BAD_ARGUMENTS** valor 4
*ocurre cuando la función **process_params** devuelve este mismo código al no poder procesar los parámetros correctamente.*
- **BAD_INPUT_PATH** valor 5
*ocurre cuando la función **open_input** devuelve **FAIL**.*
- **BAD_OUTPUT_PATH** valor 6
*ocurre cuando la función **open_output** devuelve **FAIL**.*
- **READING_ERROR** valor 7
*ocurre cuando la función **read_input** devuelve **FAIL** o **NULL**.*

1.6. Documentación

Las siguientes funciones fueron implementadas con el objetivo de encontrar una solución al problema en cuestión.

1.6.1. Funciones en assembly C

- **FILE* open_input(char* path)**
*Abre el input_file y se devuelve su fp. Si el path es NULL, se utiliza **DEFAULT_INPUT** siendo en este caso stdin.*

Parámetros:

path: Dirección del archivo a abrir

Return:

File Pointer de input o **DEFAULT_INPUT** en caso de no especificar un path.

Errores Posibles:

INPUT_OPEN_ERROR
- **FILE* open_output(char* path)**
*Abre el output_file y se devuelve su fp. Si el path es NULL se utiliza **DEFAULT_OUTPUT** siendo en este caso stdout.*

Parámetros:

path: Dirección del archivo a abrir

Return:

File Pointer de output o **DEFAULT_OUTPUT** en caso de no especificar un path.

Errores Posibles:

OUTPUT_OPEN_ERROR
- **void close_files(FILE* fp1, FILE* fp2)**
Cierra los dos archivos recibidos.

Parámetros:

fp2: File Pointer de archivo a cerrar

fp1: File Pointer de archivo a cerrar
- **void print_help()**
Imprime por consola información de los comandos y sobre el uso del programa.

- `void print_version()`
Imprime por consola la version del programa y los integrantes del grupo.
- `int process_params(int argc, char** argv, char** input_file, char** output_file)`
Procesa los parámetros de entrada del programa y almacena los paths correspondientes en los parámetros de la función.

Parámetros:

`argc`: Cantidad de argumentos del programa
`argv`: Vector de argumentos del programa
`input_file`: Puntero al string que contiene el path del input
`output_file`: Puntero al string que contiene el path del output

Return:

`SUCCESS` o `BAD_ARGUMENTS`, en el segundo caso este valor es verificado y manejado en la función `main`.

1.6.2. Funciones en assembly MIPS

- `palindrome`
Maneja el buffer tanto de lectura como de escritura, verificando si las palabras a analizar son o no palíndromas.

Parámetros:

`input file descriptor`: Puntero al input file descriptor
`input buffer size`: Tamaño del buffer
`output file descriptor`: Puntero al output file descriptor
`output buffer size`: Tamaño del buffer de salida

Return:

`SUCCESS` o `ERROR`, en el segundo caso este valor es verificado y manejado en la función `main`.

Errores Posibles:

`INPUT_MALLOC_ERROR`, `OUTPUT_MALLOC_ERROR`, `WRITE_ERROR`, `READ_ERROR`

- `mymalloc`
Funcion malloc implementada en assembly MIPS.

Parámetros:

`size`: Tamaño de memoria a pedir

Return:

`PUNTERO_AL_BLOQUE` o `ERROR`, en el segundo caso este valor es verificado y manejado en la función `palindrome`

- `myfree`
Funcion free implementada en assembly MIPS.

Parámetros:

`pointer`: Puntero al bloque a liberar.

Return:

`SUCCESS` o `ERROR`

- `myrealloc`
Funcion realloc implementada en assembly MIPS.

Parámetros:

`old_pointer`: Puntero vector original
`size`: Tamaño del vector original
`size_inc`: Incremento de memoria

Return:

`PUNTERO_AL_BLOQUE` o `ERROR`

- `get_word`

Lee una palabra, separando como espacios a los caracteres anteriormente mencionados.

Parámetros:

`input file descriptor`: Puntero al input file descriptor
`len_pointer`: puntero donde se guarda el largo de la palabra leída
`input buffer`: Puntero al buffer de entrada

Return:

`PALABRA` o `ERROR`, en el segundo caso este valor es verificado y manejado en la función `palindrome`

Errores Posibles:

`READ_ERROR`

- `get_char`

Lee un caracter.

Parámetros:

`input file descriptor`: Puntero al input file descriptor
`input buffer`: Puntero al buffer de entrada

Return:

`CHARACTER` o `ERROR`, en el segundo caso este valor es verificado y manejado en la función `get_word`

Errores Posibles:

`READ_ERROR`

- `is_palindrome`

Verifica si una palabra es o no palíndroma.

Parámetros:

`palabra`: Palabra a analizar
`size`: Longitud de la palabra

Return:

`VALOR BOOLEANO`

- `put_char`

escribe una palabra caracter.

Parámetros:

`output file descriptor`: Puntero al output file descriptor
`palindromo`: Palíndromo a escribir
`output buffer`: Puntero al buffer de entrada

Return:

`SUCCESS` o `WRITE_FAIL`, en el segundo caso este valor es verificado y manejado en la función `palindrome`

Errores Posibles:

`WRITE_ERROR`

2. Ejecución

2.1. Instrucciones para la compilación

Para compilar el programa se debe abrir una consola en el directorio donde se encuentra el archivo fuente (`tp.c`) y correr el comando: `gcc -Wall tp1.c [-o OUTPUT]`.

2.2. Instrucciones para la ejecución

Suponiendo que nuestro archivo ejecutable fuera `tp1`, los comandos de consola para ejecutarlo son:

- `./tp1 -h` para ver la ayuda.
- `./tp1 -v` para ver la versión.
- `./tp1 -i /INPUT -o /OUTPUT` para correr el programa con `INPUT` como archivo de entrada y `OUTPUT` como archivo de salida. Ambos son opcionales y son reemplazados por `stdin` y `stdout` respectivamente.

2.3. Pruebas

Para probar el correcto funcionamiento del programa se utilizo un set de prueba. A continuación se muestra la composición y resultados de las ejecuciones de dicho set. Además, notar que desde la prueba 1 hasta la prueba 9, la entrada es mediante un archivo, es decir que se provee el archivo mediante `-i test_inputN` (con N = número de prueba), y la salida por terminal. Por último, vale aclarar que las pruebas se realizaron tanto en Ubuntu como en NetBSD.

2.3.1. Prueba 1

Caso de prueba provisto por la cátedra. Vale aclarar que un carácter es considerado palíndromo.

Entrada:

```
Somos los primeros en completar el TP 0.  
Ojo que la fecha de entrega del TP0 es el martes 12 de Septiembre.
```

Salida:

```
Somos  
0  
Ojo
```

2.3.2. Prueba 2

Este caso de prueba intenta demostrar el correcto funcionamiento de la detección de espacios. Como se puede ver, en la primera linea las palabras están separadas por el carácter espacio, pero en la segunda se linea se intenta demostrar que los caracteres no validos (ver sección 1.1) también funcionan como espacios. Además, vale notar que la palabra palíndroma se detecta sin importar mayúsculas o minúsculas.

Entrada:

```
MeNEm neUquEn 1a2d323d2a1 adke  
pepe)nene/larral=dom-mod?a23_32a
```

Salida:

```
MeNEm  
neUquEn  
1a2d323d2a1  
larral  
dom-mod  
a23_32a
```

2.3.3. Prueba 3

El objetivo de esta prueba es ver el funcionamiento de los caracteres validos que no sean letras ni números. Como se vio en la sección 1.1, `y` `_` deben ser considerados como caracteres validos. Entonces esta prueba quiere demostrar el funcionamiento de dichos caracteres. Además, nuevamente notar que la detección de las palabras no es `sensitive`.

Entrada:

```
aD2eT_R_Te2Da/4004?CheVr  
peep23*** aviondaad  
neUqUeN&NarNran
```

Salida:

```
aD2eT_R_Te2Da  
4004  
daad  
neUqUeN  
NarNran
```

2.3.4. Prueba 4

Esta prueba tiene el objetivo de corroborar el caso borde donde la entrada es vacía.

Entrada:

Salida:

2.3.5. Prueba 5

El objetivo de esta prueba es verificar el correcto funcionamiento de la detección un único carácter.

Entrada:

a

Salida:

a

2.3.6. Prueba 6

El objetivo de esta prueba es corroborar el correcto funcionamiento del programa cuando se alcanza el límite inicial de tamaño del string que contiene las palabras palíndromas. En esta prueba, se ingresa una palabra palíndroma de 128 caracteres que es la capacidad máxima inicial.

La entrada para las siguientes pruebas es continua. Se hace un salto de línea con fines ilustrativos.

Entrada:

aa-
-aa

Salida:

aa-
-aa

2.3.7. Prueba 7

Esta prueba intenta demostrar el funcionamiento adecuado del programa cuando se supera el tamaño inicial del string donde se guardan las palabras palíndromas. Aquí, se ingresa una palabra palíndroma de 129 caracteres, es decir que se supera en 1 carácter la capacidad máxima. De esta manera, se esta forzando al programa a aumentar la capacidad máxima de almacenado. En otras palabras, se intenta probar el correcto funcionamiento del caso borde de la memoria, donde el programa debe hacer un realloc. Notar que se presentan dos casos de prueba, uno donde una misma palabra supera el límite, y en segundo lugar con palabras distintas.

La entrada para las siguientes pruebas es continua. Se hace un salto de línea con fines ilustrativos.

Entrada 1:

aa-
-aa

Salida 1:

aa-
-aa

Entrada 2:

aa-
-aaa menem

Salida 2:

aa-
-aaa
menem

2.3.8. Prueba 8

Esta prueba tiene como objetivo verificar el funcionamiento del programa cuando se alcanza el máximo inicial lectura. Entonces, se provee al programa una entrada de 256 caracteres.

La entrada para las siguientes pruebas es continua. Se hace un salto de línea con fines ilustrativos.

Entrada:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Salida:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

2.3.9. Prueba 9

El objetivo de esta prueba es corroborar el correcto funcionamiento del programa cuando se supera el tamaño inicial de lectura. Con dicho motivo, se ingresa una entrada de 257 caracteres, superando en 1 carácter la capacidad máxima. Al igual que la prueba 7, se esta probando el realloc pero en este caso para la lectura. **La entrada para las siguientes pruebas es continua. Se hace un salto de línea con fines ilustrativos.**

Entrada:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Salida:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-  
-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

2.3.10. Prueba 10

Se repite la prueba 1, esta vez con el objetivo de probar la entrada-salida vía archivos. La entrada se pasa por parámetro como `-i test_input1` y se pasa el archivo de salida `-i test_output10`. La salida mostrada, es lo que contiene el archivo de salida anteriormente mencionado.

Entrada:

```
Somos los primeros en completar el TP 0.  
Ojo que la fecha de entrega del TP0 es el martes 12 de Septiembre.
```

Salida:

```
Somos  
0  
Ojo
```

2.3.11. Prueba 11

Se repite la prueba 2, esta vez con el objetivo de probar la entrada-salida vía terminal. Se correrá el programa sin ningún parámetro, y se hará el ingreso por entrada estándar (indicando su finalización con `Ctrl+D`) y se obtendrá la salida de la misma forma.

Entrada:

```
MeNEm neUquEn 1a2d323d2a1 adke
```

```
pepe)nene/larral=dom-mod?a23_32a
```

Salida:

```
MeNEm  
neUquEn  
1a2d323d2a1  
larral  
dom-mod  
a23_32a
```

2.3.12. Prueba 12

Se repite la prueba 3, con el objetivo de verificar la entrada estándar y salida vía archivos. La salida se pasa por parámetros como `-i test_input12`. La salida mostrada a continuación, es lo que contiene el archivo de salida anteriormente mencionado.

Entrada:

```
aD2eT_R_Te2Da/4004?CheVr  
peep23* * * aviondaad  
neUqUeN&NarNran
```

Salida:

```
aD2eT_R_Te2Da  
4004  
daad  
neUqUeN  
NarNran
```

3. Conclusiones

La principal conclusión que obtuvimos a partir del desarrollo de este trabajo práctico es que, al trabajar en *NetBSD* se debe ser más cuidadoso al programar que en el caso de, por ejemplo, *Ubuntu*. Con esto nos referimos a ciertas situaciones particulares en las que *Ubuntu* resuelve ciertos problemas de la programación que *NetBSD* no resuelve y resulta en un error.

Por ejemplo, en la línea 291 de `tp.c` se agrega un fin de línea que no es necesario en *Ubuntu*, pues lo 'resuelve solo', pero sí lo es en *NetBSD* pues en este último, el programa no encuentra en fin de línea e imprime caracteres de más.

Por esta misma razón, notamos que es recomendable trabajar constantemente utilizando la carpeta virtual de `sshfs` para poder compilar y ejecutar el programa en *NetBSD* en lugar de hacerlo en *Ubuntu*. De todas maneras, al tener que desarrollar en *Assembler*, no queda otra opción que hacer esto recién mencionado.