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

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 caracter *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 caracter 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 hacr 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.

Sumado a todo esto, también se presento la dificultad del pasaje de mas de 4 parámetros que es la cantidad estipulada por la ABI. Esto se soluciono utilizando variables globales para los parámetros que permanecían constantes durante la ejecución del programa, como por ejemplo el tamaño de buffer de entrada y salida.

1.4. Desarrollo

El programa fue implementado en lenguaje C y assembly MIPS.

Dicho programa se inicia en C. Aquí se realiza la validación y el procesamiento de los parámetros recibidos, la apertura de archivos y su respectivo manejo de errores. Luego de tener los archivos abiertos, se obtiene el FILE DESCRIPTOR de los mismos con la función fileno y se pasa el control a la función palindrome, la cual esta implementada en MIPS. Por ultimo, en C también se realiza el manejo de errores, si los hubiese, una vez que el la función palindrome devuelve el control al programa en c.

Entonces una vez que la función palindrome tiene el control del programa, esta se encarga de llamar a las distintas funciones distribuidas en módulos que se encargan de identificar, procesar e imprimir los componentes léxicos que resulten ser palíndromos. La lógica que siguen dichas funciones es la que se presentara a continuación. Luego de almacenar los parámetros recibidos desde la función de C, la función palindrome reserva memoria para los buffer de entrada y salida a utilizar. Dicha operación, se lleva a cabo utilizando la función my_malloc provista por la cátedra. Cabe aclarar, que también se realiza el manejo de errores en caso de ser necesario. Luego de tener el espacio de los buffers reservados, se llama a la función get_word la cual se encarga de formar una palabra para que posteriormente sea procesada por is_palindrome. La función get_word llama a get_char la cual se encarga de leer un char del buffer estático de entrada, y en el caso de que este se encuentre vacío se encarga de volver a llenarlo. Luego de realizar dichas operaciones, se devuelve el carácter leído a get_word la cual verifica si dicho carácter leído es valido o un espacio, para añadirlo o no al resto de los caracteres acumulados hasta el momento. En el caso del que el ultimo carácter leído sea un espacio, se agrega un fin de linea al final de la palabra que esta próxima a procesarse. Contrariamente se agrega el nuevo carácter al resto y se vuelve a llamar a la función get_char hasta completar una palabra.

Luego de completada la palabra, el control es cedido a la función is_palindrome. Dicha función recorrer la palabra recibida por parámetros y compara el carácter i con el carácter len-i. Es decir, compara que el primer carácter sea igual al ultimo, que el segundo carácter sea igual al ante ultimo y así sucesivamente. Luego de terminado el procesamiento devuelve 1 o 0 representando True o False a la función is_palindrome. En caso de recibir True, esta función lo que hace es llamar a put_char que lo que hace es escribir la palabra en el buffer de salida.

A continuación se detalla el manejo de errores y la documentación explicita 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 ocurred 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 ocurred 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 ocurred 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 ocurred 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 ocurred 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 ocurred 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 codigo 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 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 leida

  `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:

  `CARACTER 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 `compile.sh` y correr la instrucción '`bash ./compile.sh`'.

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 -I IBUF-SIZE -O OBUF-SIZE` para correr el programa con `INPUT` como archivo de entrada y `OUTPUT` como archivo de salida con los tamaños `IBUF-SIZE` y `OBUF-SIZE` para los buffers de entrada y salida, respectivamente. Todos los parámetros son opcionales y son reemplazados por `stdin`, `stdout`, `1` y `1` respectivamente.

2.3. Pruebas

Para probar el correcto funcionamiento del programa en todas las combinaciones posibles con pruebas automáticas, se creó el archivo `testing.sh`. Este archivo puede correrse abriendo una consola en su directorio y corriendo la instrucción '`bash ./testing.sh`' (en caso de no haber compilado previamente, el mismo programa se encarga de hacerlo). Para las pruebas de stress se debe hacer lo mismo pero con `stress_testing.sh`. En estas pruebas automatizadas se prueba lo siguiente:

2.3.1. Prueba 1

Archivo de entrada vacío con valores variables de tamaños de buffer, tanto de entrada como de salida. Estos valores fueron:

- Entrada: 1 — Salida: 1
- Entrada: 10 — Salida: 1
- Entrada: 1 — Salida: 10
- Entrada: 100 — Salida: 100
- Entrada: 1000 — Salida: 100
- Entrada: 100 — Salida: 1000
- Entrada: 1 — Salida: 1000
- Entrada: 1000 — Salida: 1

2.3.2. Prueba 2

Salida a un archivo específico, tanto con un archivo '`.txt`' de entrada como con stdin. En el primer caso se verificaron, ademas, valores variables de buffers (tanto de entrada como de salida). Los tamaños de buffers fueron los siguientes:

- Entrada: 1 — Salida: 1
- Entrada: 10 — Salida: 1
- Entrada: 1 — Salida: 10
- Entrada: 100 — Salida: 100
- Entrada: 1000 — Salida: 100
- Entrada: 100 — Salida: 1000
- Entrada: 1 — Salida: 1000
- Entrada: 1000 — Salida: 1

El archivo que se utilizó fue: (NOTACIÓN: [*X] significa que habia X veces seguidas un caracter)

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

aD-2eT_R_Te2-Da/4004?CheVr
peep23***   avion{daad}
neUqUeN&NarNran

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

a[*128] menem

a[*129]

a[*256]

#$%&&/%%$#"#%&&$#))(/==)"
```

Y la salida fue comparada con el archivo:

```
Somos
0
Ojo
aD-2eT_R_Te2-Da
4004
daad
neUqUeN
NarNran
MeNEm
neUquEn
1a2d323d2a1
larral
dom-mod
a23_32a
a[*128]
menem
a[*129]
a[*256]
```

En el segundo caso, se verificaron casos específicos y siempre se utilizaron los tamaños de buffer 1 para ambos buffers. Los casos de prueba fueron los siguientes:

```
in: R                                    | out: R
in: r                                    | out: r
in: 7                                    | out: 7
in: -                                    | out: -
in: _                                    | out: _
in: Ae4-_-4Ea                            | out: Ae4-_-4Ea
in: 123_-_321&NONPALINDROME/(neUqUEn     | out: 123_-_321 [newline] neUqUEn
in: 42J-kL_-Edkl1241                     | out: VOID
in: !)$)=)?!=/=%)!?#)[{{{{}}]]]**¡¡!!}}* | out: VOID
```

Los primeros cinco casos verifican que un solo caracter del rango de valores que corresponde a los que forman *palabras* eran efectivamente considerados palabras; la sexta verifica que una palíndromo compuesto por varios caracteres pertenecientes a los rangos de caracteres 'no espacios' es correctamente identificado; la séptima verifica que en una misma línea en la que se encuentran tanto palíndromos como no palíndromos separados por caracteres 'espacio', todos son identificados por separado y los palíndromos son devueltos; por último, las últimas dos pruebas verifican que el programa no indique que un no palíndromo es un palíndromo y que una línea que sólo contiene espacios no escriba nada en la salida.

2.3.3. Prueba 3

Salida a stdout. En este caso se repitieron las pruebas recién mencionadas pero utilizando a stdout como archivo de salida. Con estas últimas dos combinaciones, tenemos las cuatro posibles; esto es:

- `stdin` — `stdout`
- `in file` — `stdout`
- `stdin` — `out file`
- `in file` — `out file`

2.3.4. Prueba 4

Valores inválidos de tamaños de buffer; se verificó que funcionara correctamente el método de seteado de valor default para el largo de los buffers (`if (bufsize <1) bufsize = 1`). Los tamaños probados fueron:

- `ibuf = -1` — `obuf = 10`
- `ibuf = 10` — `obuf = -1`
- `ibuf = -1` — `obuf = -1`
- `ibuf = 0` — `obuf = 0`

En todos los casos, el archivo de pruebas utilizado fue el de las pruebas 2 y 3.

2.3.5. Prueba 5

Pruebas de stress; se creó un archivo de 1MB (1000 lineas de 1000 caracteres) en el cual una de cada cuatro lineas contiene 40 palíndromos de 25 caracteres iguales separados por un caracter espacio (esto es 10000 palíndromos) y las otras tres de cada cuatro contienen 100 palabras no palíndromas separadas por un caracter espacio. El objetivo de estas pruebas fue ver cómo variaba el tiempo de ejecución si se variaba el tamaño de los buffers, tanto de entrada como de salida. Ver próxima sección para los resultados.

2.4. Pruebas de Stress

En esta sección mostraremos los resultados de las pruebas de stress mencionadas previamente. Las combinaciones utilizadas fueron:

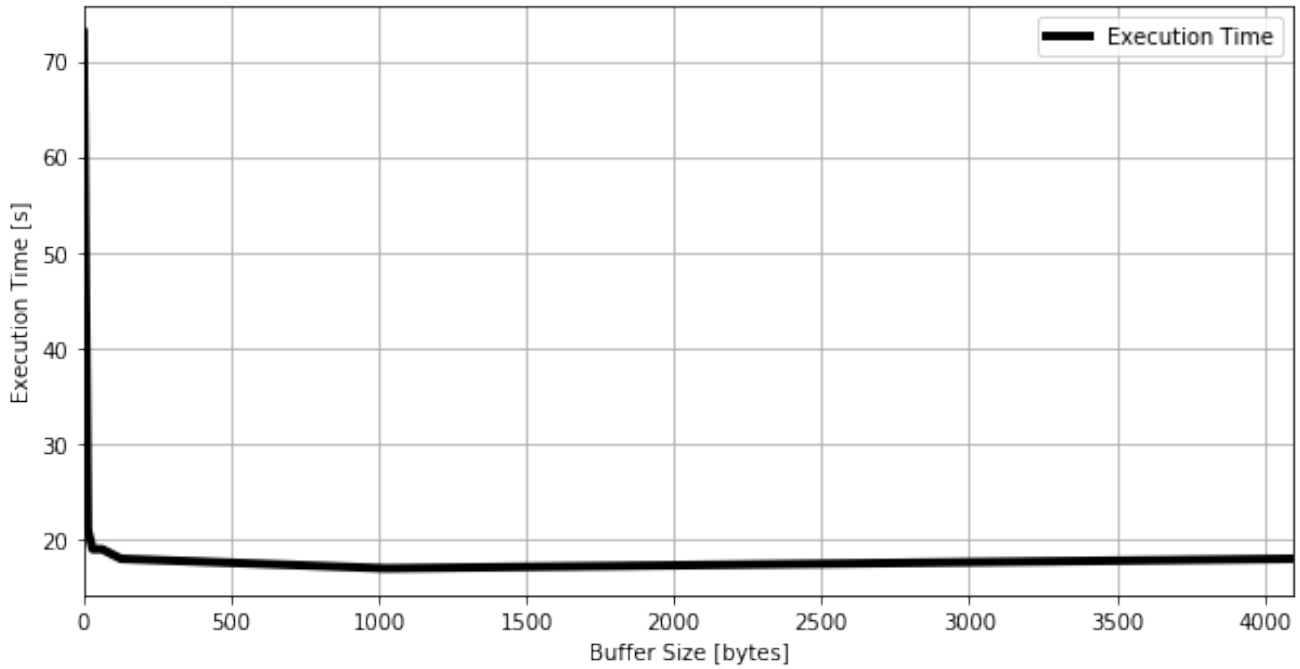
- Tamaños iguales para input buffer y output buffer con los valores 1, 16, 32, 64, 128, 1024 y 4096.
- Todas las combinaciones posibles, siempre que $i \neq o$, para los valores 1, 16, 32, 64 y 128.

En las primeras se utilizó el mismo tamaño para ambos buffers y en las últimas uno siempre era mayor al otro. Inicialmente se saltaba de 128 a 1024 y luego a 4096, pero dado que se observó que entre estos tres valores sólo había diferencia de dos segundos (y el más rapido era 1024), se decidió agregar valores intermedios entre 1 y 128 para poder ver en qué valor comenzaba a no valer la pena tener más información en memoria ya que el tiempo que se ganaba era mínimo; por eso se encuentran los valores de 16, 32 y 64.

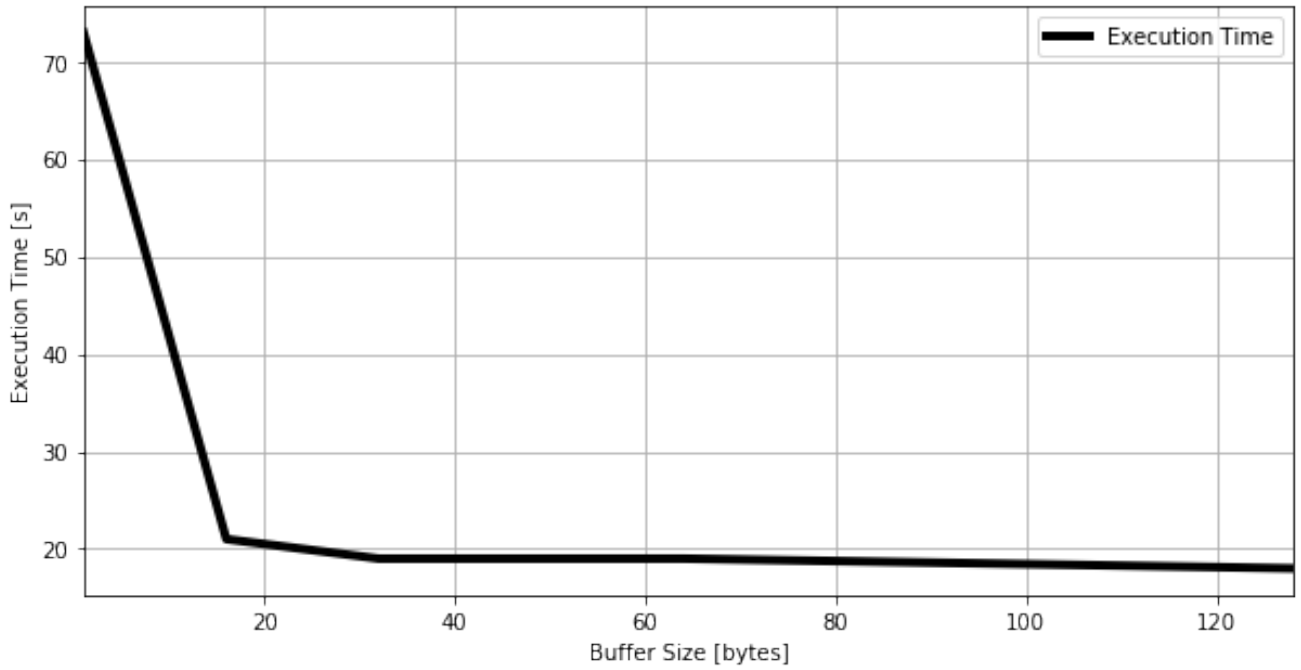
Los valores de tiempo obtenidos para las primeras siete pruebas son:

Buffer Size [bytes]	Elapsed Time [segs.]
1	73
16	21
32	19
64	19
128	18
1024	17
4096	18

Esta tabla nos permite ver que hay una gran mejora de performance con el uso de los buffers, teniendo un pico para el tamaño de buffer 1024. Ahora mostraremos un gráfico en el que se incluyen todos los valores, para tener una vision más general:

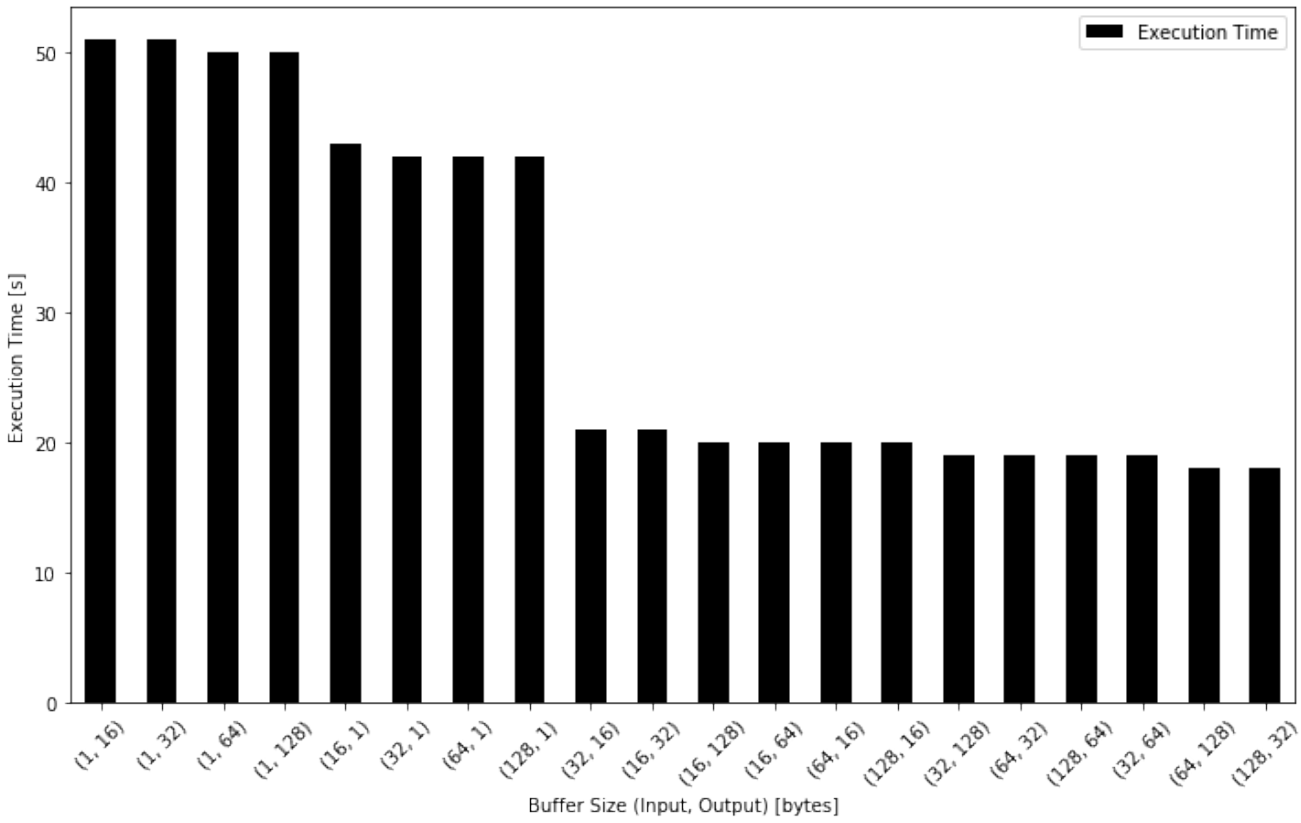


Como se puede observar, la gran distancia entre los valores iniciales (16, 32, ...) y el final (4096) hace que este último gráfico no sea muy descriptivo. Por esta razón, mostraremos uno que toma los valores entre 1 y 128; ya que para ese valor de buffer size, el tiempo ya se está estancado y podremos ver cómo varía de una mejor manera:



Vemos entonces que entre los primeros dos valores (1 y 16) hay un gran salto - más especificamente del 71 % - pero a partir del próximo valor las diferencias ya son mínimas, siendo la máxima de dos segundos (entre 16 y 32).

Finalmente, veremos los datos arrojados por los stress tests en los casos en que los buffers no tenían el mismo tamaño. Para esto, se realizó un gráfico de barras en el que se muestra el tiempo de ejecución del programa para cada par de tamaños de buffers (no se muestra tabla porque su extensión no permitiría observar correctamente los datos):



Este gráfico nos muestra datos interesantes y son los siguiente:

- En las primeras 8 columnas se puede observar que, en nuestro caso, es más importante tener un buffer de entrada que de salida; pues cuando el buffer de entrada toma valor y el buffer de salida tiene tamaño 1 (que es lo mismo que no tenerlo) los tiempos son mejores que en el caso inverso. De todas maneras, se debe decir que en las pruebas realizadas un poco menos de un cuarto de los caracteres que se leen se escriben, por lo

que tiene sentido la diferencia del aproximadamente 20 % que se observa (y si miramos el final del gráfico, el mejor resultado tiene relación $\frac{osize}{isize} = \frac{1}{4}$ [aunque podría no significar nada]).

- Una vez que ambos buffers son de más (o igual a) 16 bytes, el rendimiento es muy parecido en todos los casos.
- No se puede decir que cuanto más grande es el buffer de entrada mejor es el rendimiento ni se puede establecer una relación entre tamaños y rendimientos más que la mencionada en el primer item. Por ejemplo, la cupla (128, 64) es peor que (64, 128) pero (128, 32) es mejor que esta última.

3. Conclusiones

Una conclusión importante obtenida de este trabajo es resultante de las pruebas de Stress; donde se puede ver que la lectura de archivos es una de las tareas más lentas que se pueden realizar pero que es muy fácil de resolver con el simple hecho de realizar lecturas de más de un byte y almacenar la información en un buffer para luego acceder rápidamente, sin volver al archivo. Por otro lado, también es importante destacar que esta mejora tiene un tope y no es 'cuanto más guardo más rápido va', sino que llega un punto (cierto tamaño de buffer) en que ya no se puede obtener mayor performance con este método.

Por otro lado, notamos que es importante conocer la forma en la que se pueden pasar variables de tipo struct en MIPS Assembler, pues es una buena manera de evitar las variables globales que se han utilizado durante este trabajo manteniendo las mismas en una estructura cuyo puntero se pasa como parámetro de las funciones que utilicen dichas variables, modificarlas en las mismas y volver a guardarlas en la estructura.