Trabajo Práctico 0
Infraestructura Básica

[8637/6620] Organización de Computadoras
Curso 2
Segundo Cuatrimestre de 2020

Alumno:	Padrón
BIANCARDI, Julián	103945
CZOP, Santiago Nicolás	104057
OUTEIRO, Sebastián	92108

Índice

1. Introducción

En el siguiente informe se detallarán todos los pasos seguidos para la resolución del trabajo práctico 0 de la materia Organización de Computadoras (66.20). El mismo consiste en codificar archivos de texto a base 64 y el caso inverso, decodificar archivos en base 64 a texto. Los archivos a procesar serán recibidos por entrada estandar (stdin) o bien el nombre del mismo. El resultado de la codificación/decodificación se vera reflejado en otro archivo de texto. Ejecutaremos el mismo programa usando QEMU sobre una maquina virtual MIPS y visualizaremos la salida obtenida.

2. Desarrollo

Para el desarrollo del trabajo se dividio por bloques las tareas a realizar. Por un lado, detectar todos los comandos que el usuario ingresó para poder saber que operaciones ejecutar. Una vez que realizamos esto debemos leer los archivos que nos especificaron en la linea de comando y a medida que vamos leyendo debemos realiar una codificación/decodificación. Por último, los resultados que vayamos obteniendo debemos plasmarlos sobre un archivo de texto distinto del cual estabamos operando.

2.1. Línea de comando

Para parsear la línea de comando se optó por usar la librería "getopt.h" (ver manual: `https://man7.org/linux/man-pages/man3/getopt.3.html`). En específico, se utilizó la función *getopt_long* que nos simplifica el procesamiento de los argumentos, pudiendo así definir una lógica distinta a partir de cada parámetro. El comportamiento se definió en su totalidad en el archivo *args_parser.c*.

2.2. Lectura de Datos

Para una lectura de los archivos de input de forma ordenada y que nos permitiera leer todo su contenido, fuimos por un diseño general que permite obtener pequeños chunks de información y procesarlo. Esta operación la concentra el TDA llamado *file reader* el cual se inicializa de la siguiente manera:

```
int file_reader_init(file_reader_t* self, char* file_name);
```

Donde *file_name* es el nombre del archivo del cual queremos leer. Si este parametro es "stdin", se leerá de entrada estándar.

Luego mediante la siguiente función podemos procesar el archivo:

```
int file_reader_process(file_reader_t* self, file_reader_callback_t callback,
                        void* context);
```

Esta última función mencionada nos será de gran utilidad puesto que podremos realizar distintas operaciones sin acoplarnos al file reader. Esto se debe a que por cada lectura exitosa la función *callback* es invocada delegando asi en el usuario qué hacer con cada chunk leído.

Con este diseño, las funciones *encode* como *decode* tiene un formato que reciben solo un bucket de información el cual deben procesar por cada ciclo. Para ambas funciones se devolverá -1 en caso de un *ERROR*, tras lo cual se frena la ejecución del programa.

2.3. Encode

```
int encode64(char* source, size_t source_len, char* buffer);
```

La función encode64 codificará a base 64 hasta 3 bytes del bucket llamado "source". El resultado de la codificación se verá reflejado en un buffer de tamaño fijo de 4 bytes.

Para realizar esta operación primero debe generase la frase (phrase) de 3 bytes (24 bits), donde cada byte del mismo representa una letra distinta. En el caso de que haya menos de 3 bytes por codificar, el resto de la frase se rellena con ceros.

Una vez generada la frase de 24 bits, se extraen porciones de a 6 bits que son utilizados como índice en la tabla de codificación Base64 para obtener el caracter de salida. Se repite este proceso hasta transformar los 24 bits de la frase. Una vez completado este proceso se añade el carácter '=' al final de la salida si faltaron bytes por codificar. Es decir que si solo se ingresan 2 bytes a codificar se añade un '=', si solo se ingresan 1 byte a codificar se añaden dos '='.

2.4. Decode

```
ssize_t decode64(char* source, char* buffer);
```

Como bien explicamos anteriormente, el decode está diseñado en el formato de recibir un bucket de información denominado "source" el cual se conforma por 4 caracteres codificados en base64 de los cuales tendremos que decodificar en hasta 3 caracteres ASCII y responder los mismos en el parámetro "buffer". Adicionalmente, como respuesta la función devolverá en caso de éxito la cantidad de caracteres escritos en la cadena "buffer".

Para realizar esta operación se optó por el camino de tener una función que nos permitiera obtener el valor en base64 de cada carácter. Luego con dicho valor se realizaba un desplazamiento lógico para ubicar los bits en la posición correspondiente para poder armar la frase de 24 bits que nos permitiera descifrar los 3 caracteres ASCII tomando de a 1 byte de esta frase. Para este último paso se usó un desplazamiento lógico derecho y se pasó por una máscara que filtra para quedarse solo con el byte menos significativo.

2.5. Salida

La salida de las operaciones antes mencionadas serán escritas en un archivo especificado por el comando "-o" (o en su ausencia, en la salida estándar). Para esta operación se optó por implementar un TDA llamado *fwriter* el cual controla las operaciones de escritura, apertura y cierre de archivos.

2.6. Callbacks

A partir de los distintos módulos: *main*, *file reader*, *file writer*, *encode* y *decode*, fue necesario crear funciones que permitieran hacer de "puente". Para esto se diseñaron 2 funciones que cumplieron el rol de callback en los llamados a file reader.

Por un lado, se encuentra *encode_and_output()*, que recibe un vector de caracteres, lo subdivide en bloques de hasta 3 bytes, los envía a ser codificados, y luego los envía ser impresos.

Por otro lado, se encuentra *decode_and_output()*, que recibe un vector de caracteres, lo subdivide en bloques de 4 bytes, los envía a ser decodificados, y por último envía a ser impresos aquellos que haya recibido.

Adicionalmente, resulta importante destacar que estas funciones no controlan el tamaño del vector que reciben, sino que depende del tamaño del buffer del *file reader*. Por lo tanto, para lograr una correcta subdivisión, y por lo tanto, una correcta decodificación o codificación, necesitamos un tamaño apropiado de buffer. Se decidió lógicamente un múltiplo de 3 y 4, para que sea divisible por 3 y 4. Esto generó un acoplamiento fuerte, que podría solucionarse convirtiendo a las funciones que codifican y decodifican en un TDA que tiene "memoria" y conserva los bits sobrantes hasta que le indiquen.

3. Diagramas de secuencia

Se compuso un diagrama de secuencia. Éste intenta mostrar el funcionamiento del programa a partir de una hipotética ejecución con el comando *tp0 -i file.txt -d*.

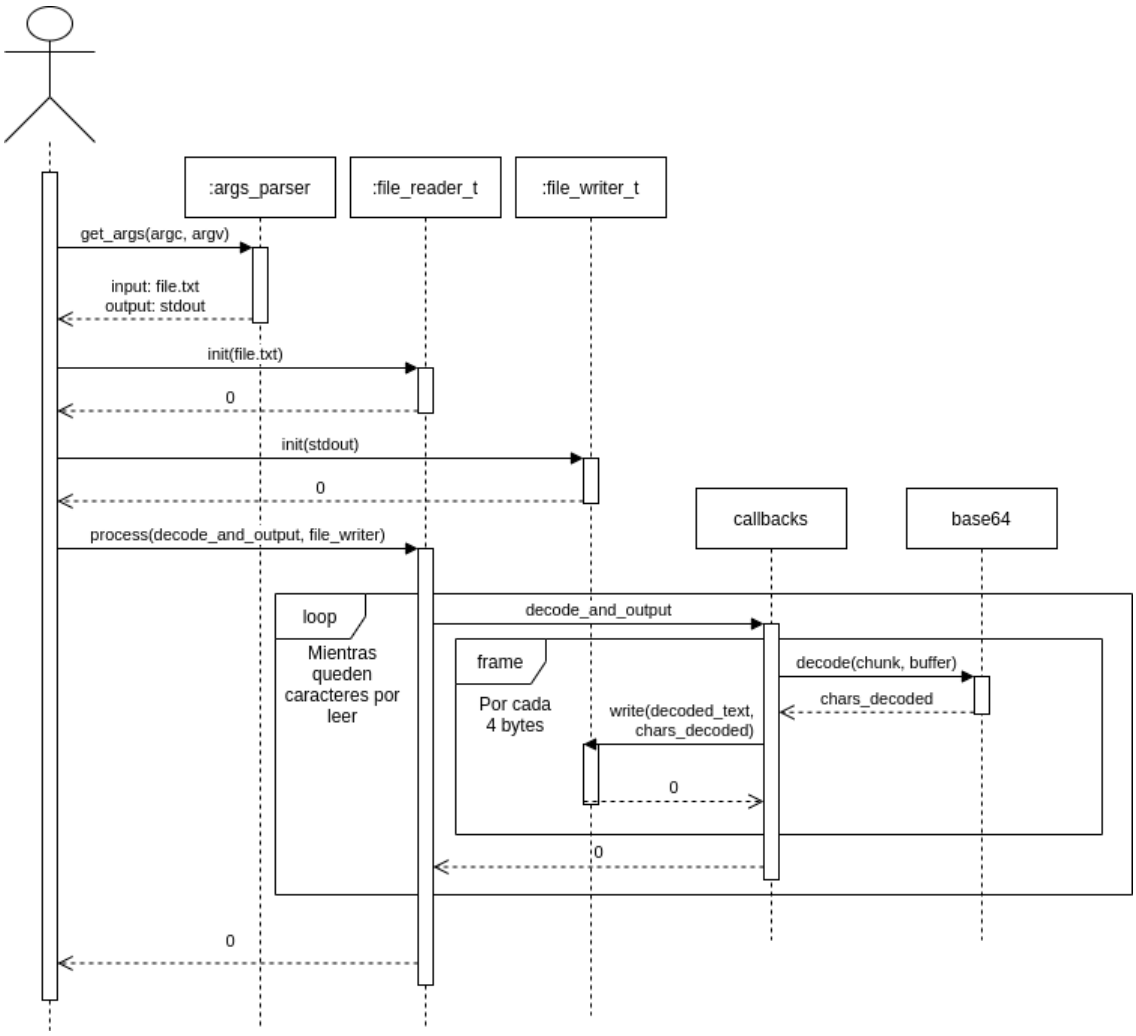

Figura 1: Diagrama de Secuencia para el Comando “tp0 -i file.txt -d”.

4. Ejecución de Pruebas

En el siguiente apartado se mostrarán las pruebas realizadas para probar el codigo del trabajo y garantizar así el correcto funcionamiento en todos los escenarios posibles.

Tanto para la función “encode” como para la función “decode” se hizo un archivo de testing que nos permitió realizar pruebas automatizadas y rápidamente configurables del funcionamiento de cada función. Al ejecutar el mismo podemos ver los siguientes casos de uso:

4.1. Pruebas Encode

En el archivo “encodeTest.c“ podemos encontrar las pruebas realizadas para la función “encode”. En dicha función se le paso un bucket de 3 caracteres ASCII y se generaba el output correspon-

diente de 4 letras codificadas en base 64.

Salida por Consola:

```
Start Testing Encode
Text was: 'Man'. Expected encode is: 'TWFu'. Actual encode result: 'TWFu'
Text was: 'e.'. Expected encode is: 'ZS4='. Actual encode result: 'ZS4='
Text was: 'e'. Expected encode is: 'ZQ=='. Actual encode result: 'ZQ=='
Text was: 'su'. Expected encode is: 'c3U='. Actual encode result: 'c3U='
Text was: 's'. Expected encode is: 'cw=='. Actual encode result: 'cw=='
Tests: OK
```

4.2. Pruebas Decode

En el archivo "decodeTest.c" podemos encontrar las pruebas realizadas para la función "decode". En dicha función se le paso un bucket de 4 letras codificadas en base 64 y se generaba el output correspondiente de hasta 3 caracteres ASCII. Adicionalmente, se realizaron pruebas negativas que buscan generar un error, pasando caracteres inválidos para Base64 o entradas menores a 4 bytes.

Salida por Consola:

```
Start Testing Decode
Input text was: 'TWFu'. Expected text is: 'Man'. Actual text result: 'Man'
Input text was: 'ZQ=='. Expected text is: 'e'. Actual text result: 'e'
Input text was: 'ZS4='. Expected text is: 'e.'. Actual text result: 'e.'
Input text was: 'ab'. Expected result is -1. Actual result is '-1'
Input text was: '!|°"'. Expected result is -1. Actual result is '-1'
Input text was: 'ABC'. Expected result is -1. Actual result is '-1'
Input text was: '[¿.)'. Expected result is -1. Actual result is '-1'
Tests: OK
```

4.3. Pruebas de Integración

Ademas de las pruebas unitarias, realizamos pruebas de integración de todo el trabajo. Estas pruebas ponen a prueba nuestro codigo con diferentes archivos que pueden presentarse como: archivos vacios, archivos de gran magnitud, archivos con caractéres varios, archivos genéricos, etc. A continuación se presentan los casos más relevantes, que fueron utilizados para probar tanto codificación como decodificación.

Caso de Archivo Vacio

Archivo de entrada - *(empty_file_input.txt)*

Archivo de salida - *(empty_file_output.txt)*

Caso de Archivo Provisto por el Enunciado

Codificación

Archivo de entrada - *(mancha_ input.txt)*

En un lugar de La Mancha de cuyo nombre no quiero acordarme

Archivo de salida - *(mancha_ output.txt)*

RW4gdW4gbHVnYXIgZGUgTGEgTWFuY2hhIGRlIGN1eW8gbm9tYnJlIG5vIHF1
aWVybyBhY29yZGFybWUK

Decodificacón

Archivo de entrada - *(mancha_ output.txt)*

RW4gdW4gbHVnYXIgZGUgTGEgTWFuY2hhIGRlIGN1eW8gbm9tYnJlIG5vIHF1
aWVybyBhY29yZGFybWUK
Archivo de salida - *(mancha_ input.txt)*

En un lugar de La Mancha de cuyo nombre no quiero acordarme

Archivos Grandes

Archivo Codificado - *lorem_ ipsum_ output.txt*

Se puede observar el archivo en el repositorio.

Archivo Decodificado - *lorem_ ipsum_ input.txt*

Se puede observar el archivo en el repositorio.

5. Código Fuente

Se puede observar el código implementado en la siguiente dirección, en la carpeta TP0:
`https://github.com/JulianBiancardi/Organizasion-de-Computadoras-66.20`

6. Código MIPS32

El código generado para MIPS32 se encuentra disponible en el mismo repositorio indicado, en la dirección /TP0/qemu.
`https://github.com/JulianBiancardi/Organizasion-de-Computadoras-66.20`

Los comandos usados en la maquina guest fueron los siguientes:

Generación de Archivos Assembly:
gcc -Wall -O0 -S main.c file_ writer.c file_ reader.c base64.c args_ parser.c

Generación del Ejecutable:
gcc -Wall -o tp0 main.s file_ writer.s file_ reader.s base64.s args_ parser.s

Object Dump del Ejecutable:

objdump -S tp0 ≫ tp0_ objdump

7. Conclusiones

En conclusión, se considera que se ha logrado implementar una solución elegante y que cumple con las indicaciones. A lo largo del proceso de diseño, programación y testeo se incorporó mucho conocimiento respecto de la codificación en Base64. En particular, resaltamos que descubrimos que los archivos convertidos desde ASCII a Base64 conllevan un incremento en su tamaño bruto (sin realizar compresiones posteriores), como puede verse en el ejemplo debajo.

```
Archivo Sin Codificar (input):
-rw-rw-r– 1 3,1K oct 21 20:55 lorem_ipsum_input.txt
Archivo Codificado (output):
-rw-rw-r– 1 4,1K oct 21 20:55 lorem_ipsum_output.txt
```

Como se puede observar, pasamos de un archivo de 3.1KB sin codificar, a un archivo codificado de un tamaño de 4.1Kb. Este incremento se debe a que cada 3 bytes obtenemos 4 de Base64, lo cual explica el 33 % de incremento.

También hemos de mencionar que aunque el archivo de assembly se logró generar sin mayores problemas, nos encontramos con instrucciones que resultan extrañas y esperamos ser capaces de comprenderlas eventualmente. El trabajo práctico se coloca a si mismo como un buen punto de partida, dado que es relativamente sencillo y se trata de código que conocemos profundamente debido a que fue implementado por nosotros.

Referencias

[1] Linux.die.net. 2020. *Getopt_Long(3): Parse Options - Linux Man Page.* [online] Disponible en: <https://linux.die.net/man/3/getopt_long>(Accedido 21 Octubre 2020).

[2] En.wikipedia.org. 2020. *Base64.* [online] Disponible en: <https://en.wikipedia.org/wiki/Base64> (Accedido 19 Octubre 2020).