Trabajo Práctico #0: Infraestructura básica

Julian Ferres, *Padrón Nro. 101483*
julianferres@gmail.com

Cecilia María Hortas, *Padrón Nro. 100687*
ceci.hortas@gmail.com

Matías Ezequiel Scakosky, *Padrón Nro. 99627*
scakosky@hotmail.com

2do. Cuatrimestre de 2018
66.20 Organización de Computadoras – Práctica Martes
Facultad de Ingeniería, Universidad de Buenos Aires

Abstract

Se propone como objetivo del Trabajo Práctico 0 familiarizarse con las herramientas de software que serán utilizadas a lo largo de la cursada. Para eso se plantea la realización de un programa en lenguaje C para codificar y decodificar información en base64. Se utiliza el programa GXemul para simular el entorno de desarrollo, una máquina MIPS que corre una versión de NetBSD.

1 Introducción

Se propone escribir un programa en lenguaje C que codifique o decodifique información de base64. Se cuentan con una serie de comandos básicos para el desarrollo del programa que serán detallados en el subtítulo de Implementacion. El objetivo principal del programa consiste en realizar una acción que puede ser codificar o decodificar a partir de un archivo de entrada y generar un archivo de salida. En caso de no recibir los nombres de archivos se utiliza por defecto los streams standar stdin y stdout.

2 Documentación relevante al diseño e implementación del programa

A partir de la consigna se determina que los comandos que debe incluir el programa son:

```
-h, --help Despliega el menú de ayuda
-V, --version Imprime la versión y cierra el programa
```

```
   -i, --input Determina la ubicación del archivo de entrada
   -o, --output Determina la ubicación del archivo de salida
   -a, --action Determina la acción que ejecuta el programa:  decodificar
o codificar
```

2.1 Diseño

El programa principal se encuentra desarrollado en la función `main`. Se detalla todo lo relativo al manejo de los comandos y se utiliza la libería `getopt.h`. Se guarda en el archivo `main.c`

Las acciones de codificar y decodificar se separan en funciones distintas y se implementan siguiendo una lógica similar. Ambas están en el archivo `code.c`

2.2 Detalle de implementación

2.3 Función encode

La función `encode` plantea un problema que radica en leer 6 bits cuando las funciones en C leen de a 1 byte como mínima unidad. Se utiliza la función `fgetc` que lee de a 1 carácter, es decir, 1 byte y se emplean las llamadas máscaras que permiten obtener los bits necesarios para procesar la información. Además, para acomodar los mismos a las posiciones que sean necesarias en cada caso se utilizan los operadores `>>` o `<<` que simbolizan `shift right` o `shift left` respectivamente. Todos los números obtenidos se utilizan como índice de la tabla de b64 para realizar la traducción y se escribe de la misma al archivo de salida o a stdout.

Caso 1: Se leen 8 bits y se desean obtener los primeros 6. Para ello se utiliza el operador `and` entre el carácter obtenido y la máscara `11111100` o `0xFC` y se realiza un shift de dos posiciones a la derecha para que esos 6 bits se localicen en los últimas 6 posiciones.

Caso 2: Se leen los próximos 8 bits. Se forman 6 bits a partir de los 2 bits sobrantes del `Caso 1` y de los 4 primeros bits del carácter leído en este caso. Los 2 bits sobrantes se obtienen con el operador `and` entre el carácter obtenido en el `Caso 1` y la máscara `00000011` o `0x03`. Se realiza un shift de cuatro posiciones a la izquierda ya que deben dejar libres las últimas posiciones para los 4 bits mencionados. Los cuatro restantes se forman a partir del operador `and` entre el carácter leído en este caso y la máscara `11110000` o `0xF0`. Se realiza un shift de cuatro posiciones a la derecha ya que deben ser los últimos 4 bits del número formado. Para armar el número de 6 bits a partir de estas dos descomposiciones se utiliza el operador `|` entre ambas variables.

Caso 3: Se leen los próximos 8 bits. En este caso se formarán dos números. El primero con los 4 bits restantes del `Caso 2` y los primeros 2 bits del carácter leído en este caso. El segundo con los últimos 6 bits del carácter leído en este caso.

Para obtener el primer número se utiliza el operador `|` entre dos variables:

-La primera se forma con los 4 bits restantes del carácter del `Caso 2`. Para ello se utiliza el operador `and` entre dicho carácter y la máscara `00001111` o `0x0F`. Además se utiliza un shift a izquierda de 2 posiciones para dejar libre las posiciones de los 2 carácteres restantes.

-La segunda se forma con los primeros 2 bits del carácter leído en este caso. Para ello se utiliza el operador `and` entre dicho carácter y la máscara `11000000` o `0xC0`. Se realiza un shift a derecha de 6 posiciones para que queden posicionados dichos bits en las últimas posiciones.

Para obtener el segundo número se utiliza el operador `and` entre el carácter leído en este caso y la máscara `00111111` o `0x3F` para obtener los últimos 6 bits.

Al finalizar estos 3 casos se vuelve al `Caso 1` y se renueva el ciclo.

En caso de tener 2 o 4 bits faltantes se rellenan con ceros y se agrega un `=` o un `==` respectivamente al final del texto para señalizar dicho agregado.

2.4 Función decode

La función `decode` plantea el problema de leer un carácter que proviene de la base64. Para traducirlo al número del cual proviene (1 a 64) se utiliza una función auxiliar que devuelve el índice de la tabla formato b64 al que corresponde dicho carácter. Siempre se trabajarán con dichos números y se los tratará como números de 6 bits. Como la lectura es de a 8 bits, los primeros 2 bits serán siempre 0. Se utiliza la función `fgetc` que lee de a 1 carácter, es decir, 1 byte y se emplean al igual que en el `encode` las llamadas máscaras y las operaciones de shift para acomodar los números a las posiciones determinadas en cada caso.

Se distinguen 3 casos:

Caso 1: Se utilizan los 6 bits del número obtenido con la función auxiliar y se traduce el próximo carácter a leer a través de la función auxiliar mencionada de la cual se obtienen también otros 6 bits. De esta manera, el número de 8 bytes se construye con el operador `|` entre los 6 bits del primero y los primeros 2 bits del segundo.

- El primero se forma a partir de un shift a izquiera de 2 posiciones para dejar libre dichas posiciones a los primeros 2 bits del segundo número formado.

- El segundo número se forma a partir del operador `and` entre el segundo indice obtenido a través de la función auxiliar y la máscara `00110000` o `0x30`. Se realiza un shift de 4 posiciones a la derecha para que queden localizados en las dos últimas posiciones.

Caso 2: Se utilizan los 4 bits que sobraron del `Caso 1` y se lee el próximo carácter, del cual se obtiene el índice de la tabla B64 con la función auxiliar mencionada y se utilizan los primeros 4 bits. Se utiliza el operador `|` para unificar ambos números.

- Para obtener el primer número se realiza un shift a izquierda de 4 posiciones para dejar libres las últimas 4 posiciones para el segundo número.

- Para obtener el segundo número se utiliza el operador `and` entre el índice devuelto por la función auxiliar en este caso y la máscara `00111100` o `0x3C`. Se utiliza esa máscara porque los primeros 2 bits siempre son 0, como fue mencionado anteriormente. Para que esos 4 bits se ubiquen en las últimas 4 posiciones se realiza un shift a derecha de valor 2.

Caso 3: Se utilizan los 2 bits sobrantes del `Caso 2` y se lee un nuevo carácter, que se traduce a partir de la función auxiliar mencionada a un nuevo índice entre 0 y 64 del cual se toman los 6 bits del mismo. Ambos números se utilizan con el operador `|`.

- El primer número se obtiene a partir de un shift a izquierda de 6 posiciones del número obtenido en el `Caso 2` a partir de la función auxiliar. De esta manera deja los últimos 6 bits libres para el segundo número.

- El segundo número se obtiene a partir del operador `and` entre el índice obtenido previamente y la máscara `00111111` o `0x3F`.

Al finalizar estos 3 casos se vuelve al `Caso 1` y se renueva el ciclo.

A medida que se fueron obteniendo los distintos números se escribieron en el archivo de salida o en la salida stdout, según corresponda. En caso de encontrar un `=` o `==` significa la finalización de la lectura.

3 Comandos para compilar el programa

Acá debería ponerse lo correspondiente para compilar en linux o netbsd? Sin optimizaciones o con optimizaciones?

4 Corridas de prueba

Qué ponemos? El código que pasó Mati del archivo aleatorio?

5 Código fuente en C

6 Código MIPS32 generado por el compilador

Agregar

7 Conclusiones

Se presentó un modelo para que los alumnos puedan tomar como referencia en la redacción de sus informes de trabajos prácticos.

References

[1] Intel Technology & Research, "Hyper-Threading Technology," 2006, http://www.intel.com/technology/hyperthread/.

[2] J. L. Hennessy and D. A. Patterson, "Computer Architecture. A Quantitative Approach," 3ra Edición, Morgan Kaufmann Publishers, 2000.

[3] J. Larus and T. Ball, "Rewriting Executable Files to Mesure Program Behavior," Tech. Report 1083, Univ. of Wisconsin, 1992.