

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 Implementación. 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 librería `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 caracteres 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 señalar 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 izquierda 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 índice 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

En esta sección se detallan los pasos para compilar el programa en NetBSD a partir del entorno proporcionado por GXemul.

- Desde el directorio donde se instaló GXemul se corre el siguiente comando para bootear la imagen del disco patrón: `hostOS ./gxemul -e 3max -d netbsd-pmax.img`
- Desde otra consola de linux se crea en el host OS con el usuario root un alias para la interfaz loopback (lo:0) con la IP 172.20.0.1 con el siguiente comando: `hostOS ifconfig lo:0 172.20.0.1`

- Luego se ejecutan los siguientes comandos para la conexión contra la interfaz creada:

```
hostOS export TERM=xterm
hostOS ssh -p 2222 root@127.0.0.1
```

- Se transfieren los archivos a compilar a NetBSD con los siguientes comandos:

```
scp -P2222 -r TP0 root@127.0.0.1:/root/TP0NetBSD
```

- Luego se ejecutan los siguientes comandos para realizar la compilación y extraer el código MIPS generado por el compilador en el sistema operativo que corre sobre GXemul:

```
root@: ls
root@: pwd
root@: mkdir TP0NetBSD
root@: cd TP0NetBSD
root@: /TP0NetBSD/TP0 gcc -Wall -O0 -S -mrnames *.c
```

- **-s** para detener el compilador luego de generar el código assembly.
- **-mrnames** para cambiar los números de los registros por sus nombres de convención

- Luego se transfiere el archivo .s para el sistema operativo sobre el cual corre GXemul:

```
hostOS scp -P2222 root@127.0.0.1:/root/TP0NetBSD/TP0/*.s /home/user
```

4 Corridas de prueba

Las pruebas realizadas se basaron en los ejemplos del enunciado. Se probaron los comandos básicos como `-h` y `-V` para probar que muestren el resultado esperado.

Luego, para probar los comandos `-a action -i input -o output` se realizó lo siguiente:

- Se probó que de omitir esos comandos la acción por default sea la ejecución de `encode` con la entrada por `stdin` y la salida generada por `stdout`.
- Se probó que de omitir nombres de archivos de `input` y `output` los archivos tomados por default eran los *stream* estándar.
- Se probó que de recibir los nombres de archivos de entrada y salida las acciones esperadas se concretaban. Se tomaron los ejemplos mencionados en el enunciado.
- Finalmente se ejecutó el siguiente comando en la terminal para verificar que archivos de tamaño creciente codificaban y decodificaban correctamente: `n=1;while ;; do head -c n </dev/urandom >/tmp/in.bin; ./main -a encode -i /tmp/in.bin -o /tmp/out.b64; ./main -a decode -i /tmp/out.b64 -o /tmp/out.bin; if diff /tmp/in.bin /tmp/out.bin; then ;; else echo ERROR: n; break; fi; echo ok: n; n=$((n+1)); rm -f /tmp/in.bin /tmp/out.b64 /tmp/out.bin; done`

5 Código fuente en C

6 Código MIPS32 generado por el compilador

6.1 main.s

```
.file      1 "main.c"
.section  .mdebug.abi32
.previous
.abicalls
.globl     HELP
.rdata
.align     2
.type      HELP, @object
.size      HELP, 231

HELP:
.ascii     "Usage:\n"
.ascii     " tp0 -h \n"
.ascii     " tp0 -V \n"
.ascii     " tp0 [options] \n"
.ascii     " Options: \n"
.ascii     " -V, --version Print version and quit. \n"
.ascii     " -h, --help Print this information. \n"
.ascii     " -i, --input Location of the input file. \n"
```

```

        .ascii      "-a, --action Program action: encode (default) or decode"
        .ascii      ". \n\000"
        .globl      VERSION
        .align      2
        .type       VERSION, @object
        .size       VERSION, 12
VERSION:
        .ascii      "2018.9.18 \n\000"
        .align      2
$LC0:
        .ascii      "version\000"
        .align      2
$LC1:
        .ascii      "help\000"
        .align      2
$LC2:
        .ascii      "input\000"
        .align      2
$LC3:
        .ascii      "output\000"
        .align      2
$LC4:
        .ascii      "action\000"
        .data
        .align      2
        .type       long_options.0, @object
        .size       long_options.0, 96
long_options.0:
        .word       $LC0
        .word       0
        .word       0
        .word       0
        .word       $LC1
        .word       0
        .word       0
        .word       0
        .word       $LC2
        .word       2
        .word       0
        .word       0
        .word       $LC3
        .word       2
        .word       0
        .word       0
        .word       $LC4
        .word       2
        .word       0
        .word       0
        .word       0
        .word       0

```

```

        .word      0
        .word      0
        .rdata
        .align     2
$LC5:
        .ascii     "Vha:i:o:\000"
        .align     2
$LC6:
        .ascii     "encode\000"
        .align     2
$LC7:
        .ascii     "decode\000"
        .align     2
$LC8:
        .ascii     "r\000"
        .align     2
$LC9:
        .ascii     "File not found \n\000"
        .align     2
$LC10:
        .ascii     "w\000"
        .align     2
$LC11:
        .ascii     "File Error \n\000"
        .text
        .align     2
        .globl     main
        .ent       main
main:
        .frame      $fp,80,$ra
        .mask       0xd0000000,-8
        .fmask      0x00000000,0
        .set        noreorder
        .cplload    $t9
        .set        reorder
        subu        $sp,$sp,80
        .cprestore  24
        sw          $ra,72($sp)
        sw          $fp,68($sp)
        sw          $gp,64($sp)
        move        $fp,$sp
        sw          $a0,80($fp)
        sw          $a1,84($fp)
        la          $v0,__sF
        sw          $v0,36($fp)
        la          $v0,__sF+88
        sw          $v0,40($fp)
        sw          $zero,44($fp)
        addu        $v0,$fp,44
        sw          $v0,16($sp)
# vars= 32, regs= 3/0, args= 24, extra= 8

```

```

        lw      $a0,80($fp)
        lw      $a1,84($fp)
        la      $a2,$LC5
        la      $a3,long_options.0
        la      $t9,getopt_long
        jal     $ra,$t9
        sw      $v0,32($fp)
        lw      $v1,32($fp)
        li      $v0,-1                                # 0xffffffffffffffff
        bne     $v1,$v0,$L20
        b       $L19
$L20:
        lw      $v0,32($fp)
        sw      $v0,56($fp)
        li      $v0,104                                # 0x68
        lw      $v1,56($fp)
        beq     $v1,$v0,$L22
        lw      $v1,56($fp)
        slt     $v0,$v1,105
        beq     $v0,$zero,$L36
        li      $v0,86                                # 0x56
        lw      $v1,56($fp)
        beq     $v1,$v0,$L23
        lw      $v1,56($fp)
        slt     $v0,$v1,87
        beq     $v0,$zero,$L37
        lw      $v0,56($fp)
        beq     $v0,$zero,$L33
        b       $L21
$L37:
        li      $v0,97                                # 0x61
        lw      $v1,56($fp)
        beq     $v1,$v0,$L24
        b       $L21
$L36:
        li      $v0,105                                # 0x69
        lw      $v1,56($fp)
        beq     $v1,$v0,$L27
        li      $v0,111                                # 0x6f
        lw      $v1,56($fp)
        beq     $v1,$v0,$L30
        b       $L21
$L22:
        la      $a0, __sF+88
        la      $a1,HELP
        la      $t9,fprintf
        jal     $ra,$t9
        sw      $zero,52($fp)
        b       $L17
$L23:

```



```

        la      $a0, __sF+88
        la      $a1, VERSION
        la      $t9, fprintf
        jal     $ra, $t9
        sw      $zero, 52($fp)
        b       $L17
$L24:
        lw      $a0, optarg
        la      $a1, $LC6
        la      $t9, strcmp
        jal     $ra, $t9
        bne     $v0, $zero, $L25
        li      $v0, 1
        sb      $v0, 48($fp)
        # 0x1
$L25:
        lw      $a0, optarg
        la      $a1, $LC7
        la      $t9, strcmp
        jal     $ra, $t9
        bne     $v0, $zero, $L27
        sb      $zero, 48($fp)
$L27:
        lw      $v0, 80($fp)
        slt     $v0, $v0, 5
        bne     $v0, $zero, $L30
        lw      $v0, 84($fp)
        addu    $v0, $v0, 16
        lw      $a0, 0($v0)
        la      $a1, $LC8
        la      $t9, fopen
        jal     $ra, $t9
        sw      $v0, 36($fp)
        lw      $v0, 36($fp)
        bne     $v0, $zero, $L30
        la      $a0, __sF+176
        la      $a1, $LC9
        la      $t9, fprintf
        jal     $ra, $t9
$L30:
        lw      $v0, 80($fp)
        slt     $v0, $v0, 7
        bne     $v0, $zero, $L21
        lw      $v0, 84($fp)
        addu    $v0, $v0, 24
        lw      $a0, 0($v0)
        la      $a1, $LC10
        la      $t9, fopen
        jal     $ra, $t9
        sw      $v0, 40($fp)
        lw      $v0, 40($fp)

```

```

        bne      $v0,$zero,$L21
        la       $a0, __sF+176
        la       $a1,$LC11
        la       $t9,fprintf
        jal      $ra,$t9
        b        $L21
$L33:
        la       $t9,abort
        jal      $ra,$t9
$L21:
        lbu      $v0,48($fp)
        beq      $v0,$zero,$L38
        lw       $a0,36($fp)
        lw       $a1,40($fp)
        la       $t9,encode
        jal      $ra,$t9
        b        $L39
$L38:
        lw       $a0,36($fp)
        lw       $a1,40($fp)
        la       $t9,decode
        jal      $ra,$t9
$L39:
        lw       $a0,36($fp)
        la       $t9,fclose
        jal      $ra,$t9
        lw       $a0,40($fp)
        la       $t9,fclose
        jal      $ra,$t9
        sw       $zero,52($fp)
        b        $L17
$L19:
        lw       $a0,36($fp)
        lw       $a1,40($fp)
        la       $t9,encode
        jal      $ra,$t9
        sw       $zero,52($fp)
$L17:
        lw       $v0,52($fp)
        move     $sp,$fp
        lw       $ra,72($sp)
        lw       $fp,68($sp)
        addu     $sp,$sp,80
        j        $ra
        .end     main

```

6.2 code.s

```
.file      1 "code.c"
.section   .mdebug.abi32
.previous
.abicalls
.globl     B64
.rdata
.align     2
.type      B64, @object
.size      B64, 64

B64:
.byte      65
.byte      66
.byte      67
.byte      68
.byte      69
.byte      70
.byte      71
.byte      72
.byte      73
.byte      74
.byte      75
.byte      76
.byte      77
.byte      78
.byte      79
.byte      80
.byte      81
.byte      82
.byte      83
.byte      84
.byte      85
.byte      86
.byte      87
.byte      88
.byte      89
.byte      90
.byte      97
.byte      98
.byte      99
.byte      100
.byte      101
.byte      102
.byte      103
.byte      104
.byte      105
.byte      106
.byte      107
```

.byte	108
.byte	109
.byte	110
.byte	111
.byte	112
.byte	113
.byte	114
.byte	115
.byte	116
.byte	117
.byte	118
.byte	119
.byte	120
.byte	121
.byte	122
.byte	48
.byte	49
.byte	50
.byte	51
.byte	52
.byte	53
.byte	54
.byte	55
.byte	56
.byte	57
.byte	43
.byte	47
.ident	"GCC: (GNU) 3.3.3 (NetBSD nb3 20040520)"

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 Measure Program Behavior,” Tech. Report 1083, Univ. of Wisconsin, 1992.