66.20 Organización de Computadoras
Trabajo Práctico 0:
Infraestructura básica

Burdet Rodrigo, *Padrón Nro. 93440*
rodrigoburdet@gmail.com

Romani Nazareno, *Padrón Nro. 83991*
nazareno.romani@gmail.com

Martinez Gaston Alberto, *Padrón Nro. 91383*
gaston.martinez.90@gmail.com

1er. Cuatrimestre de 2014
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

3 de abril de 2014

1. Objetivos

Familiarizarse con las herramientas de software que usaremos en los siguientes trabajos, implementando un programa (y su correspondiente documentación) que resuelva el problema piloto que presentaremos mas abajo.

2. Resumen

En el presente trabajo, se implementó un algoritmo que resuelve la transformación de un conjunto arbitrario de bytes en un conjunto formado por caracteres ASCII y viceversa. Los distintos valores a codificar/decodificar, son obtenidos a través de los parámetros definidos en el enunciado. El programa fue compilado tanto en la máquina host (sistema operativo Linux), como en una máquina corriendo el sistema operativo NetBSD.

3. Desarrollo

3.1. Paso 1: Configuración de Entorno de Desarrollo

El primer paso fue configurar el entorno de desarrollo, de acuerdo a la guía facilitada por la cátedra. Trabajamos con una distribución de Linux basada en Debian y con el GxEmul proporcionado por la cátedra, el cual tiene ya configurado NetBSD.

3.2. Paso 2: Implementación del programa

El programa debe ejecutarse por línea de comando y la salida del mismo dependerá del valor de los argumentos con los que se lo haya invocado.

3.2.1. Ingreso de parámetros

El formato para invocar al programa es el siguiente:

```
./tp0 [OPTIONS]
```

Los parámetros válidos que puede recibir el programa son los siguientes:

-e,	**–encode**	`(Encodes to Base64).`
-d,	**–decode**	`(Decodes from Base64).`
-i,	**–input file**	`(Reads from file or stdin).`
-o,	**–output file**	`(Writes to file or stdout).`
-v,	**–version**	`(Show version string).`
-h,	**–help**	`(Print this message and quit).`

3.2.2. Interpretación de parámetros

Para parsear los parámetros se usaron las funciones definidas en arg_parse.h. Se puede conocer más en detalle el funcionamiento de las mismas, a través de la documentación incluida en dicho archivo. Estas funciones permiten recoger los parámetros de entrada del programa y ejecutar la funcionalidad correspondiente. Estas son compatibles con NetBSD.

4. Compilación del programa

Para poder compilar el proyecto, se debe abrir una terminal Linux dentro del directorio donde se encuentra el código fuente escrito en C, y ejecutar el script *Makefile* con el comando `make` [1]. Este comando ejecuta las directivas definidas en el archivo `Makefile` generado para tal caso.

[1] Requiere tener instalado el programa *Make y el compilador GCC*

Esto generara un archivo ejecutable, llamado *tp0* [2]. Tambien se puede ejecutar el comando `make Valgrind` para compilar el programa y correrlo con *Valgrind*, de manera de poder depurarlo en modo interactivo.

5. Corridas de prueba y Mediciones

En las figuras que siguen a continuación se muestran los comandos utilizados para ejecutar el programa y se puede apreciar los resultados de las diferentes pruebas que realizamos. Cabe acotar que cuando se encodea un archivo(`./tp0 -e -i <archivo>`), también se encodea el fin del mismo.

[2]El nombre del ejecutable se puede editar desde el script o desde la consola al invocar `make`

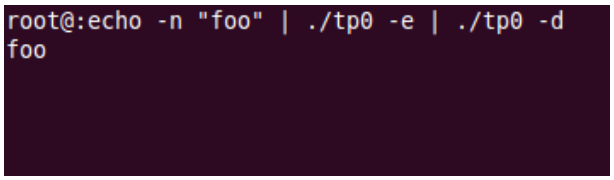

Figura 1: Codificación de texto ‘foo‘

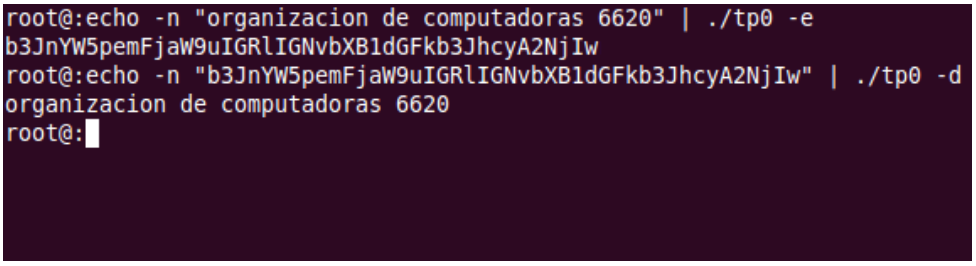

Figura 2: Codificación/Decodificación de texto ‘organizacion de computadoras 6620‘

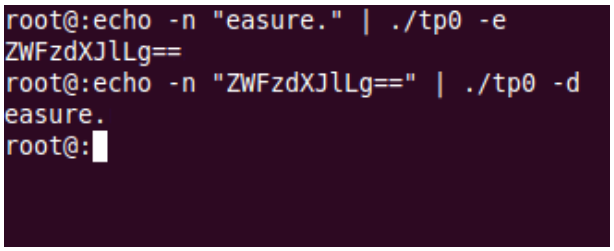

Figura 3: Codificación/Decodificación de texto con uso de caracter de padding

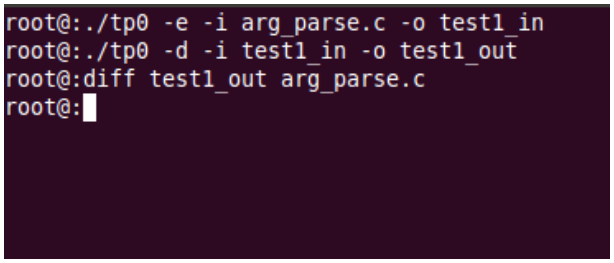

Figura 4: Codificación a través de archivo de entrada guardando la salida en otro archivo, decodificación de este último y comparación de archivos

A. Codigo C

Archivo 1: ../arg_parse.h

```c
#ifndef __ARG_PARSE_H__
#define __ARG_PARSE_H__
#include <stdlib.h>

typedef struct TParseArg TParseArg;

/** Funcion para parsear un argumento.
 * @param char*: cadena de caracteres de argumento a parsear.
 * @return void*: valor ya parseado.
 */
typedef void* (*TParseArgFunc)(char*);

/** Crea un nuevo parseador de argumentos.
 * @param int: cantidad maxima de argumentos que se pueden llegar a ingresar.
 * @return TParseArg*: TDA de parseador de argumentos.
 */
TParseArg* ParseArg_new(int c);

/** Agrega un nuevo argumento a parsear.
 * Se copian el nombre largo y el valor por defecto y se los guarda internamente.
 * @param TParseArg*: this, instancia de TDA.
 * @param TParseArgFunc: func, funcion para parsear el str de argv y devolver un valor. NULL si es flag (no tiene valor, solo importa si esta el parametro).
 * @param char: corto, nombre corto del argumento
 * @param char*: largo, nombre largo del argumento
 * @param void*: defecto, valor por defecto
 * @param size_t: tam, tama~no (memoria) del valor por defecto.
 * @return int: 0 ok, resto error
 */
int ParseArg_addArg(TParseArg* this, TParseArgFunc func, char corto, char* largo, void* defecto, size_t tam);

/** Parsea argumentos.
 * @param TParseArg*: this, instancia de TDA.
 * @param int: argc, cantidad de argumentos.
 * @param char*[]: argv, lista de cadenas de caracteres de argumentos.
 * @return int: 0 ok, resto error
 */
int ParseArg_parse(TParseArg* this, int argc, char* argv[]);

/** Devuelve el valor de un argumento.
 * Si es un flag, se devolvera un 1, se debe setear el valor por defecto al agregar los flags a 0,
 * para el resto de los valores (cuando no es flag), se debe liberar el puntero devuelto!
 * @param TParseArg*: this, instancia de TDA.
 * @param char: corto, nombre corto del parametro.
 * @return void*: puntero al valor, NULL para error o qe no este.
 */
```

```
void* ParseArg_getArg(TParseArg* this, char corto);

/** Destruye el tda.
 * @param TParseArg*: this, instancia de TDA.
 * @return int: 0 ok, resto error
 */
int ParseArg_delete(TParseArg*);

/** Funcion usada como TParseArgFunc para cunado el arguemnto es una cadena
    de caracteres */
void *ParseArg_parseStr(char*);

#endif
```

Archivo 2: ../arg_parse.c

```
#include "arg_parse.h"
#include <string.h>
#include <stdio.h>

typedef struct {
    TParseArgFunc func;
    char nombre_corto;
    char *nombre_largo;
    void* def;
    size_t def_tam;
    int encontre;
    char* buf;
} TArg;

struct TParseArg {
    int c_max, c;
    TArg *args;
};

TParseArg* ParseArg_new(int c){
    TParseArg* this = NULL;
    if(!c)
        return NULL;

    this = (TParseArg*) calloc(1, sizeof(TParseArg));
    this->c_max = c;

    this->args = (TArg*) calloc(c, sizeof(TArg));

    return this;
}

int ParseArg_addArg(TParseArg* this, TParseArgFunc func, char corto, char*
    largo, void* defecto, size_t tam){
    if(!this || this->c >= this->c_max)
        return 1;

    this->args[this->c].func = func;
    this->args[this->c].nombre_corto = corto;
```

```
39      this->args[this->c].nombre_largo = strcpy((char*) calloc(strlen(largo)
            +1, sizeof(char)), largo);
        if(tam && defecto){
41          this->args[this->c].def = calloc(1, tam);
            memcpy(this->args[this->c].def, defecto, tam);
43          this->args[this->c].def_tam = tam;
        }
45
        this->c++;
47      return 0;
    }
49
    /** Devuelve el puntero al argumento si lo encuentra en la lsita, o NULL
51   */
    TArg* encontrar_argumento_corto(TParseArg* this, char c){
53      int i=0;
        for(i=0; i < this->c; i++){
55          if(this->args[i].nombre_corto == c)
                return &(this->args[i]);
57      }

59      return NULL;
    }
61
    /** Devuelve el puntero al argumento si lo encuentra en la lsita, o NULL
63   */
    TArg* encontrar_argumento_largo(TParseArg* this, char* largo){
65      int i=0;
        for(i=0; i < this->c; i++){
67          if(strcmp(this->args[i].nombre_largo, largo) == 0)
                return &(this->args[i]);
69      }

71      return NULL;
    }
73
    int ParseArg_parse(TParseArg* this, int argc, char* argv[]){
75      int i=0;
        if(!this)
77          return 1;

79      for(i=1; i < argc; i++){
            TArg* arg = NULL;
81          if(argv[i][0] == '-'){
                if(argv[i][1] == '-')
83                  arg = encontrar_argumento_largo(this, argv[i]+2);
                else
85                  arg = encontrar_argumento_corto(this, argv[i][1]);

87              if(!arg)
                    continue;
89
                arg->encontre = 1;
91
                if(arg->func == NULL) // Es flag
```

```
93                  continue;

95              if(i+1 < argc){
                    i++;
97                  arg->buf = strcpy((char*) calloc(strlen(argv[i])+1, sizeof(
                        char)), argv[i]);
                }
99          }
        }
101
        return 0;
103 }

105 void* ParseArg_getArg(TParseArg* this, char corto){
        TArg* arg = NULL;
107     if(!this)
            return NULL;
109
        arg = encontrar_argumento_corto(this, corto);
111
        if(!arg)
113         return NULL;

115     if(arg->encontre){
            if(arg->func == NULL){
117             return (void*) 1;
            }
119
            if(arg->buf == NULL)
121             return NULL;

123         return arg->func(arg->buf);
        }
125
        return arg->def;
127 }


129
    int ParseArg_delete(TParseArg* this){
131     int i=0;
        if(!this)
133         return 1;

135     for(i=0; i < this->c; i++){
            if(this->args[i].nombre_largo)
137             free(this->args[i].nombre_largo);
            if(this->args[i].def)
139             free(this->args[i].def);
            if(this->args[i].buf)
141             free(this->args[i].buf);
        }
143     free(this->args);
        free(this);
145
        return 0;
```

```
147 }

149 void *ParseArg_parseStr(char* str){
        return strcpy((char*) calloc(strlen(str)+1, sizeof(char)), str);
151 }
```

Archivo 3: ../base_64.h

```
  /*
2  * base_64.h
   */
4 #include <stdio.h>
  #include <stdbool.h>
6
  #define ALFABETO "
      ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
8 #define PADDING_CHAR '='

10 #define ENCODE_ERROR 3
  #define DECODE_ERROR 4
12 #define WRITE_ERROR  5

14 /**
    * Escribe sobre el output_stream el contenido del
16  * input_stream codificado en base 64
    */
18 int encode(FILE* input_stream, FILE* output_stream);

20 /**
    * Escribe sobre el output_stream el contenido del
22  * input_stream decodificado de base 64.
    * Pre: El input_stream contiene solo caracteres del
24  *       "DICCIONARIO"
    */
26 int decode(FILE* input_stream, FILE* output_stream);
```

Archivo 4: ../base_64.c

```
  /*
2  * base_64.c
   */
4 #include <stdlib.h>
  #include <stdint.h>
6 #include <string.h>

8 #include "base_64.h"

10 static bool little_endian = true;

12 /**
    * Setea la variable que indica si el sistema es
14  * little o big endian
    */
16 void set_endianess(void) {
       int i = 1;
18     char *p = (char *) &i;
```

```
    if (p[0] == 1)
        little_endian = true;
    else
        little_endian = false;
}

/**
 * Invierte las posiciones de un arreglo de 4 bytes
 */
void array_invert(char temporal_array[4]) {
    char aux = 0;
    aux = temporal_array[0];
    temporal_array[0] = temporal_array[3];
    temporal_array[3] = aux;
    aux = temporal_array[1];
    temporal_array[1] = temporal_array[2];
    temporal_array[2] = aux;
}

/**
 * Transforma el contenido del arreglo 'input' en
 * una cadena con la codificacion de este en base
 * 64. El resultado se almacena en el arreglo 'output'.
 * Devuelve true si la operacion fue exitosa.
 */
bool encode_to_base64(char input[3], char output[4]) {
    int32_t temporal = 0;
    int32_t index = 0;
    int i;

    char* temporal_array = (char*) &temporal;
    char* index_array = (char*) &index;

    memcpy(&temporal, input, 3);

    if (little_endian)
        array_invert(temporal_array);

    temporal = temporal >> 8;

    for (i = 0; i < 4; i++) {
        temporal = temporal << 6;

        if (little_endian) {    //Little endian
            index_array[0] = temporal_array[3];
            temporal_array[3] = 0;
        } else {                //Big endian
            index_array[3] = temporal_array[0];
            temporal_array[0] = 0;
        }

        output[i] = ALFABETO[index];
    }

    return true;
```

```
74 }

76 /**
    * Devuelve la posicion del caracter 'c' dentro
78  * de la cadena 'string'. -1 en caso que el caracter
    * no pertenezca.
80  */
   int index_of(char c, char* string, size_t len) {
82     int index = -1;
       size_t i;
84
       for (i = 0; i < len; i++) {
86         if (c == string[i]) {
               index = i;
88             break;
           }
90     }

92     return index;
   }
94
   /**
96  * Decodifica el contenido del arreglo 'input' en
    * codificado en base 64. El resultado se almacena
98  * en el arreglo 'output'. Devuelve true si la
    * operacion fue exitosa.
100 */
   bool decode_from_base64(char input[4], char output[3], int* padding) {
102    int32_t temporal = 0;
       int i;
104    *padding = 0;
       char* temporal_array = (char*) &temporal;
106
       for (i = 3; i >= 0; i--) {
108        int index = index_of(input[i], ALFABETO, 64); //TODO: largo de la
               base
           if (index < 0) {
110            if (input[i] != PADDING_CHAR)
                   return false;
112            *padding += 1;
               if (little_endian)
114                temporal_array[3] = 0;
               else
116                temporal_array[0] = 0;

118        } else {
               if (little_endian)
120                temporal_array[3] = index;
               else
122                temporal_array[0] = index;
           }
124        temporal = temporal >> 6;
       }
126
       if (little_endian)
```

```
128         array_invert(temporal_array);

130     memcpy(output, (temporal_array + 1), 3);
        return true;
132 }

134 /**
     * Agrega los caracteres de padding necesarios segun la
136  * cantidad de bytes leidos
     */
138 void add_padding(int bytes_read, char buffer[4]) {
        if (bytes_read < 3) {
140         if (bytes_read < 3)
                buffer[3] = PADDING_CHAR;
142
            if (bytes_read < 2)
144             buffer[2] = PADDING_CHAR;
        }
146 }

148 /**
     * Escribe sobre el output_stream el contenido del
150  * input_stream codificado en base 64
     */
152 int encode(FILE* input_stream, FILE* output_stream) {
        char input_buffer[3];
154     char output_buffer[4];

156     set_endianess();

158     int bytes_read = fread(input_buffer, sizeof(char), 3, input_stream);
        while (bytes_read > 0) {
160         if (bytes_read < 3)
                input_buffer[2] = 0;
162         if (bytes_read < 2)
                input_buffer[1] = 0;
164
            if (!encode_to_base64(input_buffer, output_buffer))
166             return ENCODE_ERROR;

168         add_padding(bytes_read, output_buffer);

170         int bytes_wrote = fwrite(output_buffer, sizeof(char), 4,
                output_stream);

172         if (bytes_wrote != 4)
                return WRITE_ERROR;
174
            bytes_read = fread(input_buffer, sizeof(char), 3, input_stream);
176     }

178     return 0;
    }
180
    /**
```

```
182  * Escribe sobre el output_stream el contenido del
     * input_stream decodificado de base 64.
184  * Pre: El input_stream contiene solo caracteres del
     *      "ALFABETO"
186  */
   int decode(FILE* input_stream, FILE* output_stream) {
188     char input_buffer[4];
        char output_buffer[3];
190
        set_endianess();
192
        int padding = 0;
194     int bytes_read = fread(input_buffer, sizeof(char), 4, input_stream);
        while (bytes_read > 0) {
196         if (bytes_read < 4) //No es la cantidad correcta de bytes
                return ENCODE_ERROR; //TODO: cambiar tipo de error
198
            if (!decode_from_base64(input_buffer, output_buffer, &padding))
200             return DECODE_ERROR;

202         int bytes_to_write = 3;

204         if (padding != 0) {
                if (padding == 1) {
206                 bytes_to_write = 2;
                }
208             if (padding == 2) {
                    bytes_to_write = 1;
210             }
            }
212
            int bytes_wrote = fwrite(output_buffer, sizeof(char),
               bytes_to_write,
214                 output_stream);

216         if (bytes_wrote != bytes_to_write)
                return WRITE_ERROR;
218
            bytes_read = fread(input_buffer, sizeof(char), 4, input_stream);
220     }

222     return 0;

224 }
```

Archivo 5: ../main.c

```
   #include <stdio.h>
2  #include <stdlib.h>

4  #include "base_64.h"
   #include "arg_parse.h"
6
   void usage();
8  void version(char* nombre);
```

```
10 int main(int argc, char* argv[]){
       TParseArg* args;
12     char *output = NULL;
       char *input = NULL;
14     int* res = NULL;
       FILE* inFile;
16     FILE* outFile;

18     // Creo el parseador de argumentos
       args = ParseArg_new(6);
20     // Agrego los argumentos a parsear, si uso valores por defecto como
           NULL con tama~no 0,
       // estoy haciendo qe sean obligatorios los argumentos
22     ParseArg_addArg(args, NULL, 'h', "help", NULL, 0);
       ParseArg_addArg(args, NULL, 'v', "version", NULL, 0);
24     ParseArg_addArg(args, NULL, 'e', "encode", NULL, 0);
       ParseArg_addArg(args, NULL, 'd', "decode", NULL, 0);
26     ParseArg_addArg(args, &ParseArg_parseStr, 'o', "output", NULL, 0);
       ParseArg_addArg(args, &ParseArg_parseStr, 'i', "input", NULL, 0);
28     ParseArg_parse(args, argc, argv);

30     if(ParseArg_getArg(args, 'h')){
           usage();
32         ParseArg_delete(args);
           return 0;
34     }

36     if(ParseArg_getArg(args, 'v')){
           version(argv[0]);
38         ParseArg_delete(args);
           return 0;
40     }
       input = (char*) ParseArg_getArg(args, 'i');
42     output = (char*) ParseArg_getArg(args, 'o');

44     if(input == NULL){
           inFile = stdin;
46     }else{
           inFile = fopen(input, "r");
48         if(!inFile){
               free(input);
50             ParseArg_delete(args);
               return 1;
52         }
       }
54
       if(output == NULL){
56         outFile = stdout;
       }else{
58         outFile = fopen(output, "wb");
           if(!outFile){
60             free(output);
               ParseArg_delete(args);
62             return 1;
           }
```

```
64      }
66      if(ParseArg_getArg(args, 'e')){
            encode(inFile,outFile);
68          ParseArg_delete(args);
            return 0;
70      }
72      if(ParseArg_getArg(args, 'd')){
            decode(inFile,outFile);
74          ParseArg_delete(args);
            return 0;
76      }
78      if(outFile != stdout)
            fclose(outFile);
80
        if(inFile != stdin)
82          fclose(inFile);
84      free(res);
        free(output);
86      free(input);
        ParseArg_delete(args);
88
        return 0;
90  }
92  void version(char* nombre){
        printf("%s 1.0.0\n", nombre);
94  }
96  void usage(){
        printf("OPTIONS:\n");
98      printf("-d --decode Decodes from Base64\n");
        printf("-i --input file Reads from file or stdin\n");
100     printf("-o --output file Writes to file or stdout\n");
        printf("-v --version Show version string\n");
102     printf("-h --help Print this message and quit\n");
    }
```

B. Makefile

```
CC:=gcc
CFLAGS:= -std=c99 -Wextra -pedantic -pedantic-errors -O3 -DNDEBUG -ggdb -
    DDEBUG -fno-inline -Wall #-Werror
VFLAGS:= --track-origins=yes --leak-check=full --trace-children=yes --show-
    reachable=yes -v

RM:= rm -fr
EXEC:= tp0

.PHONY: clean all

all: $(EXEC)

$(EXEC): main.o base_64.o arg_parse.o
	$(CC) $(CFLAGS) $^ -o $@

valgrind: $(EXEC)
	valgrind $(VFLAGS) ./$(EXEC) -e

clean:
	- $(RM) *.o $(EXEC)
```

C. Conclusiones

Como se enuncia en el objetivo de este trabajo práctico, aprendimos a instalar y manejar el GxEmul, a realizar transferencias de archivos en Linux, así como también compilar y ejecutar programas en el NetBSD. Por otro lado, aprendimos a manejar y escribir informes en LaTeX. De este modo, estamos preparados para que en los próximos trabajos prácticos, nos aboquemos directamente al desarrollo de los mismos.