Infraestructura básica

66:20 Organización de Computadoras

Trabajo práctico 0

Axel Lijdens (95772)
Eduardo R Madariaga (90824)

Univesidad de Buenos Aires - FIUBA

Contents

Prólogo

Objetivos

Familiarizarse con las herramientas de software que usaremos en los siguientes trabajos, implementando un programa y su correspondiente documentación que resuelvan el problema descripto más abajo.

Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes.

Además, es necesario que el trabajo práctico incluya, la presentación de los resultados obtenidos, explicando, cuando corresponda, con fundamentos reales, las causas o razones de cada resultado obtenido.

El informe deberá respetar el modelo de referencia que se encuentra en el grupo, y se valorarán aquellos escritos usando la herramienta TEX / LATEX.

Recursos

Usaremos el programa GXemul para simular el entorno de desarrollo que utilizaremos en este y otros trabajos prácticos, una máquina MIPS corriendo una versión reciente del sistema operativo NetBSD. Durante la primera clase del curso presentaremos brevemente los pasos necesarios para la instalación y configuración del entorno de desarrollo.

Fecha de entrega

La última fecha de entrega y presentación será el jueves 5 de abril de 2018.

Informe

El informe deberá incluir:

- Documentación relevante al diseño e implementación del programa.
- Documentación relevante al proceso de compilación: cómo obtener el ejecutable a partir de los archivos fuente.
- Las corridas de prueba, con los comentarios pertinentes.
- El código fuente, en lenguaje C.
- Este enunciado.

Chapter 1

Introducción

El comando wc

El comando de Unix wc toma como entrada un archivo o stdin, y cuenta las palabras, las líneas y la cantidad de caracteres que contiene.

Programas a desarrollar

El programa a escribir, en lenguaje C, recibirá un nombre de archivo que contiene texto (o el archivo mismo por stdin) e imprimirá por stdout la cantidad de líneas, palabras y caracteres que contiene, junto con el nombre del archivo.

Ejemplos

Primero, usamos la opción -h para ver el mensaje de ayuda:

```
$ tp0 -h
Usage:
tp0 -h
tp0 -V
tp0 [options] file
Options:
-V, --version Print version and quit.
-h, --help Print this information.
-l, --words Print number of lines in file.
-w, --words Print number of words in file.
-c, --words Print number of characters in file.
-i, --input Path to input file.
Examples:
tp0 -w -i input.txt
```

```
15 Luego, lo usamos con un peque\~no fragmento de texto:
16 $echo -n "El tractorcito rojo que silbo y bufo" > entrada.
      txt
17 $tp0 -w -i entrada.txt
18 7 entrada.txt
19 $
```

Listing 1.1: mensaje de ayuda

Mediciones

Se deberá medir el tiempo insumido por el programa para el caso de los archivos alice.txt, beowulf.txt, cyclopedia.txt y elquijote.txt.

Graficar el tiempo insumido contra el tamaño de muestra. Se deberá comprobar que el programa acepta las opciones dadas, y que reporta un error ante situaciones anómalas (como no encontrar el archivo solicitado). La ejecución del programa debe realizarse bajo el entorno MIPS.

Chapter 2

Desarrollo

Compilación

Para la compilación del código se utilizo un Makefile que se muestra a continuación:

```
1  # makefile parameters
2  BIN_NAME    := tp
3  SRCDIR      := src
4  TESTDIR     := tests
5  BUILDDIR    := int
6  TARGETDIR   := target
7  SRCEXT      := c
8
9  # compiler parameters
10 CC          := gcc
11 CFLAGS      := -O3 -std=c99 -Wall -Wpedantic -Werror
12 LIB         :=
13 INC         := /usr/local/include
14 DEFINES     :=
15
16
17 #---------------------------------------------
18 # DO NOT EDIT BELOW THIS LINE
19 #---------------------------------------------
20
21 # binaries file names
22 # each binary source code is expected to be in $(SRCDIR)/<
   binary>
23 TARGET := $(TARGETDIR)/$(BIN_NAME)
24
```

```
25 # sets the src directory in the VPATH
26 VPATH := $(SRCDIR)
27
28 # sets the build directory based on the binary
29 BUILDDIR := $(BUILDDIR)
30
31 # source files
32 SRCS := $(shell find $(SRCDIR) -type f -name *.$(SRCEXT))
33
34 # object files
35 OBJS := $(patsubst %,$(BUILDDIR)/$(BIN_NAME)/%,$(SRCS:.$(
      SRCEXT)=.o))
36
37 # includes the flag to generate the dependency files when
      compiling
38 CFLAGS += -MD
39
40
41 # special definitions used for the unit tests
42 ifeq ($(MAKECMDGOALS),tests)
43     # adds an extra include so the tests can include the
          sources
44   INC += src
45
46   # sets the special define for tests
47   DEFINES := __TESTS__ $(DEFINES)
48
49   # includes the tests directory in the VPATH
50   VPATH := $(TESTDIR) $(VPATH)
51
52   # test sources
53   TEST_SRCS := $(shell find $(TESTDIR) -type f -name *.$(
         SRCEXT))
54
55   # test objects
56   OBJS := $(patsubst $(BUILDDIR)/$(BIN_NAME)/%,$(BUILDDIR)/
         tests/%,$(OBJS))
57   OBJS := $(patsubst %,$(BUILDDIR)/tests/%,$(TEST_SRCS:.$(
         SRCEXT)=.o)) $(OBJS)
58 endif
59
60 # adds the include prefix to the include directories
61 INC := $(addprefix -I,$(INC))
```

```
62
63 # adds the lib prefix to the libraries
64 LIB := $(addprefix -l,$(LIB))
65
66 # adds the define prefix to the defines
67 DEFINES := $(addprefix -D,$(DEFINES))
68
69
70 # builds the binary
71 $(TARGET): $(OBJS) | dirs
72   @$(CC) $(CFLAGS) $(INC) $(DEFINES) $^ $(LIB) -o $@
73   @echo "LD $@"
74
75 # compiles the tests
76 tests: $(TARGET) | dirs
77   python tests/run.py $(TARGET)
78
79 # shows usage
80 help:
81   @echo "Para compilar el binario:"
82   @echo
83   @echo "\t\033[1;92m$$ make\033[0m"
84   @echo
85   @echo "Para compilar y ejecutar las pruebas (require
       python 2.7+ instalado):"
86   @echo
87   @echo "\t\033[1;92m$$ make tests\033[0m"
88   @echo
89   @echo "Los binarios compilados se encuentran en \033[1;92
       m$(TARGETDIR)\033[0m."
90   @echo
91
92 # clean objects and binaries
93 clean:
94   @$(RM) -rf $(BUILDDIR) $(TARGETDIR)
95
96 # creates the directories
97 dirs:
98   @mkdir -p $(TARGETDIR)
99   @mkdir -p $(BUILDDIR)
100
101 # rule to build object files
102 $(BUILDDIR)/$(BIN_NAME)/%.o $(BUILDDIR)/tests/%.o: %.$(
```

```
    SRCEXT)
103   @mkdir -p $(basename $@)
104   @echo "CC $<"
105   @$(CC) $(CFLAGS) $(INC) $(DEFINES) $(LIB) -c -o $@ $<
106
107
108 .PHONY: clean dirs tests
109
110 # includes generated dependency files
111 -include $(OBJS:.o=.d)
```

Listing 2.1: makefile

Para la compilación del código basta con ejecutar el Makefile:

```
$ make
```

Corridas de prueba

A la hora de ejecutar las pruebas, basta usar el mismo Makefile pero ahora con lo opci'on **tests**:

```
$ make tests
```

Resultados

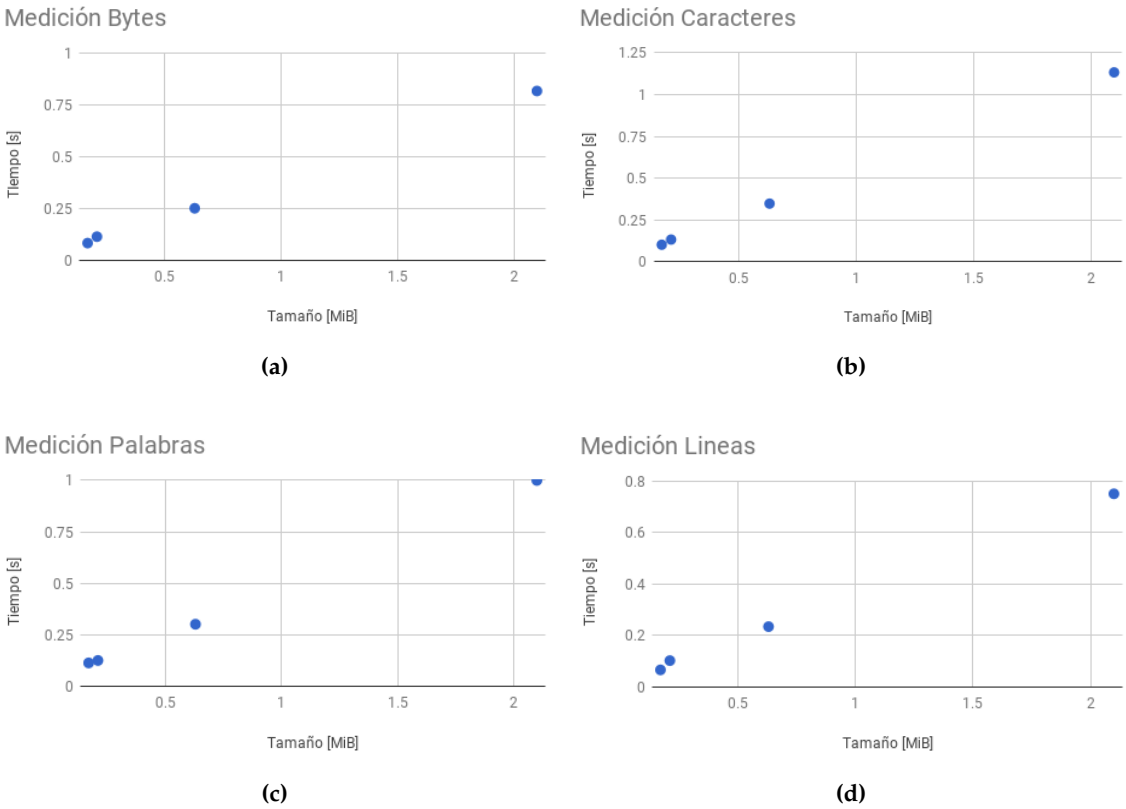
Se utilizaron los 4 archivos de palabras provistos, con los siguientes tamaños (en MiB):

	Size
Alice	0.17
Beowul/f	0.21
Cyclopedia	0.63
El quijote	2.1

En la siguiente tabla se muestran los timpos (en segundos) que se tardó en ejecutar el programa con cada entrada:

	Count Bytes	Count Chars	Count words	Count lines
Alice	0.082	0.102	0.113	0.066
Beowul/f	0.113	0.133	0.125	0.102
Cyclopedia	0.25	0.348	0.301	0.234
El quijote	0.816	1.133	1	0.75

En base a estos resultados se pueden realizar gráficos para evidenciar la tendencia como se muestran en 2.1



(a) **(b)**

(c) **(d)**

Figure 2.1: Gráficos de resultados obtenidos para cada valor de entrada

Table 2.1: My caption

	Count Type	count	real	user	sys
Alice	Bytes	177428	0m0.082s	0m0.066s	0m0.016s
	Chars	177412	0m0.102	0m0.098s	0m0.004s
	Words	30357	0m0.113s	0m0.066s	0m0.035s
	Lines	4046	0m0.066s	0m0.066s	0m0.000s
Beowulf	Bytes	224839	0m0.113s	0m0.086s	0m0.027s
	Chars	224806	0m0.133s	0m0.113s	0m0.020s
	Words	37048	0m0.125s	0m0.121s	0m0.004s
	Lines	4562	0m0.102s	0m0.082s	0m0.020s
Cyclopedia	Bytes	658543	0m0.250s	0m0.242s	0m0.008s
	Chars	658543	0m0.348s	0m0.332s	0m0.016s
	Words	105582	0m0.301s	0m0.293s	0m0.008s
	Lines	17926	0m0.234s	0m0.223s	0m0.012s
El Quijote	Bytes	2198907	0m0.816s	0m0.789s	0m0.027s
	Chars	2155340	0m1.133s	0m1.094s	0m0.020s
	Words	389470	0m1.000s	0m0.980s	0m0.020s
	Lines	37862	0m0.750s	0m0.734s	0m0.016s

Resultados utilizando MIPS

Utilizando la máquina virtual, se obtuvieron las mediciones de tiempo utilizando la utilidad **time**. Los resultados se muestran en la tabla **??**.

Código Fuente

A contiunación se muestra el código fuente:

```
1  #include <ctype.h>
2  #include <getopt.h>
3  #include <inttypes.h>
4  #include <stdbool.h>
5  #include <stddef.h>
6  #include <stdint.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11
12 /** Tipos de datos */
13
14 /** Tipo de contador para los datos de entrada */
```

```
15 typedef enum {
16   /** Cuenta los bytes */
17   counter_type_byte,
18
19   /** Cuenta los caracteres (teniendo en cuenta los multi-
       bytes) */
20   counter_type_char,
21
22   /** Cuenta las palabras (delimitadas por 'isspace') */
23   counter_type_word,
24
25   /** Cuenta las lineas (delimitadas por '\n') */
26   counter_type_line,
27
28   /** Valor cuando no se especifico nungun contador. */
29   counter_type_invalid,
30
31 } counter_type_t;
32
33 /** Parametros parseados de la linea de comandos. */
34 struct args {
35   /* el tipo de contador a utilizar para los datos de
       entrada. */
36   counter_type_t counter_type;
37
38   /* Path del archivo con los datos de entrada */
39   const char *path;
40
41   /* Boolean indica si se usa stdin */
42   bool is_stdin;
43 };
44
45 /** Estructuras de datos */
46
47 /** Estructura que uitliza getopt_log para parsear los
       argumentos de linea de
48  * comandos. */
49 static const struct option _long_opts[] = {
50     {.name = "help", .has_arg = no_argument, .flag = NULL,
         .val = 'h'},
51     {.name = "version", .has_arg = no_argument, .flag =
         NULL, .val = 'V'},
52     {.name = "bytes", .has_arg = no_argument, .flag = NULL,
```

```
        .val = 'b'},
53      {.name = "chars", .has_arg = no_argument, .flag = NULL,
            .val = 'c'},
54      {.name = "words", .has_arg = no_argument, .flag = NULL,
            .val = 'w'},
55      {.name = "lines", .has_arg = no_argument, .flag = NULL,
            .val = 'l'},
56      {.name = "input", .has_arg = required_argument, .flag =
            NULL, .val = 'i'},
57      {0},
58  };
59
60  /** Funciones */
61
62  /**
63   * @brief Imprime un mensaje de ayuda y termina el programa
        .
64   *
65   * @param bin_name argv[0].
66   */
67  static void _print_help(const char *bin_name) {
68    printf("USE: %s [OPTIONS]\n", bin_name);
69    printf("Valid options:\n");
70    printf("  -h, --help        Prints this message and exits
        .\n");
71    printf("  -V, --version     Prints version and exits.\n")
        ;
72    printf("  -b, --bytes       Counts input file's bytes and
         exits.\n");
73    printf("  -c, --chars       Counts input file's
        characters and exits.\n");
74    printf("  -w, --words       Counts input file's words and
         exits.\n");
75    printf("  -l, --lines       Counts input file's lines and
         exits.\n");
76    printf(
77        "  -i, --input [FILE] means the input will follow the
             path after -i, if "
78        "it is inexistant, stdio will be used.\n");
79    printf("\n");
80    printf(
81        "FILE is the name of the file to read, o '-' to read
           from "
```

```
82        "STDIN.\n");
83  }
84
85  /**
86   * @brief Imprime la version del programa y termina.
87   *
88   * @param bin_name argv[0].
89   */
90  static void _print_version(const char *bin_name) {
91    printf("%s, version 1.00\n", bin_name);
92  }
93
94  /**
95   * @brief Realiza el parseo de los parametros de linea de
         comandos.
96   *
97   * @param args Estructura que contiene los parametros
         parseados.
98   * @param argc
99   * @param argv
100  */
101 static void _arg_parse(struct args *args, int argc, const
       char **argv) {
102   counter_type_t type = counter_type_invalid;
103   args->is_stdin = true;
104   int ch = -1;
105
106   while ((ch = getopt_long(argc, (char **)argv, "hVbcwli:",
          _long_opts, NULL)) != -1) {
107     switch (ch) {
108       case 'h':
109         _print_help(argv[0]);
110         exit(0);
111         break;
112
113       case 'V':
114         _print_version(argv[0]);
115         exit(0);
116         break;
117
118       case 'b':
119         type = counter_type_byte;
120         break;
```

```
121
122       case 'c':
123         type = counter_type_char;
124         break;
125
126       case 'w':
127         type = counter_type_word;
128         break;
129
130       case 'l':
131         type = counter_type_line;
132         break;
133
134       case 'i':
135         args->path = argv[optind - 1];
136         args->is_stdin = (strcmp("-", args->path) == 0);
137         break;
138
139       /* this is returned when a required argument was not
              provided */
140       case ':':
141       case '?':
142         exit(1);
143     }
144   }
145
146   if (type == counter_type_invalid) {
147     printf("Counter not specified.\n");
148     exit(1);
149   }
150
151   /* llena la estructura de salida */
152   args->counter_type = type;
153 }
154
155 /**
156  * @brief Lee del `input` hasta que no haya mas datos y
        aplica el contador
157  * especificado.
158  *
159  * @param input Archivo de donde leer los datos.
160  * @param counter_type Tipo de contador a utilizar.
161  * @return Resultado del contador.
```

```
162  */
163 static uint64_t _process_input(FILE *input, counter_type_t
       counter_type) {
164   unsigned int counter = 0;
165   char buffer[2048];
166   size_t buffer_len = 0;
167   char new_byte, prev_byte = 0;
168
169   while (buffer_len = fread(buffer, 1, sizeof(buffer),
        input), buffer_len > 0) {
170     for (size_t i = 0; i < buffer_len; i++) {
171       new_byte = buffer[i];
172       switch (counter_type) {
173         case counter_type_byte:
174           counter++;
175           break;
176
177         case counter_type_char:
178           counter += (new_byte & 0xc0) != 0x80;
179           break;
180
181         case counter_type_word:
182           if (prev_byte == 0 || (isspace(prev_byte) && !
                isspace(new_byte))) {
183             counter++;
184           }
185           break;
186
187         case counter_type_line:
188           if (new_byte == '\n') {
189             counter++;
190           }
191           break;
192
193         case counter_type_invalid:
194           return 0;
195       }
196
197       prev_byte = new_byte;
198     }
199   }
200
201   return counter;
```

```
202 }
203
204 int main(int argc, const char *argv[]) {
205   struct args args = {0};
206
207   /* parsea la linea de comandos */
208   _arg_parse(&args, argc, argv);
209
210   /* Si es STDin, pone el archivo como stdin. Si no abrimos
          con la ruta */
211   FILE *file;
212
213   if (args.is_stdin) {
214     file = stdin;
215   } else {
216     file = fopen(args.path, "r");
217     if (file == 0) {
218       perror("Error");
219       exit(0);
220     }
221   }
222
223   /* procesa la entrada */
224   uint64_t count = _process_input(file, args.counter_type);
225   printf("%" PRIu64 "\n", count);
226
227   fclose(file);
228   return EXIT_SUCCESS;
229 }
```

Listing 2.2: código fuente del programa