Trabajo Práctico N°1

"Conjunto de instrucciones MIPS"

Belén Beltran, Padrón Nro. 91.718
belubeltran@gmail.com

Pablo Ariel Rodriguez, Padrón Nro. 93.970
prodriguez@fi.uba.ar

Federico Martín Rossi, Padrón Nro. 92.086
federicomrossi@gmail.com

2do. Cuatrimestre 2013
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

Índice

1. Introducción

En el presente trabajo se tiene como objetivo la comparación entre dos algoritmos de ordenamiento: el *Bubblesort* y el *Shellsort*. Para realizar dicha comparación entre ambos se realizó la implementación en lenguaje C de cada uno de estos. A su vez, para comparar el rendimiento entre código de alto nivel y de código nativo, se desarrollo la implementación del Heapsort en assembly MIPS, con el fin de poder comparar los tiempos de ejecución de ambos programas y realizar así un estudio de las mejoras que se producen.

La totalidad del trabajo se ha realizado en una plataforma *NetBSD/MIPS-32* mediante el *GXEmul* [1].

Todos los archivos y códigos fuente aquí mencionados, así como también el presente informe, pueden ser descargados como un archivo comprimido ZIP del repositorio del grupo[1].

2. Compilación

La herramienta para compilar tanto el código asembly como C será el *GCC* [2]. Para tratar de equiparar al máximo el *S* generado por ambas implementaciones, se utilizará el flag de gcc "-O0" para que no realice optimizaciones sobre el código en lenguaje C.

Para automatizar las tareas de compilación se hace uso de la herramienta *GNU Make*. Los Makefiles utilizados para la compilación se incluyen junto al resto de los archivos fuentes del presente trabajo [2].

3. Utilización

En los siguientes apartados se especifica la forma en la que deben ser ejecutados los programas implementados tanto en C como en assembly MIPS.

3.1. Implementación en C

El resultado de compilación utilizando el comando *make* será un archivo ejecutable de nombre *tp1*, que podrá ser invocado con los siguientes parámetros:

- *-h*: Imprime ayuda para la utilización del programa;
- *-V*: Imprimer la versión actual del programa;
- *-b [ARGS]*: El programa recibe nombres de archivos de texto o strings ingresados por *stdin*, ordenandolos utilizando el algoritmo Bubblersort. Para utilizar *stdin* deberá omitirse [ARGS] y luego introducir las palabras;
- *-s [ARGS]*: El programa recibe nombres de archivos de texto o strings ingresados por *stdin*, ordenandolos utilizando el algoritmo Heapsort. Para utilizar *stdin* deberá omitirse [ARGS] y luego introducir las palabras.

3.2. Implementación en Assembly

El resultado de compilación utilizando el comando *make* será un archivo ejecutable de nombre *tp1*, el cual aceptará un archivo de texto como argumento y lo ordenará con el algoritmo Shellsort.

4. Implementación

En lo que sigue de la sección, se presentarán los códigos fuente de la implementación del algoritmo. Aquellos lectores interesados en la implementación completa del programa, pueden dirigirse al apéndic ubicado al final del presente informe.

[1] URI del Repositorio: `https://github.com/federicomrossi/6620-trabajos-practicos-2C2013/tree/master/tp1`

[2] Los archivos se encuentran separados según la implemetación a la que pertenecen, por lo que habrán dos Makefiles distintos, uno para la implementación en lenguaje C y otro para la implementación en assembly

4.1. Implementación en C

La implementación del programa fue divida en los siguientes módulos:

- **tp1**: Programa principal responsable de interpretar los comandos pasados por la terminal de modo que realice las tareas solicitadas por el usuario. Su principal función es encadenar el funcionamiento de los otros módulos y mostrar por pantalla el resultado obtenido;
- **bubblesort**: Módulo encargado de implementar el algoritmo de ordenamiento Bubblesort. Recibe como parámetros un arreglo de palabras desordenado y el tamaño del mismo. Como resultado devuelve dicho arreglo ordenado.
- **heapsort**: Módulo encargado de implementar el algoritmo de ordenamiento Heapsort. Recibe como parámetros un arreglo de palabras desordenado y el tamaño del mismo. Como resultado devuelve dicho arreglo ordenado.

4.1.1. Algoritmo *Bubblesort*

En el *Código 1* se muestra el header de la librería, donde se declara la función *bubblesort*, mientras que en el *Código 2* se muestra la definición de la librería.

Código 1: *"bubblesort.h"*

```c
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA BUBBLESORT
 *
 * ******************************************************************************
 * ******************************************************************************/



#ifndef BUBBLESORT_H
#define BUBBLESORT_H



// Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void bubblesort_(char* words[], int arraysize);



#endif
```

Código 2: *"bubblesort.c"*

```c
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA BUBBLESORT
 *
 * ******************************************************************************
 * ******************************************************************************/


#include "bubblesort.h"
#include <stdbool.h>
#include <string.h>



// Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
// arreglo de palabras.
```

```
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void bubblesort_(char* words[], int arraysize)
{
  // Variables de procesamiento
  bool huboIntercambio;
  int i;
  int n = arraysize;
  char* sAux;

  // Recorremos el arreglo haciendo intercambios hasta que ya no se registre
  // ningun cambio realizado.
  do
  {
    huboIntercambio = false;

    for(i = 1; i < n; i++)
    {
      // Si el de indice menor es mayor que el de indice superior, los
      // intercambiamos
      if(strcasecmp(words[i-1], words[i]) > 0)
      {
        sAux = words[i-1];
        words[i-1] = words[i];
        words[i] = sAux;

        // Cambiamos el flag para registrar que hubo un cambio
        huboIntercambio = true;
      }
    }

    // Como el elemento del indice superior se encuentra ya ordenado una
    // vez finalizada la pasada, se reduce en uno la cantidad de indices
    // a iterar en la proxima pasada.
    n -= 1;

  } while(huboIntercambio);
}
```

4.1.2. Algoritmo *Heapsort*

En el *Código 3* se muestra el header de la librería, donde se declara la función *heapsort*, mientras que en el *Código 4* se muestra la definición de la librería.

Código 3: *"heapsort.h"*

```
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA HEAPSORT
 *
 * ******************************************************************************
 * ******************************************************************************/


#ifndef HEAPSORT_H
#define HEAPSORT_H



// Funcion que aplica el algoritmo de ordenamiento Heapsort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void heapsort_(char* words[], int arraysize);


```

```
#endif
```

Código 4: *"heapsort.c"*

```
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA HEAPSORT
 *
 * ******************************************************************************
 * ******************************************************************************/


#include "heapsort.h"
#include <string.h>


void heap(char* words[],int raiz, int fin){
    char finished = 0;
    char* tmp;
    int hijo;
    while (raiz*2+1 <= fin && !finished){
        //Elijo el hijo mayor
        if(raiz*2+1==fin || strcasecmp(words[raiz*2+1],words[raiz*2+2]) > 0 ){
            hijo=raiz*2+1;
        }else{
            hijo=raiz*2+2;
        }
        //Checkeo si es mayor e intercambio
        if(strcasecmp(words[raiz],words[hijo]) < 0){
            tmp=words[raiz];
            words[raiz]=words[hijo];
            words[hijo]=tmp;
            raiz=hijo;
        }else{
            finished=1;
        }
    }
}



// Funcion que aplica el algoritmo de ordenamiento Heapsort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void heapsort_(char* words[], int arraysize){
    int i;
    char* tmp;
    //Armo Heap
    for (i=(arraysize/2)-1;i>=0;i--){
        heap(words,i,arraysize-1);
    }

    //Extraigo raiz y armo heap
    for(i=arraysize-1;i>0;i--){
        tmp=words[0];
        words[0]=words[i];
        words[i]=tmp;
        heap(words,0,i-1);
    }
}
```

4.2. Implementación en Assembly

La implementación del programa fue divida en los siguientes módulos:

- **tp1**: Solamente recibe un texto como argumento por linea de comandos y lo imprime ordenandolo mediante heapsort. Esta implementado en lenguaje C;
- **strcmpi**: Función implementada en assembly MIPS que se encarga de realizar la comparación de strings sin ser sensible a mayúsculas;
- **heap**: Función implementada en assembly MIPS que se encarga de [...];
- **heapsort**: Implementación en assembly MIPS del algoritmo de ordenamiento Heapsort.

4.2.1. Algoritmo *Heapsort*

En el *Código 5* se muestra la implementación en assembly del algoritmo Heapsort.

Código 5: *"heapsort.S"*

```
#ifndef USE_MIPS_ASSEMBLY
#define USE_MIPS_ASSEMBLY
#include <mips/regdef.h>
#include <sys/syscall.h>


# Utilizacion de registros:
#
# s0: words
# s1: arraysize
# s2: variable local "i"
# s3: variable local "tmp"
# t0: variable local auxiliar
# t1: variable local auxiliar
#


.text
.align  2
.globl  heapsort
.extern heap

      # ABA: 16; SRA: 16; LTA:24
heapsort: subu  sp,sp,56        # Creamos stack frame.
      sw    ra,52(sp)       # Return address
      sw    $fp,48(sp)        # Frame pointer
      sw    gp,44(sp)       # Global pointer
      move  $fp,sp          # Establecemos la base

      # Almacenamos contenido de registros
      sw    s0,40(sp)
      sw    s1,36(sp)
      sw    s2,32(sp)
      sw    s3,28(sp)

      # Almacenamos los parametros recibidos
      sw    a0,56(sp)       # words
      sw    a1,60(sp)       # arraysize

      # Almacenamos los parametros guardados en fp en variables
      move  s0,a0         # words
      move  s1,a1           # arraysize

      # Armamos el heap
      srl   t0,s1,1         # t0 = arraysize / 2
      subu  s2,t0,1         # i = arraysize / 2 - 1
FOR_1:    slt   t0,s2,zero        # i es menor que 0
      bne   t0,zero,FOR_1_FIN   # Si t0 = 1 saltamos a FOR_1_FIN

      add   a1,s2,zero        # Cargamos i en a1
      subu  a2,s1,1         # Cargamos arraysize-1 en a2
      jal   heap          # Saltamos a funcion "heap"

      subu  s2,s2,1         # i = i-1 (decrementamos i)
      j   FOR_1           # Saltamos a etiqueta FOR_1

      # Iteramos sobre el heap intercambiando la raiz con el ultimo
      # elemento del arreglo, rearmando luego un heap desde los indices
```

```
59      # 0 al n-1
60  FOR_1_FIN:  subu  s2,s1,1         # i = arraysize-1
61  FOR_2:    beq   s2,zero,FOR_2_FIN   # Si i = 0 saltamos a FOR_2_FIN
62
63      lw    s3,0(s0)        # tmp = words[0]
64      sll   t1,s2,2         # t1 = i * 4
65      add   t1,t1,s0        # t1 = words[i]
66      lw    t2,0(t1)        # t2 = dato words[i]
67      sw    t2,0(s0)        # words[0] = words[i]
68      sw    s3,0(t1)        # words[i] = tmp
69
70      add   a0,s0,zero        # Cargamos words en a0
71      add   a1,zero,zero      # Cargamos 0 en a1
72      subu  a2,s2,1         # a2 = i -1
73      jal   heap          # Saltamos a funcion "heap"
74
75      subu  s2,s2,1         # i = i-1 (decrementamos i)
76      j   FOR_2           # Saltamos a etiqueta FOR_2
77
78      # Reestablecemos valores iniciales
79  FOR_2_FIN:  lw    a0,56(sp)       # words
80      lw    a1,60(sp)       # arraysize
81
82      lw    s0,40(sp)
83      lw    s1,36(sp)
84      lw    s2,32(sp)
85      lw    s3,28(sp)
86
87      move  sp,$fp
88      lw    gp,44(sp)       # Global pointer
89      lw    $fp,48(sp)        # Frame pointer
90      lw    ra,52(sp)       # Return address
91      addi  sp,sp,56
92
93      jr    ra
94
95  #endif
```

5. Debugging

Para analizar el correcto funcionamiento del programa se crearon casos de prueba pertinentes, considerando combinaciones diferentes en el ingreso de parámetros al programa principal, como también tomando en cuenta las diferentes salidas obtenidas a partir del algorirmo *pgm*. Los resultados fueron comparados con los casos esperados y así se determinó el correcto funcionamiento del programa en su totalidad.

6. Pruebas

6.1. Pruebas al ingreso y egreso de datos del programa

A modo de prueba del programa, se ingresaron ciertos parámetros a éste y se analizó la salida obtenida. A continuación se presentan algunas pruebas realizadas y los resultados obtenidos.

El primer caso consiste en crear un tablero de resolución 11x16 y que el resultado sea enviado a la salida estandar. Se debe ingresar por consola lo siguiente:

```
$ ./tp0 -r 11x16 -o -
```

La salida del programa es la siguiente:

```
P2
# -
11 16
1
```

```
1 1 0 0 1 1 0 1 0 1 0
1 1 0 0 1 1 0 1 0 1 0
0 0 1 1 0 0 1 0 1 0 1
0 0 1 1 0 0 1 0 1 0 1
1 1 0 0 1 1 0 1 0 1 0
1 1 0 0 1 1 0 1 0 1 0
0 0 1 1 0 0 1 0 1 0 1
0 0 1 1 0 0 1 0 1 0 1
1 1 0 0 1 1 0 1 0 1 0
1 1 0 0 1 1 0 1 0 1 0
0 0 1 1 0 0 1 0 1 0 1
0 0 1 1 0 0 1 0 1 0 1
1 1 0 0 1 1 0 1 0 1 0
1 1 0 0 1 1 0 1 0 1 0
0 0 1 1 0 0 1 0 1 0 1
0 0 1 1 0 0 1 0 1 0 1
```

La segunda prueba consiste en crear un tablero de resolución 16x11 y también enviar la salida a consola. Se debe ingresar por la línea de comandos:

```
$ ./tp0 -r 16x11 -o -
```

La salida del programa es la siguiente:

```
P2
# -
16 11
1
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
```

La tercer prueba consiste en generar el tablero de tamaño default 8x8 (se genera en caso de no ingresar la opción -r), ingresando por consola:

```
$ ./tp0 -o -
```

El resultado del programa se ve por salida standard y es el siguiente:

```
P2
# -
8 8
1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
```

```
0 1 0 1 0 1 0 1
```

Una prueba más interesante, es aquella en la cual la resolución permite ver una imagen bastante amplia. En este caso se considera una resolución de 200x200 y se decide guardar la salida en un archivo. Se debe ingresar por consola:

```
$ ./tp0 -r 200x200 -o 200x200.pgm
```

La salida en formato de imagen *PGM* se muestra en la *Figura 1*.

6.2. Tiempos de ejecución

Se quiere medir el tiempo que tarda el programa en la máquina virtual MIPS32 en crear un tablero de ajedrez con ciertas características. Para ello se realizó un módulo aparte que, con la ayuda del comando *GNU "time"* [4], mide los tiempos de la realización de un tablero de resolución 1x1 hasta uno de 1000x1000.

Los datos obtenidos se pueden observar en el *Cuadro 1*.

Resolución	**Tiempo de ejecución [s]**
1x1	0,051
100x100	0,316
200x200	1,148
300x300	2,5
400x400	4,434
500x500	6,844
600x600	9,848
700x700	13,5
800x800	17,449
900x900	22,152
1000x1000	27,613

Cuadro 1: *Tiempos en segundos obtenidos en la ejecución del programa para distintas resoluciones.*

En la *Figura 2* se presenta un diagrama resumido de los tiempos de ejecución obtenidos. Como es de esperar, se obtiene una curva cuadrática.

Referencias

[1] The NetBSD project, http://www.netbsd.org/

[2] GCC, the GNU Compiler Collection, http://gcc.gnu.org/

[3] PGM format speciffication, http://netpbm.sourceforge.net/doc/pgm.html

[4] time man page, http://unixhelp.ed.ac.uk/CGI/man-cgi?time

[5] J. L. Hennessy and D. A. Patterson, "Computer Architecture. A Quantitative Approach," 4th Edition, Morgan Kaufmann Publishers, 2000.

Apéndices

A. Implementación completa en lenguaje C

A.1. *tp0.c*. Implementación del main del programa

A.2. *pgm.h*. Declaración de la librería Pgm

A.3. *pgm.c*. Definición de la librería Pgm

B. Generación del código completo en assembly MIPS

B.1. *tp0.s*. Generación del main del programa

B.2. *pgm.s*. Generación del algoritmo Pgm