Universidad De Buenos Aires

Facultad De Ingeniería

66.20 Organización De Computadoras

Trabajo Práctico 1

Integrantes:
Gonzalo Beviglia - 93144
Federico Quevedo - 93159
Damián Manoff - 93169

19 de noviembre de 2013

Índice

1. Diseño e implementación

Concluida la primer parte del trabajo, se pide implementar puramente en MIPS assembly el mismo programa que ya se obtenía automaticamente, con el mínimo cambio que hace interesante la partida, mezclar código C con assembly.
Para esto se deberá respetar la ABI propuesta por la cátedra y vista en clases, además de algunas pautas pedidas:

- Los archivos serán abiertos en C, pero manejados mediante syscall desde assembly.
- La funcion *reverse(int fd, int outfd)* será puramente implementada en assembly con código propio.
- Las llamadas a reserva de memoria se harán con funciones proveídas por la cátedra.

2. Performance

En esta parte se evaluará la performance de lo implementado con respecto al tp0 anteriormente entregado. Para ello se invertirá el libro "El Príncipe" de Nicolás Maquiavelo. El tamaño de dicho texto en formato de texto plano es de 305864 bytes (298KB).

Los tiempos se midieron utilizando el comando Unix *time*.

2.1. Tiempo *reverse TP0*

```
real 0m0.539s

user 0m0.008s

sys 0m0.016s
```

2.2. Tiempo *reverse TP1*

```
real 0m0.740s

user 0m0.032s

sys 0m0.054s
```

3. Compilación del programa

Para el compilado del programa hicimos el siguiente makefile:

```
CFLAGS=-g -Wall -o x
MEMFLAGS=valgrind --leak-check=full --track-origins=yes -v
CC=gcc
MAIN=main.c
REVERSE=reverse.S
MYMALLOC=sys_mmap/mymalloc.S
EXEC=main
TESTSCRIPT=TestFiles/tests.sh

all: $(EXEC) clean
test: all runTests cleanExec

.SILENT:
$(EXEC): $(MAIN) $(REVERSE) $(MYMALLOC)
	$(CC) $(CFLAGS) $(MAIN) $(REVERSE) $(MYMALLOC) -o $(EXEC)

.SILENT:
run: $(EXEC)
	./$(EXEC)

.SILENT:
runTests: $(EXEC)
	./$(TESTSCRIPT)

.SILENT:
memCheck: $(EXEC)
	$(MEMFLAGS) ./$(EXEC) TestFiles/test TestFiles/test1 TestFiles/test2

.SILENT:
clean: $(MIDDLEFILE)
	rm -f *.o

.SILENT:
cleanExec: $(EXEC)
	rm -f $(EXEC)
```

La ejecución normal de este make file produce el archivo ejecutable y ademas elimina los intermediarios.

Se puede tambien llamar pasando como parametro el nombre del archivo intermediario para generarlo, o el nombre del ejecutable, que realizara lo mismo que la ejecución por defecto pero sin eliminar el intermediario.

Para corroborar que no se estuviera perdiendo memoria tambien incluimos el parametro *memCheck* que corre el programa con valgrind informando si hubo o no alguna perdida.

Desde el mismo makefile tambien incluimos la posibilidad de correr las pruebas, y por ultimo, la de eliminar los archivos generados, tanto intermediarios como programa final.

4. Código Fuente

4.1. Código fuente C

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
```

```c
#include "reverse.h"
int main(int argc, char** argv) {

        //int fPtr = 0;
        FILE* outFd = fopen("salida.out","w");
        // Rev from stdin.
        if( argc == 1 ) {
                reverse(STDIN_FILENO, fileno( outFd ));
                //reverse(STDIN_FILENO, 1);
                fclose(outFd);
                return 0;
        }

        // Option may have been passed.
        if( argc == 2 ) {
                // Option was matched.
                //if( checkOption(argv[1]) ) return 0;
        }
        unsigned i;
        for( i = 1 ; i < argc ; i++ ) {
                int fPtr = open(argv[i], O_RDONLY);

                // Handling opening file error.
                if( !fPtr )
                 fprintf(stderr, "Error: unable to open file %s\n", argv[i]);
                else {
                        reverse(fPtr,fileno( outFd ));
                        close(fPtr);
                        fclose(outFd);
                }
        }
        return 0;
}
```

```c
#include "reverse.h"

/*
 * Prints the output of the given file after its lines were
 * reversed. File pointer must be already opened for reading
 * and must be closed after this call.
 *
 * fPtr: file to be reversed.
 */
int reverse(int infd, int outfd)
{
        int result = reverseS(infd,outfd);
        return result;


}
```

```c
#ifndef _CALLBACK_H_
#define _CALLBACK_H_

extern int reverseS(int a, int B);
```

```
int reverse(int infd, int outfd);

#endif
```

4.2. Código assembly MIPS

A continuación se detallará el código assembly para la arquitectura MIPS de las funciones implementadas para dar vuelta las líneas.

```
#include <sys/syscall.h>
#include <mips/regdef.h>
        .text
        .align  2
        .globl  reverse

        .ent    reverse
reverse:
        subu    sp, sp, 32
        sw      ra, 24(sp)
        sw      $fp, 20(sp)
        sw      gp, 16(sp)

        # a0: infd, input file
        # a1: outfd, output file

        # s0: current pos input file
        # s1: current pos output file
        # s2: current value input file
        # s3: buffer size
        # s4: buffer pointer
        # s5: file length
        # s6: character readed

        move s0, a0                     # Save input file
        move s1, a1                     # Save output file


readFile:
        li      s3, 30                  # InitialBuffer
        addiu t0, s3, 2         # InitialBuffer = InitialBuffer + 2
        move t0, a0
        jal mymalloc

        move s4, v0                     # Buffer pointer
        move s5, zero           # file length

        li v0, SYS_read # system call for write to read
        move a0, s0                     # file descriptor
        move a1, s6                     # s6 <- character
        li a2, 1                        # read 1 byte
        syscall                         # read file
        bltz v0, read_error

fileLoop:
        lb t0, 0(s6)            # Load character value
        beqz t0, endFile        # eof validation

        addiu s5, s5, 1         # Add 1 to lenght
```

```
        addiu t0, s5, 1         # t0 <- length + 1
        beq s3, t0, increaseMalloc      # if (length + 1) == bufferSize

continueLoop:
        addu t0, s4, s5         # t0 = buffer pointer + file length
        subu t0, t0, 1

        lb t1, 0(s6)            # Load character value
        sb t1, 0(t0)            # save character

        li t0, 10                       # \n ascii value
        lb t1, 0(s6)            # Load character value
        beq t1, t0, endFile     # if character == '\n' break

        li v0, SYS_read         # system call for write to read
        move a0, s0                     # file descriptor
        move a1, s6                     # s6 <- character
        li a2, 1                        # read 1 byte
        syscall                         # read file
        bltz v0, read_error

        b fileLoop

increaseMalloc:
        addu s3, s3, s3 # InitialBuffer = InitialBuffer * 2
        addiu t0, s3, 2 # InitialBuffer = InitialBuffer + 2
        move a0, t0
        jal mymalloc    # v0 <- new buffer

        move t1, s3             # Original length
        move t2, zero   # currentPosition = 0

increaseMallocLoop:
        beq t2, t1, endIncrease # if currentPosition == originalLength

        addu t3, s4, t2
        lb t4, 0(t3)            # load character at old buffer

        addu t3, v0, t2
        sb t4, 0(t3)            # Save character at new buffer

        addiu t2, t2, 1

        b increaseMallocLoop


endIncrease:
        move s4, v0             # old buffer = new buffer
        b continueLoop


endFile:
        move    s7, s5          # s7 <- fileLength
        subu    s7, s7, 2       # fileLength = fileLength - 2

reverseLoop:
```

```
        addu    t0, s7, s4      # t2 <- buffer element index

        li              v0, SYS_write   # system call for write to file
        move    a0, s1                  # file descriptor
        move    a1, t0                  # address of buffer from which to write
        li              a2, 1                   # write length
        syscall                                 # write to file
        bltz v0, write_error

        beqz    s7, end         # end reverting string

        subu    s7, s7, 1       # buffer index - 1

        b reverseLoop

read_error:
        li v0, 1        # error code 1
        b end

write_error:
        li v0, 2        # error code 2
        b end

end:
        lw      ra, 24(sp)
        lw      $fp, 20(sp)
        lw      gp, 16(sp)
        addiu   sp, sp, 32
        jr ra
.end reverse
```

5. Conclusiones

En primer lugar llamó la atención que el tiempo del programa puramente en assembler sea mayor que la implementación en C. Pero luego se pudo llegar a la conclusión que el tiempo ganado son las optimizaciones del compilador *gcc*, además de alguna falta producida mezclando assembler de dos lugares distintos (*reverse.S* y *mymalloc.S*)
Otro aspecto a observar es la fragilidad con la que assembler permite trabajar. Es decir, cualquier sentencia mal escrita llevaba al trabajo a terminar como un *segmentation fault*, pero como ventaja se entendía claramente porque se llegaba a ese error, cosa que en C no sucede ya que no se puede ver que es lo que realmente hacen las funciones utilizadas frecuentemente, tales como los *printf,fopen,getc*, etc.