UNIVERSIDAD DE BUENOS AIRES

Facultad de Ingeniería
Departamento de Electrónica

Organización de Computadoras (66.20)

TRABAJO PRÁCTICO Nº 1

Cuatrimestre y año: 2º Cuatrimestre 2016
Profesor Titular: José Luis Hamkalo
Docente a cargo del TP: Leandro Santi
Alumno:

Padrón	***Nombre***	***Email***
	Rinaldi, Lautaro Ezequiel	

Observaciones:
Nota final:

Documentación relevante al diseño e implementación.

Las funciones adicionales implementadas en C fueron:

int print_digits(int fd, char * buffer, size_t n)

Escribe en un buffer el numero entero pasado por parámetro, dígito a dígito (en forma de chars).

int buffer_write_char(int fd, char * buffer, char c)

Escribe en el buffer el carácter pasado por parámetro.

Si el buffer se llena, invoca a buffer_flush que escribe todo el contenido del buffer en el fd pasado por parámetro.

Retorna EXITO si no hubo inconvenientes, y ERROR si no pudo escribir el buffer en el fd pasado.

int buffer_flush(int fd, char * buffer)

Vuelca la información escrita en el buffer en el archivo que se corresponde con el file_descriptor pasado por parámetro. El archivo debe estar previamente abierto.

Si se pudo escribir sin problemas, setea final_buffer en 0 (es decir, el ultimo dato valido del buffer) y retorna EXITO.

En caso de falla, retorna ERROR.

Diagrama de Stack Frame.

Diagrama del Stack Frame de la función **print_digits.**

ABA de caller	**padding**	52($fp)
	a2 = n	48($fp)
	a1 = buffer	44($fp)
	a0 = fd	40($fp)
SRA	**s0**	36($fp)
	ra	32($fp)
	fp	28($fp)
	gp	24($fp)
LTA	**c**	20($fp)
	r	16($fp)
ABA de callee		12($fp)
		8($fp)
		4($fp)
		0($fp)

Diagrama del Stack Frame de la función **buffer_write_char.**

ABA de caller	**padding**	52($fp)
	padding	48($fp)
	a1 = buffer	44($fp)
	a0 = fd	40($fp)
SRA	**padding**	36($fp)
	ra	32($fp)
	fp	28($fp)
	gp	24($fp)
LTA	**cod_ret**	20($fp)
	c	16($fp)
ABA de callee		12($fp)
		8($fp)
		4($fp)
		0($fp)

Diagrama del Stack Frame de la función **buffer_flush.**

ABA de caller	**padding**	44($fp)
	padding	40($fp)
	a1 = buffer	36($fp)
	a0 = fd	32($fp)
SRA	**s0**	28($fp)
	ra	24($fp)
	fp	20($fp)
	gp	16($fp)
ABA de callee		12($fp)
		8($fp)
		4($fp)
		0($fp)

Diagrama del Stack Frame de la función **mips32_plot.**

ABA de caller	**padding**	(BUF_SZ + 100) ($fp)
	padding	(BUF_SZ + 96)($fp)
	padding	(BUF_SZ + 92)($fp)
	parms	(BUF_SZ + 88)($fp)
SRA	**s0**	(BUF_SZ + 84)($fp)
	ra	(BUF_SZ + 80)($fp)
	fp	(BUF_SZ + 76)($fp)
	gp	(BUF_SZ + 72)($fp)
LTA	**. . .**	**. . .**
	buffer	72($fp)
	padding	68($fp)
	cpi	64($fp)
	cpr	60($fp)
	c	56($fp)
	y	52($fp)
	x	48($fp)
	absz	44($fp)
	si	40($fp)
	sr	36($fp)
	zi	32($fp)
	zr	28($fp)
	ci	24($fp)
	cr	20($fp)
	res	16($fp)
ABA de callee		12($fp)
		8($fp)
		4($fp)
		0($fp)

Compilación

Para la compilación del presente trabajo práctico, deben ejecutarse los siguientes comandos

- make clean
- make makefiles
- make

Casos de prueba

Se puede correr una batería de pruebas ejecutando los siguientes scripts:

sh prueba1.sh

Se ejecutan casos de prueba que prueban los puntos cardinales y los casos de prueba que figuran en Wikipedia, para poder contrastar los resultados.

El código del script es:

set -x

 # Casos de prueba espaciales (o puntos cardinales)

./tp1 -o centro.pgm

./tp1 -c 0+1i -o abajo.pgm

./tp1 -c 0-1i -o arriba.pgm

./tp1 -c 1+0i -o izquierda.pgm

./tp1 -c -1+0i -o derecha.pgm

./tp1 -c 1-1i -o arriba_izquierda.pgm

./tp1 -c -1-1i -o arriba_derecha.pgm

./tp1 -c -1+1i -o abajo_derecha.pgm

./tp1 -c 1+1i -o abajo_izquierda.pgm

 # Casos de prueba extraidos de Wikipedia: https://en.wikipedia.org/wiki/Julia_set

./tp1 -C -0.4+0.6i -o wiki_02.pgm

./tp1 -C 0.285+0i -o wiki_03.pgm

./tp1 -C 0.285+0.01i -o wiki_04.pgm

./tp1 -C 0.45+0.1428i -o wiki_05.pgm

./tp1 -C -0.70176-0.3842i -o wiki_06.pgm

./tp1 -C -0.835-0.2321i -o wiki_07.pgm

./tp1 -C -0.8+0.156i -o wiki_08.pgm

./tp1 -C -0.7269+0.1889i -o wiki_09.pgm

sh prueba2.sh

Se ejecutan los casos de prueba que figuran en el enunciado del trabajo práctico. El código del script es:

set -x

 # Casos de prueba de la sección 5.4 del enunciado del TP.

./tp1 -o uno.pgm

 # Casos de prueba del apéndice A del enunciado del TP.

./tp1 -c 0.01+0i -r 1x1 -o -

./tp1 -c 10+0i -r 1x1 -o -

./tp1 -c 0+0i -r 0x1 -o -

./tp1 -o /tmp

./tp1 -c 1+3 -o -

./tp1 -c "" -o -

Conclusiones.

Por medio del presente trabajo práctico se pudo aprender:

- La importancia de la verificación de errores en las operaciones de lectura/escritura en los archivos.

- La importancia de hacer una buena verificación de los parámetros de entrada del programa, ya que si estos ingresaran en forma inválida, provocarían, sin duda alguna, errores en la ejecución del programa.

- Tomar conciencia de la gran cantidad de trabajo que nos ahorra el compilador, ya que por cada línea que nosotros escribimos en un lenguaje de alto nivel como ser el C, el compilador genera una enorme cantidad de líneas en código ensamblador.

- Técnicas y procesos de desarrollo para implementar funciones en un lenguaje de bajo nivel: Dividir las funciones a implementar en funciones más pequeñas, intentando al principio llevarlas a su mínima expresión con el objetivo de familiarizarnos con el lenguaje, y luego ir agrupándolas para lograr conseguir la función deseada, sin efectuar tantos llamados sucesivos. Comencé programando funciones elementales como ser: Sumar 2 números, hacer un desplazamiento a izquierda de un dato, hacer comparaciones de datos, plantear tomas de decisiones mediante sentencias condicionales (if), leer de memoria el valor de un vector. Luego pude notar que al juntar las diferentes funciones, era necesario ir salvando los valores intermedios que quedaban guardados en los registros en memoria, para poder dejar libres dichos registros que se necesitaban para otras operaciones. Ahí también comprendí que es una buena práctica, cada vez que se tiene que trabajar con una variable local, recuperarla del stack frame, operar y luego volver a almacenar su valor actualizado en memoria.

- Técnicas y procesos de debugging: Generar mains que invoquen a las funciones elementales (previamente se las definió momentáneamente como globales, para poder acceder a ellas) y utilizar distintos datos de entrada para efectuar pruebas.

- La importancia de utilizar la ABI, para poder acoplar un programa creado en un lenguaje de alto nivel con otra parte del código que está implementada en un lenguaje de bajo nivel, sin generar interferencias y logrando el acople necesario para que resulte funcional.

- Familiarizse con los manuales de un determinado procesador, y comprender el funcionamiento de su arquitectura (en este caso, MIPS32).

- Notar que se pueden programar funciones en distintos lenguajes, siempre y cuando se respeten las convenciones de llamada de funciones y sus prototipos, para luego unificar todo en un sólo programa. Esto permite aprovechar las cualidades individuales de cada programador, sacando el máximo beneficio del lenguaje del que tenga conocimiento, y luego juntarlo con lo que han hecho otros en un lenguaje de programación diferente, sin tener que invertir tiempo y dinero, además desperdiciar sus conocimientos previos, haciendo que un programador experto en un determinado lenguaje tenga que migrar hacia otro.

- Comparar la facilidad de programación y escalabilidad que nos brinda un lenguaje de alto nivel contra el poder tomar dominio de la arquitectura del procesador en un lenguaje de bajo nivel, en momento críticos del programa en caso de ser necesario.

- Ser consciente de todas las operaciones que realiza la máquina por cada sentencia de alto nivel que nosotros escribimos, para a la hora de programar sectores críticos en cuanto a eficiencia, poder tomar decisiones correctas con menos desperdicios de recursos.

- Puede notarse que en sectores críticos, uno podría programar dicha sección en un lenguaje de bajo nivel, y si la complejidad de la aplicación lo permite, traer las variables de memoria a registros y trabajar intensamente con ellos (en especial si son variables que cambian constantemente de valor o son temporales) ahorrando el tiempo que requeriría la computadora en estar actualizando todos los datos en memoria (aunque es cierto que gran parte de este trabajo, lo realiza el compilador en caso de poner algún tipo de optimización).