Arquitectura de Computadoras
TP2 - Assembler y System Calls

Formato de archivos ejecutables y librerías

El formato que utiliza Linux para archivos ejecutables y librerías es ELF[1]. En los archivos de Assembly existen dos importantes secciones, “.text” y “.data”. En líneas generales “.text” es la zona de código y de todos los elementos que no deben modificarse, si se intenta modificar el sistema operativo puede generar un error (dependiendo de la implementación). La zona “.data” es la sección que puede contener datos inicializados que se pueden modificar pero no se pueden ejecutar. Si se intenta ejecutar, idealmente se producirá un error. Una sección adicional es la “.bss”, que no ocupa espacio en el archivo binario, sino que se inicializa con ceros cuando el programa es cargado en memoria. Se lo conoce mnemotécnicamente cómo “better save space”.

[1] http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Nota: vea la página http://www.nasm.us/doc/nasmdoc3.html para ver las directivas para reservar memoria, declarar datos y constantes.

Nota: puede consultar distinta información sobre la arquitectura de una pc intel en http://stanislavs.org/helppc/

Instrucciones para esta guía

Analizar y probar los fuentes de ejemplo para esta guía. Los archivos README que acompañan a cada uno, contienen **descripciones y explicaciones** adicionales importantes.

Punto de entrada

Es necesario conocer el punto de entrada al programa para saber cual es la primera instrucción. Esto se hace con la etiqueta "**_start:**", que da el pie a la primera instrucción que se va a ejecutar.

Stack

Para llamar a las funciones, es necesario el uso de un stack, de esta forma, se puede "conocer" la historia de dichas funciones. Ya que cuando se llama a una función, se guarda en el stack la dirección de memoria próxima a ejecutar al volver de dicha función.

El stack además es una buena zona para declarar variables privadas, (automáticas), ya que cuando termina la función que las declaró, el stack debería volver al estado en el que fue entregado para no tener ningún problema. Es decir, al momento de ejecutar la instrucción ret, el stack debe estar como se recibió.

Linux reserva una buena cantidad de espacio para el stack. Se puede conocer dicho tamaño mediante el comando.

```
$> ulimit -s
```

Si se llenase, nuestro programa abortaría con un error.

Ejercicio 1 – Hello

Hacer un programa que escriba por salida estándar "Hello World". ¿Qué es el valor **10** en el archivo de ejemplo?

Ejercicio 2 – Happy

Hacer un programa que defina, en una zona de datos, una cadena de caracteres con el siguiente string: "`h4ppy c0d1ng`" y la convierta a mayúscula. El resultado debe ser "`H4PPY C0D1NG`". Muestre por consola el resultado. Utilice como convención que los strings están terminados en 0. Implemente funciones.

Ejercicio 3 – num2str

Agregar a la biblioteca una función que recibe un número y una zona de memoria, y transforme el número en un string, terminado con cero, en la zona de memoria pasada como parámetro.

Nota: Puede utilizar la instrucción div o idiv

Ejercicio 4 – suma

Dado un número n, imprimir la suma de los primeros n números naturales (No utilizar una fórmula).

Ejercicio 5 – multiplos

Dado un número n y un valor k, imprimir todos los valores múltiplos de n desde 1 hasta k.

Ejercicio 6 – fact

Dado un número n, imprimir su factorial. Tenga cuidado con los argumentos de la función.

Ejercicio 7 – menor

Escribir un programa que dado un array de números enteros, de 4 bytes, encuentre el menor, y lo imprima por salida estándar.

Ejercicio 8 – ordenar

Escribir un programa que ordene un array de números enteros, de 4 bytes, e imprima el resultado ordenado por pantalla.

Argumentos de línea de comando

Los programas cuando se ejecutan por línea de comando pueden recibir argumentos. Estos son útiles para modificar el comportamiento de dicho programa. Por ejemplo, cuando ejecutamos:

```
$> vi hello.asm
```

El programa vi recibe el string “hello.asm” en uno de sus argumentos, y éste sabe que tiene que buscar dicho archivo y abrirlo.

En C, para hacer uso de los argumentos del programa, uno tiene que declararlos en el main. Un programa de ejemplo:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
      printf("Cantidad de argumentos: %d\n", argc);
```

```
	int i = 0;
	while(argv[i] != NULL) {
		printf("argv[%d]: %s\n", i, argv[i]);
		i++;
	}
	return 0;
}
```

Imprime por pantalla en la pantalla los argumentos del programa. Por ejemplo, si se ejecutara:

```
$> ./argumentos hola mundo por línea
```

Daría como resultado:

```
Cantidad de argumentos: 5
argv[0]: ./argumentos
argv[1]: hola
argv[2]: mundo
argv[3]: por
argv[4]: línea
```

Para destacar, el primer argumento es siempre el nombre del programa que se está corriendo y el último es NULL. Es decir, que argv[argc] == NULL

Hay que tener en cuenta, que esto es independiente del lenguaje, es decir, es parte del funcionamiento del sistema operativo. Que se puedan recibir como argumentos de main en es una configuración inicial que C hace antes de llamar a nuestro código. Si fuera en Java, la JVM, toma estos argumentos, y luego crea un objeto de tipo String[ ].

Un sistema tipo Unix, el Sistema Operativo, antes de darle el control al programa configura el stack para que tenga los argumentos del programa. Cuando el programa recibe el control, el registro ESP queda configurado con los siguientes datos:

ESP	Cantidad de Argumentos
ESP + 4	Path al programa
ESP + 8	dirección del 1er argumento
ESP + 12	dirección del 2do argumento
ESP + 16	dirección del 3er argumento
ESP + (n+2)*4	dirección del n-argumento
	NULL (4 bytes)
	dirección de la 1er variable de entorno
	dirección de la 2da variable de entorno
	dirección de la 3ra variable de entorno
	dirección de la n-variable de entorno
	NULL (4 bytes)

Por ejemplo, para saber cuál es la cantidad de argumentos del programa, habría que ejecutar una instrucción al comienzo del programa

```
mov eax, [esp]
```

Pero no es práctico utilizar esp, así que el valor de esp se guarda en ebp, y se arma una estructura llamada **stack frame** que sirve para preservarlo.

```
mov ebp, esp
```

Es decir, que ahora la cantidad de argumentos se pueden acceder a través de ebp

```
mov eax, [ebp]
```

Ejercicio 9 – Cant args

Escriba un programa que imprima en pantalla la cantidad de argumentos que tiene dicho programa

Ejercicio 10 – argumentos

Expanda el ejercicio anterior para imprimir todos los argumentos de un programa, mostrando un resultado similar al ejemplo escrito en C.

Ejercicio 11 – Variable de entorno

Hay una variable de entorno llamada **USER**, que indica cuál es el usuario actual logueado en el sistema. Imprima en pantalla cuál es dicho usuario.

Ejercicio 12 – recorre

Escriba un programa que recorra el stack iterativamente desde el final hasta el principio, imprimiendo cuantos bytes se recorrieron, puede fallar con un error para cortar la iteración, en lugar de detectar el final del stack.

System calls

Lista de syscalls y sus argumentos: https://syscalls.gael.in/

Para leer documentación de cada función, puede también utilizar la página del manual de Linux. Por ejemplo, para entender el funcionamiento de write:

```
$> man 2 write
```

Interrupciones de software

Los programas se corren en espacios distintos de memoria que en el que está el Sistema Operativo (OS), que también es un programa. Si consideramos que el OS posee un gran set de funciones que pueden ser llamadas por los programas, resultaría muy poco práctico acceder a dichas funciones mediante la instrucción call de assembler. Por qué?

Para evitar el problema de conocer, para cada versión del sistema operativo, la posición exacta de cada función se utilizan las Interrupciones de Software que el OS administra haciendo uso de la interrupt descriptor table (IDT), que éste configura en el arranque del mismo para “indexar” las funciones utilitarias.

En el caso de Linux, se hace uso de la entrada **80h** para acceder al sistema operativo y colocando en el registro eax el número de syscall a utilizar y los argumentos en el resto de los registros (de ser necesarios) para la syscall llamada.

Ejercicio 13 – pid

Los programas que están corriendo tienen un valor identificatorio en el sistema operativo. Se llama “pid” o process id. Se puede listar la lista de los programas corriendo, su pid y otros datos corriendo el comando

```
$> ps ax
```

Investigue el system call getpid y escriba un programa que imprima su pid

Ejercicio 14 – fork

Existe una herramienta que le permite a Linux “duplicar” procesos. Esta syscall se llama fork. Cuando un proceso se duplica, se crea una nueva copia de este. El proceso original se llama **padre** y al proceso nuevo, se lo llama **hijo**. Dicha función devuelve un argumento particular para identificar cual es cual.

Se dice que la función fork retorna dos veces, porque cuando el proceso **padre** ejecuta fork y se retorna el control, ahora hay dos programas que volvieron de esa llamada.

Investigar cómo utilizarlo y realizar un programa que realice un fork, y que el proceso “hijo” imprima un mensaje “Soy el hijo” y su padre imprima un mensaje “Soy el padre”.

Nota: Ejecutar “man 2 fork” en la terminal para obtener parte de la información.

Ejercicio 15 – suspender

Escribir un programa que suspenda la ejecución del mismo por n segundos y luego termine (no utilizar ciclos ni rutinas que dependan de la velocidad de la PC).

Ejercicio 16 – sys read

Investigar la manera de llamar al syscall read. Realizar un programa que reciba por entrada estándar (file descriptor 0) un string, lo convierta a mayúscula y lo imprima por salida estándar (file descriptor 1).

Ejercicio 17 – Código bugueado

Ud. encuentra este código en un sitio de internet para repasar el concepto y uso de system calls, el programador intentó realizar una rutina que lea una tecla de estándar input y que pasarla de minúscula a mayúscula, pero no funciona. Ud. debe encontrar y explicar todos los errores del código y mostrar la solución para que funcione correctamente.

No utilice la computadora para encontrar los errores. En tal caso, utilícela una vez resuelto el ejercicio, solo para confirmar su diagnóstico.

```
section .bss

section .data
      Buff resb 1

section .text
      global _start

_start:

Read:  mov eax,3      ; Specify sys_read call
       mov ebx,0      ; Specify File Descriptor 0: Standard Input
       mov ecx,[Buff] ; Pass offset of the buffer to read to
       mov edx,1      ; Tell sys_read to read one char from stdin
       int 80h        ; Call sys_read

       cmp eax,0       ; Look at sys_read's return value in EAX
       je Exit         ; Jump If Equal to 0 (0 means EOF) to Exit
                       ; or fall through to test for lowercase
       cmp byte [Buff],61h  ; Test input char against lowercase 'a'
       jb Write        ; If below 'a' in ASCII chart, not lowercase
       cmp byte [Buff],7Ah  ; Test input char against lowercase 'z'
       ja Write        ; If above 'z' in ASCII chart, not lowercase
                       ; At this point, we have a lowercase character
       sub byte [Buff],20h ; Subtract 20h from lowercase to give
                           ; uppercase...

                       ; ...and then write out the char to stdout
Write: mov eax,4       ; Specify sys_write call
       mov ebx,1       ; Specify File Descriptor 1: Standard output
       mov ecx,Buff    ; Pass address of the character to write
       mov edx,1       ; Pass number of chars to write
       int 80h         ; Call sys_write...
       jmp Read        ; ...then go to the beginning to get another
                       ; character

Exit:  mov eax,1       ; Code for Exit Syscall
       mov ebx,0       ; Return a code of zero to Linux
       int 80H         ; Make kernel call to exit program
```

Por otro lado encontró en la documentación de Linux, la siguiente información y pudo verificar que es correcta con respecto a los system call y la tabla ascii

```
1. sys_exit
Syntax: int sys_exit(int status)
```

```
Source: kernel/exit.c
Action: terminate the current process
```

```
2. sys_fork
Syntax: int sys_fork()
Source: arch/i386/kernel/process.c
Action: create a child process
```

```
3. sys_read
Syntax: ssize_t sys_read(unsigned int fd, char * buf, size_t count)
Source: fs/read_write.c
Action: read from a file descriptor
```

```
4. sys_write
Syntax: ssize_t sys_write(unsigned int fd, const char * buf, size_t count)
Source: fs/read_write.c
Action: write to a file descriptor
```

Ejercicio 18 – Preguntas

Un alumno que todavía no cursó esta materia le pide a usted que le explique:

1. ¿Qué es una system call? ¿Quién la escribió? ¿Dónde se encuentran ubicadas?
2. ¿Cómo se llama a una system call desde assembler de Linux? Muestre con un ejemplo de llamada a read.
3. ¿Qué es el número de ID?

Como usted no recuerda de memoria el funcionamiento ingresa a un sistema operativo Linux y ejecuta los comandos “man 2 read” y obtiene la siguiente salida:

```
READ(2)                Linux Programmer's Manual                READ(2)

NAME  top
       read - read from a file descriptor

SYNOPSIS    top
       #include <unistd.h>

       ssize_t read(int fd, void buf[.count], size_t count);
DESCRIPTION top
       read() attempts to read up to count bytes from file descriptor fd
into the buffer starting at buf.

       On files that support seeking, the read operation commences at the
file offset, and the file offset is incremented by the number of bytes
```

V20241117 0302

```
read.  If the file offset is at or past the end of file, no bytes are
read, and read() returns zero.

       If count is zero, read() may detect the errors described below. In
the absence of any errors, or if read() does not check for errors, a
read() with a count of 0 returns zero and has no other effects.

       According to POSIX.1, if count is greater than SSIZE_MAX, the
result is implementation-defined; see NOTES for the upper limit on Linux.
```

%eax	Name	%ebx	%ecx	%edx	%esi	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-