Introducción a la programación en el lenguaje Ensamblador

HISTORIAL DE REVISIONES			
NÚMERO	FECHA	MODIFICACIONES	NOMBRE
v1.0.0	2015 Mars 5		CA

Índice

1. Introducción

1.1. Objetivos de la Practica

Objetivos específicos:

- **Toolchain**:Practicar con las herramientas de edición, compilación *gcc*, ensamblaje *as*, encadenador *ld* .
- **Depuración**: Depurador *gdb*. Ejecutar un programa en modo *paso a paso* con el debugger.
- **Análisis**:Practicar con las herramientas de análisis *readelf*, *objdump*. Diferenciar entre el módulo objeto reubicable y el módulo objeto ejecutable.Desensamblar el código ejecutable.
- **Estructura**:Comprender la estructura de un módulo fuente mínimo en lenguaje ensamblador.
- **Lenguaje Ensamblador**:Familiarizarse con el lenguaje de programación en ensamblador GNU_Assembly *gas* basado en el lenguaje desarrollado por AT&T
- **Traductor Ensamblador**:Conocer las directivas del traductor assembler *as*.
- **Lenguaje Máquina**:Familiarizarse con la representación de direcciones, datos e instrucciones en lenguaje máquina
- **Hardware**:Comprender la arquitectura de un computador, cpu y memoria, desde el punto de vista del programador: memoria interna y memoria externa.
- **Kernel**:Comprender la relación entre un proceso de usuario y el sistema operativo a través de las *llamadas al sistema*.

Objetivos generales:

- Programación en lenguaje ensamblador
- Comprender el funcionamiento de la computadora desde el punto de vista de un programador de bajo nivel.
- Relacionar las propiedades de un lenguaje de alto nivel con un lenguaje de bajo nivel.
- Utilizar herramientas de desarrollo de bajo nivel.
  - Desarrollar habilidades de intuición y deducción en el empleo de las herramientas y en buscar qué herramientas.

Por lo tanto el lenguaje ensamblador puede ser utilizado como una herramienta para descubrir la arquitectura y el funcionamiento de la computadora. No es el objetivo central de esta asignatura ser un experto en lenguajes de programación ni en el desarrollo de algoritmos, objetivos centrales de otras asignaturas.

1.2. Recursos

- Disponible en miaulario:
  - Los módulos fuente as_groundup.tar utilizados están disponibles en el servidor de miaulario de la UPNA.
  - El libro de texto en que se basan los guiones de prácticas en lenguaje ensamblador : [programming_from_ground_up].
  - Hojas de referencia rápida
- Libros de texto: ver apartado de Bibliografía

1.3. Evaluacion

- Es necesario guardar todos los programas utilizados (fuentes y binarios) en el pendrive ya que el contenido puede ser evaluado en cualquier momento durante las prácticas a lo largo de todo el curso y durante el examen.
- Las respuestas a las cuestiones que se plantean a lo largo de las prácticas estarán respondidas en la memoria.
- Los conceptos utilizados en prácticas tanto sobre la arquitectura de un computador como sobre las herramientas de desarrollo forman parte del contenido a evaluar en las pruebas parciales y finales.
- Entrega de la memoria:
  - La semana de la sesión de prácticas antes del sábado a las 14:00
  - Se entregará **un único archivo** en formato **PDF**.
  - Nombre del archivo: apellido1_apellido2_tituloguionpractica.pdf
  - El archivo de contener:
    I. los datos personales del alumno
    II. preguntas del guión y respuestas.
    III. módulos del código fuente

1.4. Herramientas

- Es necesario para la ejecución de la práctica:
  - LIBROOOOOOOOOOOOOOOOOOOOO
  - REFCARDSSSSSSSSSSSSSSSSSSS
  - Hojas con los ORGANIGRAMAS
  - Tomar Apuntesssssssssssssss
- Fuentes en lenguaje AT&T:
  - exit.s
- Plataforma: GNU/linux(AMD64)/x86_64(Intel ó AMD)
- Recursos GNU:
  - Herramienta integrada de desarrollo: Emacs,u otra herramienta IDE.
  - as : ensamblador del lenguaje AT&T
  - ld : linker
  - cc : compilador de C
  - gcc : driver de compilación
  - gdb : depurador.
  - readelf: metainformación del fichero ELF
  - objdump: volcado de metainformación de los módulos objeto reubicable .o y ejecutable
- Disponible en red y en local(/usr/share/doc ó info ó man):
- Libro de Prácticas
  - http://programminggroundup.blogspot.com.es/2007/01/programming-from-ground-up.html
- Assemblers
  - https://sourceware.org/binutils/docs/as/

  - https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html : AT&T
  - https://software.intel.com/en-us/articles/introduction-to-x64-assembly : INTEL
- Fundamental
  - https://sourceware.org/binutils/docs-2.26/as/i386_002dDependent.html#i386_002dDependent: syntax, mnemonics, register
- Mnemónicos
  - http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/toc.htm: i386 (32 bits)
  - http://www.felixcloutier.com/x86/ : amd64 (64 bits)
  - https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html : AT&T
- Arquitectura (Registros,...)
  - https://en.wikipedia.org/wiki/FLAGS_register
  - https://software.intel.com/en-us/articles/introduction-to-x64-assembly: Intel oficial Vol 1 Basic Architecture
  - http://support.amd.com/TechDocs/24593.pdf : AMD oficial. Vol 3. 2.3 Summary of Registers and Data Types
- Manuales GNU
  - $ `man`
  - manual_GDB
  - Emacs
  - binutils: as,ld,objdump,...
  - gcc

2. Programación

2.1. Metodología

- Leer la descripción del programa
- Pseudocódigo:
  - Desarrollar el algoritmo definiendo las estructuras de datos y estructuras de instrucciones.
    - constantes,variables, arrays, punteros, inicalizaciones, bucles, sentencias selección, funciones y parámetros, entrada y salida del programa, etc
- Organigrama para un lenguaje de alto nivel
- Programación en lenguaje de alto nivel más comentarios
- Comando de compilación y ejecución.
- Test mediante el debugger y validación.
  - Ejecución en modo paso paso mediante la herramienta Debugger.
- Pseudocódigo:
  - Traducir el pseudocódigo de alto nivel a pseudocódigo de bajo nivel
    - secciones, variables, arrays, punteros, inicializaciones, bucles, sentencias selección, subrutinas y parámetros, entrada y salida del programa etc.
- Traducir el Organigrama de alto nivel a un Organigrama en un lenguaje de bajo nivel
- Programación en lenguaje de bajo nivel más comentarios
- Test mediante el debugger y validación.
  - Ejecución en modo paso paso mediante la herramienta Debugger.

2.2. Módulo Fuente: Cabecera

- Cabecera

```
Programa:       suma.s
Lenguaje:       ensamblador x86-64 AT&T
Descripción:   Suma de los primeros 5 números naturales
Entrada:        Ninguna
Salida:         echo $?
Compilación:   gcc -nostartfiles -o suma suma.s
S.O:            GNU/linux 3.2.0 ubuntu 12.04 x86-64
Librería:      /usr/lib/x86_64-linux-gnu/libc.so
CPU:            Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
Compilador:     gcc version 4.6.4 (Ubuntu/Linaro 4.6.4-1ubuntu1~12.04)
Ensamblador:    GNU assembler version 2.22
Linker/Loader:  GNU ld (GNU Binutils for Ubuntu) 2.22
Asignatura:     Estructura de Computadores
Fecha:          20/09/2016
Autor:          Cándido Aramburu
```

3. Depuración

3.1. Conceptos

- Depurador GDB de la fundación GNU: Gnu DebuGger
- Ideas a poner en práctica

```
Compilar incluyendo la tabla de símbolos
Arrancar debugger en modo texto o en modo ventanas
Cargar módulo a depurar y tabla de símbolos
modo paso a paso
Comenzar la ejecución hasta la ruptura
Entorno  de ventanas
visualizar memoria
ayuda
Simbolos de referencia a memoria: Direcciones y contenidos
Formatos y tamaños
instrucciones y datos
endianess
registers
Completion
segmentos de los registros
Salir del bucle
Terminar la ejecución
Reiniciar
Desensamblar
Salir
```

3.2. Comandos

- Incluir la tabla de símbolos en el módulo objeto ejecutable
  - `gcc -g -nostartfiles -o suma suma.s`
  - `gcc -g -o exit exit.c`
- Comandos a practicar

- Para su realización es necesario que el programa a depurar tenga definida alguna variable en la memoria principal e inicializado algún registro.

```
$ gdb
C-x a
C-x a
q
$ gdb -tui
file sum
info source
info sources
b _start
info breakpoints
run
h layout
h focus
focus src : navegar
focus cmd : navegar
C-x a
C-l
p _start
h p
h x
x _start
x /bt _start, x /ht _start, x /wt _start, x /gt _start
x /i _start
p símbolo_variable
x símbolo_variable
x &símbolo_variable
x /h &símbolo_variable  -> little endian
n
info registers
info reTAB
p $rax
p $eax
p $ax
p $ah
p $rflags
p $eflags
p /t $eflags
p $rip
n
RETURN
ejecutar la penúltima instrucción del bucle
until
c
start
pending? n
info breakTAB
disas /r _start
disas /m _start
layout split
layout src
q
```

3.3. Comentarios

- Comentarios
  - nota: si lanzamos `gdb exit &` no aparece el prompt (gdb)

- Ayuda
  - https://sourceware.org/gdb/onlinedocs/gdb/Readline-Movement-Commands.html#Readline-Movement-Commands

3.4. EMACS: Depurador GDB

- `emacs exit.c &`
- M-x gdb
- M-x gdb-many-windows
  - Emacs Debugger

```
Current directory is ~/tutoriales/C_tutorial/groundup/exit/standard/
GNU gdb (GDB) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from exit...done.

(gdb)
```

importante
Observar que se ha leído la tabla de símbolos: "Reading symbols from exit. . . done."--→ DONE!!!

4. exit.c (I)

- Antes de realizar la programación en lenguaje ASM la realizamos en lenguaje C.
- Objetivo del programa: Salir del programa entregando un parámetro al kernel linux.
  - Utilizamos la función *exit()* de la librería *libc*. Atención: no utilizar la función *_exit()*.
  - Ayuda: `man 3 exit`. Apuntes de C (librería libc)

4.1. Modulo Fuente

- `exit.c`

```c
#include <stdlib.h>

void main (void)
{
exit (0xFF);
}
```

4.1.1. Librería libc

- La función *exit(0xFF)* es una función de la librería standard *libc.so*.
- La función *exit(0xFF)* está definida en la cabecera *stdlib.h*.
- Una librería es un agrupamiento de funciones (exit(),printf(),getchar(), etc) que suelen estar codificadas en un único fichero como la librería standard de C "libc".

4.1.2. Compilación

- `gcc -o exit exit.c`
  - `ls -l` comprobar que se ha generado el módulo objeto ejecutable *exit*
  - `file exit.c` imprime las propiedades del fichero *exit.c*
  - `file exit`: imprime las propiedades del fichero *exit*
- `gcc --save-temps -o exit exit.c`
  - Si utilizamos la opción --save-temps : genera el módulo ensamblador, módulo objeto y módulo ejecutable
  - `ls -l`, comprobar que se han generado los módulos .s y .o
  - `file exit.s` imprime las propiedades del fichero *exit.s*
  - `file exit.s` imprime las propiedades del fichero *exit.o*
- `gedit exit.s`
  - Observar las instrucciones del lenguaje ensamblador *AT&T*
  - Observar las directivas del traductor ensamblador *as*

4.1.3. Ejecución

- `ls -l exit*`: compruebo que me encuentro en la carpeta que contiene los módulos fuente y ejecutable.
- `file exit` : compruebo que tenemos disponible el módulo objeto ejecutable en el formato ELF.
- `./exit` : Ejecución del módulo objeto ejecutable.
- `echo $?` : Imprime el valor entregado por el programa *exit* al kernel *linux*.

4.1.4. Cuestiones

1. Módulos objeto reubicable y ejecutable?
   - ¿Tamaño en bytes?
2. Módulo ensamblador
   - Instrucciones del módulo ensamblador y su significado
   - Símbolo main
     - qué tipo de símbolo es en la sintaxis del lenguaje ensamblador
     - significado de las directivas asociadas a main

5. exit.s (I)

- Objetivo del programa: Salir del programa entregando un parámetro al kernel linux.
- Hay que relacionar este módulo fuente en lenguaje ensamblador con el módulo fuente del ejercicio anterior en lenguaje C.

5.1. Módulo fuente

- Lenguaje AT&T 32 bits

```
###     PURPOSE

###     INPUT

###     OUTPUT echo $?

###     VARIABLES

        .section .data

        .section .text

        .globl _start
_start:                         # program entry point
        movl $1, %eax
        movl $0xFF, %ebx
        int $0x80
```

- Lenguaje AT&T 64 bits

```
###     PURPOSE         programa con el código mínimo

###     INPUT           ninguna

###     OUTPUT          echo $?

###     VARIABLES       ninguna

        .section .data

        .section .text

        .globl _start
_start:                         # dirección del contador de programa al ejecutar el  ↩
    proceso
        movq $60, %rax
        movq $0xFF, %rdi
        syscall


        .end
```

5.1.1. Compilación 64 bits

- Compilar
  - Con el front-end `gcc`:

```
gcc -nostartfiles -o exit exit.s
```

    - *-nostartfiles*: opción para indicar que la función principal no se llama **main**
  - Con el toolchain: `as-ld`

```
as -o exit.o exit.s
ld -o exit exit.o
```

5.1.2. Ejecución

- Ejecución:

```
./exit
echo $?
```

5.1.3. Cuestiones

1. ¿ Qué programa llama al objeto ejecutable *exit* resultado de compilar exit.s?
2. ¿ Qué valor devuelve la función exit.s y a qué programa o proceso se lo devuelve?
3. ¿ Qué directivas hay en el programa fuente y que significa cada una ?
4. ¿ Qué etiqueta es obligatoria para *as*, de qué tipo es y qué función desempeña?
5. ¿ Cual es el tamaño de los registros de propósito general RAX y RDI?
6. ¿ Por qué vale 60 el contenido del registro RAX?
7. ¿ Cuál es la función del registro RDI en este módulo?
8. ¿ Que función realiza la instrucción `syscall`?

5.1.4. Cuestiones

- Responder a las siguientes cuestiones:

1. ¿Cual es son las funciones de los registros?
   - RIP,RSP,RAX,RDI,RFLAG
2. ¿Cual ha sido el contenido del registro RFLAG?¿y el del flag de Signo SF?
3. ¿En que dirección de memoria se inicia el programa?
4. ¿Cuál es la dirección de memoria de la última instrucción del programa?
5. Especificar cuantos bytes ocupa cada instrucción.
6. ¿Qué comando GDB imprime el contenido del contador de programa? Indicar el contenido antes de ejecutar la última instrucción.
7. ¿A qué objeto apunta el puntero contador de programa antes de ejecutar la última instrucción?¿De qúe tipo es dicho objeto?
8. Examinar la memoria a la que apunta el contador de programa antes de ejecutar la última instrucción.

6. exit.c (II)

6.1. Desemsamblaje

1. La herramienta *objdump* desde un terminal.
   - obtener el módulo objeto reubicable con `gcc -g -o exit.o exit.c`
   - `$objdump -S exit.o`

```
exit.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
  Al finalizar la ejecución ejecutar en el terminal: echo $? y comprobar que  ↩
      visualizamos el valor de retorno.
*/
#include <stdlib.h>

int main (void)
{
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
  exit(0xFF);
   4:   bf ff 00 00 00          mov    $0xff,%edi
   9:   e8 00 00 00 00          callq  e <main+0xe>
```

6.2. Cuestiones

- ¿Qué operación realiza la sentencia en ensamblador `callq 0x4003b0 <exit@plt>` ?
- ¿Qué relación tiene el valor 255 con el módulo fuente en C?
- ¿Qué relación tiene el registro RDI con el módulo fuente en C?
- ¿Qué instrucción del código ensamblador utiliza el registro RDI?
  - ¿Cuales son los operandos fuente y destino de dicha instrucción?
  - ¿Qué tipos de direccionamiento utiliza dicha instrucción?
- ¿En que dirección se encuentra la primera instrucción máquina de la función main?
  - ¿Cual es la primera instrucción máquina de la función main?

7. exit.s (II)

7.1. Análisis de los diferentes módulos exit

7.1.1. Módulos a analizar

- Se pueden analizar los módulos objetos ejecutable y reubicable tando del programa **exit.c** como del programa **exit.s**

7.1.2. Herramientas

- Utilización de las herramientas de análisis de GNU:
  - file, readelf (Displays information about ELF files), objdump
  - las herramientas readelf y objdump son casi equivalentes. readelf es más detallista que objdump y objdump permite desensamblar.
- manuales: man file ,man readelf, man objdump

7.1.3. Análisis

- Análisis de los Modulos Objeto Binarios

```
file exit.o                    -> propiedades del módulo objeto reubicable
file exit                      -> propiedades del módulo ejecutable
size exit.o                    -> Tamaño de cada sección y el total
size exit                      -> Tamaño de cada sección y el total
readelf -x .text  exit         -> imprime la sección .text en hexadecimal
objdump  -d exit.o             -> Disassembly of executable section: la salida está en  ↩
    ASM y binario
objdump  -S exit.o             -> Disassembly of executable section: la salida está en  ↩
    los lenguajes C(si existe), ASM y binario
objdump  -t exit.o             -> imprime la tabla de símbolos, se puede comprobar si hay ↩
     secciones .debug para el Depurador
```

7.1.4. Cuestiones

1. Comparar el tamaño de los módulos reubicable y ejecutable. ¿A qué es debido?
2. Comparar el tamaño de los módulos ejecutables correspondientes a los módulos fuente exit.c y exit.s
3. Indicar del módulo objeto la clase de objeto, el tipo de datos, máquina, sistema operativo, interfaz ABI y tipo de módulo.
4. ¿ Cuál es el punto de entrada del módulo reubicable y cuál el del módulo ejecutable?
5. Si observamos la columna de direcciones de cada uno de los dos módulos objeto ¿Cuál es la diferencia?¿Cuál es la relación con el linker?Nota:es necesario que el módulo objeto ejecutable se haya obtenido a partir del assembler as y ld y no del gcc.
6. La sección de texto del módulo ejecutable en que dirección virtual de la memoria comienza y cual es su tamaño.
7. ¿ Cuántas secciones tiene el módulo objeto?
8. ¿ Cuál es el código máquina de la instrucción `syscall` ?
9. Si observamos la compilación de *exit.s* en que dirección se encuentra el punto de entrada a la función exit() de la librería libc.

8. Apéndice

8.1. Compilación

8.1.1. Librerías

- `gcc -o exit exit.c`
  - Durante la compilación se enlaza automáticamente con la librería libc.
    - Equivale a compilar `gcc -o exit exit.c -lc` donde la opción *-l* explicitamente indica enlazar con libc.
    - Las librerías tienen todas ellas el prefijo *lib* (libc,libm,libxxx) que no se escribe junto a la opción *-l*

8.1.2. Compilación de un módulo AT&T 32 bits

1. Altenativa *gcc*
   - `gcc -m32 -nostartfiles -o exit exit.s`
2. Alternativa *as-ld*
   - `as --32 -o exit.o exit.s`
   - `ld -melf_i386 -o exit exit.o`

8.2. Enlazamiento: linker

8.2.1. Procedimiento

- El linker tiene que enlazar los diferentes modulos objeto reubicables **.o* para lo cual debe de resolver el valor de las direcciones de memoria al que hacen referencia los diferentes módulos.
- En C normalmente utilizaremos funciones de la librería standard libc.
- Aunque no especifiquemos ninguna librería al compilar un módulo fuente, el linker siempre enlaza automáticamente con funciones del sistema operativo necesarias para la correcta ejecución de nuestro programa.
- Este enlazamiento se puede realizar de forma estática o dinámica
  - El enlazamiento estático se realiza antes de la ejecución y los módulos **.o* quedan enlazados en el módulo final ejecutable.
  - El enlazamiento dinámico se realiza durante la ejecución del módulo ejecutable.
    - Por ejemplo el enlazamiento con la función exit() de la librería dinámica libc.so se realiza dinamicamente. Es decir por un lado se carga en memoria la función *exit()* y por otro lado se carga en memoria el programa `exit` (resultado de compilar exit.s). Son dos cargas en memoria independientes. Durante la ejecución cuando nuestro programa `exit` llame a *exit* el linker tendrá que calcular la dirección donde comienza *exit*.

8.2.2. Estructura de los módulos binarios

- `objdump -S exit.o`
- `objdump -S exit`

8.3. Procedimiento de Ejecución del módulo objeto ejecutable

- Cuando ejecutamos un programa a través de la interfaz shell de gnu/linux (prompt $), se llama al sistema operativo para la creación del proceso. Una vez creado el proceso, el S.O. lo llama para que la CPU lo ejecute. Una vez finalizada la ejecución del proceso se retorna al sistema operativo que lo llamó devolviendo este un valor al S.O.. Mediante el comando `echo $?` imprimimos el parámetro recibido por el sistema operativo

8.4. Llamadas al sistema

8.4.1. Descripción Llamada al Sistema EXIT (32 bits)

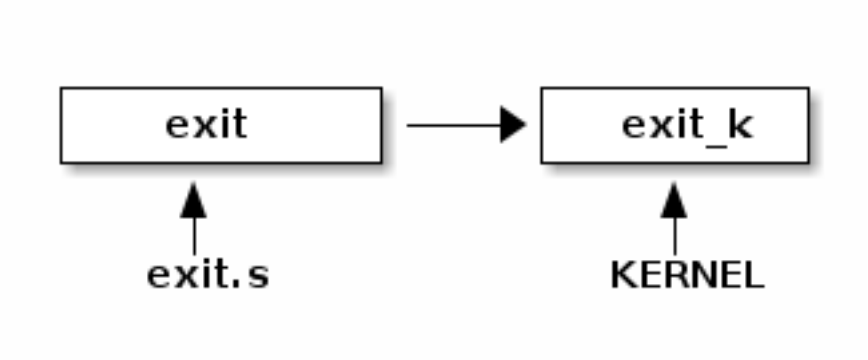
- El programa exit.s realiza una **llamada al sistema** a ejecutar por el kernel mediante la instrucción *int 0x80*. Los parametros de la llamada a la función se realizan mediante los registros EAX y EBX. EAX contiene el código que hace referencia al nombre de la función [exit] y EBX contiene el valor del argumento de la función [0xFF].
- La instrucción *int* recibe el nombre de interrupción software y el campo de operando *0x80* es una **llamada al sistema** o llamada al kernel del sistema operativo. El programa exit.s se **interrumpe** , a través de la instrucción ensamblador `int 0x80`, para realizar una llamada al sistema.

8.4.2. Descripción Llamada al Sistema EXIT (64 bits)

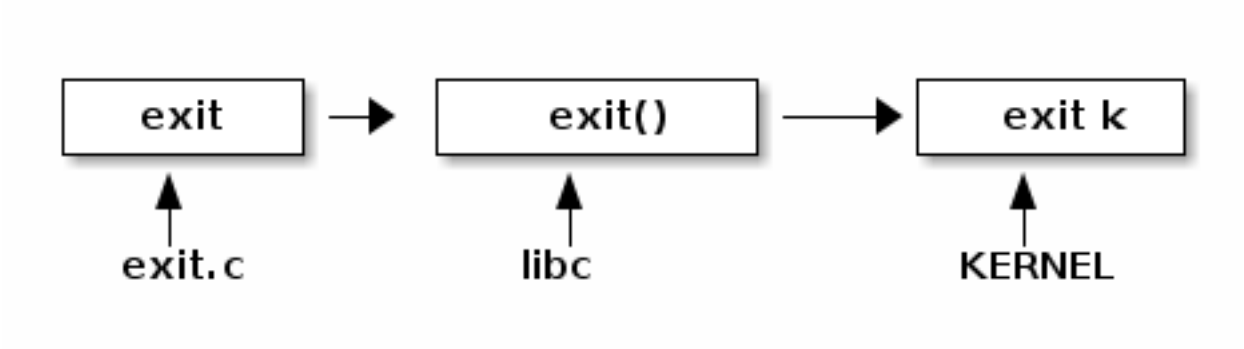
- El programa exit.s realiza una **llamada al sistema**, una llamada al kernel, mediante la instrucción *syscall*. Los parametros que se pasan al kernel, antes de la llamada, se realizan mediante los registros RAX y RDI. RAX contiene el código que hace referencia al nombre de la función que debe de ejecutar el kernel, en este caso el 60 es el código de la función EXIT .RDI contiene el valor del argumento de la función [0xFF].
- La instrucción *syscall* recibe el nombre de **llamada al sistema** o llamada al kernel del sistema operativo. El programa exit.s se **interrumpe** a través de la instrucción ensamblador `syscall`, para realizar una llamada al sistema.

- ¿la función exit del kernel que operación realiza? → manual: `man 3 exit`

importante
Todos los programas escritos en lenguaje ensamblador deben de finalizar mediante la llamada al sistema *EXIT*



8.4.3. Descripción llamada al sistema EXIT desde el lenguaje C



8.5. Registros punteros a la pila

- (gdb) p $rsp
- (gdb) p $sp → sp es un alias de stack pointer (top de la pila) por lo que equivale al registro $RSP
- (gdb) p $rbp
- (gdb) p $fp → fp es un alias de frame pointer (bottom de la pila) por lo que equivale al registro $RBP

9. Bibliografía

9.1. Libros

[1] [programming_from_ground_up] Programming from the Ground Up. Jonathan Bartlett: GNU Assembler, AT&T language

[2] [Paul Carter: Netwide Assembler, Intel language