



TEC | Tecnológico
de Costa Rica

Escuela de Ingeniería Electrónica

EL4313 – Laboratorio de Estructuras de Microprocesadores

Guía de inicio: Lenguaje ensamblador x86_64 para Linux **Parte 1: Ensamblado del primer programa**

Tabla de Contenidos:

1. Hardware de un sistema computacional de uso general:	3
1.1 ¿Cómo determino si mi computadora tiene un microprocesador de 64 bits?:.....	3
2. Sistema operativo – Software del sistema computacional:.....	4
2.1 ¿Cómo determino si mi computadora usa un sistema operativo de 64 o de 32 bits?:	4
3. Herramientas para construir un primer programa con ensamblador	5
3.1 ¿Cómo se verifica si un paquete de software está instalado en Linux?	5
3.2 ¿Cómo se instala un paquete de software en Linux (Debian/Ubuntu)?.....	6
3.3 ¿Qué paquetes de software necesito instalar para escribir un primer programa usando ensamblador x86_64?	7
NASM – Netwide Assembler:	7
Editor de texto – NANO (o algún otro de preferencia)	8
GDB: The GNU Project Debugger	8
4. Escribir, ensamblar, enlazar y ejecutar: el primer programa.....	9
4.1 Edición del código del programa:	10
4.2 Ensamblar:	11
4.3 Ligar o “Linkear” el programa:	12
4.4 Ejecutar el programa:.....	12
Análisis: ¿Qué es un “Segmentation fault”?	12
4.5 Depurar el programa usando GDB:.....	14
5. Manejo del teclado como periférico de entrada:	20
6. Control de flujo de ejecución del programa:	20
¿Qué es el flujo de ejecución de un programa?	20
Control de flujo:	22
Etiquetas:	22
Comparaciones:	23
Saltos:.....	23
Anexo: Tabla de llamadas de sistema para Linux x86_64.....	25

1. Hardware de un sistema computacional de uso general:

El microprocesador es la unidad principal de ejecución en un sistema computacional. Es decir, se encarga de leer instrucciones descritas en el software y ejecutarlas.

En las arquitecturas modernas de microprocesadores de propósito general para computadoras de escritorio (típicamente Intel y AMD) se trabaja con microprocesadores de 32 y de 64 bits, es decir, que cada una de las instrucciones o los datos que se intercambian con la memoria tiene un ancho de palabra de 64 bits.

La mayoría de paquetes de software operan en un modo de compatibilidad entre 64 y 32 bits, sin embargo hacia futuro se espera que la operación en 32 bits gradualmente sea desplazada por la de 64 bits de forma exclusiva.

En este proyecto se parte del hecho que se va a trabajar con arquitectura de hardware de 64 bits exclusivamente.

1.1 ¿Cómo determino si mi computadora tiene un microprocesador de 64 bits?:

Windows	Linux
<ul style="list-style-type: none">• Abrir una consola de comandos (cmd)• Usar el comando “set” y buscar las entradas:<ul style="list-style-type: none">• PROCESSOR_ARCHITECTURE• PROCESSOR_IDENTIFIER• Las entradas deben indicar una arquitectura de 64 bits, como la siguiente: PROCESSOR_ARCHITECTURE=AMD64 PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 58 Stepping 9, GenuineIntel	<ul style="list-style-type: none">• Abrir una terminal o consola• Usar el comando “lscpu”• Buscar la entrada CPU op-mode(s). Esta debe indicar si el procesador es capaz de operar en modo de 32 bits, 64 bits o ambos. Por ejemplo: CPU op-mode(s): 32-bit, 64-bit

2. Sistema operativo – Software del sistema computacional:

El sistema operativo, es el componente principal de software, que se encarga de administrar los recursos de hardware del sistema computacional y ofrecer a las diferentes aplicaciones de software los mecanismos para utilizarlos. El sistema operativo provee soporte a todas las funcionalidades del sistema de cómputo.

Existe mucha variedad de sistemas operativos que se pueden agrupar en 3 grandes familias: Apple, Microsoft y Linux.

En este proyecto se va a trabajar con Linux. Cualquier distribución de Linux debe ofrecer las funcionalidades necesarias para realizar el proyecto, sin embargo, las guías de instalación y los ejemplos descritos en este documento se fundamentan en el uso de la distribución Lubuntu (<http://lubuntu.net>), y por lo tanto, se recomienda usar una distribución semejante si no se está familiarizado con Linux.

Para este curso, se busca desarrollar capacidades de trabajar con el modo de operación de 64 bits. Para lograr esto, es necesario que tanto el microprocesador como el sistema operativo trabajen en 64 bits.

Ya se describió la forma en la que se puede determinar si el procesador soporta el modo de operación de 64bits, ahora se debe averiguar si el sistema operativo funciona en 64 bits.

2.1 ¿Cómo determino si mi computadora usa un sistema operativo de 64 o de 32 bits?:

Windows	Linux
<ul style="list-style-type: none">• Abrir una consola de comandos (cmd)• Usar el comando: wmic os get osarchitecture• La salida del comando debe indicar la arquitectura: OSArchitecture 64-bit	<ul style="list-style-type: none">• Abrir una terminal o consola• Usar el comando “uname -a”• La información va a indicar la arquitectura del sistema operativo. Por ejemplo: Linux user-pc 4.4.0-22-generic #40-Ubuntu SMP Thu May 12 22:03:46 UTC 2016 x86_64 GNU/Linux <p><i>Nota: Si la arquitectura mostrada es i686 o i686 significa que se tiene instalado un Linux de 32 bits</i></p>

3. Herramientas para construir un primer programa con ensamblador

Se parte del hecho que ya se ha instalado una versión de Linux de 64 bits como base para trabajar. Si no se cuenta con una instalación de Linux nativa en la computadora de trabajo, se puede optar por realizar una instalación nueva sobre una máquina virtual, de manera que desde Windows se tiene acceso a un Linux para realizar las prácticas.

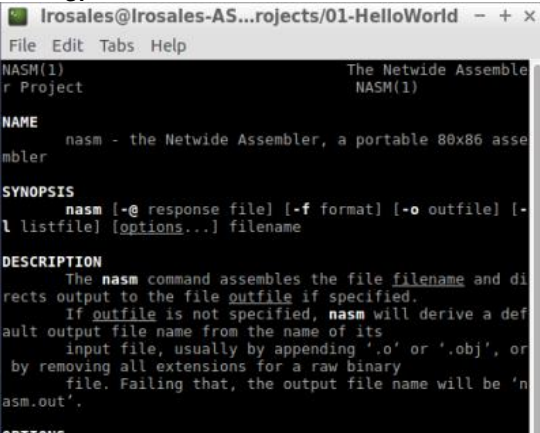
Si se desea utilizar la máquina virtual, puede descargar e instalar en su sistema Windows las siguientes herramientas (todas de uso libre):

Virtual Box: Sistema para construir y correr máquinas virtuales	https://www.virtualbox.org
Lubuntu Distribución de Ubuntu Linux de tipo “ligero”	http://lubuntu.net/
Ubuntu: Si prefiere una versión más completa de Linux, puede optar por usar Ubuntu	http://www.ubuntu.com/
Los pasos generales son: <ol style="list-style-type: none">1. Instalar Virtual Box2. Construir una nueva máquina virtual. Se recomienda como mínimo con 1GB de espacio de Disco Duro y 2GB de Memoria RAM.3. Configurar al menos 1 de los adaptadores de red de la máquina virtual para que funcione en modo NAT, ya que el sistema Linux va a necesitar conexión a Internet para algunas herramientas.4. Descargar la versión de Linux preferida, como un archivo de imagen (iso)5. Configurar la máquina virtual para instalar Linux a partir del archivo iso descargado.6. Instalar Linux en la máquina virtual. Como en cualquier otra versión de Linux, va a ser necesario definir su usuario.7. Una vez instalado, pruebe su acceso a Linux, y verifique que la máquina virtual tiene acceso a Internet (desde la terminal, ping www.google.com)	

3.1 ¿Cómo se verifica si un paquete de software está instalado en Linux?

En los sistemas Linux, se incluyen 2 comandos que son de utilidad para determinar si un determinado paquete de software está instalado:

whereis	help y/o manpages
<p>“whereis” es un comando que desde la terminal, busca un determinado paquete de software. Cuando lo encuentra, indica el folder o directorio dentro del sistema de archivos donde se encuentra instalado:</p> <pre>user@pc\$ whereis nano user@pc\$ nano: /bin/nano</pre> <p>Cuando no lo encuentra se indica una ruta en blanco:</p> <pre>user@pc\$ whereis warcraft user@pc\$ warcraft:</pre>	<p>La mayoría de paquetes de software en Linux incluyen una opción de ayuda, que muestra información sobre ¿cómo se usa? Dependiendo de la aplicación, se pueden usar diferentes opciones:</p> <p>-h: Desde la terminal se llama a la aplicación pero con el parámetro “-h” luego de la aplicación. Si la aplicación tiene una opción de ayuda o help, despliega la información.</p> <pre>user@pc\$ nasm -h usage: nasm -o[output file] -f [format] ...</pre>

	<p>man: Es el comando para “manual” y cuando se llama desde la terminal junto con un determinado nombre de aplicación, se muestra el manual creado para la misma.</p> <p>Para salir de la vista del manual, se usa la tecla q</p> <pre>user@pc\$ man nasm</pre> 
--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.2 ¿Cómo se instala un paquete de software en Linux (Debian/Ubuntu)?

En los sistemas Linux basados en distribuciones de Debian y Ubuntu, la instalación de paquetes y herramientas de software se trabaja utilizando el *Advanced Packaging Tool* o “**apt**” que es un administrador de paquetes que utiliza la conexión de la computadora a Internet para conectarse a un repositorio, desde el cual descarga todos los componentes de software necesarios para instalar el paquete y sus dependencias.

Entonces, los pasos para instalar un determinado paquete de software son los siguientes:

1. Debe contarse con acceso a Internet.
 - a. Se puede verificar con el navegador de internet o bien desde la terminal usando el comando **ping** a un sitio de internet (ej: **ping www.tec.ac.cr**)
2. Luego de confirmar el acceso a Internet, se deben actualizar las fuentes desde las que se descargan los paquetes o repositorios.

Para esto, se usa el comando **apt-get update**, precedido por un **sudo**.

El **sudo**, significa que la siguiente acción se hace como super-usuario, es decir: el usuario comprende que se va a modificar el sistema, y como tal, es probable que se deba ingresar el password del usuario.

```
user@linux$ sudo apt-get update
password for user: *****
Hit:1 http://cr.archive.ubuntu.com/ubuntu xenial InRelease
Get:2 http://cr.archive.ubuntu.com/ubuntu xenial-updates
InRelease [94,5 kB]
Get:3 http://security.ubuntu.com/ubuntu xenial-security InRelease
[94,5 kB]
```

```
Hit:4 http://cr.archive.ubuntu.com/ubuntu xenial-backports
InRelease
Fetched 189 kB in 0s (238 kB/s)
Reading package lists... Done
```

- Una vez actualizadas las fuentes, se solicita la instalación del paquete de interés usando el comando **apt-get install <nombre_del_paquete>**, igualmente precedido por **sudo**. Por ejemplo, para instalar el paquete de software llamado NASM, se utiliza este comando:

```
user@linux$ sudo apt-get install nasm
password for user: *****
Reading package lists... Done
Building dependency tree
Reading state information... Done
...a partir de este momento, se presentan instrucciones para la instalación en la
pantalla. Las instrucciones pueden cambiar dependiendo del paquete a instalar
```

3.3 ¿Qué paquetes de software necesito instalar para escribir un primer programa usando ensamblador x86_64?

Luego de completar la instalación del sistema operativo y familiarizarse un poco con Linux, además de confirmar que la computadora tiene el Hardware y el Software adecuados (64 bits), es necesario instalar 3 herramientas fundamentales:

Editor	Es una herramienta que permite editar archivos para escribir código. Cualquier editor de texto funciona, sin embargo hay algunos que ofrecen más ventajas que otros. En este tutorial, se utiliza NANO por facilidad de uso y de acceso desde una consola.
Ensamblador	Es la herramienta que toma el código escrito en un archivo y lo convierte a un archivo objeto o archivo en código máquina, es decir: instrucciones que el procesador maneja directamente.
Debugger	Es una herramienta que permite ejecutar un programa de forma controlada, es decir: paso por paso, permitiendo visualizar el estado de las diferentes variables, registros y otros elementos en cada una de las instrucciones ejecutadas.

Para esta guía, se van a utilizar los siguientes paquetes de software:

NASM – Netwide Assembler:

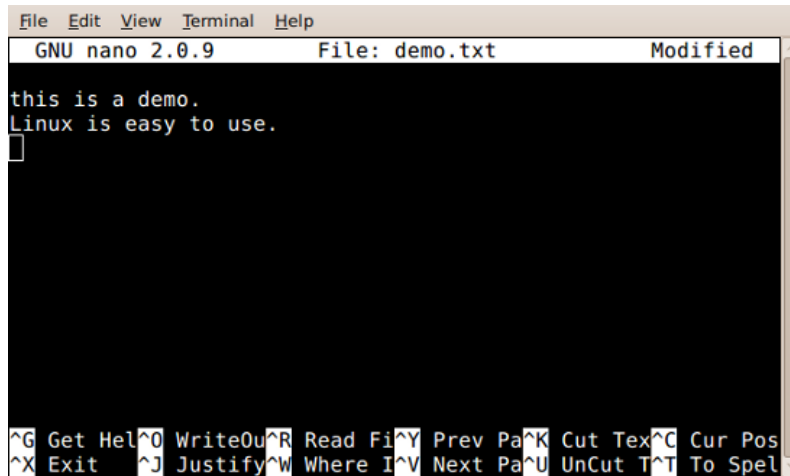
NASM es una herramienta para ensamblar/desensamblar programas de software en las diferentes arquitecturas de 32 y 64 bits. Opera en sistemas operativos Linux, y es de uso libre, aunque también existe una versión disponible para Windows, pero este curso se enfoca en utilizar Linux como sistema operativo base.

Puede leer más sobre NASM en: <http://www.nasm.us/index.php>

Editor de texto – NANO (o algún otro de preferencia)

En esta guía, como los ejercicios se manejan desde la línea de comandos (texto solamente, sin ambiente gráfico) se va a utilizar el editor llamado NANO, sin embargo se pueden usar alternativas como VIM, leafpad, gedit, etc.

Para más información sobre NANO, se puede visitar: <https://www.nano-editor.org/>



GDB: The GNU Project Debugger

GDB es una herramienta que permite ejecutar paso-a-paso un programa sobre el Linux, con la particularidad de ser ligero, fácil de portar y permite visualizar el estado de diferentes registros, variables y estructuras dentro del microprocesador en cualquier momento.

GDB se puede instalar con las indicaciones en la sección 3.2.

Para más información se puede visitar: <https://www.gnu.org/software/gdb/>

4. Escribir, ensamblar, enlazar y ejecutar: el primer programa.

Para el primer ejemplo, se va a construir un programa simple usando lenguaje ensamblador para arquitectura de 64 bits, que imprime un mensaje predefinido (Hola Mundo!) en la pantalla de la computadora.

El primer punto a comprender, es que el microprocesador cuenta con una estructura interna de registros: Cada registro tiene una función específica y almacena un código de operación o un parámetro para ejecutar la operación.

En función a los valores almacenados en los registros, cuando el programa haga un llamado al sistema operativo (Linux), este va a asignar recursos del microprocesador para realizar la operación indicada con los parámetros indicados.

La siguiente figura muestra los registros existentes en un microprocesador de 64 bits con capacidad de soporte de modo “*legacy*”. Para efectos iniciales, se va a trabajar con los 16 registros de propósito general y en modo de 64 bits, que se detallan en la tabla siguiente.

Register or Stack	Legacy and Compatibility Modes			64-Bit Mode		
	Name	Number	Size (bits)	Name	Number	Size (bits)
General-Purpose Registers (GPRs)	EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP	8	32	RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8–R15	16	64
128-Bit XMM Registers	XMM0–XMM7	8	128	XMM0–XMM15	16	128
64-Bit MMX Registers	MMX0–MMX7	8	64	MMX0–MMX7	8	64
x87 Registers	FPR0–FPR7	8	80	FPR0–FPR7	8	80
Instruction Pointer	EIP	1	32	RIP	1	64
Flags	EFLAGS	1	32	RFLAGS	1	64
Stack	–		16 or 32	–		64

Registro		Propósito
#	Nombre	
0	rax	Registro acumulador: Acarrea el resultado o el código de llamada de sistema a usar
1	rbx	Registro base
2	rcx	4to Argumento de llamada de sistema / registro contador
3	rdx	3er Argumento de llamada de sistema / paso de datos
4	rsi	2do Argumento de llamada de sistema
5	rdi	1er Argumento de llamada de sistema
6	rbp	Base Pointer – Indica el inicio del segmento de código en el stack
7	rsp	Stack Pointer – Indica la posición actual de la ejecución en el segmento de código
8	r8	Propósito variado Paso de argumentos adicionales Recolección de resultados
9	r9	
10	r10	
11	r11	
12	r12	
13	r13	
14	r14	
15	r15	

4.1 Edición del código del programa:

Un programa en ensamblador de 64 bits debe tener 2 secciones:

- **Datos:** Donde se definen las constantes que se van a utilizar
Para nuestro ejemplo, vamos a necesitar una constante para almacenar la cadena de texto que deseamos imprimir, y otra constante que almacena el tamaño de esta cadena de texto.

Entonces:

```
section .data
    cons_hola: db 'Hola Mundo!'
    cons_tamano: equ $-cons_hola
```

- **Código:** Donde se definen variables y se escribe de forma ordenada, los valores a cada registro y las invocaciones o llamadas al sistema para realizar las operaciones.

En este caso, lo que se desea es imprimir, para que Linux interprete adecuadamente la operación imprimir, se debe cargar los registros con esta configuración:

- **rax=1:** En rax se almacena el código de la operación que el sistema debe ejecutar. La operación imprimir (sys_write) es la número 1. La tabla al final del documento muestra los diferentes valores para llamadas de sistema que se pueden usar.
- **rdi=1:** En rdi se almacena el primer parámetro, que en este caso es indicar ¿Dónde va a imprimir el sistema? El valor 1, indica la salida por defecto, es decir: la pantalla o consola del sistema.
- **rsi=cons_hola:** En rsi se almacena el segundo argumento, que es la constante donde se definió la cadena de texto a imprimir
- **rdx=cons_tamano:** En rdx se almacena el 3er parámetro, que en este caso indica la cantidad de caracteres que se van a imprimir.

Luego de configurar los registros, se invoca al sistema operativo para que ejecute la operación.

Esto se hace mediante “**syscall**”

En resumen, el programa quedaría de la siguiente manera:

```
section .data
    cons_hola: db 'Hola mundo!',0xa
    cons_tamano: equ $-cons_hola

section .text
    global _start
_start:
    mov rax,1
    mov rdi,1
    mov rsi,cons_hola
    mov rdx,cons_tamano
    syscall
```

Se puede agregar comentarios para hacer que el código sea más fácil de leer. Los comentarios se agregan después de un punto y coma (;). Por ejemplo:

```
;-----Segmento de datos-----
;Aquí se declaran las constantes que se van a usar en el programa
section .data
    cons_hola: db 'Hola mundo!',0xa
    cons_tamano: equ $-cons_hola

;-----Segmento de código-----
;Contiene la secuencia de ejecución del programa
;La ejecución inicia en "_start", que es una etiqueta o referencia
section .text
    global _start
_start:
    mov rax,1
    mov rdi,1
    mov rsi,cons_hola
    mov rdx,cons_tamano
    syscall
```

El siguiente paso, es ensamblar y ligar el programa para convertirlo en un archivo ejecutable desde el sistema operativo.

4.2 Ensamblar:

Luego de editar el archivo donde se escribe el código (ej: hola-mundo.asm) se utiliza NASM para ensamblarlo.

Debe indicarse:

- El tipo de sistema de archivos para el que se está compilando.
 - Esto se hace con la opción **-f** y el valor **"elf64"**
- El archivo ensamblado de salida. Debe ser un archivo de tipo objeto (**.o**).
 - Por ejemplo: *hola-mundo.o*
- El archivo donde está el código en ensamblador que se va a ensamblar
 - Por ejemplo: *hola-mundo.asm*

Entonces, el proceso de ensamblaje sería:

```
user@linux$ nasm -f elf64 -o hola-mundo.o hola-mundo.asm
```

Si el ensamblaje se hace correctamente, la terminal no debería mostrar errores y se debería crear en el mismo directorio el archivo **.o** deseado. Recuerde que puede utilizar el comando **"ls"** para mostrar los archivos existentes en el directorio. Por ejemplo:

```
user@linux:~/Documents/$ ls
hola-mundo.asm
user@linux:~/Documents/$ nasm -f elf64 -o hola-mundo.o hola-mundo.asm
user@linux:~/Documents/$ ls
hola-mundo.asm  hola-mundo.o
```

4.3 Ligar o “Linkear” el programa:

El proceso de ligado o “link” es necesario para obtener un archivo ejecutable a partir del archivo ensamblado (**hola-mundo.o**)

En Linux esto se logra con el comando “**ld**”, indicando el nombre del ejecutable deseado y el archivo ensamblado que funciona como referencia.

Osea:

```
user@linux$ ld -o hola-ejecutable hola-mundo.o
```

Si el proceso de ligado es correcto, entonces no se presentan mensajes de error en la consola y se genera el archivo ejecutable en el directorio actual. Por ejemplo:

```
user@linux:~/Documents/$ ls
hola-mundo.asm  hola-mundo.o

user@linux:~/Documents/$ ld -o hola-ejecutable hola-mundo.o
user@linux:~/Documents/$ ls
hola-mundo.asm  hola-mundo.o  hola-ejecutable
```

4.4 Ejecutar el programa:

Finalmente, se tiene un programa ejecutable que se puede lanzar desde la terminal de Linux. Para esto, simplemente se utiliza:

```
./<archivo_ejecutable>
```

Siempre se debe anteponer el ./ para ejecutar un programa desde la terminal. En este caso particular:

```
user@linux:~/Documents/$ ls
hola-mundo.asm  hola-mundo.o  hola-ejecutable
user@linux:~/Documents/$ ./hola-ejecutable
Hola mundo! Segmentation fault (core dumped)
user@linux:~/Documents/$
```

Analisis: ¿Qué es un “Segmentation fault”?

Significa que se dio un error a nivel de sistema operativo que es percibido por el microprocesador como un intento de acceso a recursos (registros por ejemplo) de forma no permitida.

En este caso particular, nótese del código escrito que se cargaron los registros *rax*, *rdi*, *rsi* *rax*, *rdi*, *rsi* y *rdx* con diferentes valores, luego se llama al sistema y se termina el programa.

No es permitido que un programa deje los recursos del microprocesador previamente cargados y luego simplemente se salga de la ejecución, porque posteriormente, estos recursos van a ser utilizados por otros programas.

La forma correcta de indicar que nuestro programa terminó su ejecución adecuadamente y liberar los recursos asignados por el sistema operativo, es utilizando la llamada al sistema número 60 (sys_exit) que requiere como parámetro adicional un 0 en rdi, y nuevamente llamar al sistema.

De esta manera, el código completo para nuestro programa de “hola mundo” (incluyendo comentarios) sería el siguiente:

```
#####
;Ejemplo#1 - HolaMundo con Ensamblador de 64-bits (usando Linux)
;EL4313 - Laboratorio de Estructura de Microprocesadores
;2S2016-LCRA
#####

;-----Segmento de datos-----
;Aquí se declaran las constantes que se van a usar en el programa
section .data
    cons_hola: db 'Hola mundo!',0xa
    cons_tamano: equ $-cons_hola

;-----Segmento de código-----
;Contiene la secuencia de ejecución del programa
;La ejecución inicia en "_start", que es una etiqueta o referencia
section .text
    global _start
_start:
    mov rax,1
    mov rdi,1
    mov rsi,cons_hola
    mov rdx,cons_tamano
    syscall
;Luego de completar la primera operación, se deben recargar registros con las
;condiciones para la siguiente operación, en este caso: sys_exit (60)
    mov rax,60          ;se carga la llamada 60d (sys_exit) en rax
    mov rdi,0           ;en rdi se carga un 0
    syscall             ;se llama al sistema.

;fin del programa
```

Si se repiten las operaciones de ensamblado, ligado y ejecución, ahora los resultados son:

```
user@linux:~/Documents/$ ls
hola-mundo.asm
user@linux:~/Documents/$ nasm -f elf64 -o hola-mundo.o hola-mundo.asm
user@linux:~/Documents/$ ls
hola-mundo.asm  hola-mundo.o

user@linux:~/Documents/$ ld -o hola-ejecutable hola-mundo.o
user@linux:~/Documents/$ ls
hola-mundo.asm  hola-mundo.o  hola-ejecutable

user@linux:~/Documents/$ ./hola-ejecutable
Hola mundo!
user@linux:~/Documents/$
```

4.5 Depurar el programa usando GDB:

Un debugger, es una herramienta de software que permite ejecutar de forma controlada un determinado programa. En la mayoría de sistemas Linux se incluye el GDB como debugger, o bien, se pueden utilizar los pasos de la sección 3.2 para instalarlo. En la sección 3.3 se indicó la información general acerca de GDB como herramienta, y en esta sección se demuestra brevemente ¿Cómo se usa GDB para depurar el programa de “HolaMundo” que se trabajó en las secciones 4.1 hasta 4.4.

El código ASM de la sección 4.4 se va a modificar para imprimir 2 líneas de texto, esto con el fin de hacer más didáctico el ejercicio de depuración. También note que se definen 2 etiquetas o variables globales (*_segunda* y *_tercera*) que se van a utilizar en el ejemplo.

El código que debe utilizar es entonces el siguiente:

```
;#####
;Ejemplo#2 - HolaMundo para usar GBD (debugger)
;EL4313 - Laboratorio de Estructura de Microprocesadores
;#####

;-----Segmento de datos-----
section .data
    linea_uno: db 'Hola mundo! Primera linea',0xa
    l1_tamano: equ $-linea_uno

    linea_dos: db 'Hola mundo! Segunda linea',0xa
    l2_tamano: equ $-linea_dos

;-----Segmento de codigo-----
section .text
    global _start          ;Definicion de La etiqueta inicial
    global _segunda        ;Etiqueta para depurar el programa
    global _tercera        ; Etiqueta para depurar el programa

_start:
    ;Imprimir la primera linea
    mov rax,1              ;rax = sys_write (1)
    mov rdi,1              ;rdi = 1
    mov rsi,linea_uno      ;rsi = linea_uno
    mov rdx,l1_tamano      ;rdx = tamaño de linea_uno
    syscall                ;Llamar al sistema

    ;Imprimir la segunda línea
    mov rax,1              ;rax = sys_write (1)
    mov rdi,1              ;rdi = 1
    mov rsi,linea_dos      ;rsi = linea_dos
    mov rdx,l2_tamano      ;rdx = tamaño de linea_uno

_segunda:
    syscall                ;Llamar al sistema

    ;Liberar los recursos
    mov rax,60             ;rax=sys_exit (60)
    mov rdi,0              ;rdi=0

_tercera:
    syscall                ;Llamar al sistema

    ;fin del programa
```

Luego de repetir los pasos 4.2 y 4.3 para ensamblar y ligar el programa (incluso se puede ejecutar nuevamente el paso 4.4 para asegurarse que el programa se ejecuta adecuadamente)

```
user@linux:~ASM$ ls
hola2.asm hola-ejecutable hola-mundo.asm hola-mundo.o
user@linux:~ASM$ nasm -f elf64 -o hola2.o hola2.asm
user@linux:~ASM$ ld -o hola2-exe hola2.o
user@linux:~ASM$ ls
hola2.asm hola2-exe hola2.o hola-ejecutable hola-mundo.asm hola-mundo.o
user@linux:~ASM$ ./hola2-exe
Hola mundo! Primera linea
Hola mundo! Segunda linea
user@linux:~ASM$
```

Para depurar utilizando GDB, se deben seguir estos pasos:

1. Levantar GDB con el ejecutable de interés. En este ejemplo, se trata de **hola2-exe**. Si se ejecuta correctamente, la línea de comandos debe cambiar y pasa a mostrar la consola de comandos de GDB, que se muestra como (**gdb**):

```
user@linux:~ASM$ gdb hola2-exe
GNU gdb (Ubuntu 7.11-0ubuntu1) 7.11
Copyright (C) 2016 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-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 hola2-exe...(no debugging symbols found)...done.
(gdb)
```

2. Ejecutar el programa sin paradas: Para saber que el programa se ha cargado correctamente al ambiente de GDB, se puede usar el comando "run" desde la consola de GDB, con lo que se puede ver la ejecución del mismo.

Una vez concluida la ejecución, GDB muestra alguna información de interés:

- a. ¿Cuál es el identificador o número de proceso que el sistema operativo asignó para la ejecución?
- b. Si el programa se terminó en condiciones normales

```
(gdb) run
Starting program: /home/lrosales/Documents/NASM-Projects/01-HelloWorld/hola2-exe
Hola mundo! Primera linea
Hola mundo! Segunda linea
[Inferior 1 (process 1302) exited normally]
(gdb)
```

3. Agregar breakpoints o "puntos de chequeo" en el código. Los breakpoints se deben especificar explícitamente. En el caso de usar GDB con ensamblador, los breakpoints se asignan a las etiquetas que se definieron en el código (ya que no se está

trabajando con funciones o clases como se utiliza en programación de alto nivel)

Para agregar breakpoints desde el contexto de GDB se utiliza el comando **break <etiqueta>**

En este caso, y según las etiquetas que se mostraron en el código, tenemos los siguientes posibles breakpoints:

```
(gdb) break _start
Breakpoint 1 at 0x4000b0
(gdb) break _segunda
Breakpoint 2 at 0x4000e4
(gdb) break _tercera
Breakpoint 3 at 0x4000f0
(gdb)
```

Nótese que en cada nuevo breakpoint agregado, se indica la dirección en el stack o pila de código donde se está agregando el punto de parada.

4. Una vez indicados los breakpoints, se puede ejecutar nuevamente (**run**) el programa. En este caso, cuando se llegue al primer breakpoint se detiene la ejecución, y GDB muestra el programa detenido (inicialmente, en **_start**)

```
(gdb) run
Starting program: /home/lrosales/Documents/NASM-Projects/01-HelloWorld/hola2-exe

Breakpoint 1, 0x000000004000b0 in _start ()
(gdb)
```

Para permitir que el programa se siga ejecutando luego de un breakpoint, se usa el comando **continue**

5. Una vez que el programa está detenido por motivo de un breakpoint, es posible visualizar los valores que se están cargando en los registros, de manera que se pueda asegurar que son correctos de acuerdo al código del programa (ASM).
Una forma de visualizar todos los registros (en hexadecimal y en decimal) es usando el comando **info registers**:

```
(gdb) run
Starting program: /home/lrosales/Documents/NASM-Projects/01-HelloWorld/hola2-exe

Breakpoint 1, 0x000000004000b0 in _start ()
(gdb) info registers
rax                0x0          0
rbx                0x0          0
rcx                0x0          0
rdx                0x0          0
rsi                0x0          0
rdi                0x0          0
rbp                0x0          0x0
rsp                0x7fffffff2c0  0x7fffffff2c0
r8                 0x0          0
r9                 0x0          0
r10                0x0          0
r11                0x0          0
```



```

r12          0x0      0
r13          0x0      0
r14          0x0      0
r15          0x0      0
rip          0x4000b0 0x4000b0 <_start>
eflags       0x202    [ IF ]
cs           0x33     51
ss           0x2b     43
ds           0x0      0
es           0x0      0
fs           0x0      0
gs           0x0      0
(gdb)

```

Si se desea visualizar solamente un registro en específico, se usa el **comando print \$<nombre del registro>**

El comando **print**, por defecto, muestra el valor del registro en formato decimal. Pero se puede cambiar la especificación para mostrar en formato:

- **print/x \$<registro>** para visualizar en formato hexadecimal
- **print/d \$<registro>** para visualizar en formato decimal
- **print/t \$<registro>** para visualizar en formato binario
- **print/c \$<registro>** para visualizar en formato de caracter

Por ejemplo:

```

(gdb) run
Starting program: /home/lrosales/Documents/NASM-Projects/01-HelloWorld/hola2-exe

Breakpoint 1, 0x0000000004000b0 in _start ()
(gdb) print $rax
$2 = 0
(gdb) print/x $rax
$3 = 0x0
(gdb) print/t $rax
$4 = 0
(gdb) print/c $rax
$5 = 0 '\000'
(gdb)

```

Observe (del resultado de **info registers**) que en este caso, todos los registros tienen valores de cero, lo cual tiene sentido porque al detener la ejecución en la etiqueta **_start** no se ha cargado ningún valor a registros y aún no se ejecuta ninguna instrucción.

Si se repite el ejercicio para visualizar todos los registros en el siguiente breakpoint (**_segunda**), el resultado es el siguiente:

```

(gdb) run
Starting program: /home/lrosales/Documents/NASM-Projects/01-HelloWorld/hola2-exe
Breakpoint 1, 0x0000000004000b0 in _start ()
(gdb) continue
Continuing.

```

```

Hola mundo! Primera linea
Breakpoint 2, 0x0000000004000e4 in _segunda ()
(gdb) info registers
rax          0x1      1
rbx          0x0      0
rcx          0x4000cb  4194507
rdx          0x1a     26
rsi          0x60010e  6291726
rdi          0x1      1
rbp          0x0      0x0
rsp          0x7fffffff2c0  0x7fffffff2c0
r8           0x0      0
r9           0x0      0
r10          0x0      0
r11          0x202     514
r12          0x0      0
r13          0x0      0
r14          0x0      0
r15          0x0      0
rip          0x4000e4  0x4000e4 <_segunda>
eflags      0x202     [ IF ]
cs           0x33     51
ss           0x2b     43
ds           0x0      0
es           0x0      0
fs           0x0      0
gs           0x0      0
(gdb)

```

Observe que los valores de los registros `rax`, `rdx`, `rsi` y `rdi` cambiaron, y ahora muestran los valores que se cargaron según el código de ensamblador.

- `rax` tiene un 1, que es para llamar un `sys_write`
- `rdi` tiene un 1, que es el segundo parámetro para hacer el `sys_write`
- `rsi` tiene el valor que corresponde a la dirección de memoria en la que se almacena la cadena de texto que se desea imprimir
- `rdx` muestra como la longitud de la cadena (26 caracteres) según esta tabla:

Base (rsi)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0x60010e	H	o	l	a		M	u	n	d	o	i		P	r	i	m	e	r	a		L	i	n	e	a	0xa

- Finalmente, se pueden imprimir contenidos en una determinada dirección de memoria, utilizando el comando `x/ <dirección>`.
 Esto es particularmente útil para asegurarse que los parámetros que se pasan a los registros son correctos.
 El comando `x/` conserva las condiciones de formato que usa el comando `print`. Es decir, se puede complementar con un `/c` para imprimir el contenido de cada celda de memoria en formato de carácter ASCII.
 Por ejemplo, en este caso particular vamos a retomar la tabla que indica los caracteres esperados en cada una de las direcciones de memoria para poder imprimirlos.
 Recuerde que en este caso, el registro `rsi` debe tener la dirección base (donde empieza la cadena de texto) y luego podemos ir incrementándola uno a uno para asegurarse que la cadena es correcta y está completa.
 Por ejemplo:

```

(gdb) run
Starting program: /home/lrosales/Documents/NASM-Projects/01-HelloWorld/hola2-exe
Hola mundo! Primera linea

Breakpoint 1, 0x000000004000e4 in _segunda ()
(gdb) print/x $rsi
$1 = 0x60010e
(gdb) print/d $rdx
$3 = 26

```

A partir del valor de `$rsi` se obtiene la dirección de inicio (0x60010e) y a partir del valor de `$rdx` se conoce la cantidad (decimal) de caracteres que se debe imprimir (26).

Entonces, se usa el comando `x/c 0x60010e` para imprimir el carácter ASCII almacenado en esa dirección de memoria, que en este caso, resulta ser una `'H'`, como era de esperarse en la cadena `'Hola mundo! Segunda linea'`:

```

(gdb) x/c 0x60010e
0x60010e:      72 'H'

```

Una ventaja que tiene el uso del comando `x/` en GDB, es que con solamente presionar enter, se repite el comando anterior, pero aumentando secuencialmente (uno a uno) los bytes en la dirección. De esta forma es más simple verificar cada una de las direcciones de memoria paso por paso:

```

(gdb) x/c 0x60010e
0x60010e:      72 'H'
(gdb)
0x60010f:     111 'o'
(gdb)
0x600110:     108 'l'
(gdb)
0x600111:      97 'a'
(gdb)
0x600112:      32 ' '
(gdb)
0x600113:     109 'm'
(gdb)
0x600114:     117 'u'
(gdb)
0x600115:     110 'n'
(gdb)
0x600116:     100 'd'
(gdb)
0x600117:     111 'o'
(gdb)
0x600118:      33 '!'
(gdb)
0x600119:      32 ' '
(gdb)
0x60011a:      83 'S'
(gdb)

```

```

0x60011b:     101 'e'
(gdb)
0x60011c:     103 'g'
(gdb)
0x60011d:     117 'u'
(gdb)
0x60011e:     110 'n'
(gdb)
0x60011f:     100 'd'
(gdb)
0x600120:      97 'a'
(gdb)
0x600121:      32 ' '
(gdb)
0x600122:     108 'l'
(gdb)
0x600123:     105 'i'
(gdb)
0x600124:     110 'n'
(gdb)
0x600125:     101 'e'
(gdb)
0x600126:      97 'a'
(gdb)
0x600127:      10 '\n'
(gdb)
0x600128:       0 '\000'

```

5. Manejo del teclado como periférico de entrada:

El teclado, es uno de los dispositivos periféricos básicos en un sistema computacional estándar, cumple la función de ser un periférico de entrada (que recibe información de parte del usuario) y que complementa a la pantalla o dispositivo periférico de salida (que muestra información al usuario).

De la misma forma que se trabajó con la llamada a sistema “sys_write” para mostrar contenidos en la pantalla, la captura de información desde el usuario se hace con una llamada al sistema, que según se puede consultar de la tabla de llamadas se trata de “sys_read” que tiene un valor de 0 para sistemas operativos Linux de 64 bits.

El uso de la llamada de sistema entonces, sería el siguiente:

```
section .data
    db variable ;direccion de memoria donde se almacena la tecla capturada

section .text
    global _start

_start:
mov rax,0 ;rax = "sys_read"
mov rdi,0 ;rdi = 0 (standard input = teclado)
mov rsi,variable ;rsi = direccion de memoria donde almacenar la tecla capturada
mov rdx,N ;rdx= "N" cuantos bytes (tecleados) se deben capturar
syscall ;Llamar al sistema
```

El archivo `03_manejo_teclado.asm` contiene código suficiente para ejecutar una aplicación de prueba, la cual lee un solo evento del teclado (una tecla) que luego se imprime en la misma consola. Repita los pasos de ensamblado y ligado del código ensamblador para producir un ejecutable, con resultados como los siguientes:

```
user@linux:~/ejecutables$ ./03_manejo_teclado
Presione una tecla, y luego Enter: x
Usted presiono la tecla: x
Fin del programa.
user@linux:~/ejecutables$
```

6. Control de flujo de ejecución del programa:

¿Qué es el flujo de ejecución de un programa?

Los programas (ya sean escritos en ensamblador o en otro lenguaje), cuando son ejecutados por el sistema operativo, son cargados en una sección de la memoria RAM del sistema computacional.

Cada programa en ejecución se puede entonces visualizar como una secuencia de instrucciones que debe ejecutarse de forma ordenada: una por una. Cada una de las entradas en memoria corresponde a una instrucción, y existe un registro especial dentro del microprocesador que se encarga de llevar el ritmo de la ejecución. Este registro se llama el contador de programa o **program counter** y permite apuntar a la instrucción actual y determinar ¿cuál es la siguiente?

Por ejemplo, recuerde el primer programa escrito en ensamblador (`01_hola_mundo.asm`), una vez ensamblado y linkado esta listo para ser cargado en memoria. Una manera de visualizar la pila de memoria del programa desde Linux es usando el comando ***objdump***, con el cual se “des-ensambla” el programa y se muestra la ubicación relativa en memoria. Se dice relativa, porque la ubicación absoluta del programa en memoria va a ser definida por el sistema operativo en el momento que se ejecute el programa. La salida de ***objdump*** muestra un punto de partida o un *offset* con respecto a esa dirección inicial o de inserción.

Para utilizar ***objdump*** se debe indicar:

- El tipo de des-ensamble a realizar. Se recomienda usar “-d” para desensamblar el segmento de código, que es donde se almacenan las instrucciones. Sin embargo, se puede usar la opción “-D” para desensamblar todo el programa.
- El tipo de arquitectura del microprocesador. Se recomienda usar “-m i386:x86-64”

Información del programa	<pre> user@linux:~/ ejecutables\$ objdump -D -m i386:x86-64 01_hola_mundo 01_hola_mundo: file format elf64-x86-64 </pre>	
Segmento de código	<pre> Disassembly of section .text: </pre>	
Inicio de ejecución	<pre> 00000000004000b0 <start>: 4000b0: b8 01 00 00 00 4000b5: bf 01 00 00 00 4000ba: 48 be d8 00 60 00 00 4000c1: 00 00 00 4000c4: ba 0c 00 00 00 4000c9: 0f 05 4000cb: b8 3c 00 00 00 4000d0: bf 00 00 00 00 4000d5: 0f 05 </pre>	<pre> mov \$0x1,%eax mov \$0x1,%edi movabs \$0x6000d8,%rsi mov \$0xc,%edx syscall mov \$0x3c,%eax mov \$0x0,%edi syscall </pre>
Segmento de datos	<pre> Disassembly of section .data: 00000000006000d8 <cons_hola>: 6000d8: 48 6f 6000da: 6c 6000db: 61 6000dc: 20 6d 75 6000df: 6e 6000e0: 64 6f 6000e2: 21 0a </pre>	<pre> rex.W outsl %ds:(%rsi),(%dx) insb (%dx),%es:(%rdi) (bad) and %ch,0x75(%rbp) outsb %ds:(%rsi),(%dx) outsl %fs:(%rsi),(%dx) and %ecx,(%rdx) </pre>

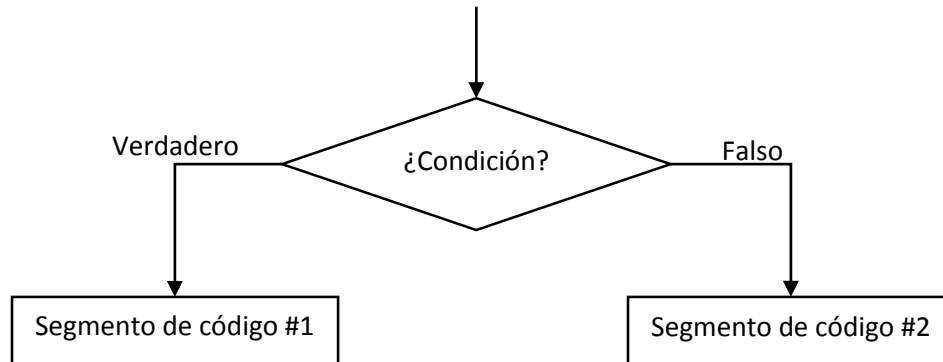
Contenidos de memoria (Hex) Contenidos de memoria (texto)

Observe que las instrucciones y sus operandos cargados (relativamente) en memoria se pueden visualizar en hexadecimal o también en formato de texto, usando caracteres ASCII (en la medida de lo posible, existen limitaciones y se observa que el dump muestra algunos de los nombres de los registros usando la convención de 32 bits como `%eax` en lugar de `rax`)

Control de flujo:

Los programas no siempre son una sola secuencia estricta de instrucciones, en realidad, son sub-secuencias de instrucciones que se ejecutan en un orden variable, dependiendo del comportamiento del programa en el tiempo (que esta influenciado por el comportamiento del usuario, periféricos de entrada, temporizaciones, etc).

En lenguajes de programación de alto nivel, estas variaciones del flujo de ejecución se regulan con mecanismos de control como las estructuras “*if*” o “*si condicionados*” que típicamente permiten variar la ejecución entre uno u otro bloque de código según una condición particular.



En el caso del lenguaje ensamblador, existen diferentes mecanismos para controlar el flujo, todos basados en 3 elementos fundamentales: etiquetas, comparación (entre registros) y saltos.

Etiquetas:

A nivel de código, las etiquetas son referencias que apuntan a la ubicación en memoria de una instrucción en particular. El contador de programa (PC) puede apuntarse a una etiqueta en cualquier momento durante la ejecución del programa.

Las etiquetas en lenguaje ensamblador se escriben en el segmento de código (**section .text**) y se escriben con las siguientes reglas:

- Inician con un punto .
- Terminan con dos puntos :
- Por ejemplo, la etiqueta “.primer_bloque:” en el siguiente fragmento de código:

```
_start:
;Primer bloque de codigo
.primer_bloque:
    mov rax,1           ;rax = "sys_write"
    mov rdi,1           ;rdi = 1 (standard output = pantalla)
    mov rsi,msg_1       ;rsi = mensaje a imprimir
    mov rdx,tamano_msj_1 ;rdx=tamano del mensaje
    syscall
```

Comparaciones:

El lenguaje ensamblador ofrece la instrucción **cmp** para realizar la comparación entre registros, direcciones de memoria o constantes (además de las combinaciones entre estos).

La instrucción **cmp** sin embargo no produce un resultado directo, sino que modifica el registro del microprocesador que contiene las banderas o indicadores. En función a esas banderas es que luego el programa puede modificar el flujo de ejecución chequeando las banderas y saltando a una determinada etiqueta.

Salto:

Las operaciones de salto o “**jump**” son las que complementan la operación de la comparación **cmp**, de manera que al chequear el resultado de las banderas (registro de estado del procesador) se puede determinar el resultado de la comparación y saltar a una etiqueta definida en el código.

Las posibles instrucciones de salto disponibles en el lenguaje ensamblador x86_64 se resumen en la siguiente tabla.

Instrucción	Tipo de salto
JE	Jump Equal Salta a la etiqueta dada cuando los 2 operadores en cmp son iguales
JZ	Jump Zero Salta a la etiqueta dada cuando la bandera de zero es acertada. Se puede usar como el JE o para detectar cuando un registro en particular alcanza valor de cero.
JNE	Jump Not Equal Salta a la etiqueta dada cuando los 2 operadores en cmp son diferentes
JNZ	Jump Not Zero Salta a la etiqueta dada cuando la bandera de zero no es acertada. Se puede usar como el JNE o para detectar cuando un registro en particular mantiene un valor diferente a cero.
JG	Jump Greater Salta a la etiqueta dada cuando el primer operando del cmp es estrictamente mayor que el segundo operando
JGE	Jump Greater or Equal Salta a la etiqueta dada cuando el primer operando del cmp es mayor o igual que el segundo operando
JA	Jump Above Al igual que JG , salta a la etiqueta dada cuando el primer operando es estrictamente mayor que el segundo, pero realiza la comparación ignorando el signo del operando
JAE	Jump Above or Equal Al igual que JGE , salta a la etiqueta dada cuando el primer operando es mayor o igual que el segundo, pero realiza la comparación ignorando el signo del operando

En el siguiente ejemplo se muestra la ejecución “por bloques” del segmento de código de un programa a partir de comparaciones. El código completo se encuentra en el archivo **04_control_flujo.asm**. Observe las etiquetas indicadas en el código y compare con el orden de ejecución de bloques en el diagrama adjunto.

```

section .data
    num1: equ 100
    num2: equ 50

section .text
    global _start

_start:

.primero_bloque:
    mov rax,1
    mov rdi,1
    mov rsi,msg_1
    mov rdx,tamano_msj_1
    syscall

    mov rax,num1
    mov rbx,num2
    cmp rax,rbx
    je .segundo_bloque
    jne .tercer_bloque

.segundo_bloque :
    mov rax,1
    mov rdi,1
    mov rsi,msg_2
    mov rdx,tamano_msj_2
    syscall

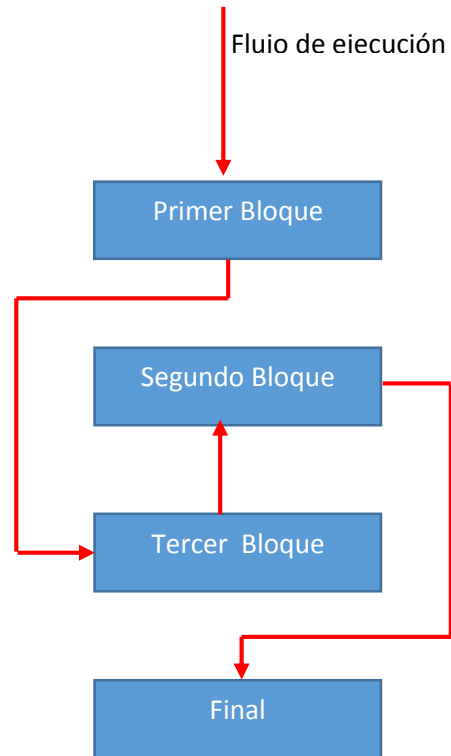
    mov rax,num1
    mov rbx,num2
    cmp rax,rbx
    jg .final

.tercer_bloque:
    mov rdi,1
    mov rsi,msg_3
    mov rdx,tamano_msj_3
    syscall

    mov rax,num1
    mov rbx,num1
    cmp rax,rbx
    je .segundo_bloque
    jne .tercer_bloque

.final:
    mov rax,60
    mov rdi,0
    syscall

```



Programa en ejecución

```
user@linux:~/$ ./04_control_flujo
```

```

Este es el primer bloque de código.
Este es el tercer bloque de código.
Este es el segundo bloque de código.

```

```
user@linux:~/$
```


Anexo: Tabla de llamadas de sistema para Linux x86_64

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
4	sys_stat	const char *filename	struct stat *statbuf				
5	sys_fstat	unsigned int fd	struct stat *statbuf				
6	sys_lstat	const char *filename	struct stat *statbuf				
7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs			
8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin			
9	sys_mmap	unsigned long addr	unsigned long len	unsigned long prot	unsigned long flags	unsigned long fd	unsigned long off
10	sys_mprotect	unsigned long start	size_t len	unsigned long prot			
11	sys_munmap	unsigned long addr	size_t len				
12	sys_brk	unsigned long brk					
13	sys_rt_sigaction	int sig	const struct sigaction *act	struct sigaction *oact	size_t sigsetsize		
14	sys_rt_sigprocmask	int how	sigset_t *nset	sigset_t *oset	size_t sigsetsize		
15	sys_rt_sigreturn	unsigned long __unused					
16	sys_ioctl	unsigned int fd	unsigned int cmd	unsigned long arg			
17	sys_pread64	unsigned long fd	char *buf	size_t count	loff_t pos		
18	sys_pwrite64	unsigned int fd	const char *buf	size_t count	loff_t pos		
19	sys_readv	unsigned long fd	const struct iovec *vec	unsigned long vlen			
20	sys_writev	unsigned long fd	const struct iovec *vec	unsigned long vlen			
21	sys_access	const char *filename	int mode				
22	sys_pipe	int *filedes					
23	sys_select	int n	fd_set *inp	fd_set *outp	fd_set *exp	struct timeval *tvp	
24	sys_sched_yield						
25	sys_mremap	unsigned long addr	unsigned long old_len	unsigned long new_len	unsigned long flags	unsigned long new_addr	
26	sys_msync	unsigned long start	size_t len	int flags			
27	sys_mincore	unsigned long start	size_t len	unsigned char *vec			
28	sys_madvise	unsigned long start	size_t len_in	int behavior			
29	sys_shmget	key_t key	size_t size	int shmflg			
30	sys_shmat	int shmid	char *shmaddr	int shmflg			
31	sys_shmctl	int shmid	int cmd	struct shmids *buf			
32	sys_dup	unsigned int fildes					

33	sys_dup2	unsigned int oldfd	unsigned int newfd				
34	sys_pause						
35	sys_nanosleep	struct timespec *rqtp	struct timespec *rmtp				
36	sys_getitimer	int which	struct itimerval *value				
37	sys_alarm	unsigned int seconds					
38	sys_setitimer	int which	struct itimerval *value	struct itimerval *ovalue			
39	sys_getpid						
40	sys_sendfile	int out_fd	int in_fd	off_t *offset	size_t count		
41	sys_socket	int family	int type	int protocol			
42	sys_connect	int fd	struct sockaddr *useraddr	int addrlen			
43	sys_accept	int fd	struct sockaddr *upeer_sockaddr	int *upeer_addrlen			
44	sys_sendto	int fd	void *buff	size_t len	unsigned flags	struct sockaddr *addr	int addr_len
45	sys_recvfrom	int fd	void *ubuf	size_t size	unsigned flags	struct sockaddr *addr	int *addr_len
46	sys_sendmsg	int fd	struct msghdr *msg	unsigned flags			
47	sys_recvmsg	int fd	struct msghdr *msg	unsigned int flags			
48	sys_shutdown	int fd	int how				
49	sys_bind	int fd	struct sockaddr *umyaddr	int addrlen			
50	sys_listen	int fd	int backlog				
51	sys_getsockname	int fd	struct sockaddr *usockaddr	int *usockaddr_len			
52	sys_getpeername	int fd	struct sockaddr *usockaddr	int *usockaddr_len			
53	sys_socketpair	int family	int type	int protocol	int *usockvec		
54	sys_setsockopt	int fd	int level	int optname	char *optval	int optlen	
55	sys_getsockopt	int fd	int level	int optname	char *optval	int *optlen	
56	sys_clone	unsigned long clone_flags	unsigned long newsp	void *parent_tid	void *child_tid		
57	sys_fork						
58	sys_vfork						
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]			
60	sys_exit	int error_code					
61	sys_wait4	pid_t upid	int *stat_addr	int options	struct rusage *ru		
62	sys_kill	pid_t pid	int sig				
63	sys_uname	struct old_utsname *name					
64	sys_semget	key_t key	int nsems	int semflg			
65	sys_semoop	int semid	struct sembuf *tsops	unsigned nsops			
66	sys_semctl	int semid	int semnum	int cmd	union semun arg		
67	sys_shmctl	char *shmaddr					
68	sys_msgsget	key_t key	int msgflg				

69	sys_msgsnd	int msqid	struct msgbuf *msgp	size_t msgsz	int msgflg		
70	sys_msgrcv	int msqid	struct msgbuf *msgp	size_t msgsz	long msgtyp	int msgflg	
71	sys_msgctl	int msqid	int cmd	struct msqid_ds *buf			
72	sys_fcntl	unsigned int fd	unsigned int cmd	unsigned long arg			
73	sys_flock	unsigned int fd	unsigned int cmd				
74	sys_fsync	unsigned int fd					
75	sys_fdatasync	unsigned int fd					
76	sys_truncate	const char *path	long length				
77	sys_ftruncate	unsigned int fd	unsigned long length				
78	sys_getdents	unsigned int fd	struct linux_dirent *dirent	unsigned int count			
79	sys_getcwd	char *buf	unsigned long size				
80	sys_chdir	const char *filename					
81	sys_fchdir	unsigned int fd					
82	sys_rename	const char *oldname	const char *newname				
83	sys_mkdir	const char *pathname	int mode				
84	sys_rmdir	const char *pathname					
85	sys_creat	const char *pathname	int mode				
86	sys_link	const char *oldname	const char *newname				
87	sys_unlink	const char *pathname					
88	sys_symlink	const char *oldname	const char *newname				
89	sys_readlink	const char *path	char *buf	int bufsiz			
90	sys_chmod	const char *filename	mode_t mode				
91	sys_fchmod	unsigned int fd	mode_t mode				
92	sys_chown	const char *filename	uid_t user	gid_t group			
93	sys_fchown	unsigned int fd	uid_t user	gid_t group			
94	sys_lchown	const char *filename	uid_t user	gid_t group			
95	sys_umask	int mask					
96	sys_gettimeofday	struct timeval *tv	struct timezone *tz				
97	sys_getrlimit	unsigned int resource	struct rlimit *rlim				
98	sys_getrusage	int who	struct rusage *ru				
99	sys_sysinfo	struct sysinfo *info					
100	sys_times	struct sysinfo *info					
101	sys_ptrace	long request	long pid	unsigned long addr	unsigned long data		
102	sys_getuid						

103	sys_syslog	int type	char *buf	int len			
104	sys_getgid						
105	sys_setuid	uid_t uid					
106	sys_setgid	gid_t gid					
107	sys_geteuid						
108	sys_getegid						
109	sys_setpgid	pid_t pid	pid_t pgid				
110	sys_getppid						
111	sys_getpgrp						
112	sys_setsid						
113	sys_setreuid	uid_t ruid	uid_t euid				
114	sys_setregid	gid_t rgid	gid_t egid				
115	sys_getgroups	int gidsetsize	gid_t *grouplist				
116	sys_setgroups	int gidsetsize	gid_t *grouplist				
117	sys_setresuid	uid_t *ruid	uid_t *euid	uid_t *suid			
118	sys_getresuid	uid_t *ruid	uid_t *euid	uid_t *suid			
119	sys_setresgid	gid_t rgid	gid_t egid	gid_t sgid			
120	sys_getresgid	gid_t *rgid	gid_t *egid	gid_t *sgid			
121	sys_getpgid	pid_t pid					
122	sys_setfsuid	uid_t uid					
123	sys_setfsgid	gid_t gid					
124	sys_getsid	pid_t pid					
125	sys_capget	cap_user_header_t header	cap_user_data_t dataptr				
126	sys_capset	cap_user_header_t header	const cap_user_data_t data				
127	sys_rt_sigpending	sigset_t *set	size_t sigsetsize				
128	sys_rt_sigtimedwait	const sigset_t *uthese	siginfo_t *uinfo	const struct timespec *uts	size_t sigsetsize		
129	sys_rt_sigqueueinfo	pid_t pid	int sig	siginfo_t *uinfo			
130	sys_rt_sigsuspend	sigset_t *unewset	size_t sigsetsize				
131	sys_sigaltstack	const stack_t *uss	stack_t *uoss				
132	sys_utime	char *filename	struct utimbuf *times				
133	sys_mknod	const char *filename	int mode	unsigned dev			
134	sys_uselib	NOT IMPLEMENTED					
135	sys_personality	unsigned int personality					
136	sys_ustat	unsigned dev	struct ustat *ubuf				

137	sys_statfs	const char *pathname	struct statfs *buf				
138	sys_fstatfs	unsigned int fd	struct statfs *buf				
139	sys_sysfs	int option	unsigned long arg1	unsigned long arg2			
140	sys_getpriority	int which	int who				
141	sys_setpriority	int which	int who	int niceval			
142	sys_sched_setparam	pid_t pid	struct sched_param *param				
143	sys_sched_getparam	pid_t pid	struct sched_param *param				
144	sys_sched_setscheduler	pid_t pid	int policy	struct sched_param *param			
145	sys_sched_getscheduler	pid_t pid					
146	sys_sched_get_priority_max	int policy					
147	sys_sched_get_priority_min	int policy					
148	sys_sched_rr_get_interval	pid_t pid	struct timespec *interval				
149	sys_mlock	unsigned long start	size_t len				
150	sys_munlock	unsigned long start	size_t len				
151	sys_mlockall	int flags					
152	sys_munlockall						
153	sys_vhangup						
154	sys_modify_ldt	int func	void *ptr	unsigned long bytecount			
155	sys_pivot_root	const char *new_root	const char *put_old				
156	sys__sysctl	struct __sysctl_args *args					
157	sys_prctl	int option	unsigned long arg2	unsigned long arg3	unsigned long arg4		unsigned long arg5
158	sys_arch_prctl	struct task_struct *task	int code	unsigned long *addr			
159	sys_adjtimex	struct timex *txc_p					
160	sys_setrlimit	unsigned int resource	struct rlimit *rlim				
161	sys_chroot	const char *filename					
162	sys_sync						
163	sys_acct	const char *name					
164	sys_settimeofday	struct timeval *tv	struct timezone *tz				

165	sys_moun t	char *dev_name	char *dir_name	char *type	unsigned long flags	void *data	
166	sys_umou nt2	const char *target	int flags				
167	sys_swapo n	const char *specialfile	int swap_flags				
168	sys_swapo ff	const char *specialfile					
169	sys_reboo t	int magic1	int magic2	unsigned int cmd	void *arg		
170	sys_setho stname	char *name	int len				
171	sys_setdo mainname	char *name	int len				
172	sys_iopl	unsigned int level	struct pt_regs *regs				
173	sys_ioper m	unsigned long from	unsigned long num	int turn_on			
174	sys_create _module	REMOVED IN Linux 2.6					
175	sys_init_m odule	void *umod	unsigned long len	const char *uargs			
176	sys_delete _module	const chat *name_user	unsigned int flags				
177	sys_get_k ernel_sym s	REMOVED IN Linux 2.6					
178	sys_query _module	REMOVED IN Linux 2.6					
179	sys_quota ctl	unsigned int cmd	const char *special	qid_t id	void *addr		
180	sys_nfsser vctl	NOT IMPLEMENTED					
181	sys_getpm sg	NOT IMPLEMENTED					
182	sys_putp msg	NOT IMPLEMENTED					
183	sys_afs_sy scall	NOT IMPLEMENTED					
184	sys_tuxcal l	NOT IMPLEMENTED					
185	sys_securi ty	NOT IMPLEMENTED					
186	sys_gettid						
187	sys_reada head	int fd	loff_t offset	size_t count			
188	sys_setxat tr	const char *pathname	const char *name	const void *value	size_t size	int flags	
189	sys_lsetxa ttr	const char *pathname	const char *name	const void *value	size_t size	int flags	
190	sys_fsetxa ttr	int fd	const char *name	const void *value	size_t size	int flags	
191	sys_getxat tr	const char *pathname	const char *name	void *value	size_t size		
192	sys_lgetxa ttr	const char *pathname	const char *name	void *value	size_t size		
193	sys_fgetxa ttr	int fd	const har *name	void *value	size_t size		
194	sys_listxat tr	const char *pathname	char *list	size_t size			
195	sys_llistxa ttr	const char *pathname	char *list	size_t size			
196	sys_flistxa ttr	int fd	char *list	size_t size			

197	sys_removexattr	const char *pathname	const char *name				
198	sys_lremovexattr	const char *pathname	const char *name				
199	sys_fremovexattr	int fd	const char *name				
200	sys_tkill	pid_t pid	ing sig				
201	sys_time	time_t *tloc					
202	sys_futex	u32 *uaddr	int op	u32 val	struct timespec *utime	u32 *uaddr2	u32 val3
203	sys_sched_setaffinity	pid_t pid	unsigned int len	unsigned long *user_mask_ptr			
204	sys_sched_getaffinity	pid_t pid	unsigned int len	unsigned long *user_mask_ptr			
205	sys_set_thread_area	NOT IMPLEMENTED. Use arch_prctl					
206	sys_io_setup	unsigned nr_events	aio_context_t *ctxp				
207	sys_io_destroy	aio_context_t ctx					
208	sys_io_getevents	aio_context_t ctx_id	long min_nr	long nr	struct io_event *events		
209	sys_io_submit	aio_context_t ctx_id	long nr	struct iocb **iocbpp			
210	sys_io_cancel	aio_context_t ctx_id	struct iocb *iocb	struct io_event *result			
211	sys_get_thread_area	NOT IMPLEMENTED. Use arch_prctl					
212	sys_lookup_dcookie	u64 cookie64	long buf	long len			
213	sys_epoll_create	int size					
214	sys_epoll_ctl_old	NOT IMPLEMENTED					
215	sys_epoll_wait_old	NOT IMPLEMENTED					
216	sys_remap_file_pages	unsigned long start	unsigned long size	unsigned long prot	unsigned long pgoff	unsigned long flags	
217	sys_getdents64	unsigned int fd	struct linux_dirent64 *dirent	unsigned int count			
218	sys_set_tid_address	int *tidptr					
219	sys_restart_syscall						
220	sys_semtime	int semid	struct sembuf *tsops	unsigned nsops	const struct timespec *timeout		
221	sys_fadvise64	int fd	loff_t offset	size_t len	int advice		
222	sys_timer_create	const clockid_t which_clock	struct sigevent *timer_event_spec	timer_t *created_timer_id			
223	sys_timer_settime	timer_t timer_id	int flags	const struct itimerspec *new_setting	struct itimerspec *old_setting		
224	sys_timer_gettime	timer_t timer_id	struct itimerspec *setting				

225	sys_timer_getoverrun	timer_t timer_id					
226	sys_timer_delete	timer_t timer_id					
227	sys_clock_settime	const clockid_t which_clock	const struct timespec *tp				
228	sys_clock_gettime	const clockid_t which_clock	struct timespec *tp				
229	sys_clock_getres	const clockid_t which_clock	struct timespec *tp				
230	sys_clock_nanosleep	const clockid_t which_clock	int flags	const struct timespec *rqtp	struct timespec *rmtp		
231	sys_exit_group	int error_code					
232	sys_epoll_wait	int epfd	struct epoll_event *events	int maxevents	int timeout		
233	sys_epoll_ctl	int epfd	int op	int fd	struct epoll_event *event		
234	sys_tgkill	pid_t tgid	pid_t pid	int sig			
235	sys_utimes	char *filename	struct timeval *utimes				
236	sys_vserver	NOT IMPLEMENTED					
237	sys_mbind	unsigned long start	unsigned long len	unsigned long mode	unsigned long *nmask	unsigned long maxnode	unsigned flags
238	sys_set_mempolicy	int mode	unsigned long *nmask	unsigned long maxnode			
239	sys_get_mempolicy	int *policy	unsigned long *nmask	unsigned long maxnode	unsigned long addr	unsigned long flags	
240	sys_mq_open	const char *u_name	int oflag	mode_t mode	struct mq_attr *u_attr		
241	sys_mq_unlink	const char *u_name					
242	sys_mq_timedsend	mqd_t mqdes	const char *u_msg_ptr	size_t msg_len	unsigned int msg_prio	const struct timespec *u_abs_timeout	
243	sys_mq_timedreceive	mqd_t mqdes	char *u_msg_ptr	size_t msg_len	unsigned int *u_msg_prio	const struct timespec *u_abs_timeout	
244	sys_mq_notify	mqd_t mqdes	const struct sigevent *u_notification				
245	sys_mq_getattr	mqd_t mqdes	const struct mq_attr *u_mqstat	struct mq_attr *u_omqstat			
246	sys_kexec_load	unsigned long entry	unsigned long nr_segments	struct kexec_segment *segments	unsigned long flags		
247	sys_waitid	int which	pid_t upid	struct siginfo *infop	int options	struct rusage *ru	
248	sys_add_key	const char * _type	const char * _description	const void * _payload	size_t plen		
249	sys_request_key	const char * _type	const char * _description	const char * _callout_info	key_serial_t destringid		
250	sys_keyctl	int option	unsigned long arg2	unsigned long arg3	unsigned long arg4	unsigned long arg5	
251	sys_ioprio_set	int which	int who	int ioprio			
252	sys_ioprio_get	int which	int who				
253	sys_inotify_init						

254	sys_inotify_add_watch	int fd	const char *pathname	u32 mask			
255	sys_inotify_rm_watch	int fd	__s32 wd				
256	sys_migrate_pages	pid_t pid	unsigned long maxnode	const unsigned long *old_nodes	const unsigned long *new_nodes		
257	sys_opentat	int dfd	const char *filename	int flags	int mode		
258	sys_mkdirat	int dfd	const char *pathname	int mode			
259	sys_mknodat	int dfd	const char *filename	int mode	unsigned dev		
260	sys_fchmodat	int dfd	const char *filename	uid_t user	gid_t group	int flag	
261	sys_futimesat	int dfd	const char *filename	struct timeval *utimes			
262	sys_newfsstatat	int dfd	const char *filename	struct stat *statbuf	int flag		
263	sys_unlinkat	int dfd	const char *pathname	int flag			
264	sys_renameat	int oldfd	const char *oldname	int newfd	const char *newname		
265	sys_linkat	int oldfd	const char *oldname	int newfd	const char *newname	int flags	
266	sys_symlinkat	const char *oldname	int newfd	const char *newname			
267	sys_readlinkat	int dfd	const char *pathname	char *buf	int bufsiz		
268	sys_fchmodat	int dfd	const char *filename	mode_t mode			
269	sys_faccessat	int dfd	const char *filename	int mode			
270	sys_pselect6	int n	fd_set *inp	fd_set *outp	fd_set *exp	struct timespec *tsp	void *sig
271	sys_ppoll	struct pollfd *ufds	unsigned int nfds	struct timespec *tsp	const sigset_t *sigmask	size_t sigsetsize	
272	sys_unshare	unsigned long unshare_flags					
273	sys_set_robust_list	struct robust_list_head *head	size_t len				
274	sys_get_robust_list	int pid	struct robust_list_head **head_ptr	size_t *len_ptr			
275	sys_splice	int fd_in	loff_t *off_in	int fd_out	loff_t *off_out	size_t len	unsigned int flags
276	sys_tee	int fdin	int fdout	size_t len	unsigned int flags		
277	sys_sync_file_range	long fd	loff_t offset	loff_t bytes	long flags		
278	sys_vmsplice	int fd	const struct iovec *iov	unsigned long nr_segs	unsigned int flags		
279	sys_move_pages	pid_t pid	unsigned long nr_pages	const void **pages	const int *nodes	int *status	int flags
280	sys_utimensat	int dfd	const char *filename	struct timespec *utimes	int flags		
281	sys_epoll_pwait	int epfd	struct epoll_event *events	int maxevents	int timeout	const sigset_t *sigmask	size_t sigsetsize
282	sys_signalfd	int ufd	sigset_t *user_mask	size_t sizemask			
283	sys_timerfd_create	int clockid	int flags				

284	sys_eventfd	unsigned int count					
285	sys_fallocate	long fd	long mode	loff_t offset	loff_t len		
286	sys_timerfd_settime	int ufd	int flags	const struct itimerspec *utmr	struct itimerspec *otmr		
287	sys_timerfd_gettime	int ufd	struct itimerspec *otmr				
288	sys_accept4	int fd	struct sockaddr *upeer_sockaddr	int *upeer_addrlen	int flags		
289	sys_signalfd4	int ufd	sigset_t *user_mask	size_t sizemask	int flags		
290	sys_eventfd2	unsigned int count	int flags				
291	sys_epoll_create1	int flags					
292	sys_dup3	unsigned int oldfd	unsigned int newfd	int flags			
293	sys_pipe2	int *filedes	int flags				
294	sys_inotify_init1	int flags					
295	sys_preadv	unsigned long fd	const struct iovec *vec	unsigned long vlen	unsigned long pos_l	unsigned long pos_h	
296	sys_pwritev	unsigned long fd	const struct iovec *vec	unsigned long vlen	unsigned long pos_l	unsigned long pos_h	
297	sys_rt_tgsi_gqueueinfo	pid_t tgid	pid_t pid	int sig	siginfo_t *uinfo		
298	sys_perf_event_open	struct perf_event_attr *attr_uptr	pid_t pid	int cpu	int group_fd	unsigned long flags	
299	sys_recvmmsg	int fd	struct mmsghdr *mmsg	unsigned int vlen	unsigned int flags	struct timespec *timeout	
300	sys_fanotify_init	unsigned int flags	unsigned int event_f_flags				
301	sys_fanotify_mark	long fanotify_fd	long flags	__u64 mask	long dfd	long pathname	
302	sys_prlimit64	pid_t pid	unsigned int resource	const struct rlimit64 *new_rlim	struct rlimit64 *old_rlim		
303	sys_name_to_handle_at	int dfd	const char *name	struct file_handle *handle	int *mnt_id	int flag	
304	sys_open_by_handle_at	int dfd	const char *name	struct file_handle *handle	int *mnt_id	int flags	
305	sys_clock_adjtime	clockid_t which_clock	struct timex *tx				
306	sys_syncfs	int fd					
307	sys_sendmmsg	int fd	struct mmsghdr *mmsg	unsigned int vlen	unsigned int flags		
308	sys_setns	int fd	int nstype				
309	sys_getcpu	unsigned *cpup	unsigned *nodep	struct getcpu_cache *unused			
310	sys_process_vm_readv	pid_t pid	const struct iovec *lvec	unsigned long liovcnt	const struct iovec *rvec	unsigned long riovcnt	unsigned long flags
311	sys_process_vm_writev	pid_t pid	const struct iovec *lvec	unsigned long liovcnt	const struct iovec *rvec	unsigned long riovcnt	unsigned long flags
312	sys_kcmp	pid_t pid1	pid_t pid2	int type	unsigned long idx1	unsigned long idx2	
313	sys_finit_module	int fd	const char __user *uargs	int flags			

314	sys_sched_setattr	pid_t pid	struct sched_attr __user *attr	unsigned int flags			
315	sys_sched_getattr	pid_t pid	struct sched_attr __user *attr	unsigned int size	unsigned int flags		
316	sys_renameat2	int olddfd	const char __user *oldname	int newdfd, const char __user *newname	unsigned int flags		
317	sys_seccomp	unsigned int op	unsigned int flags	const char __user *uargs			
318	sys_getrandom	char __user *buf	size_t count	unsigned int flags			
319	sys_memfd_create	const char __user *uname_ptr	unsigned int flags				
320	sys_kexec_file_load	int kernel_fd	int initrd_fd	unsigned long cmdline_len	const char __user *cmdline_ptr	unsigned long flags	
321	sys_bpf	int cmd	union bpf_attr *attr	unsigned int size			
322	stub_execveat	int dfd	const char __user *filename	const char __user *const __user *argv	const char __user *const __user *envp	int flags	