APÉNDICE 1
ENSAMBLADOR EN ENTORNOS LINUX / UNIX

1. Herramientas

Para desarrollar programas ensamblador en Linux, deberemos elegir el compilador de ensamblador que queremos utilizar. La elección lógica es usar ***GAS (as)***, el estándar que lleva toda distribución Linux.

GAS tiene la desventaja de utilizar la sintaxis AT&T, que difiere bastante de la sintaxis Intel (la que conocemos del ensamblador bajo MSDOS).
Una alternativa es utilizar ***NASM***, que ofrece la misma funcionalidad que GAS, pero utiliza la sintaxis de Intel.

Sin embargo, puesto que ***gcc*** *utiliza* ***as*** *como ensamblador, para realizar la práctica de optimización de código con ensamblador en línea* ***debemos dominar la sintaxis AT&T.***

Así pues, los programas que hagamos en esta parte de la práctica los desarrollaremos utilizando las herramientas NASM y GAS.

2. Programar con sintaxis Intel. NASM.

En NASM, un programa ensamblador aparece dividido en tres secciones (datos constantes, variables e instrucciones de programa).

Veamos el ejemplo ***"Hola mundo"*** programado para NASM:

```
section .data
      mensaje     db "hola mundo",0xA
      longitud    equ $ - mensaje

section .text
      global _inicio          ;definimos el punto de entrada
_inicio:
      mov edx,longitud        ;EDX=long. de la cadena
      mov ecx,mensaje         ;ECX=cadena a imprimir
      mov ebx,1               ;EBX=manejador de fichero (STDOUT)
      mov eax,4               ;EAX=función sys_write() del kernel
      int 0x80                ;interrupc. 80 (llamada al kernel)

      mov ebx,0               ;EBX=código de salida al SO
      mov eax,1               ;EAX=función sys_exit() del kernel
      int 0x80                ;interrupc. 80 (llamada al kernel)
```

El fichero lo debemos guardar con extensión ***.asm*** (*hola.asm*). La compilación se lleva a cabo mediante las siguientes órdenes de shell:

```
nasm -f    hola.asm
ld   -s -o hola   hola.o
```

lo que nos genera el ejecutable *hola* que nos mostrará el mensaje definido.

Descripción detallada del programa *"Hola mundo"* (NASM).
En la sección ***.data*** definiremos constantes (datos que no modificaremos durante la ejecución del programa). Aquí podemos incluir mensajes o tamaños de buffers. Las directivas utilizadas para definir los tipos de datos (DB, DD, etc) son las mismas y se usan del mismo modo que en el ensamblador de Intel bajo MSDOS.

```
section .data
      mensaje     db "hola mundo",0xA
      longitud    equ $ - mensaje
```

En este caso, la definición de la cadena de texto es idéntica a la forma en que se hacía en MSDOS (salvo que ***no hay*** que terminarla con el carácter ***$***). Por otro lado, la función del sistema que escribe una cadena de texto, necesita la longitud exacta de la cadena a escribir (ya que no hemos hecho uso de un carácter de final de cadena). En este caso, *longitud* es una constante que almacena el número de bytes (caracteres) desde el comienzo de *mensaje* hasta la definición de *longitud* (el final de *mensaje*).

En la sección ***.bss*** definiremos las variables del programa. Aunque no lo hemos definido en el ejemplo anterior, suele contener entradas como las siguientes:

```
fichero:    resb  256   ;REServa 256 Bytes
caracter:   resb  1     ;REServa 1 Byte (8 bits)
palabra:    resw  1     ;REServa 1 Word (palabra, 16 bits)
numero:     resd  1     ;REServa 1 DoubleWord (doble palabra, 32bits)
num_real:   resq  1     ;REServa 1 float de doble precision (64 bits)
precision:  rest  1     ;REServa 1 float de precision extendida (128 bits)
```

El acceso desde el programa a estas variables se realiza de la forma que ya conocemos.

Por último, la sección ***.text*** es donde escribimos el código ensamblador, de la misma forma que hacíamos en MSDOS. La sintaxis y el acceso a los registros es igual, sin embargo encontraremos algunas diferencias en cuanto a las llamadas al sistema (int 21h en MSDOS) o el acceso a los parámetros de la línea de comandos, etc.

El comienzo del programa se indica mediante la directiva ***global _start*** al principio de la sección *.text* . Es la forma que tenemos para indicarle al kernel cuál es la primera instrucción a ejecutar en nuestro programa. Recordemos cómo indicábamos el punto de entrada en ensamblador de Intel bajo MSDOS (al final del texto del programa, con la directiva *END*):

```
codigo segment 'code'
      main PROC
      ....
      main ENDP
codigo ends
END main
```

En este caso, el funcionamiento es el mismo, indicando justo después del inicio de la sección cuál es el punto de entrada (inicio del programa). Al igual que entonces, ese punto puede ser cualquiera, y lo indicamos con una etiqueta:

```
section .text
      global _start           ;definimos el punto de entrada
_start:
```

En el ejemplo anterior, el resto del programa simplemente define una constante de cadena (el mensaje a mostrar) y hace uso de dos llamadas al sistema para mostrar una cadena por pantalla y para terminar el programa (ver el código para más detalle).

Interacción con el sistema operativo. Llamadas al sistema.
En el ejemplo hemos hecho uso de la interrupción 80h (su funcionalidad se podría equiparar a la 21h del MSDOS) para mostrar la cadena de texto y para terminar el programa y salir al SO.

De hecho, la interacción con el sistema operativo se lleva a cabo a través de las llamadas al sistema (al kernel, realmente) a través de esta interrupción.
Al igual que cuando usábamos la interrupción 21h en MSDOS, ahora debemos indicar en EAX la función del sistema que queremos usar. Si esa función necesita argumentos, estos deben indicarse en los registros EBX, ECX, EDX, ESI, EDI y EBP (en este orden).

Así, para terminar un programa, haremos uso de la función *sys_exit()* que tiene asignado el número 1 (lo damos en EAX), e indicaremos el código de retorno en EBX:

```
mov eax,1
mov ebx,0
int 0x80
```

es muy similar a lo que ya hacíamos en MSDOS (y en este caso como si hubiésemos hecho *return 0;* en el *main* de un programa en C).

En cuanto a la función utilizada para mostrar una cadena, hay que tener en cuenta que todo en Linux / Unix es tratado como un fichero, por lo que para mostrar información por pantalla debemos hacer uso del descriptor STDOUT (definido en */usr/include/unistd.h* con el valor 1). El resto de parámetros que necesita esta función son la dirección de comienzo de la cadena a mostrar (ECX) y la longitud de dicha cadena (EDX).

La sección 5 de este guión presenta una descripción detallada de todas las funciones del sistema, números asignados, y los argumentos que necesitan para su ejecución. De todas formas, podemos obtener más información sobre las llamadas en el archivo que las define (*/usr/include/sys/syscall.h*) y en las páginas de manual del sistema (o con la orden *info*):

```
man 2 exit              info exit
man 2 read              info read
man 2 write             info write
man 2 open              info open
...                     ...
```

Acceso a la pila en un programa ensamblador en Linux.
La estructura de la pila al iniciar un programa en Linux es muy diferente a la estructura en MSDOS. Mientras que entonces la pila estaba vacía al empezar, y los argumentos de la línea de comandos se almacenaban en el PSP (y recuperarlos era engorroso), en el caso de Linux la pila es inicializada con dichos argumentos.

Al igual que cuando programamos en C, en Linux el kernel establece el valor de ciertas variables de entorno que necesitará el programa, y también inicializa el vector de argumentos de línea de comandos y el contador.

Todos esos datos quedan cargados en la pila del programa, de acuerdo a la siguiente estructura:

argc	contador del número de argumentos. Entero de 32bits
argv[0]	nombre del programa. Puntero a la cadena de texto (32bits)
argv[1] argv[2] argv[3] . . . . . . argv[argc-1]	argumentos pasados por la línea de comandos. Punteros a las cadenas de texto (32bits cada uno)
NULL	fin de los argumentos (32bits)
env[1] env[2] env[3] . . . . . . env[n]	variables de entorno para el programa. Punteros a las cadenas de texto (32bits cada uno)
NULL	Fin de las variables de entorno (32bits)

Cuando el kernel carga nuestro programa, establece esa estructura en la pila del programa. Así, si nuestro programa fue llamado de la siguiente forma:

```
$ miprograma  37  hola
```

la pila contendrá la siguiente información:

3	argc
"miprograma"	argv[0]
"37"	argv[1]
"hola"	argv[2]
NULL	fin de los argumentos
...variables de entorno...	
NULL	fin de las variables

Para acceder a cada uno de los argumentos y variables de entorno, vamos recorriendo la pila, extrayendo (*pop*) los valores, teniendo siempre en cuenta que *argc* y *argv[0]* siempre están presentes. La extracción de los argumentos y variables de entorno se debe hacer al principio del programa (para evitar la pérdida de algunos por la manipulación a lo largo del programa):

```
. . .
. . .
_start:
      pop eax           ;extraer el contador de argumentos
      pop ebx           ;extraer nombre del programa (el puntero)
      argumentos:
            pop ecx           ;vamos extrayendo los argumentos
            test ecx,ecx      ;comprobamos si llegamos al NULL
            jnz argumentos
      entorno:
            pop edx           ;vamos extrayendo las variables
            test edx,edx      ;comprobamos si llegamos al NULL
            jnz entorno
. . .
. . .
```

Acceso a ficheros desde un programa ensamblador en Linux.
Veamos un ejemplo más complejo, en el que hagamos uso de la pila y los argumentos de comando de línea, y usemos llamadas al sistema para acceder a ficheros:

```
...
...
pop ebx  ;extraer "argc"
pop ebx  ;extraer argv[0]

pop ebx  ;extraer el primer argumento real (puntero a una cadena)
mov eax,5           ;función para sys_open()
mov ecx,0           ;O_RDONLY (definido en  fcntl.h)
int 0x80            ;interrupc. 80 (llamada al kernel)

test eax,eax  ;comprobar si devuelve error o el descriptor
jns leer_del_fichero

hubo_error:
   mov ebx,eax      ;salir al SO devolviendo el código de error
   mov eax,1
   int 0x80

leer_del_fichero:
   mov ebx,eax      ;no hubo error=>devuelve el descriptor de fich
   mov eax,3        ;función para sys_read()
   mov ecx,buffer   ;variable donde guardaremos lo leido del fich
   mov edx,tamano   ;tamaño de lectura
   int 0x80
   js hubo_error

mostrar_por_pantalla:
   mov edx,eax      ;longitud de la cadena a escribir
   mov eax,4        ;función sys_write()
   mov ebx,1        ;descriptor de STDOUT
   int 0x80
...
...
```

El código anterior lee el nombre de un fichero de la línea de órdenes, y utiliza llamadas al sistema para abrirlo, leer la información que contiene, y mostrarla por pantalla. Básicamente, funciona como el programa "*cat*", aunque habría que mejorarlo para leer toda la información del fichero (el ejemplo completo se mostrará en la sección 6 de este apéndice).

3. Sintaxis AT&T. Ensamblador de GNU

GAS (**G**nu **AS**sembler) utiliza la sintaxis de AT&T, que tiene pequeñas diferencias con respecto a la sintaxis estándar de Intel (usada en TASM, MASM, etc). Las principales diferencias se detallan a continuación:

- En AT&T, a los **nombres de los registros** se les añade el prefijo **%**
  AT&T: `%eax`
  INTEL: `eax`

- En AT&T, el **destino** se coloca a la **derecha** y el **fuente** a la **izquierda** (en Intel es al revés)
  las siguientes instrucciones cargan en **ebx** el valor de **eax**

  AT&T: `movl %eax, %ebx`
  INTEL: `mov ebx, eax`

- En AT&T, a los **valores inmediatos** se les añade el prefijo **$**
  en el siguiente ejemplo, la primera instrucción carga la dirección de la variable en **eax**; la segunda carga el valor en **ebx**

```
AT&T:    movl $var, %eax
         movl $0xf02, %ebx

INTEL:   mov eax, offset var
         mov ebx, 0f02h
```

- En AT&T, el **tamaño del resultado** se especifica con sufijos (**b**, **w** o **l**) en las instrucciones (en Intel cuando hay ambigüedad se utiliza byte ptr, word ptr o dword ptr). Si lo omitimos, GAS intentará "adivinar" el tamaño, y es algo que no queremos que haga...

```
AT&T:    movb var, %ah
         movw %bx, %ax

INTEL:   mov ah, byte ptr var
         mov ax, bx


AT&T:    movb %bl,%al
         movw %bx, %ax
         movl %ebx,%eax
         movl (%ebx),%eax

INTEL:   mov al,bl
         mov ax, bx
         mov eax,ebx
         mov eax, dword ptr [ebx]
```

- **Direccionamiento a memoria:**

Es uno de los aspectos que más cambian. Veamos la sintaxis de Intel para hacer un direccionamiento a base, con índice y desplazamiento:

```
[ base + indice*escala + desplazamiento ]
```

en la sintaxis AT&T esto queda como sigue:

```
desplazamiento ( base , indice , escala )
```

Veamos dos ejemplos:

AT&T: `movl array (, %eax, 4), %edx`
INTEL: `mov edx, array[eax*4]`

```
AT&T:    movl (%ebx) , %eax
         movl 3(%ebx) , %eax
```

INTEL:
```
mov eax , [ebx]
mov eax,[ebx+3]
```

- **Salto lejano**

AT&T:
```
lcall $sección, $offset
ljmp  $sección, $offset
lret  $V
```

INTEL:
```
call far sección:offset
jmp far  sección:offset
ret      far V
```

- **Nemotécnico.** Varían los nemotécnicos de algunas instrucciones

AT&T:
```
movswl %ax, %ecx
movzbw %ah, %cx
cbtw
cwtl
cwtd
cltd
```

INTEL:
```
movsx ecx, ax
movzx cx, ah
cbw
cwde
cwd
cdq
```

- **Directivas del compilador.**
  Como vimos, los programas ensamblador, además de las instrucciones que componen el programa, contienen órdenes al compilador que le servirán para definir las secciones del programa, definir los tipos de datos, macros, etc. (directivas del compilador).

  Como comentamos más arriba, hay diferencias en cuanto a algunas directivas al programar con el ensamblador GAS o NASM.

  - En ambos ensambladores hay que definir las secciones de datos constantes, variables y código utilizando los mismos nombres (***.data .bss .text***). Sin embargo, la directiva utilizada para definir las secciones difiere de un ensamblador a otro:

NASM	GAS
section .data	[.section] .data
section .bss	[.section] .bss
section .text	[.section] .text

- En ambos ensambladores, la etiqueta de entrada al programa ensamblador debe ser ***_start***. Sin embargo, la directiva utilizada difiere de un ensamblador a otro:

NASM	GAS
section .text global _start _start:	.text .globl _start _start:

- La definición de datos constantes se lleva a cabo utilizando de la misma forma, pero utilizando palabras reservadas diferentes:

NASM	GAS
section .data cadena db "un texto" longitud equ $ - cadena cero dw 0 letra db 'A'	.data cadena: .ascii "un texto" longitud = . – cadena cero: .hword 0 letra: .byte 'A'

En la página web del DJGPP podemos encontrar una guía detallada sobre la sintaxis AT&T, y ejemplos de ensamblador en línea:

http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

El programa "Hola mundo" con GAS.

Veamos el ejemplo que explicamos para NASM, esta vez en el formato AT&T (sólo hay que tener en cuenta las diferencias comentadas anteriormente):

```
.section .data
      mensaje: .ascii "hola mundo \n"
      longitud = . - mensaje

.section .text
      .globl _start           #definimos el punto de entrada
_start:
      movl $longitud,%edx     #EDX=long. de la cadena
      movl $mensaje,%ecx      #ECX=cadena a imprimir
      movl $1,%ebx            #EBX=manejador de fichero (STDOUT)
      movl $4,%eax            #EAX=función sys_write() del kernel
      int $0x80               #interrupc. 80 (llamada al kernel)

      movl $0,%ebx            #EBX=código de salida al SO
      movl $1,%eax            #EAX=función sys_exit() del kernel
      int $0x80               #interrupc. 80 (llamada al kernel)
```

El fichero lo debemos guardar con extensión **.*s*** (*hola.s*). La compilación se lleva a cabo mediante las siguientes órdenes de shell:

```
as –o hola.o hola.s
ld –o hola   hola.o
```

lo que nos genera el ejecutable *hola* que nos mostrará el mensaje definido.

Vemos varias diferencias, tanto en cuanto a la sintaxis como en las directivas al compilador (palabras reservadas).

Descripción del programa *"Hola mundo"* (GAS).
El ejemplo es prácticamente igual al que vimos en la sintaxis Intel (NASM), de hecho se utilizan las mismas instrucciones. Sólo hay que tener en cuenta las diferencias de sintaxis comentadas más arriba.

4. Formato binario de un ejecutable ELF

Durante el proceso de carga de un programa ELF (ejecutable bajo Linux) se inicializan diversas zonas de memoria (zona de variables y la pila) y los registros del procesador.

Veamos cómo actúan dichos procesos y el estado en que queda la memoria y los registros (ya hemos visto la pila), aunque la información que demos aquí sólo será aplicable a programas ensamblador "planos" (programados para gas / nasm y compilados con estos); la inicialización de la pila y registros no será la misma si compilamos y linkamos con gcc (éste inserta su propio código de inicio antes de pasar el control a la función *main*).

La fuente de información sobre el formato ELF más completa y detallada es el fichero fuente del kernel */usr/source/fs/binfmt_elf.c*
Los procesos de carga e inicialización quedan descritos en el fichero fuente del kernel */usr/include/linux/sched.h*

Ejecución de un programa de usuario
Todo programa de usuario es ejecutado mediante la función del sistema *sys_execve()* , normalmente al escribir el nombre en el prompt del shell. A continuación diversos procesos del kernel se ponen en marcha para cargar el programa en memoria y comenzar su ejecución:

Función del sistema	Fichero del kernel	Comentarios
shell		escribimos el nombre del programa y pulsamos ENTER
execve()		el shell llama a la función correspondiente de *libc*
sys_execve()		*libc* pasa la llamada al kernel
sys_execve()	arch/i386/kernel/process.c	la llamada llega al espacio del kernel
do_execve()	fs/exec.c	abre el fichero
search_binary_handler()	fs/exec.c	obtiene el tipo de ejecutable
load_elf_binary()	fs/binfmt_elf.c	carga el binario ELF y las librerías necesarias. Inicializa la memoria
start_thread()	include/asm-i386/processor.h	pasa el control al código del programa de usuario

El programa ELF en memoria
Una vez cargado el binario ELF, la estructura de la memoria asignada al programa de usuario es la siguiente:

dirección 0x08048000

código	sección .text (código máquina del programa)
datos	sección .data (datos constantes del programa)
bss	sección .bss (variables del programa)
. . . .	espacio libre de memoria
pila argumentos vars.entorno	pila del programa, inicializada con los argumentos de la línea de comandos y las variables de entorno
nombre del ejecutable	duplicado en la pila (argv[0])
NULL	32 bits inicializados al valor 0
dirección 0xBFFFFFFF	

La pila crece hacia direcciones de memoria menores (hacia arriba), hasta encontrarse con la sección .bss . Esta sección de variables del programa es totalmente inicializada con el valor 0 en el inicio, de forma que cualquier variable definida no contendrá un valor aleatorio al principio. Podemos evitarnos el trabajo de inicializar una variable a 0 en nuestro programa, simplemente definiéndola en esta sección.
El espacio libre de memoria después del .bss queda para asignación dinámica de memoria (*malloc()*).

Inicialización de los registros
Al pasar el control a la primera instrucción de nuestro programa, el kernel (según su versión) habrá puesto los valores de los registros generales (EAX, EBX, ECX, EDX, ESI, EDI, EBP) a cero, o bien habrá dejado los valores que tenían justo antes de que el programa llamador hiciese la llamada:

- En el kernel de versión 2.0 EAX y EDX quedan inicializados a 0, mientras que el resto contienen los valores que tenían justo antes de la llamada a *sys_execve()*
- En el kernel de versión 2.2 todos los registros generales quedan inicializados a 0

5. *Lista de algunas de las llamadas al sistema*

La siguiente tabla muestra la lista de llamadas al sistema. Como ya hemos comentado, estas llamadas son como un API entre el espacio del kernel y del programa de usuario.

A la izquierda quedan los números de las funciones (valores a poner en EAX para hacer la llamada). A la derecha aparecen los tipos de valores que espera en cada registro de carácter general (parámetros de la función) antes de llamar a la interrupción 80h. Tras cada llamada, se devuelve un número entero en EAX (código de retorno).

%eax	Función	Fuentes del kernel	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-

12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-
23	sys_setuid	kernel/sys.c	uid_t	-	-	-	-
24	sys_getuid	kernel/sched.c	-	-	-	-	-
25	sys_stime	kernel/time.c	int *	-	-	-	-
26	sys_ptrace	arch/i386/kernel/ptrace.c	long	long	long	long	-
27	sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
28	sys_fstat	fs/stat.c	unsigned int	struct __old_kernel_stat *	-	-	-
29	sys_pause	arch/i386/kernel/sys_i386.c	-	-	-	-	-
30	sys_utime	fs/open.c	char *	struct utimbuf *	-	-	-
33	sys_access	fs/open.c	const char *	int	-	-	-
34	sys_nice	kernel/sched.c	int	-	-	-	-
36	sys_sync	fs/buffer.c	-	-	-	-	-
37	sys_kill	kernel/signal.c	int	int	-	-	-
38	sys_rename	fs/namei.c	const char *	const char *	-	-	-
39	sys_mkdir	fs/namei.c	const char *	int	-	-	-
40	sys_rmdir	fs/namei.c	const char *	-	-	-	-
41	sys_dup	fs/fcntl.c	unsigned int	-	-	-	-
42	sys_pipe	arch/i386/kernel/sys_i386.c	unsigned long *	-	-	-	-
43	sys_times	kernel/sys.c	struct tms *	-	-	-	-
45	sys_brk	mm/mmap.c	unsigned long	-	-	-	-
46	sys_setgid	kernel/sys.c	gid_t	-	-	-	-
47	sys_getgid	kernel/sched.c	-	-	-	-	-
48	sys_signal	kernel/signal.c	int	__sighandler_t	-	-	-
49	sys_geteuid	kernel/sched.c	-	-	-	-	-
50	sys_getegid	kernel/sched.c	-	-	-	-	-
51	sys_acct	kernel/acct.c	const char *	-	-	-	-
52	sys_umount	fs/super.c	char *	int	-	-	-
54	sys_ioctl	fs/ioctl.c	unsigned int	unsigned int	unsigned long	-	-
55	sys_fcntl	fs/fcntl.c	unsigned int	unsigned int	unsigned long	-	-
57	sys_setpgid	kernel/sys.c	pid_t	pid_t	-	-	-
59	sys_olduname	arch/i386/kernel/sys_i386.c	struct oldold_utsname *	-	-	-	-
60	sys_umask	kernel/sys.c	int	-	-	-	-
61	sys_chroot	fs/open.c	const char *	-	-	-	-
62	sys_ustat	fs/super.c	dev_t	struct ustat *	-	-	-
63	sys_dup2	fs/fcntl.c	unsigned int	unsigned int	-	-	-
64	sys_getppid	kernel/sched.c	-	-	-	-	-
65	sys_getpgrp	kernel/sys.c	-	-	-	-	-
66	sys_setsid	kernel/sys.c	-	-	-	-	-
67	sys_sigaction	arch/i386/kernel/signal.c	int	const struct old_sigaction *	struct old_sigaction *	-	-
68	sys_sgetmask	kernel/signal.c	-	-	-	-	-
69	sys_ssetmask	kernel/signal.c	int	-	-	-	-
70	sys_setreuid	kernel/sys.c	uid_t	uid_t	-	-	-
71	sys_setregid	kernel/sys.c	gid_t	gid_t	-	-	-
72	sys_sigsuspend	arch/i386/kernel/signal.c	int	int	old_sigset_t	-	-
73	sys_sigpending	kernel/signal.c	old_sigset_t *	-	-	-	-
74	sys_sethostname	kernel/sys.c	char *	int	-	-	-

75	sys_setrlimit	kernel/sys.c	unsigned int	struct rlimit *	-	-	-
76	sys_getrlimit	kernel/sys.c	unsigned int	struct rlimit *	-	-	-
77	sys_getrusage	kernel/sys.c	int	struct rusage *	-	-	-
78	sys_gettimeofday	kernel/time.c	struct timeval *	struct timezone *	-	-	-
79	sys_settimeofday	kernel/time.c	struct timeval *	struct timezone *	-	-	-
80	sys_getgroups	kernel/sys.c	int	gid_t *	-	-	-
81	sys_setgroups	kernel/sys.c	int	gid_t *	-	-	-
82	old_select	arch/i386/kernel/sys_i386.c	struct sel_arg_struct *	-	-	-	-
83	sys_symlink	fs/namei.c	const char *	const char *	-	-	-
84	sys_lstat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
85	sys_readlink	fs/stat.c	const char *	char *	int	-	-
86	sys_uselib	fs/exec.c	const char *	-	-	-	-
87	sys_swapon	mm/swapfile.c	const char *	int	-	-	-
88	sys_reboot	kernel/sys.c	int	int	int	void *	-
89	old_readdir	fs/readdir.c	unsigned int	void *	unsigned int	-	-
90	old_mmap	arch/i386/kernel/sys_i386.c	struct mmap_arg_struct *	-	-	-	-
91	sys_munmap	mm/mmap.c	unsigned long	size_t	-	-	-
92	sys_truncate	fs/open.c	const char *	unsigned long	-	-	-
93	sys_ftruncate	fs/open.c	unsigned int	unsigned long	-	-	-
94	sys_fchmod	fs/open.c	unsigned int	mode_t	-	-	-
95	sys_fchown	fs/open.c	unsigned int	uid_t	gid_t	-	-
96	sys_getpriority	kernel/sys.c	int	int	-	-	-
97	sys_setpriority	kernel/sys.c	int	int	int	-	-
99	sys_statfs	fs/open.c	const char *	struct statfs *	-	-	-
100	sys_fstatfs	fs/open.c	unsigned int	struct statfs *	-	-	-
101	sys_ioperm	arch/i386/kernel/ioport.c	unsigned long	unsigned long	int	-	-
102	sys_socketcall	net/socket.c	int	unsigned long *	-	-	-
103	sys_syslog	kernel/printk.c	int	char *	int	-	-
104	sys_setitimer	kernel/itimer.c	int	struct itimerval *	struct itimerval *	-	-
105	sys_getitimer	kernel/itimer.c	int	struct itimerval *	-	-	-
106	sys_newstat	fs/stat.c	char *	struct stat *	-	-	-
107	sys_newlstat	fs/stat.c	char *	struct stat *	-	-	-
108	sys_newfstat	fs/stat.c	unsigned int	struct stat *	-	-	-
109	sys_uname	arch/i386/kernel/sys_i386.c	struct old_utsname *	-	-	-	-
110	sys_iopl	arch/i386/kernel/ioport.c	unsigned long	-	-	-	-
111	sys_vhangup	fs/open.c	-	-	-	-	-
112	sys_idle	arch/i386/kernel/process.c	-	-	-	-	-
113	sys_vm86old	arch/i386/kernel/vm86.c	unsigned long	struct vm86plus_struct *	-	-	-
114	sys_wait4	kernel/exit.c	pid_t	unsigned long *	int options	struct rusage *	-
115	sys_swapoff	mm/swapfile.c	const char *	-	-	-	-
116	sys_sysinfo	kernel/info.c	struct sysinfo *	-	-	-	-
117	sys_ipc(*Note)	arch/i386/kernel/sys_i386.c	uint	int	int	int	void *
118	sys_fsync	fs/buffer.c	unsigned int	-	-	-	-
119	sys_sigreturn	arch/i386/kernel/signal.c	unsigned long	-	-	-	-
120	sys_clone	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
121	sys_setdomainname	kernel/sys.c	char *	int	-	-	-
122	sys_newuname	kernel/sys.c	struct new_utsname *	-	-	-	-
123	sys_modify_ldt	arch/i386/kernel/ldt.c	int	void *	unsigned long	-	-
124	sys_adjtimex	kernel/time.c	struct timex *	-	-	-	-
125	sys_mprotect	mm/mprotect.c	unsigned long	size_t	unsigned long	-	-
126	sys_sigprocmask	kernel/signal.c	int	old_sigset_t *	old_sigset_t *	-	-
127	sys_create_module	kernel/module.c	const char *	size_t	-	-	-

128	sys_init_module	kernel/module.c	const char *	struct module *	-	-	-
129	sys_delete_module	kernel/module.c	const char *	-	-	-	-
130	sys_get_kernel_syms	kernel/module.c	struct kernel_sym *	-	-	-	-
131	sys_quotactl	fs/dquot.c	int	const char *	int	caddr_t	-
132	sys_getpgid	kernel/sys.c	pid_t	-	-	-	-
133	sys_fchdir	fs/open.c	unsigned int	-	-	-	-
134	sys_bdflush	fs/buffer.c	int	long	-	-	-
135	sys_sysfs	fs/super.c	int	unsigned long	unsigned long	-	-
136	sys_personality	kernel/exec_domain.c	unsigned long	-	-	-	-
138	sys_setfsuid	kernel/sys.c	uid_t	-	-	-	-
139	sys_setfsgid	kernel/sys.c	gid_t	-	-	-	-
140	sys_llseek	fs/read_write.c	unsigned int	unsigned long	unsigned long	loff_t *	unsigned int
141	sys_getdents	fs/readdir.c	unsigned int	void *	unsigned int	-	-
142	sys_select	fs/select.c	int	fd_set *	fd_set *	fd_set *	struct timeval *
143	sys_flock	fs/locks.c	unsigned int	unsigned int	-	-	-
144	sys_msync	mm/filemap.c	unsigned long	size_t	int	-	-
145	sys_readv	fs/read_write.c	unsigned long	const struct iovec *	unsigned long	-	-
146	sys_writev	fs/read_write.c	unsigned long	const struct iovec *	unsigned long	-	-
147	sys_getsid	kernel/sys.c	pid_t	-	-	-	-
148	sys_fdatasync	fs/buffer.c	unsigned int	-	-	-	-
149	sys_sysctl	kernel/sysctl.c	struct __sysctl_args *	-	-	-	-
150	sys_mlock	mm/mlock.c	unsigned long	size_t	-	-	-
151	sys_munlock	mm/mlock.c	unsigned long	size_t	-	-	-
152	sys_mlockall	mm/mlock.c	int	-	-	-	-
153	sys_munlockall	mm/mlock.c	-	-	-	-	-
154	sys_sched_setparam	kernel/sched.c	pid_t	struct sched_param *	-	-	-
155	sys_sched_getparam	kernel/sched.c	pid_t	struct sched_param *	-	-	-
156	sys_sched_setscheduler	kernel/sched.c	pid_t	int	struct sched_param *	-	-
157	sys_sched_getscheduler	kernel/sched.c	pid_t	-	-	-	-
158	sys_sched_yield	kernel/sched.c	-	-	-	-	-
159	sys_sched_get_priority_max	kernel/sched.c	int	-	-	-	-
160	sys_sched_get_priority_min	kernel/sched.c	int	-	-	-	-
161	sys_sched_rr_get_interval	kernel/sched.c	pid_t	struct timespec *	-	-	-
162	sys_nanosleep	kernel/sched.c	struct timespec *	struct timespec *	-	-	-
163	sys_mremap	mm/mremap.c	unsigned long	unsigned long	unsigned long	unsigned long	-
164	sys_setresuid	kernel/sys.c	uid_t	uid_t	uid_t	-	-
165	sys_getresuid	kernel/sys.c	uid_t *	uid_t *	uid_t *	-	-
166	sys_vm86	arch/i386/kernel/vm86.c	struct vm86_struct *	-	-	-	-
167	sys_query_module	kernel/module.c	const char *	int	char *	size_t	size_t *
168	sys_poll	fs/select.c	struct pollfd *	unsigned int	long	-	-
169	sys_nfsservctl	fs/filesystems.c	int	void *	void *	-	-
170	sys_setresgid	kernel/sys.c	gid_t	gid_t	gid_t	-	-
171	sys_getresgid	kernel/sys.c	gid_t *	gid_t *	gid_t *	-	-
172	sys_prctl	kernel/sys.c	int	unsigned long	unsigned long	unsigned long	unsigned long
173	sys_rt_sigreturn	arch/i386/kernel/signal.c	unsigned long	-	-	-	-
174	sys_rt_sigaction	kernel/signal.c	int	const struct sigaction *	struct sigaction *	size_t	-
175	sys_rt_sigprocmask	kernel/signal.c	int	sigset_t *	sigset_t *	size_t	-

176	sys_rt_sigpending	kernel/signal.c	sigset_t *	size_t	-	-	-
177	sys_rt_sigtimedwait	kernel/signal.c	const sigset_t *	siginfo_t *	const struct timespec *	size_t	-
178	sys_rt_sigqueueinfo	kernel/signal.c	int	int	siginfo_t *	-	-
179	sys_rt_sigsuspend	arch/i386/kernel/signal.c	sigset_t *	size_t	-	-	-
180	sys_pread	fs/read_write.c	unsigned int	char *	size_t	loff_t	-
181	sys_pwrite	fs/read_write.c	unsigned int	const char *	size_t	loff_t	-
182	sys_chown	fs/open.c	const char *	uid_t	gid_t	-	-
183	sys_getcwd	fs/dcache.c	char *	unsigned long	-	-	-
184	sys_capget	kernel/capability.c	cap_user_header_t	cap_user_data_t	-	-	-
185	sys_capset	kernel/capability.c	cap_user_header_t	const cap_user_data_t	-	-	-
186	sys_sigaltstack	arch/i386/kernel/signal.c	const stack_t *	stack_t *	-	-	-
187	sys_sendfile	mm/filemap.c	int	int	off_t *	size_t	-
190	sys_vfork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-

6. *Ejemplos*

Uso de macros en ensamblador (GAS)
Igual que en MSDOS, podemos hacer uso de macros para facilitar la programación. Para ello, debemos utilizar la directiva .macro de la siguiente forma:

```
.macro nombreMacro
      instrucciones
.endm
```

En el ejemplo definiremos una macro para terminar el programa, otra para mostrar una cadena por salida estándar, y otra para leer cadenas de texto desde entrada estándar.
La forma de llamar a una macro es idéntica a como se hacía bajo MSDOS, incluso en la forma de pasarle los valores:

```
#COMPILAR:
#     as -o m.o m.s
#     ls -o m   m.o

.macro terminar
      movl $1,%eax
      movl $0,%ebx
      int  $0x80
.endm

#espera ECX=cadena ; EDX=longitud
.macro escribir_cadena cadena longitud
      movl $4,%eax
      movl $1,%ebx      #stdout
      movl \cadena,%ecx
      movl \longitud,%edx
      int  $0x80
.endm

#espera ECX=cadena ; EDX=longitud
.macro leer_cadena cadena longitud
      movl $3,%eax
      movl $0,%ebx      #stdin
      movl \cadena,%ecx
      movl \longitud,%edx
      int  $0x80
.endm
```

```
.data
      retorno:  .byte  0x0A
      mensaje1: .ascii "\n Introduce una cadena: "
      longitud1 = . - mensaje1
      buffer: .ascii "          "

.text
      .globl _start

_start:
      escribir_cadena $mensaje1 $longitud1
      leer_cadena     $buffer $10

      escribir_cadena $retorno $1
      escribir_cadena $buffer $10

      escribir_cadena $retorno $1
      terminar
```

Uso de funciones en ensamblador (GAS)
Igual que en MSDOS, podemos hacer uso de funciones para facilitar la programación. Dentro de la sección ***.text*** (y antes del punto de entrada al programa) podemos definir las diferentes funciones (subrutinas), utilizando una etiqueta que indiquen el inicio de la función y cuidando siempre terminar la ejecución de ésta con la instrucción ***ret***. Una función tendrá el siguiente aspecto:

```
nombreFuncion:
      instrucciones
      ret
```

Veamos el ejemplo anterior (macros) utilizando tres subrutinas. Como se verá en el programa principal, el paso de parámetros a la función hay que hacerlo a través de la pila o en los registros o variables globales del programa (según como se haya programado la subrutina):

```
#COMPILAR:
#     as -o f.o f.s
#     ls -o f   f.o

.data
      retorno:  .byte  0x0A
      mensaje1: .ascii "\n Introduce una cadena: "
      longitud1 = . - mensaje1
      buffer: .ascii "          "

.text
      .globl _start

funcion_terminar:
      movl $1,%eax
      movl $0,%ebx
      int  $0x80
      ret

#parámetros  ECX=cadena ; EDX=longitud
funcion_escribir_cadena:
```

```
        movl $4,%eax
        movl $1,%ebx      #stdout
        int  $0x80
        ret

#parámetros  ECX=cadena ; EDX=longitud
funcion_leer_cadena:
        movl $3,%eax
        movl $0,%ebx      #stdin
        int  $0x80
        ret

_start:
        #los parámetros se pasan en los registros
        movl $mensaje1,%ecx
        movl $longitud1,%edx
        call funcion_escribir_cadena

        movl $buffer,%ecx
        movl $10,%edx
        call funcion_leer_cadena

        movl $retorno,%ecx
        movl $1,%edx
        call funcion_escribir_cadena

        movl $buffer,%ecx
        movl $10,%edx
        call funcion_escribir_cadena

        movl $retorno,%ecx
        movl $1,%edx
        call funcion_escribir_cadena

        #esta última no necesita ningún parámetro
        call funcion_terminar
```

Lectura de parámetros de la línea de comandos (GAS)

Veamos un ejemplo de lectura de los argumentos de línea de comando, programado para el ensamblador GAS. En este ejemplo se hace uso de todo lo descrito en "*Acceso a la pila en un programa ensamblador en Linux*":

```
#COMPILAR:
#     as -o parametros.o parametros.s
#     ls -o parametros   parametros.o
.data
        retorno:    .byte 0x0A

.text
        .globl _start
_start:
        pop %eax    #extraer de la pila el ARGC
        repetir:    #bucle para recorrer todos los argumentos
              pop %eax    #extraer el ARGV[i]
              testl %eax,%eax #comprobar si es NULL
              jz terminar
              call funcion_pintar_cadena  #llamada a la funcion
              jmp repetir

        terminar:
```

```
            movl $1, %eax     #funcion del sistema para terminar
            movl $0, %ebx
            int $0x80

funcion_pintar_cadena:        #definicion de una funcion
      movl %eax,%ecx          #el parametro ha sido pasado en EAX
      xorl %edx,%edx
    contar:             #debemos calcular la longitud del param.
      movb (%ecx,%edx,$1),%al
      testb %al,%al           #comprobar el caracter de la cadena
      jz fin_contar
      incl %edx         #vamos incrementando el calculo en EDX
      jmp contar
    fin_contar:
      movl $4,%eax      #una vez calculada la longitud,se muestra
      movl $1,%ebx
      int $0x80

      movl $4,%eax            #mostramos el RETORNO_CARRO
      movl $retorno,%ecx
      movl $1,%edx            #es un solo caracter
      int $0x80

      ret
```

Para cada parámetro llamamos a una función que lo muestre por salida estándar. Para ello debe calcular la longitud de la cadena (argumento actual), contando uno por uno cada carácter que la forma. Tras cada argumento impreso, se hace un retorno de carro (es una cadena de caracteres de longitud 1). El programa muestra también el nombre del ejecutable como primer argumento (sería casi inmediato hacer que sólo muestre los argumentos reales).

Lectura del contenido de un fichero (NASM)

El siguiente ejemplo lee los 1024 primeros bytes de un fichero que le pasemos como primer argumento por la línea de comandos y los muestra por salida estándar. En este ejemplo, la sintaxis utilizada ha sido la de Intel (NASM).

```
;COMPILAR:
;     nasm -f elf acceso_a_fich.asm
;     ld -s -o acceso_a_fich acceso_a_fich.o
section .data
      mensaje     db 0xA,"---vamos a probar esto---",0xA
      longitud    equ $ - mensaje
      mensaje2    db 0xA,"---hemos terminado---",0xA
      longitud2   equ $ - mensaje2

      tamano      equ 1024

section .bss
      buffer:     resb  1024

section .text
      global _start           ;definimos el punto de entrada
_start:
      mov edx,longitud        ;EDX=long. de la cadena
      mov ecx,mensaje         ;ECX=cadena a imprimir
      mov ebx,1               ;EBX=manejador de fichero (STDOUT)
```

```
mov eax,4               ;EAX=función sys_write() del kernel
int 0x80                ;interrupc. 80 (llamada al kernel)

pop ebx     ;extraer "argc"
pop ebx     ;extraer argv[0] (nombre del ejecutable)

pop ebx     ;extraer el primer arg real (puntero a cadena)
mov eax,5         ;función para sys_open()
mov ecx,0         ;O_RDONLY (definido en  fcntl.h)
int 0x80          ;interrupc. 80 (llamada al kernel)

test eax,eax      ;comprobar si dev. error o el descriptor
jns leer_del_fichero

hubo_error:
mov ebx,eax       ;terminar, devolviendo el código de error
mov eax,1
int 0x80

leer_del_fichero:
      mov ebx,eax       ;no hay error=>devuelve descriptor
      push ebx
      mov eax,3         ;función para sys_read()
      mov ecx,buffer    ;variable donde guardamos lo leido
      mov edx,tamano    ;tamaño de lectura
      int 0x80
      js hubo_error

mostrar_por_pantalla:
      mov edx,eax       ;longitud de la cadena a escribir
      mov eax,4         ;función sys_write()
      mov ebx,1         ;descriptor de STDOUT
      int 0x80

cerrar_fichero:
      pop ebx
      mov eax,6         ;función para cerrar un fichero
      int 0x80


      mov edx,longitud2       ;EDX=long. de la cadena
      mov ecx,mensaje2        ;ECX=cadena a imprimir
      mov ebx,1         ;EBX=manejador de fichero (STDOUT)
      mov eax,4         ;EAX=función sys_write() del kernel
      int 0x80          ;interrupc. 80 (llamada al kernel)


mov ebx,0               ;EBX=código de salida al SO
mov eax,1               ;EAX=función sys_exit() del kernel
int 0x80                ;interrupc. 80 (llamada al kernel)
```

Hemos hecho uso de datos constantes (sección *.data*) y variables (sección *.bss*) donde guardamos los datos leídos del fichero. El acceso al fichero para abrirlo, leerlo y cerrarlo se hace mediante las funciones del sistema (*int 80h*) de forma muy sencilla.

7. Depuración de código. Uso de gdb

Al igual que con el Turbo Debugger, en Linux podemos hacer uso del ***gdb*** para depurar el código que hemos escrito (trazar paso a paso, comprobar el valor de ciertos registros en cada momento, etc).

Para ello, debemos ensamblar nuestros programas con una opción especial del ***as*** :

```
as –a --gstabs  –o prog.o  prog.s
```

la opción ***–a*** nos mostrará un listado de memoria durante el proceso de ensamblaje, donde podremos ver la localización de las variables y código respecto al principio de los segmentos de código y datos. La opción ***—gstabs*** introduce información de depuración en el fichero binario, que luego usará ***gdb***.

```
host:~/asm$ as -a --gstabs -o h.o hola.s
GAS LISTING hola.s                              page 1
   1                            ## hola.s
   2
   3                            ## COMPILAR:
   4                            ## as -o hola.o hola.s
   5                            ## ld -o hola   hola.o
   6
   7                            ## muestra una cadena de
   8
   9
########################################################################
  10                            .section .data
  11                    hola:
  12 0000 486F6C61              .ascii  "Hola!\n"
  12      210A
  13                    hola_len:
  14 0006 06000000              .long   . - hola
  15
########################################################################
  16                            .section .text
  17                            .globl _start
  18
  19                    _start:
  20 0000 31DB                  xorl %ebx, %ebx         # %ebx = 0
  21 0002 B8040000              movl $4, %eax           # llamada a write()
  21      00
  22 0007 31DB                  xorl %ebx, %ebx         # %ebx = 0
  23 0009 43                    incl %ebx               # %ebx = 1, fd = stdout
  24 000a 8D0D0000              leal hola, %ecx         # %ecx ---> hola
  24      0000
  25 0010 8B150600              movl hola_len, %edx     # %edx = longitud
  25      0000
  26 0016 CD80                  int $0x80               # ejecuta write()
  27
  28                            ## termina con la llamada a la funcion _exit()
  29 0018 31C0                  xorl %eax, %eax         # %eax = 0
  30 001a 40                    incl %eax               # %eax = 1 _exit ()
  31 001b 31DB                  xorl %ebx, %ebx         # %ebx = 0 cod. retorno
  32 001d CD80                  int $0x80               # ejecuta _exit ()
 GAS LISTING hola.s                             page 2


DEFINED SYMBOLS
             hola.s:11     .data:00000000 hola
             hola.s:13     .data:00000006 hola_len
             hola.s:19     .text:00000000 _start
NO UNDEFINED SYMBOLS
```

El proceso de linkado se lleva a cabo con la instrucción que conocemos:

```
ld –o prog  prog.o
```

Primer paso de la depuración: llamar a ***gdb*** indicándole el ejecutable a depurar

```
gdb ./prog
```

```
host:~/asm$ gdb ./h
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

La orden ***l*** muestra el texto del programa de 10 en 10 líneas:

```
(gdb) l
1               ## hola.s
2
3               ## COMPILAR:
4               ## as -o hola.o hola.s
5               ## ld -o hola   hola.o
6
7               ## muestra una cadena de texto utilizando la llamada al sistema
write()
8
9       ############################################################################
10              .section .data
(gdb) l
11      hola:
12              .ascii  "Hola!\n"
13      hola_len:
14              .long   . - hola
15      ############################################################################
16              .section .text
17              .globl _start
18
19      _start:
20              xorl %ebx, %ebx         # %ebx = 0
(gdb) l
21              movl $4, %eax           # llamada a write()
22              xorl %ebx, %ebx         # %ebx = 0
23              incl %ebx               # %ebx = 1, fd = stdout
24              leal hola, %ecx         # %ecx ---> hola
25              movl hola_len, %edx     # %edx = longitud
26              int $0x80               # ejecuta write()
27
28              ## termina con la llamada a la funcion _exit()
29              xorl %eax, %eax         # %eax = 0
30              incl %eax               # %eax = 1 _exit ()
(gdb)
```

Antes de ejecutar el programa debemos establecer dos puntos de ruptura (***break***): uno correspondiente a la etiqueta de comienzo del programa (_start) y otro en la línea siguiente (en el primero no para, pero es necesario ponerlo...). Vemos que al poner el primer punto, nos indica un número de línea. Nosotros debemos poner otro punto en la línea cuyo número es el siguiente al que nos acaba de indicar.
Una vez hecho esto, ya podemos ejecutar el programa (***run***):

```
(gdb) break _start
Breakpoint 1 at 0x8048074: file hola.s, line 20.
(gdb) break 21
Breakpoint 2 at 0x8048076: file hola.s, line 21.
(gdb) run
Starting program: /home/pedro/asm_linux/asm-tut/h
Breakpoint 2, _start () at hola.s:21
21              movl $4, %eax           # llamada a write()
Current language:  auto; currently asm
```

Podemos ir viendo los valores de los registros mediante ***info registers*** o bien con ***p/x $registro***

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbffff990       0xbffff990
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048076        0x8048076
eflags         0x200246 2097734
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x0      0
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
xmm0           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm1           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm2           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm3           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm4           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm5           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm6           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm7           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
mxcsr          0x0      0
orig_eax       0xffffffff       -1

(gdb) p/x $eax
$1 = 0x0
```

La traza paso a paso del programa la haremos con la orden ***step*** . A cada paso nos va mostrando la instrucción a ejecutar a continuación; mediante las órdenes anteriores podremos ir viendo cómo cambia el contenido de los registros:

```
(gdb) step
22              xorl %ebx, %ebx         # %ebx = 0
(gdb) step
23              incl %ebx               # %ebx = 1, fd = stdout
(gdb) step
24              leal hola, %ecx         # %ecx ---> hola
(gdb) p/x $eax
$2 = 0x4
(gdb) p/x $ebx
$3 = 0x1
```

8. Trabajo a desarrollar

Probar los programas de ejemplo del guión (tanto los escritos para NASM como los escritos para GAS) y comprobar el correcto funcionamiento de los mismos, corrigiendo cualquier error que pudiera encontrarse en ellos.

Pasar los programas de la sección 6, escritos en la sintaxis AT&T (GAS), a la sintaxis Intel (NASM).

Pasar los programas de la sección 6, escritos en la sintaxis Intel (NASM), a la sintaxis AT&T (GAS).

9. Enlaces interesantes

[1] http://linuxassembly.org
[2] http://linuxassembly.org/articles/linasm.html
[3] http://www.leto.net/papers/writing-a-useful-program-with-nasm.txt
[4] http://linuxassembly.org/howto/hello.html
[5] http://linuxassembly.org/startup.html
[6] http://linuxassembly.org/articles/startup.html
[7] http://www.janw.easynet.be/eng.html
[8] http://www.gnu.org/manual/gas
[9] http://www.gnu.org/onlinedocs/gcc_toc.html
[10] http://www.gnu.org/manual/gdb-4.17/gdb.html