Bisoños Usuarios de GNU/Linux de Mallorca y Alrededores | Bergantells Usuaris de GNU/Linux de Mallorca i Afegitons

Tus primeros programas en ensamblador bajo Linux **(63104 lectures)**

Per **Guillem Cantallops Ramis**, ***Beowulf*** (http://bulma.net/beowulf/)

Creado el 24/10/2001 00:18 modificado el 24/10/2001 00:18

Esto no es un tutorial de ensamblador, ni de arquitectura del Intel x86. No necesitas ser un experto ni mucho menos, pero no voy a explicar lo que es un registro, o lo que es la pila. Si ya sabes algo acerca de todo eso, y lo que quieres es aprender las particularidades más importantes de Linux a base de ejemplos, sigue leyendo...

Tus primeros programas en ensamblador bajo Linux

Esto es más que nada una pequeña recopilación de lo que me hizo más falta a mi para empezar con esto, de las cosas que no estaban totalmente claras en la documentación existente, o de las cosas que estaban desperdigadas entre varios documentos. Pero siguen siendo muy recomendables los manuales, tutorials y how-to's que podeis encontrar al respecto en LinuxDoc[1]. He sacado muchas cosas de ahí, especialmente del "Linux Assembly HOWTO", aunque no he seguido la estructura de ninguno de esos documentos y he intentado siempre que ha sido posible no limitarme a traducir.

Básicamente esto no es completo, ni autocontenido (jeje, por eso recomiendo leer también otras cosas `O:-)`, y no me hago responsable de los daños cerebrales que te pueda provocar la lectura de este artículo `;-)`.

Ensambladores disponibles

Hay otras alternativas, pero las dos más importantes a mi entender son GAS y NASM.

NASM (Netwide Assembler)

Este acepta sintaxis **Intel** así que si estás acostumbrado a programar bajo **MS-DOS** por ejemplo con ensambladores de **Borland** o de (argh!) **Micro$oft**, quizás te costará menos acostumbrarte ya que no tendrás que aprender la sintaxis **AT&T** típica del mundo **UNIX**, que veremos más tarde.

El NASM tiene un desensamblador, el NDISASM, y en general no está nada mal pero tiene algunos problemas (habeis notado que no he puesto enlaces a su web?)

- El primer problema es que está muy orientado a **Intel x86** así que si quieres hacer algo para otra plataforma tienes un problema. Habiendo herramientas multiplataforma, no le veo mucho sentido.
- El segundo problema es que no se integra demasiado bién con las herramientas de GNU (no es que no se pueda, es que hay soluciones mucho mejores para eso) y ha tenido **problemas de licencia**. Creo que se distribuye bajo GNU GPL *y bajo otra licencia*, y si no recuerdo mal el año pasado incluso se quitó el proyecto de SourceForge[2] precisamente por discrepancias respecto a las licencias.
- Además el NASM usa sintaxis **Intel**, y a mi no me gusta aunque la he usado durante años. Si programas en sistemas tipo **UNIX** no hay nada como estar bién ambientado así que hay que usar la sintaxis **AT&T**, que además no te recuerda para nada a **Micro$oft**. Si, estas últimas opiniones son discutibles, pero el artículo lo escribo yo así que no usaré NASM ni sintaxis **Intel**... si alguien quiere hacer un contra-artículo no se lo impediré `];-)`.

GAS (GNU Assembler)

Este me gusta, mira por donde. Cada vez me parezco más a Richard Stallman[3] `X'-DDD`

Bueno, coñas aparte, este ensamblador es muy bueno. Es el que usa el **gcc**. Suele venir en el paquete `binutils` que típicamente traen todas las distribuciones de **Linux**. Entre otras cosas ese paquete contiene el ensamblador propiamente dicho (`as`), el enlazador (`ld`) y un montón de utilidades más. Lo típico es usar el `as`, luego el `ld` (a veces con un montón de parámetros), etc. Aquí usaré directamente el `gcc` porque es muy "listo" y (por lo menos para programas sencillos) lo hará todo automáticamente con un único comando como `gcc programa.s -o programa`. Lo típico es poner extensión `.s` a los ficheros que contengan código fuente en ensamblador.

En realidad el GAS soporta sintaxis **Intel** en las últimas versiones (se activa con la directiva `.intel_syntax`), pero yo soy muy cabezota. Antes de seguir con detalles escabrosos, contaré lo esencial de la sintaxis **AT&T**.

Sintaxis AT&T para bisoños

El GAS es un ensamblador pensado para funcionar esencialmente en sistemas **UNIX** de **32 bits** así que la sintaxis **AT&T** le sienta como un guante. Se parece bastante a la que se suele usar para CPUs de la familia **Motorola** 68000. Técnicamente sirve para lo mismo que la sintaxis **Intel**, pero se usa mucho en los sistemas **UNIX** y a mi personalmente me parece más cómoda y regular. Principales cosas a tener en cuenta (pongo la lista aunque no hay nada como ver unos ejemplos para entenderlo):

- Los nombres de los registros llevan el prefijo `%`. Es decir, son `%eax`, `%ebx`, etc.
- El orden de los operandos es origen primero y destino después mientras que la sintaxis **Intel** es al revés. Es decir, que lo que en sintaxis **Intel** era `mov eax,edx` en sintaxis **AT&T** es `mov %edx,%eax`.
- El tamaño de los operandos se puede especificar añadiendo un sufijo a la instrucción. Así `b` indica *byte* (8 bits), `w` indica *word* (16 bits), y `l` indica *long* (32 bits). Estrictamente deberia ponerse siempre el sufijo pero el GAS no es tan restrictivo: usa el tamaño adecuado si puede deducirse de los operandos, y si no toma por defecto 32 bits. La instrucción estrictamente correcta en el ejemplo anterior seria `movl %edx,%eax` que en este caso tendria el mismo efecto.
- Los operandos inmediatos llevan el prefijo `$`. Así, para sumar 5 al registro EAX escribiria `addl $5,%eax`. La notación para octal, hexadecimal, etc., va como en C.
- Si no ponemos ese prefijo, el operando se interpretará como una dirección de memoria.
- Las indirecciones se hacen con paréntesis tal que así `testb $0x80,17(%ebp)` (en este caso vemos el estado del bit más alto del byte situado en la posición 17 a partir de donde apunta el registro `%ebp`).

Dos enfoques diferentes

Ahora veremos como al programar en **Linux** uno no puede hacer las guarradas que se podian hacer en **MS-DOS**. Básicamente hay dos formas de programar (que pueden combinarse): podemos hacer directamente llamadas a sistema (*system calls*) o podemos utilizar las funciones de la GLIBC.

La primera opción es más "ligera" y parece más razonable ya que si usamos ensamblador seguramente es porque queremos optimizar algo. Sin embargo es fácil encontrarnos re-inventando partes de la GLIBC si elegimos esa opción (si las mejoramos en algo tiene un cierto pase, pero si no es bastante triste).

La segunda opción nos da mucho más juego porque las funciones de la GLIBC suelen ser más elaboradas y típicamente ahorran trabajo, pero son más "pesadas".

Un ejemplo bastante claro de todo esto es la diferencia entre `write()` y `printf()`.

Llamadas a Sistema

En este caso se utiliza la famosa interrupción `0x80`, al estilo de la antigua interrupción... a ver... cual era... la 0x21? del **MS-DOS**. Aunque estrictamente no es lo mismo, la verdad es que se parece mucho.

Metemos el número de *sytem call* en el registro EAX (puede encontrarse la lista completa de llamadas en

`/usr/src/linux/include/asm/unistd.h`), y los parámetros (hasta seis) en EBX, ECX, EDX, ESI, EDI, y EBP (por ese orden). Luego hacemos un `int 0x80`. El valor de retorno está en EAX, y si es negativo se ha producido un error (es el `-errno` de la GLIBC). No se toca la pila para nada.

Por si os hace ilusión saberlo, al iniciarse un proceso la pila suele contener el número de argumentos (`argc`) y luego la lista de punteros a los argumentos (`*argv`) seguida de las variables de entorno (pares nombre=valor, cada par termina en carácter nulo y la lista completa termina con un par nulo).

Veamos el archiconocido "hello world" usando llamadas a sistema... este ejemplo está en todas partes, pero sudé sangre para encontrar la manera de hacerlo con las librerias `;-)`.

```
.data   # lo dice el nombre ;-)

msg:    .string "Hello, world!\n"       # la cadena de siempre
        len = . - msg                   # y su longitud (no preguntes)

.text   # es típico que la zona de código se llame así :-?

        .global main    # decimos donde empezamos, el "entry point"

main:
        movl    $len, %edx      # tercer argumento, longitud de la cadena
        movl    $msg, %ecx      # segundo argumento, puntero a la cadena
        movl    $1, %ebx        # primer argumento, handler (1 = STDOUT)
        movl    $4, %eax        # num. de llamada a sistema (4 = write)
        int     $0x80           # llamada al kernel

        movl    $0, %ebx        # primer argumento, código de salida
        movl    $1, %eax        # num. de llamada a sistema (1 = exit)
        int     $0x80           # llamada al kernel
```

Tachán! No era tan dificil, eh? Bueno, aquí se ve como hacemos dos *system calls*, la primera a write() y la segunda a exit(). Sabiendo el número de la llamada que interesa y los argumentos que necesita (ver páginas del `man`) se puede hacer cualquier llamada de forma análoga.

No estaria de más comprobar errores y tal, pero bueno... En general está tirado, a que si? `;-)`

Llamadas a funciones de la GLIBC

Esto es un poco más complicado, pero no mucho. Básicamente también necesitamos una buena idea acerca de los argumentos que necesita la función, y para eso no hay nada mejor que el `man`.

Luego apilamos los argumentos empezando por el último (para que al desapilarlos la función empiece por el primero `;-)`) y llamamos a la correspondiente función por su nombre, haciendo algo así como `call printf`. Finalmente ajustamos el puntero de pila para "liberar" el espacio que ocupaban los argumentos.

Y siempre es así, la mar de simple... Veamos un ejemplo del "hello world" usando la GLIBC:

```
.data

msg:    .string "Hello, world!"         # la cadena de siempre
fmt:    .string "%s\n"                  # el formato para el printf()

.text

        .global main

main:
        pushl $msg                      # segundo argumento, la cadena
        pushl $fmt                      # primer argumento, el formato
        call printf                     # llamada a printf()
```

```
        addl $8, %esp                   # dejamos la pila como estaba

        ret                             # volvemos
```

Si, ya sé que el printf() podia hacerse con un solo argumento... era por darle un poco de emoción ;-)

Un ejemplo un poco más interesante

Este programa ya lo tenia hecho, los comentarios están en catalán pero creo que se entiende muy bién. Simplemente le pide cuatro valores al usuario (x, y, a, b) y calcula z=(x+y)*(a+b). Los cálculos son triviales, lo más interesante aquí es el uso de scanf() y printf()

```
### prova.s: exemple d'assemblador AT&T per Intel x86.
###
### Utilitza scanf() i printf() de la glibc per fer E/S.
### No s'ha optimitzat gens per fer-lo mes clar.
###
### Copyright (C) 2001, guillem(ensaimada)cantallops.net
### This is free software distributed under the terms of
### GNU GPL. Please read http://www.gnu.org/licenses/gpl.txt
###
### Imprimir amb:       'mpage -s4 -1 -o -bA4 < prova.s | lpr'.
### Editar amb:         'vim prova.s -c "set ts=4"'.
### Compilar amb:       'gcc prova.s -o prova'.
### Provar amb:         (x=1, y=2, a=-3, b=-4) -> (z=-21).



.data   ### dades

# formats per scanf() i printf(), no tocar

sfmt:   .string "%d"
pfmt:   .string "%d\n"

# variables, posar les que facin falta

x:              .int    0
y:              .int    0
z:              .int    0
a:              .int    0
b:              .int    0

# (variables auxiliars)

T1:             .int    0
T2:             .int    0
T3:             .int    0



.text   ### codi

                .global main    # indicam a on es l'inici, no tocar

main:   # inici, no tocar

# porcions de codi, canviar com calgui

                pushl $x                # get(x)
                pushl $sfmt
                call scanf
                addl $8, %esp

                pushl $y                # get(y)
                pushl $sfmt
```

```
                call scanf
                addl $8, %esp

                pushl $a                # get(a)
                pushl $sfmt
                call scanf
                addl $8, %esp

                pushl $b                # get(b)
                pushl $sfmt
                call scanf
                addl $8, %esp

                movl x, %eax            # T1:=x+y
                addl y, %eax
                movl %eax, T1

                movl a, %eax            # T2:=a+b
                addl b, %eax
                movl %eax, T2

                movl T1, %eax           # T3:=T1*T2
                mull T2
                movl %eax, T3

                movl T3, %eax           # z:=T3
                movl %eax, z

                pushl z                 # put(z)
                pushl $pfmt
                call printf
                addl $8, %esp

                jmp end                 # stop

end:    # acabament, no tocar
                ret
```

Ya sé que pueden hacerse muchas otras cosas en ensamblador, pero tus primeros programas **tienen** que ser algo así. Espero que **alguien** haya aprendido **algo** gracias a este artículo :-D

Lista de enlaces de este artículo:

1. http://linuxdoc.org/
2. http://sourceforge.net/
3. http://stallman.org/

E-mail del autor: beowulf _ARROBA_ bulma.net
Podrás encontrar este artículo e información adicional en: http://bulma.net/body.phtml?nIdNoticia=941