SISTEMAS OPERATIVOS – 95.03

DOCENTES:
MENDEZ, MARIANO
SIMÓ PIQUERES, ADEODATO

TRABAJO PRÁCTICO N° 0 : “LAB 0”

FECHA: 22/03/19

ALUMNO: SCHISCHLO, FRANCO DANIEL

PADRÓN: 100615

Ej: kern0-boot

Compilar *kern0* y lanzarlo en QEMU tal y como se ha indicado. Responder:

- ¿emite algún aviso el proceso de compilado o enlazado? Si lo hubo, indicar cómo usar la opción –`entry` de *ld(1)* para subsanarlo.

  Respuesta: El proceso de enlazado emite el siguiente aviso: “ld: warning: cannot find entry symbol _start; defaulting to 0000000000100000”, se soluciona agregando al enlazar, el tag “--entry ADDRESS” siendo ADDRESS el nombre de la función de entrada (main en mi caso).

- ¿cuánta CPU consume el proceso *qemu-system-i386* mientras ejecuta este kernel? ¿Qué está haciendo?

  Respuesta: Consume el 100% del CPU, está emulando el kernel.

Ej: kern0-quit

Respuesta:

Dump registros:

```
QEMU 2.8.1 monitor - type 'help' for more information
(qemu) info registers
EAX=2badb002 EBX=00009500 ECX=00100000 EDX=00000511
ESI=00000000 EDI=00102000 EBP=00000000 ESP=00006f0c
EIP=00100000 EFL=00000006 [-----P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     000caa68 00000027
IDT=     00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
```

```
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000
XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000
XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000
XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000
XMM07=00000000000000000000000000000000
(qemu) QEMU: Terminated
```

Ej: kern0-hlt

Leer la página de Wikipedia HLT (x86 instruction), y responder:

- una vez invocado `hlt` ¿cuándo se reanuda la ejecución?

  Respuesta: Se reanuda cuando es lanzada una nueva interrupción externa, que debe ser atendida por el procesador.

Usar el comando powertop para comprobar el consumo de recursos de ambas versiones del kernel. En particular, para cada versión, anotar:

- columna *Usage:* fragmento de tiempo usado por QEMU en cada segundo.

  Respuesta:

  Version SIN hlt: 983,6 ms/s

  Version CON hlt: 11,8 ms/s

- Columna *Events/s:* número de veces por segundo que QEMU reclama la atención de la CPU.

  Respuesta:

  Version SIN hlt: 219,1

  Version CON hlt: 137,9

Ej: kern0-gdb

Mostrar una sesión de GDB en la que se realicen los siguientes pasos:

- poner un breakpoint en la función *comienzo* (p.ej. `b comienzo`)
- continuar la ejecución hasta ese punto (`c`)
- mostrar el valor del *stack pointer* en ese momento (`p $esp`), así como del registro *%eax* en formato hexadecimal (`p/x $eax`).

Respuesta:

```
Reading symbols from kern0...done.
Remote debugging using 127.0.0.1:7508
0x0000fff0 in ?? ()
(gdb) b main
Breakpoint 1 at 0x100000: file kern0.c, line 3.
(gdb) c
Continuing.

Breakpoint 1, main () at kern0.c:3
3         asm("hlt");
(gdb) p $esp
$1 = (void *) 0x6f0c
(gdb) p/x $eax
$2 = 0x2badb002
(gdb)
```

Responder:

- ¿Por qué hace falta el modificador `/x` al imprimir *%eax*, y no así *%esp*?

Respuesta: Es necesario, ya que si quiero ver el valor mágico '`0x2BADB002`' debo pasar de base 10 (visión del registro sin modificador) a base 16. El contenido de ESP es indefinido, ya que no es modificado por el boot loader.

- ¿Qué significado tiene el valor que contiene *%eax*, y cómo llegó hasta ahí? (Revisar la documentación de Multiboot, en particular la sección Machine state.)

Respuesta: El valor mágico '`0x2BADB002`', le indica al sistema operativo que fue cargado por un "Multiboot-compliant boot loader", a diferencia de otros loaders. El mismo boot loader se encarga de cambiar el contenido de eax.

- el estándar Multiboot proporciona cierta informacion (Multiboot Information) que se puede consultar desde la función principal vía el registro *%ebx*. Desde el breakpoint en *comienzo* imprimir, con el comando `x/4xw`, los cuatro primeros valores enteros de dicha información, y

explicar qué significan. A continuación, usar y mostrar las distintas invocaciones de `x/...` `$ebx + …` necesarias para imprimir:

•el campo *flags* en formato binario

Respuesta:
(gdb) x/1tw $ebx
0x9500: 00000000000000000000001001001111

•la cantidad de memoria “baja” en formato decimal (en KiB)

Respuesta:
(gdb) x/2dw $ebx
0x9500: 591 639
El dato que corresponde es 639.

•la línea de comandos o “cadena de arranque” recibida por el kernel (al igual que en C, las expresiones de GDB permiten dereferenciar con el operador `*`)

Respuesta:
(gdb) x/5xw $ebx
0x9500: 0x0000024f 0x0000027f 0x0001fb80 0x8000ffff
0x9510: 0x00101000
El dato que corresponde es 0x00101000.
Luego hago:
(gdb) print (char*) 0x00101000
$3 = 0x101000 "kern0 "

•si está presente (¿cómo saberlo?), el nombre del gestor de arranque.

Respuesta:Es válido solo si el bit 9 en flags está seteado. En este caso es válido.
(gdb) x/17xw $ebx
0x9500: 0x0000024f 0x0000027f 0x0001fb80 0x8000ffff
0x9510: 0x00101000 0x00000000 0x00101000 0x00000000
0x9520: 0x00000000 0x00000000 0x00000000 0x00000090
0x9530: 0x00009000 0x00000000 0x00000000 0x00000000
0x9540: 0x00101007
El dato que corresponde es 0x00101007.
Luego hago:
(gdb) print (char*) 0x00101007
$2 = 0x101007 "qemu"
Lo cual es correcto.

•la cantidad de memoria “alta”, en MiB. (Hacerlo en dos pasos, primero un comando `x` y a continuación un comando `p`(*print*) con una expresión que use la variable de GDB `$__`.)

Respuesta:
(gdb) x/3dw $ebx
0x9500: 591 639 129920
(gdb) set $memMiB= 129920/1048,576
(gdb) print $memMiB
$1 = 123

Respuesta:

(gdb) x/4xw $ebx

0x9500: 0x0000024f 0x0000027f 0x0001fb80 0x8000ffff

0x0000024f: Corresponde a flags, se encarga de indicar la presencia y validad de los demas campos de la estructura de información. También es usado como indicador de versión.

0x0000027f: Corresponde a mem_lower, indica la cantidad de memoria en kilobytes de la memoria baja. Válido solo si el bit 0 de flags esta seteado.

0x0001fb80: Corresponde a mem_upper, indica la cantidad de memoria en kilobytes de la memoria alta. Válido solo si el bit 0 de flags esta seteado.

0x8000ffff: Corresponde a boot_device, indica de que dispositivo de disco, de la BIOS, el boot loader cargó la imagen del Sistema Operativo. Válido solo si el bit 1 de flags esta seteado.

Ej: make-flags

Recompilar el kernel usando `make`.

1.¿Qué compilador usa *make* por omisión? ¿Es o no gcc? Explicar cómo se podría forzar el uso de clang:

Respuesta: Usa el valor CC, que por default es el compilador de C. Sí,
por una sola vez, desde la línea de comandos; “make CC=clang”.

•para todas las compilaciones del proyecto.

Respuesta: Modificar el archivo Makefile y agregar en la primera linea: CC=gcc

2.Leer la sección sobre `-ffreestanding` en Guide to Bare Metal Programming with GCC(la segunda sección, sobre `-nostdlib`, se cubre en el lab x86); responder:

•¿Se pueden usar booleanos en modo “free standing”?

Respuesta: Sí, al estar incluido `<stdbool.h>`.

•¿Dónde se definen los tipos `uint8_t`, `int32_t`, etc.?

Respuesta: Se definen en `<stdint.h>`.

•Teniendo en cuenta los tamaños de `char`, `short`, `int`, `long` y `long long`, escribir un archivo *c99int.h* con las directivas `typedef` necesarias para definir tipos enteros propios de 8, 16, 32 y 64 bits, con signo y sin signo. Por ejemplo:

```
typedef short uint16_t;
typedef unsigned short uint16_t;
```

Hacerlo de manera que las mismas directivas puedan usarse en x86 y x86_64. Se recomienda poner especial atención en el signo de `char` y el tamaño de `long`.

Contenido de *c99int.h:*

```
typedef char int8_t;
typedef unsigned char uint8_t;
typedef short int16_t;
typedef unsigned short uint16_t;
typedef int int32_t;
typedef unsigned int uint32_t;
# if __WORDSIZE == 64
typedef long int        int64_t;
typedef unsigned long int    uint64_t;
# else
__extension__
typedef long long int        int64_t;
typedef unsigned long long int    uint64_t;
# endif
```

Ej: make-pattern

•¿Cómo funciona la regla que compila *boot.S* a *boot.o*?

Respuesta: Primero aparece que compilador se va a utilizar por default, en este caso es GCC, luego los flags que son necesarios. El tag -c indica que no se corra el linker. Y por último, $< indica el nombre del archivo del primer requisito, en este caso es único y es “boot.s”.

•¿Qué son las variables $@, $^ y $<?

Respuesta:

$@: Es el objetivo de la regla en la que aparece.

$^: Es la lista de requisitos de la regla en que aparece.

$<: Es el primer requisito de la regla.

•La regla *kern0* usa $^ y la regla con *%.S* usa $<. ¿Qué ocurriría si se intercambiaran estas dos variables entre ambas reglas?

Respuesta: En la regla de kern0 solo se enlazaría boot.o, y la regla con %.S no cambiaría en nada.

Ej: make-implicit

•¿Mediante qué regla se genera el archivo *kern0.o*?

Respuesta:

%.o: %.S

$(CC) $(CFLAGS) -c $<

•Eliminar la regla `%.o: %.S` y ejecutar `make clean kern0`:

•¿Se llega a generar el archivo *boot.o*?

Respuesta: Se genera pero muestra que la arquitectura de boot.o es incompatible con la salida de i386.

•¿Ocurre algún otro error? (Si no ocurre, mostrar la salida del comando `uname -m`).

Respuesta: x86_64

•¿Se puede subsanar el error sin re-introducir la regla eliminada?

Respuesta: Sí, teniendo un SO x86_32. (Probé remover -m32 de los flags, y cambiar el formato elf de ld, para que use `elf32_x86_64`, pero no funcionó).

Ej: kern0-vga

Explicar el código anterior, en particular:

•qué se imprime por pantalla al arrancar.

Respuesta: Se imprime el mensaje OK, con letras blancas y fondo verde.
qué representan cada uno de los valores enteros (incluyendo `0xb8000`).

Respuesta: Para un texto por pantalla, se debe escribir directamente en la memoria de video que arranca en `0xb8000.`

Cada letra se representa con 2 bytes. 1 byte se utiliza para almacenar el codigo ASCII, y el otro byte corresponde al atributo de la letra (color de esta y color del fondo). En detalle, los primeros 4 bits bajos, representan el color de la letra, y los siguientes 3 bits, el color del fondo.

En este caso:

79: 79 (Decimal) => 4f (Hexadecimal) => O (ASCII)

47: 47 (Decimal) => 00101111 (Binario) 1111 => F (Hexadecimal) color letra, 010 => 2 (Hexadecimal) color fondo.

75: 75 (Decimal) => 4b (Hexadecimal) => K (ASCII)

47: idem 47 anterior.

•por qué se usa el modificador *volatile* para el puntero al buffer.

Respuesta: Se usa para avisarle al compilador que no optimice esa variable, ya que podría modificar el orden de los datos.

Ahora, implementar una función más genérica para imprimir en el buffer VGA:

`static void vga_write(const char *s, int8_t linea, uint8_t color) { ... }`
donde se escribe la cadena en la línea indicada de la pantalla (si `linea` es menor que cero, se empieza a contar desde abajo).

Respuesta:

```
#include "c99int.h"
#define LONG_FILA 160 // 160 bytes = 80 char *2 (2 bytes per char)
#define MIN_FILA 0
#define MAX_FILA 24


static const volatile char * const VGABUF = (volatile char *)
0xb8000;

static void vga_write(const char *s, int8_t linea, uint8_t color)
{
     volatile char* buf = (volatile char*) VGABUF;

     if (linea > MAX_FILA) return;
     if (linea < MIN_FILA) linea = (MAX_FILA+1) + linea;

     uint8_t line = linea;

     buf = buf + line * LONG_FILA;

     while (*s != 0) {
        *buf++ = *s++;
        *buf++ = color;
    }
 }
```

Ej: kern0-const

Supongamos que se definiera `VGABUF` como una variable global *const* (de tal manera que no pueda ser modificada y que por tanto apunte siempre al comienzo del buffer):

```
static const volatile char *VGABUF = (volatile char *) 0xb8000;
```

1.Explicar los errores o avisos de compilación que ocurren al recompilar el código original con la nueva definición:

```
void comienzo() {
    volatile char *buf = VGABUF;
    *buf++ = 79;
    *buf++ = 47;
    // ...
}
```

¿Es adecuada la declaración de `VGABUF` propuesta? ¿Se puede agregar o quitar algo para subsanar el error?

Respuesta: Se genera un warning avisando que se esta descartando el calificador “const”, cuando se asigna a la variable buf. Es adecuada, ya que no permite la modificacion de VGABUF usando el mismo puntero, el warning se puede ignorar, casteando a volatile char*.

2.La declaración de `VGABUF` resultante del punto anterior ¿permite avanzar directamente la variable global? ¿Qué ocurre al añadir la siguiente línea a la función *comienzo*?

Respuesta: Sí, permite avanzar la dirección (en este caso 0xb8000 + 80).

```
void comienzo(void) {
    VGABUF += 80;  // ?!?!

    volatile char *buf = VGABUF;
    // ...
}
```

¿Se puede de alguna manera reintroducir el modificador *const* para que no compile esta nueva versión, pero siga compilando el código original? Justificar y explicar el cambio.

Respuesta: Se puede añadir el modificador const aquí:

static volatile char * **const** VGABUF = (volatile char *) 0xb8000, resultando en error al querer modificar la dirreción de VGABUF, pero no así el valor usando la indirección.

3.Finalmente, sobre la solución del punto anterior: ¿qué ocurre al ejecutar la siguiente versión?

```
void comienzo(void) {
    *(VGABUF + 120) += 88;

    volatile char *buf = VGABUF;
    // ...
}
```

¿Se podría cambiar el *tipo* de `VGABUF` para que no se permita el uso directo de `VGABUF`? (Pero siga compilando, *sin modificaciones*, el código original.) Justificar por qué el nuevo tipo produce error al usar `*VGABUF`, y por qué no se hace necesario cambiar el código original.

Respuesta:
Agregando el modificador const como se ha dicho en el punto (1) :
static **const** volatile char * const VGABUF = (volatile char *) 0xb8000.

Se produce error ya que se intenta modificar la dirección de VGABUF, y el valor al cual apunta VGABUF a través de si mismo (indirección). No es necesario modificar el codigo original porque se produce un warning que se resuelve casteando como dicho en el punto (1).

Ej: kern0-endian

1.Compilar el siguiente programa y justificar la salida que produce en la terminal:

```
#include <stdio.h>

int main(void) {
    unsigned int i = 0x00646c72;
    printf("H%x Wo%s\n", 57616, (char *) &i);
}
```

Respuesta: Se imprime “He110 World”, con %x el valor 57616 en hexadecimal (e110) y con %s el valor 0x00646c72 en formato string, que en memoria está de la siguiente forma; 0x726c6400 al ser little-endian.

A continuación, reescribir el código para una arquitectura big-endian, de manera que imprima exactamente lo mismo.

Respuesta:

```
#include <stdio.h>

int main(void) {
    unsigned int i = 0x00726c6400 ;
    printf("H%x Wo%s\n", 57616, (char *) &i);
}
```

2.Cambiar el código de `comienzo()` para imprimir el mismo mensaje original con una sola asignación, “abusando” de un puntero a entero:

```
static ... *VGABUF = ...

void comienzo(void) {
    volatile unsigned *p = VGABUF;
    *p = 0x...

    while (1)
        asm("hlt");
}
```

Ayuda: convertir los valores 47, 75 y 79 de base 10 a base 16, y componer directamente una constante entera en hexadecimal.

Respuesta: `*p = 0x2F4B2F4F`

3.Usar un puntero a `uint64_t` para imprimir en la segunda línea de la pantalla la palabra `HOLA`, en negro sobre amarillo.

Realizar la inicialización de “p” de dos maneras distintas:

```
// Versión 1
volatile uint64_t *p = VGABUF + 160;
*p = 0x...

// Versión 2
volatile uint64_t *p = VGABUF;
p += 160;
*p = 0x...
```

Justificar las diferencias de comportamiento entre ambas versiones.

Respuesta: `*p=0xE041E04cE04fE048`, y la versión que corresponde es la primera.
La diferencia esta en la forma en que me desplazo a la fila para escribir en memoria.

Version 1: VGABUF + 160 es asignado a p (lo cual es correcto ya que debo desplazarme 80 caracteres en la pantalla, y por cada caracter se ocupa 2 bytes, luego 80*2=160 bytes por linea.
Version 2: p += 160, me avanza el puntero de p 160 veces, pero el tipo de p al que apunta es uint64_t, esto es 8 bytes, luego 160*8= 1280 bytes en total. Extra: Lo imprime en la linea 8 (1280/160=8).

Ej: kern0-objdump

Incluir en la entrega la versión final del archivo *Makefile* (incluyendo los cambios propuestos en esta tarea), y un archivo *kern0.c* con:

- la declaración correcta de la variable `VGABUF`
- la función **estática** `vga_write()`
- la función `comienzo()`, ahora con sendas invocaciones a `vga_write()` para imprimir, primero, el mensaje original en la línea 0; a continuación, `HOLA` en la siguiente línea, en negro sobre amarillo.

Sobre este código:

1.Obtener el código máquina del binario final usando *objdump:*

```
$ make
$ objdump -d kern0
```

Típicamente, se guarda la salida en un archivo con extensión *.asm* para tenerlo siempre a mano. Se puede usar la funcionalidad de redirección del intérprete de comandos:

```
$ objdump -d kern0 >kern0.asm
```

Añadir al archivo *Makefile* una invocación de manera que se genere *kern0.asm* automáticamente tras la fase de enlazado, y se borre como parte de la regla *clean*. Usar la variable `$@` según corresponda.

Respuesta:

```
1.
2.kern0:     file format elf32-i386
3.
4.
5.Disassembly of section .text:
6.
7.00100000 <multiboot>:
8.  100000: 02 b0 ad 1b 00 00        add    0x1bad(%eax),%dh
9.  100006: 00 00                    add    %al,(%eax)
10.  100008:     fe 4f 52                 decb   0x52(%edi)
11.  10000b:     e4                       .byte 0xe4
12.
13.0010000c <vga_write>:
14.  10000c:     80 fa 18                 cmp    $0x18,%dl
15.  10000f:     7f 36                    jg     100047
<vga_write+0x3b>
16.  100011:     56                       push   %esi
17.  100012:     53                       push   %ebx
18.  100013:     89 d3                    mov    %edx,%ebx
```

```
19.  100015:    8d 72 19                lea    0x19(%edx),%esi
20.  100018:    84 d2                   test   %dl,%dl
21.  10001a:    0f 48 de                cmovs  %esi,%ebx
22.  10001d:    0f b6 d3                movzbl %bl,%edx
23.  100020:    69 d2 a0 00 00 00       imul   $0xa0,%edx,%edx
24.  100026:    81 c2 00 80 0b 00       add    $0xb8000,%edx
25.  10002c:    0f b6 18                movzbl (%eax),%ebx
26.  10002f:    84 db                   test   %bl,%bl
27.  100031:    74 12                   je     100045
<vga_write+0x39>
28.  100033:    83 c0 01                add    $0x1,%eax
29.  100036:    88 1a                   mov    %bl,(%edx)
30.  100038:    88 4a 01                mov    %cl,0x1(%edx)
31.  10003b:    0f b6 18                movzbl (%eax),%ebx
32.  10003e:    8d 52 02                lea    0x2(%edx),%edx
33.  100041:    84 db                   test   %bl,%bl
34.  100043:    75 ee                   jne    100033
<vga_write+0x27>
35.  100045:    5b                      pop    %ebx
36.  100046:    5e                      pop    %esi
37.  100047:    c3                      ret
38.
39.00100048 <main>:
40.  100048:    b9 2f 00 00 00          mov    $0x2f,%ecx
41.  10004d:    ba 00 00 00 00          mov    $0x0,%edx
42.  100052:    b8 73 00 10 00          mov    $0x100073,%eax
43.  100057:    e8 b0 ff ff ff          call   10000c
<vga_write>
44.  10005c:    b9 e0 00 00 00          mov    $0xe0,%ecx
45.  100061:    ba 01 00 00 00          mov    $0x1,%edx
46.  100066:    b8 76 00 10 00          mov    $0x100076,%eax
47.  10006b:    e8 9c ff ff ff          call   10000c
<vga_write>
48.  100070:    f4                      hlt
49.  100071:    eb fd                   jmp    100070
<main+0x28>
```

¿En qué cambia el código generado si se recompila con la opción `-fno-inline` de GCC?

Respuesta: Reemplaza los llamados a función por la función misma. No altera el source file, ya que se produce durante la compilación (es parecido al momento en el que el preprocesador expande los macros, en donde sí se modifica el source file). Se pueden ver los cambios en el .asm generado.

Sustituir la opción `-d` de *objdump* por `-S`, y explicar las diferencias en el resultado.

Respuesta: -S produce lo mismo que -d pero mezclado con código del archivo fuente.

De la salida de *objdump -S* sobre el binario compilado con `-fno-inline`, incluir la sección correspondiente a la función `vga_write()` y explicar cada instrucción de assembler en relación al código C original.

Respuesta:

```
kern0:     file format elf32-i386


Disassembly of section .text:

0010000c <vga_write>:
static const volatile char * const VGABUF = (volatile char *)
0xb8000;

static void vga_write(const char *s, int8_t linea, uint8_t color)
{
     volatile char* buf = (volatile char*) VGABUF;

     if (linea > MAX_FILA) return;
  10000c: 80 fa 18                cmp    $0x18,%dl
  10000f: 7f 36                   jg     100047 <vga_write+0x3b>
```

***)Compara linea con el valor hexadecimal 0x18 (24 decimal), si es mayor, hace retorno.**

```
#define MAX_FILA 24


static const volatile char * const VGABUF = (volatile char *)
0xb8000;

static void vga_write(const char *s, int8_t linea, uint8_t color)
{
  100011: 56                      push   %esi
  100012: 53                      push   %ebx
  100013: 89 d3                   mov    %edx,%ebx
     volatile char* buf = (volatile char*) VGABUF;

     if (linea > MAX_FILA) return;
     if (linea < MIN_FILA) linea = (MAX_FILA+1) + linea;
  100015: 8d 72 19                lea    0x19(%edx),%esi
  100018: 84 d2                   test   %dl,%dl
  10001a: 0f 48 de                cmovs  %esi,%ebx
```

***) Se hace la suma de MAXFILA+1 (0x19) con linea, se hace un test, y si el SF es 1, es decir es negativo, se hace un salto.**

```
     uint8_t line = linea;

     buf = buf + line * LONG_FILA;
  10001d: 0f b6 d3                movzbl %bl,%edx
  100020: 69 d2 a0 00 00 00       imul   $0xa0,%edx,%edx
  100026: 81 c2 00 80 0b 00       add    $0xb8000,%edx
```

***) imul hace la multiplicación de 160 * edx y lo deja en edx (0xa0 es 160 en hexa)**

***) Le suma el contenido de buf inicial (0xb8000) y lo almacena nuevamente en edx.**

```
    while (*s != 0) {
 10002c: 0f b6 18                movzbl (%eax),%ebx
 10002f: 84 db                   test   %bl,%bl
 100031: 74 12                   je     100045 <vga_write+0x39>
```

***) Se mueve el contenido de la dirección de memoria de eax (string) a ebx, expandiendose a long y rellenando con ceros a izquierda.**
***) Test hace AND bit a bit del contenido de la parte baja de ebx, con él mismo. El resultado es 0, sí y solo sí, ambos son 0. Setea el ZF si el resultado es 0.**
***) Salta a la direccion 100045 si el ZF es 1. Es decir, si el string llegó a su fin.**

```
        *buf++ = *s++;
 100033: 83 c0 01                add    $0x1,%eax
 100036: 88 1a                   mov    %bl,(%edx)
```

***) En eax esta cargada la dirección del string, le suma uno (s++).**
***) Mueve la parte baja de ebx (previamente estaba cargado el contenido del string en la primera posición de memoria) a la dirección contenida en edx (buf)**

```
        *buf++ = color;
 100038: 88 4a 01                mov    %cl,0x1(%edx)
```

***) Mueve el contenido de la parte baja de ecx (int color) a la dirección contenida en edx+1 ([edx+1])**

```
    uint8_t line = linea;

    buf = buf + line * LONG_FILA;

    while (*s != 0) {
 10003b: 0f b6 18                movzbl (%eax),%ebx
        *buf++ = *s++;
        *buf++ = color;
 10003e: 8d 52 02                lea    0x2(%edx),%edx

    uint8_t line = linea;

    buf = buf + line * LONG_FILA;

    while (*s != 0) {
 100041: 84 db                   test   %bl,%bl
 100043: 75 ee                   jne    100033 <vga_write+0x27>
        *buf++ = *s++;
        *buf++ = color;
   }
}
```

```
  100045: 5b                    pop    %ebx
  100046: 5e                    pop    %esi
  100047: c3                    ret
```

Archivo kern0.c:

```
#include "c99int.h"
#define LONG_FILA 160 // 160 bytes = 80 char *2 (2 bytes per char)
#define MIN_FILA 0
#define MAX_FILA 24


static const volatile char * const VGABUF = (volatile char *)
0xb8000;

static void vga_write(const char *s, int8_t linea, uint8_t color)
{
     volatile char* buf = (volatile char*) VGABUF;

     if (linea > MAX_FILA) return;
     if (linea < MIN_FILA) linea = (MAX_FILA+1) + linea;

     uint8_t line = linea;

     buf = buf + line * LONG_FILA;

     while (*s != 0) {
        *buf++ = *s++;
        *buf++ = color;
    }
 }

void main(void) {

    vga_write("OK",0,47);
    vga_write("HOLA",1,224);

    while (1) asm("hlt");
}
```

Archivo Makefile:

```
CFLAGS := -g -m32 -O1 -fno-inline
SRCS := $(wildcard *.c)  # usar wildcard *.c
OBJS := $(patsubst %.c,%.o,$(SRCS))  # usar patsubst sobre SRCS

QEMU := qemu-system-i386 -serial mon:stdio
KERN := kern0
BOOT := -kernel $(KERN)

qemu: $(KERN)
     $(QEMU) $(BOOT)

qemu-gdb: $(KERN)
     $(QEMU) -kernel kern0 -S -gdb tcp:127.0.0.1:7508 $(BOOT)

gdb:
     gdb -q -s kern0 -n -ex 'target remote 127.0.0.1:7508'

kern0: boot.o $(OBJS)
     ld -m elf_i386 -Ttext 0x100000 --entry main $^ -o $@
     objdump -S kern0 >kern0.asm
     # Verificar imagen Multiboot v1.
     grub-file --is-x86-multiboot $@

%.o: %.S
     $(CC) $(CFLAGS) -c $<

clean:
     rm -f kern0 *.o *.asm core

.PHONY: clean qemu qemu-gdb gdb
```