# Kern0
# Sistemas Operativos
# (75.08 / 95.03)

**Apellidos: Escobar Benítez**

**Nombres: María Soledad**

**Padrón: 97877**

# Resolución

## 1. Compilar un kernel y lanzarlo en QEMU

### 1.1. Ej. kern0-boot:

1.1.1. El proceso de enlazado emite el siguiente aviso:
*"ld: aviso: no se puede encontrar el símbolo de entrada _start; se usa por defecto 0000000000100000".*
Para subsanarlo agrego *-entry=comienzo* a la linea que utilizo para el enlazado:
*"ld -m elf_i386 -Ttext 0x100000 --entry=comienzo kern0.o boot.o -o kern0".*
De esta manera le indico a *ld* donde se encuentra el main, el cual se llama comienzo en este ejemplo, así le indico por donde debe comenzar el programa.

1.1.2. El proceso qemu-system-i386 consume 100,3% de CPU mientras está siendo ejecutado.
A simple vista no hace nada, pues el programa es un ciclo infinito.

### 1.2. Ej. kern0-quit:

1.2.1. Resultado de la ejecución del comando *info registers* en el monitro de QEMU:

```
(qemu) info registers
EAX=2badb002 EBX=00009500 ECX=00100000 EDX=00000511
ESI=00000000 EDI=00102000 EBP=00000000 ESP=00006f08
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)

**1.3. Ej. kern0-hlt:**

1.3.1. La ejecución se reanuda cuando se necesita que el procesador realice alguna acción.

1.3.2. Resultados de ejecución del comando *powertop*:

1. Versión con *asm(htl):*
   *Uso: 1,7 ms/s.*
   *Eventos: 68,5.*
2. Versión sin *asm(hlt)*:
   Uso: 494,3 ms/s.
   Eventos: 56,9.

**1.4. Ej. kern0-gdb:**

1.4.1. Sesión de Gdb:

```
$ gdb -q -s kern0 -n -ex 'target remote 127.0.0.1:7508'
Leyendo símbolos desde kern0...hecho.
Remote debugging using 127.0.0.1:7508
aviso: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000fff0 in ?? ()
(gdb) b comienzo
Punto de interrupción 1 at 0x100000: file kern0.c, line 3.
(gdb) c
Continuando.

Breakpoint 1, comienzo () at kern0.c:3
3                   asm("hlt");
(gdb) p $esp
$1 = (void *) 0x6f08
(gdb) p/x $eax
$2 = 0x2badb002
(gdb)
```

1.4.2. Se necesita el modificador /x porque lo que se muestra es el valor que se encuentra dentro del registro %eax, lo que hace el /x es considerarlo como un valor entero e imprimirlo en hexadecimal. No se lo utiliza al mostrar el valor de %esp, porque lo que se ve es el puntero al stack.

1.4.3. El valor que contiene %eax es '0x2BADB002' y está ahí para indicar al sistema operativo que fue cargado por un cargador de arranque compatible con Multiboot.

1.4.4. %ebx contiene la dirección física de la estructura de información de datos de Multiboot proporcionada por el boot loader.

1. Resultado de ejecución del comando en gdb:

   ```
   (gdb) x/4xw $ebx
   0x9500:     0x0000024f   0x0000027f  0x0001fb80  0x8000ffff
   ```

   Lo que se muestra son los campos flags, mem_lower, mem_upper y boot_device. Ya que estoy examinando lo que hay en las primeras 4 palabras, o sea, en los primeros 16 bytes.
2. Campo *flags* en formato binario:

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

3. Cantidad de memoria "baja" en formato decimal, en KiB:
   *(gdb) x/1dw $ebx + 4*
   *0x9504: 639*
   Según la documentación de Multiboot la cantidad de memoria baja se guarda en kilobytes por lo que para pasar a KiB debo realizar una conversión:
   *(gdb) p/d (*$_*1000)/1024*
   *$22 = 624*
   *Dado que 1KB = 1000Bytes y 1KiB = 1024Bytes lo que hago es pasar primero el valor a Bytes, multiplicando por 1000, y luego pasarlo a KiB, dividiendo por 1024.*
4. Línea de comandos o "cadena de arranque" recibida por el kernel:
   *(gdb) x/1xw $ebx + 16*
   *0x9510: 0x00101000*
5. Obtengo el nombre del gestor de arranque:
   Puedo saber si está presente el nombre del gestor de arranque viendo si el bit 9 de flags está seteado, como vale 1 entonces puedo ver el nombre del gestor de arranque.
   Dirección física del nombre del gestor de arranque:
   *(gdb) x/1xw $ebx + 64*
   *0x9540: 0x00101007*
   Nombre del gestor de arranque:
   *(gdb) x/s 0x00101007*
   *0x101007: "qemu"*
6. Cantidad de memoria "alta" en MiB:
   Obtengo la cantidad de memoria alta, que está en KB por default:
   *(gdb) x/1dw $ebx + 8*
   *0x9508: 129920*
   Luego utilizo la variable $_ para obtener el valor de memoria alta, que quedó guardado allí luego de utilizar x/, lo multiplico por 1000 para hacer un pasaje a Bytes y lo divido por 1024^² Bytes para obtener su valor en MiB:
   *(gdb) p/d (*$_*1000)/(1024 * 1024)*
   *$25 = 123*

## 2. Makefile y flags de compilación

### 2.1. make-flags:

2.1.1. El compilador por omisión para C que utiliza Make es *CC*. No es GCC. Cc es el compilador de Unix y Gcc es el compilador del sistema operativo GNU.

2.1.2. Sección sobre –ffreestanding:

1. Si se pueden utilizar booleanos pues el archivo de encabezado <stdbool.h> está incluido entre los archivos de encabezado estándar disponible.
2. Los tipos mencionados se definen en el archivo de cabecera <stdint.h>.

### 2.2. make-pattern:

2.2.1. La regla que compila boot.S a boot.o llama al compilador CC y le pasa los flags definidos en CFLAGS.

2.2.2. Significados:

1. "$@": Representa el nombre del fichero de salida, es decir, el ejecutable.
2. "$^": Representa el nombre de los archivos .o requeridos para crear el ejecutable.
3. "$<": Representa el nombre de los ficheros de entrada, es decir, los archivos .c y .S.

2.2.3. Si cambio la regla kern0 $^ por $< y la regla %.S $< por $^ lo que sucede es ...

**2.3. Make-implicit:**

2.3.1.

2.3.2. Elimino la regla %.o: %.S:

1. El archivo kern0.o se llega a generar.
2. El error que ocurre es el siguiente:

```
$ make clean kern0
rm -f kern0 *.o core
cc    -c -o boot.o boot.S
cc -g -std=c99 -Wall -Wextra -Wpedantic -m32 -O1 -fasm -
ffreestanding   -c -o kern0.o kern0.c
ld -m elf_i386 -Ttext 0x100000 --entry=comienzo boot.o kern0.o -o
kern0
ld: la arquitectura i386:x86-64 del fichero de entrada `boot.o'
es incompatible con la salida i386
makefile:8: recipe for target 'kern0' failed
make: *** [kern0] Error 1
```

Sucede porque al quitar la regla %.o: %.S ya no se le indica al compilador que debe compilar para x86 de 32 bits, con la opción -m32.

2.1. Ejecución del comando uname -m:

```
$ uname -m
x86_64
```

3.

**2.4. make-wildcard:**

2.4.1. Uso de wildcard y patsubst:

```
SRCS := $(wildcard *.c)
OBJS := $(patsubst %.c, %.o, $(SRCS))
```

**3. El buffer VGA**

**3.1. kern0-vga:**

3.1.1. Por pantalla se imprimen las siglas Ok en el borde superior izquierdo, con letras blancas en un fondo verde.

3.1.2. Significado de los valores enteros:

1. 0xb8000: es donde reside la memoria de pantalla de video de los monitores a color.
2. El valor 79 corresponde a la letra O en codigo ascii en decimal.
3. El valor 75 corresponde a la letra k en codigo ascii en decimal.
4. El valor 47 corresponde al color que se asigna como fondo a cada letra, 47 en decimal corresponde a 2f en hexadecimal que es la

suma de 20 (para el color verde) + 0f (para el color de letra blanco).

3.1.3. El modificador *volatile* le indica al compilador que no debe realizar optimizaciones a esa variable. A veces el compilador asume que el valor de una variable permanece constante si el programador no la modifica, por lo que si la variable es modificada desde otro lugar el compilador puede ignorarlo y no actualizar la variable.

3.2. kern0-const:

3.2.1. Warning del compilador:

*kern0.c: In function 'comienzo':*
*kern0.c:5:26: warning: initialization discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers] volatile char *buf = VGABUF;*

Lo que indica este aviso es que el calificador 'const' indica que el char al que apunta el puntero será constante, de solo lectura, y no podrá modificarse ya que se almacena en el data segment.
La declaración no resulta adecuada por tratarse de un puntero. Aunque el valor del puntero no se modifica, dado que lo que modificamos es el valor almacenado en la variable buf, se puede subsanar este warning quitando el calificador const de la declaración de la variable.

3.2.2. La variable globlal no avanza, lo que avanza es la variable buf. El programa se ejecuta correctamente.
Luego de quitar el calificador const, al añadir la nueva linea de código no sucede absolutamente nada al compilar, se compila correctamente, al ejecutar el programa la palabra 'OK' se desplazó en la posición x, es decir, sobre la misma linea a la derecha.
Aun utilizando el calificador const el programa compila, aunque con un warning, y se ejecuta correctamente, la unica manera de introducir el calificador const para que no compile sería utilizándolo de la siguiente manera:
*static volatile char *const VGABUF = (volatile char *) 0xb8000;*
obteniendo el siguiente error:
*kern0.c: In function 'comienzo':*
*kern0.c:4:9: error: assignment of read-only variable 'VGABUF'*
*VGABUF += 80;*
*^~*
Esto sucede porque usar char *const VGABUF indica un puntero constante un char, por lo que no se puede modificarse el puntero. Sin embargo con la definición const volatile char *VGABUF indica que es un puntero a un char que permanece constante, pero el puntero en sí puede modificarse.

3.2.3. Agrego la linea *(VGABUF + 120) += 88; al código:
El programa compila correctamente sin ningún warning o error.
Puedo modificar el tipo de VGABUF de la siguiente, manera para que *el código no compile:*
*static const volatile char VGABUF = (volatile char *) 0xb8000;*
De esta manera se genera un error al intentar modificar *VGABUF pues estoy indicando que el char al que apunta el puntero es constante por lo que se genera el siguiente error:

```
kern0.c: In function 'comienzo':
kern0.c:4:19: error: assignment of read-only location '*753784'
   *(VGABUF + 120) += 88;
                   ^~
kern0.c:6:26: warning: initialization discards 'const' qualifier
from pointer target type [-Wdiscarded-qualifiers]
     volatile char *buf = VGABUF;
                          ^~~~~~
```

Esto no genera errores en el código original porque en el mismo no se modifica la variable VGABUF.

### 3.3. kern0-endian:

3.3.1. La salida que produce la terminal es la frase 'He110 World'.

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

La palabra He110 se compone con el carácter 'H' + el valor 57616 que corresponde a los caracteres 'e110', el valor 57616 está en decimal y corresponde al número e110 en hexadecimal.
La variable i contiene los caracteres 'rld' que completan la palabra World. Pues como los datos se almacenan en formato little endian el orden de los caracteres es 72 = r, 6c = l, 64 = d.
El código para big-endian sería el siguiente:

```
int main(void) {
    unsigned int i = 0x726c6400;
    printf("H%x Wo%s\n", 57616, (char *) &i);
}
```

El valor de i lo modifico para que se reproduzca la concatenación de caracteres 'rld' ya que se almacenan en el orden r = 72, l = 6c y d = 64.
El valor decimal 57616 no se modifica porque al no ser una variable que se almacena en memoria no se ve modificado mas que por el pasaje de decimal a hexadecimal.

3.3.2. Cambio el código de comienzo():

```
static volatile unsigned *VGABUF = (volatile unsigned *) 0xb8000;

void comienzo(void) {
    volatile unsigned *p = VGABUF;
    *p = 0x2f4b2f4f;

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

3.3.3. Imprimir en la segunda linea de la pantalla la palabra HOLA en negro sobre amarillo:
No hay diferencia de comportamiento durante la ejecución del programa. La diferencia entre ambas versiones reside en que en la versión 1 el puntero *p apunta directamente a la segunda linea de la pantalla, en cambio en la versión 2 el puntero *p apunta inicialmente a la primer linea de la pantalla y luego al hacer p +=

160 se mueve el puntero a la segunda linea, luego en ambas versiones se le asigna el valor que corresponde a la palabra en el color pedido, haciendo E0 = E0 (para el fondo amarillo) + 00 (para las letras negras) e intercalando las letras para formar la palabra HOLA = 484f4c41, además tomando en cuenta la disposición del orden en que se almacenan los numeros en little-endian.

3.4. **kern0-objdump:**

3.4.1.

3.4.2. Lo que cambia en el código generado si se recompila con la opción -fno-inline de GCC es que desensambla la función comienzo() y vga_write() por separado.

**3.4.3. Sustitución de la opción -d de objdump por -S:**

```
$ objdump -d kern0

kern0:     formato del fichero elf32-i386

Desensamblado de la sección .text:

00100000 <multiboot>:
  100000:   02 b0 ad 1b 00 00       add    0x1bad(%eax),%dh
  100006:   00 00                   add    %al,(%eax)
  100008:   fe 4f 52                decb   0x52(%edi)
  10000b:   e4                      .byte 0xe4

0010000c <comienzo>:
  10000c:   eb fe                   jmp    10000c <comienzo>
$ objdump -S kern0

kern0:     formato del fichero elf32-i386

Desensamblado de la sección .text:

00100000 <multiboot>:
  100000:   02 b0 ad 1b 00 00       add    0x1bad(%eax),%dh
  100006:   00 00                   add    %al,(%eax)
  100008:   fe 4f 52                decb   0x52(%edi)
  10000b:   e4                      .byte 0xe4

0010000c <comienzo>:
comienzo:
      jmp comienzo
  10000c:   eb fe                   jmp    10000c <comienzo>
```

La opción -d desensambla las instrucciones de máquina desde el archivo objeto.

La opción -S muestra además de las instrucciones de máquina desensambladas, el código fuente.

3.4.4. Salida de objdump -S sobre el binario compilado con -fno-inline:

```
static void vga_write(const char *s, int8_t linea, uint8_t color) {
```

```
   0: 53                    push   %ebx
```
1. *Llamado a la función vga_write:*
```
   1: e8 fc ff ff ff        call   2 <vga_write+0x2>
```
2. *Se realiza el calculo de la posición en la pantalla en la que se va a imprimir la palabra:*
```
   6: 81 c3 02 00 00 00     add    $0x2,%ebx
      int pos = (linea * 160);
```
3. *Guarda el resultado de la posición en pantalla en el stack:*
```
   c: 0f be d2              movsbl %dl,%edx
```
4. *Carga el valor del stack y se incrementa la dirección a la que apunta el puntero VGABUF.*
```
   f: 8d 14 92              lea    (%edx,%edx,4),%edx
  12: c1 e2 05              shl    $0x5,%edx
```
5. *Sumando al puntero al que apunta vgabuf el valor calculado y que fue guardado en el stack:*
```
      VGABUF += pos;
```
6. *Asugna a la variable buf el valor actualizado de VGABUF:*
```
  15: 03 93 00 00 00 00     add    0x0(%ebx),%edx
  1b: 89 93 00 00 00 00     mov    %edx,0x0(%ebx)
      volatile char *buf = VGABUF;

      while( *s != 0 ) {
```
7. *Suma el valor de cada carácter al valor apuntado por buf. Avanza tanto en buf como en s. A buf le asigna el valor luego de haber avanzado una posición sobre s.*
```
  21: 0f b6 18              movzbl (%eax),%ebx
  24: 84 db                 test   %bl,%bl
  26: 74 12                 je     3a <vga_write+0x3a>
        *buf++ = *s++;
```
8. *Cambia el color sumando el valor de color al buf:*
```
  28: 83 c0 01              add    $0x1,%eax
  2b: 88 1a                 mov    %bl,(%edx)
        *buf++ = color;
  2d: 88 4a 01              mov    %cl,0x1(%edx)
      while( *s != 0 ) {
  30: 0f b6 18              movzbl (%eax),%ebx
        *buf++ = color;
  33: 8d 52 02              lea    0x2(%edx),%edx
      while( *s != 0 ) {
  36: 84 db                 test   %bl,%bl
  38: 75 ee                 jne    28 <vga_write+0x28>
    }
  }
```

**Código makefile:**

```
CFLAGS := -g -m32 -O1

CFLAGS := -g -std=c99 -Wall -Wextra -Wpedantic
```

```
CFLAGS += -m32 -O1 -fasm -ffreestanding

SRCS := $(wildcard *.c)
OBJS := $(patsubst %.c, %.o, $(SRCS))

QEMU := qemu-system-i386 -serial mon:stdio
KERN := kern0

BOOT := -kernel $(KERN)

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

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

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'

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

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

**Codigo kerno.c completo:**

```
#include <stdint.h>
static volatile char *VGABUF = (volatile char *) 0xb8000;

static void vga_write(const char *s, int8_t linea, uint8_t color) {
      int pos = (linea * 160);
      VGABUF += pos;
      volatile char *buf = VGABUF;

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

```
void comienzo(void) {
    const char *s1 = "OK";
    uint8_t color1 = 47;

    vga_write(s1, 0, color1);

    const char *s2 = "HOLA";
    uint8_t color2 = 224;

    vga_write(s2, 1, color2);

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