

Kern2

Un kernel con reloj y tres tareas

75.08 / 95.03

Sistemas Operativos

Gabriela Azcona - 95363
Jazmín Ferreiro - 97266

Repositorio: <https://github.com/jazminferreiro/kernel>
Entrega: 22 de junio

Creación de stacks en el kernel

Explicar: ¿qué significa “estar alineado”?

Si las direcciones de memoria virtuales son un múltiplo de las direcciones de memoria física dentro del hardware entonces se dice que la memoria está alineada. Esto no siempre sucede naturalmente en un sistema operativo para lo cual es necesario incluir un padding al principio o al final de la estructura de datos.

Por ejemplo si creamos un stack alineado en 16 bytes cada lectura será de una media palabra, el principio de cada variable buscará en una dirección que sea un múltiplo de 2 bytes. Si por ejemplo quisiéramos guardar una variable que ocupara menos tamaño, como un char (8 bits) y a continuación guardar otra variable que por ejemplo ocupase 2 bytes, sin una alineación quedarían concatenadas y cuando se fuera a buscar el valor de la segunda variable se tendrían que hacer dos lecturas. Con la alineación agregamos un espacio vacío luego de los 8 bytes que completen 16 bytes para continuar guardando las siguientes variables en el bloque de 2 bytes siguiente. En este caso se deberá hacer una sola lectura para obtener la variable. Se desperdicia memoria para mejorar el rendimiento.

Mostrar la sintaxis de C/GCC para alinear a 32 bits el arreglo kstack anterior.
para alinear a 32 bits (4 bytes) se debe usar la sintaxis `.align 4`

¿A qué valor se está inicializando kstack? ¿Varía entre la versión C y la versión ASM? (Leer la documentación de as sobre la directiva [.space](#).)

Si reservamos memoria para el stack usando la sintaxis de C “`unsigned char kstack[8192];`” la memoria no se está inicializando con nada. Tiene basura. En cambio utilizando la sintaxis de assembler “`.space size, fill`” se reserva el espacio de memoria de tamaño definido por el valor de size y lo llena con el valor de fill. si la coma y la variable fill son omitidas se toma por default cero. En este caso se omiten entonces se estaría inicializando el stack en cero.

Explicar la diferencia entre las directivas [.align](#) y [.p2align](#) de as, y mostrar cómo alinear el stack del kernel a 4 KiB usando cada una de ellas.

`.align` define el alineamiento del stack en bytes, mientras que `p2align` se define con el valor al cual se debe elevar 2 para definir el alineamiento, en este caso debería ser 12 para alinear 4 KB, $2^{12} = 4096$.

kern2-cmdline

Mostrar cómo implementar la misma concatenación, de manera correcta, usando strncat(3).

```
#include "decls.h"
#include "multiboot.h"
#include <string.h>

void kmain(const multiboot_info_t *mbi) {
    vga_write("kern2 loading.....", 8, 0x70);

    if (mbi->flags) {
        char buf[256] = "cmdline: ";
        char *cmdline = (void *) mbi->cmdline;
        // Aquí usar strlcat() para concatenar cmdline a buf.
        strncat(buf, cmdline, sizeof(buf) - strlen(buf) - 1);
        vga_write(buf, 9, 0x07);

        // A remplazar por una llamada a two_stacks(),
        // definida en stacks.S.
        extern two_stacks();
        two_stacks();
        vga_write("vga_write() from stack1", 12, 0x17);
        vga_write("vga_write() from stack2", 13, 0x90);
    }
}
```

se concatena el valor de buf con el valor de cmdline, es decir que se copia la memoria de cmdline después del /0 de buf pero con un tope límite para no sobrepasar el tamaño del buffer.(el tamaño del buffer menos el tamaño del valor de la variable en buf y uno para el /0 del final)

```
char *strncat(char *dest, const char *src, size_t n){
    char * result = dest;
    while (*dest != 0)
        dest++;
    while (n > 0){
        *dest++ = *src++;
        n--;
        if(*src == 0){
            return result;
        }
    }
}
```

```

*dest = 0;
return result;
}

```

```

QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92300+07ED2300 C980

Booting from ROM...
kern2 loading.....
cmdline: kernel param1=hola param2=adios

vga_write() from stack1
vga_write() from stack2

```

Explicar cómo se comporta `strcat(3)` si, erróneamente, se declarase `buf` con tamaño 12. ¿Introduce algún error el código?

Si el buffer `buf` es de tamaño 12 se sobrepasa la memoria que se declaró, accediendo a memoria que no me pertenece, y en el mensaje no se imprimirán completamente los parámetros de entrada.

```

QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92300+07ED2300 C980

Booting from ROM...
kern2 loading.....
cmdline: ke

vga_write() from stack1
vga_write() from stack2

```

Compilar el siguiente programa, y explicar por qué se imprimen dos líneas distintas, en lugar de la misma dos veces:

```

jaz:~/Escritorio/jaz/fiuba/SISOP/kernel2 (master) $ ./a.out
sizeof buf = 256
sizeof buf = 8

```

La realidad es que en C no existe un array como parámetro. La función recibe un puntero y no tiene forma de saber que este es un array por eso al imprimir el size resulta que imprime el valor del puntero.

Ej: kern2-idt

1. ¿Cuántos bytes ocupa una entrada en la IDT?
8 bytes
2. ¿Cuántas entradas como máximo puede albergar la IDT?
256
3. ¿Cuál es el valor máximo aceptable para el campo *limit* del registro *IDTR*?
 $256 * 8 - 1 = 2047$
4. Indicar qué valor exacto tomará el campo *limit* para una IDT de 64 descriptores solamente.
 $64 * 8 - 1 = 511$
5. Consultar la sección 6.1 y explicar la diferencia entre interrupciones (§6.3) y excepciones (§6.4).

Las interrupciones y excepciones son eventos disparados por un programa en ejecución o algún componente de la computadora, que deben ser atendidos por el kernel. Las interrupciones pueden ocurrir en cualquier momento, por señales del hardware, como una señal de un teclado o del timer del cpu. Las excepciones ocurren cuando el procesador detecta un error durante la ejecución de un proceso, como es el caso de la división por cero o una violación de página.

Ej: kern2-isr

version A

```
$ make qemu-gdb
cc -g -m32 -O1 -ffreestanding -fno-stack-protector -nostdinc -idirafter
lib -I/usr/lib/gcc/x86_64-pc-linux-gnu/8.1.1/include
-I/usr/lib/gcc/x86_64-pc-linux-gnu/8.1.1/include-fixed -c -o interrupts.o
interrupts.c
ld -m elf_i386 -Ttext 0x100000 boot.o decls.o kern2-swap.o lib/string.o
stacks.o tasks.o func.o contador.o idt_entry.o interrupts.o handlers.o -o
kernel
grub-file --is-x86-multiboot kernel
qemu-system-i386 -serial mon:stdio -d guest_errors -S -gdb
tcp:127.0.0.1:7508 -kernel kernel
```

```

$ make gdb
gdb -q -s kernel -n -ex 'target remote 127.0.0.1:7508'
Reading symbols from kernel...done.
Remote debugging using 127.0.0.1:7508
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb) display/i $pc
1: x/i $pc
=> 0xffff0:    add %al,(%eax)
(gdb) b idt_init
Breakpoint 1 at 0x100909: file interrupts.c, line 32.
(gdb) c
Continuing.

Breakpoint 1, idt_init () at interrupts.c:32
32    void idt_init() {
1: x/i $pc
=> 0x100909 <idt_init>:    push    %ebx
(gdb) finish
Run till exit from #0  idt_init () at interrupts.c:32
kmain (mbi=0x9500) at kern2-swap.c:23
23        asm("int3");
1: x/i $pc
=> 0x10011b <kmain+46>:    int3
(gdb) x/10i $pc
=> 0x10011b <kmain+46>:    int3
    0x10011c <kmain+47>:    mov     $0xe0,%ecx
    0x100121 <kmain+52>:    mov     $0x12,%edx
    0x100126 <kmain+57>:    lea     -0x1629(%ebx),%eax
    0x10012c <kmain+63>:    call   0x100700 <vga_write2>
    0x100131 <kmain+68>:    add     $0x18,%esp
    0x100134 <kmain+71>:    pop     %ebx
    0x100135 <kmain+72>:    ret
    0x100136 <__x86.get_pc_thunk.bx>:    mov     (%esp),%ebx
    0x100139 <__x86.get_pc_thunk.bx+3>:    ret
(gdb) print $esp
$1 = (void *) 0x104fd8
(gdb) x/xw $esp
0x104fd8:    0x001009bc
(gdb) print $cs
$2 = 8
(gdb) print $eflags
$3 = [ AF ]
(gdb) print/x $eflags

```

```

$4 = 0x12
(gdb) stepi
breakpoint () at idt_entry.S:4
4          test %eax, %eax
1: x/i $pc
=> 0x1008bc <breakpoint+1>:    test    %eax,%eax
(gdb) print $esp
$5 = (void *) 0x104fcc
(gdb) x/3wx $sp
0x104fcc:    0x0010011c    0x00000008    0x00000012
(gdb) print $eflags
$6 = [ AF ]
(gdb) stepi
5          iret
1: x/i $pc
=> 0x1008be <breakpoint+3>:    iret
(gdb) print $eflags
$7 = [ ]
(gdb) print/x $eflags
$8 = 0x2
(gdb) stepi
kmain (mbi=0x9500) at kern2-swap.c:30
30          vga_write2("Funciona vga_write2?", 18, 0xE0);
1: x/i $pc
=> 0x10011c <kmain+47>:    mov     $0xe0,%ecx
(gdb) display/i $pc
2: x/i $pc
=> 0x10011c <kmain+47>:    mov     $0xe0,%ecx

```

-
Version B:

```

$ make gdb
gdb -q -s kernel -n -ex 'target remote 127.0.0.1:7508'
Reading symbols from kernel...done.
Remote debugging using 127.0.0.1:7508
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff0 in ?? ()
(gdb) display/i $pc
1: x/i $pc
=> 0xffff0:    add %al,(%eax)
(gdb) b idt_init

```

Breakpoint 1 at 0x100909: file interrupts.c, line 32.

(gdb) c

Continuing.

Breakpoint 1, idt_init () at interrupts.c:32

```
32 void idt_init() {
```

1: x/i \$pc

=> 0x100909 <idt_init>: push %ebx

(gdb) finish

Run till exit from #0 idt_init () at interrupts.c:32

kmain (mbi=0x9500) at kern2-swap.c:23

```
23 asm("int3");
```

1: x/i \$pc

=> 0x10011b <kmain+46>: int3

(gdb) x/10i \$pc

=> 0x10011b <kmain+46>: int3

0x10011c <kmain+47>: mov \$0xe0,%ecx

0x100121 <kmain+52>: mov \$0x12,%edx

0x100126 <kmain+57>: lea -0x1629(%ebx),%eax

0x10012c <kmain+63>: call 0x100700 <vga_write2>

0x100131 <kmain+68>: add \$0x18,%esp

0x100134 <kmain+71>: pop %ebx

0x100135 <kmain+72>: ret

0x100136 <__x86.get_pc_thunk.bx>: mov (%esp),%ebx

0x100139 <__x86.get_pc_thunk.bx+3>: ret

(gdb) print \$esp

\$1 = (void *) 0x104fd8

(gdb) x/xw \$esp

0x104fd8: 0x001009bc

(gdb) i r cs

cs 0x8 8

(gdb) print \$eflags

\$2 = [AF]

(gdb) print/x \$eflags

\$3 = 0x12

(gdb) stepi

breakpoint () at idt_entry.S:4

```
4 test %eax, %eax
```

1: x/i \$pc

=> 0x1008bc <breakpoint+1>: test %eax,%eax

(gdb) print \$esp

\$4 = (void *) 0x104fcc

(gdb) x/3wx \$sp

0x104fcc: 0x0010011c 0x00000008 0x00000012

(gdb) stepi


```

breakpoint () at idt_entry.S:5
5          ret
1: x/i $pc
=> 0x1008be <breakpoint+3>:    ret
(gdb) print $eflags
$5 = [ ]
(gdb) print/x $eflags
$6 = 0x2
(gdb) stepi
kmain (mbi=0x9500) at kern2-swap.c:30
30          vga_write2("Funciona vga_write2?", 18, 0xE0);
1: x/i $pc
=> 0x10011c <kmain+47>:    mov     $0xe0,%ecx

```

1. Para cada una de las siguientes maneras de guardar/restaurar registros en *breakpoint*, indicar si es correcto (en el sentido de hacer su ejecución “invisible”), y justificar por qué:

- // Opción A.

```

breakpoint:
    pusha
    ...
    call vga_write2
    popa
    iret

```

En este caso sí resulta invisible, pues las instrucciones *pusha* y *popa* se encargan de guardar y recuperar todos los registros de propósito general

- // Opción B.

```

breakpoint:
    push %eax
    push %edx
    push %ecx
    ...
    call vga_write2
    pop %ecx
    pop %edx
    pop %eax
    iret

```

Esta forma también resulta útil, pues se preservan los registros que serán modificados por la llamada a *vga_write2*.

- // Opción C.

```

breakpoint:
    push %ebx

```

```

push %esi
push %edi
...
call vga_write2
pop %edi
pop %esi
pop %ebx
iret

```

En este caso no resultaría invisible, pues la función `vga_write2` va a modificar los registros de propósito general `%eax`, `%ecx` y `%edx`.

2. Responder de nuevo la pregunta anterior, sustituyendo en el código `vga_write2` por `vga_write`. (Nota: el código representado con ... correspondería a la nueva convención de llamadas.)

La opción A sigue siendo válida, pues guardará todos los registros de propósito general. Las otras llamadas no resultan suficientes, pues no guardan algunos registros que pueden cambiar por las llamadas siguientes.

3. Si la ejecución del manejador debe ser enteramente invisible ¿no sería necesario guardar y restaurar el registro `EFLAGS` a mano? ¿Por qué?

No es necesario ya que se salva automáticamente en el stack al suceder el interrupt y la instrucción `iret` lo restaura

4. ¿En qué stack se ejecuta la función `vga_write()`?

Se ejecuta en el stack del kernel que está ejecutando la excepción de breakpoint

Ej: kern2-div

```

asm("div %4"
    : "=a"(linea), "=c"(color)
    : "0"(18), "1"(0xE0), "b"(1), "d"(0));

```

Explicar el funcionamiento exacto de la línea `asm(...)`:

- ¿qué cómputo se está realizando?

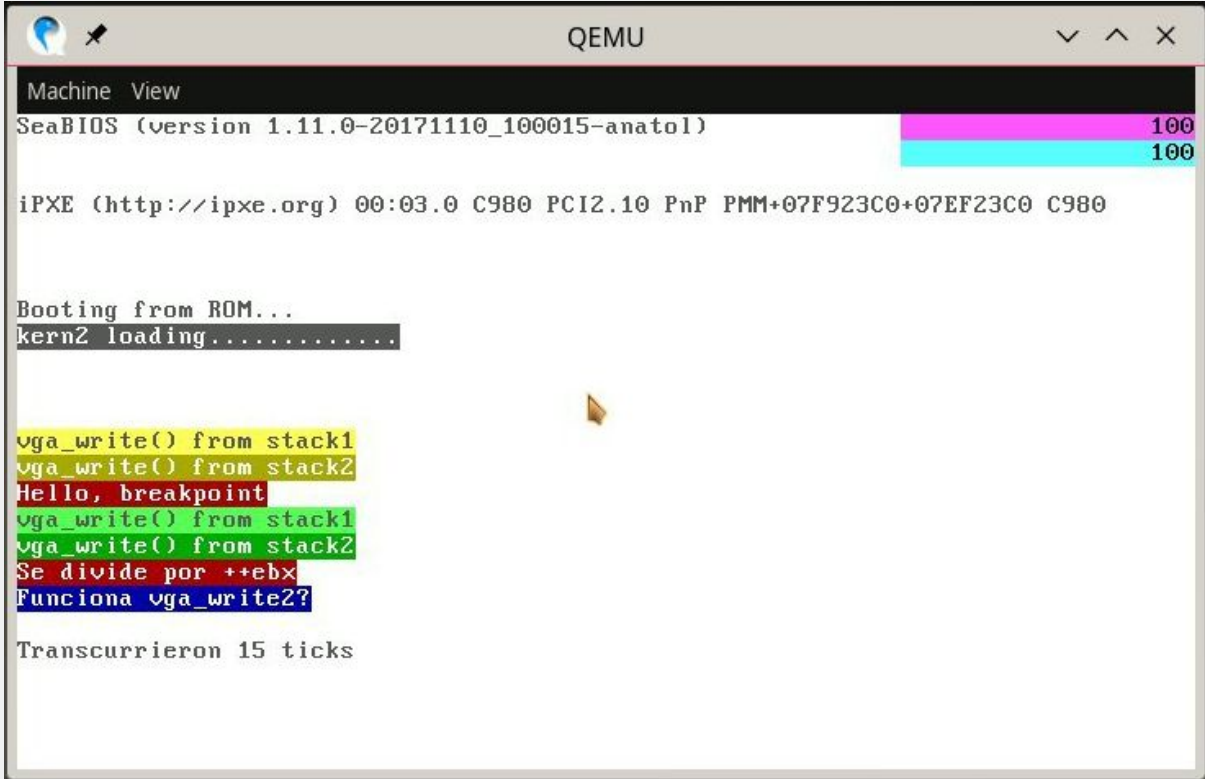
Se está realizando la instrucción de assembler `div`. La cual realiza una división entre los el par de registros `edx:eax` por el valor de `ebx`. El resultado de la división de guarda en el registro `eax` y el resto en `edx`. En este caso el primer registro tendrá el valor 18 y el segundo 0xE0, mientras que en `ebx` estará el valor 1, por la división que se realiza es por uno, no habrá resto, y el registro `edx` no cambiará su valor.

- ¿de dónde sale el valor de la variable `color`?

Como se define en la función `asm` los resultados se guardarán en `línea` y en `color`. Por lo que `línea` tendrá el valor resultado de `edx`, o sea la línea 18, y el `color` tendrá el resultado de la division guardado en `eax`, osea 0xE0 que será el color negro sobre amarillo.

- ¿por qué se da valor 0 a `%edx`?

Es necesario para inicializar el registro edx y que no contenga un número grande porque sino al realizar la división generará un error de overflow.



```
Machine View
SeaBIOS (version 1.11.0-20171110_100015-anatol)
100
100

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F923C0+07EF23C0 C980

Booting from ROM...
kern2 loading.....

vga_write() from stack1
vga_write() from stack2
Hello, breakpoint
vga_write() from stack1
vga_write() from stack2
Se divide por ++ebx
Funciona vga_write2?

Transcurrieron 15 ticks
```

Makefile

```
-----  
-  
CPPFLAGS := -ffreestanding -fno-stack-protector -nostdinc -idirafter  
lib  
GCC_PATH := /usr/lib/gcc/x86_64-pc-linux-gnu/8.1.1  
CPPFLAGS += -I$(GCC_PATH)/include -I$(GCC_PATH)/include-fixed  
  
CFLAGS := -g -m32 -O1  
  
QEMU := qemu-system-i386 -serial mon:stdio  
KERN := kernel  
BOOT := -kernel $(KERN) $(QEMU_EXTRA)  
  
qemu: $(KERN)  
      $(QEMU) $(BOOT)  
  
qemu-gdb: $(KERN)  
          $(QEMU) -d guest_errors -S -gdb tcp:127.0.0.1:7508 $(BOOT)  
  
gdb:  
      gdb -q -s kernel -n -ex 'target remote 127.0.0.1:7508'  
  
kernel: boot.o write.o kern2.o lib/string.o stacks.o tasks.o func.o  
contador.o idt_entry.o interrupts.o handlers.o  
      ld -m elf_i386 -Ttext 0x100000 $^ -o $@  
      grub-file --is-x86-multiboot $@  
  
%.o: %.S  
      $(CC) $(CFLAGS) $(CPPFLAGS) -c $<  
      #$(CC) $(CFLAGS) -c $<  
  
clean:  
      rm -f kernel *.o core  
  
.PHONY: clean qemu qemu-gdb gdb
```

boot.S

-

```
#include "multiboot.h"

#define KSTACK_SIZE 8192

.align 4
multiboot:
    .long MULTIBOOT_HEADER_MAGIC
    .long 0
    .long -(MULTIBOOT_HEADER_MAGIC)

.globl _start
_start:
    // Paso 1: Configurar el stack antes de llamar a kmain.
    movl $0, %ebp
    movl $(kstack + KSTACK_SIZE), %esp
    push %ebp

    // Paso 2: pasar la información multiboot a kmain. Si el
    // kernel no arrancó vía Multiboot, se debe pasar NULL.
    mov $0, %ecx
    test %eax, MULTIBOOT_BOOTLOADER_MAGIC
    cmovne %ecx, %ebx
    push %ebx
    call kmain

halt:
    hlt
    jmp halt

.data
.p2align 12
kstack:
    .space KSTACK_SIZE
```

kern2.c

```
-----  
-  
#include "decls.h"  
#include "multiboot.h"  
#include "interrupts.h"  
  
#define USTACK_SIZE 4096  
  
void kmain(const multiboot_info_t *mbi) {  
    int8_t linea;  
    uint8_t color;  
  
    vga_write("kern2 loading.....", 8, 0x70);  
  
    two_stacks();  
    two_stacks_c();  
    contador_run();  
  
    idt_init();  
    asm("int3");  
    irq_init();  
  
    asm("div %4"  
        : "=a"(linea), "=c"(color)  
        : "0"(18), "1"(0xE0), "b"(0), "d"(0));  
  
    vga_write2("Funciona vga_write?", linea, color);  
}  
  
static uint8_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));  
static uint8_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));  
  
void two_stacks_c() {  
    // Inicializar al *tope* de cada pila.  
    uintptr_t *a = (uintptr_t *) &stack1[USTACK_SIZE-1];  
    uintptr_t *b = (uintptr_t *) &stack2[USTACK_SIZE];  
  
    // Preparar, en stack1, la llamada:  
    // vga_write("vga_write() from stack1", 15, 0x57);  
    *(a--) = (uintptr_t) 0x57;  
    *(a--) = (uintptr_t) 15;  
    *(a) = (uintptr_t) "vga_write() from stack1";  
  
    // Preparar, en s2, la llamada:
```

```

// vga_write("vga_write() from stack2", 16, 0xD0);
b -= 3;
b[0] = (uintptr_t) "vga_write() from stack2";
b[1] = 16;
b[2] = 0xD0;

// Primera llamada usando task_exec().
task_exec((uintptr_t) vga_write, (uintptr_t) a);

// Segunda llamada con ASM directo.
asm("push %%ebp; movl %%esp, %%ebp; movl %0, %%esp; call *%1;
leave;"
: /* no outputs */
: "r"(b), "r"(vga_write));
}

```

decls.h

```
-----  
-  
#ifndef KERN2_DECL_H  
#define KERN2_DECL_H  
  
#include <stdint.h>  
  
void __attribute__((regparm(3)))  
vga_write2(const char *s, int8_t linea, uint8_t color);  
void __attribute__((regparm(2))) vga_write_cyan(const char *s, int8_t  
linea);  
void vga_write(const char *s, int8_t linea, uint8_t color);  
  
extern void task_exec(uintptr_t entry, uintptr_t stack);  
extern void task_swap(uintptr_t *esp);  
extern void two_stacks();  
void two_stacks_c();  
void contador_run();  
  
extern void breakpoint();  
extern void divzero();  
void irq_init();  
void timer();  
extern void ack_irq();  
extern void timer_asm();  
  
#endif
```


write.c

```
-----  
-  
#include "decls.h"  
  
volatile char *const VGABUF = (volatile char *) 0xb8000;  
  
void vga_write(const char *s, int8_t linea, uint8_t color){  
    volatile char *buf = VGABUF;  
    buf += 2 * 80 * linea;  
  
    while( *s != 0 ){  
        *buf++ = *s++;  
        *buf++ = color;  
    }  
}  
  
void __attribute__((regparm(2))) vga_write_cyan(const char *s, int8_t  
linea) {  
    vga_write(s, linea, 0xB0);  
}
```

func.S

-

.align 4

.text

.globl vga_write2

vga_write2:

push %ebp

movl %esp, %ebp

// color esta en ecx

push %ecx

// linea esta en edx

push %edx

// *s esta en eax

push %eax

call vga_write

leave

ret

stacks.S

```
-----  
-  
#define USTACK_SIZE 4096  
  
.data  
    .align 4096  
stack1:  
    .space USTACK_SIZE  
stack1_top:  
  
    .p2align 12  
stack2:  
    .space USTACK_SIZE  
stack2_top:  
  
msg1:  
    .asciz "vga_write() from stack1"  
msg2:  
    .asciz "vga_write() from stack2"  
    .align 4  
    .text  
    .globl two_stacks  
two_stacks:  
    // Preámbulo estándar  
    push %ebx  
    push %ebp  
    movl %esp, %ebp  
  
    // Registros para apuntar a stack1 y stack2.  
    mov $stack1_top, %eax  
    mov $stack2_top, %ebx  
  
    // Cargar argumentos a ambos stacks en paralelo.  
    movl $0x17, -4(%eax)  
    movl $0x90, -4(%ebx)  
  
    movl $12, -8(%eax)  
    movl $13, -8(%ebx)  
  
    movl $msg1, -12(%eax)  
    movl $msg2, -12(%ebx)  
  
    // Realizar primera llamada con stack1. Ayuda: usar LEA  
    // con el mismo offset que los últimos MOV para calcular  
    // la dirección deseada de ESP.
```

```
leal -12(%eax), %esp
call vga_write

// Restaurar stack original. ¿Es %ebp suficiente?
movl %ebp, %esp

// Realizar segunda llamada con stack2.
leal -12(%ebx), %esp
call vga_write

// Restaurar registros callee-saved, si se usaron.
movl %ebp, %esp

leave
pop %ebx
ret
```

tasks.S

```
-----  
-// Realiza una llamada a "entry" sobre el stack proporcionado.  
.align 4  
  
.text  
  
.globl task_exec  
  
task_exec:  
    // Preámbulo estándar  
    push %ebp  
    movl %esp, %ebp  
  
    //obtener parametros del stack  
    movl 8(%ebp), %eax  
    movl 12(%ebp), %esp  
    //llamado a funcion pasada por stack  
    call %eax  
  
    //return  
    leave  
    ret  
  
////////////////////////////////////  
// Pone en ejecución la tarea cuyo stack está en '*esp', cuyo  
// valor se intercambia por el valor actual de %esp. Guarda y  
// restaura todos los callee-saved registers.  
//  
.globl task_swap  
task_swap:  
  
//    guardar, en el stack de la tarea actual, los registros que son  
callee-saved: ebx, edi, esi  
    push %ebx  
    push %edi  
    push %esi  
    push %ebp  
  
// cargar en %esp el stack de la nueva tarea, y guardar en la variable esp  
el valor previo de %esp  
    movl 20(%esp), %eax // %eax = esp <-- variable esp  
    movl %esp, %edx     // %edx = %esp <-- stack actual  
    movl (%eax), %esp   // %esp = *esp <-- lo que apunta var esp  
    movl %edx, (%eax)   // esp debe apuntar a lo que habia en %esp  
                        // => *esp = %esp
```

// restaurar, desde el nuevo stack, los registros que fueron guardados por una llamada previa a task_swap(), y retornar (con la instrucción ret) a la nueva tarea.

pop %ebp

pop %esi

pop %edi

pop %ebx

ret

contador.c

```
-----  
-  
#include "decls.h"  
  
#define COUNTLEN 20  
#define TICKS (1ULL << 15)  
#define DELAY(x) (TICKS << (x))  
#define USTACK_SIZE 4096  
  
static volatile char *const VGABUF = (volatile void *) 0xb8000;  
  
static uintptr_t esp;  
static uint8_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));  
static uint8_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));  
  
static void yield() {  
    if (esp)  
        task_swap(&esp);  
}  
  
static void contador_yield(unsigned lim, uint8_t linea, char color) {  
    char counter[COUNTLEN] = {'0'}; // ASCII digit counter (RTL).  
  
    while (lim--) {  
        char *c = &counter[COUNTLEN];  
        volatile char *buf = VGABUF + 160 * linea + 2 * (80 - COUNTLEN);  
  
        unsigned p = 0;  
        unsigned long long i = 0;  
  
        while (i++ < DELAY(6)) // Usar un entero menor si va demasiado  
lento.  
            ;  
  
        while (counter[p] == '9') {  
            counter[p++] = '0';  
        }  
  
        if (!counter[p]++) {  
            counter[p] = '1';  
        }  
  
        while (c-- > counter) {  
            *buf++ = *c;  
            *buf++ = color;  
        }  
    }  
}
```

```

    }

    yield();
}

}

void contador_run() {
    // Configurar stack1 y stack2 con los valores apropiados.
    uintptr_t *a = (uintptr_t *) &stack1[USTACK_SIZE-1];
    uintptr_t *b = (uintptr_t *) &stack2[USTACK_SIZE-1];

    //contador_yield(100, 0, 0x2F);
    *(a--) = (uintptr_t) 0x2F;
    *(a--) = (uintptr_t) 0;
    *(a) = (uintptr_t) 100;

    //contador_yield(100, 1, 0x4F);
    *(b--) = (uintptr_t) 0x4F;
    *(b--) = (uintptr_t) 1;
    *(b--) = (uintptr_t) 100;

    *(b--) = 0; //
    *(b--) = (uintptr_t) contador_yield; // para retorno de task_swap

    *(b--) = 0; //ebx
    *(b--) = 0; //esi
    *(b--) = 0; //edi
    *(b) = 0; //ebx

    // Actualizar la variable estática 'esp' para que apunte
    // al del segundo contador.
    esp = (uintptr_t) b;

    // Lanzar el primer contador con task_exec.
    task_exec((uintptr_t) contador_yield, (uintptr_t) a);
}

```


interrupts.c

```
-----  
-  
#include "decls.h"  
#include "interrupts.h"  
  
static struct IDTR idtr;  
static struct Gate idt[256];  
  
// Multiboot siempre define "8" como el segmento de código.  
// (Ver campo CS en `info registers` de QEMU.)  
static const uint8_t KSEG_CODE = 8;  
  
// Identificador de "Interrupt gate de 32 bits" (ver IA32-3A,  
// tabla 6-2: IDT Gate Descriptors).  
static const uint8_t STS_IG32 = 0xE;  
  
void idt_install(uint8_t n, void (*handler)(void)) {  
    uintptr_t addr = (uintptr_t) handler;  
  
    idt[n].rpl = 0;  
    idt[n].type = STS_IG32;  
    idt[n].segment = KSEG_CODE;  
  
    idt[n].off_15_0 = addr & 0xFFFF;  
    idt[n].off_31_16 = addr >> 16;  
  
    idt[n].present = 1;  
}  
  
void idt_init() {  
    // (1) Instalar manejadores ("interrupt service routines").  
    // excepciones  
    idt_install(T_BRKPT, breakpoint);  
    // faults  
    idt_install(T_DIVIDE, divzero);  
  
    // (2) Configurar ubicación de la IDT.  
    idtr.base = (uintptr_t) idt;  
    idtr.limit = 256*8 - 1 ;  
  
    // (3) Activar IDT.  
    asm("lidt %0" : : "m"(idtr));  
}
```

```

////////////////////////////////////
//////////////////IRQ////////////////////////////////////
////////////////////////////////////
#define outb(port, data) asm("outb %b0,%w1" : : "a"(data), "d"(port));

static void irq_remap() {
    outb(0x20, 0x11);
    outb(0xA0, 0x11);
    outb(0x21, 0x20);
    outb(0xA1, 0x28);
    outb(0x21, 0x04);
    outb(0xA1, 0x02);
    outb(0x21, 0x01);
    outb(0xA1, 0x01);
    outb(0x21, 0x00);
    outb(0xA1, 0x00);
}

void irq_init() {
    // (1) Redefinir códigos para IRQs.
    irq_remap();

    // (2) Instalar manejadores.
    idt_install(T_TIMER, timer_asm);
    idt_install(T_KEYBOARD, ack_irq);

    // (3) Habilitar interrupciones.
    asm("sti");
}

```

idt_entry.S

#define PIC1 0x20
#define ACK_IRQ 0x20

.globl breakpoint
breakpoint:

```
// (1) Guardar registros.
push %eax
push %edx
push %ecx
// Preámbulo estándar
push %ebp
movl %esp, %ebp

// (2) Preparar argumentos de la llamada.
movl $0xB0, %ecx
movl $14, %edx
movl $breakpoint_msg, %eax

// (3) Invocar a vga_write2()
// vga_write2("Hello, breakpoint", 14, 0xB0);
call vga_write2

// (4) Restaurar registros.
leave
pop %ecx
pop %edx
pop %eax

// (5) Finalizar ejecución del manejador.
iret
```

.globl ack_irq
ack_irq:
// Indicar que se manejó la interrupción.
movl \$ACK_IRQ, %eax
outb %al, \$PIC1
iret

```

.globl timer_asm
timer_asm:
    // Guardar registros.
    push %eax
    push %ecx
    push %edx
    // hay que guardar?
    call timer
    // Restaurar registros.
    pop %edx
    pop %ecx
    pop %eax

    jmp ack_irq

.globl divzero
divzero:
    // Guardar registros
    push %eax
    push %ecx
    push %edx
    // incrementar ebx para cuando se reintente
    inc %ebx

    // (3) Invocar a vga_write_cyan()
    // vga_write_cyan("Se divide por ++ebx", 17);
    movl $17, %edx
    movl $divzero_msg, %eax
    call vga_write_cyan

    // Restaurar registros
    pop %edx
    pop %ecx
    pop %eax
    iret

.data
divzero_msg:
    .asciz "Se divide por ++ebx"
breakpoint_msg:
    .asciz "Hello, breakpoint"

```

handlers.c

```
-----  
-  
#include "decls.h"  
  
static unsigned ticks;  
  
void timer() {  
    if (++ticks == 15) {  
        vga_write("Transcurrieron 15 ticks", 20, 0x07);  
    }  
}
```