contador.c

Page 1/2

```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 exit() {
    uintptr_t tmp = esp;
    esp = 0;
    task_swap(&tmp);
}


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;
    a -= 3;
    a[2] = 0x2F;
    a[1] = 0;
    a[0] = 100;


    uintptr_t *b = (uintptr_t*) stack2 + USTACK_SIZE;
```

contador.c

Page 2/2

```c
    b -= 3;
    b[2] = 0x4F;
    b[1] = 1;
    b[0] = 90;

    // Llamada a exit al finalizar contador_yield
    *(--b) = (uintptr_t)exit;

    // Simulo que el primer swap no es el primero
    *(--b) = (uintptr_t)contador_yield;

    // Seteo los registros calle save a 0
    *(--b) = 0;
    *(--b) = 0;
    *(--b) = 0;
    *(--b) = 0;


    // Actualizar la variable estÃ¡tica âM-^@M-^XespâM-^@M-^Y 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);
}
```

handlers.c

Page 1/1

```c
#include "decls.h"

static unsigned ticks;

void timer() {
    if (++ticks == 15) {
        vga_write("Transcurrieron 15 ticks", 20, 0x07);
    }
}
```

interrupts.c

Page 1/2

```c
#include "decls.h"
#include "interrupts.h"

#define IDT_SIZE 256
static struct IDTR idtr;
static struct Gate idt[IDT_SIZE];

// Multiboot siempre define "8" como el segmento de cÃ³digo.
// (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;

#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, 0x0);
    outb(0xA1, 0x0);
}

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 & 0xFF;
    idt[n].off_31_16 = addr >> 16;

    idt[n].present = 1;
}

void idt_init() {
    // (1) Instalar manejadores ("interrupt service routines").
    idt_install(T_BRKPT, breakpoint);

    // (2) Configurar ubicaciÃ³n de la IDT.
    idtr.base = (uintptr_t) idt;
    idtr.limit = 8 * IDT_SIZE - 1 ;

    // (3) Activar IDT.
    asm("lidt %0" : : "m"(idtr));
}

void irq_init() {
    // (1) Redefinir cÃ³digos para IRQs.
    irq_remap();

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

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

interrupts.c

Page 2/2

```
67  }
```

kern2.c

Page 1/1

```
1   #include "decls.h"
2   #include "multiboot.h"
3
4   #define USTACK_SIZE 4096
5
6   void kmain(const multiboot_info_t *mbi) {
7       if (mbi) {
8           vga_write("kern2 loading.....", 8, 0x70);
9
10          two_stacks();
11          two_stacks_c();
12          contador_run();
13
14          idt_init();
15          irq_init();
16          asm("int3");
17
18          vga_write2("Funciona vga_write2?", 18, 0xE0);
19
20
21      }
22      asm("hlt");
23  }
24
25  static uint8_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));
26  static uint8_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));
27
28  void two_stacks_c() {
29      // Inicializar al *tope* de cada pila.
30      uintptr_t *a = (uintptr_t*) stack1 + USTACK_SIZE;
31      uintptr_t *b = (uintptr_t*) stack2 + USTACK_SIZE;
32
33      // Preparar, en stack1, la llamada:
34
35      *(a--) = 0x57;
36      *(a--) = 15;
37      *(a) = (uintptr_t) "vga_write() from stack1";
38
39      // Preparar, en s2, la llamada:
40
41      b -= 3;
42      b[2] = 0xD0;
43      b[1] = 16;
44      b[0] = (uintptr_t) "vga_write() from stack2";
45
46      task_exec((uintptr_t) vga_write, (uintptr_t) a);
47
48      asm("movl %0, %%esp; call *%1; movl %%ebp, %%esp"
49          :
50          : "r"(b), "r"(vga_write));
51  }
```

mbinfo.c

Page 1/1

```c
#include "decls.h"
#include "lib/string.h"
#include "multiboot.h"

#define KB_TO_MB_SHIFT 10 // 1KB*2^10->1MB

void print_mbinfo(const struct multiboot_info *mbi){
    char mem[256] = "Physical memory: ";
    char tmp[64] = {0};

    uint32_t total_size = mbi->mem_upper - mbi->mem_lower;
    if (fmt_int(total_size>>KB_TO_MB_SHIFT, tmp, sizeof tmp)) {
        strlcat(mem, tmp, sizeof mem);
        strlcat(mem, "MiB total", sizeof mem);
    }
    memset(tmp,0, sizeof(tmp));
    if (fmt_int(mbi->mem_lower, tmp, sizeof tmp)) {
        strlcat(mem, " (", sizeof mem);
        strlcat(mem, tmp, sizeof mem);
        strlcat(mem, " KiB base", sizeof mem);
    }

    memset(tmp,0, sizeof(tmp));
    if (fmt_int(mbi->mem_upper, tmp, sizeof tmp)) {
        strlcat(mem, ", ", sizeof mem);
        strlcat(mem, tmp, sizeof mem);
        strlcat(mem, " KiB extended) ", sizeof mem);
    }

    vga_write(mem, 10, 0x07);
}
```

write.c

Page 1/1

```c
#include "multiboot.h"
#include "decls.h"

#define VGABUF ((volatile char *) 0xB8000)
#define ROWS 25 // numero de filas de la pantalla
#define COLUMNS 80 // numero de columnas de la pantalla

static size_t int_width(uint64_t val) {
    size_t width = 0;
    while (val>0){
        val/=10;
        width++;
    }
    return width;
}

// Escribe en âM-^@M-^XsâM-^@M-^Y el valor de âM-^@M-^XvalâM-^@M-^Y en base 10 s
i su anchura
// es menor que âM-^@M-^XbufsizeâM-^@M-^Y. En ese caso devuelve true, caso de
// no haber espacio suficiente no hace nada y devuelve false.
bool fmt_int(uint64_t val, char *s, size_t bufsize) {
    size_t l = int_width(val);

    if (l >= bufsize)  // Pregunta: Â¿por quÃ© no "l > bufsize"?
                       // Respuesta: para agregar el \0
        return false;

    for (size_t i = l; i > 0; i--) {
        char ascii_digit = '0'+val %10;
        s[i-1]= ascii_digit;
        val/=10;
    }

    s[l]='\0';
    return true;
}

void vga_write(const char *s, int8_t linea, uint8_t color) {
    if (linea < 0) {
        linea = ROWS + linea;
    }

    volatile char* buff = VGABUF + linea * COLUMNS * 2;
    while (*s != '\0') {
        *buff++ = *s++;
        *buff++ = color;
    }

}
```

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_top, %esp
    push %ebp

    // Paso 2: pasar la informaciÃ³n multiboot a kmain. Si el
    // kernel no arrancÃ³ vÃa Multiboot, se debe pasar NULL.
    // Usar una instrucciÃ³n de comparaciÃ³n (TEST o CMP) para
    // comparar con MULTIBOOT_BOOTLOADER_MAGIC, pero no usar
    // un salto a continuaciÃ³n, sino una instrucciÃ³n CMOVcc
    // (copia condicional).
    // ...
    movl $0, %ecx
    cmp $MULTIBOOT_BOOTLOADER_MAGIC, %eax
    cmove %ebx, %ecx
    push %ecx

    call kmain
halt:
    hlt
    jmp halt

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

funcs.S


```
.data

.text
.globl vga_write2
vga_write2:
    push %ebp
    movl %esp, %ebp

    push %ecx
    push %edx
    push %eax
    call vga_write

    leave
    ret
```

idt_entry.S

Page 1/1

```
#define PIC1 0x20
#define ACK_IRQ 0x20

.globl ack_irq
ack_irq:
        // Indicar que se manejÃ³ la interrupciÃ³n.
        movl $ACK_IRQ, %eax
        outb %al, $PIC1
        iret


.globl breakpoint
breakpoint:
        // (1) Guardar registros.
        pusha
        // (2) Preparar argumentos de la llamada.
        // vga_write2("Hello, breakpoint", 14, 0xB0)
        movl $0xB0, %ecx
        movl $14, %edx
        movl $breakpoint_msg, %eax
        // (3) Invocar a vga_write2()
        call vga_write2
        // (4) Restaurar registros.
        popa
        // (5) Finalizar ejecuciÃ³n del manejador.
        iret


.globl timer_asm
timer_asm:
        // Guardar registros.
        pusha
        call timer
        // Restaurar registros.
        popa
        jmp ack_irq


.data
breakpoint_msg:
        .asciz "Hello, breakpoint"


```

stacks.S

Page 1/1

```
#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"

.text
.globl two_stacks
two_stacks:
        // PreÃ¡mbulo estÃ¡ndar
        push %ebp
        movl %esp, %ebp
        push %ebx

        // Registros para apuntar a stack1 y stack2.
        mov $stack1_top, %eax
        mov $stack2_top, %ebx

        // Cargar argumentos a ambos stacks en paralelo. Ayuda:
        // usar offsets respecto a %eax ($stack1_top), y lo mismo
        // para el registro usado para stack2_top.
        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 Ãºltimos MOV para calcular
        // la direcciÃ³n deseada de ESP.
        leal -12(%eax), %esp
        call vga_write

        // Restaurar stack original. Â¿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.
        pop %ebx

        leave
        ret
```

tasks.S


```
1  .data
2
3  .text
4  .globl task_exec
5  task_exec:
6
7      push %ebp
8      movl %esp, %ebp
9
10     movl 8(%ebp), %eax
11     movl 12(%ebp), %esp
12     call *%eax
13
14     leave
15     ret
16
17 .globl task_swap
18 task_swap:
19     push %ebp
20     push %ebx
21     push %edi
22     push %esi
23
24     movl 20(%esp), %eax
25
26     movl %esp, %ecx
27     movl (%eax), %esp
28     movl %ecx, (%eax)
29
30     pop %esi
31     pop %edi
32     pop %ebx
33     pop %ebp
34
35     ret
```

Table of Content