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 1/2

```c
#include "decls.h"

#define RELEASE_CODE 0x80
#define PROMPT_CURSOR '_'
#define MAX_SIZE 81
#define SPACE ' '
#define LEFT_SHIFT 42
#define RIGHT_SHIFT 54
#define BACKSPACE '\b'
#define SIMPLE_QUOTATION_MARK '\''
#define ENTER '\n'
#define ENIE 164

/**
 * Handler para el timer (IRQ0). Escribe un carÃ¡cter cada segundo.
 */
static const uint8_t hz_ratio = 18;  // Default IRQ0 freq (18.222 Hz).

void timer() {
    static char chars[MAX_SIZE];
    static unsigned ticks;
    static int8_t line = 21;
    static uint8_t idx = 0;

    if (++ticks % hz_ratio == 0) {
        chars[idx] = '.';
        chars[++idx] = '\0';
        vga_write(chars, line, 0x07);
    }

    if (idx >= sizeof(chars) - 1) {
        line++;
        idx = 0;
    }
}

/**
 * Mapa de "scancodes" a caracteres ASCII en un teclado QWERTY.
 */
static unsigned char klayout[128] = {
    //0-9
    0,   0,   '1', '2', '3', '4', '5', '6', '7', '8',
    //10-19
    '9', '0', 0,   0, BACKSPACE, 0, 'q', 'w', 'e', 'r',
    //20-29
    't', 'y', 'u', 'i', 'o', 'p', '[', ']',   ENTER,   0,
    //30-40
    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ENIE, SIMPLE_QUOTATION_MARK,
    //41-50
    0, 0,   0,   'z', 'x', 'c', 'v', 'b', 'n', 'm',
    //51-60
    ',', '.', '_', 0, 0, 0, SPACE, 0,0,0};

static const uint8_t KBD_PORT = 0x60;

static bool is_shift_pressed(uint8_t scancode) {

    bool released = scancode & RELEASE_CODE;
    scancode &= ~RELEASE_CODE;

    static bool pressed;
    if (scancode == RIGHT_SHIFT || scancode == LEFT_SHIFT) {
        pressed = !released;
    }
    return pressed;
}
```

handlers.c

Page 2/2

```c

/**
 * Handler para el teclado (IRQ1).
 *
 * Imprime la letra correspondiente por pantalla.
 */
void keyboard() {
    uint8_t code;
    static uint8_t actual_index = 0;
    static unsigned char kbd_entry_line[MAX_SIZE];

    asm volatile("inb %1,%0" : "=a"(code) : "n"(KBD_PORT));

    int8_t offset = is_shift_pressed(code)? -32 : 0;

    if (code >= sizeof(klayout) || !klayout[code])
        return;

    if (klayout[code] == BACKSPACE) {
        if (!actual_index)
            actual_index=1;
        kbd_entry_line[actual_index] = SPACE;
        kbd_entry_line[--actual_index] = PROMPT_CURSOR;
    } else {
        kbd_entry_line[actual_index] = klayout[code] + offset;
        kbd_entry_line[++actual_index] = PROMPT_CURSOR;
    }
    vga_write((char*)kbd_entry_line, 19, 0x0A);

}
```

interrupts.c — Page 1/2

```
1  #include "decls.h"
2  #include "interrupts.h"
3
4  #define IDT_SIZE 256
5  static struct IDTR idtr;
6  static struct Gate idt[IDT_SIZE];
7
8  // Multiboot siempre define "8" como el segmento de cÃ³digo.
9  // (Ver campo CS en `info registers` de QEMU.)
10 static const uint8_t KSEG_CODE = 8;
11
12 // Identificador de "Interrupt gate de 32 bits" (ver IA32-3A,
13 // tabla 6-2: IDT Gate Descriptors).
14 static const uint8_t STS_IG32 = 0xE;
15
16 #define outb(port, data) \
17         asm("outb %b0,%w1" : : "a"(data), "d"(port));
18
19 static void irq_remap() {
20     outb(0x20, 0x11);
21     outb(0xA0, 0x11);
22     outb(0x21, 0x20);
23     outb(0xA1, 0x28);
24     outb(0x21, 0x04);
25     outb(0xA1, 0x02);
26     outb(0x21, 0x01);
27     outb(0xA1, 0x01);
28     outb(0x21, 0x0);
29     outb(0xA1, 0x0);
30 }
31
32 void idt_install(uint8_t n, void (*handler)(void)) {
33     uintptr_t addr = (uintptr_t) handler;
34
35     idt[n].rpl = 0;
36     idt[n].type = STS_IG32;
37     idt[n].segment = KSEG_CODE;
38
39     idt[n].off_15_0 = addr & 0xFF;
40     idt[n].off_31_16 = addr >> 16;
41
42     idt[n].present = 1;
43 }
44
45 void idt_init() {
46     // (1) Instalar manejadores ("interrupt service routines").
47     idt_install(T_BRKPT, breakpoint);
48
49     // (2) Configurar ubicaciÃ³n de la IDT.
50     idtr.base = (uintptr_t) idt;
51     idtr.limit = 8 * IDT_SIZE - 1 ;
52
53     // (3) Activar IDT.
54     asm("lidt %0" : : "m"(idtr));
55 }
56
57 void irq_init() {
58     // (1) Redefinir cÃ³digos para IRQs.
59     irq_remap();
60
61     // (2) Instalar manejadores.
62     idt_install(T_TIMER, timer_asm);
63     idt_install(T_KEYBOARD, keyboard_asm);
64     idt_install(T_DIVIDE, divzero);
65
66     // (3) Habilitar interrupciones.
```

interrupts.c — Page 2/2

```
67     asm("sti");
68 }
```

kern2.c


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

#define USTACK_SIZE 4096

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;
        strlcat(buf, cmdline, sizeof(buf));
        vga_write(buf, 9, 0x07);

        print_mbinfo(mbi);
    }
        two_stacks();
        two_stacks_c();
        contador_run();

        idt_init();
        irq_init();
        asm("int3");

        int8_t linea;
        uint8_t color;
        asm("div %4"
            : "=a"(linea), "=c"(color)
            : "0"(18), "1"(0xE0), "b"(0), "d"(0));

        vga_write2("Funciona vga_write2?", linea, color);

    asm("hlt");
}

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

    // Preparar, en stack1, la llamada:

    *(a--) = 0x57;
    *(a--) = 15;
    *(a) = (uintptr_t) "vga_write() from stack1";

    // Preparar, en s2, la llamada:

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

    task_exec((uintptr_t) vga_write, (uintptr_t) a);

    asm("movl %0, %%esp; call *%1; movl %%ebp, %%esp"
        :
        : "r"(b), "r"(vga_write));
}
```

mbinfo.c


```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;
    }

}

void __attribute__((regparm(2)))
vga_write_cyan(const char *s, int8_t linea) {
    vga_write(s, linea, 0xB0);
}
```

boot.S

Page 1/1

```
#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 vÃ-a 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

Page 1/1

```
.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 1/2

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

.globl ack_irq
ack_irq:
    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

.globl keyboard_asm
keyboard_asm:
    pusha
    call keyboard

    popa
    jmp ack_irq

.globl divzero
divzero:
    // (1) Guardar registros.
    add $1, %ebx
    push %eax
    push %ecx
    push %edx

    // (2) Preparar argumentos de la llamada.
    //vga_write_cyan("Se divide por ++ebx", 17);

    movl $17, %edx
    movl $divzero_msg, %eax

    // (3) Invocar a vga_write_cyan()
    call vga_write_cyan

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

    // (5) Finalizar ejecuciÃ³n del manejador.
    iret
```

idt_entry.S


```
67
68  .data
69  breakpoint_msg:
70      .asciz "Hello, breakpoint"
71
72  divzero_msg:
73      .asciz "Se divide por ++ebx"
74
```

stacks.S


```
1   #define USTACK_SIZE 4096
2
3   .data
4           .align 4096
5   stack1:
6           .space USTACK_SIZE
7   stack1_top:
8
9           .p2align 12
10  stack2:
11          .space USTACK_SIZE
12  stack2_top:
13
14  msg1:
15          .asciz "vga_write() from stack1"
16  msg2:
17          .asciz "vga_write() from stack2"
18
19  .text
20  .globl two_stacks
21  two_stacks:
22          // PreÃ¡mbulo estÃ¡ndar
23          push %ebp
24          movl %esp, %ebp
25          push %ebx
26
27          // Registros para apuntar a stack1 y stack2.
28          mov $stack1_top, %eax
29          mov $stack2_top, %ebx
30
31          // Cargar argumentos a ambos stacks en paralelo. Ayuda:
32          // usar offsets respecto a %eax ($stack1_top), y lo mismo
33          // para el registro usado para stack2_top.
34          movl $0x17, -4(%eax)
35          movl $0x90, -4(%ebx)
36
37          movl $12,  -8(%eax)
38          movl $13, -8(%ebx)
39
40          movl $msg1,  -12(%eax)
41          movl $msg2, -12(%ebx)
42
43          // Realizar primera llamada con stack1. Ayuda: usar LEA
44          // con el mismo offset que los Ãºltimos MOV para calcular
45          // la direcciÃ³n deseada de ESP.
46          leal -12(%eax), %esp
47          call vga_write
48
49          // Restaurar stack original. Â¿Es %ebp suficiente?
50          movl %ebp, %esp
51
52          // Realizar segunda llamada con stack2.
53          leal -12(%ebx), %esp
54          call vga_write
55
56          // Restaurar registros callee-saved, si se usaron.
57          pop %ebx
58
59          leave
60          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