contador.c

Page 1/2

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

#define COUNTLEN 20
#define TICKS (1ULL << 15)
#define DELAY(x) (TICKS << (x))

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(unsigned lim, uint8_t linea, char color,const bool round_robin_mode) {
    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;
        }

        if (!round_robin_mode)
            yield();
    }
    if (round_robin_mode)
        kill_current_task();
}

static void contador_yield(unsigned lim, uint8_t linea, char color) {
    contador(lim, linea, color, false);
}

void round_robin(unsigned lim, uint8_t linea, char color) {
    contador(lim, linea, color, true);
}
```

contador.c

Page 2/2

```c
}

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] = 200;


    uintptr_t *b = (uintptr_t*) stack2 + USTACK_SIZE;
    b -= 3;
    b[2] = 0x4F;
    b[1] = 1;
    b[0] = 100;

    // 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

Page 1/2

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

static void contador1() {
    round_robin(600, 6, 0x3E);
}

static void contador2() {
    round_robin(200, 7, 0x2A);
}

static void contador3() {
    round_robin(400, 8, 0x1F);
}

void contador_spawn() {
    spawn(contador1);
    spawn(contador2);
    spawn(contador3);
}

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));
}

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();
```

kern2.c

Page 2/2

```

        contador_spawn();
        sched_init();
        //round_robin(420, 9, 0x1F);

        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");
}
```

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);
}
```

sched.c

Page 1/2

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

#define MAX_TASK 5
#define IF_FLAG 0x200

static struct Task Tasks[MAX_TASK];
static struct Task *current = NULL;

bool getFreeTask(struct Task **new_task) {
    size_t i=0;
    bool new_free_task=false;
    while ((i<MAX_TASK) && !new_free_task){
        if (Tasks[i].status == FREE){
            (*new_task) = &Tasks[i];
            new_free_task=true;
            }
        i++;
    }
    return new_free_task;
}

void sched_init() {
    size_t  i = 0;
    while (i<MAX_TASK && Tasks[i].status != READY){
        i++;
    }
    if (Tasks[i].status == READY) {
        current = &Tasks[i];
        current->status = RUNNING;
    }
}

void initialize_task(struct Task **task, void (*entry)(void)) {
    (*task)->status = READY;

    uint8_t* stack = &(*task)->stack[USTACK_SIZE] - sizeof(struct TaskFrame);
    (*task)->frame = (struct TaskFrame *)stack;

    (*task)->frame->ebp = 0;
    (*task)->frame->esp = 0;
    (*task)->frame->eax = 0;
    (*task)->frame->ebx = 0;
    (*task)->frame->ecx = 0;
    (*task)->frame->edx = 0;
    (*task)->frame->edi = 0;
    (*task)->frame->esi = 0;
    (*task)->frame->eip = (uint32_t)entry;
    (*task)->frame->cs = 0x8;
    (*task)->frame->eflags = IF_FLAG;
}

void spawn(void (*entry)(void)) {
    struct Task* new_task = NULL;
    bool success = getFreeTask(&new_task);
    if (!success)
        return;
    initialize_task(&new_task, entry);
}


static bool first_call = true;

void sched(struct TaskFrame *tf) {
    bool ready_task_found = false;
    struct Task *previous = current;
```

sched.c

Page 2/2

```
67
68      size_t task_index = 0;
69      while ((task_index < MAX_TASK) && (&Tasks[task_index] != previous)){
70          task_index++;
71      }
72
73      previous->status = READY;
74
75      while (!ready_task_found){
76          task_index = (task_index+1) % MAX_TASK;
77
78          if (Tasks[task_index].status == READY){
79              ready_task_found = true;
80
81              if (!first_call){
82                  previous->frame = tf;
83              } else
84                  first_call = false;
85              current = &Tasks[task_index];
86              current->status = RUNNING;
87
88              asm("movl %0, %%esp\n"
89                  "popa\n"
90                  "iret\n"
91              :
92              : "g"(current->frame)
93              : "memory");
94          }
95      }
96  }
97
98  void kill_current_task(){
99      current->status = DYING;
100     halt();
101 }
```

write.c

Page 1/1

```
1   #include "multiboot.h"
2   #include "decls.h"
3
4   #define VGABUF ((volatile char *) 0xB8000)
5   #define ROWS 25 // numero de filas de la pantalla
6   #define COLUMNS 80 // numero de columnas de la pantalla
7
8   static size_t int_width(uint64_t val) {
9       size_t width = 0;
10      while (val>0){
11          val/=10;
12          width++;
13      }
14      return width;
15  }
16
17  // Escribe en âM-^@M-^XsâM-^@M-^Y el valor de âM-^@M-^XvalâM-^@M-^Y en base 10 s
    i su anchura
18  // es menor que âM-^@M-^XbufsizeâM-^@M-^Y. En ese caso devuelve true, caso de
19  // no haber espacio suficiente no hace nada y devuelve false.
20  bool fmt_int(uint64_t val, char *s, size_t bufsize) {
21      size_t l = int_width(val);
22
23      if (l >= bufsize)  // Pregunta: Â¿por quÃ© no "l > bufsize"?
24                         // Respuesta: para agregar el \0
25          return false;
26
27      for (size_t i = l; i > 0; i--) {
28          char ascii_digit = '0'+val %10;
29          s[i-1]= ascii_digit;
30          val/=10;
31      }
32
33      s[l]='\0';
34      return true;
35  }
36
37  void vga_write(const char *s, int8_t linea, uint8_t color) {
38      if (linea < 0) {
39          linea = ROWS + linea;
40      }
41
42      volatile char* buff = VGABUF + linea * COLUMNS * 2;
43      while (*s != '\0') {
44          *buff++ = *s++;
45          *buff++ = color;
46      }
47
48  }
49
50  void __attribute__((regparm(2)))
51  vga_write_cyan(const char *s, int8_t linea) {
52      vga_write(s, linea, 0xB0);
53  }
```

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

.globl halt
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

    // Ack *antes* de llamar a sched()
    movl $ACK_IRQ, %eax
    outb %al, $PIC1

    // Llamada a sched con argumento
    push %esp
    call sched

    // Retornar (si se volviÃ³ de sched)
    addl $4, %esp
    popa
    iret

.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

```

idt_entry.S

Page 2/2

```
    // (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

.data
breakpoint_msg:
    .asciz "Hello, breakpoint"

divzero_msg:
    .asciz "Se divide por ++ebx"

```

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

Page 1/1

```
.data

.text
.globl task_exec
task_exec:

    push %ebp
    movl %esp, %ebp

    movl 8(%ebp), %eax
    movl 12(%ebp), %esp
    call *%eax

    leave
    ret

.globl task_swap
task_swap:
    push %ebp
    push %ebx
    push %edi
    push %esi

    movl 20(%esp), %eax

    movl %esp, %ecx
    movl (%eax), %esp
    movl %ecx, (%eax)

    pop %esi
    pop %edi
    pop %ebx
    pop %ebp

    ret
```

Table of Content

Table of Contents