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

#define COUNTLEN 20
#define TICKS (1ULL << 15)
#define DELAY(x) (TICKS << (x))
#define USTACK_SIZE 1024

static volatile char *const VGABUF = (volatile void *) 0xb8000;

static uintptr_t esp;
static uint32_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));
static uint32_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));

static void yield() {
    if (esp)
        task_swap(&esp);
}

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

static void contador(unsigned lim, uint8_t linea, char color, bool do_yield){
    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 (do_yield){
            yield();
        }
    }
}

void contador_round_robin(unsigned lim, uint8_t linea, char color) {
    contador (lim, linea, color, false);
    kill_task();
}

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


void contador_run() {

    uintptr_t *a = stack1 + sizeof(stack1);
    uintptr_t *b = stack2 + sizeof(stack2);

    *(--a) = 0x2F;  //Color
    *(--a) = 0;     //Linea
    *(--a) = 100;   //Limite

    *(--b) = 0x4F;  //Color
    *(--b) = 1;     //Linea
    *(--b) = 90;    //Limite

    //Direccion de retorno de la funcion contador_yield
    *(--b) = (uintptr_t)exit;

    //Direccion de retorno de la funcion task_swap en la primer iteracion
    *(--b) = (uintptr_t)contador_yield;

    //Registros calle-saved (ebp, ebx, esi, edi)
    *(--b) = 0;
    *(--b) = 0;
    *(--b) = 0;
    *(--b) = 0;

    esp = (uintptr_t)b;

    task_exec((uintptr_t) contador_yield, (uintptr_t) a);
}
```

```
#include "decls.h"
#include <stdbool.h>

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

void timer() {
    static char chars[81];
    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.
 */

#define CURSOR '^'
#define LEFT_ARROW '='       //Ascii que no se usa
#define RIGHT_ARROW '#'      //Ascii que no se usa
#define CAPSLOCK '!'         //Ascii que no se usa
#define MAX_WRITE 79
#define SPACE ' '
#define BACKSPACE '\b'
#define ENTER '\n'


static char klayout[128] = {
    0, 0, '1', '2', '3', '4', '5', '6', '7', '8',
    '9', '0', 0, 0, BACKSPACE, 0, 'q', 'w', 'e', 'r',
    't', 'y', 'u', 'i', 'o', 'p', '[', ']', ENTER, 0,
    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\'',
    0, 0, 0, 'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.',
    '/', 0, 0, 0, SPACE, CAPSLOCK, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, LEFT_ARROW, 0, RIGHT_ARROW};

static const uint8_t KBD_PORT = 0x60;

static bool use_upper(uint8_t code, int caps_lock) {
    static bool shift_pressed;

    bool released = code & 0x80;
    code = code & ~0x80;

    if (code == 42 || code == 54) {
        shift_pressed = !released;
    }

    if (caps_lock) {
        return !shift_pressed;
    }

    return shift_pressed;
}

/**
 * Handler para el teclado (IRQ1).
```

```c
 *
 * Imprime la letra correspondiente por pantalla.
 */
void keyboard() {
    uint8_t code;
    static char chars[MAX_WRITE];
    static char chars_selected[MAX_WRITE];
    static uint8_t idx = 0;
    static int init = 0;
    static int caps_lock = 0;

    if (init == 0) {
        for (int i = 0; i < MAX_WRITE; i++) {
            chars[i] = SPACE;
            chars_selected[i] = SPACE;
        }
        init = 1;
    }
    asm volatile("inb %1,%0" : "=a"(code) : "n"(KBD_PORT));

    int8_t upper_shift = use_upper(code, caps_lock) ? -32 : 0;

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

    if (klayout[code] < ’a’ || klayout[code] > ’z’) {
        //No es letra, no aplica mayuscula
        upper_shift = 0;
    }

    if (klayout[code] == BACKSPACE) {
        if (idx == 0) {
            idx++;
        }
        chars_selected[idx] = SPACE;
        chars[--idx] = SPACE;
        chars_selected[idx] = CURSOR;
    } else if (klayout[code] == LEFT_ARROW) {

        chars_selected[idx] = ’ ’;
        if (idx == 0) {
            idx++;
        }
        idx--;
        chars_selected[idx] = CURSOR;
    } else if (klayout[code] == RIGHT_ARROW) {

        chars_selected[idx] = SPACE;
        if (idx == MAX_WRITE) {
            idx--;
        }
        idx++;
        chars_selected[idx] = CURSOR;
    } else if (klayout[code] == ENTER) {

        //Borra toda la linea
        for (int i = 0; i < MAX_WRITE; i++) {
            chars[i] = SPACE;
            chars_selected[i] = SPACE;
        }
        idx = 0;
        chars_selected[idx] = CURSOR;
    } else if (klayout[code] == CAPSLOCK) {

        caps_lock = caps_lock ? 0 : 1;
    } else {

        if (idx >= MAX_WRITE) {
            chars_selected[idx] = SPACE;
            while (idx--)
```

```
                chars[idx] = SPACE;
            idx = 0;
            chars_selected[idx] = CURSOR;
        }
        chars_selected[idx] = SPACE;
        chars[idx++] = klayout[code] + upper_shift;
        chars_selected[idx] = CURSOR;
    }

    vga_write(chars, 19, 0x0A);
    vga_write(chars_selected, 20, 0x0A);
}
```

```c

#include "decls.h"
#include "interrupts.h"

#define IDT_DESCRIPTORS 256

static struct IDTR idtr;
static struct Gate idt[IDT_DESCRIPTORS];


// Multiboot siempre define "8" como el segmento de codigo.
// (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_init(){

    idt_install(T_BRKPT, breakpoint);
    idt_install(T_DIVIDE, divzero);

    idtr.base = (uintptr_t) idt;
    idtr.limit = IDT_DESCRIPTORS * 8 - 1;

    asm("lidt %0" : : "m"(idtr)); //activar el uso de la IDT configurada
}


void irq_init() {
    irq_remap();

    idt_install(T_TIMER, timer_asm);
    idt_install(T_KEYBOARD, keyboard_asm);

    asm("sti");
}


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;
}
```

kern2.c

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

#define MAX 1500


static void contador1() {
    contador_round_robin(MAX, 2, (char)0xB0);
}

static void contador2() {
    contador_round_robin(MAX, 3, (char)0xD0);
}

static void contador3() {
    contador_round_robin(MAX, 4, (char)0xE0);
}

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

void lab_kernel(){

    contador_run();

    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);

    kill_task();
}

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

    if (mbi && 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();

    //Ejercicios antes del promocional
    spawn(lab_kernel);

    //Ejercicio promocional
    contador_spawn();
    sched_init();

    idt_init();
    irq_init();
}
```

```c
#include "decls.h"

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

    int size = (mbi->mem_upper - mbi->mem_lower) >> 10;
    if (fmt_int(size, tmp, sizeof tmp)) {
        strlcat(mem, tmp, sizeof mem);
        strlcat(mem, " MiB total", sizeof mem);
    }

    char tmp2[64] = "";
    if (fmt_int(mbi->mem_lower, tmp2, sizeof tmp2)) {
        strlcat(mem, " (", sizeof mem);
        strlcat(mem, tmp2, sizeof mem);
        strlcat(mem, " KiB base", sizeof mem);
    }

    char tmp3[64] = "";
    if (fmt_int(mbi->mem_upper, tmp3, sizeof tmp3)) {
        strlcat(mem, ", ", sizeof mem);
        strlcat(mem, tmp3, sizeof mem);
        strlcat(mem, " KiB extended) ", sizeof mem);
    }

    vga_write(mem, 10, 0x07);

}
```

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

#define MAX_TASK 10
#define IF 0x200
#define STACK_SIZE 4096

static struct Task tasks[MAX_TASK];
static struct Task *current;
static bool first_iteration = true;

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

void spawn(void (*entry)(void)) {
    struct Task* new_task = NULL;
    for (size_t i = 0; i < MAX_TASK; i++){
        if (tasks[i].status == FREE){
            new_task = &tasks[i];
            break;
        }
    }

    new_task->status = READY;

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

    new_task->frame->edi = 0;
    new_task->frame->esi = 0;
    new_task->frame->ebp = 0;
    new_task->frame->esp = 0;
    new_task->frame->ebx = 0;
    new_task->frame->edx = 0;
    new_task->frame->ecx = 0;
    new_task->frame->eax = 0;

    new_task->frame->cs = 8;
    new_task->frame->eip = (uint32_t)entry;
    new_task->frame->eflags = IF;
}

void sched(struct TaskFrame *tf) {
    struct Task *new = NULL;
    struct Task *old = current;
    size_t i;

    for (i = 0; i < MAX_TASK; i++){
        if (&tasks[i] == old){
            break;
        }
    }

    old->status = READY;
    while (!new){
        i++;
        if (i >= MAX_TASK){
            i = 0;
        }
        if (tasks[i].status == READY){
            new = &tasks[i];
            if (first_iteration){
                first_iteration = false;
```

```c
            } else {
                old->frame = tf;
            }
            current = new;
            current->status = RUNNING;

            asm("movl %0, %%esp\n"
                "popa\n"
                "iret\n"
                :
                : "g"(current->frame)
                : "memory");
        }
    }

}

void kill_task(){
    current->status = DYING;
    halt();
}
```

two_stacks_c.c

```c
#include "decls.h"
#define USTACK_SIZE 1024

static uint32_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));
static uint32_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));

void two_stacks_c() {
    // Inicializar al *tope* de cada pila.
    uintptr_t *a = stack1 + sizeof(stack1);
    uintptr_t *b = stack2 + sizeof(stack2);

    // Preparar, en stack1, la llamada:
    //vga_write("vga_write() from stack1", 15, 0x57);

    // AYUDA 1: se puede usar alguna forma de pre- o post-
    // incremento/decremento, segun corresponda:
    //
    //     *(a++) = ...
    //     *(++a) = ...
    //     *(a--) = ...
    //     *(--a) = ...


    *(--a) = 0x57;

    *(--a) = 15;

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

    // AYUDA 2: para apuntar a la cadena con el mensaje,
    // es suficiente con el siguiente cast:
    //
    //   ... a ... = (uintptr_t) "vga_write() from stack1";

    // Preparar, en s2, la llamada:
    //vga_write("vga_write() from stack2", 16, 0xD0);

    // AYUDA 3: para esta segunda llamada, usar esta forma de
    // asignacion alternativa:
    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. Importante: no
    // olvidar restaurar el valor de %esp al terminar, y
    // compilar con: -fasm -fno-omit-frame-pointer.

    asm("movl %0, %%esp; call *%1; movl %%ebp, %%esp"
        : /* no outputs */
        : "r"(b), "r"(vga_write));
}

```

write.c

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

#define COLUMNS 80
#define ROWS 25

#define VGABUF ((volatile char *) 0xB8000)

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;
    }
}


bool fmt_int(uint64_t val, char *s, size_t bufsize){
    uint64_t digits = 0;
    uint64_t aux = val;
    while (aux > 0){
        digits++;
        aux /= 10;
    }

    if (digits >= bufsize){  //>= para agregar el \0
        return false;
    }

    for (int i = digits - 1; i >= 0; i--){
        s[i] = val % 10 + ’0’;
        val /= 10;
    }
    s[bufsize - 1] = ’\0’;
    return true;
}

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

```
// 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_end, %esp
    push %ebp

    // Paso 2: pasar la informacion multiboot a kmain. Si el
    // kernel no arranco via Multiboot, se debe pasar NULL.
    movl $0, %ecx
    CMP $MULTIBOOT_BOOTLOADER_MAGIC, %eax
    cmove %ebx, %ecx
    push %ecx

    // Usar una instruccion de comparacion (TEST o CMP) para
    // comparar con MULTIBOOT_BOOTLOADER_MAGIC, pero no usar
    // un salto a continuacion, sino una instruccion CMOVcc
    // (copia condicional).
    // ...

    call kmain

.globl halt
halt:
    hlt
    jmp halt

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

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

    push %ecx //color
    push %edx //linea
    push %eax //mensaje
    call vga_write

    leave
    ret
```

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

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


.globl timer_asm
timer_asm:
    // Guardar registros e invocar handler
    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   //caller saved
    push %ecx   //caller saved
    push %edx   //caller saved

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

    movl $17, %edx              //linea
    movl $divzero_msg, %eax     //mensaje

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

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

    // (5) Finalizar ejecucion del manejador.
    iret


.globl breakpoint
breakpoint:
    // (1) Guardar registros.
    pusha

    // (2) Preparar argumentos de la llamada.
    //vga_write2("Hello, breakpoint", 14, 0xB0);

    movl $0xB0, %ecx            //color
    movl $14, %edx              //linea
    movl $breakpoint_msg, %eax  //mensaje

    // (3) Invocar a vga_write2()
    call vga_write2

    // (4) Restaurar registros.
    popa

    // (5) Finalizar ejecucion del manejador.
    iret

.data
breakpoint_msg:
    .asciz "Hello, breakpoint"

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

```
// 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"


.text
.align 4
.globl two_stacks
two_stacks:
    // Preambulo estandar
    push %ebp
    push %ebx
    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. 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 ultimos MOV para calcular
    // la direccion 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.
    movl %ebp, %esp
    popl %ebx
    popl %ebp

    ret
```

tasks.S

```

.text
.align 4
.globl task_exec
task_exec:
    // Preambulo estandar
    push %ebp
    movl %esp, %ebp

    movl 8(%ebp), %ecx //entry
    movl 12(%ebp), %eax //stack

    leal 0(%eax), %esp
    call *%ecx

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

    ret



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

    movl 20(%esp), %eax //eax = puntero a esp siguiente


    movl %esp, %ecx
    movl 0(%eax), %esp
    movl %ecx, 0(%eax)

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

    ret
```

Table of Contents