```
1  #include "decls.h"
2
3  #define COUNTLEN 20
4  #define TICKS (1ULL << 15)
5  #define DELAY(x) (TICKS << (x))
6  #define USTACK_SIZE 1024
7
8  static volatile char *const VGABUF = (volatile void *) 0xb8000;
9
10 static uintptr_t esp;
11 static uint32_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));
12 static uint32_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));
13
14 static void yield() {
15     if (esp)
16         task_swap(&esp);
17 }
18
19 static void exit() {
20     uintptr_t tmp = esp;
21     esp = 0;
22     task_swap(&tmp);
23 }
24
25 static void contador_yield(unsigned lim, uint8_t linea, char color) {
26     char counter[COUNTLEN] = {'0'};  // ASCII digit counter (RTL).
27
28     while (lim--) {
29         char *c = &counter[COUNTLEN];
30         volatile char *buf = VGABUF + 160 * linea + 2 * (80 - COUNTLEN);
31
32         unsigned p = 0;
33         unsigned long long i = 0;
34
35         while (i++ < DELAY(6))  // Usar un entero menor si va demasiado lento.
36             ;
37
38         while (counter[p] == '9') {
39             counter[p++] = '0';
40         }
41
42         if (!counter[p]++) {
43             counter[p] = '1';
44         }
45
46         while (c-- > counter) {
47             *buf++ = *c;
48             *buf++ = color;
49         }
50
51         yield();
52     }
53 }
54
55
56 void contador_run() {
57
58     uintptr_t *a = stack1 + sizeof(stack1);
59     uintptr_t *b = stack2 + sizeof(stack2);
60
61     *(--a) = 0x2F;  //Color
62     *(--a) = 0;     //Linea
63     *(--a) = 100;   //Limite
64
65     *(--b) = 0x4F;  //Color
66     *(--b) = 1;     //Linea
67     *(--b) = 90;    //Limite
68
69     //Direccion de retorno de la funcion contador_yield
70     *(--b) = (uintptr_t)exit;
```

```
71
72      //Direccion de retorno de la funcion task_swap en la primer iteracion
73      *(--b) = (uintptr_t)contador_yield;
74
75      //Registros calle-saved (ebp, ebx, esi, edi)
76      *(--b) = 0;
77      *(--b) = 0;
78      *(--b) = 0;
79      *(--b) = 0;
80
81      esp = (uintptr_t)b;
82
83      task_exec((uintptr_t) contador_yield, (uintptr_t) a);
84  }
```

```c
#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, 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).
```

```
71   *
72   * Imprime la letra correspondiente por pantalla.
73   */
74  void keyboard() {
75      uint8_t code;
76      static char chars[MAX_WRITE];
77      static char chars_selected[MAX_WRITE];
78      static uint8_t idx = 0;
79      static int init = 0;
80      static int caps_lock = 0;
81
82      if (init == 0) {
83          for (int i = 0; i < MAX_WRITE; i++) {
84              chars[i] = SPACE;
85              chars_selected[i] = SPACE;
86          }
87          init = 1;
88      }
89      asm volatile("inb %1,%0" : "=a"(code) : "n"(KBD_PORT));
90
91      int8_t upper_shift = use_upper(code, caps_lock) ? -32 : 0;
92
93      if (code >= sizeof(klayout) || !klayout[code])
94          return;
95
96      if (klayout[code] < 'a' || klayout[code] > 'z') {
97          //No es letra, no aplica mayuscula
98          upper_shift = 0;
99      }
100
101     if (klayout[code] == BACKSPACE) {
102         if (idx == 0) {
103             idx++;
104         }
105         chars_selected[idx] = SPACE;
106         chars[--idx] = SPACE;
107         chars_selected[idx] = CURSOR;
108     } else if (klayout[code] == LEFT_ARROW) {
109
110         chars_selected[idx] = ' ';
111         if (idx == 0) {
112             idx++;
113         }
114         idx--;
115         chars_selected[idx] = CURSOR;
116     } else if (klayout[code] == RIGHT_ARROW) {
117
118         chars_selected[idx] = SPACE;
119         if (idx == MAX_WRITE) {
120             idx--;
121         }
122         idx++;
123         chars_selected[idx] = CURSOR;
124     } else if (klayout[code] == ENTER) {
125
126         //Borra toda la linea
127         for (int i = 0; i < MAX_WRITE; i++) {
128             chars[i] = SPACE;
129             chars_selected[i] = SPACE;
130         }
131         idx = 0;
132         chars_selected[idx] = CURSOR;
133     } else if (klayout[code] == CAPSLOCK) {
134
135         caps_lock = caps_lock ? 0 : 1;
136     } else {
137
138         if (idx >= MAX_WRITE) {
139             chars_selected[idx] = SPACE;
140             while (idx--)
```

```
141                 chars[idx] = SPACE;
142             idx = 0;
143             chars_selected[idx] = CURSOR;
144         }
145         chars_selected[idx] = SPACE;
146         chars[idx++] = klayout[code] + upper_shift;
147         chars_selected[idx] = CURSOR;
148     }
149 
150     vga_write(chars, 19, 0x0A);
151     vga_write(chars_selected, 20, 0x0A);
152 }
```

```
1
2  #include "decls.h"
3  #include "interrupts.h"
4
5  #define IDT_DESCRIPTORS 256
6
7  static struct IDTR idtr;
8  static struct Gate idt[IDT_DESCRIPTORS];
9
10
11 // Multiboot siempre define "8" como el segmento de codigo.
12 // (Ver campo CS en `info registers` de QEMU.)
13 static const uint8_t KSEG_CODE = 8;
14
15 // Identificador de "Interrupt gate de 32 bits" (ver IA32-3A,
16 // tabla 6-2: IDT Gate Descriptors).
17 static const uint8_t STS_IG32 = 0xE;
18
19
20 #define outb(port, data) \
21         asm("outb %b0,%w1" : : "a"(data), "d"(port));
22
23 static void irq_remap() {
24     outb(0x20, 0x11);
25     outb(0xA0, 0x11);
26     outb(0x21, 0x20);
27     outb(0xA1, 0x28);
28     outb(0x21, 0x04);
29     outb(0xA1, 0x02);
30     outb(0x21, 0x01);
31     outb(0xA1, 0x01);
32     outb(0x21, 0x0);
33     outb(0xA1, 0x0);
34 }
35
36
37 void idt_init(){
38
39     idt_install(T_BRKPT, breakpoint);
40     idt_install(T_DIVIDE, divzero);
41
42     idtr.base = (uintptr_t) idt;
43     idtr.limit = IDT_DESCRIPTORS * 8 - 1;
44
45     asm("lidt %0" : : "m"(idtr)); //activar el uso de la IDT configurada
46 }
47
48
49 void irq_init() {
50     irq_remap();
51
52     idt_install(T_TIMER, timer_asm);
53     idt_install(T_KEYBOARD, keyboard_asm);
54
55     asm("sti");
56 }
57
58
59 void idt_install(uint8_t n, void (*handler)(void)){
60     uintptr_t addr = (uintptr_t) handler;
61
62     idt[n].rpl = 0;
63     idt[n].type = STS_IG32;
64     idt[n].segment = KSEG_CODE;
65
66     idt[n].off_15_0 = addr & 0xFFFF;
67     idt[n].off_31_16 = addr >> 16;
68
69     idt[n].present = 1;
70 }
```

```
1  #include "decls.h"
2
3  #define USTACK_SIZE 4096
4
5  void kmain(const multiboot_info_t *mbi) {
6      vga_write("kern2 loading.....", 8, 0x70);
7
8      if (mbi && mbi->flags) {
9          char buf[256] = "cmdline: ";
10         char *cmdline = (void *) mbi->cmdline;
11
12         strlcat (buf, cmdline, sizeof(buf));
13         vga_write(buf, 9, 0x07);
14
15         print_mbinfo(mbi);
16     }
17
18     two_stacks();
19     two_stacks_c();
20
21     contador_run();
22
23
24     idt_init();
25     irq_init();
26     asm("int3");
27
28
29     int8_t linea;
30     uint8_t color;
31
32     asm("div %4"
33         : "=a"(linea), "=c"(color)
34         : "0"(18), "1"(0xE0), "b"(0), "d"(0));
35
36     vga_write2("Funciona vga_write2?", linea, color);
37 }
38
39
```

```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);

}
```

```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));
}
```

```
1  #include "decls.h"
2  #include "multiboot.h"
3
4  #define COLUMNS 80
5  #define ROWS 25
6
7  #define VGABUF ((volatile char *) 0xB8000)
8
9  void vga_write(const char *s, int8_t linea, uint8_t color) {
10     if (linea < 0) {
11         linea = ROWS + linea;
12     }
13     volatile char* buff = VGABUF + linea * COLUMNS * 2;
14     while (*s != '\0') {
15         *buff++ = *s++;
16         *buff++ = color;
17     }
18 }
19
20
21 bool fmt_int(uint64_t val, char *s, size_t bufsize){
22     uint64_t digits = 0;
23     uint64_t aux = val;
24     while (aux > 0){
25         digits++;
26         aux /= 10;
27     }
28
29     if (digits >= bufsize){  //>= para agregar el \0
30         return false;
31     }
32
33     for (int i = digits - 1; i >= 0; i--){
34         s[i] = val % 10 + '0';
35         val /= 10;
36     }
37     s[bufsize - 1] = '\0';
38     return true;
39 }
40
41 void __attribute__((regparm(2)))
42 vga_write_cyan(const char *s, int8_t linea) {
43     vga_write(s, linea, 0xB0);
44 }
45
46
```

```
1  // boot.S
2
3  #include "multiboot.h"
4
5  #define KSTACK_SIZE 8192
6
7  .align 4
8  multiboot:
9      .long MULTIBOOT_HEADER_MAGIC
10     .long 0
11     .long -(MULTIBOOT_HEADER_MAGIC)
12
13 .globl _start
14 _start:
15     // Paso 1: Configurar el stack antes de llamar a kmain.
16     movl $0, %ebp
17     movl $kstack_end, %esp
18     push %ebp
19
20     // Paso 2: pasar la informacion multiboot a kmain. Si el
21     // kernel no arranco via Multiboot, se debe pasar NULL.
22     movl $0, %ecx
23     CMP $MULTIBOOT_BOOTLOADER_MAGIC, %eax
24     cmove %ebx, %ecx
25     push %ecx
26
27     // Usar una instruccion de comparacion (TEST o CMP) para
28     // comparar con MULTIBOOT_BOOTLOADER_MAGIC, pero no usar
29     // un salto a continuacion, sino una instruccion CMOVcc
30     // (copia condicional).
31     // ...
32
33     call kmain
34
35 halt:
36     hlt
37     jmp halt
38
39 .data
40 .p2align 12
41 kstack:
42     .space KSTACK_SIZE
43 kstack_end:
44
```

```
1  .globl vga_write2
2  vga_write2:
3      push %ebp
4      movl %esp, %ebp
5
6      push %ecx //color
7      push %edx //linea
8      push %eax //mensaje
9      call vga_write
10
11     leave
12     ret
```

```
1  #define PIC1 0x20
2  #define ACK_IRQ 0x20
3
4  .globl ack_irq
5  ack_irq:
6      // Indicar que se manejo la interrupcion.
7      movl $ACK_IRQ, %eax
8      outb %al, $PIC1
9      iret
10
11
12 .globl timer_asm
13 timer_asm:
14         pusha
15         call timer
16         popa
17
18         jmp ack_irq
19
20 .globl keyboard_asm
21 keyboard_asm:
22         pusha
23         call keyboard
24         popa
25
26         jmp ack_irq
27
28
29 .globl divzero
30 divzero:
31     // (1) Guardar registros.
32
33     add $1, %ebx
34     push %eax   //caller saved
35     push %ecx   //caller saved
36     push %edx   //caller saved
37
38     // (2) Preparar argumentos de la llamada.
39     //vga_write_cyan("Se divide por ++ebx", 17);
40
41     movl $17, %edx              //linea
42     movl $divzero_msg, %eax     //mensaje
43
44     // (3) Invocar a vga_write_cyan()
45     call vga_write_cyan
46
47     // (4) Restaurar registros.
48     pop %edx
49     pop %ecx
50     pop %eax
51
52     // (5) Finalizar ejecucion del manejador.
53     iret
54
55
56 .globl breakpoint
57 breakpoint:
58     // (1) Guardar registros.
59     pusha
60
61     // (2) Preparar argumentos de la llamada.
62     //vga_write2("Hello, breakpoint", 14, 0xB0);
63
64     movl $0xB0, %ecx            //color
65     movl $14, %edx              //linea
66     movl $breakpoint_msg, %eax  //mensaje
67
68     // (3) Invocar a vga_write2()
69     call vga_write2
70
```

```
71      // (4) Restaurar registros.
72      popa
73
74      // (5) Finalizar ejecucion del manejador.
75      iret
76
77  .data
78  breakpoint_msg:
79      .asciz "Hello, breakpoint"
80
81  divzero_msg:
82      .asciz "Se divide por ++ebx"
```

```
1  // stacks.S
2  #define USTACK_SIZE 4096
3
4  .data
5      .align 4096
6  stack1:
7      .space USTACK_SIZE
8  stack1_top:
9
10     .p2align 12
11 stack2:
12     .space USTACK_SIZE
13 stack2_top:
14
15 msg1:
16     .asciz "vga_write() from stack1"
17 msg2:
18     .asciz "vga_write() from stack2"
19
20
21 .text
22 .align 4
23 .globl two_stacks
24 two_stacks:
25     // Preambulo estandar
26     push %ebp
27     push %ebx
28     movl %esp, %ebp
29
30     // Registros para apuntar a stack1 y stack2.
31     mov $stack1_top, %eax
32     mov $stack2_top, %ebx
33
34     // Cargar argumentos a ambos stacks en paralelo. Ayuda:
35     // usar offsets respecto a %eax ($stack1_top), y lo mismo
36     // para el registro usado para stack2_top.
37     movl $0x17, -4(%eax)
38     movl $0x90, -4(%ebx)
39
40     movl $12, -8(%eax)
41     movl $13, -8(%ebx)
42
43     movl $msg1, -12(%eax)
44     movl $msg2, -12(%ebx)
45
46     // Realizar primera llamada con stack1. Ayuda: usar LEA
47     // con el mismo offset que los ultimos MOV para calcular
48     // la direccion deseada de ESP.
49     leal -12(%eax), %esp
50     call vga_write
51
52     // Restaurar stack original. Â¿Es %ebp suficiente?
53     movl %ebp, %esp
54
55     // Realizar segunda llamada con stack2.
56     leal -12(%ebx), %esp
57     call vga_write
58
59     // Restaurar registros callee-saved, si se usaron.
60     movl %ebp, %esp
61     popl %ebx
62     popl %ebp
63
64     ret
```

```
1
2   .text
3   .align 4
4   .globl task_exec
5   task_exec:
6       // Preambulo estandar
7       push %ebp
8       movl %esp, %ebp
9
10      movl 8(%ebp), %ecx //entry
11      movl 12(%ebp), %eax //stack
12
13      leal 0(%eax), %esp
14      call *%ecx
15
16      // Restaurar registros callee-saved, si se usaron.
17      movl %ebp, %esp
18      popl %ebp
19
20      ret
21
22
23
24  .globl task_swap
25  task_swap:
26      push %ebp
27      push %ebx
28      push %edi
29      push %esi
30
31      movl 20(%esp), %eax //eax = puntero a esp siguiente
32
33
34      movl %esp, %ecx
35      movl 0(%eax), %esp
36      movl %ecx, 0(%eax)
37
38      pop %esi
39      pop %edi
40      pop %ebx
41      pop %ebp
42
43      ret
```

Table of Contents