

# Sistemas Operativos (75.08): Lab Kernel

Lavandeira, Lucas (#98042)  
lucaslavandeira@gmail.com

Rozanec, Matias (#97404)  
rozanecm@gmail.com

22.jun.2018

# Parte I

## Código

Listing 1: contador.c

```
1 #include "decls.h"
2
3 #define COUNTLEN 20
4 #define TICKS (1ULL << 15)
5 #define DELAY(x) (TICKS << (x))
6 #define USTACK_SIZE 4096
7
8 static volatile char *const VGABUF = (volatile void *) 0xb8000;
9
10 static uintptr_t esp;
11 static uint8_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));
12 static uint8_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));
13
14 static void exit() {
15     uintptr_t tmp = esp;
16     esp = 0;
17     if (tmp)
18         task_swap(&tmp);
19 }
20
21 static void yield() {
22     if (esp)
23         task_swap(&esp);
24 }
25
26 static void contador_yield(unsigned lim, uint8_t linea, char color) {
27     char counter[COUNTLEN] = {'0'}; // ASCII digit counter (RTL).
28
29     while (lim-- > 0) {
30         char *c = &counter[COUNTLEN];
31         volatile char *buf = VGABUF + 160 * linea + 2 * (80 - COUNTLEN);
32
33         unsigned p = 0;
34         unsigned long long i = 0;
35
36         while (i++ < DELAY(6)) // Usar un entero menor si va demasiado lento.
37             ;
38
39         while (counter[p] == '9') {
40             counter[p++] = '0';
41         }
42
43         if (!counter[p]++) {
44             counter[p] = '1';
45         }
46
47         while (c-- > counter) {
48             *buf++ = *c;
49             *buf++ = color;
50         }
51
52         yield();
53     }
54     // exit();
55 }
56
57 void contador_run() {
58     // Inicializar al *tope* de cada pila.
59     uintptr_t *a = (uintptr_t*) (stack1 + USTACK_SIZE);
60     uintptr_t *b = (uintptr_t*) (stack2 + USTACK_SIZE);
61
62     *--a = 0x2F;
63     *--a = 0;
64     *--a = 100;
65
66     *--b = 0x4F;
67     *--b = 1;
68     *--b = 10;
69     *--b = (uintptr_t) exit;
70     *--b = (uintptr_t) contador_yield;
71     *--b = 0;
72     *--b = 0;
73     *--b = 0;
74     *--b = 0;
75
76     esp = (uintptr_t) b;
77
78     task_exec((uintptr_t) contador_yield, (uintptr_t) a);
79 }
80
```

Listing 2: handlers.c

```

1 #include <stdbool.h>
2 #include "decls.h"
3
4 /**
5  * Handler para el timer (IRQ0). Escribe un caracter cada segundo.
6  */
7 static const uint8_t hz_ratio = 18; // Default IRQ0 freq (18.222 Hz).
8
9 void timer() {
10     static char chars[81];
11     static unsigned ticks;
12     static int8_t line = 20;
13     static uint8_t idx = 0;
14
15     if (++ticks % hz_ratio == 0) {
16         chars[idx] = '.';
17         chars[++idx] = '\0';
18         vga_write(chars, line, 0x07);
19     }
20
21     if (idx >= sizeof(chars)) {
22         line++;
23         idx = 0;
24     }
25 }
26
27 /**
28  * Mapa de "scancodes" a caracteres ASCII en un teclado QWERTY.
29  */
30 static char klayout[128] = {
31     0, 0, '1', '2', '3', '4', '5', '6', '7', '8', // 0-9
32     '9', '0', 0, 0, 0, 0, 'q', 'w', 'e', 'r', // 10-19
33     't', 'y', 'u', 'i', 'o', 'p', 0, 0, 0, 0, // 20-29
34     'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 0, // 30-40
35     0, 0, 0, 'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '; // 41-52
36 };
37 static const uint8_t KBD_PORT = 0x60;
38
39 static bool use_upper(uint8_t code) {
40     // return false;
41     static bool shift_pressed;
42
43     bool released = code & 0x80;
44     code = code & ~0x80;
45
46     if (code == 42 || code == 54) {
47         shift_pressed = !released;
48     }
49
50     return shift_pressed;
51 }
52
53 /**
54  * Handler para el teclado (IRQ1).
55  *
56  * Imprime la letra correspondiente por pantalla.
57  */
58 void keyboard() {
59     uint8_t code;
60     static char chars[81];
61     static uint8_t idx = 0;
62
63     asm volatile("inb %1,%0" : "=a"(code) : "n"(KBD_PORT));
64
65     int8_t upper_shift = use_upper(code) ? -32 : 0;
66
67     if (code >= sizeof(klayout) || !klayout[code])
68         return;
69
70     if (idx == 80) {
71         while (idx--)
72             chars[idx] = ' ';
73     }
74
75     chars[idx++] = klayout[code] + upper_shift;
76     vga_write(chars, 19, 0x0A);
77 }
78

```

Listing 3: interrupts.c

```

1 #include "decls.h"
2 #include "interrupts.h"
3 #define outb(port, data) \
4     asm("outb %b0,%w1" : : "a"(data), "d"(port));
5
6 static struct IDTR idtr;
7 static struct Gate idt[256];
8

```

```

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

```

Listing 4: kern0.c

```

1 #include "decls.h"
2 #include "multiboot.h"
3 #include "string.h"
4 #include "interrupts.h"
5
6 #define USTACK_SIZE 4096
7
8 unsigned char __attribute__((aligned(4096))) kstack[8192];
9
10 static uint8_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));
11 static uint8_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));
12
13 void two_stacks_c() {
14     // Inicializar al *tope* de cada pila.
15     uintptr_t *a = (uintptr_t*) (stack1 + USTACK_SIZE);
16     uintptr_t *b = (uintptr_t*) (stack2 + USTACK_SIZE);
17
18     // Preparar, en stack1, la llamada:
19     //   vga_write("vga_write() from stack1", 15, 0x57);
20
21     *--a = 0x57;
22     *--a = 15;
23     *--a = (uintptr_t) "vga_write() from stack1";
24
25     // AYUDA 1: se puede usar alguna forma de pre- o post-
26     // incremento/decremento, según corresponda:
27     //
28     //     *(a++) = ...
29     //     *(++a) = ...
30     //     *(a--) = ...
31     //     *--a = ...

```

```

32
33 // AYUDA 2: para apuntar a la cadena con el mensaje,
34 // es suficiente con el siguiente cast:
35 //
36 // ... a ... = (uintptr_t) "vga_write() from stack1";
37
38 // Preparar, en s2, la llamada:
39 // vga_write("vga_write() from stack2", 16, 0xD0);
40
41 // AYUDA 3: para esta segunda llamada, usar esta forma de
42 // asignación alternativa:
43 b -= 3;
44 b[0] = (uintptr_t) "vga_write() from stack2";
45 b[1] = 16;
46 b[2] = 0xD0;
47
48 // Primera llamada usando task_exec().
49 task_exec((uintptr_t) vga_write, (uintptr_t) a);
50
51 // Segunda llamada con ASM directo. Importante: no
52 // olvidar restaurar el valor de %esp al terminar, y
53 // compilar con: -fasm -fno-omit-frame-pointer.
54 asm("movl %0, %%esp; call *%1; movl %%ebp, %%esp"
55 : /* no outputs */
56 : "r"(b), "r"(vga_write));
57 }
58
59 void kmain(const multiboot_info_t *mbi) {
60     int8_t linea;
61     uint8_t color;
62
63     vga_write("kern2 loading.....", 8, 0x70);
64
65     if (mbi->flags & MULTIBOOT_INFO_CMDLINE) {
66         char buf[256] = "cmdline: ";
67         char *cmdline = (void *) mbi->cmdline;
68
69         strlcat(buf, cmdline, 256);
70         vga_write(buf, 9, 0x07);
71     }
72
73     char mem[256] = "Physical memory: ";
74     char tmp[64] = "";
75
76     if (fmt_int(mbi->mem_upper - mbi->mem_lower, tmp, sizeof tmp)) {
77         strlcat(mem, tmp, sizeof mem);
78         strlcat(mem, "MiB total", sizeof mem);
79     }
80
81     vga_write(mem, 10, 0x07);
82
83     /* A remplazar por una llamada a two_stacks(),
84      * definida en stacks.S.
85      */
86     /*
87     vga_write("vga_write() from stack1", 12, 0x17);
88     vga_write("vga_write() from stack2", 13, 0x90);
89     */
90     two_stacks();
91     two_stacks_c();
92
93     contador_run();
94
95     vga_write("antes del 2", 18, 0xE0);
96     vga_write2("Funciona vga_write2?", 18, 0xE0);
97
98     // Código ejercicio kern2-idt.
99     idt_init(); // (a)
100    irq_init();
101    asm("int3"); // (b)
102
103    asm("div %4"
104        : "=a"(linea), "=c"(color)
105        : "0"(18), "1"(0xE0), "b"(1), "d"(0));
106
107    vga_write("Funciona vga_write2?", linea, color);
108
109    asm("hlt");
110 }

```

Listing 5: write.c

```

1 #include "decls.h"
2 #define VGABUF ((volatile char *) 0xb8000)
3
4 void vga_write(const char *s, int8_t linea, uint8_t color) {
5     linea = linea % 24;
6     int row_offset = linea * 160;
7     volatile char* buf = VGABUF;
8     for(int i = 0; s[i] != '\0'; i++) {
9         buf[row_offset + 2 * i] = s[i];

```

```

10     buf[row_offset + 2 * i + 1] = color;
11 }
12 }
13 }
14
15 static size_t int_width(uint64_t val) {
16     size_t result = 1;
17     while (val /= 10) {
18         result++;
19     }
20
21     return result;
22 }
23
24 bool fmt_int(uint64_t val, char *s, size_t bufsize) {
25     size_t l = int_width(val);
26
27     if (l >= bufsize)
28         return false;
29
30     s += l;
31
32     uint8_t zero = '0';
33
34     while (val) {
35         *--s = zero + (val % 10);
36         val /= 10;
37     }
38     return true;
39 }
40
41
42 void __attribute__((regparm(2)))
43 vga_write_cyan(const char *s, int8_t linea) {
44     vga_write(s, linea, 0xB0);
45 }

```

Listing 6: boot.S

```

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 $kstackend, %esp
18     push %ebp
19
20     // Paso 2: pasar la información multiboot a kmain. Si el
21     // kernel no arrancó vía Multiboot, se debe pasar NULL.
22     //
23     // Usar una instrucción de comparación (TEST o CMP) para
24     // comparar con MULTIBOOT_BOOTLOADER_MAGIC, pero no usar
25     // un salto a continuación, sino una instrucción CMOVcc
26     // (copia condicional).
27     // ...
28
29     /* from multiboot.h:
30     ** This should be in %eax.
31     ** #define MULTIBOOT_BOOTLOADER_MAGIC 0x2BADB002
32     */
33
34     movl $0, %edx
35     test MULTIBOOT_HEADER_MAGIC, %eax
36     cmovb %ebx, %edx
37     push %edx
38     call kmain
39
40 halt:
41     hlt
42     jmp halt
43
44 .data
45 .p2align 12
46 kstack:
47     .space KSTACK_SIZE
48 kstackend:

```

Listing 7: idt\_entry.S

```

1  #define PIC1 0x20
2  #define ACK_IRQ 0x20
3
4
5  .globl breakpoint
6  breakpoint:
7      // (1) Guardar registros.
8      push %eax
9      push %edx
10     push %ecx
11
12     // (4) Restaurar registros.
13     pop %ecx
14     pop %edx
15     pop %eax
16
17     // (5) Finalizar ejecución del manejador.
18     iret
19
20
21 .globl ack_irq
22 ack_irq:
23     // Indicar que se manejó la interrupción.
24     movl $ACK_IRQ, %eax
25     outb %al, $PIC1
26     iret
27
28
29 .globl timer_asm
30 timer_asm:
31     // (1) Guardar registros.
32     push %eax
33     push %edx
34     push %ecx
35
36     call timer
37
38     // (4) Restaurar registros.
39     pop %ecx
40     pop %edx
41     pop %eax
42     jmp ack_irq
43
44
45 .globl divzero
46 divzero:
47     // (1) Guardar registros.
48     push %eax
49     push %edx
50     push %ecx
51
52     movl $divzero_msg, %eax
53     movl $17, %ecx
54     call vga_write_cyan
55
56     // (4) Restaurar registros.
57     pop %ecx
58     pop %edx
59     pop %eax
60     jmp ack_irq
61
62 .globl keyboard_asm
63 keyboard_asm:
64     // (1) Guardar registros.
65     push %eax
66     push %edx
67     push %ecx
68
69     call keyboard
70
71     // (4) Restaurar registros.
72     pop %ecx
73     pop %edx
74     pop %eax
75     jmp ack_irq
76
77 .data
78 breakpoint_msg:
79     .asciz "Hello, breakpoint"
80
81
82 divzero_msg:
83     .asciz "Se divide por ++ebx"

```

Listing 8: stacks.S

```

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  // stacks.S continuado
20  .text
21  .globl two_stacks
22  two_stacks:
23     // Preámbulo estándar
24     push %ebp
25     movl %esp, %ebp
26
27     // Registros para apuntar a stack1 y stack2.
28     mov $stack1_top, %eax
29     mov $stack2_top, %ecx    // Decidir qué registro usar.
30
31     // Cargar argumentos a ambos stacks en paralelo. Ayuda:
32     // usar offsets respecto a %eax ($stack1_top), y lo mismo
33     // para el registro usado para stack2_top.
34     movl $0x17, -4(%eax)
35     movl $0x90, -4(%edi)
36
37     movl $12, -8(%eax)
38     movl $13, -8(%edi)
39
40     movl $msg1, -12(%eax)
41     movl $msg2, -12(%edi)
42
43     // Realizar primera llamada con stack1. Ayuda: usar LEA
44     // con el mismo offset que los últimos MOV para calcular
45     // la dirección deseada de ESP.
46     leal -12(%eax), %esp
47     call vga_write
48
49     // Restaurar stack original. ¿Es %ebp suficiente?
50     movl %ebp, %esp
51
52     // Realizar segunda llamada con stack2.
53     leal -12(%edi), %esp
54     call vga_write
55
56     // Restaurar registros callee-saved, si se usaron.
57  //     ...
58
59     leave
60     ret

```

Listing 9: tasks.S

```

1  .text
2  .globl task_exec
3  task_exec:
4      // preambulo estandar
5      push %ebp
6      movl %esp, %ebp
7
8      movl 12(%ebp), %esp
9      call *8(%ebp)
10
11     /* restauro stack */
12     movl %ebp, %esp
13     leave
14     ret
15
16  .globl vga_write2
17  vga_write2:
18     push %ebp
19     movl %esp, %ebp
20
21     push %ecx
22     push %edx
23     push %eax
24
25     call vga_write
26
27     leave
28     ret
29
30  .globl task_swap
31  // Pone en ejecución la tarea cuyo stack está en '*esp', cuyo
32  // valor se intercambia por el valor actual de %esp. Guarda y
33  // restaura todos los callee-saved registers.

```



```

34 //void task_swap(uintptr_t *esp);
35
36 /* Implementar en tasks.S la función task_swap(). Como se indicó arriba,
37 esta función recibe como parámetro la ubicación en memoria de la variable
38 esp que apunta al stack de la tarea en "stand-by". La responsabilidad de
39 esta función es:
40
41     guardar, en el stack de la tarea actual, los registros que son callee-saved
42     Según https://wiki.skullsecurity.org/index.php?title=Registers, los
43     callee-saved regs son ebx, esi, edi, ebp.
44
45     cargar en %esp el stack de la nueva tarea, y guardar en la variable
46     esp el valor previo de %esp
47
48     restaurar, desde el nuevo stack, los registros que fueron guardados por
49     una llamada previa a task_swap(), y retornar (con la instrucción ret)
50     a la nueva tarea.
51
52 Para esta función, se recomienda no usar el preámbulo, esto es, no modificar
53 el registro %ebp al entrar en la función.
54 */
55 task_swap:
56     push %ebx
57     push %esi
58     push %edi
59     push %ebp
60
61     mov 20(%esp), %edx
62     mov %esp, %eax
63     mov (%edx), %esp
64     mov %eax, (%edx)
65
66     pop %ebp
67     pop %edi
68     pop %esi
69     pop %ebx
70     ret

```