contador.c Jun 21, 18 18:09 Page 1/2 #include "decls.h" 1 #include "sched.h" 2 3 #define COUNTLEN 20 4 #define TICKS (1ULL << 15)</pre> 5 #define DELAY(x) (TICKS << (x))</pre> 6 #define USTACK_SIZE 1024 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))); static uint32_t stack2[USTACK_SIZE] __attribute__((aligned(4096))); 12 13 14 static void yield() { 15 if (esp) 16 task_swap(&esp); 17 } 18

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

volatile char *buf = VGABUF + 160 * linea + 2 * (80 - COUNTLEN);

while (i++ < DELAY(6)) // Usar un entero menor si va demasiado lento.

char counter[COUNTLEN] = $\{'0'\};$ // ASCII digit counter (RTL).

19

20

21

22

23 24 25

26

27 28

29

30

31

33

34 35

36 37 38

39

41 42

43

44 45 46

47

48

49 50 51

52 53

54

59

60

63

69

70

} 61 62

static void exit() {

esp = 0;

uintptr_t tmp = esp;

task_swap(&tmp);

while (lim--) {

unsigned p = 0;

char *c = &counter[COUNTLEN];

unsigned long long i = 0;

while (counter[p] == '9') {

counter[p++] = '0';

counter[p] = '1';

while (c-- > counter) {

*buf++ = color;

contador (lim, linea, color, false);

contador (lim, linea, color, true);

uintptr_t *a = stack1 + sizeof(stack1);

*buf++ = *c;

if (do_yield) {

}

kill_task();

void contador_run() {

}

yield();

if (!counter[p]++)

void contador_round_robin(unsigned lim, uint8_t linea, char color) {

static void contador_yield(unsigned lim, uint8_t linea, char color) {

contador.c Jun 21, 18 18:09 Page 2/2 uintptr_t *b = stack2 + sizeof(stack2); 71 72 *(--a) = 0x2F;73 //Color *(--a) = 0;//Linea 74 *(--a) = 100;//Limite 75 76 * (--b) = 0x4F;//Color 77 *(--b) = 1;//Linea 78 *(--b) = 90;//Limite 79 80 //Direccion de retorno de la funcion contador_yield 81 $*(--b) = (uintptr_t) exit;$ 82 83 //Direccion de retorno de la funcion task_swap en la primer iteracion 84 *(--b) = (uintptr_t)contador_yield; 85 86 //Registros calle-saved (ebp, ebx, esi, edi) 87 *(--b) = 0;88 *(--b) = 0;89 *(--b) = 0;90 *(--b) = 0;91 92 93 $esp = (uintptr_t)b;$ 94 task_exec((uintptr_t) contador_yield, (uintptr_t) a); 95 96 }

```
Jun 21, 18 17:26 handlers.c Page 1/3
```

```
#include "decls.h"
1
   #include <stdbool.h>
2
3
4
    * Handler para el timer (IRQ0). Escribe un caracter cada segundo.
5
6
   static const uint8_t hz_ratio = 18; // Default IRQ0 freq (18.222 Hz).
8
   void timer() {
9
10
       static char chars[81];
       static unsigned ticks;
11
       static int8_t line = 21;
12
       static uint8_t idx = 0;
13
14
       if (++ticks % hz_ratio == 0) {
            chars[idx] = '.';
16
            chars[++idx] = ' \setminus 0';
17
            vga_write(chars, line, 0x07);
18
19
20
       if (idx >= sizeof(chars) - 1) {
21
           line++;
22
            idx = 0;
24
       }
   }
25
26
27
    * Mapa de "scancodes" a caracteres ASCII en un teclado QWERTY.
28
29
30
   #define CURSOR '^'
31
   #define LEFT_ARROW '='
                                 //Ascii que no se usa
   #define RIGHT_ARROW '#'
33
                                 //Ascii que no se usa
   #define CAPSLOCK '!'
                                 //Ascii que no se usa
34
   #define MAX_WRITE 79
35
   #define SPACE ''
   #define BACKSPACE '\b'
37
   #define ENTER '\n'
38
39
   41
42
43
44
45
46
       '/', 0, 0, 0, SPACE, CAPSLOCK, 0, 0, 0, 0, 0, 0, 0, 0, 0,
47
       0, 0, 0, 0, 0, 0, LEFT_ARROW, 0, RIGHT_ARROW);
49
   static const uint8_t KBD_PORT = 0x60;
50
51
   static bool use_upper(uint8_t code, int caps_lock) {
52
       static bool shift_pressed;
53
54
       bool released = code & 0x80;
55
       code = code & ~0x80;
56
57
58
       if (code == 42 || code == 54) {
            shift_pressed = !released;
59
60
61
       if (caps_lock) {
62
           return !shift_pressed;
63
64
65
       return shift_pressed;
66
   }
67
68
    * Handler para el teclado (IRQ1).
70
```

```
handlers.c
Jun 21, 18 17:26
                                                                                     Page 2/3
71
       Imprime la letra correspondiente por pantalla.
72
73
74
    void keyboard()
        uint8_t code;
75
        static char chars[MAX_WRITE];
76
        static char chars_selected[MAX_WRITE];
77
        static uint8_t idx = 0;
78
        static int init = 0;
79
        static int caps_lock = 0;
80
81
        if (init == 0) {
82
             for (int i = 0; i < MAX_WRITE; i++) {</pre>
83
                 chars[i] = SPACE;
84
                 chars_selected[i] = SPACE;
85
86
             init = 1;
87
88
89
        asm volatile("inb \%1,\%0" : "=a"(code) : "n"(KBD_PORT));
90
        int8_t upper_shift = use_upper(code, caps_lock) ? -32 : 0;
91
92
        if (code >= sizeof(klayout) || !klayout[code])
93
             return;
94
95
        if (klayout[code] < 'a' || klayout[code] > 'z') {
96
             //No es letra, no aplica mayuscula
97
             upper_shift = 0;
98
99
100
101
        if (klayout[code] == BACKSPACE) {
             if (idx == 0) {
102
                 idx++;
103
104
105
             chars_selected[idx] = SPACE;
106
             chars[--idx] = SPACE;
             chars_selected[idx] = CURSOR;
107
        } else if (klayout[code] == LEFT_ARROW) {
108
109
             chars_selected[idx] = '';
             if (idx == 0) {
111
                 idx++;
112
113
             idx--;
114
             chars_selected[idx] = CURSOR;
115
        } else if (klayout[code] == RIGHT_ARROW) {
116
117
             chars_selected[idx] = SPACE;
118
119
             if (idx == MAX_WRITE) {
                 idx--;
120
121
122
             idx++;
             chars_selected[idx] = CURSOR;
123
        } else if (klayout[code] == ENTER) {
124
125
             //Borra toda la linea
126
             for (int i = 0; i < MAX_WRITE; i++) {</pre>
127
128
                 chars[i] = SPACE;
                 chars_selected[i] = SPACE;
129
130
             idx = 0;
131
             chars_selected[idx] = CURSOR;
132
        } else if (klayout[code] == CAPSLOCK) {
133
134
             caps_lock = caps_lock ? 0 : 1;
135
136
        } else {
137
             if (idx >= MAX WRITE)
138
                 chars_selected[idx] = SPACE;
139
                 while (idx--)
140
```

handlers.c Jun 21, 18 17:26 Page 3/3 chars[idx] = SPACE;141 142 idx = 0;chars_selected[idx] = CURSOR; 143 144 chars_selected[idx] = SPACE; 145 chars[idx++] = klayout[code] + upper_shift; 146 chars_selected[idx] = CURSOR; 147 } 148 149 vga_write(chars, 19, 0x0A); vga_write(chars_selected, 20, 0x0A); 150 151 152

Jun 21, 18 17:30 **interrupts.c** Page 1/1

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

Jun 21, 18 18:09 **kern2.c** Page 1/1

```
#include "decls.h"
1
   #include "sched.h"
2
3
   #define MAX 1500
4
5
6
7
   static void contador1() {
        contador_round_robin(MAX, 2, (char)0xB0);
8
9
10
   static void contador2() {
11
        contador_round_robin(MAX, 3, (char)0xD0);
12
13
14
   static void contador3() {
15
        contador_round_robin(MAX, 4, (char)0xE0);
16
   }
17
18
19
   void contador_spawn() {
20
        spawn(contador1);
        spawn (contador2);
21
        spawn (contador3);
22
23
24
   void lab_kernel() {
25
26
        contador_run();
27
28
        asm("int3");
29
30
31
        int8_t linea;
32
        uint8_t color;
33
34
35
        asm("div %4"
            : "=a"(linea), "=c"(color)
36
             : "0"(18), "1"(0xE0), "b"(0), "d"(0));
37
38
        vga_write2("Funciona vga_write2?", linea, color);
39
40
41
        kill_task();
42
43
   void kmain(const multiboot_info_t *mbi) {
44
        vga_write("kern2 loading.....", 8, 0x70);
45
46
        if (mbi && mbi->flags) {
47
            char buf[256] = "cmdline: ";
48
            char *cmdline = (void *) mbi->cmdline;
49
50
51
            strlcat (buf, cmdline, sizeof(buf));
            vga\_write(buf, 9, 0x07);
52
53
            print_mbinfo(mbi);
54
        }
55
56
57
        two_stacks();
58
        two_stacks_c();
59
        //Ejercicios antes del promocional
60
61
        spawn(lab_kernel);
62
        //Ejercicio promocional
63
        contador_spawn();
64
        sched_init();
65
66
        idt_init();
67
68
        irq_init();
69
```

Jun 15, 18 14:26 **mbinfo.c** Page 1/1

```
#include "decls.h"
1
2
    void print_mbinfo(const struct multiboot_info *mbi){
3
         char mem[256] = "Physical memory: ";
         char tmp[64] = "";
5
6
         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);
8
9
10
11
12
         char tmp2[64] = "";
13
         if (fmt_int(mbi->mem_lower, tmp2, sizeof tmp2)) {
14
              strlcat(mem, "(", sizeof mem);
strlcat(mem, tmp2, sizeof mem);
16
              strlcat(mem, "KiB base", sizeof mem);
17
         }
18
19
         char tmp3[64] = "";
20
         if (fmt_int(mbi->mem_upper, tmp3, sizeof tmp3)) {
21
              strlcat(mem, ",", sizeof mem);
22
              strlcat(mem, tmp3, sizeof mem);
24
              strlcat(mem, "KiB extended)", sizeof mem);
         }
25
26
         vga\_write(mem, 10, 0x07);
27
28
   }
29
```

```
sched.c
Jun 21, 18 18:09
                                                                                      Page 1/2
    #include "decls.h"
1
    #include "sched.h"
 2
 3
    #define MAX_TASK 10
 4
    #define IF 0x200
 5
    #define STACK_SIZE 4096
 6
    static struct Task tasks[MAX_TASK];
 8
    static struct Task *current;
 9
    static bool first_iteration = true;
10
11
    void sched_init() {
12
         for (size t i = 0; i < MAX TASK; i++) {</pre>
13
             if (tasks[i].status == READY) {
14
                  current = &tasks[i];
15
                  current->status = RUNNING;
16
                  return;
17
             }
18
19
         }
20
    }
21
    void spawn(void (*entry)(void)) {
22
         struct Task* new_task = NULL;
23
         for (size_t i = 0; i < MAX_TASK; i++) {</pre>
24
             if (tasks[i].status == FREE) {
25
                  new_task = &tasks[i];
26
                  break;
27
28
         }
29
30
31
        new_task->status = READY;
        uint8_t* stack = &new_task->stack[STACK_SIZE] - sizeof(struct TaskFrame);
33
        new_task->frame = (struct TaskFrame *)stack;
34
35
         new_task->frame->edi = 0;
36
        new_task -> frame -> esi = 0;
37
         new_task -> frame -> ebp = 0;
38
         new task->frame->esp = 0;
39
         new_task - frame - bx = 0;
41
        new_task - frame - edx = 0;
        new_task -> frame -> ecx = 0;
42
43
        new task->frame->eax = 0;
44
45
        new_task->frame->cs = 8;
        new_task->frame->eip = (uint32_t)entry;
46
        new_task->frame->eflags = IF;
47
48
49
    void sched(struct TaskFrame *tf) {
50
51
        struct Task *new = NULL;
         struct Task *old = current;
52
53
         size_t i;
54
         for (i = 0; i < MAX_TASK; i++) {</pre>
55
             if (&tasks[i] == old) {
56
57
                 break;
58
             }
         }
59
60
         old->status = READY;
61
        while (!new) {
62
             i++;
63
             if (i >= MAX_TASK) {
64
                  i = 0;
65
66
             if (tasks[i].status == READY) {
```

new = &tasks[i];
if (first_iteration) {

first_iteration = false;

67 68

69

70

```
sched.c
Jun 21, 18 18:09
                                                                                         Page 2/2
71
                   } else {
72
                       old->frame = tf;
73
                  current = new;
74
                  current->status = RUNNING;
75
76
                  asm("movl %0, %%esp\n"
77
                       "popa\n"
"iret\n"
78
79
80
                       : "g"(current->frame)
81
                       : "memory");
82
            }
83
84
85
86
87
88
    void kill_task() {
         current->status = DYING;
89
         halt();
90
    }
91
```

Jun 21, 18 17:43

two stacks c.c

```
Page 1/1
   #include "decls.h"
1
   #define USTACK_SIZE 1024
2
3
   static uint32_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));
static uint32_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));
4
5
6
   void two_stacks_c() {
7
        // Inicializar al *tope* de cada pila.
8
        uintptr_t *a = stack1 + sizeof(stack1);
9
        uintptr_t *b = stack2 + sizeof(stack2);
10
11
        // Preparar, en stack1, la llamada:
12
        //vga_write("vga_write() from stack1", 15, 0x57);
13
14
        // AYUDA 1: se puede usar alguna forma de pre- o post-
15
        // incremento/decremento, segun corresponda:
16
        //
17
                *(a++) = ...
        //
18
                *(++a) = ...
19
                *(a--) = ...
20
                *(--a) = ...
21
22
        *(--a) = 0x57;
24
25
        *(--a) = 15;
26
27
        *(--a) = (uintptr_t) "vga_write() from stack1";
28
29
        // AYUDA 2: para apuntar a la cadena con el mensaje,
30
        // es suficiente con el siguiente cast:
31
32
        //
             ... a ... = (uintptr_t) "vga_write() from stack1";
33
34
35
        // Preparar, en s2, la llamada:
        //vga_write("vga_write() from stack2", 16, 0xD0);
36
37
        // AYUDA 3: para esta segunda llamada, usar esta forma de
38
        // asignacion alternativa:
39
        b = 3;
40
        b[0] = (uintptr_t) "vga_write() from stack2";
41
        b[1] = 16;
42
        b[2] = 0xD0;
43
44
45
        // Primera llamada usando task_exec().
        task_exec((uintptr_t) vga_write, (uintptr_t) a);
46
47
        // Segunda llamada con ASM directo. Importante: no
48
49
        // olvidar restaurar el valor de %esp al terminar, y
        // compilar con: -fasm -fno-omit-frame-pointer.
50
51
        asm("movl %0, %%esp; call *%1; movl %%ebp, %%esp"
52
             : /* no outputs */
53
             : "r"(b), "r"(vga_write));
54
   }
55
56
```

Jun 14, 18 17:03 **write.c** Page 1/1

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

Jun 21, 18 17:12 **boot.S** Page 1/1

```
// boot.S
1
   #include "multiboot.h"
3
   #define KSTACK_SIZE 8192
5
6
7
   .align 4
   multiboot:
8
        .long MULTIBOOT_HEADER_MAGIC
9
10
        .long 0
        .long - (MULTIBOOT_HEADER_MAGIC)
11
12
   .globl _start
13
14
   _start:
       // Paso 1: Configurar el stack antes de llamar a kmain.
15
       movl $0, %ebp
16
       movl $kstack_end, %esp
17
       push %ebp
18
19
        // Paso 2: pasar la informacion multiboot a kmain. Si el
20
        // kernel no arranco via Multiboot, se debe pasar NULL.
21
       movl $0, %ecx
22
       CMP $MULTIBOOT_BOOTLOADER_MAGIC, %eax
24
       cmove %ebx, %ecx
       push %ecx
25
26
        // Usar una instruccion de comparacion (TEST o CMP) para
27
        // comparar con MULTIBOOT_BOOTLOADER_MAGIC, pero no usar
28
        // un salto a continuacion, sino una instruccion CMOVcc
29
       // (copia condicional).
30
        // ...
31
32
       call kmain
33
34
35
   .globl halt
36
   halt:
       hlt
37
       jmp halt
38
39
40
   .data
   .p2align 12
41
   kstack:
42
        .space KSTACK_SIZE
43
44
   kstack_end:
45
```

```
funcs.S
                                                                                Page 1/1
Jun 15, 18 14:31
1
    .globl vga_write2
2
   vga_write2:
        push %ebp
3
       movl %esp, %ebp
 5
       push %ecx //color
 6
       push %edx //linea
        push %eax //mensaje
8
        call vga_write
9
10
        leave
        ret
12
```

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

70

idt_entry.S Jun 21, 18 17:31 Page 2/2 // (2) Preparar argumentos de la llamada. //vga_write2("Hello, breakpoint", 14, 0xB0); 71 72 73 //color movl \$0xB0, %ecx 74 movl \$14, %edx //linea 75 movl \$breakpoint_msg, %eax //mensaje 76 77 // (3) Invocar a vga_write2() 78 call vga_write2 79 80 // (4) Restaurar registros. 81 popa 82 83 // (5) Finalizar ejecucion del manejador. 84 iret 85 86 .data 87 breakpoint_msg: .asciz "Hello, breakpoint" 88 89 90 divzero_msg: 91 .asciz "Se divide por ++ebx" 92

Jun 21, 18 13:31 **stacks.S** Page 1/1

```
// stacks.S
1
   #define USTACK_SIZE 4096
2
3
4
   .data
        .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
   msq1:
15
        .asciz "vga_write() from stack1"
16
   msq2:
17
        .asciz "vga_write() from stack2"
18
19
20
   .text
21
   .align 4
22
   .globl two_stacks
23
   two_stacks:
24
        // Preambulo estandar
25
26
        push %ebp
        push %ebx
27
        movl %esp, %ebp
28
29
        // Registros para apuntar a stack1 y stack2.
30
        mov $stack1_top, %eax
31
        mov $stack2_top, %ebx
33
        // Cargar argumentos a ambos stacks en paralelo. Ayuda:
34
35
        // usar offsets respecto a %eax ($stack1_top), y lo mismo
        // para el registro usado para stack2_top.
36
        mov1 $0x17, -4(\%eax)
37
        movl $0x90, -4(%ebx)
38
39
        movl $12, -8(%eax)
        movl $13, -8(%ebx)
41
42
        movl $msg1, -12(%eax)
movl $msg2, -12(%ebx)
43
44
45
        // Realizar primera llamada con stack1. Ayuda: usar LEA
46
        // con el mismo offset que los ultimos MOV para calcular
47
        // la direccion deseada de ESP.
        leal -12(%eax), %esp
49
        call vga_write
50
51
        // Restaurar stack original. ¿Es %ebp suficiente?
52
53
        movl %ebp, %esp
54
        // Realizar segunda llamada con stack2.
55
        leal -12 (%ebx), %esp
56
        call vga_write
57
58
        // Restaurar registros callee-saved, si se usaron.
59
        movl %ebp, %esp
60
        popl %ebx
61
        popl %ebp
62
63
        ret
64
```

Jun 21, 18 13:31 **tasks.S** Page 1/1

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

				[/5.0	8] Sistemas	Operativos
Jur	n 21, 18 18:10	Table	of Content			Page 1/1
1	Table of Contents					
2	1 contador.c sheets	1 to	2 (2) pages	1- 2	97 lines	;
3	2 handlers.c sheets	3 to	5 (3) pages	3- 5	153 lines	3
4	3 interrupts.c sheets	6 to	6 (1) pages	6- 6	71 lines	;
5	4 kern2.c sheets	7 to	7 (1) pages	7- 7	70 lines	3
6	5 mbinfo.c sheets	8 to	8 (1) pages	8- 8	30 lines	3
7	6 sched.c sheets	9 to	10 (2) pages	9- 10	92 lines	3
8	7 two_stacks_c.c sheets	11 to	11 (1) pages	11- 11	57 lines	3
9	8 write.c sheets	12 to	12 (1) pages	12- 12	47 lines	}
10	9 boot.S sheets	13 to	13 (1) pages	13- 13	46 lines	}
11	10 funcs.S sheets	14 to	14 (1) pages	14- 14	13 lines	3
12	11 idt_entry.S sheets	15 to	16 (2) pages	15- 16	93 lines	}
13	12 stacks.S sheets	17 to	17 (1) pages	17- 17	65 lines	}
14	13 tasks.S sheets	18 to	18 (1) pages	18- 18	44 lines	}