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

```
#include "decls.h"

#define COUNTLEN 20
#define TICKS (1ULL << 15)
#define DELAY(x) (TICKS << (x))
#define USTACK_SIZE 4096

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;
    if (tmp)
        task_swap(&tmp);
}

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

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

        yield();
    }
//    exit();
}

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

    *(--a) = 0x2F;
    *(--a) = 0;
    *(--a) = 100;

    *(--b) = 0x4F;
    *(--b) = 1;
    *(--b) = 10;
    *(--b) = (uintptr_t) exit;
    *(--b) = (uintptr_t) contador_yield;
    *(--b) = 0;
    *(--b) = 0;
    *(--b) = 0;
    *(--b) = 0;


    esp = (uintptr_t) b;

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

Listing 2: handlers.c

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

/**
 * Handler para el timer (IRQ0). Escribe un car  cter 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 = 20;
    static uint8_t idx = 0;

    if (++ticks % hz_ratio == 0) {
        chars[idx] = '.';
        chars[++idx] = '\0';
        vga_write(chars, line, 0x07);
    }

    if (idx >= sizeof(chars)) {
        line++;
        idx = 0;
    }
}

/**
 * Mapa de "scancodes" a caracteres ASCII en un teclado QWERTY.
 */
static char klayout[128] = {
    0,   0,   '1', '2', '3', '4', '5', '6', '7', '8',             // 0-9
    '9', '0', 0,   0,   0,   0,   'q', 'w', 'e', 'r',             // 10-19
    't', 'y', 'u', 'i', 'o', 'p', 0,   0,   0,   0,               // 20-29
    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 0,   0,          // 30-40
    0,   0,   0,   'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.'};  // 41-52

static const uint8_t KBD_PORT = 0x60;

static bool use_upper(uint8_t code) {
    // return false;
    static bool shift_pressed;

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

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

    return shift_pressed;
}

/**
 * Handler para el teclado (IRQ1).
 *
 * Imprime la letra correspondiente por pantalla.
 */
void keyboard() {
    uint8_t code;
    static char chars[81];
    static uint8_t idx = 0;

    asm volatile("inb %1,%0" : "=a"(code) : "n"(KBD_PORT));

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

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

    if (idx == 80) {
        while (idx--)
            chars[idx] = ' ';
    }

    chars[idx++] = klayout[code] + upper_shift;
    vga_write(chars, 19, 0x0A);

}
```

Listing 3: interrupts.c

```c
#include "decls.h"
#include "interrupts.h"
#define outb(port, data) \
        asm("outb %b0,%w1" : : "a"(data), "d"(port));

static struct IDTR idtr;
static struct Gate idt[256];

```

```

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() {
    // (1) Instalar manejadores ("interrupt service routines").
    idt_install(T_BRKPT, breakpoint);

    idtr.base = (uintptr_t) idt;
    idtr.limit = sizeof idt;

    // (3) Activar IDT.
    asm("lidt %0" : : "m"(idtr));

}

// Multiboot siempre define "8" como el segmento de código.
// (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;

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;
}

void irq_init() {
    irq_remap();

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

    asm("sti");
}
```

Listing 4: kern0.c

```
#include "decls.h"
#include "multiboot.h"
#include "string.h"
#include "interrupts.h"

#define USTACK_SIZE 4096

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:
//    vga_write("vga_write() from stack1", 15, 0x57);

    *(--a) = 0x57;
    *(--a) = 15;
    *(--a) = (uintptr_t) "vga_write() from stack1";

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

    // AYUDA 2: para apuntar a la cadena con el mensaje,
```

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

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      cmove %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
13         // (2) Preparar argumentos de la llamada.
14         mov $breakpoint_msg, %eax
15         mov $14, %edx
16         mov $0xE0, %ecx
17
18         // (3) Invocar a vga_write2()
19         call vga_write2
20
21         // (4) Restaurar registros.
22         pop %ecx
23         pop %edx
24         pop %eax
25
26         // (5) Finalizar ejecución del manejador.
27         iret
28
29
30 .globl ack_irq
31 ack_irq:
32         // Indicar que se manejó la interrupción.
33         movl $ACK_IRQ, %eax
34         outb %al, $PIC1
35         iret
36
37
38 .globl timer_asm
39 timer_asm:
40         // (1) Guardar registros.
41         push %eax
42         push %edx
43         push %ecx
44
45         call timer
46
47         // (4) Restaurar registros.
48         pop %ecx
49         pop %edx
50         pop %eax
51         jmp ack_irq
52
53
54 .globl divzero
55 divzero:
56         // (1) Guardar registros.
57         push %eax
58         push %edx
59         push %ecx
60
61         movl $divzero_msg, %eax
62         movl $17, %ecx
63         call vga_write_cyan
64
65         // (4) Restaurar registros.
66         pop %ecx
67         pop %edx
68         pop %eax
69         jmp ack_irq
70
71 .globl keyboard_asm
72 keyboard_asm:
73         // (1) Guardar registros.
74         push %eax
75         push %edx
76         push %ecx
77
78         call keyboard
79
80         // (4) Restaurar registros.
81         pop %ecx
82         pop %edx
83         pop %eax
84         jmp ack_irq
85
86 .data
87 breakpoint_msg:
88         .asciz "Hello, breakpoint"
89
90
91 divzero_msg:
92         .asciz "Se divide por ++ebx"
```

Listing 8: stacks.S

```
// 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"
// stacks.S continuado
.text
.globl two_stacks
two_stacks:
        // Preámbulo estándar
        push %ebp
        movl %esp, %ebp

        // Registros para apuntar a stack1 y stack2.
        mov $stack1_top, %eax
        mov $stack2_top, %ecx   // Decidir qué registro usar.

        // 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(%edi)

        movl $12, -8(%eax)
        movl $13, -8(%edi)

        movl $msg1, -12(%eax)
        movl $msg2, -12(%edi)

        // Realizar primera llamada con stack1. Ayuda: usar LEA
        // con el mismo offset que los últimos MOV para calcular
        // la dirección deseada de ESP.
        leal -12(%eax), %esp
        call vga_write

        // Restaurar stack original. ¿Es %ebp suficiente?
        movl %ebp, %esp

        // Realizar segunda llamada con stack2.
        leal -12(%edi), %esp
        call vga_write

        // Restaurar registros callee-saved, si se usaron.
//        ...

        leave
        ret
```

Listing 9: tasks.S

```
.text
.globl task_exec
task_exec:
    // preambulo estandar
    push %ebp
    movl %esp, %ebp

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

    /* restauro stack */
    movl %ebp, %esp
    leave
    ret

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

    push %ecx
    push %edx
    push %eax

    call vga_write

```

```
    leave
    ret

.globl task_swap
// Pone en ejecución la tarea cuyo stack está en `*esp`, cuyo
// valor se intercambia por el valor actual de %esp. Guarda y
// restaura todos los callee-saved registers.
//void task_swap(uintptr_t *esp);

/* Implementar en tasks.S la función task_swap(). Como se indicó arriba,
esta función recibe como parámetro la ubicación en memoria de la variable
esp que apunta al stack de la tarea en "stand-by". La responsabilidad de
esta función es:

        guardar, en el stack de la tarea actual, los registros que son callee-saved
        Segun https://wiki.skullsecurity.org/index.php?title=Registers, los
        callee-saved regs son ebx, esi, edi, ebp.

        cargar en %esp el stack de la nueva tarea, y guardar en la variable
        esp el valor previo de %esp

        restaurar, desde el nuevo stack, los registros que fueron guardados por
        una llamada previa a task_swap(), y retornar (con la instrucción ret)
        a la nueva tarea.

Para esta función, se recomienda no usar el preámbulo, esto es, no modificar
el registro %ebp al entrar en la función.
*/
task_swap:
    push %ebx
    push %esi
    push %edi
    push %ebp

    mov  20(%esp), %edx
    mov  %esp, %eax
    mov  (%edx), %esp
    mov  %eax, (%edx)

    pop %ebp
    pop %edi
    pop %esi
    pop %ebx
    ret
```