Sistemas Operativos (75.08): Lab Kernel

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

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

29.jun.2018

Parte I
Código desalojo

Listing 1: contador.c

```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);
}


static void contador1() {
```

```c
    contador_yield(50000000, 2, 0x2F);
}

static void contador2() {
    contador_yield(50000000, 3, 0x6F);
}

static void contador3() {
    contador_yield(50000000, 4, 0x4F);
}

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

Listing 2: kern0.c

```c
#include "decls.h"
#include "multiboot.h"
#include "string.h"
#include "interrupts.h"
#include "sched.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,
    // 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
    // asignación 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));
}

void kmain(const multiboot_info_t *mbi) {
    int8_t linea;
    uint8_t color;

    vga_write("kern2 loading............", 8, 0x70);

    if (mbi->flags & MULTIBOOT_INFO_CMDLINE) {
        char buf[256] = "cmdline: ";
        char *cmdline = (void *) mbi->cmdline;

        strlcat(buf, cmdline, 256);
        vga_write(buf, 9, 0x07);
    }

    char mem[256] = "Physical memory: ";
```

```
    char tmp[64] = "";

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

    vga_write(mem, 10, 0x07);

    /* A remplazar por una llamada a two_stacks(),
     * definida en stacks.S.
     * */
    /*
    vga_write("vga_write() from stack1", 12, 0x17);
    vga_write("vga_write() from stack2", 13, 0x90);
    */
    two_stacks();
    two_stacks_c();

    contador_run();

    vga_write("antes del 2", 18, 0xE0);
    vga_write2("Funciona vga_write2?", 18, 0xE0);

    /* From Ej: kern2-task:
     * Por último, como "bootstrap" del planificador, se necesita una
     * llamada a la función sched_init() desde kmain(), antes de las llamadas
     * a idt_init()/irq_init(). Esto se necesita para que haya una
     * tarea inicial en ejecución.
     * */
    sched_init();
    // Código ejercicio kern2-idt.
    idt_init();   // (a)
    irq_init();
    asm("int3");  // (b)

    asm("div %4"
    : "=a"(linea), "=c"(color)
    : "0"(18), "1"(0xE0), "b"(1), "d"(0));

    vga_write("Funciona vga_write2?", linea, color);

    contador_spawn();
    asm("hlt");
}
```

Listing 3: idt_entry.S

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


.globl breakpoint
breakpoint:
        // (1) Guardar registros.
        push %eax
        push %edx
        push %ecx


        // (2) Preparar argumentos de la llamada.
        mov $breakpoint_msg, %eax
        mov $14, %edx
        mov $0xE0, %ecx

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

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

        // (5) Finalizar ejecución del manejador.
        iret


.globl ack_irq
ack_irq:
        // Indicar que se manejó la interrupción.
        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 volvió de sched)
    addl $4, %esp
    popa
    iret


.globl divzero
divzero:
        // (1) Guardar registros.
        push %eax
        push %edx
        push %ecx

        movl $divzero_msg, %eax
        movl $17, %ecx
        call vga_write_cyan

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

.globl keyboard_asm
keyboard_asm:
        // (1) Guardar registros.
        push %eax
        push %edx
        push %ecx

        call keyboard

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

.data
breakpoint_msg:
        .asciz "Hello, breakpoint"


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

Listing 4: sched.c

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

#define MAX_TASK 10

static struct Task Tasks[MAX_TASK];
static struct Task *current;

void sched_init() {
    current = &Tasks[0];

    for (int i = 0; i < MAX_TASK; i++) {
        Tasks[i].status = FREE;
        Tasks[i].frame = 0;
    }

    current->status = RUNNING;
}

void spawn(void (*entry)(void)) {
    for(int i = 0; i < MAX_TASK; i ++) {
        if (Tasks[i].status == FREE) {
            Tasks[i].status = READY;

            size_t frame_size = sizeof(struct TaskFrame);
            uint8_t* stack_top = &Tasks[i].stack[4096];

            Tasks[i].frame = stack_top - frame_size;

            Tasks[i].frame->edi = 0;
            Tasks[i].frame->esi = 0;
            Tasks[i].frame->ebp = 0;
            Tasks[i].frame->esp = 0;
            Tasks[i].frame->eax = 0;
            Tasks[i].frame->ecx = 0;
```

```
            Tasks[i].frame->edx = 0;
            Tasks[i].frame->ebx = 0;

            Tasks[i].frame->eflags = 0x0200;  // flag IF = 1

            Tasks[i].frame->cs = 8;      // Multiboot siempre pone '8'
                                         // como CS (ver interrupts.c)
            Tasks[i].frame->eip = entry;
            return;
        }
    }
}

void sched(struct TaskFrame *tf) {
    struct Task *new = 0;
    struct Task *old = current;

    int running_pos = 0;
    for (int i = 0; i < MAX_TASK; i++) {
        if (Tasks[i].status == RUNNING) {
            running_pos = i;
            break;
        }
    }

    int pos = running_pos;
    while (!new) {
        if (Tasks[pos].status == READY) {
            new = &Tasks[pos];
        }
        pos++;

        if (pos == MAX_TASK) {
            pos = 0;
        }
    }

    old->status = READY;
    old->frame = tf;

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

Listing 5: sched.h

```
void sched_init();
void contador_spawn();


enum TaskStatus {
    FREE = 0,
    READY,
    RUNNING,
    DYING,
};

struct TaskFrame {
    uint32_t edi;
    uint32_t esi;
    uint32_t ebp;
    uint32_t esp;
    uint32_t ebx;
    uint32_t edx;
    uint32_t ecx;
    uint32_t eax;
    /* below here defined by x86 hardware */
    uint32_t eip;
    uint16_t cs;
    uint16_t padding;
    uint32_t eflags;
} __attribute__((packed));


struct Task {
    uint8_t stack[4096];
    enum TaskStatus status;
    struct TaskFrame *frame;
};
```