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: kern0.c

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

#define USTACK_SIZE 4096

unsigned char __attribute__ ((aligned (4096))) kstack[8192];

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) {
    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);
```

```
92      vga_write2("Funciona vga_write2?", 18, 0xE0);
93
94      asm("hlt");
95  }
```

Listing 3: kern0.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  }
```

Listing 4: kern0.c

```
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:
```

```
    hlt
    jmp halt

.data
.p2align 12
kstack:
    .space KSTACK_SIZE
kstackend:
```

Listing 5: kern0.c

```
// 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 6: kern0.c

```
.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

```

```
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         Segun 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
```