FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

Sistemas Operativos 75.08

1° cuatrimestre 2018

Lab Kern2

Alejandro Daneri 97839
alejandrodaneri07@gmail.com

Rodrigo Aparicio 98967
rodrigoaparicio6@gmail.com

kern2-stack

Explicar: ¿qué significa “estar alineado”?

“Estar alineado” consiste en una restricción que impone el sistema operativo hacia las direcciones permitidas para los tipos de datos primitivos, requiriéndoles a los mismos que sean un múltiplo de K (típicamente 2, 4 u 8). Estas restricciones simplifican el diseño del hardware, formando una interfaz entre el procesador y la memoria.

Mostrar la sintaxis de C/GCC para alinear a 32 bits el arreglo kstack anterior.

La sintaxis para alinear a 32 bits el arreglo sería de la siguiente manera:
unsigned char kstack[8192] __attribute__ ((aligned (4));

¿A qué valor se está inicializando kstack? ¿Varía entre la versión C y la versión ASM? (Leer la documentación de as sobre la directiva .space.)

.space únicamente reserva un espacio en memoria pero no indica nada acerca del alineamiento del espacio reservado.
En el código de C el arreglo siempre debería tener una dirección múltiplo de 4.

Explicar la diferencia entre las directivas .align y .p2align de as, y mostrar cómo alinear el stack del kernel a 4 KiB usando cada una de ellas.

La diferencia consiste en que .p2align k avanza el location counter a la siguiente dirección más cercana que sea múltiplo de 2^k.
Por el otro lado .align k avanza el location counter a una dirección múltiplo de k.

Para alinear el stack a 4KB se puede hacer:
```
.p2align 12 // 2^12 = 4096 bytes = 4KB
.align   4096   // 4096 bytes = 4KB
```

La nueva versión del archivo boot.S se detalla en la sección final que muestra el código fuente.

Finalmente: mostrar en una sesión de GDB los valores de %esp y %eip al entrar en kmain, así como los valores almacenados en el stack en ese momento.

```
$ make gdb
gdb -q -s -kernel kern2 -n -ex 'target remote 127.0.0.1:7508'
Leyendo símbolos desde kern2...hecho.
Remote debugging using 127.0.0.1:7508
0x0000fff0 in ?? ()
(gdb) b kmain
Punto de interrupción 1 at 0x10002d: file kern2.c, line 5.
(gdb) c
Continuando.

Breakpoint 1, kmain (mbi=0x9500) at kern2.c:5
5     void kmain(const multiboot_info_t *mbi)
(gdb) p/x $esp
```

```
$1 = 0x104ff4
(gdb) p/x $eip
$2 = 0x10002d
(gdb) x/12 $esp
0x104ff4:     0x0010002a     0x00009500     0x00000000     0x6e72656b
0x105004:     0x71002032     0x00756d65     0x00000000     0x00000000
0x105014:     0x00000000     0x00000000     0x00000000     0x00000000
```

kern2-cmdline

Mostrar cómo implementar la misma concatenación, de manera correcta, usando strncat(3).

```
    if (mbi->flags) {
        char buf[256] = "cmdline: ";
        char *cmdline = (void *) mbi->cmdline;
        strncat(buf, cmdline, sizeof(buf)-strlen(buf)-1);
        vga_write(buf, 9, 0x07);

        print_mbinfo(mbi);
    }
```

De esta manera se evita escribir mas alla del tamaño real de buf (256).

Explicar cómo se comporta strlcat(3) si, erróneamente, se declarase buf con tamaño 12. ¿Introduce algún error el código?
En ese caso, lo que se quiera concatenar luego de los 9 caracteres (cmdline:) serán solo 2 caracteres mas, ya que uno quedará reservado para el caracter '\0' y sin introducir ningun error en el código.

Compilar el siguiente programa, y explicar por qué se imprimen dos líneas distintas, en lugar de la misma dos veces:

```
#include <stdio.h>

static void printf_sizeof_buf(char buf[256]) {
    printf("sizeof buf = %zu\n", sizeof buf);
}

int main(void) {
    char buf[256];
    printf("sizeof buf = %zu\n", sizeof buf);
    printf_sizeof_buf(buf);
}
```

Porque el sizeof buf de la función *print_sizeof_buf* lo que hace es devolver el tamaño de un char*, que es 8.

kern2-regcall

A continuación, mostrar con objdump -d el código generado por GCC para la llamada a *vga_write2()* desde la función principal:

```
0010007d <vga_write2>:
  10007d:   55                      push   %ebp
  10007e:   89 e5                   mov    %esp,%ebp
```

```
100080:   51                      push   %ecx
100081:   52                      push   %edx
100082:   50                      push   %eax
100083:   e8 7d 01 00 00          call   100205 <vga_write>
100088:   c9                      leave
100089:   c3                      ret
```

kern2-swap

Explicar, para el stack de cada contador, cuántas posiciones se asignan, y qué representa cada una.
En el stack **a** se asignaron tres posiciones, los argumentos para contador_yield.
En el stack **b** se asignaron tres posiciones en las cuales se pasan los tres argumentos para contador_yield. También se pasa el exit para simular un return y luego un puntero a la función contador_yield. Continuando, las cuatro posiciones que siguen se reservan para los registros callee saved,

kern2-exit

Cuando ambos contadores llegan al valor que se le asignó al segundo, el primero realiza otra iteración mientras que el segundo termina su ejecución por lo que el primero no puede terminar de realizar las próximas iteraciones restantes.

kern2-idt

¿Cuántos bytes ocupa una entrada en la IDT?

Una entrada de la IDT ocupa 8 bytes.

¿Cuántas entradas como máximo puede albergar la IDT?

Puede albergar como máximo hasta 256 entradas.

¿Cuál es el valor máximo aceptable para el campo limit del registro IDTR?

Como maximo puede haber 256 entradas de 8 bytes, por lo que limit puede ser como maximo 256 * 8 - 1.

Indicar qué valor exacto tomará el campo limit para una IDT de 64 descriptores solamente.

64 * 8 – 1 = valor de limit = 511.

Consultar la sección 6.1 y explicar la diferencia entre interrupciones (§6.3) y excepciones (§6.4).

Las excepciones ocurren cuando se ejecuta un instrucción que genera un error, por ejemplo, cuando se realiza una división por cero, o al acceder a una posición de memoria inválida.
En cambio, las interrupciones pueden ser generadas directamente por software, llamando a las instrucciones INT n. A su vez, las interrupciones pueden ocurrir por señales emitidas por el hardware, por ejemplo, una señal que es generada por

un periférico cuando un usuario interactúa con él.

kern2-isr

Sesión GDB versión A

```
$ make gdb
gdb -q -s -kernel kern2 -n -ex 'target remote 127.0.0.1:7508'
Leyendo símbolos desde kern2...hecho.
Remote debugging using 127.0.0.1:7508
0x0000fff0 in ?? ()
(gdb) display/i $pc
1: x/i $pc
=> 0xfff0:  add    %al,(%eax)
(gdb) b idt_init
Punto de interrupción 1 at 0x100343: file interrupts.c, line 30.
(gdb) c
Continuando.

Breakpoint 1, idt_init () at interrupts.c:30
30        idt_install(T_BRKPT, breakpoint);
1: x/i $pc
=> 0x100343 <idt_init+3>:    push   $0x10009d
(gdb) finish
Correr hasta la salida desde #0  idt_init () at interrupts.c:30
kmain (mbi=0x9500) at kern2.c:16
16            asm("int3");  // (b)
1: x/i $pc
=> 0x100145 <kmain+49>: int3
(gdb) disas
Dump of assembler code for function kmain:
   0x00100114 <+0>:     push   %ebp
   0x00100115 <+1>:     mov    %esp,%ebp
   0x00100117 <+3>:     sub    $0x8,%esp
   0x0010011a <+6>:     cmpl   $0x0,0x8(%ebp)
   0x0010011e <+10>:    je     0x10015d <kmain+73>
   0x00100120 <+12>:    sub    $0x4,%esp
   0x00100123 <+15>:    push   $0x70
   0x00100125 <+17>:    push   $0x8
   0x00100127 <+19>:    push   $0x100d01
   0x0010012c <+24>:    call   0x100411 <vga_write>
   0x00100131 <+29>:    call   0x100020 <two_stacks>
   0x00100136 <+34>:    call   0x1000ae <two_stacks_c>
   0x0010013b <+39>:    call   0x100261 <contador_run>
   0x00100140 <+44>:    call   0x100340 <idt_init>
=> 0x00100145 <+49>:    int3
   0x00100146 <+50>:    mov    $0xe0,%ecx
   0x0010014b <+55>:    mov    $0x12,%edx
   0x00100150 <+60>:    mov    $0x100d1c,%eax
   0x00100155 <+65>:    call   0x1000a1 <vga_write2>
   0x0010015a <+70>:    add    $0x10,%esp
   0x0010015d <+73>:    hlt
   0x0010015e <+74>:    leave
   0x0010015f <+75>:    ret
End of assembler dump.
(gdb) p $esp
$1 = (void *) 0x107fd8
(gdb) x/xw $esp
```

```
0x107fd8:   0x00100d01
(gdb) p $cs
$2 = 8
(gdb) p $eflags
$3 = [ AF ]
(gdb) p/x $eflags
$4 = 0x12
(gdb) stepi
breakpoint () at idt_entry.S:4
4         test %eax, %eax
1: x/i $pc
=> 0x10009e <breakpoint+1>:   test   %eax,%eax
(gdb) x/xw $esp
0x107fcc:   0x00100146
(gdb) x/4w $sp
0x107fcc:   0x00100146  0x00000008  0x00000012  0x00100d01
(gdb) p $eflags
$5 = [ AF ]
(gdb) stepi
5         iret
1: x/i $pc
=> 0x1000a0 <breakpoint+3>:   iret
(gdb) p $eflags
$6 = [ PF ]
(gdb) p/x $eflags
$7 = 0x6
(gdb) stepi
kmain (mbi=0x9500) at kern2.c:18
18            vga_write2("Funciona vga_write2?", 18, 0xE0);
1: x/i $pc
=> 0x100146 <kmain+50>: mov    $0xe0,%ecx
(gdb) disas
Dump of assembler code for function kmain:
   0x00100114 <+0>:     push   %ebp
   0x00100115 <+1>:     mov    %esp,%ebp
   0x00100117 <+3>:     sub    $0x8,%esp
   0x0010011a <+6>:     cmpl   $0x0,0x8(%ebp)
   0x0010011e <+10>:    je     0x10015d <kmain+73>
   0x00100120 <+12>:    sub    $0x4,%esp
   0x00100123 <+15>:    push   $0x70
   0x00100125 <+17>:    push   $0x8
   0x00100127 <+19>:    push   $0x100d01
   0x0010012c <+24>:    call   0x100411 <vga_write>
   0x00100131 <+29>:    call   0x100020 <two_stacks>
   0x00100136 <+34>:    call   0x1000ae <two_stacks_c>
   0x0010013b <+39>:    call   0x100261 <contador_run>
   0x00100140 <+44>:    call   0x100340 <idt_init>
   0x00100145 <+49>:    int3
=> 0x00100146 <+50>:    mov    $0xe0,%ecx
   0x0010014b <+55>:    mov    $0x12,%edx
   0x00100150 <+60>:    mov    $0x100d1c,%eax
   0x00100155 <+65>:    call   0x1000a1 <vga_write2>
   0x0010015a <+70>:    add    $0x10,%esp
   0x0010015d <+73>:    hlt
   0x0010015e <+74>:    leave
   0x0010015f <+75>:    ret
End of assembler dump.
```

```
(gdb) stepi
kmain (mbi=0x9500) at kern2.c:18
18            vga_write2("Funciona vga_write2?", 18, 0xE0);
1: x/i $pc
=> 0x100146 <kmain+50>: mov    $0xe0,%ecx
(gdb) p $eflags
$1 = [ AF ]
(gdb) p/x $eflags
$2 = 0x12
(gdb) p $sp
$3 = (void *) 0x107fd8
(gdb) p $cs
$4 = 8
(gdb) x/4w $sp
0x107fd8:   0x00100d01  0x00000008  0x00000070  0x00000000
```

Sesión GDB versión B

```
$ make gdb
gdb -q -s -kernel kern2 -n -ex 'target remote 127.0.0.1:7508'
Leyendo símbolos desde kern2...hecho.
Remote debugging using 127.0.0.1:7508
0x0000fff0 in ?? ()
(gdb) display/i $pc
1: x/i $pc
=> 0xfff0:  add    %al,(%eax)
(gdb) b idt_init
Punto de interrupción 1 at 0x100343: file interrupts.c, line 30.
(gdb) c
Continuando.

Breakpoint 1, idt_init () at interrupts.c:30
30        idt_install(T_BRKPT, breakpoint);
1: x/i $pc
=> 0x100343 <idt_init+3>:    push   $0x10009d
(gdb) finish
Correr hasta la salida desde #0  idt_init () at interrupts.c:30
kmain (mbi=0x9500) at kern2.c:16
16            asm("int3");  // (b)
1: x/i $pc
=> 0x100145 <kmain+49>: int3
(gdb) disas
Dump of assembler code for function kmain:
   0x00100114 <+0>:     push   %ebp
   0x00100115 <+1>:     mov    %esp,%ebp
   0x00100117 <+3>:     sub    $0x8,%esp
   0x0010011a <+6>:     cmpl   $0x0,0x8(%ebp)
   0x0010011e <+10>:    je     0x10015d <kmain+73>
   0x00100120 <+12>:    sub    $0x4,%esp
   0x00100123 <+15>:    push   $0x70
   0x00100125 <+17>:    push   $0x8
   0x00100127 <+19>:    push   $0x100d01
   0x0010012c <+24>:    call   0x100411 <vga_write>
   0x00100131 <+29>:    call   0x100020 <two_stacks>
   0x00100136 <+34>:    call   0x1000ae <two_stacks_c>
   0x0010013b <+39>:    call   0x100261 <contador_run>
   0x00100140 <+44>:    call   0x100340 <idt_init>
=> 0x00100145 <+49>:    int3
```

```
   0x00100146 <+50>:    mov    $0xe0,%ecx
   0x0010014b <+55>:    mov    $0x12,%edx
   0x00100150 <+60>:    mov    $0x100d1c,%eax
   0x00100155 <+65>:    call   0x1000a1 <vga_write2>
   0x0010015a <+70>:    add    $0x10,%esp
   0x0010015d <+73>:    hlt
   0x0010015e <+74>:    leave
   0x0010015f <+75>:    ret
End of assembler dump.
(gdb) p $sp
$1 = (void *) 0x107fd8
(gdb) x/xw $sp
0x107fd8:   0x00100d01
(gdb) p $cs
$2 = 8
(gdb) p $eflags
$3 = [ AF ]
(gdb) p/x $eflags
$4 = 0x12
(gdb) stepi
breakpoint () at idt_entry.S:4
4         test %eax, %eax
1: x/i $pc
=> 0x10009e <breakpoint+1>:  test   %eax,%eax
(gdb) x/4w $sp
0x107fcc:   0x00100146  0x00000008  0x00000012  0x00100d01
(gdb) p $eflags
$5 = [ AF ]
(gdb) p/x $eflags
$6 = 0x12
(gdb) stepi
5         ret
1: x/i $pc
=> 0x1000a0 <breakpoint+3>:  ret
(gdb) p $eflags
$7 = [ PF ]
(gdb) p/x $eflags
$8 = 0x6
(gdb) stepi
kmain (mbi=0x9500) at kern2.c:18
18            vga_write2("Funciona vga_write2?", 18, 0xE0);
1: x/i $pc
=> 0x100146 <kmain+50>: mov    $0xe0,%ecx
(gdb) disas
Dump of assembler code for function kmain:
   0x00100114 <+0>:     push   %ebp
   0x00100115 <+1>:     mov    %esp,%ebp
   0x00100117 <+3>:     sub    $0x8,%esp
   0x0010011a <+6>:     cmpl   $0x0,0x8(%ebp)
   0x0010011e <+10>:    je     0x10015d <kmain+73>
   0x00100120 <+12>:    sub    $0x4,%esp
   0x00100123 <+15>:    push   $0x70
   0x00100125 <+17>:    push   $0x8
   0x00100127 <+19>:    push   $0x100d01
   0x0010012c <+24>:    call   0x100411 <vga_write>
   0x00100131 <+29>:    call   0x100020 <two_stacks>
   0x00100136 <+34>:    call   0x1000ae <two_stacks_c>
```

```
   0x0010013b <+39>:    call   0x100261 <contador_run>
   0x00100140 <+44>:    call   0x100340 <idt_init>
   0x00100145 <+49>:    int3
=> 0x00100146 <+50>:    mov    $0xe0,%ecx
   0x0010014b <+55>:    mov    $0x12,%edx
   0x00100150 <+60>:    mov    $0x100d1c,%eax
   0x00100155 <+65>:    call   0x1000a1 <vga_write2>
   0x0010015a <+70>:    add    $0x10,%esp
   0x0010015d <+73>:    hlt
   0x0010015e <+74>:    leave
   0x0010015f <+75>:    ret
End of assembler dump.
1: x/i $pc
=> 0x100146 <kmain+50>: mov    $0xe0,%ecx
(gdb) p $sp
$1 = (void *) 0x107fd0
(gdb) x/4x $sp
0x107fd0:   0x00000008  0x00000012  0x00100d01  0x00000008
(gdb) p $cs
$2 = 8
(gdb) p $eflags
$3 = [ PF ]
(gdb) p/x $eflags
$4 = 0x6
```

Para la versión A, el %esp con el *iret* avanzó 2 bytes más que la versión B con *ret*. Esos dos bytes son los registros %cs y %eflags. En la versión A se ve como el %esp apunta dos direcciones superiores en el stack, por lo tanto el se restablecieron los valores anteriores que tenían ambos registros.
Para la versión B, *ret* no restablece dichos el estado de dichos registros. El %esp permanece 2 bytes debajo en el stack y no se restablece el valor de %cs ni %eflags. Dejando a %eflags con un valor incorrecto que cambió al ejecutar instrucciones antes de retornar.

Para cada una de las siguientes maneras de guardar/restaurar registros en *breakpoint*, indicar si es correcto (en el sentido de hacer su ejecución "invisible"), y justificar por qué:

OPCIÓN A: Es correcto, *pusha* guarda todos los registros con el valor que tenían y luego al hacer *popa* se restauran a como estaban antes de la instrucción.

OPCIÓN B: Es correcto ya que se están guardando solo los registros que son caller-saved.

OPCIÓN C: No es correcto, ya que deberían guardarse los registros caller-saved.

Responder de nuevo la pregunta anterior, sustituyendo en el código vga_write2 por vga_write.

Para los casos A y B sería correcto, por los mismos motivos. En cambio, para la opción C dependería si la implementación de vga_write usa los registros caller-saved. De todas formas, no sería lo correcto utilizarlo así.

Si la ejecución del manejador debe ser enteramente invisible ¿no sería necesario guardar y restaurar el registro *EFLAGS* a mano? ¿Por qué?

No es necesario ya que el valor del registro EFLAGS se guarda automáticamente en

el stack, y luego de la ejecución del manejador vuelve a su valor anterior mediante la instruccion IRET.

¿En qué stack se ejecuta la función *vga_write()?*

vga_write() se ejecuta en el stack del manejador, la misma que se está utilizando en *kmain()*.

kern2-div:

Explicar el funcionamiento exacto de:

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

¿Qué cómputo se está realizando?
Primero se guardan los valores iniciales en registros:

Registro	Valor
eax	18
ecx	0xE0
ebx	1
edx	0

La instrucción *div %4* realiza la división entre el valor en %edx:%eax y %ebx (que es lo que esta guardado en el registro 4.

Como se indico en la tabla, el valor de %edx es 0. Por lo tanto, se hace la siguiente división %eax/%ebx (18/1), y el resultado de la misma se guarda en %eax

¿De dónde sale el valor de la variable *color*?
El valor de variable color sale del valor inicial que se le dio al registro ecx (0xE0) ya que no se utilizó en la división.

¿Por qué se da valor 0 a *%edx?*
Como el dividendo es %edx:%eax, al asignarle 0 a %edx, el dividendo toma solo el valor que tiene %eax.

decls.h

Page 1/1

```
#ifndef KERN2_DECL_H
#define KERN2_DECL_H

#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#define USTACK_SIZE 4096

struct multiboot_info;

// mbinfo.c (ejercicio opcional kern2-meminfo)
void print_mbinfo(const struct multiboot_info *mbi);
bool fmt_int(uint64_t val, char *s, size_t bufsize);

// stacks.S
void two_stacks(void);

// kern2.c
void two_stacks_c(void);

// tasks.S
// Realiza una llamada a "entry" sobre el stack proporcionado.
void task_exec(uintptr_t entry, uintptr_t stack);
void task_swap(uintptr_t *esp);

// contador.c
void contador_run(void);
void round_robin(unsigned lim, uint8_t linea, char color);
void halt();

// interrupts.c
void idt_init(void);
void idt_install(uint8_t code, void (*handler)(void));
void irq_init(void);

// idt_entry.S
void divzero(void);
void breakpoint(void);
void ack_irq(void);
void timer_asm(void);
void keyboard_asm(void);

// handlers.c
void timer(void);
void keyboard(void);

// sched.c
void sched_init(void);

// funcs.S
__attribute__((regparm(3))) void vga_write2(const char *s,
                                            int8_t linea,
                                            uint8_t color);

// write.c
void vga_write(const char *s, int8_t linea, uint8_t color);

__attribute__((regparm(2))) void vga_write_cyan(const char *s, int8_t linea);

#endif
```

interrupts.h

Page 1/1

```
#ifndef INTERRUPTS_H
#define INTERRUPTS_H

#include <stdint.h>

// IDTR Register (see IA32-3A, Â§6.10 INTERRUPT DESCRIPTOR TABLE).
struct IDTR {
    uint16_t limit;  // Limit
    uint32_t base;   // Base address
} __attribute__((packed));


// Gate descriptors for interrupts (see IA32-3A, Â§6.11 IDT DESCRIPTORS).
struct Gate {
    unsigned off_15_0 : 16;   // Low 16 bits of offset in segment.
    unsigned segment : 16;    // Segment selector (always KSEG_CODE).
    unsigned reserved1 : 8;   // Unused/reserved.
    unsigned type : 4;        // Type (always STS_IG32).
    unsigned system : 1;      // System bit (must be 0).
    unsigned rpl : 2;         // Requestor Privilege Level (always 0).
    unsigned present : 1;     // Present (must be 1 if active).
    unsigned off_31_16 : 16;  // High bits of offset in segment.
};


// x86 exception numbers (see IA32-3A, Â§6.3 SOURCES OF INTERRUPTS).
enum Exception {
    T_DIVIDE = 0,     // Divide error
    T_DEBUG = 1,      // Debug exception
    T_NMI = 2,        // Non-maskable interrupt
    T_BRKPT = 3,      // Breakpoint
    T_OFLOW = 4,      // Overflow
    T_BOUND = 5,      // Bounds check
    T_ILLOP = 6,      // Illegal opcode
    T_DEVICE = 7,     // Device not available
    T_DBLFLT = 8,     // Double fault
    /* T_COPROC */    // Reserved (not generated by recent processors)
    T_TSS = 10,       // Invalid task switch segment
    T_SEGNP = 11,     // Segment not present
    T_STACK = 12,     // Stack exception
    T_GPFLT = 13,     // General protection fault
    T_PGFLT = 14,     // Page fault
    /* T_RES */       // Reserved
    T_FPERR = 16,     // Floating point error
    T_ALIGN = 17,     // Aligment check
    T_MCHK = 18,      // Machine check
    T_SIMDERR = 19,   // SIMD floating point error
};

// kern2 interrupt numbers: we map IRQ0 to 32, and count from there.
enum Interrupt {
    T_TIMER = 32,      // IRQ0
    T_KEYBOARD = 33,   // IRQ1
};

#endif
```

multiboot.h

Page 1/4

```
/* multiboot.h - Multiboot header file. */
/* Copyright (C) 1999,2003,2007,2008,2009  Free Software Foundation, Inc.
 *
 *  Permission is hereby granted, free of charge, to any person obtaining a copy
 *  of this software and associated documentation files (the "Software"), to
 *  deal in the Software without restriction, including without limitation the
 *  rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
 *  sell copies of the Software, and to permit persons to whom the Software is
 *  furnished to do so, subject to the following conditions:
 *
 *  The above copyright notice and this permission notice shall be included in
 *  all copies or substantial portions of the Software.
 *
 *  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 *  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 *  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL ANY
 *  DEVELOPER OR DISTRIBUTOR BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
,
 *  WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR
 *  IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE
.
 */

/*
 * Changes 2017-07-31 (dato@fi.uba.ar):
 *   - use gcc-defined __ASSEMBLER__ guard, instead of ASM_FILE.
 *   - include <stdint.h> to replace manual typedefs.
 */

#ifndef MULTIBOOT_HEADER
#define MULTIBOOT_HEADER 1

/* How many bytes from the start of the file we search for the header. */
#define MULTIBOOT_SEARCH 8192

/* The magic field should contain this. */
#define MULTIBOOT_HEADER_MAGIC 0x1BADB002

/* This should be in %eax. */
#define MULTIBOOT_BOOTLOADER_MAGIC 0x2BADB002

/* The bits in the required part of flags field we don't support. */
#define MULTIBOOT_UNSUPPORTED 0x0000fffc

/* Alignment of multiboot modules. */
#define MULTIBOOT_MOD_ALIGN 0x00001000

/* Alignment of the multiboot info structure. */
#define MULTIBOOT_INFO_ALIGN 0x00000004

/* Flags set in the 'flags' member of the multiboot header. */

/* Align all boot modules on i386 page (4KB) boundaries. */
#define MULTIBOOT_PAGE_ALIGN 0x00000001

/* Must pass memory information to OS. */
#define MULTIBOOT_MEMORY_INFO 0x00000002

/* Must pass video information to OS. */
#define MULTIBOOT_VIDEO_MODE 0x00000004

/* This flag indicates the use of the address fields in the header. */
#define MULTIBOOT_AOUT_KLUDGE 0x00010000

/* Flags to be set in the 'flags' member of the multiboot info structure. */

```

multiboot.h

Page 2/4

```
/* is there basic lower/upper memory information? */
#define MULTIBOOT_INFO_MEMORY 0x00000001
/* is there a boot device set? */
#define MULTIBOOT_INFO_BOOTDEV 0x00000002
/* is the command-line defined? */
#define MULTIBOOT_INFO_CMDLINE 0x00000004
/* are there modules to do something with? */
#define MULTIBOOT_INFO_MODS 0x00000008

/* These next two are mutually exclusive */

/* is there a symbol table loaded? */
#define MULTIBOOT_INFO_AOUT_SYMS 0x00000010
/* is there an ELF section header table? */
#define MULTIBOOT_INFO_ELF_SHDR 0X00000020

/* is there a full memory map? */
#define MULTIBOOT_INFO_MEM_MAP 0x00000040

/* Is there drive info? */
#define MULTIBOOT_INFO_DRIVE_INFO 0x00000080

/* Is there a config table? */
#define MULTIBOOT_INFO_CONFIG_TABLE 0x00000100

/* Is there a boot loader name? */
#define MULTIBOOT_INFO_BOOT_LOADER_NAME 0x00000200

/* Is there a APM table? */
#define MULTIBOOT_INFO_APM_TABLE 0x00000400

/* Is there video information? */
#define MULTIBOOT_INFO_VIDEO_INFO 0x00000800

#ifndef __ASSEMBLER__

#include <stdint.h>

struct multiboot_header {
    /* Must be MULTIBOOT_MAGIC - see above. */
    uint32_t magic;

    /* Feature flags. */
    uint32_t flags;

    /* The above fields plus this one must equal 0 mod 2^32. */
    uint32_t checksum;

    /* These are only valid if MULTIBOOT_AOUT_KLUDGE is set. */
    uint32_t header_addr;
    uint32_t load_addr;
    uint32_t load_end_addr;
    uint32_t bss_end_addr;
    uint32_t entry_addr;

    /* These are only valid if MULTIBOOT_VIDEO_MODE is set. */
    uint32_t mode_type;
    uint32_t width;
    uint32_t height;
    uint32_t depth;
};

/* The symbol table for a.out. */
struct multiboot_aout_symbol_table {
    uint32_t tabsize;
    uint32_t strsize;
```

multiboot.h


```
131     uint32_t addr;
132     uint32_t reserved;
133 };
134 typedef struct multiboot_aout_symbol_table multiboot_aout_symbol_table_t;
135
136 /* The section header table for ELF. */
137 struct multiboot_elf_section_header_table {
138     uint32_t num;
139     uint32_t size;
140     uint32_t addr;
141     uint32_t shndx;
142 };
143 typedef struct multiboot_elf_section_header_table
144     multiboot_elf_section_header_table_t;
145
146 struct multiboot_info {
147     /* Multiboot info version number */
148     uint32_t flags;
149
150     /* Available memory from BIOS */
151     uint32_t mem_lower;
152     uint32_t mem_upper;
153
154     /* "root" partition */
155     uint32_t boot_device;
156
157     /* Kernel command line */
158     uint32_t cmdline;
159
160     /* Boot-Module list */
161     uint32_t mods_count;
162     uint32_t mods_addr;
163
164     union {
165         multiboot_aout_symbol_table_t aout_sym;
166         multiboot_elf_section_header_table_t elf_sec;
167     } u;
168
169     /* Memory Mapping buffer */
170     uint32_t mmap_length;
171     uint32_t mmap_addr;
172
173     /* Drive Info buffer */
174     uint32_t drives_length;
175     uint32_t drives_addr;
176
177     /* ROM configuration table */
178     uint32_t config_table;
179
180     /* Boot Loader Name */
181     uint32_t boot_loader_name;
182
183     /* APM table */
184     uint32_t apm_table;
185
186     /* Video */
187     uint32_t vbe_control_info;
188     uint32_t vbe_mode_info;
189     uint16_t vbe_mode;
190     uint16_t vbe_interface_seg;
191     uint16_t vbe_interface_off;
192     uint16_t vbe_interface_len;
193 };
194 typedef struct multiboot_info multiboot_info_t;
195
196 struct multiboot_mmap_entry {
```

multiboot.h


```
197     uint32_t size;
198     uint64_t addr;
199     uint64_t len;
200 #define MULTIBOOT_MEMORY_AVAILABLE 1
201 #define MULTIBOOT_MEMORY_RESERVED 2
202     uint32_t type;
203 } __attribute__((packed));
204 typedef struct multiboot_mmap_entry multiboot_memory_map_t;
205
206 struct multiboot_mod_list {
207     /* the memory used goes from bytes 'mod_start' to 'mod_end-1' inclusive */
208     uint32_t mod_start;
209     uint32_t mod_end;
210
211     /* Module command line */
212     uint32_t cmdline;
213
214     /* padding to take it to 16 bytes (must be zero) */
215     uint32_t pad;
216 };
217 typedef struct multiboot_mod_list multiboot_module_t;
218
219 #endif /* ! __ASSEMBLER__ */
220
221 #endif /* ! MULTIBOOT_HEADER */
```

sched.h

Page 1/1

```c
#ifndef KERN2_SCHED_H
#define KERN2_SCHED_H

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[USTACK_SIZE];
    enum TaskStatus status;
    struct TaskFrame *frame;
};

void sched_init();

void spawn(void (*entry)(void));

void sched(struct TaskFrame *tf);

void kill_current_task();

#endif //KERN2_SCHED_H
```

contador.c

Page 1/2

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

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

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;
    task_swap(&tmp);
}


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

static void contador(unsigned lim, uint8_t linea, char color,const bool round_robin_mode) {
    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;
        }

        if (!round_robin_mode)
            yield();
    }
    if (round_robin_mode)
        kill_current_task();
}

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

void round_robin(unsigned lim, uint8_t linea, char color) {
    contador(lim, linea, color, true);
```

contador.c


```
}

void contador_run() {
    // Configurar stack1 y stack2 con los valores apropiados.
    uintptr_t *a = (uintptr_t*) stack1 + USTACK_SIZE;
    a -= 3;
    a[2] = 0x2F;
    a[1] = 0;
    a[0] = 200;


    uintptr_t *b = (uintptr_t*) stack2 + USTACK_SIZE;
    b -= 3;
    b[2] = 0x4F;
    b[1] = 1;
    b[0] = 100;

    // Llamada a exit al finalizar contador_yield
    *(--b) = (uintptr_t)exit;

    // Simulo que el primer swap no es el primero
    *(--b) = (uintptr_t)contador_yield;

    // Seteo los registros calle save a 0
    *(--b) = 0;
    *(--b) = 0;
    *(--b) = 0;
    *(--b) = 0;


    // Actualizar la variable estÃ¡tica âM-^@M-^XespâM-^@M-^Y para que apunte
    // al del segundo contador.
    esp = (uintptr_t) b;

    // Lanzar el primer contador con task_exec.
    task_exec((uintptr_t) contador_yield, (uintptr_t) a);
}
```

handlers.c


```
#include "decls.h"

#define RELEASE_CODE 0x80
#define PROMPT_CURSOR '_'
#define MAX_SIZE 81
#define SPACE ' '
#define LEFT_SHIFT 42
#define RIGHT_SHIFT 54
#define BACKSPACE '\b'
#define SIMPLE_QUOTATION_MARK '\''
#define ENTER '\n'
#define ENIE 164

/**
 * Handler para el timer (IRQ0). Escribe un carÃ¡cter cada segundo.
 */
static const uint8_t hz_ratio = 18;  // Default IRQ0 freq (18.222 Hz).

void timer() {
    static char chars[MAX_SIZE];
    static unsigned ticks;
    static int8_t line = 21;
    static uint8_t idx = 0;

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

    if (idx >= sizeof(chars) - 1) {
        line++;
        idx = 0;
    }
}

/**
 * Mapa de "scancodes" a caracteres ASCII en un teclado QWERTY.
 */
static unsigned char klayout[128] = {
    //0-9
    0,   0,   '1', '2', '3', '4', '5', '6', '7', '8',
    //10-19
    '9', '0', 0,   0, BACKSPACE, 0, 'q', 'w', 'e', 'r',
    //20-29
    't', 'y', 'u', 'i', 'o', 'p', '[', ']',   ENTER,   0,
    //30-40
    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ENIE, SIMPLE_QUOTATION_MARK,
    //41-50
    0, 0,   0,   'z', 'x', 'c', 'v', 'b', 'n', 'm',
    //51-60
    ',', '.', '_', 0, 0, 0, SPACE, 0,0,0};

static const uint8_t KBD_PORT = 0x60;

static bool is_shift_pressed(uint8_t scancode) {

    bool released = scancode & RELEASE_CODE;
    scancode &= ~RELEASE_CODE;

    static bool pressed;
    if (scancode == RIGHT_SHIFT || scancode == LEFT_SHIFT) {
        pressed = !released;
    }
    return pressed;
}
```

handlers.c

Page 2/2

```
67
68  /**
69   * Handler para el teclado (IRQ1).
70   *
71   * Imprime la letra correspondiente por pantalla.
72   */
73  void keyboard() {
74      uint8_t code;
75      static uint8_t actual_index = 0;
76      static unsigned char kbd_entry_line[MAX_SIZE];
77
78      asm volatile("inb %1,%0" : "=a"(code) : "n"(KBD_PORT));
79
80      int8_t offset = is_shift_pressed(code)? -32 : 0;
81
82      if (code >= sizeof(klayout) || !klayout[code])
83          return;
84
85      if (klayout[code] == BACKSPACE) {
86          if (!actual_index)
87              actual_index=1;
88          kbd_entry_line[actual_index] = SPACE;
89          kbd_entry_line[--actual_index] = PROMPT_CURSOR;
90      } else {
91          kbd_entry_line[actual_index] = klayout[code] + offset;
92          kbd_entry_line[++actual_index] = PROMPT_CURSOR;
93      }
94      vga_write((char*)kbd_entry_line, 19, 0x0A);
95
96  }
```

interrupts.c

Page 1/2

```
1   #include "decls.h"
2   #include "interrupts.h"
3
4   #define IDT_SIZE 256
5   static struct IDTR idtr;
6   static struct Gate idt[IDT_SIZE];
7
8   // Multiboot siempre define "8" como el segmento de cÃ³digo.
9   // (Ver campo CS en `info registers` de QEMU.)
10  static const uint8_t KSEG_CODE = 8;
11
12  // Identificador de "Interrupt gate de 32 bits" (ver IA32-3A,
13  // tabla 6-2: IDT Gate Descriptors).
14  static const uint8_t STS_IG32 = 0xE;
15
16  #define outb(port, data) \
17          asm("outb %b0,%w1" : : "a"(data), "d"(port));
18
19  static void irq_remap() {
20      outb(0x20, 0x11);
21      outb(0xA0, 0x11);
22      outb(0x21, 0x20);
23      outb(0xA1, 0x28);
24      outb(0x21, 0x04);
25      outb(0xA1, 0x02);
26      outb(0x21, 0x01);
27      outb(0xA1, 0x01);
28      outb(0x21, 0x0);
29      outb(0xA1, 0x0);
30  }
31
32  void idt_install(uint8_t n, void (*handler)(void)) {
33      uintptr_t addr = (uintptr_t) handler;
34
35      idt[n].rpl = 0;
36      idt[n].type = STS_IG32;
37      idt[n].segment = KSEG_CODE;
38
39      idt[n].off_15_0 = addr & 0xFF;
40      idt[n].off_31_16 = addr >> 16;
41
42      idt[n].present = 1;
43  }
44
45  void idt_init() {
46      // (1) Instalar manejadores ("interrupt service routines").
47      idt_install(T_BRKPT, breakpoint);
48
49      // (2) Configurar ubicaciÃ³n de la IDT.
50      idtr.base = (uintptr_t) idt;
51      idtr.limit = 8 * IDT_SIZE - 1 ;
52
53      // (3) Activar IDT.
54      asm("lidt %0" : : "m"(idtr));
55  }
56
57  void irq_init() {
58      // (1) Redefinir cÃ³digos para IRQs.
59      irq_remap();
60
61      // (2) Instalar manejadores.
62      idt_install(T_TIMER, timer_asm);
63      idt_install(T_KEYBOARD, keyboard_asm);
64      idt_install(T_DIVIDE, divzero);
65
66      // (3) Habilitar interrupciones.
```

interrupts.c

Page 2/2

```
67      asm("sti");
68  }
```

kern2.c

Page 1/2

```
1   #include "decls.h"
2   #include "multiboot.h"
3   #include "lib/string.h"
4   #include "sched.h"
5
6   static void contador1() {
7       round_robin(600, 6, 0x3E);
8   }
9
10  static void contador2() {
11      round_robin(200, 7, 0x2A);
12  }
13
14  static void contador3() {
15      round_robin(400, 8, 0x1F);
16  }
17
18  void contador_spawn() {
19      spawn(contador1);
20      spawn(contador2);
21      spawn(contador3);
22  }
23
24  static uint8_t stack1[USTACK_SIZE] __attribute__((aligned(4096)));
25  static uint8_t stack2[USTACK_SIZE] __attribute__((aligned(4096)));
26
27  void two_stacks_c() {
28      // Inicializar al *tope* de cada pila.
29      uintptr_t *a = (uintptr_t*) stack1 + USTACK_SIZE;
30      uintptr_t *b = (uintptr_t*) stack2 + USTACK_SIZE;
31
32      // Preparar, en stack1, la llamada:
33
34      *(a--) = 0x57;
35      *(a--) = 15;
36      *(a) = (uintptr_t) "vga_write() from stack1";
37
38      // Preparar, en s2, la llamada:
39
40      b -= 3;
41      b[2] = 0xD0;
42      b[1] = 16;
43      b[0] = (uintptr_t) "vga_write() from stack2";
44
45      task_exec((uintptr_t) vga_write, (uintptr_t) a);
46
47      asm("movl %0, %%esp; call *%1; movl %%ebp, %%esp"
48      :
49      : "r"(b), "r"(vga_write));
50  }
51
52  void kmain(const multiboot_info_t *mbi) {
53      vga_write("kern2 loading.....", 8, 0x70);
54
55      if (mbi->flags) {
56          char buf[256] = "cmdline: ";
57          char *cmdline = (void *) mbi->cmdline;
58          strlcat(buf, cmdline, sizeof(buf));
59          vga_write(buf, 9, 0x07);
60
61          print_mbinfo(mbi);
62      }
63
64      two_stacks();
65      two_stacks_c();
66
```

kern2.c


```
67      contador_run();
68
69      contador_spawn();
70      sched_init();
71
72      idt_init();
73      irq_init();
74      asm("int3");
75
76      int8_t linea;
77      uint8_t color;
78      asm("div %4"
79          : "=a"(linea), "=c"(color)
80          : "0"(18), "1"(0xE0), "b"(0), "d"(0));
81
82      vga_write2("Funciona vga_write2?", linea, color);
83
84      asm("hlt");
85  }
86
87
88
```

mbinfo.c


```
1   #include "decls.h"
2   #include "lib/string.h"
3   #include "multiboot.h"
4
5   #define KB_TO_MB_SHIFT 10 // 1KB*2^10->1MB
6
7   void print_mbinfo(const struct multiboot_info *mbi){
8       char mem[256] = "Physical memory: ";
9       char tmp[64] = {0};
10
11      uint32_t total_size = mbi->mem_upper - mbi->mem_lower;
12      if (fmt_int(total_size>>KB_TO_MB_SHIFT, tmp, sizeof tmp)) {
13          strlcat(mem, tmp, sizeof mem);
14          strlcat(mem, "MiB total", sizeof mem);
15      }
16      memset(tmp,0, sizeof(tmp));
17      if (fmt_int(mbi->mem_lower, tmp, sizeof tmp)) {
18          strlcat(mem, " (", sizeof mem);
19          strlcat(mem, tmp, sizeof mem);
20          strlcat(mem, " KiB base", sizeof mem);
21      }
22
23      memset(tmp,0, sizeof(tmp));
24      if (fmt_int(mbi->mem_upper, tmp, sizeof tmp)) {
25          strlcat(mem, ", ", sizeof mem);
26          strlcat(mem, tmp, sizeof mem);
27          strlcat(mem, " KiB extended) ", sizeof mem);
28      }
29
30      vga_write(mem, 10, 0x07);
31  }
```

sched.c

Page 1/2

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

#define MAX_TASK 5
#define IF_FLAG 0x200

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

bool getFreeTask(struct Task **new_task) {
    size_t i=0;
    bool new_free_task=false;
    while ((i<MAX_TASK) && !new_free_task){
        if (Tasks[i].status == FREE){
            (*new_task) = &Tasks[i];
            new_free_task=true;
            }
        i++;
    }
    return new_free_task;
}

void sched_init() {
    size_t  i = 0;
    while (i<MAX_TASK && Tasks[i].status != READY){
        i++;
    }
    if (Tasks[i].status == READY) {
        current = &Tasks[i];
        current->status = RUNNING;
    }
}

void initialize_task(struct Task **task, void (*entry)(void)) {
    (*task)->status = READY;

    uint8_t* stack = &(*task)->stack[USTACK_SIZE] - sizeof(struct TaskFrame);
    (*task)->frame = (struct TaskFrame *)stack;

    (*task)->frame->ebp = 0;
    (*task)->frame->esp = 0;
    (*task)->frame->eax = 0;
    (*task)->frame->ebx = 0;
    (*task)->frame->ecx = 0;
    (*task)->frame->edx = 0;
    (*task)->frame->edi = 0;
    (*task)->frame->esi = 0;
    (*task)->frame->eip = (uint32_t)entry;
    (*task)->frame->cs = 0x8;
    (*task)->frame->eflags = IF_FLAG;
}

void spawn(void (*entry)(void)) {
    struct Task* new_task = NULL;
    bool success = getFreeTask(&new_task);
    if (!success)
        return;
    initialize_task(&new_task, entry);
}


static bool first_call = true;

void sched(struct TaskFrame *tf) {
    bool ready_task_found = false;
    struct Task *previous = current;
```

sched.c

Page 2/2

```c

    size_t task_index = 0;
    while ((task_index < MAX_TASK) && (&Tasks[task_index] != previous)){
        task_index++;
    }

    previous->status = READY;

    while (!ready_task_found){
        task_index = (task_index+1) % MAX_TASK;

        if (Tasks[task_index].status == READY){
            ready_task_found = true;

            if (!first_call){
                previous->frame = tf;
            } else
                first_call = false;
            current = &Tasks[task_index];
            current->status = RUNNING;

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

void kill_current_task(){
    current->status = DYING;
    halt();
}
```

write.c

Page 1/1

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

#define VGABUF ((volatile char *) 0xB8000)
#define ROWS 25 // numero de filas de la pantalla
#define COLUMNS 80 // numero de columnas de la pantalla

static size_t int_width(uint64_t val) {
    size_t width = 0;
    while (val>0){
        val/=10;
        width++;
    }
    return width;
}

// Escribe en âM-^@M-^XsâM-^@M-^Y el valor de âM-^@M-^XvalâM-^@M-^Y en base 10 s
i su anchura
// es menor que âM-^@M-^XbufsizeâM-^@M-^Y. En ese caso devuelve true, caso de
// no haber espacio suficiente no hace nada y devuelve false.
bool fmt_int(uint64_t val, char *s, size_t bufsize) {
    size_t l = int_width(val);

    if (l >= bufsize)  // Pregunta: Â¿por quÃ© no "l > bufsize"?
                       // Respuesta: para agregar el \0
        return false;

    for (size_t i = l; i > 0; i--) {
        char ascii_digit = '0'+val %10;
        s[i-1]= ascii_digit;
        val/=10;
    }

    s[l]='\0';
    return true;
}

void vga_write(const char *s, int8_t linea, uint8_t color) {
    if (linea < 0) {
        linea = ROWS + linea;
    }

    volatile char* buff = VGABUF + linea * COLUMNS * 2;
    while (*s != '\0') {
        *buff++ = *s++;
        *buff++ = color;
    }

}

void __attribute__((regparm(2)))
vga_write_cyan(const char *s, int8_t linea) {
    vga_write(s, linea, 0xB0);
}
```

boot.S

Page 1/1

```
#include "multiboot.h"

#define KSTACK_SIZE 8192

.align 4
multiboot:
    .long MULTIBOOT_HEADER_MAGIC
    .long 0
    .long -(MULTIBOOT_HEADER_MAGIC)

.globl _start
_start:
    // Paso 1: Configurar el stack antes de llamar a kmain.
    movl $0, %ebp
    movl $kstack_top, %esp
    push %ebp

    // Paso 2: pasar la informaciÃ³n multiboot a kmain. Si el
    // kernel no arrancÃ³ vÃ-a Multiboot, se debe pasar NULL.
    // Usar una instrucciÃ³n de comparaciÃ³n (TEST o CMP) para
    // comparar con MULTIBOOT_BOOTLOADER_MAGIC, pero no usar
    // un salto a continuaciÃ³n, sino una instrucciÃ³n CMOVcc
    // (copia condicional).
    movl $0, %ecx
    cmp $MULTIBOOT_BOOTLOADER_MAGIC, %eax
    cmove %ebx, %ecx
    push %ecx

    call kmain

.globl halt
halt:
    hlt
    jmp halt

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

funcs.S

Page 1/1

```
.text

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

    push %ecx
    push %edx
    push %eax
    call vga_write

    leave
    ret
```

idt_entry.S

Page 1/2

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

.globl ack_irq
ack_irq:
    movl $ACK_IRQ, %eax
    outb %al, $PIC1
    iret

.globl breakpoint
breakpoint:
    // (1) Guardar registros.
    pusha
    // (2) Preparar argumentos de la llamada.
    // vga_write2("Hello, breakpoint", 14, 0xB0)
    movl $0xB0, %ecx
    movl $14, %edx
    movl $breakpoint_msg, %eax
    // (3) Invocar a vga_write2()
    call vga_write2
    // (4) Restaurar registros.
    popa
    // (5) Finalizar ejecuciÃ³n del manejador.
    iret

.globl timer_asm
timer_asm:
    // Guardar registros.
    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 volviÃ³ de sched)
    addl $4, %esp
    popa
    iret

.globl keyboard_asm
keyboard_asm:
    pusha
    call keyboard

    popa
    jmp ack_irq

.globl divzero
divzero:
    // (1) Guardar registros.
    add $1, %ebx
    push %eax
    push %ecx
    push %edx

    // (2) Preparar argumentos de la llamada.
    //vga_write_cyan("Se divide por ++ebx", 17);

    movl $17, %edx
    movl $divzero_msg, %eax

```

idt_entry.S


```
67      // (3) Invocar a vga_write_cyan()
68      call vga_write_cyan
69
70      // (4) Restaurar registros.
71      pop %edx
72      pop %ecx
73      pop %eax
74
75      // (5) Finalizar ejecuciÃ³n del manejador.
76      iret
77
78  .data
79  breakpoint_msg:
80      .asciz "Hello, breakpoint"
81
82  divzero_msg:
83      .asciz "Se divide por ++ebx"
84
```

stacks.S


```
1   #define USTACK_SIZE 4096
2
3   .data
4           .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  msg1:
15          .asciz "vga_write() from stack1"
16  msg2:
17          .asciz "vga_write() from stack2"
18
19  .text
20  .globl two_stacks
21  two_stacks:
22          // PreÃ¡mbulo estÃ¡ndar
23          push %ebp
24          movl %esp, %ebp
25          push %ebx
26
27          // Registros para apuntar a stack1 y stack2.
28          mov $stack1_top, %eax
29          mov $stack2_top, %ebx
30
31          // Cargar argumentos a ambos stacks en paralelo. Ayuda:
32          // usar offsets respecto a %eax ($stack1_top), y lo mismo
33          // para el registro usado para stack2_top.
34          movl $0x17, -4(%eax)
35          movl $0x90, -4(%ebx)
36
37          movl $12,  -8(%eax)
38          movl $13, -8(%ebx)
39
40          movl $msg1,  -12(%eax)
41          movl $msg2, -12(%ebx)
42
43          // Realizar primera llamada con stack1. Ayuda: usar LEA
44          // con el mismo offset que los Ãºltimos MOV para calcular
45          // la direcciÃ³n deseada de ESP.
46          leal -12(%eax), %esp
47          call vga_write
48
49          // Restaurar stack original. Â¿Es %ebp suficiente?
50          movl %ebp, %esp
51
52          // Realizar segunda llamada con stack2.
53          leal -12(%ebx), %esp
54          call vga_write
55
56          // Restaurar registros callee-saved, si se usaron.
57          pop %ebx
58
59          leave
60          ret
```

tasks.S


```
1  .data
2
3  .text
4  .globl task_exec
5  task_exec:
6
7      push %ebp
8      movl %esp, %ebp
9
10     movl 8(%ebp), %eax
11     movl 12(%ebp), %esp
12     call *%eax
13
14     leave
15     ret
16
17 .globl task_swap
18 task_swap:
19     push %ebp
20     push %ebx
21     push %edi
22     push %esi
23
24     movl 20(%esp), %eax
25
26     movl %esp, %ecx
27     movl (%eax), %esp
28     movl %ecx, (%eax)
29
30     pop %esi
31     pop %edi
32     pop %ebx
33     pop %ebp
34
35     ret
```

Table of Content