### 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.

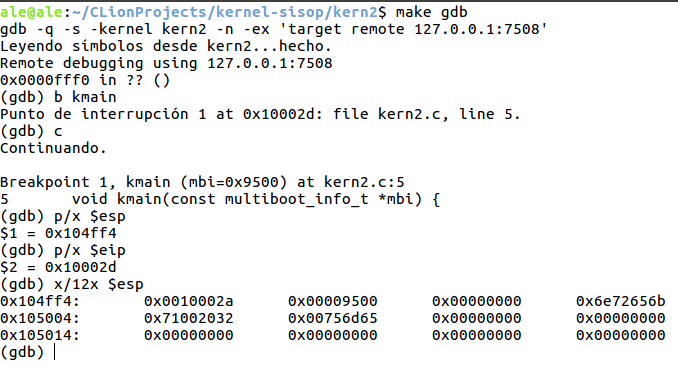
Para alinear el stack a 4KB se puede hacer:

.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.**



### 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-meminfo

El código realizado para esta sección se encuentra en la próxima sección.

### kern2-regcall

### FALTA objump -d

### 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

**SESION GDB VERSION A**

ale@ale:~/CLionProjects/kernel-sisop/kern2/parte 3$ 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

-------------------------------------

### SESION GDB VERSION B

ale@ale:~/CLionProjects/kernel-sisop/kern2/parte 3$ 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

---------------------------------------------

Las dos secciones marcadas es donde cambian pareciera

**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:**

OPCION 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.

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

OPCION 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.