Alumnos: Matias Feld y Agustin Zorzano

1.0 Creación de stacks en el kernel

1.1 kern2-stack

1. ***Explicar: ¿qué significa "estar alineado"?***

Estar alineado significa que la dirección de memoria del primer byte es múltiplo de la alineación.

Por ejemplo, alineado a 4 significa que la dirección de memoria del primer byte es múltiplo de 4, por lo que estará en direcciones tales como 0, 4, 8, 12, 16, 32, etc.

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

unsigned char kstack[8192] __attribute__((aligned(4)));

3. ***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 size, fill" inicializa "size" bytes con el valor fill. Si se omite fill, entonces se rellenará con ceros.

En el caso de c, al crearse el stack no se rellena con nada, sino que tiene el valor que tenía la memoria en ese momento.

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.***

.align n → alinea el stack a n bytes
.p2align n → alinea el stack a 2^n bytes

Para alinear 4Kb se puede usar .align 4096 o .p2align 12

5. ***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.***

```
(gdb) p $esp
$1 = (void *) 0x104ff4

(gdb) p $eip
$2 = (void (*)()) 0x100200 <kmain>
```

Valores del stack

```
(gdb) x/100x $esp
0x104ff4:   0x0010002a   0x00009500   0x00000000   0x6e72656b
0x105004:   0x71002032   0x00756d65   0x00000000   0x00000000
```

1.2 kern2-cmdline

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

Para obtener un comportamiento similar hay que hacer

strncat(dest , src, sizeof(dest) – strlen(dest) – 1);

Esto es porque strncat espera recibir la cantidad de bytes disponibles en el buffer de destino, sin incluir el \0.

```
int main(){
   char mem[256] = "Physical Memory: ";

   size_t size = sizeof(mem) - strlen(mem) - 1;

   strncat(mem, "126 MiB total.", size);

   printf("Con strncat: \n\n%s\n", mem);
}
```

Imprime correctamente: ‘Physical Memory: 126 MiB total.’

2. ***Explicar cómo se comporta strlcat(3) si, erróneamente, se declarase buf con tamaño 12. ¿Introduce algún error el código?***

Solamente guardará en buf los caracteres que entren (contando el '\0'). Los demás no se tienen en cuenta. Esto no producirá ningún error, pero al imprimir buf se imprimirán menos caracteres de lo esperado.

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

Imprime:
sizeof buf = 256
sizeof buf = 8

Se imprimen dos líneas distintas porque en la función void printf_sizeof_buf(char buf[256]) se recibe un puntero char (aunque diga char[256]).

Por lo tanto, en la segunda línea se imprime el tamaño de un puntero. Esto significa que recibir como parámetro char *buf o char buf[] en una función es lo mismo (es un puntero).

En la primer línea se imprime la cantidad de bytes del buffer, porque no es un puntero, se creó en el stack.

2.0 Concurrencia cooperativa

2.1 kern2-regcall

1. ***Mostrar con objdump -d el código generado por GCC para la siguiente llamada a vga_write2() desde la función principal***

```
00100104 <kmain>:
100104:        55                      push %ebp
100105:        89 e5                   mov %esp,%ebp
100107:        83 ec 0c                sub $0xc,%esp
10010a:        6a 70                   push $0x70
10010c:6a 08                    push $0x8
10010e:        68 31 09 10 00  push $0x100931
100113:        e8 23 00 00 00  call 10013b <vga_write>
100118:        e8 e3 fe ff ff          call 100000 <two_stacks>
10011d:        e8 7c ff ff ff          call 10009e <two_stacks_c>
100122:        b9 e0 00 00 00  mov $0xe0,%ecx
100127:        ba 12 00 00 00  mov $0x12,%edx
10012c:        b8 4c 09 10 00  mov $0x10094c,%eax
100131:        e8 5b ff ff ff          call 100091 <vga_write2>
100136:        83 c4 10                add $0x10,%esp
100139:        c9                      leave
10013a:        c3                      ret
```

Se ve que, antes del call, guarda los parámetros en los registros y no en la pila (en negrita).

2.2 kern2-swap

1. ***Explicar, para el stack de cada contador, cuántas posiciones se asignan, y qué representa cada una.***

Para el primer contador se asignan 3 posiciones:

```
*(--a) = 0x2F;          //Color
*(--a) = 0;             //Línea
*(--a) = 100;           //Límite
```

Para el segundo contador se asignan 9 posiciones:

```
*(--b) = 0x4F;          //Color
*(--b) = 1;             //Línea
*(--b) = 100;           //Límite

//Dirección de retorno de la función contador_yield()
*(--b) = 0;

//Dirección de retorno de la función task_swap() en la primer iteración
*(--b) = (uintptr_t)contador_yield;

//Registros calle-saved (ebp, ebx, esi, edi)
*(--b) = 0;
*(--b) = 0;
*(--b) = 0;
*(--b) = 0;
```

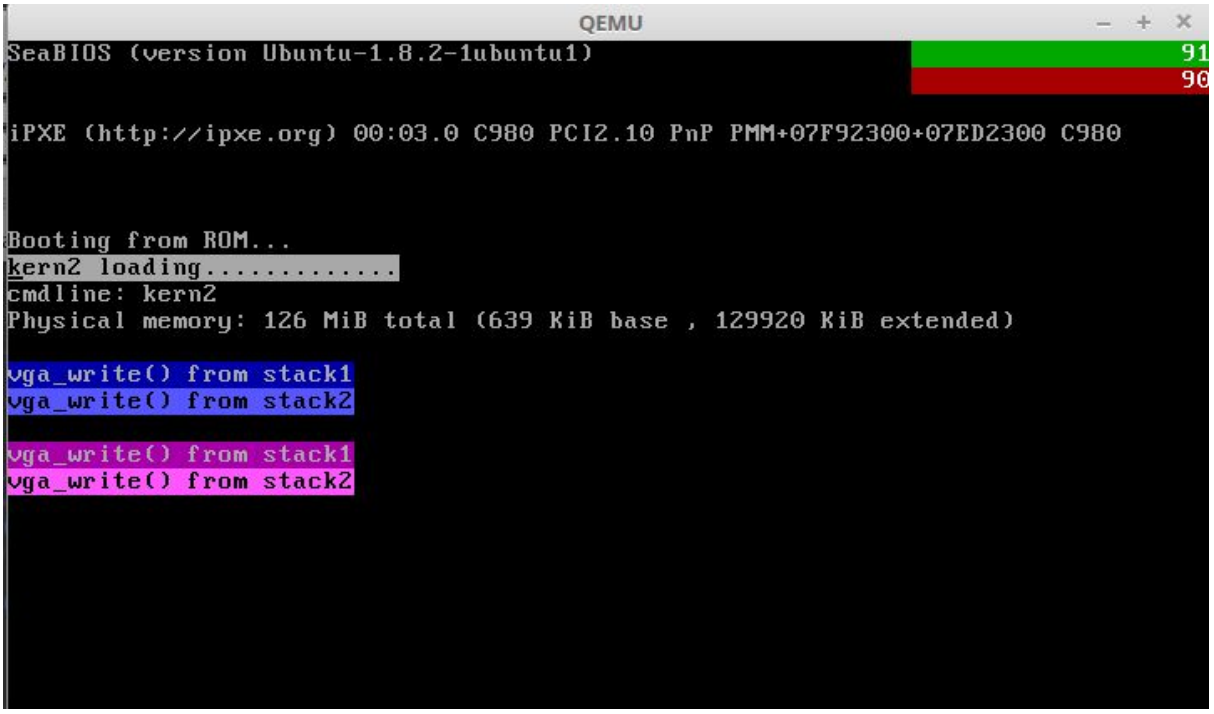
2.3 kern2-exit

1. ***Reducir ahora el número de iteraciones del segundo contador, y describir qué tipo de error ocurre.***

Si al primer contador se le pone limite 100, y al segundo 90, lo que ocurre es lo siguiente:

Cuando el primer contador le vuelva a dar el paso al segundo, este va a salir del ciclo terminando con la función, y va a saltar a la dirección de retorno, que en este caso es 0.

Por eso, el kernel queda tildado y no avanza, porque la cpu se queda en la instrucción en la dirección 0, sea cual sea.



Como se puede ver, el rojo no le cede devuelta el control al verde porque ya salió de la función. Además, no se imprime la última línea (“Funciona vga_write2?”) porque nunca llega a esa parte del código, que llegaría desde el contador verde, porque fue el ejecutado desde contador_run().

3.0 Interrupciones

3.1 kern2-idt

1. ***¿Cuántos bytes ocupa una entrada en la IDT?***

Cada entrada ocupa 64 bits o 8 bytes.

2. ***¿Cuántas entradas como máximo puede albergar la IDT?***

Puede contener como máximo 256 entradas, es decir que se indexa con 8 bits.

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

El campo limit es el tamaño total de la IDT. Como máximo puede haber 256 entradas de 8 bytes, por lo que limit puede ser como máximo 8 * 256 - 1 = 2047.

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

limit = 8 * 64 – 1 = 511

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

Las interrupciones ocurren por señales del hardware, como por ejemplo los periféricos que le avisan al procesador algo. También pueden ser generadas por software explícitamente, llamando a la instrucción INT xx, por ejemplo para syscalls.

Las excepciones ocurren cuando el procesador encuentra un error al ejecutar una instrucción. Por ejemplo, realizar una división por 0, o acceder a una dirección de memoria inválida.

3.2 kern2-isr

1. ***Una sesión de GDB que muestre el estado de la pila antes, durante y después de la ejecución del manejador.***

➔ VERSIÓN A (IRET):

   A. ***Poner un breakpoint en la función idt_init() y, una vez dentro, finalizar su ejecución con el comando de GDB finish. La ejecución debería haberse detenido en la misma instrucción int3. Mostrar:***

```
Continuing.

Breakpoint 1, idt_init () at interrupts.c:22
22                  idt_install(T_BRKPT, breakpoint);
1: x/i $pc
=> 0x100348 <idt_init+3>:         push $0x10007d

(gdb) finish
Run till exit from #0 idt_init () at interrupts.c:22
kmain (mbi=0x9500) at kern2.c:25
25          asm("int3");
1: x/i $pc
=> 0x10014e <kmain+150>:      int3     (Próxima instrucción).
```

- el valor de %esp (print $esp)

```
(gdb) print $esp
$1 = (void *) 0x107ee8
```

- el valor de (%esp) (x/xw $esp)

```
(gdb) x/xw $esp
0x107ee8:        0x6c646d63
```

- el valor de $cs

```
(gdb) print $cs
$2 = 8
```

- el resultado de print $eflags y print/x $eflags

```
(gdb) print $eflags
$3 = [ AF ]
(gdb) print/x $eflags
$4 = 0x12
```

B. ***Ejecutar la instrucción int3 mediante el comando de GDB stepi. La ejecución debería saltar directamente a la instrucción test %eax, %eax.***

```
(gdb) stepi
breakpoint () at idt_entry.S:6
6           test %eax, %eax
1: x/i $pc
=> 0x10007e <breakpoint+1>:    test %eax,%eax

(gdb) print $esp
$5 = (void *) 0x107edc
```

Avanzó de 0x107ee8 a 0x107edc, es decir 12 bytes o 3 posiciones.

```
(gdb) x/3wx $esp
0x107edc:       0x0010014f      0x00000008      0x00000012

(gdb) print $eflags
$6 = [ AF ]
```

Los valores de la pila son, en orden, el valor del Instruction pointer siguiente al que tenía antes de entrar en la excepción (int 3), el valor de $cs y el valor de $eflags.

C. ***Avanzar una instrucción más con stepi, ejecutando la instrucción TEST. Mostrar, como anteriormente, el valor del registro EFLAGS.***

```
(gdb) stepi
7           iret
1: x/i $pc
=> 0x100080 <breakpoint+3>:    iret

(gdb) print $eflags
$7 = [ PF ]

(gdb) print/x $eflags
$8 = 0x6
```

D. Avanzar, por última vez, una instrucción, de manera que se ejecute IRET para la sesión A, y RET para la sesión B. Mostrar, de nuevo lo pedido que en el punto 1; y explicar cualquier diferencia entre ambas versiones A y B.

```
(gdb) stepi
kmain (mbi=0x9500) at kern2.c:27
27          vga_write2("Funciona vga_write2?", 18, 0xE0);
1: x/i $pc
=> 0x10014f <kmain+151>:        mov $0xe0,%ecx

(gdb) print $esp
$9 = (void *) 0x107ee8

(gdb) x/xw $esp
0x107ee8:       0x6c646d63

(gdb) print $cs
$10 = 8

(gdb) print $eflags
$11 = [ AF ]

(gdb) print/x $eflags
$12 = 0x12
```

Todos los valores son iguales que antes de entrar en la interrupción. Esto es porque la instrucción IRET saca de la pila el %eip, %cs y %eflags, y luego resume la ejecución donde estaba.

➔ VERSIÓN B (RET):

A. ***Poner un breakpoint en la función idt_init() y, una vez dentro, finalizar su ejecución con el comando de GDB finish. La ejecución debería haberse detenido en la misma instrucción int3. Mostrar:***

```
Continuing.

Breakpoint 1, idt_init () at interrupts.c:22
22                  idt_install(T_BRKPT, breakpoint);
1: x/i $pc
=> 0x100348 <idt_init+3>:         push $0x10007d

(gdb) finish
Run till exit from #0 idt_init () at interrupts.c:22
kmain (mbi=0x9500) at kern2.c:25
25          asm("int3");
1: x/i $pc
=> 0x10014e <kmain+150>:       int3     (Próxima instrucción).
```

- el valor de %esp (print $esp)

```
(gdb) print $esp
$1 = (void *) 0x107ee8
```

- el valor de (%esp) (x/xw $esp)

```
(gdb) x/xw $esp
0x107ee8:        0x6c646d63
```

- el valor de $cs

```
(gdb) print $cs
$2 = 8
```

- el resultado de print $eflags y print/x $eflags

```
(gdb) print $eflags
$3 = [ AF ]
(gdb) print/x $eflags
$4 = 0x12
```

B. ***Ejecutar la instrucción int3 mediante el comando de GDB stepi. La ejecución debería saltar directamente a la instrucción test %eax, %eax.***

```
(gdb) stepi
breakpoint () at idt_entry.S:6
6           test %eax, %eax
1: x/i $pc
=> 0x10007e <breakpoint+1>:    test %eax,%eax

(gdb) print $esp
$5 = (void *) 0x107edc
```

Avanzó de 0x107ee8 a 0x107edc, es decir 12 bytes o 3 posiciones.

```
(gdb) x/3wx $esp
0x107edc:       0x0010014f      0x00000008      0x00000012

(gdb) print $eflags
$6 = [ AF ]
```

Los valores de la pila son, en orden, el valor del Instruction pointer siguiente al que tenía antes de entrar en la excepción (int 3), el valor de $cs y el valor de $eflags.

C. ***Avanzar una instrucción más con stepi, ejecutando la instrucción TEST. Mostrar, como anteriormente, el valor del registro EFLAGS.***

```
(gdb) stepi
7           ret
1: x/i $pc
=> 0x100080 <breakpoint+3>:    ret

(gdb) print $eflags
$7 = [ PF ]

(gdb) print/x $eflags
$8 = 0x6
```

D. Avanzar, por última vez, una instrucción, de manera que se ejecute IRET para la sesión A, y RET para la sesión B. Mostrar, de nuevo lo pedido que en el punto 1; y explicar cualquier diferencia entre ambas versiones A y B.

```
(gdb) stepi
kmain (mbi=0x9500) at kern2.c:27
27          vga_write2("Funciona vga_write2?", 18, 0xE0);
1: x/i $pc
=> 0x10014f <kmain+151>:        mov $0xe0,%ecx

(gdb) print $esp
$9 = (void *) 0x107ee0

(gdb) x/xw $esp
0x107ee0:       0x00000008

(gdb) print $cs
$10 = 8

(gdb) print $eflags
$11 = [ PF ]

(gdb) print/x $eflags
$12 = 0x6
```

No se restauró todo como estaba antes. El eflags y cs no se restauraron. Tampoco se sacó del stack los 3 valores que se agregaron. Si vga_write2 tomara argumentos por la pila no hubiera funcionado el programa porque tendría otros argumentos que ret no saca de la pila e iret si. Pudo llegar a vga_write2 únicamente porque el tope de la pila tenía el valor del instruction pointer, si tuviera otra cosa no hubiera llegado.

Versión final de breakpoint()

1. ***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 porque pusha guarda absolutamente todos los registros con el valor que tenían. Al hacer popa se restauran a como estaban antes de la instrucción.

OPCIÓN B: Es correcto porque guarda únicamente los registros que son caller-saved. Como la función de la excepción no es llamada explícitamente, el programa interrumpido no siempre puede guardarlos. Los calle-saved no los guarda porque no los usa. Si vga_write2 los usara los debería guardar esa función.

OPCIÓN C: No es correcto, ya que los registros caller-saved nadie va a guardarlos, y de hecho se reemplazan antes de llamar a vga_write2 con otros valores.

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

La respuesta es la misma para OPCIONES A y B. Para la OPCIÓN C, podría funcionar o no dependiendo de si vga_write usa o no los registros caller-saved. Sin embargo, nunca sería correcto implementarlo así.

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

No, como se explicó en el ejercicio anterior, el valor de EFLAGS es guardado automáticamente en la pila, y restaurado mediante la instrucción IRET.

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

Se ejecuta en el stack de la función que genera la excepción, que en este caso es el mismo que el del kernel (función kmain).

3.3 kern2-div

1. ***Explicar el funcionamiento exacto de la línea asm(...) del punto anterior:***

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

Primero se guardan valores iniciales en los registros indicados:

%eax (0) = 18
%ecx (1) = 0xE0
%ebx = 1
%edx = 0

Luego, div %4, lo que hace es dividir el valor en EDX:EAX por lo contenido en el registro 4, que es EBX.

Como %edx es 0, hace %eax = %eax / %ebx.
Por lo tanto, hace 18 / 1;

Por último guarda en la variable línea el valor de %eax, que es el resultado de la división (18), y guarda en la variable color el valor de %ecx, que es 0xE0 (el inicial), porque no cambió en la división.

A %edx se le da valor 0 para que el dividendo sea únicamente el valor de %eax, porque EDX:EAX es igual a EAX cuando edx = 0.

2. ***Asignar a %ebx el valor 0 en lugar de 1, y comprobar que se genera una triple falla al ejecutar el kernel***

Se produce una triple falla porque todavía no implementamos el handler de la excepción correspondiente a dividir por 0.

3.4 kern2-kbd

En el desarrollo de esta sección, se realizaron una serie de agregados. A continuación se enumeran los agregados realizados:

1) Se agregaron teclas especiales al listado de teclas para luego agregarles una funcionalidad específica. Entre estas se encuentran: “Backspace”, “Enter”, “Space”, “CapsLock”, “Left Arrow” y “Right Arrow”.

2) Se agregó la posibilidad de borrar el texto escrito utilizando la tecla “Backspace” como tecla para indicar que se desea borrar.

3) Se agregó la posibilidad de escribir espacios con la barra espaciadora (“Space”).

4) La funcionalidad de la tecla “Enter” será la de borrar todo el contenido de la línea.

5) Se agregó un cursor que indica la posición en la cual se escribirá el próximo caracter. Este cursor se muestra en pantalla con el caracter ‘^’.

6) Se agregó la posibilidad de mover el cursor de posición utilizando las flechas izquierda o derecha. De esta forma el usuario podrá mover el cursor y escribir en otras posiciones.

7) Cuando se aprieta “CapsLock”, se cambia la escritura de las letras de minúscula a mayúscula y viceversa. Además, el funcionamiento del “Shift” dependerá del estado de “CapsLock”. Si escribe en mayúscula, al mantener apretado “Shift” escribirá en minúscula y viceversa.

8) El funcionamiento del “Shift” se limitó únicamente a las letras debido a que en las otras teclas, el offset no es fijo y, además, el símbolo de la tecla dependerá del tipo de teclado que se tenga (si es teclado español o teclado inglés).

Aclaraciones: En el código se puede apreciar que las teclas “Left Arrow”, “Right Arrow” y “CapsLock” tienen definido como carácter de escritura los caracteres ‘=’, ‘#’ y ‘!’. Esto es debido a que necesitamos agregar un caracter para poder distinguir que se presionaron dichas teclas. Como esos caracteres no los definimos, decidimos utilizarlos ya que no podrán ser escritos por el usuario.