75.08 Sistemas Operativos

Entrega Kernel 2

Nombre y Apellido: Sebastián Ezequiel Blanco
Padrón: 98539
Nombre y Apellido: Martín Nicolás Pérez
Padrón: 97378
Fecha de Entrega: 22/06/2018
GitHub: https://github.com/BlancoSebastianEzequiel/LabKernel2

Índice

1. Creación de stacks en el kernel

1.1. Ej: kern2-stack

- *Explicar: ¿qué significa "estar alineado"?*
  Estar alineado significa que nuestro espacio de memoria será múltiplo de un numero que es el que se utiliza para alinear, es decir, que los primeros n bit empezaran en cero.

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

```
1    unsigned char stack1[8192] __attribute__((aligned(32)));
```

- *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.)*
  Esta inicializado en cero ya que el segundo argumento de space que es fill no es especificado, y en tal caso su valor es cero. Si varia en la version c ya que en esta ultima el valor de la stack en C tiene datos basura

- *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.*
  Ambas rellenean el contador de ubicación (en la subsección actual) con un límite de almacenamiento particular. La primera expresión (que debe ser absoluta) es la alineación requerida. La diferencia es que align alinea por el valor que se le pasa y p2align alinea por el valor que se le pasa como potencia de dos, es decir que align 4096 alinea a 4Kb y p2align 12 también.

- *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.*

```
1  $ make gdb
2  gdb -q -s kern2 -n -ex 'target remote 127.0.0.1:7508'
3  Leyendo simbolos desde kern2...hecho.
4  Remote debugging using 127.0.0.1:7508
5  0x0000fff0 in ?? ()
6  (gdb) b kmain
7  Punto de interrupcion 1 at 0x100129: file kern2.c, line 62.
```

```
(gdb) c
Continuando.

Breakpoint 1, kmain (mbi=0x9500) at kern2.c:62
62  void kmain(const multiboot_info_t *mbi) {
(gdb) p $esp
$1 = (void *) 0x104ff4
(gdb) p/x $eip
$2 = 0x100129
(gdb) info frame
Stack level 0, frame at 0x104ff8:
eip = 0x100129 in kmain (kern2.c:62); saved eip = 0x10002d
called by frame at 0x104ffc
source language c.
Arglist at 0x104ff0, args: mbi=0x9500
Locals at 0x104ff0, Previous frame's sp is 0x104ff8
Saved registers:
eip at 0x104ff4
```

1.2. Ej: kern2-cmdline

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

```
char *strncat(char *dest, const char *src, size_t n) {
    size_t dest_len = strlen(dest);
    size_t i;
    for (i = 0 ; i < n && src[i] != '\0' ; i++) {
        dest[dest_len + i] = src[i];
    }
    dest[dest_len + i] = '\0';
    return dest;
}
```

- *Explicar cómo se comporta strlcat(3) si, erróneamente, se declarase buf con tamaño 12. ¿Introduce algún error el código?*
  No introduce error. Esta función recibe un tercer parámetro que es el tamaño máximo que la cadena final tendrá. En este caso, si buf tuviera tamaño 12, no seria un problema porque vemos que como tercer parámetro recibe el tamaño de la cadena destino, mas el tamaño de la cadena fuente mas uno. Entonces como el tamaño máximo es mayor tamaño de la cadena destino, se copia la cadena fuente a la destino.

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

```
1  #include <stdio.h>
2  static void printf_sizeof_buf(char buf[256]) {
3      printf("sizeof buf = %zu\n", sizeof buf);
4  }
5  int main(void) {
6      char buf[256];
7      printf("sizeof buf = %zu\n", sizeof buf);
8      printf_sizeof_buf(buf);
9  }
```

Esto es erróneo porque al tener como argumento char buf[256] recibe como parámetro un puntero y a este le aplica el operador size of, lo cual te da como resultado cuanto ocupa un puntero en memoria. Que en general pueden ser 4 u 8 bytes. En cambio, el otro caso funciona porque recibe como parámetro el puntero a un buffer y lo que hace es calcular las distancia entre este puntero y el ultimo caracter. Para que ambos resulten iguales la funcion debiera tener como parametro char* buff solamente.

2. Concurrencia cooperativa

2.1. Ej: kern2-swap

- *Explicar, para el stack de cada contador, cuántas posiciones se asignan, y qué representa cada una.*
  En el caso del segundo stack, se pasan los tres argumentos que la función contado_yield necesita, luego se pasa un puntero a la función exit y luego un puntero a la función contador_yield. Finalmente se inicializan las cuatro posiciones siguientes donde se guardaran los cuatro registros callee saved. Esto se hace porque cunado se cambia de stack, este retira de la stack dichos registros. Por lo tanto luego de ello apuntaría a contador_yield.
  Cuando se hace el swap en la función task_swap y se hace el switch entre stacks, la stack actual apunta a la segunda stack. Le saca los cuatro registros callee saved y queda apuntando a la función contador_yield. Por lo tanto cuando ejecuta la instrucción ret salta a la función contador_yield. En caso de que el segundo contador termine antes, cuando termina el loop principal, salta a otra función de retorno que en este caso es la que se guardo antes y que seria exit.
  En el caso del primer contador solo se guardan los tres parámetros. No es necesario guardar direcciones de retorno ya que es con este stack que se llama

a la función, por lo tanto cuando se hace un call a contador_yield se guarda la dirección de retorno en la cuata posición del stack.

3. Interrupciones: reloj y teclado

3.1. Ej: kern2-idt

- *¿Cuántos bytes ocupa una entrada en la IDT?*
  Una entrada en la IDT tiene un tamaño fijo de 8 bytes.

- *¿Cuántas entradas como máximo puede albergar la IDT?*
  Como máximo la IDT puede albergar 256 entradas, a estas se las conoce como descriptores.

- *¿Cuál es el valor máximo aceptable para el campo limit del registro IDTR?*
  El valor máximo aceptable para el campo limit depende de la cantidad de descriptores válidos requeridos para representar las interrupciones y excepciones que pueden llegar a ocurrir. Debería ser (8*N - 1), expresado en bytes. Siendo N la cantidad de entradas de la IDT, el valor 8 refiere a el tamaño fijo de las entradas de la tabla. Todos los descriptores vacios deben tener el flag present seteado en 0.

- *Indicar qué valor exacto tomará el campo limit para una IDT de 64 descriptores solamente.*
  Con una IDT de 64 descriptores el valor del campo limit será ((8 * 64) -1) = 511

- *Consultar la sección 6.1 y explicar la diferencia entre interrupciones (§6.3) y excepciones (§6.4).*
  Las interrupciones y las excepciones son eventos externos al procesador son eventos externos al procesador, que requieren la atencion del mismo para ser manejadas.

  A primera vista, las diferencias entre interrupciones y excepciones radican en el origen y finalidad de las mismas.

  Las interrupciones ocurren en momentos randoms de los programas y responden a señales externas al sistema. Estas señales son generadas tanto por el Hardware, por ejemplo para que el procesador responda al pedido del teclado para detectar el ingreso de entrada de datos, o bien por el Software, mediante la ejecución de la instrucción INT.

Las excepciones ocurren cuando el procesador encuentra un problema en la ejecución de una instrucción, por ejemplo dividir por cero, page faults, violaciones de protección, etc. En las arquitecturas *Machine-Check*, tambien es posible que se generen excepciones generadas por errores internos del Hardware o errores de Bug, llamadas *Machine-Check Exceptions*.

3.2. Ej: kern2-isr

3.2.1. Sesión de GDB

Se debe seguir el mismo guión dos veces:

```
1 .globl breakpoint
2 breakpoint:
3     nop
4     test %eax, %eax
5     iret
```

- *versión A: usando la implementación aumentada del manejador:*

```
1   0x0000fff0 in ?? ()
2   (gdb) display/i $pc
3   1: x/i $pc
4   => 0xfff0:  add    %al,(%eax)
5   (gdb) b idt_init
6   Punto de interrupcion 1 at 0x100586: file interrupts.c, line 27.
7   (gdb) c
8   Continuando.
9
10  Breakpoint 1, idt_init () at interrupts.c:27
11  27      idt_install(T_BRKPT, breakpoint);
12  1: x/i $pc
13  => 0x100586 <idt_init+3>:   push   $0x1000a8
14  (gdb) finish
15  Correr hasta la salida desde #0  idt_init () at interrupts.c:27
16  kmain (mbi=0x9500) at kern2.c:73
17  73      asm("int3");  // (b)
18  1: x/i $pc
19  => 0x100193 <kmain+76>: int3
20  (gdb) x/10i $pc
21  => 0x100193 <kmain+76>: int3
22  0x100194 <kmain+77>:    mov    $0x0,%edx
```

```
0x100199 <kmain+82>:    mov    $0xe0, %ecx
0x10019e <kmain+87>:    mov    $0x12, %eax
0x1001a3 <kmain+92>:    mov    %edx, %ebx
0x1001a5 <kmain+94>:    div    %ebx
0x1001a7 <kmain+96>:    movzbl %cl, %ecx
0x1001aa <kmain+99>:    movsbl %al, %edx
0x1001ad <kmain+102>:   mov    $0x100f77, %eax
0x1001b2 <kmain+107>:   call   0x10009b <vga_write2>
(gdb) print $esp
$1 = (void *) 0x104d78
(gdb) x/xw $esp
0x104d78:   0x00100f5c
(gdb) print $cs
$2 = 8
(gdb) print $eflags
$3 = [ ]
(gdb) print/x $eflags
$4 = 0x2
(gdb) stepi
breakpoint () at idt_entry.S:17
17          test %eax, %eax
1: x/i $pc
=> 0x1000a9 <breakpoint+1>: test   %eax, %eax
(gdb) print $esp
$5 = (void *) 0x104d6c
```

- *¿Cuantas posiciones avanzo?* Avanzo 12 posiciones
- *¿qué representa cada valor?*
  Representa el contenido actual de la memoria del stack en formato word

```
(gdb) x/12wx $sp
0x104d6c:   0x00100194  0x00000008  0x00000002  0x00100f5c
0x104d7c:   0x00000008  0x00000070  0x00000000  0x00000000
0x104d8c:   0x00000000  0x00000000  0x00000000  0x00000000
(gdb) print $eflags
$6 = [ ]
(gdb) print/x $eflags
$7 = 0x2
(gdb) stepi
18          iret
1: x/i $pc
=> 0x1000ab <breakpoint+3>: iret
```

```
(gdb) print $eflags
$8 = [ PF ZF ]
(gdb) print/x $eflags
$9 = 0x46
(gdb) stepi
kmain (mbi=0x9500) at kern2.c:78
78      asm("div %4"
1: x/i $pc
=> 0x100194 <kmain+77>: mov    $0x0, %edx
(gdb) x/10i $pc
=> 0x100194 <kmain+77>: mov    $0x0, %edx
0x100199 <kmain+82>:    mov    $0xe0, %ecx
0x10019e <kmain+87>:    mov    $0x12, %eax
0x1001a3 <kmain+92>:    mov    %edx, %ebx
0x1001a5 <kmain+94>:    div    %ebx
0x1001a7 <kmain+96>:    movzbl %cl, %ecx
0x1001aa <kmain+99>:    movsbl %al, %edx
0x1001ad <kmain+102>:   mov    $0x100f77, %eax
0x1001b2 <kmain+107>:   call   0x10009b <vga_write2>
0x1001b7 <kmain+112>:   add    $0x10, %esp
(gdb) print $esp
$12 = (void *) 0x104d78
(gdb) x/xw $esp
0x104d78:   0x00100f5c
(gdb) print $cs
$13 = 8
(gdb) print $eflags
$10 = [ ]
(gdb) print/x $eflags
$11 = 0x2
```

- *Versión B: con el mismo manejador, pero cambiando la instrucción IRET por una instrucción RET.*

```
0x0000fff0 in ?? ()
(gdb) display/i $pc
1: x/i $pc
=> 0xfff0:  add    %al,(%eax)
(gdb) b idt_init
Punto de interrupcion 1 at 0x100586: file interrupts.c, line 27.
(gdb) c
Continuando.
```

```
9
10  Breakpoint 1, idt_init () at interrupts.c:27
11  27      idt_install(T_BRKPT, breakpoint);
12  1: x/i $pc
13  => 0x100586 <idt_init+3>:   push   $0x1000a8
14  (gdb) finish
15  Correr hasta la salida desde #0  idt_init () at interrupts.c:27
16  kmain (mbi=0x9500) at kern2.c:73
17  73      asm("int3");  // (b)
18  1: x/i $pc
19  => 0x100193 <kmain+76>: int3
20  (gdb) x/10i $pc
21  => 0x100193 <kmain+76>: int3
22  0x100194 <kmain+77>:    mov    $0x0, %edx
23  0x100199 <kmain+82>:    mov    $0xe0, %ecx
24  0x10019e <kmain+87>:    mov    $0x12, %eax
25  0x1001a3 <kmain+92>:    mov    %edx, %ebx
26  0x1001a5 <kmain+94>:    div    %ebx
27  0x1001a7 <kmain+96>:    movzbl  %cl, %ecx
28  0x1001aa <kmain+99>:    movsbl  %al, %edx
29  0x1001ad <kmain+102>:   mov    $0x100f77, %eax
30  0x1001b2 <kmain+107>:   call   0x10009b <vga_write2>
31  (gdb) print $esp
32  $1 = (void *) 0x104d78
33  (gdb) x/xw $esp
34  0x104d78:   0x00100f5c
35  (gdb) print $cs
36  $2 = 8
37  (gdb) print $eflags
38  $3 = [ ]
39  (gdb) print/x $eflags
40  $4 = 0x2
41  (gdb) stepi
42  breakpoint () at idt_entry.S:26
43  26          test  %eax,  %eax
44  1: x/i $pc
45  => 0x1000a9 <breakpoint+1>: test   %eax, %eax
46  (gdb) print $esp
47  $5 = (void *) 0x104d6c
```

- *¿Cuantas posiciones avanzo?* Avanzo 12 posiciones

- *¿qué representa cada valor?*
  Representa el contenido actual de la memoria del stack en formato word

```
(gdb) x/12wx $sp
0x104d6c:   0x00100194  0x00000008  0x00000002  0x00100f5c
0x104d7c:   0x00000008  0x00000070  0x00000000  0x00000000
0x104d8c:   0x00000000  0x00000000  0x00000000  0x00000000
(gdb) print $eflags
$6 = [ ]
(gdb) print/x $eflags
$7 = 0x2
(gdb) stepi
27          ret
1: x/i $pc
=> 0x1000ab <breakpoint+3>: ret
(gdb) print $eflags
$8 = [ PF ZF ]
(gdb) print/x $eflags
$9 = 0x46
(gdb) stepi
kmain (mbi=0x9500) at kern2.c:78
78      asm("div  %4"
1: x/i $pc
=> 0x100194 <kmain+77>: mov    $0x0, %edx
(gdb) x/10i $pc
=> 0x100194 <kmain+77>: mov    $0x0, %edx
0x100199 <kmain+82>:    mov    $0xe0, %ecx
0x10019e <kmain+87>:    mov    $0x12, %eax
0x1001a3 <kmain+92>:    mov    %edx, %ebx
0x1001a5 <kmain+94>:    div    %ebx
0x1001a7 <kmain+96>:    movzbl  %cl, %ecx
0x1001aa <kmain+99>:    movsbl  %al, %edx
0x1001ad <kmain+102>:   mov    $0x100f77, %eax
0x1001b2 <kmain+107>:   call   0x10009b <vga_write2>
0x1001b7 <kmain+112>:   add    $0x10, %esp
(gdb) print $esp
$10 = (void *) 0x104d70
(gdb) x/xw $esp
0x104d70:   0x00000008
(gdb) print $cs
$11 = 8
(gdb) print $eflags
$12 = [ PF ZF ]
(gdb) print/x $eflags
```

```
42  $13 = 0x46
```

- *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

```
1   breakpoint:
2   pusha
3   ...
4   call vga_write2
5   popa
6   iret
```

  - Opcion B

```
1   breakpoint:
2   push %eax
3   push %edx
4   push %ecx
5   ...
6   call vga_write2
7   pop %ecx
8   pop %edx
9   pop %eax
10  iret
```

  - Opcion C

```
1   breakpoint:
2   push %ebx
3   push %esi
4   push %edi
5   ...
6   call vga_write2
7   pop %edi
8   pop %esi
9   pop %ebx
10  iret
```

La opcion correcta es la B, ya que al llamar a la funcion, se supone que los registros callee saved, son salvados de manera tal que al volver de la funcion, estos mantienen su valor. En cambio, los registros caller saved no. Por lo tanto, la opcion A hace trabajo de mas al salvar todos los registros y la C

no funcionaria ya que salva los registros callee saved que no es necesario ya que de eso se encarga la funcion llamada pero no salva los caller saved y al volver se podria perder si valor.

3.3. Ej: kern2-div

Explicar el funcionamiento exacto de la línea asm(...):

```
1     void kmain(const multiboot_info_t *mbi) {
2     int8_t linea;
3     uint8_t color;
4
5     // ...
6
7     idt_init();
8     irq_init();
9
10    asm("div %4"
11    : "=a"(linea), "=c"(color)
12    : "0"(18), "1"(0xE0), "b"(1), "d"(0));
13
14    vga_write2("Funciona vga_write2?", linea, color);
15    }
```

- *¿qué cómputo se está realizando?*
  Esta dividiendo %eax por %ebx que le asigna el valor uno

- *¿de dónde sale el valor de la variable color?*
  El valor se lo asigna en caso de que la division haya sido exitosa y el valor sale del primer registro "1"(0xE0) por eso.

- *¿por qué se da valor 0 a %edx?*
  Se le da valor cero porque el resto de la division se produce entre EDX:EAX y el EBX.