Lab 2 - Kernel mínimo y llamadas en x86

[75.08 / 95.03] Sistemas Operativos
Segundo cuatrimestre de 2019

Alumno:	Montiel, Ramiro
Número de padrón:	100870
Email:	tec.rimontiel@gmail.com

Índice

1. Ej: kern0-boot

¿emite algún aviso el proceso de compilado o enlazado?
Si, el error de ld fue: ld: aviso: no se puede encontrar el símbolo de entrada start; se usa por defecto 0000000000100000 El comando -entry de ld hace que se le indique en que función comenzará el programa. Indicando con e comienzo"podemos corregir este error.

¿cuánta CPU consume el proceso qemu-system-i386 mientras ejecuta este kernel? ¿Qué está haciendo?
El consumo de CPU del kernel es: 100,7% segun el comando top. Lo que está haciendo es ejecutar un loop infinito con ninguna instruccion dentro.

2. Ej: kern0-hlt

una vez invocado hlt ¿cuándo se reanuda la ejecución?
Se reanuda cuando ocurra una interrupción.

Uso de procesador:

USO	Eventos/seg
Sin hlt:	990,7 ms/s 0,00
Con hlt:	7,5 ms/s 152,6

3. Ej: kern0-gdb

Mostrar una sesión de GDB en la que se realicen los siguientes pasos:
Stack Pointer: $esp = 0x6f08

Registro eax = 0x2badb002

Registro ebx = 0x0000024f 0x0000027f 0x0001fb80 0x8000ffff
El primer Word indica los flags de configuracion
El segundo Word indica la direccion inicial de memoria asignada
El tercer Word indica la direccion final de memoria asignada
El cuarto Word dice de que dispositivo el bootloader cargó el sistema operativo.

Campo flags en formato binario (x/1tw $ebx): 00000000000000000000001001001111

la cantidad de memoria ‘‘baja’’ en formato decimal (x/2dw $ebx): 591 639 <- Este es el que nos importa
¿en qué unidad está?
Está en KiloBytes.
La línea de comandos o ‘‘cadena de arranque’’ recibida por el kernel (x/s 0x00101000) = "kern0 ".

4. Ej: kern0-vga

Qué se imprime por pantalla al arrancar?
Se imprime arriba a la izquierda de la pantalla la palabra: OK

Qué representan cada uno de los valores enteros (incluyendo 0xb8000)?
0xb8000 es la direccion de memoria para el buffer de video, esta direccion recibe un caracter ascii para imprimir en pantalla, en el siguiente byte debe ingresarse el color del cual queremos que esa letra se imprima en pantalla.

Por qué se usa el modificador volatile para el puntero al buffer? Se usa para indicarle al compilador que el contenido de esa variable puede ser modificado fuera de nuestro programa. Entonces cada vez que nuestro programa lo quiera usar, va a volver a leer el contenido de esa variable. Recordemos que esa es la direccion de memoria del buffer de video, que podria tambien ser utilizado y modificado por otros programas tambien.

Ahora, implementar una función más genérica para imprimir en el buffer VGA:

```
static void vga_write(const char *s, int linea, int color){
    //La pantalla tiene 80 Caracteres de ancho y 24 de alto.
    int i = 0;
    if(linea < 0){
        while(linea < 0){
            linea++;
            i++;
        }
        linea = (24-i);
    }

    volatile char *buf = VGABUF+(160*linea);

    while(*s != 0){
        *buf++ = *s++;
        *buf++ = color;
    }
}
```

Figura 1: Funcion vga_write.

5. Ej: kern0-endian

La salida del programa es: "He110 World"Para que el programa funcione en una arquitectura big endian.
Habría que modificar la linea:
unsigned int i = 0x00646c72; //'null','d','l','r'
Por: unsigned int i = 0x726c6400;

El siguiente código cumple con los ejercicios 2 y 3:

```
void comienzo(void) {
    //79 = 4F Hexa = 'O'; 47 = 2F Hexa = Verde; 75 = 4B Hexa = 'K'
    //HOLA : H = 48 (Hexa); O = 4F (Hexa); L = 4C; A = 41 (Hexa)
    // Fondo amarillo (Hexa) = EF
    volatile unsigned *p = VGABUF;
    *p = 0x2F4B2F4F;
    volatile long long int *p2 = VGABUF + 160;
    *p2 = 0xEF41EF4CEF4FEF48;
    while (1)
        asm("hlt");
}
```

Figura 2: Escribe Ok en 1er linea y HOLA en 3er linea

6. Ej: x86-write

¿Por qué se le resta 1 al resultado de sizeof?
Para no imprimir el caracter de fin de string. ¿Funcionaría el programa si se declarase msg como const char msg = ·..°";? ¿Por qué?
No funcionaria, porque el sizeof devuelve la longitud de un puntero a char, lo cual no es lo mismo que la longitud del contenido de msg. Explicacion libc_hello: Se pushea a la pila la variable len que tiene la cantidad de caracteres del string Se puseha a la pila el string a imprimir Se pushea a la pila el numero 1 Se llama a la syscall "write"Se pushea a la pila el numero 7 Se llama a la syscall "_exit"

Mostrar un hex dump de la salida del programa en assembler. Se puede obtener con el comando od:

```
Hex dump de la salida de ./libc_hello (con instruccion .ascii):
0000000  48  65  6c  6c  6f  2c  20  77  6f  72  6c  64  21  0a
         H   e   l   l   o   ,       w   o   r   l   d   !  \n
0000016

Hex dump de la salida de ./libc_hello (con instruccion .asciz):
0000000  48  65  6c  6c  6f  2c  20  77  6f  72  6c  64  21  0a  00
         H   e   l   l   o   ,       w   o   r   l   d   !  \n  \0
0000017
```

Figura 3: Hex dump con directiva .ascii y .asciz

La diferencia entre ambos es el \0ál final del segundo. Esto se debe a que .asciz imprime strings con un zero al final.

7. Ej: x86-libc

Compilar y ejecutar el archivo completo int80_hi.S. Mostrar la salida de nm -undefined para este nuevo binario:

```
Salida de nm -u int80_hi:
         w __cxa_finalize@@GLIBC_2.1.3
         w __gmon_start__
         w _ITM_deregisterTMCloneTable
         w _ITM_registerTMCloneTable
         U __libc_start_main@@GLIBC_2.

Salida de nm -u int80_hi_strlen:
         w __cxa_finalize@@GLIBC_2.1.3
         w __gmon_start__
         w _ITM_deregisterTMCloneTable
         w _ITM_registerTMCloneTable
         U __libc_start_main@@GLIBC_2.0
         U strlen@@GLIBC_2.0
```

Figura 4: Comando nm -u

¿qué significa que un registro sea callee-saved en lugar de caller-saved?
Callee-saved significa que la funcion que esta utilizando ese registro se encarga de guardarlo.
Caller-saved significa que la funcion que llama a otra funcion se encarga de guardar ese registro que esté utilizando.

En x86 ¿de qué tipo, caller-saved o callee-saved, es cada registro según la convención de llamadas de GCC?
Son Caller-saved: EAX, ECX y EDX.
Son Callee-saved: EBP, EBX, EDI y ESI.

8. Ej: x86-frames

¿Dónde se encuentra (de haberlo) el primer argumento de f?
Se encuentra en %ebp+8.
¿Dónde se encuentra la dirección a la que retorna f cuando ejecute ret?
Se encuentra en %ebp+4.
¿Dónde se encuentra el valor de %ebp de la función anterior, que invocó a f?
Se encuentra en %ebp.
¿Dónde se encuentra la dirección a la que retornará la función que invocó a f?
Se encuentra en %ebp+4.

Código de la función backtrace:

```
void backtrace(){

    int *frameEbp = (int *) __builtin_frame_address(0);
    frameEbp = (int*)*frameEbp;
    int i = 1;

    while(frameEbp != NULL){
        printf("#%d [0x%x] 0x%x ( 0x%x 0x%x 0x%x )\n", i++, frameEbp, *(frameEbp+1), *(frameEbp+2),
        *(frameEbp+3), *(frameEbp+4));
        frameEbp = (int*)*frameEbp;
    }
}
```

Figura 5: Funcion backtrace

Sesion de GDB:

```
(gdb) b backtrace
Punto de interrupción 1 at 0x8048506: file backtrace.c, line 5.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/rnet/Escritorio/UBA/tpSisopKern0/x86-frames/backtrace

Breakpoint 1, backtrace () at backtrace.c:5
5 void backtrace(){
(gdb) bt
#0  backtrace () at backtrace.c:5
#1  0x0804855d in my_write (fd=2, msg=0x80486bb, count=15) at backtrace.c:18
#2  0x080485a9 in recurse (level=0) at backtrace.c:27
#3  0x080485ba in recurse (level=1) at backtrace.c:25
#4  0x080485ba in recurse (level=2) at backtrace.c:25
#5  0x080485ba in recurse (level=3) at backtrace.c:25
#6  0x080485ba in recurse (level=4) at backtrace.c:25
#7  0x080485ba in recurse (level=5) at backtrace.c:25
#8  0x080485cc in start_call_tree () at backtrace.c:31
```

```
#9  0x080485e7 in main () at backtrace.c:35
(gdb) list
1 #include <stdint.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void backtrace(){
6
7     int *frameEbp = (int *)__builtin_frame_address(0);
8     frameEbp = (int*)*frameEbp;
9     int i = 1;
10
(gdb) until 14
#1 [0xbffff068] 0x80485a9 ( 0x2 0x80486bb 0xf )
#2 [0xbffff088] 0x80485ba ( 0x0 0x0 0x0 )
#3 [0xbffff0a8] 0x80485ba ( 0x1 0xf63d4e2e 0xb7fffaf8 )
#4 [0xbffff0c8] 0x80485ba ( 0x2 0x1 0xb7fd1410 )
#5 [0xbffff0e8] 0x80485ba ( 0x3 0x0 0x0 )
#6 [0xbffff108] 0x80485ba ( 0x4 0xbffff3a0 0xb7e064a9 )
#7 [0xbffff128] 0x80485cc ( 0x5 0x0 0xbffff20c )
#8 [0xbffff148] 0x80485e7 ( 0xb7fe79b0 0xbffff170 0x0 )
#9 [0xbffff158] 0xb7deee81 ( 0xb7fae000 0xb7fae000 0x0 )
backtrace () at backtrace.c:15
15 }
(gdb) up
#1  0x0804855d in my_write (fd=2, msg=0x80486bb, count=15) at backtrace.c:18
18     backtrace();
(gdb) p/x $ebp
$1 = 0xbffff068
(gdb) up
#2  0x080485a9 in recurse (level=0) at backtrace.c:27
27         my_write(2, "Hello, world!\n", 15);
(gdb) p/x $ebp
$2 = 0xbffff088
(gdb) up
#3  0x080485ba in recurse (level=1) at backtrace.c:25
25         recurse(level - 1);
(gdb) p/x $ebp
$3 = 0xbffff0a8
(gdb) up
#4  0x080485ba in recurse (level=2) at backtrace.c:25
25         recurse(level - 1);
(gdb) p/x $ebp
$4 = 0xbffff0c8
(gdb) up
#5  0x080485ba in recurse (level=3) at backtrace.c:25
25         recurse(level - 1);
(gdb) p/x $ebp
$5 = 0xbffff0e8
(gdb) up
#6  0x080485ba in recurse (level=4) at backtrace.c:25
25         recurse(level - 1);
(gdb) p/x $ebp
$6 = 0xbffff108
```

```
(gdb) up
#7  0x080485ba in recurse (level=5) at backtrace.c:25
25         recurse(level - 1);
(gdb) p/x $ebp
$7 = 0xbffff128
(gdb) up
#8  0x080485cc in start_call_tree () at backtrace.c:31
31     recurse(5);
(gdb) p/x $ebp
$8 = 0xbffff148
(gdb) up
#9  0x080485e7 in main () at backtrace.c:35
35     start_call_tree();
(gdb) p/x $ebp
$9 = 0xbffff158
(gdb) up
Initial frame selected; you cannot go up.
(gdb) frame 0
#0  backtrace () at backtrace.c:15
15 }
(gdb) c
Continuando.
=> write(2, 0x80486bb, 15)
Hello, world!
[Inferior 1 (process 1832) exited normally]
```