資工三 405235035 王博輝

作業系統概論 期中上機

1. **請問你是用懶人包，或者自行設定 trace kernel 的環境？**

   使用 virtual machine 懶人包

2. **設定 breakpoint 於 start_kernel，並印出 start_kernel 的 call stack，及 call stack 中各個函數的位址**

```
(gdb) b start_kernel 設定breakpoint於start_kernel
Breakpoint 1 at 0xffffffff822dbf08: file init/main.c, line 490.
(gdb) c
Continuing.

Breakpoint 1, start_kernel () at init/main.c:490
490     {
(gdb) bt 印出start_kernel的call stack
#0  start_kernel () at init/main.c:490   call stack中各個函數的位址
#1  0xffffffff822db72d in x86_64_start_reservations (
    real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>)
    at arch/x86/kernel/head64.c:200
#2  0xffffffff822db6f0 in x86_64_start_kernel (
    real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>)
    at arch/x86/kernel/head64.c:189
#3  0x0000000000000000 in ?? ()
```

3. **找出中斷向量表的開始位址**

```
(gdb) b _set_gate
Breakpoint 2 at 0xffffffff810004b8: _set_gate. (3 locations)
(gdb) c
Continuing.

Breakpoint 2, _set_gate (gate=1, type=14, addr=0xffffffff81d0fa70 <debug>, dpl=0, ist=3, seg=16)
    at ./arch/x86/include/asm/desc.h:361
361     {
(gdb) p idt_table   中斷向量表的開始位址
$2 = 0xffffffff825fb000 <idt_table>
```

4. **找出 system call 的進入點**

```
(gdb) b syscall_init
Breakpoint 3 at 0xffffffff81022d10: file arch/x86/kernel/cpu/common.c, line 1172.
(gdb) c
Continuing.

Breakpoint 2, _set_gate (gate=3, type=14, addr=0xffffffff81d0fae0 <int3>, dpl=3, ist=3, seg=16)
    at ./arch/x86/include/asm/desc.h:361
361     {
(gdb) p system_call
$4 = {<text variable, no debug info>} 0xffffffff81d0e100 <system_call>
```

system call的進入點

5. 印出 CPU 初始化的時候，「除以零」這個中斷的 interrupt service routine 的位址

第零號中斷「除以零的ISR位址」即為 ffffffff,81d0,f850

```
(gdb) b trap_init
Breakpoint 10 at 0xffffffff822df531: file arch/x86/kernel/traps.c, line 945.
(gdb) p/x *(struct gate_struct64*) 0xffffffff825fb000
$7 = {offset_low = 0xf850, segment = 0x10, ist = 0x0, zero0 = 0x0, type = 0xe, dpl = 0x0, p = 0x1,
  offset_middle = 0x81d0, offset_high = 0xffffffff, zero1 = 0x0}
```

6. 於「執行」 divideByZero 時，使用 gdb 對中斷處理常式（interrupt service routine，ISR 的位址）進行除錯

(a) 印出當下正在執行的 ISR 的位址

(b) 印出 ISR 的程式碼（組合語言）

```
(gdb) b divide_error
Breakpoint 11 at 0xffffffff81d0f850  file arch/x86/kernel/entry_64.S, line 1020.
(gdb) c
Continuing.

Breakpoint 11, <signal handler called>
(gdb) disassemble /m divide_error
Dump of assembler code for function divide_error:
1143    idtentry divide_error do_divide_error has_error_code=0
   0xffffffff81d0f853 <+3>:    pushq  $0xffffffffffffffff
   0xffffffff81d0f855 <+5>:    sub    $0x78,%rsp
   0xffffffff81d0f859 <+9>:    callq  0xffffffff81d0fd20 <error_entry>
   0xffffffff81d0f85e <+14>:   mov    %rsp,%rdi
   0xffffffff81d0f861 <+17>:   xor    %esi,%esi
   0xffffffff81d0f863 <+19>:   callq  0xffffffff8100688f <do_divide_error>
   0xffffffff81d0f868 <+24>:   jmpq   0xffffffff81d0fdd0 <error_exit>
   0xffffffff81d0f86d:   nopl
(%rax)
   0xffffffff81d0f870 <+0>:    data16 xchg %ax,%ax
End of assembler dump.
```

當下正在執行的ISR的位址

ISR的程式碼（組合語言）

(c) 印出該 ISR 所使用的堆疊的位址，請問此時 ISR 所使用的 stack 是從哪邊載入

```
(gdb) info registers
rax            0xa      10
rbx            0x400400 4195328
rcx            0x0      0
rdx            0x0      0
rsi            0x7ffe6c081d60   140730710891872
rdi            0x0      0
rbp            0x7ffe6c082020   0x7ffe6c082020
rsp            0xffff88000e58ffd8       0xffff88000e58ffd8     兩者相差40個bytes
               push ss、sp、flags、cs、ip後的位置
(gdb) p/x *(struct tss_struct *) 0xffff88000fa10ec0    ISR所使用的stack從這這開始載入
$8 = {x86_tss = {reserved1 = 0x0, sp0 = 0xffff88000e590000, sp1 = 0x0, sp2 = 0x0, reserved2 = 0x0, ist = {
      0xffff88000fa06000, 0xffff88000fa07000, 0xffff88000fa09000, 0xffff88000fa0a000, 0x0, 0x0, 0x0},
    reserved3 = 0x0, reserved4 = 0x0, reserved5 = 0x0, io_bitmap_base = 0x80}, io_bitmap = {
    0xffffffffffffffff <repeats 1025 times>}, stack = {0x0 <repeats 64 times>}}
```

7. （延續問題 5.a）請解釋 ISR 組合語言的意義

(a) error_entry

```
(gdb) disassemble /m error_entry
Dump of assembler code for function error_entry:
1368            cld
=> 0xffffffff81d0fd20 <+0>:     cld

1369            movq %rdi, RDI+8(%rsp)
   0xffffffff81d0fd21 <+1>:     mov    %rdi,0x78(%rsp)

1370            movq %rsi, RSI+8(%rsp)
   0xffffffff81d0fd26 <+6>:     mov    %rsi,0x70(%rsp)

1371            movq %rdx, RDX+8(%rsp)
   0xffffffff81d0fd2b <+11>:    mov    %rdx,0x68(%rsp)

1372            movq %rcx, RCX+8(%rsp)
   0xffffffff81d0fd30 <+16>:    mov    %rcx,0x60(%rsp)

1373            movq %rax, RAX+8(%rsp)
   0xffffffff81d0fd35 <+21>:    mov    %rax,0x58(%rsp)

1374            movq  %r8,  R8+8(%rsp)
   0xffffffff81d0fd3a <+26>:    mov    %r8,0x50(%rsp)        下略
```

可以發現 error_entry 的意思是利用 mov 指令將原本 user space

中的暫存器全部存進 kernel 所維護的 stack 之中，用來進行參數傳遞或在例外、信號處理完後，以便將原本進行到一半的程序繼續進行。

(b) do_divide_error

```
(gdb) disassemble /m do_divide_error
Dump of assembler code for function do_divide_error:
312     DO_ERROR(X86_TRAP_DE,     SIGFPE,  "divide error",            divide_error)
   0xffffffff8100688f <+0>:     push   %rbp
   0xffffffff81006890 <+1>:     mov    %rsp,%rbp
   0xffffffff81006893 <+4>:     mov    $0x8,%r8d
   0xffffffff81006899 <+10>:    mov    $0x0,%ecx
   0xffffffff8100689e <+15>:    mov    $0xffffffff8216ce59,%rdx
   0xffffffff810068a5 <+22>:    callq  0xffffffff81005d83 <do_error_trap>
   0xffffffff810068aa <+27>:    pop    %rbp
   0xffffffff810068ab <+28>:    retq
```

do_divide_error 會呼叫 do_error_trap，再由 do_error_trap 持續向下呼叫，最後會到達__send_signal，__send_signal 會在原本的 task 的結構中寫入 flag(並且會檢查這個 task 是否已經被設立了其它 flag)，以告訴這個 task 發生了 exception 的情況，之後再進行所發生的例外處理。