lab1 pdf

2019年3月15日 1:58

- E1/2:

AT&T	Intel
%eax	eax
movl %eax, %ebx	mov ebx eax(dst is always on right, load ebx)
movl $_booga, %eax	mov eax, _booga
movl $0xd00d, %ebx	mov ebx,d00d (load ebx)
movw %ax, %bx	mov bx, ax
imm32(baseptr, indexptr, indexscale)	[baseptr + indexptr*indexscale + imm32]

- E3：
  - **ljmp $PORT_MODE_CSEG, $protcseg**
    这一句话设置了CS和IP寄存器
  - boolloader最后一条在main.c里bootmain()，**((void (*)(void)) (ELFHDR->e_entry))();**设置断点后发现该地址为0x7d71，然后再si下一条指令，得其地址为0x1000c
  - 扇区的个数ELFHDR->e_phnum决定
- E5:

  bios里地址0x100000附近的值：

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
```

  loader里地址0x100000附近的值：

```
(gdb) b *0x7d71
Breakpoint 1 at 0x7d71
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d71:      call   *0x10018

Breakpoint 1, 0x00007d71 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x3000b812      0x220f0011      0xc0200fd8
```

```
jos@cosmic:~/jos-2019-spring$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001f3f  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000890  f0101f40  00101f40  00002f40  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00004909  f01027d0  001027d0  000037d0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00001a9d  f01070d9  001070d9  000080d9  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .note.gnu.property 0000001c  f0108b78  00108b78  00009b78  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .data         00009300  f0109000  00109000  0000a000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  6 .data.rel.local 00001000  f0113000  00113000  00014000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  7 .data.rel.ro.local 00000060  f0114000  00114000  00015000  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  8 .got          00000008  f0114060  00114060  00015060  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .got.plt      0000000c  f0114068  00114068  00015068  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 10 .bss          00000648  f0114080  00114080  00015080  2**5
                  CONTENTS, ALLOC, LOAD, DATA
 11 .comment      00000023  00000000  00000000  000156c8  2**0
                  CONTENTS, READONLY
```

0x100000开始的是.text，所以load结束后，存放的是executable instruction

- E6:

  正常情况下的执行情况：

```
(gdb) b *0x7c2d
Breakpoint 1 at 0x7c2d
(gdb) c
Continuing.
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87c32

Breakpoint 1, 0x00007c2d in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0x7c32:      mov    $0x10,%ax
```

  修改Makefrag里的link address的值后：（改为0x8c00）

```
(gdb) b *0x7c2d
Breakpoint 1 at 0x7c2d
(gdb) c
Continuing.
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x88c32

Breakpoint 1, 0x00007c2d in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0x48,%cs:(%esi)
```

  0x7c2d的代码为：

```
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
  7c2d: ea                    .byte 0xea
```

  可见0x7c2d在正常的link address情况下能正确转换为32-bit模式，否则就不行

- E7：

  决定paging的关键代码在如下位置：/kern/entry.S

```
# Turn on paging.
movl    %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0
```

  由于%cr0寄存器是不能or操作的，所以用eax作为中介将其置为1。若关闭它，即注释掉这部分代码，然后0x7d71开始运行，到movl $0x0, %ebp时就会报错，因为这时会把0xf010002c视作物理地址而不是虚拟地址，这个超过了RAM的范围，触发hardware exception

```
(gdb)
=> 0xf010002c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
(gdb)
Remote connection closed
```

```
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format
 tcp::26000 -D qemu.log  -S
qemu-system-i386: Trying to execute code outside RAM or ROM at 0xf010002c
```

- E8:

  要求我们能输出八进制，直接仿照附近几个输出数字的写，注意前面要添上0

```
case 'o':
    // Replace this with your code.
    num = getuint(&ap, lflag);
    putch('0', putdat);
    base = 8;
    goto number;
```

- E9:

  要求我们能支持正号前面有+号，所以对case d进行改动

```
// flag to pad on the front
case '+':
    plusflag = 1;
    goto reswitch;
case 'd':
    num = getint(&ap, lflag);
    if ((long long) num < 0) {
        putch('-', putdat);
        num = -(long long) num;
    }else if(plusflag && num){
        putch('+', putdat);
    }
    base = 10;
    goto number;
```

1. console.c提供cputchar，printf.c在putch里使用它来将字符写入putdat
2. /***** Text-mode CGA/VGA display output *****/

   简单来说这是CGA/VGA显示的判断，如果屏幕满了就向下滚动一行，并且将新出来的填为黑色。
3. fmt指向的是格式化字符串指针，ap是指向的是用来替换第一个参数的的指针。

   int x = 1, y = 3, z = 4; cprintf("x %d, y %x, z %d\n", x, y, z);
   vcprintf (fmt=0xf010238d "x %d, y %x, z %d\n", ap=0xf0110e34 "\001")
   cons_putc (c=120) x
   cons_putc (c=32)
   va_arg(0xf0110e34, int) before …34, after …38
   cons_putc (c=49) 1
   cons_putc (c=44) ,
   cons_putc (c=32)
   cons_putc (c=121) y
   cons_putc (c=32)
   va_arg(0xf0110e38, int) before …38, after …3c
   cons_putc (c=51) 3
   cons_putc (c=44) ,
   cons_putc (c=32)
   cons_putc (c=122) z
   cons_putc (c=32)
   va_arg(0xf0110e3c, int) before …3c, after …40

cons_putc (c=52) 4
cons_putc (c=10) \n
cprintf (fmt=0xf01025af "%s")

```
(gdb) i s
#0  cons_putc (c=120) at kern/console.c:70
#1  0xf01007d8 in cputchar (c=120) at kern/console.c:458
#2  0xf0100d35 in putch (ch=120, cnt=0xf0110dfc) at kern/printf.c:12
#3  0xf010167c in vprintfmt (putch=0xf0100d17 <putch>, putdat=0xf0110dfc,
    fmt=0xf010238d "x %d, y %x, z %d\n", ap=0xf0110e34 "\001") at lib/printfmt.c:106
#4  0xf0100d71 in vcprintf (fmt=0xf010238d "x %d, y %x, z %d\n", ap=0xf0110e34 "\001")
    at kern/printf.c:21
#5  0xf0100d8b in cprintf (fmt=0xf010238d "x %d, y %x, z %d\n") at kern/printf.c:32
#6  0xf0100be3 in monitor (tf=0x0) at kern/monitor.c:219
#7  0xf01001c5 in i386_init () at kern/init.c:52
#8  0xf010003e in relocated () at kern/entry.S:80
(gdb) x/8x 0xf0110e34
0xf0110e34:     0x00000001      0x00000003      0x00000004      0xf0102544
0xf0110e44:     0xf0110fdf      0xf0110e78      0xf0100d71      0xf0100d17
```

1. unsigned int i = 0x00646c72; cprintf("H%x Wo%s", 57616, &i)

   57616按照十六进制即为0xe110，根据ap得到第二个参数为0xf0110e9c，按照%s输出即找出该数字对应的ASCII码，即为rld。

   如果改成big-endian的话，第一个参数应该是0x10e10000，第二个参数用**htonl**转换大小端即可。

```
(gdb) i  stack
#0  cons_putc (c=72) at kern/console.c:70
#1  0xf01007d8 in cputchar (c=72) at kern/console.c:458
#2  0xf0100b10 in putch (ch=72, cnt=0xf0110dfc) at kern/printf.c:12
#3  0xf01014c2 in vprintfmt (putch=0xf0100af2 <putch>, putdat=0xf0110dfc,
    fmt=0xf0102153 "H%x Wo%s", ap=0xf0110e34 <incomplete sequence \341>) at lib/printfmt.c:124
#4  0xf0100b4c in vcprintf (fmt=0xf0102153 "H%x Wo%s", ap=0xf0110e34 <incomplete sequence \341>)
    at kern/printf.c:21
#5  0xf0100b66 in cprintf (fmt=0xf0102153 "H%x Wo%s") at kern/printf.c:32
#6  0xf01009b3 in monitor (tf=0x0) at kern/monitor.c:167
#7  0xf01001c5 in i386_init () at kern/init.c:52
#8  0xf010003e in relocated () at kern/entry.S:80
(gdb) x/8x 0xf0110e34
0xf0110e34:     0x0000e110      0xf0110e9c      0xf011404c      0x00000005
0xf0110e44:     0xf0110fdf      0xf0110e78      0xf0100b4c      0xf0100af2
(gdb) x/s 0xf0110e9c
0xf0110e9c:     "rld"
```

1. cprintf("x=%d y=%d", 3)

   output: x=3 y=-267317640

   参数不够时，ap+4后得到栈上的下一个的值，强制类型转换为signed int后为-267317640输出即可。

```
(gdb) i s
#0  cons_putc (c=120) at kern/console.c:70
#1  0xf01007d8 in cputchar (c=120) at kern/console.c:458
#2  0xf0100b02 in putch (ch=120, cnt=0xf0110e0c) at kern/printf.c:12
#3  0xf01014b4 in vprintfmt (putch=0xf0100ae4 <putch>, putdat=0xf0110e0c,
    fmt=0xf0102133 "x=%d y=%d", ap=0xf0110e44 "\003") at lib/printfmt.c:124
#4  0xf0100b3e in vcprintf (fmt=0xf0102133 "x=%d y=%d", ap=0xf0110e44 "\003")
#5  0xf0100b58 in cprintf (fmt=0xf0102133 "x=%d y=%d") at kern/printf.c:32
#6  0xf01009a5 in monitor (tf=0x0) at kern/monitor.c:168
#7  0xf01001c5 in i386_init () at kern/init.c:52
#8  0xf010003e in relocated () at kern/entry.S:80
(gdb) x/8x 0xf0110e44
0xf0110e44:     0x00000003      0xf0110e78      0xf0100b3e      0xf0100ae4
0xf0110e54:     0xf0110e6c      0xf0101d9c      0xf0110ea8      0xf0101d9c
```

1. 在每个ap参数链表的第一个都增加一个参数表示参数的个数，然后每次先判断参数的个数再输出。

- E10:

每次%n都读入一个byte，这里要检测是否有overflow的情况。putdat在这里就是传入的参数的长度。

具体来说就是**cprintf("%s%n", ntest, &chnum1);** 里ntest的size-1（测试样例要求）。只要检测

该大小是否超过了一个byte所能表示的最大值，即255 - 1 = 254，然后将chnum置为成功读入的byte数；如果不满足就将chnum置为-1。

```
const char *null_error = "\nerror! writing through NULL pointer! (%n argument)\n";
const char *overflow_error = "\nwarning! The value %n argument pointed to has been overflowed!\n";
//data to replace %?, one byte max
void* ptr = va_arg(ap, void*);
//cprintf("ptr: %d\n", *ptr);
if(!ptr){
    printfmt(putch,putdat,"%s",null_error);
}else{
    //cprintf("putdat: %d\n", *(unsigned int*)putdat);
    //sizeof(...) - 1
    if( *(unsigned int *)putdat > 255 - 1 ){
        printfmt(putch,putdat,"%s",overflow_error);
        *(char*)ptr = -1;
    }else
        *(char*)ptr = *(char*)putdat;
}
break;
```

- E11:

  这里要实现printf里的左对齐功能，即先把实际字符输出来，再根据对齐长度判断右侧是否需要补空格。

```
if(padc == '-'){
    padc = ' ';
    printnum(putch,putdat,num,base,0,padc);
    // put in a blank
    while (--width > 0)
        putch(padc, putdat);
    return;
}
```

- E12:

  内核为栈提前留位置：/kern/entry.S

```
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl    $0x0,%ebp           # nuke frame pointer

# Set the stack pointer
movl    $(bootstacktop),%esp
```

- E13/14/15

  要求我们对ebp实现回溯，即找到test_backtrace的地址。利用read_ebp()函数获得ebp寄存器的值，根据文档的提示，当ebp不等于0时，把ebp-ebp+28直接的内存地址按照(int*)的格式输出，然后更新ebp的值。

  然后要提供函数名、文件名、eip的行号，在kdebug文件里增加debuginfo函数里stab_binsearch对line的处理。

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    cprintf("Stack backtrace\n");
    uint32_t ebp = read_ebp();
    cprintf("ebp : %x\n", ebp);
    while(ebp != 0){
        uint32_t eip = *(int*)(ebp+4);
        cprintf("  eip %08x  ebp %08x  args %08x %08x %08x %08x %08x\n",
                eip, ebp,
                *(int*)(ebp+8),*(int*)(ebp+12),*(int*)(ebp+16),*(int*)(ebp+20),*(int*)(ebp+24));
        struct Eipdebuginfo info;
        if(debuginfo_eip(eip,&info)>=0){
            cprintf("         %s:%d %.*s+%d\n",
            info.eip_file, info.eip_line,
            info.eip_fn_namelen, info.eip_fn_name, eip-info.eip_fn_addr);
        }
        ebp = *(int*)ebp;
    }
    overflow_me();
        cprintf("Backtrace success\n");
    return 0;
```

- E16:

  该部分要求写出一个像ICS里手动控制ret地址跳转地址的过程。简单来说就是修改start_overflow的ret_addr，让其指向do_overflow()的地址，然后在该ret_addr上4个byte里存放原来start_overflow函数的ret_addr，目的是让do_overflow完成其过程后能跳转回start_overflow()的下一行以继续下面的流程。

```
void
start_overflow(void)
{
    // You should use a techique similar to buffer overflow
    // to invoke the do_overflow function and
    // the procedure must return normally.

    // And you must use the "cprintf" function with %n specifier
    // you augmented in the "Exercise 9" to do this job.

    // hint: You can use the read_pretaddr function to retrieve
    //       the pointer to the function call return address;

    char str[256] = {};
    int nstr = 0;

    //Lab1 Code
    char* pret_addr = (char *) read_pretaddr();
    //char* overflow_addr = (char*) ((uint32_t)do_overflow);
    uint32_t overflow_addr = (uint32_t) do_overflow;
    int i;
    for(i = 0; i < 4; ++i){
        //store original ret_addr in before+4
        memset(pret_addr+4+i, *(pret_addr+i), 1);
    }
    for(i = 0; i < 4; ++i){
        //set overflow ret_addr
        memset(pret_addr+i, (overflow_addr>>(8*i)) & 0xFF, 1);
    }
}
```

- E17:

  该部分是添加一个指令，能让其算出参数指令所花的时间。新增一个关于新指令的mon_time函数，利用网上查到的rdtsc样例，内嵌汇编代码获得eax,ebx的值来代表64bit的数，最后使其

有类似计时的功能。

```
static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
    { "kerninfo", "Display information about the kernel", mon_kerninfo },
    { "time", "Display time about kernel", mon_time},
};
```

```
uint64_t rdtsc(){
    uint32_t lo,hi;

    __asm__ __volatile__
    (
    "rdtsc":"=a"(lo),"=d"(hi)
    );
    return (uint64_t)hi<<32|lo;
}
int
mon_time(int argc, char **argv, struct Trapframe *tf){
    uint64_t begin = 0, end = 0;
    char c[256];
    bool found = false;
    int i;
    //cprintf("%s\n", argv[0]);
    for (i = 0; i < ARRAY_SIZE(commands); i++) {
        if (strcmp(argv[1], commands[i].name) == 0){
            begin = rdtsc();
            commands[i].func(argc-1, argv+1, tf);
            end = rdtsc();
            strcpy(c, argv[0]);
            found = true;
            break;
        }
    }
    if(found)
        cprintf("%s cycles:%d\n", c, end - begin);
    else
        cprintf("%s\n", "Command not found!");
    return 0;
}
```