Lab1 reports

YANG Weikun, 18th Sept 2013 21:00 UTC+8000

1100012442

Ex. 1

Reading AT&T x86 asm syntax - done

Ex. 2

Tracing the very first instructions executed by CPU on start up:

```
# first instructions
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
[f000:e05b] 0xfe05b:    jmp    0xfc85e

# load CR0
[f000:c85e] 0xfc85e:    mov    %cr0,%eax
# enable cache and write-back
[f000:c861] 0xfc861:    and    $0x9fffffff,%eax
[f000:c867] 0xfc867:    mov    %eax,%cr0

# disable interrupt
[f000:c86a] 0xfc86a:    cli
# clear direction flag
[f000:c86b] 0xfc86b:    cld

# enable Non-Maskable-Interrupt
# and reads a CMOS Shutdown Status register
# ref: http://wiki.osdev.org/CMOS
# details: http://www.bioscentral.com/misc/cmosmap.htm
[f000:c86c] 0xfc86c:    mov    $0x8f,%eax
[f000:c872] 0xfc872:    out    %al,$0x70
[f000:c874] 0xfc874:    in     $0x71,%al

# make sure computer is 'Power-On'
[f000:c876] 0xfc876:    cmp    $0x0,%al
[f000:c878] 0xfc878:    jne    0xfc88d

# setting up stack [SS:SP]=[0000:7000]
[f000:c87a] 0xfc87a:    xor    %ax,%ax
[f000:c87c] 0xfc87c:    mov    %ax,%ss
[f000:c87e] 0xfc87e:    mov    $0x7000,%esp
[f000:c884] 0xfc884:    mov    $0xf4b2c,%edx
[f000:c88a] 0xfc88a:    jmp    0xfc719
[f000:c719] 0xfc719:    mov    %eax,%ecx
```

```
[f000:c71c] 0xfc71c:    cli
[f000:c71d] 0xfc71d:    cld

[f000:c71e] 0xfc71e:    mov    $0x8f,%eax
[f000:c724] 0xfc724:    out    %al,$0x70
[f000:c726] 0xfc726:    in     $0x71,%al        # 0x92 is System Control
Port A
[f000:c728] 0xfc728:    in     $0x92,%al        # enable A20, so address
does not wrap back.
[f000:c72a] 0xfc72a:    or     $0x2,%al
[f000:c72c] 0xfc72c:    out    %al,$0x92

# load Interrupt-Descriptor-Table and
# Global-Descriptor-Table
[f000:c72e] 0xfc72e:    lidtw  %cs:-0x31cc
[f000:c734] 0xfc734:    lgdtw  %cs:-0x3188
[f000:c73a] 0xfc73a:    mov    %cr0,%eax
# enter protected-mode
[f000:c73d] 0xfc73d:    or     $0x1,%eax
[f000:c741] 0xfc741:    mov    %eax,%cr0
[f000:c744] 0xfc744:    ljmpl  $0x8,$0xfc74c
# load 0x10 to DS,ES,SS,FS,GS
0xfc74c:    mov    $0x10,%eax
0xfc751:    mov    %eax,%ds
0xfc753:    mov    %eax,%es
0xfc755:    mov    %eax,%ss
0xfc757:    mov    %eax,%fs
0xfc759:    mov    %eax,%gs

# and much more
...
```

Ex.3

break at 0x7c00:

```
0x7c00: cli
0x7c01: cld
0x7c02: xor    %ax,%ax
0x7c04: mov    %ax,%ds
0x7c06: mov    %ax,%es
0x7c08: mov    %ax,%ss
0x7c0a: in     $0x64,%al
0x7c0c: test   $0x2,%al
0x7c0e: jne    0x7c0a
0x7c10: mov    $0xd1,%al
0x7c12: out    %al,$0x64
0x7c14: in     $0x64,%al
```

```
0x7c16: test   $0x2,%al
0x7c18: jne    0x7c14
0x7c1a: mov    $0xdf,%al
0x7c1c: out    %al,$0x60

# load GDT
0x7c1e: lgdtw  0x7c64

# Set CPU to Protected Mode
0x7c23: mov    %cr0,%eax
0x7c26: or     $0x1,%eax
0x7c2a: mov    %eax,%cr0

# Switch to 32-bit Protected Mode
# $0x8 here means the 2nd entry in GDT (each entry is 8 bytes long)
0x7c2d: ljmp   $0x8,$0x7c32

# Set segment registers to 3rd entry of GDT
0x7c32: mov    $0x10,%ax
0x7c36: mov    %eax,%ds
0x7c38: mov    %eax,%es
0x7c3a: mov    %eax,%fs
0x7c3c: mov    %eax,%gs
0x7c3e: mov    %eax,%ss
# Use code as stack.
0x7c40: mov    $0x7c00,%esp
# Call into bootmain()
0x7c45: call   0x7d0b

# Now in C code with proper stack handling
0x7d0b: push   %ebp
0x7d0c: mov    %esp,%ebp
0x7d0e: push   %esi
0x7d0f: push   %ebx
# readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
# read 4K bytes from disk to address 0x10000
0x7d10: push   $0x0
0x7d12: push   $0x1000
0x7d17: push   $0x10000
0x7d1c: call   0x7cd2

    # inside readseg(uint32_t pa, uint32_t count, uint32_t offset)
    0x7cd2: push   %ebp
    0x7cd3: mov    %esp,%ebp
    0x7cd5: push   %edi
    0x7cd6: push   %esi
    0x7cd7: push   %ebx

    # load parameter uint32_t pa
    0x7cd8: mov    0x8(%ebp),%ebx
    # load parameter uint32_t offset
```

```
    0x7cdb: mov    0x10(%ebp),%esi
    # load parameter uint32_t count
    0x7cde: mov    0xc(%ebp),%edi

    # end_pa = pa + count; count never used later,
    # so end_pa is using %edi
    0x7ce1: add    %ebx,%edi

    # pa &= ~(SECTSIZE - 1); rounding to sector boundary
    0x7ce3: and    $0xfffffe00,%ebx

    # offset = (offset / SECTSIZE) + 1; calc. sector number
    0x7ce9: shr    $0x9,%esi
    0x7cec: inc    %esi

    # loop, read sectors
    0x7ced: jmp    0x7cff
    0x7cef: push   %esi
    0x7cf0: push   %ebx

    # readsect((uint8_t*) pa, offset);
    0x7cf1: call   0x7c81

        ......(readsect)

    # pa += SECTSIZE;
    0x7cf6: add    $0x200,%ebx
    # offset++;
    0x7cfc: inc    %esi
    # balance stack
    0x7cfd: pop    %eax
    0x7cfe: pop    %edx
    0x7cfe: pop    %edx
    # while (pa < end_pa) ...
    0x7cff: cmp    %edi,%ebx
    0x7d01: jb     0x7cef

    # restore registers and return
    0x7d03: lea    -0xc(%ebp),%esp
    0x7d06: pop    %ebx
    0x7d07: pop    %esi
    0x7d08: pop    %edi
    0x7d09: pop    %ebp
    0x7d0a: ret

# back to bootmain
# stack balancing
0x7d21: add    $0xc,%esp

# if (ELFHDR->e_magic != ELF_MAGIC)
     goto bad;
```

```
0x7d24: cmpl   $0x464c457f,0x10000
0x7d2e: jne    0x7d69

# ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff); in %ebx
# eph = ph + ELFHDR->e_phnum; in %esi
0x7d30: mov    0x1001c,%ebx
0x7d36: add    $0x10000,%ebx
0x7d3c: movzwl 0x1002c,%eax
0x7d43: shl    $0x5,%eax
0x7d46: lea    (%ebx,%eax,1),%esi

# begin of for loop
0x7d49: jmp    0x7d5f
0x7d4b: pushl  0x4(%ebx)
0x7d4e: pushl  0x14(%ebx)
0x7d51: pushl  0xc(%ebx)
0x7d54: call   0x7cd2
# ph++;
0x7d59: add    $0x20,%ebx

0x7d5c: add    $0xc,%esp
# ph < eph
0x7d5f: cmp    %esi,%ebx
0x7d61: jb     0x7d4b

# call function pointer of kernel entry
0x7d63: call   *0x10018

......
```

Q&A

1.

The processor starts executing 32-bit code since

```
    0x7c2d: ljmp   $0x8,$0x7c32
```

the switch is cause by setting the least-significant-bit of CR0 by

```
0x7c23: mov    %cr0,%eax
0x7c26: or     $0x1,%eax
0x7c2a: mov    %eax,%cr0
```

2.

The last instruction that the boot loader executed is

```
0x7d63: call   *0x10018
```

and the first instruction of the kernel is

```
0x10000c:   movw   $0x1234,0x472
```

3.

first instruction of kernel is in **kern/entry.S, line 44**, and is loaded into memory at **0x10000c**

4.

The bootloader finds the number of program headers in **e_phnum**, starting at **e_phoff**

```
strcut Elf{
    ...
    uint32_t e_phoff;
    uint16_t e_phnum;
    ...
}
```

Then for each program header, the bootloader obtains its size from **uint32_t p_memsz**

```
struct Proghdr {
    ...
    uint32_t p_offset;
    uint32_t p_pa;
    uint32_t p_memsz;
    ..
};
```

struct definitions are from **inc/elf.h**

Ex.4

comments on *pointers.c*

```
int a[4];                   // a is allocated on stack
int *b = malloc(16);        // b's contents is allocated in heap
int *c;                     // c is uninitialised
```

```
int i;
printf("1: a = %p, b = %p, c = %p\n", a, b, c);
# output 1: a = 0x7fff5e0ae8e0, b = 0x7fa1f1c03ab0, c = 0x101b51000

c = a;                      // c now points to the same array as a
for (i = 0; i < 4; i++)
    a[i] = 100 + i;
c[0] = 200;                 // c[0] writes to a[0]

printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n", ...

# output 2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103


c[1] = 300;                 // c[1] is a[1]

*(c + 2) = 301;             // *(c+2) is c[2] or a[2]

3[c] = 302;                 // 3[c] is c[3] ? this is uncommon syntax

printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n", ...

# output 3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302


c = c + 1;                  // c now points to a[1]

*c = 400;

printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n", ...

# output 4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302


c = (int *) ((char *) c + 1);   // c now points to (char*)a[5]

*c = 500;                       // writing 4 bytes 0x000001F2 to memory at c

printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n", ...

# output 5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302


b = (int *) a + 1;              // b points to a[1], original contents lost

c = (int *) ((char *) a + 1);   // c points to (char*)a[1]

printf("6: a = %p, b = %p, c = %p\n", a, b, c);

# output 6: a = 0x7fff5e0ae8e0, b = 0x7fff5e0ae8e4, c = 0x7fff5e0ae8e1
```

output of **$ i386-jos-elf-objdump -h obj/kern/kernel**

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000184e  f0100000  00100000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000704  f0101860  00101860  00002860  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00004555  f0101f64  00101f64  00002f64  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00008b12  f01064b9  001064b9  000074b9  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f010f000  0010f000  00010000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000644  f0119300  00119300  0001a300  2**5
                  ALLOC
  6 .comment      00000011  00000000  00000000  0001a300  2**0
                  CONTENTS, READONLY
```

the kernel expects itself to be loaded to ***physical*** address 0x00100000, and execute at (***virtual***) address 0xf0100000

Ex. 5

After **boot/Makefrag**'s link address is changed from 0x7C00 to 0x7D00, the instructions generated will believe that it's IP starts from 0x7D00, and that related data is also 0x100 bytes away. However the loading of the bootloader is done by **BIOS**, to a hard-wired address of 0x7C00. So when it comes to this particular instruction that access somewhere in the *memory*, things break apart:

```
[   0:7c1e] 0x7c1e: lgdtw  0x7d64
```

At physical address 0x7D64 there is plain rubbish. So a incorrect GDT gets loaded. Then our bootloader tries to switch to 32-bit protected mode by a *ljmp* that puts out processor into real trouble

```
[   0:7c2d] 0x7c2d: ljmp   $0x8,$0x7d32
```

Ex. 6

Break at **0x7c00** and examine memory at 0x100000 (this is a **physical** address) by

```
x/8x 0x100000
```

will show uninitialised memory (zeros).

Then break at kernel's entry point, memory at 0x100000 is kernel's instructions.

```
# kern/entry.S, line 44
0x10000c:   movw   $0x1234,0x472
...
```

Ex. 7

Before paging is enabled, both 0x00100000 and 0xf0100000 are *physical* addresss, holding kernel's code and uninitialised memory respectively.

After paging is enabled, those two addresses are both *virtual*, and maps to the same *physical* address 0x00100000.

If paging is not enabled by removing this instruction in **kern/entry.S line 62**:

```
movl    %eax, %cr0
```

the following instructions will jump to ***physical*** memory at **$(bootstacktop)** which is 0xf010002c in value, out of the actual memory boundary of my QEMU.

```
mov $relocated, %eax
jmp *%eax
```

Ex. 8

code in *vprintfmt(...)*, in **printfmt.c**:

```
...
    case 'o':
        num = getuint(&ap, lflag);
        base = 8;
        goto number;
...
```

Q&A

1. console.c encapsulates device details and export three generic character I/O functions:

```
void
cputchar(int c);

int
getchar(void);

int
iscons(int fdnum)
```

printf.c uses *cputchar()*.

2.

```
1  if (crt_pos >= CRT_SIZE) {
2          int i;
3          memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) *
sizeof(uint16_t));
4          for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5                  crt_buf[i] = 0x0700 | ' ';
6          crt_pos -= CRT_COLS;
7  }
```

CRT_SIZE is the product of the number of rows and columns (25x80) in this default text mode display. **crt_pos** is the cursor position. when the cursor moves out of the bottom line, we should move the contents of the text buffer ***up*** by one row. That is done by the **memcpy** call. Then the bottom line is filled with empty characters again.

3.

In *cprintf()*, ***fmt*** points to a const string at $0xf0101b65, ***ap*** points to the parameter on stack just below *fmt*.

Tracing into *cons_putc*, *va_arg* and *vcprintf* is way too exhaustive. Macro definitions are explained instead, to prove that I got all these:

```
// sizeof(n), rounded up to a multiple of sizeof(int)
#define _INTSIZEOF(n)   ((sizeof(n)+sizeof(int)-1)&~(sizeof(int) - 1) )

// make a va_list pointing right after `v` (which should be the first
variable in list)
#define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) )

// get the current argument, and moves `ap` to point to the next
#define va_arg(ap,t) ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )

// cleanup
#define va_end(ap)    ( ap = (va_list)0 )
```

GCC's approach towards variable arguments is easy to understand and indeed effective.

4.

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

int i is 4 bytes, in little-endian encoding which reads "*rld\0*" in ASCII. And 57616 in hex is *0xE110*. So the output is **"He110 world"**

If x86 were big-endian, *int i* as a string would be "*\0dlr*" (will not be printed since cprintf recognize '\0' as terminator), and hex representation of 57616 would not change.

5.

This malformed call to *cprintf*:

```
cprintf("x=%d y=%d", 3);
```

when accessing 'y', *va_arg* will access 4 bytes on stack, just below '3' was pushed. This region could be saved %ebp, or local variables.

6.

Of course it would still be possible to have functions with variable number of parameters, even if the parameter list is reversed.

a) Modify **va_start(ap, ...)** so **ap** references the first parameter in the variable list.

b) Modify **va_arg(ap, type)** so **ap** moves to the next parameter (address decreases) everytime *va_arg* is called.

c) Modify other macros accordingly (not much to change).

Ex. 9

Kernel Stack is initialised in *kern/entry.S* line 74 and 77:

```
movl    $0x0,%ebp
# $(bootstacktop) is 0xf0118000(virtual) or 0x00118000(physical)
movl    $(bootstacktop),%esp
```

The entire stack space grows downwards from 0x00118000, to somewhere above 0x00100000. About 96KB.

Ex. 10

test_backtrace is at 0xf010004a

test_backtrace saves *%ebp*, *%ebx*, and subtracts %esp by 20 , which is five 4-byte words that include two parameters of *cprintf*, one parameter for a recursive call (reused on stack top), three parameters for *mon_backtrace*, plus the return address pushed on stack at recursive calls.

In total, **EIGHT** 4-byte words (32 bytes) for each nesting level.

Ex. 11

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // base pointer
    unsigned int *bp;

    struct Eipdebuginfo info;
    cprintf("Stack backtrace:\n");

    // load %ebp to bp, using inline asm
    __asm__ ("movl %%ebp, %0":"=r"(bp) :);

    // entry.S set the first %ebp to be zero, that's how we identify the
end.
    while (bp != 0){

        // display ebp, return address, and parameters on stack
        cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n",
            bp, bp[1], bp[2], bp[3], bp[4], bp[5], bp[6]);

        // load and display debug info
        debuginfo_eip(bp[1], &info);
        cprintf("         %s:%d: %.*s+%d\n",
            info.eip_file, info.eip_line,
            info.eip_fn_namelen, info.eip_fn_name,
            bp[1]-(unsigned int)info.eip_fn_addr);

        // bp points to the previous stack frame's saved %ebp,
        // according to typical x86 calling conventions.
        bp = (unsigned int*)(bp[0]);
    }
    return 0;
}
```

Ex. 12

loading line numbers: inside *debuginfo_eip*

```
...
// N_SLINE - text segment line number
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
if (lline <= rline){
    // line numbers stored in `uint16_t n_desc'
    info->eip_line = stabs[lline].n_desc;
}else{
    return -1;
}
...
```