CS 444/544 OS II
Lab Tutorial #2 (part 2)
Stack and Calling Convention

Acknowledgement: Slides drawn heavily from Yeongjin Jiang
Contents

• Stack and calling convention

• Exercise 7-11
Exercise 7: Virtual Memory

• 0xf0000000 == KERNBASE

• Virtual address 0xf0000000 ~ 0xffffffff
  • Access physical address at (Virtual address – KERNBASE)

• E.g.,
  • 0xf0123456 -> 0x123456
  • 0xf0000001 -> 0x1
Exercise 8

• Read lib/printfmt.c, for vprintfmt()

• Look at cases ‘x’ and ‘u’ as an example of hexadecimal and decimal

• Implement the case ‘o’
  • Similar to ‘x’ and ‘u’
  • It’s easy...
Exercise 9 ~ 11: Stack Backtrace

• Must understand how stack works in x86.

Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the test_backtrace function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words? NOTE. you’ll have to manually translate all breakpoint and memory addresses to linear addresses.
Function call in x86

In kern/init.c

```c
38 // Test the stack backtrace function (lab 1 only)
39 test_backtrace(5);

10 // Test the stack backtrace function (lab 1 only)
11 void
12 test_backtrace(int x)
13 {
14 cprintf("entering test_backtrace %d\n", x);
15 if (x > 0)
16     test_backtrace(x-1);
17 else
18     mon_backtrace(0, 0, 0);
19     cprintf("leaving test_backtrace %d\n", x);
20 }
```

test_backtrace(5) -> test_backtrace(4) -> test_backtrace(3) -> 2 -> 1 -> mon_backtrace(0,0,0)...

How this recursion can work in x86 computer?
x86 Stack

• All local variables are stored in the stack.
  %ebp

• A function call creates a new stack
  • Start with ebp, ends with esp
  ebp-8
  ebp-c
  ebp-10

• Grows downward!
  • Push(A), subtract 4 from esp and store A to there...
  • Pop, get the value at esp and add 4 to esp
  esp+4
  esp
Function call example

my_function(MY_ARG1, MY_ARG2) {
    int A;
    int B;
    int C;
    other_function(ARG1, ARG2)
}

%ebp
Saved EBP
Local C
Local B
Local A
Return Addr
ebp-8
ebp-c
ebp-10
esp
esp+4
ARG 2
ARG 1
%esp
How x86 manages stack?

- Let’s debug calling test_backtrace
- Set the breakpoint at *i386_init
How x86 manages stack?

- Examine instructions...

```
0x010000a6 <i386_init>:  push  %ebp
0x010000ad <i386_init>+:  call  0x010001bc <__x86.get_pc_thunk.bx>
0x010000b2 <i386_init+12>:  add  $0x11256,%ebx
0x010000b8 <i386_init+18>:  mov  $0x0f113060,%edx
0x010000be <i386_init+24>:  mov  $0x0f1136a0,%eax
0x010000c4 <i386_init+30>:  sub  %edx,%eax
0x010000c6 <i386_init+32>:  push  %eax
0x010000c7 <i386_init+33>:  push  $0x0
0x010000c9 <i386_init+35>:  push  %edx
0x010000ca <i386_init+36>:  call  0x0100179a <memcpy>
0x010000cf <i386_init+41>:  call  0x0100611 <cons_init>
0x010000d4 <i386_init+46>:  add  $0x8,%esp
0x010000d7 <i386_init+49>:  push  $0x1aac
0x010000dc <i386_init+54>:  lea   -0x0f6f1(%ebx),%eax
0x010000e2 <i386_init+60>:  push  %eax
0x010000e3 <i386_init+61>:  call  0x01000b86 <printf>
0x010000e8 <i386_init+66>:  movl  $0x5,(%esp)
0x010000ef <i386_init+73>:  call  0x0100040 <test_backtrace>
```

```
0x010000f4 <i386_init+78>:  add  $0x10,%esp
0x010000f7 <i386_init+81>:  sub  $0xc,%esp
0x010000fa <i386_init+84>:  push  $0x0
0x010000fc <i386_init+86>:  call  0x01009ce <monitor>
```
How x86 manages stack?

- **Call**
  - Push addr of next instr.
  - To return to there after func().
- **Jump to target.**
How x86 manages stack?

• In test backtrace

```
gdb-peda$ disas test_backtrace
Dump of assembler code for function test_backtrace:
0xf0100040 <+0>: push %ebp
0xf0100041 <+1>: mov %esp,%ebp
0xf0100043 <+3>: push %esi
0xf0100044 <+4>: push %ebx
```

![Diagram showing stack management in x86]

```c
MY_ARG1 (5)
```

- Return Addr: 0xf01000f4
- Saved EBP
- Saved ESI
- Saved EBX
How x86 manages stack?

• Call

```
0xf0100098 <+88>:  lea    -0x1(%esi),%eax
0xf010009b <+91>:  push   %eax
0xf010009c <+92>:  call   0xf0100040 <test_backtrace>
```

test_backtrace(x-1)
How x86 manages stack?

```c
if (x > 0)
    test_backtrace(x-1);
else
    mon_backtrace(0, 0, 0);
```
How x86 manages stack?

```
0xf0100091 <+81>:    pop    %ebx
0xf0100092 <+82>:    pop    %esi
0xf0100093 <+83>:    pop    %ebp
0xf0100094 <+84>:    ret

ret == pop %eip
```
How x86 manages stack?

- Call

  test_backtrace(x-1)

  \[
  \begin{align*}
  \text{Return Addr} & \quad 0xf01000f4 \\
  \text{Saved EBP} \\
  \text{Saved ESI} \\
  \text{Saved EBX} \\
  \text{ARG 1 = x-1}
  \end{align*}
  \]

  \[
  \begin{align*}
  0xf0100098 & \quad \text{lea} -0x1(%esi),%eax \\
  0xf010009b & \quad \text{push} \quad %eax \\
  0xf010009c & \quad \text{call} \quad 0xf0100040 \quad \text{<test_backtrace>}
  \end{align*}
  \]
x86 Stack

• ebp points the boundary of the stack (bottom)
• esp points to the other boundary of the stack (top)

• ebp[0] stores saved ebp
• ebp[1] stores return address
• ebp[2] stores 1\textsuperscript{st} argument
• ebp[3] stores 2\textsuperscript{nd} argument
• ...
Hint – Exercise 11

Stack backtrace:

```
ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
kern/monitor.c:143: monitor+106
```

• `int *ebp = (int *) read_ebp();`
  • `cprintf("ebp %08x", ebp)...`

• EIP == return address
  • `ebp[1] – why?`

• Args?
  • Print `ebp[2 ~ 6]...`