

OS Assignment 1: Documentation

October 30, 2013

Group No.12

Group members: Ma Xuan(1110580), Wang Shaoyao(1110539)
Wu Zhengwei(1110548), Yuan Zhaozheng(1110556), Zhang Chao(1110557)

Development Environment Setup

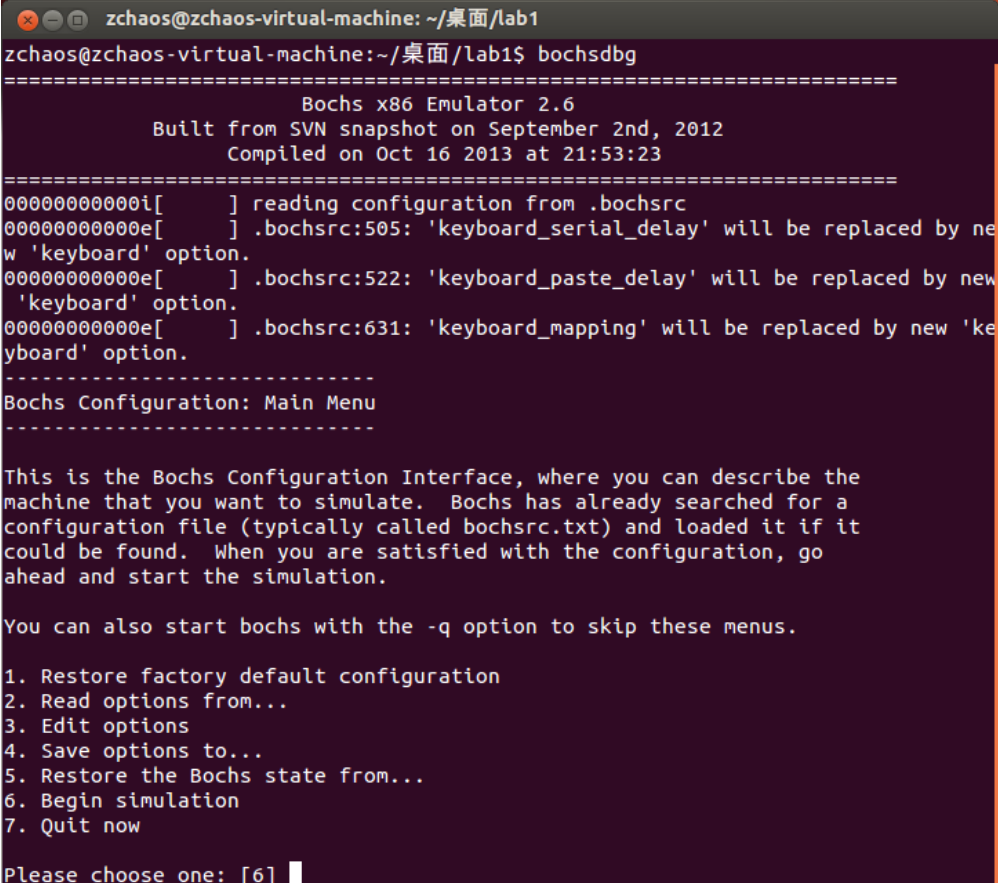
Some packages need to be installed in advance:

```
sudo apt-get install build-essential
sudo apt-get install libx11-dev
sudo apt-get install libxrandr-dev
sudo apt-get install xorg-dev
5 sudo apt-get install libgtk2.0-dev
sudo apt-get install vgabios
```

Configure and install Bochs-2.6:

```
tar -xvf bochs-2.6.tar.gz
cd bochs-2.6
./configure --enable-debugger --enable-disasm
make
5 sudo make install
make clean
```

Run Bochs in Terminal:



```
zchaos@zchaos-virtual-machine: ~/桌面/lab1
zchaos@zchaos-virtual-machine:~/桌面/lab1$ bochsdbg
=====
                Bochs x86 Emulator 2.6
        Built from SVN snapshot on September 2nd, 2012
        Compiled on Oct 16 2013 at 21:53:23
=====
00000000000i[      ] reading configuration from .bochsrc
00000000000e[      ] .bochsrc:505: 'keyboard_serial_delay' will be replaced by new
w 'keyboard' option.
00000000000e[      ] .bochsrc:522: 'keyboard_paste_delay' will be replaced by new
'keyboard' option.
00000000000e[      ] .bochsrc:631: 'keyboard_mapping' will be replaced by new 'ke
yboard' option.
-----
Bochs Configuration: Main Menu
-----

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate. Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found. When you are satisfied with the configuration, go
ahead and start the simulation.

You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [6]
```

Exercise 1

Description

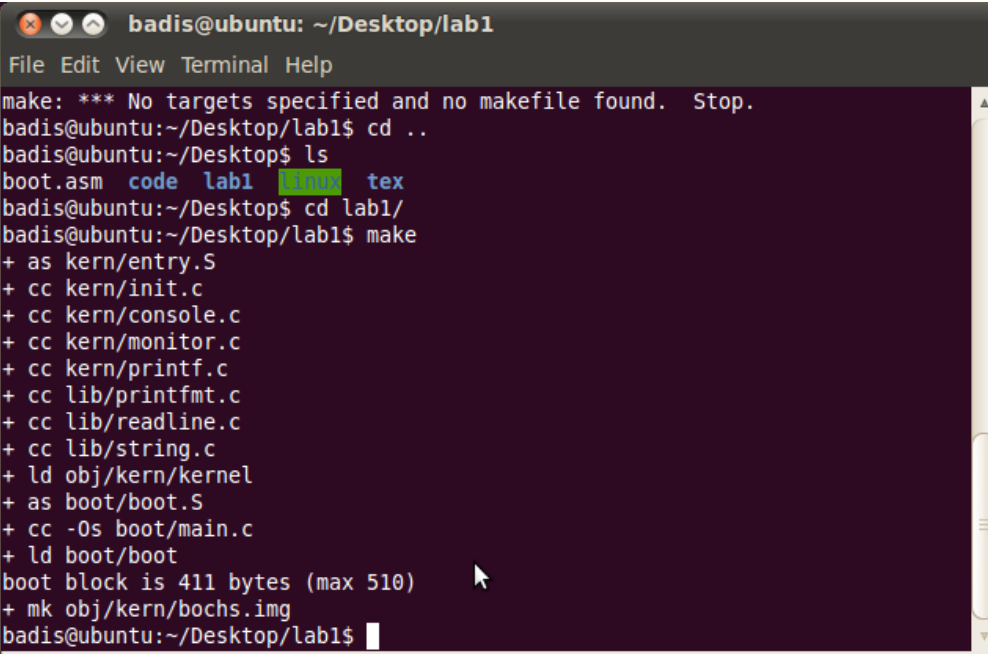
Here we have a emulated 'load' module of a simple operating system, read through the source code then explain in details how is an OS loaded using Bochs.

Solution

Simulate the x86

Extract the Lab 1 files into local directory, then type `$make` or `$gmake` in the `lab1` directory to build boot loader and kernel. The boot loader consists of one assembly language source file, `boot/boot.S`, and one C source file, `boot/main.c`.

Note that the version of default gcc compiler should be no higher than 4.4.x, otherwise might cause error.

A terminal window titled 'badis@ubuntu: ~/Desktop/lab1' with a menu bar (File, Edit, View, Terminal, Help). The terminal shows the following commands and output:

```
make: *** No targets specified and no makefile found. Stop.
badis@ubuntu:~/Desktop/lab1$ cd ..
badis@ubuntu:~/Desktop$ ls
boot.asm  code  lab1  linux  tex
badis@ubuntu:~/Desktop$ cd lab1/
badis@ubuntu:~/Desktop/lab1$ make
+ as kern/entry.S
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 411 bytes (max 510)
+ mk obj/kern/bochs.img
badis@ubuntu:~/Desktop/lab1$
```

Before actually running Bochs, first type `$vi .bochsrc` in `lab1` directory, find `"romimage : file = $BXSHARE/BIOS-bochs-latest,address = 0xf0000"` at line77, then change it into `"romimage : file = $BXSHARE/BIOS-bochs-latest"`, otherwise would cause the error showed below while running Bochs.

```

You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [6]
=====
Event type: PANIC
Device: [MEM0 ]
Message: ROM: System BIOS must end at 0xfffff

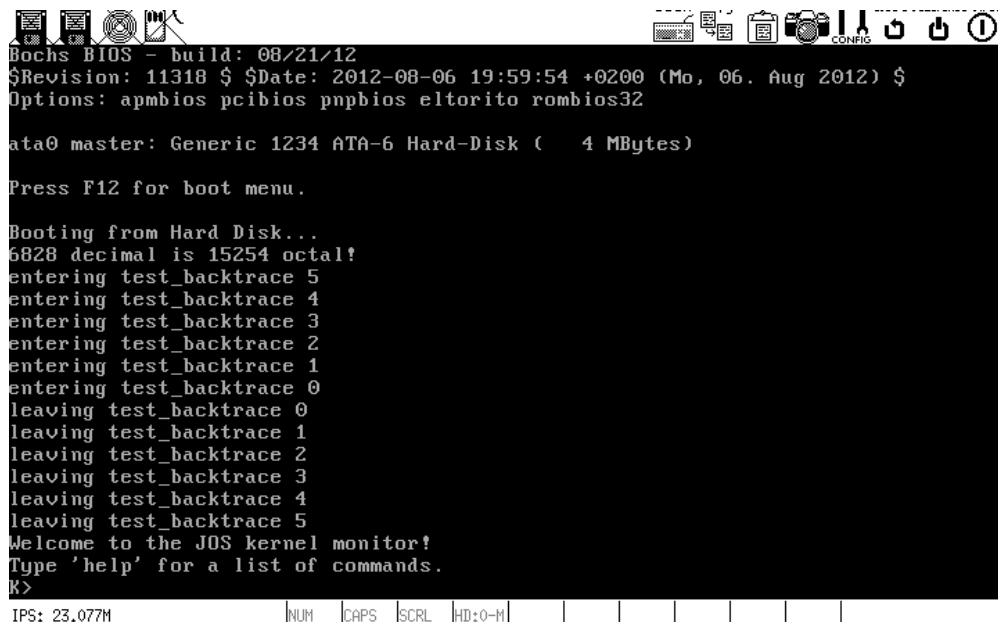
A PANIC has occurred. Do you want to:
  cont      - continue execution
  alwayscont - continue execution, and don't ask again.
              This affects only PANIC events from device [MEM0 ]
  die       - stop execution now
  abort     - dump core
  debug     - continue and return to bochs debugger
Choose one of the actions above: [die] █

```

Now type `$bochs` to run Bochs, supplying the file `obj/kern/kernel.img` as the contents of the PC's virtual hard disk. This hard disk image contains both boot loader(`obj/boot/boot`) and kernel(`obj/kern/kernel`).

Bochs has read the file `.bochsrc` which describes the virtual x86 PC it will emulate for the kernel. Chosen '6' to begin simulation, Bochs has now started the simulated machine, and is ready to execute the first instruction. An window should have popped up to show the "virtual display" of the simulated PC, it's still blank since the simulated PC hasn't booted yet.

Then type in 'c' after `< bochs:1 >` to continue. The Bochs window should look like this:



```

Bochs BIOS - build: 08/21/12
$Revision: 11318 $ $Date: 2012-08-06 19:59:54 +0200 (Mo, 06. Aug 2012) $
Options: apmbios pcibios pnpbios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk ( 4 MBytes)

Press F12 for boot menu.

Booting from Hard Disk...
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

IPS: 23.077M  NUM  CAPS  SCRL  HD:0-M

```

Load the Kernel

boot/main.c is the C language part of boot loader, and it's responsible for loading the executable code(in ELF format) of kernel into memory from the disk image. To make sense out of main.c, we need to know what an ELF binary is.

When we compile and link the kernel, the compiler transforms each '*.c' file into an object file('*.o') containing assembly language instructions encoded in the compact binary format. The linker then combines all of the compiled object files into a single binary image such as obj/kern/kernel, which is a binary in the ELF format.

An ELF binary starts with a fixed-length ELF header, followed by a variable-length program header listing each of the program sections to be loaded into memory at a specified address. The C definitions for these ELF headers are in inc/elf.h. The program sections we're interested in are:

.text: The program's executable instructions.

.rodata: Read-only data.

.data: The program's initialized data.

We can examine the full list of the names, sizes, and link addresses of all the sections in the kernel executable by typing:

```
$objdump -h obj/kern/kernel
```

There is one more field in the ELF header that is important, named e_entry. It holds the link address of the entry point in the program: the memory address in the program's text section at which the program should begin executing. We can see the entry point by typing:

```
$objdump -f obj/kern/kernel
```

```
sol@ubuntu:~/文档/Code/lab1$ objdump -h ./obj/kern/kernel
./obj/kern/kernel:      file format elf32-i386

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 .text          00001715  f0100000  f0100000  00001000  2**4
    CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata         00000738  f0101720  f0101720  00002720  2**5
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab          00003259  f0101e58  f0101e58  00002e58  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr        0000170d  f01050b1  f01050b1  000060b1  2**0
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data           00008320  f0107000  f0107000  00008000  2**12
    CONTENTS, ALLOC, LOAD, DATA
  5 .bss            00000660  f010f320  f010f320  00010320  2**5
    ALLOC
  6 .comment        0000002a  00000000  00000000  00010320  2**0
    CONTENTS, READONLY
sol@ubuntu:~/文档/Code/lab1$
sol@ubuntu:~/文档/Code/lab1$ objdump -f ./obj/kern/kernel
./obj/kern/kernel:      file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0xf010000c
```

We can see that physical address 0xf010000c is where the boot loader actually loaded the kernel.

Exercise 2

Description

Use Bochs to trace into the JOS kernel and find where the new virtual-to-physical mapping takes effect. Then examine the Global Descriptor Table (GDT) that the code uses to achieve this effect, and make sure you understand what's going on.

Solution

```

Please choose one: [6] 6
Next at t=0
(0) [0x00000000fffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b      ; ea5be000f0
<bochs:1> b 0x010000c
<bochs:2> c
(0) Breakpoint 1, 0x0010000c in ?? ()
Next at t=1234916044
(0) [0x00000000010000c] 0008:0010000c (unk. ctxt): mov word ptr ds:0x472, 0x1234 ; 66c705720400003412
<bochs:3> s
Next at t=1234916045
(0) [0x000000000100015] 0008:00100015 (unk. ctxt): lgdt ds:0x10f018    ; 0f011518f01000
<bochs:4> 

```

The GDT that boot loader defined while loading the initial data:

```

        .p2align 2                # force 4 byte alignment
gdt:    SEG_NULL                  # null seg
        SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
        SEG(STA_W, 0x0, 0xffffffff)      # data seg
5
gdtdesc: .word    0x17            # sizeof(gdt) - 1
        .long    gdt            # address gdt

```

In kern.entry.S we can find the GDT of kernel:

```

.globl    _start
_start:
    movw    $0x1234,0x472        # warm boot

5    # Establish our own GDT in place of the boot loaders temporary GDT.
    lgdt    RELOC(mygdtdesc)    # load descriptor table

#define RELOC(x) ((x) - KERNBASE)

10 # setup the GDT
    .p2align 2                # force 4 byte alignment
mygdt:
    SEG_NULL                  # null seg
    SEG(STA_X|STA_R, -KERNBASE, 0xffffffff) # code seg
15    SEG(STA_W, -KERNBASE, 0xffffffff)      # data seg
mygdtdesc:
    .word    0x17            # sizeof(mygdt) - 1
    .long    RELOC(mygdt)    # address mygdt

```

```
f01000dd <test_backtrace>:
#include <kern/console.h>

// Test the stack backtrace function (lab 1 only)
5 void
test_backtrace(int x)
{
f01000dd: 55          push    %ebp
f01000de: 89 e5      mov     %esp,%ebp
10 ...
}
```

BIOS

Start Bochs, we again see the first instruction to be executed:

```
Please choose one: [6] 6
Next at t=0
(0) [0x00000000fffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b      ; ea5be000f0
<bochs:1> □
```

From this output we can see that: The PC starts executing in real mode, with CS = 0xf000 and IP = 0xffff0, the f000:fff0 is the segmented address that translates to 0x000ffff0 in real mode.

The first instruction to be executed is a jmp instruction, which jumps to the real mode segmented address CS = 0xf000 and IP = 0xe05b.

The segmented address 0xf000:fff0 was translated into a physical address according to the formula:

$physical\ address = 16 * segment + offset$

So, when the PC sets CS to 0xf000 and IP to 0xffff0, the physical address referenced is: 0xffff0.

After initializing the PCI bus and all the important devices the BIOS knows about, it searches for a bootable device then reads the boot loader from the bootable disk and transfers control to it.

Stack

We can find the caller function of *mon_backtrace()* in lab1/kern/inin.c, which would be evoked when start Bochs.

Listing 1: test_backtrace()

```
// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
5   cprintf("entering test_backtrace %d\n", x);
   if (x > 0)
       test_backtrace(x-1);
   else
       mon_backtrace(0, 0, 0);
10  cprintf("leaving test_backtrace %d\n", x);
}
```

Start Bochs and set break point at 0x1000dd, then use command 'c' to check how this function is called:

```
<bochs:1> b 0x1000dd
<bochs:2> c
6828 decimal is 15254 octal!
(0) Breakpoint 1, 0x001000dd in ?? ()
Next at t=1269460054
(0) [0x00000000001000dd] 0008:f01000dd (unk. ctxt): push ebp ; 55
<bochs:3> c
entering test_backtrace 5
(0) Breakpoint 1, 0x001000dd in ?? ()
Next at t=1269463217
(0) [0x00000000001000dd] 0008:f01000dd (unk. ctxt): push ebp ; 55
<bochs:4> c
entering test_backtrace 4
(0) Breakpoint 1, 0x001000dd in ?? ()
Next at t=1269466380
(0) [0x00000000001000dd] 0008:f01000dd (unk. ctxt): push ebp ; 55
<bochs:5> c
entering test_backtrace 3
(0) Breakpoint 1, 0x001000dd in ?? ()
Next at t=1269469543
(0) [0x00000000001000dd] 0008:f01000dd (unk. ctxt): push ebp ; 55
<bochs:6> c
entering test_backtrace 2
(0) Breakpoint 1, 0x001000dd in ?? ()
Next at t=1269472706
(0) [0x00000000001000dd] 0008:f01000dd (unk. ctxt): push ebp ; 55
<bochs:7> c
entering test_backtrace 1
(0) Breakpoint 1, 0x001000dd in ?? ()
Next at t=1269475869
(0) [0x00000000001000dd] 0008:f01000dd (unk. ctxt): push ebp ; 55
<bochs:8> c
entering test_backtrace 0
leaving test_backtrace 0
```

Memory Management

The load address of a binary is the memory address at which a binary is actually loaded. And the link address of a binary is the memory address for which the binary is linked.

Operating system kernels often like to be linked and run at very high virtual address, such as 0xf0100000, in order to leave the lower part of the processor's virtual address space for user programs to use.

Since we can't actually load the kernel at physical address 0xf0100000, we will use the processor's memory management hardware to map virtual address 0xf0100000 - the link address at which the kernel code expects to run - to physical address 0x00100000 - where the boot loader actually loaded the kernel. This way, although the kernel's virtual address is high enough to leave plenty of address space for user processes, it will be loaded in physical memory at the 1MB point in the PC's RAM, just above the BIOS ROM.

When the JOS kernel first starts up, we'll initially use segmentation to establish our desired virtual-to-physical mapping, because it is quick and easy.

Exercise 3

Description

One segment of code is omitted which is necessary to print octal numbers using patterns of the form `%o`. Find and fill in this code fragment.

Solution

Listing 2: The initial fragment of code in 'printfmt.c'

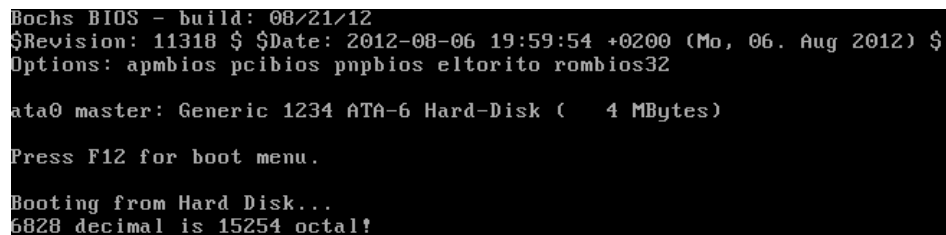
```
// (unsigned) octal
case 'o':
    // Replace this with your code.
    putchar('X', putdat);
5    putchar('X', putdat);
    putchar('X', putdat);
    break;
```

Reference on the implementation of printing decimal numbers and utilize function `getuint()`, we can implement this fragment as follow:

Listing 3: Implementation of printing octal numbers

```
// (unsigned) octal
case 'o':
    num = getuint(&ap, lflag);
    base = 8; // octal flag
5    goto number;
```

After filling in the code fragment, \$make at directory /lab1 in the Terminal, then restart Bochs and we shall see that the printing octal numbers function works just fine:



```
Bochs BIOS - build: 08/21/12
$Revision: 11318 $ $Date: 2012-08-06 19:59:54 +0200 (Mo, 06. Aug 2012) $
Options: apmbios pcibios pnpbios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk ( 4 MBytes)

Press F12 for boot menu.

Booting from Hard Disk...
6828 decimal is 15254 octal!
```

Exercise 4

Description

Implement a stack backtrace function, which you should call `mon.backtrace()`. A prototype for this function is already waiting for you in `kern/monitor.c`. You can do it entirely in C, but you may find the `read_ebp()` function in `inc/x86.h` useful. You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user. The backtrace function should display a listing of function call frames in the following format:

```
Stack backtrace:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
  ...
```

The first line printed reflects the currently executing function, namely *mon_backtrace* itself, the second line reflects the function that called *mon_backtrace*, the third line reflects the function that called that one, and so on. You should print all the outstanding stack frames. By studying *kern/entry.S* you'll find that there is an easy way to tell when to stop.

Within each line, the *ebp* value indicates the base pointer into the stack used by that function: i.e., the position of the stack pointer just after the function was entered and the function prologue code set up the base pointer. The listed *eip* value is the function's return instruction pointer: the instruction address to which control will return when the function returns. The return instruction pointer typically points to the instruction after the call instruction. Finally, the five hex values listed after *args* are the first five arguments to the function in question, which would have been pushed on the stack just before the function was called. If the function was called with fewer than five arguments, of course, then not all five of these values will be useful.

Solution

When typing in a command in Bochs window, it is first read in and processed by void *monitor()*, then this function calls *runcmd()* to match the input command with the command list created in */kern/monitor.c*:

Listing 4: Functions to process the command

```
static int
runcmd(char *buf, struct Trapframe *tf)
{
    ...
5 // Lookup and invoke the command
    if (argc == 0)
        return 0;
    for (i = 0; i < NCOMMANDS; i++) {
        if (strcmp(argv[0], commands[i].name) == 0)
10         return commands[i].func(argc, argv, tf);
    }
    cprintf("Unknown command '%s'\n", argv[0]);
    ...
}

15 void
monitor(struct Trapframe *tf)
{
    char *buf;

20
    cprintf("Welcome to the JOS kernel monitor!\n");
    cprintf("Type 'help' for a list of commands.\n");

    while (1) {
```

```

25     buf = readline("K> ");
        if (buf != NULL)
            // call runcmd() to match the input buf
            if (runcmd(buf, tf) < 0)
                break;
30     }
}

```

Add up command *backtrace* in the command list displayed in Bochs window:

Listing 5: Command list

```

struct Command {
    const char *name;
    const char *desc;
    // return -1 to force monitor to exit
5    int (*func)(int argc, char** argv, struct Trapframe* tf);
};

static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
10    { "kerninfo", "Display information about the kernel", mon_kerninfo },
    { "backtrace", "Display information about the backtrace",
      mon_backtrace } // add command backtrace
};

```

Now implement function `mon_backtrace()` as the format required:

Listing 6: Implementation of `mon_backtrace()`

```

int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
5    uint32_t eip = eip;
    uint32_t ebp = read_ebp();
    cprintf("Stack backtrace:\n");

    uint32_t esp = ebp;
10    int j = 0;
    while (ebp != 0) {
        cprintf("ebp %08x eip %08x ", ebp, eip);
        ebp = *(uint32_t *) (esp);
        esp += 4; // read the next address
15        eip = *(uint32_t *) (esp);
        esp += 4;

        cprintf("args "); // display 5 arguments
        for (j = 0; j < 5; j++) {
20            cprintf("%08x ", *(uint32_t *) (esp));
            esp += 4;
        }
    }
}

```

```

    }
    cprintf("\n");
    esp = ebp;
25  }
    return 0;
}

// defined in /lab1/inc/x86.h
30 static __inline uint32_t
read_ebp(void)
{
    uint32_t ebp;
    __asm __volatile("movl %%ebp,%0" : "=r" (ebp));
35  return ebp;
}

```

Type \$make in lab1 directory then restart Bochs, type in command *backtrace* in Bochs window and we shall see that the information of stack backtrace is displayed as required:

The screenshot shows the Bochs x86 emulator window. The title bar reads "Bochs x86 emulator, http://bochs.sourceforge.net/". The window contains a terminal-like interface with the following text:

```

6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:
ebp f010ef18 eip f01009e7 args 00000000 00000000 00000000 00000000 f01009e9
ebp f010ef38 eip f0100124 args 00000000 00000001 f010ef78 00000000 f01009e9
ebp f010ef58 eip f0100106 args 00000001 00000002 f010ef98 00000000 f01009e9
ebp f010ef78 eip f0100106 args 00000002 00000003 f010efb8 00000000 f01009e9
ebp f010ef98 eip f0100106 args 00000003 00000004 00000000 00000000 00010094
ebp f010efb8 eip f0100106 args 00000004 00000005 00000000 00000000 00000000
ebp f010efd8 eip f0100106 args 00000005 00001aac 00000660 00000000 00000000
ebp f010eff8 eip f0100187 args 00000000 00000000 0000ffff 10cf9b00 0000ffff
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
IPS: 24,080M NUM CAPS SCRL HD:0-M
Type 'help' for a list of commands.
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
backtrace - Display information about the backtrace
K> backtrace
Stack backtrace:
ebp f010ef68 eip f010085f args 00000001 f010ef80 00000000 f010efc8 f010f580
ebp f010efd8 eip f0100967 args 00000000 00001aac 00000660 00000000 00000000
ebp f010eff8 eip f0100193 args 00000000 00000000 0000ffff 10cf9b00 0000ffff
K>

```

make grade

After all exercises are finished, we can run `$make grade` to test our solutions with the grading program in `'grade.sh'`. However, a minor error occurred, we can find more detail in `/lab1/bochs.log`.

```
sol@ubuntu:/media/D2E6AB8FE6AB7281/TDDOWNLOAD/第1次上机作业/实验工程/第一次作业环境/lab1$ make grade
make all
make[1]: 正在进入目录 `/media/D2E6AB8FE6AB7281/TDDOWNLOAD/第1次上机作业/实验工程/第一次作业环境/lab1'
make[1]:正在离开目录 `/media/D2E6AB8FE6AB7281/TDDOWNLOAD/第1次上机作业/实验工程/第一次作业环境/lab1'
make[1]: 正在进入目录 `/media/D2E6AB8FE6AB7281/TDDOWNLOAD/第1次上机作业/实验工程/第一次作业环境/lab1'
+ as kern/entry.S
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 411 bytes (max 510)
+ mk obj/kern/bochs.img
make[1]:正在离开目录 `/media/D2E6AB8FE6AB7281/TDDOWNLOAD/第1次上机作业/实验工程/第一次作业环境/lab1'
sh ./grade.sh
gmake[1]: 正在进入目录 `/media/D2E6AB8FE6AB7281/TDDOWNLOAD/第1次上机作业/实验工程/第一次作业环境/lab1'
gmake[1]:正在离开目录 `/media/D2E6AB8FE6AB7281/TDDOWNLOAD/第1次上机作业/实验工程/第一次作业环境/lab1'
gmake[1]: 正在进入目录 `/media/D2E6AB8FE6AB7281/TDDOWNLOAD/第1次上机作业/实验工程/第一次作业环境/lab1'
gmake[1]: 没有什么可以做的为 `all'.
gmake[1]:正在离开目录 `/media/D2E6AB8FE6AB7281/TDDOWNLOAD/第1次上机作业/实验工程/第一次作业环境/lab1'
Printf: grep: bochs.out: 没有那个文件或目录
WRONG (.5s)
Backtrace: grep: bochs.out: 没有那个文件或目录
Count WRONGgrep: bochs.out: 没有那个文件或目录
, Args WRONG () (.5s)
Score: 0/50
make: *** [grade] 错误 1
```

To correct this error, find `"out = /dev/stdout"` at line8 in `grade.sh`, and replace it with `"out = bochs.out"`, then create a file named `"bochs.out"` under `lab1` directory. Now `$make grade` again, we can see that our solutions are correct and have gained full score.

```
sh ./grade.sh
make[1]: 正在进入目录 `/home/sol/文档/Code/lab1'
make[1]:正在离开目录 `/home/sol/文档/Code/lab1'
make[1]: 正在进入目录 `/home/sol/文档/Code/lab1'
make[1]: 没有什么可以做的为 `all'.
make[1]:正在离开目录 `/home/sol/文档/Code/lab1'
Printf: OK (.5s)
Backtrace: Count OK, Args OK (.5s)
Score: 50/50
```

Acknowledgement

Group work

Yuan Zhaozheng(1110556), Zhang Chao(1110557) and Wu Zhengwei(1110548) helped work out most part of Exercise 1, 2;

Ma Xuan(1110580) finished Exercise 3;

Wang Shaoyao(1110539) finished Exercise 4;

Wu Zhengwei(1110548) summarized results and drafted this documentation, other group members provided suggestions for refinement.