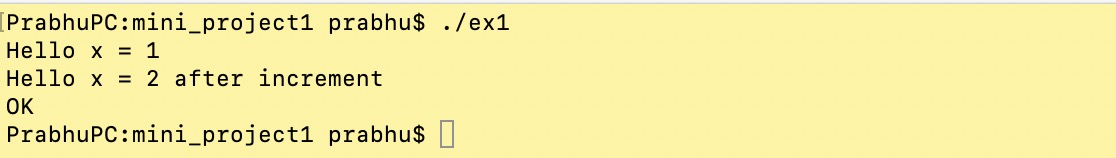
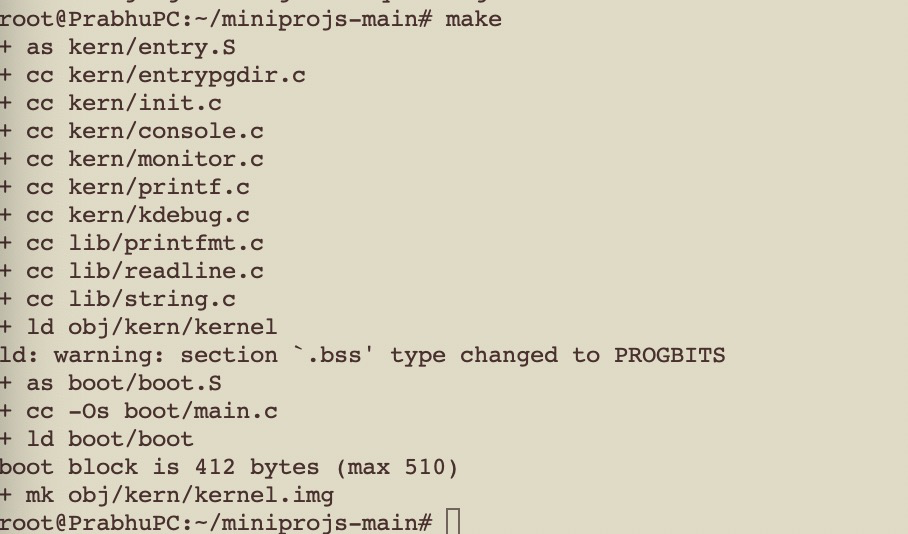
Part1.

Ex1.

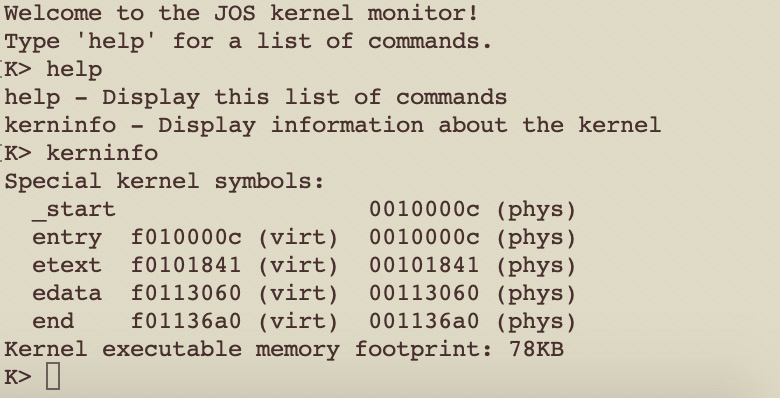


Ex2.

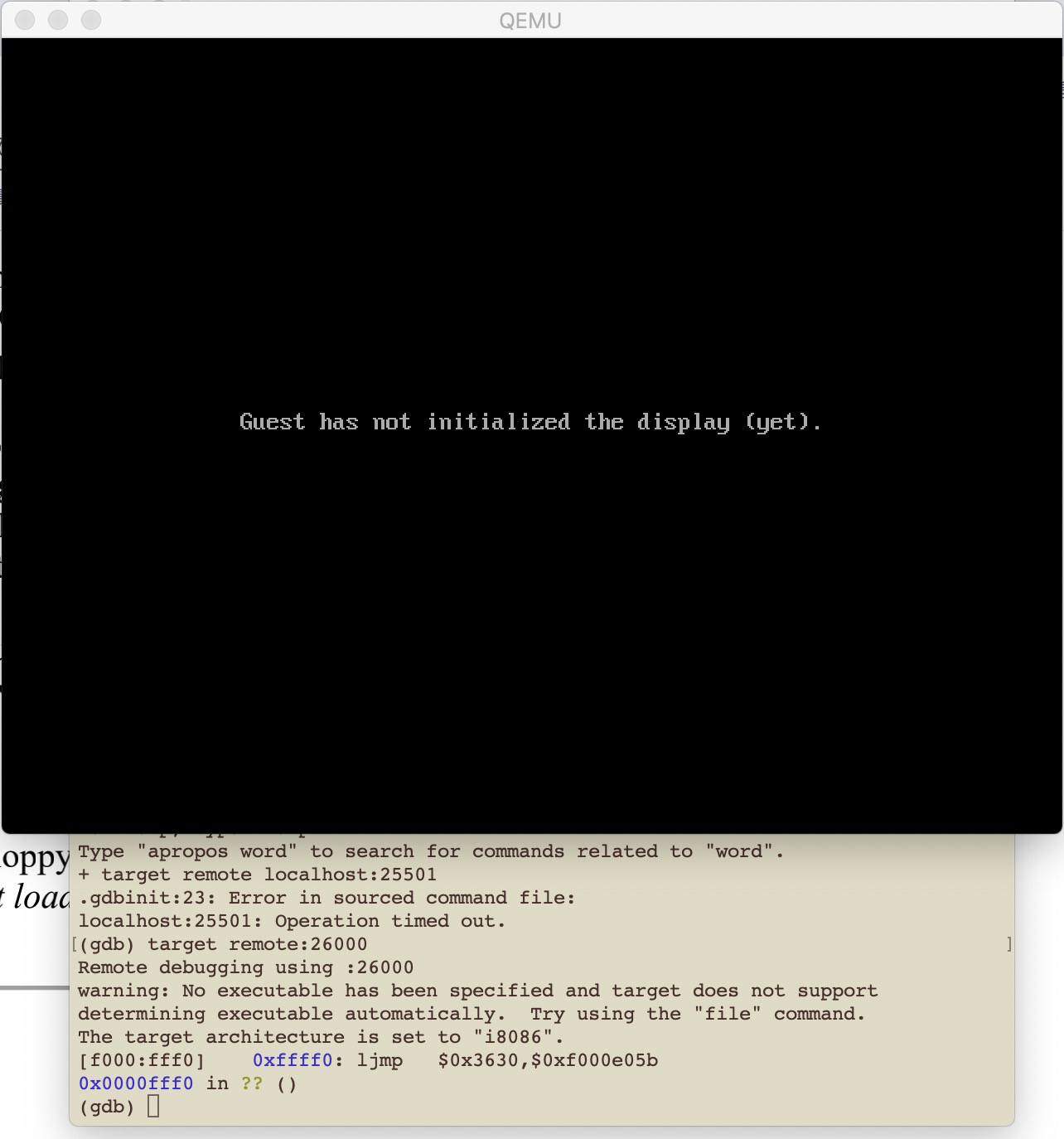


Start qemu with image

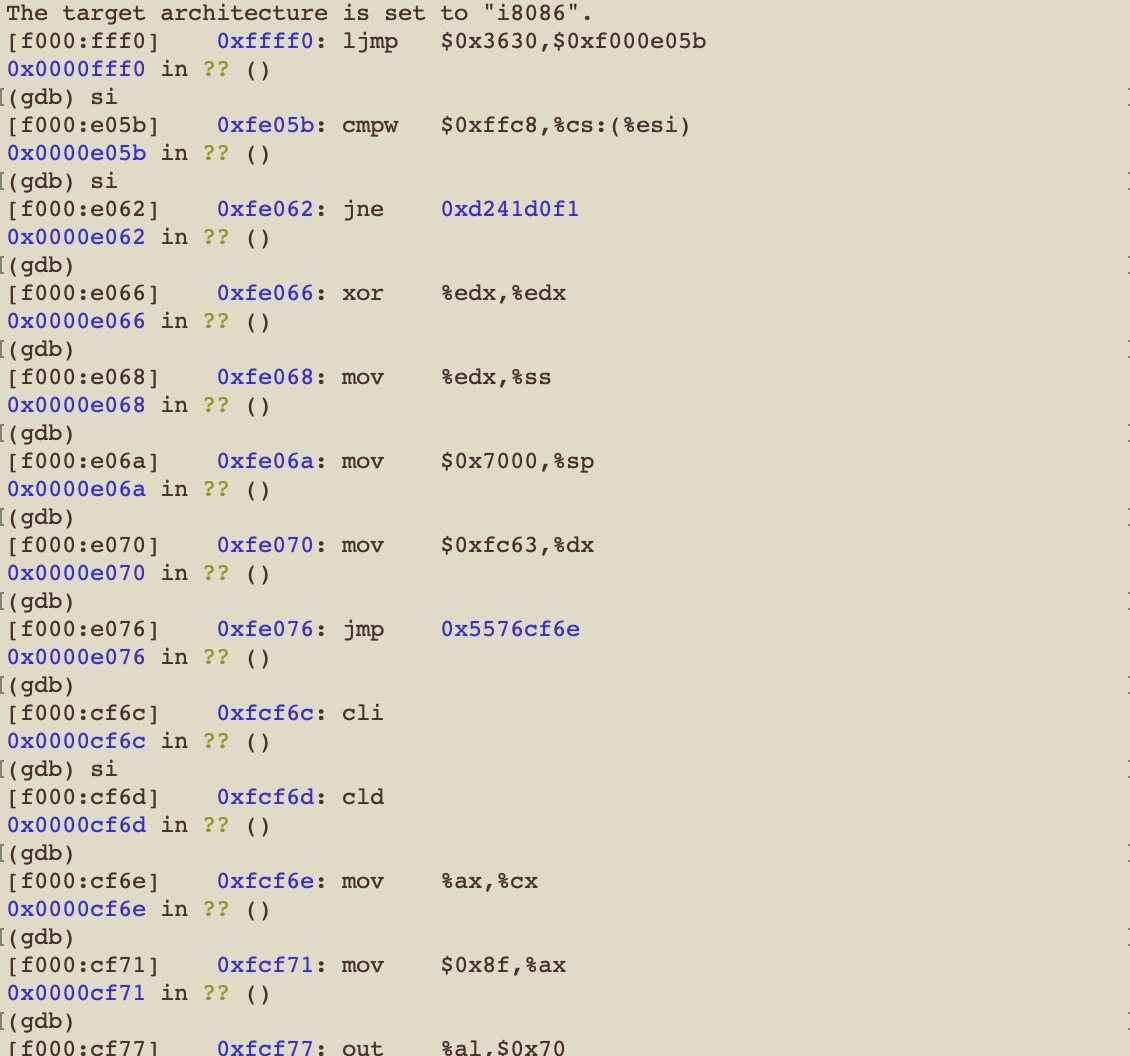


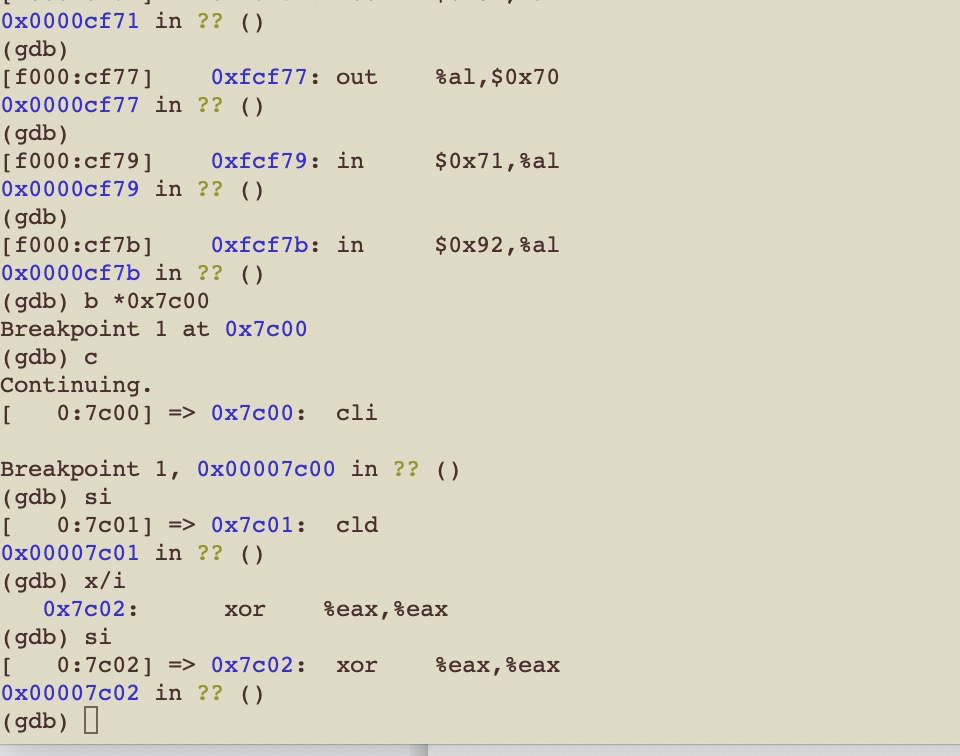


Start with gdb



Ex2, use gdb to debug kernel





Question:

1.At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

**Answer: In boot.asm: line 53,**

**# Jump to next instruction, but in 32-bit code segment.**

**# Switches processor into 32-bit mode.**

**ljmp $PROT\_MODE\_CSEG, $protcseg**

1. What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

Answer:

**The last instruction is as below in boot/main.c:58:**

**// call the entry point from the ELF header**

**// note: does not return! ((void (\*)(void)) (ELFHDR->e\_entry))();**

**The first instruction of the kernel is as below:**

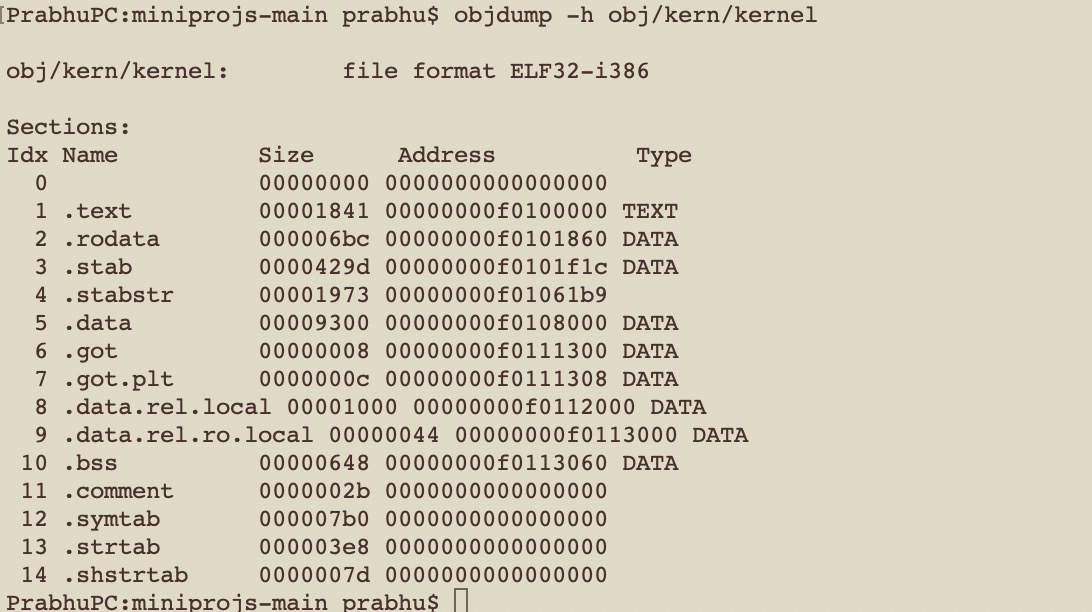
**movw $0x1234,0x472**

1. How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

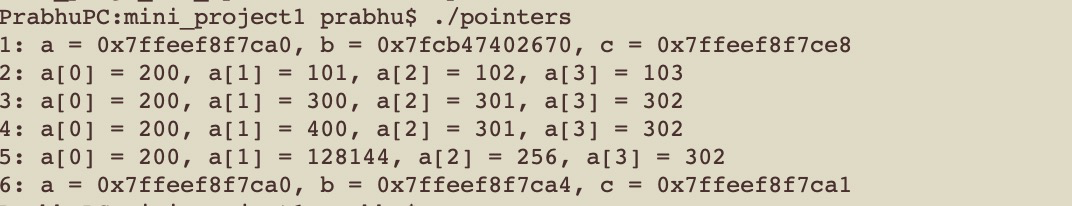
**Answer:**

**The kernel is loaded into memory by program segment, and the ELF header of each segment has the formation, the information is stored in Program Header Table.**

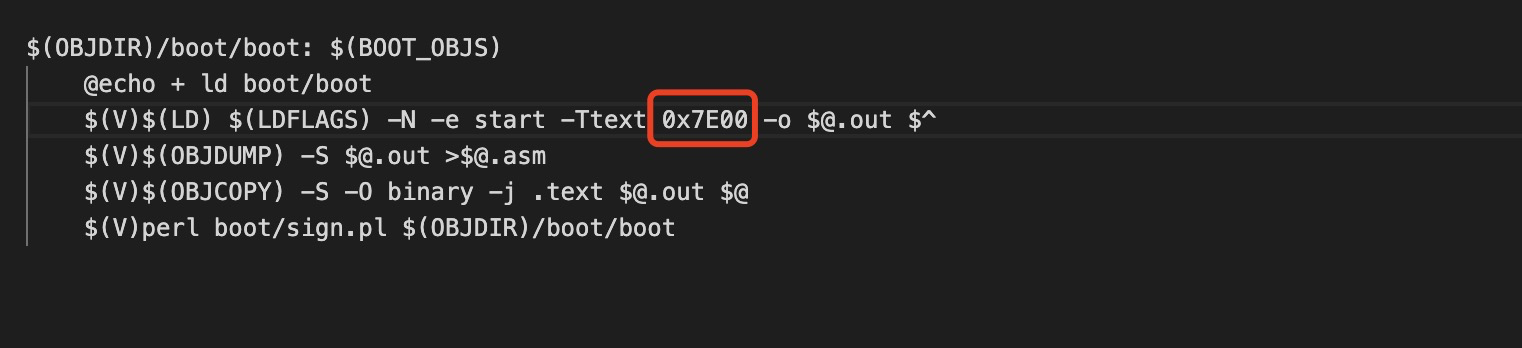
Ex4. ELF header



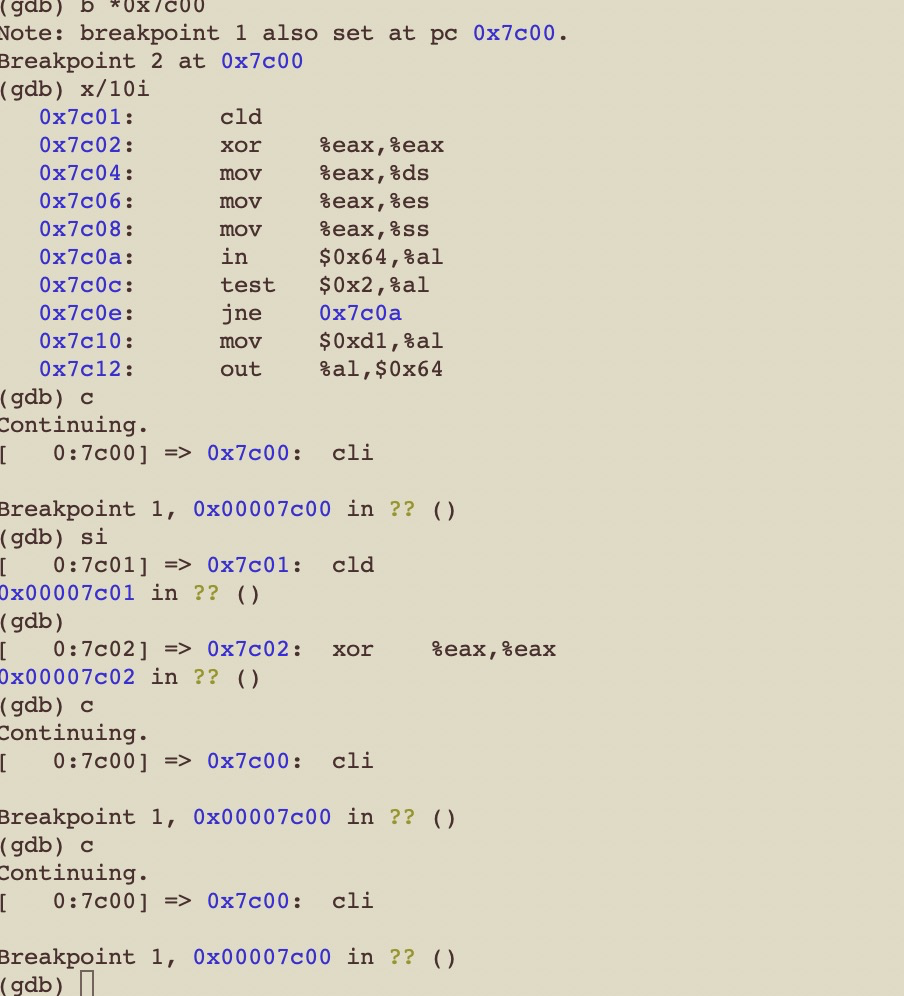
Run the application pointers



Ex5. change the file boot/Makefrag “0x7C00” -> “0x7E00”



It causes the kernel always reboot, and hit the breakpoint 0x7c00, again and again.

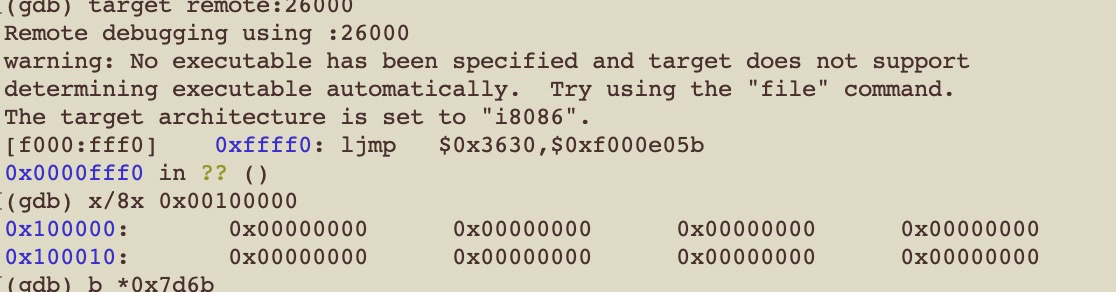


After change the link information, the elf header is as below:

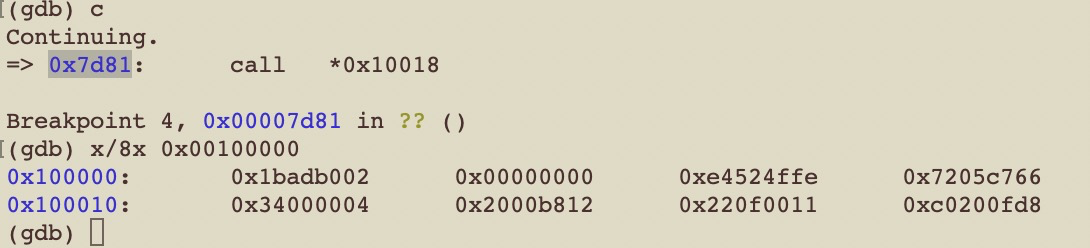


Ex6.

At the beginning of the boot loader, check the memory of 0x00100000



When the kernel is loaded, check the memory again



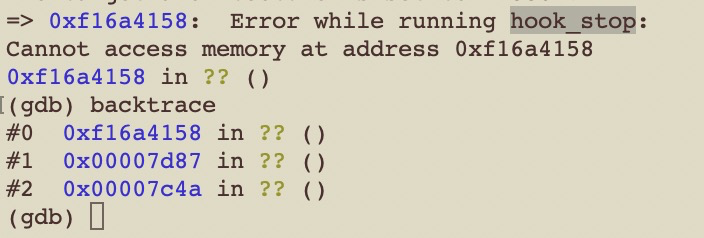
Why are they different? What is there at the second breakpoint?

Answer:

1. They are different due to boot/main load some section data of the kernel into the address 0x00100000.
2. The breakpoint should at call entry of the kernel (0x7d81)

Ex7.

After comment out the code “movl %eax, %cr0” in kern/entry.S, the paging is disable, the kernel cannot access the address larger than the memory size.



What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place?

Yes. The mapping enable the memory paging, let the kernel can access the address larger than the memory size.

Ex 8

1. Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

**Answer: all the functions without static modifier are exported in console.c. Function cputchar in console.c is called by putch in printf.c.**

1. Explain the following from console.c:

// crt\_pos: the position of the input char

// CRT\_SIZE: the max size of the display

if (crt\_pos >= CRT\_SIZE) {

int i;

// move all the display content to the upper one line, for example line 1-> line 2; line2->line 3

memmove(crt\_buf, crt\_buf + CRT\_COLS, (CRT\_SIZE - CRT\_COLS) \* sizeof(uint16\_t));

for (i = CRT\_SIZE - CRT\_COLS; i < CRT\_SIZE; i++)

crt\_buf[i] = 0x0700 | ' ';

crt\_pos -= CRT\_COLS;

}

1. Trace the execution of the following code step-by-step:

int x = 1, y = 3, z = 4;

cprintf("x %d, y %x, z %d\n", x, y, z);

1. In the call to cprintf(), to what does fmt point? To what does ap point?

**Answer:**

**fmt points to “x %d, y %x, z %d\n”**

**ap points to “x, y, z”**

1. List (in order of execution) each call to cons\_ putc, va\_ arg, and vcprintf. For cons. putc, list its argument as well. For va arg, list what ap points to before and after the call., For vcprintf list the values of its two arguments.

**Answer:**

4.

**Answer: i=0x726c6400; no need to change.**

1. **Answer: the output after “y=” is a random number, due to no y value is provided.**
2. **Answer: The format argument should be the last argument for the function.**

**Ex9.**

How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

**Answer:**

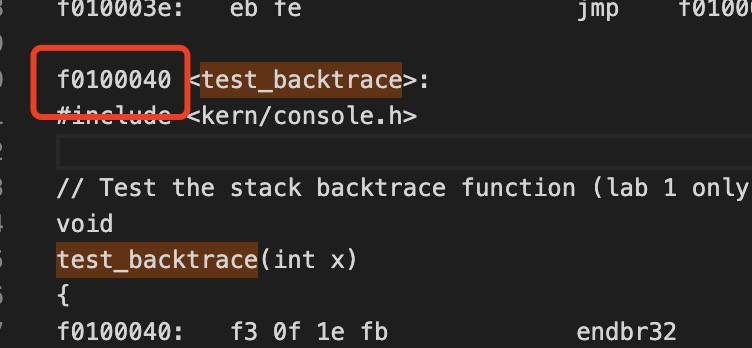
**The pointer is point to bootstacktop.**

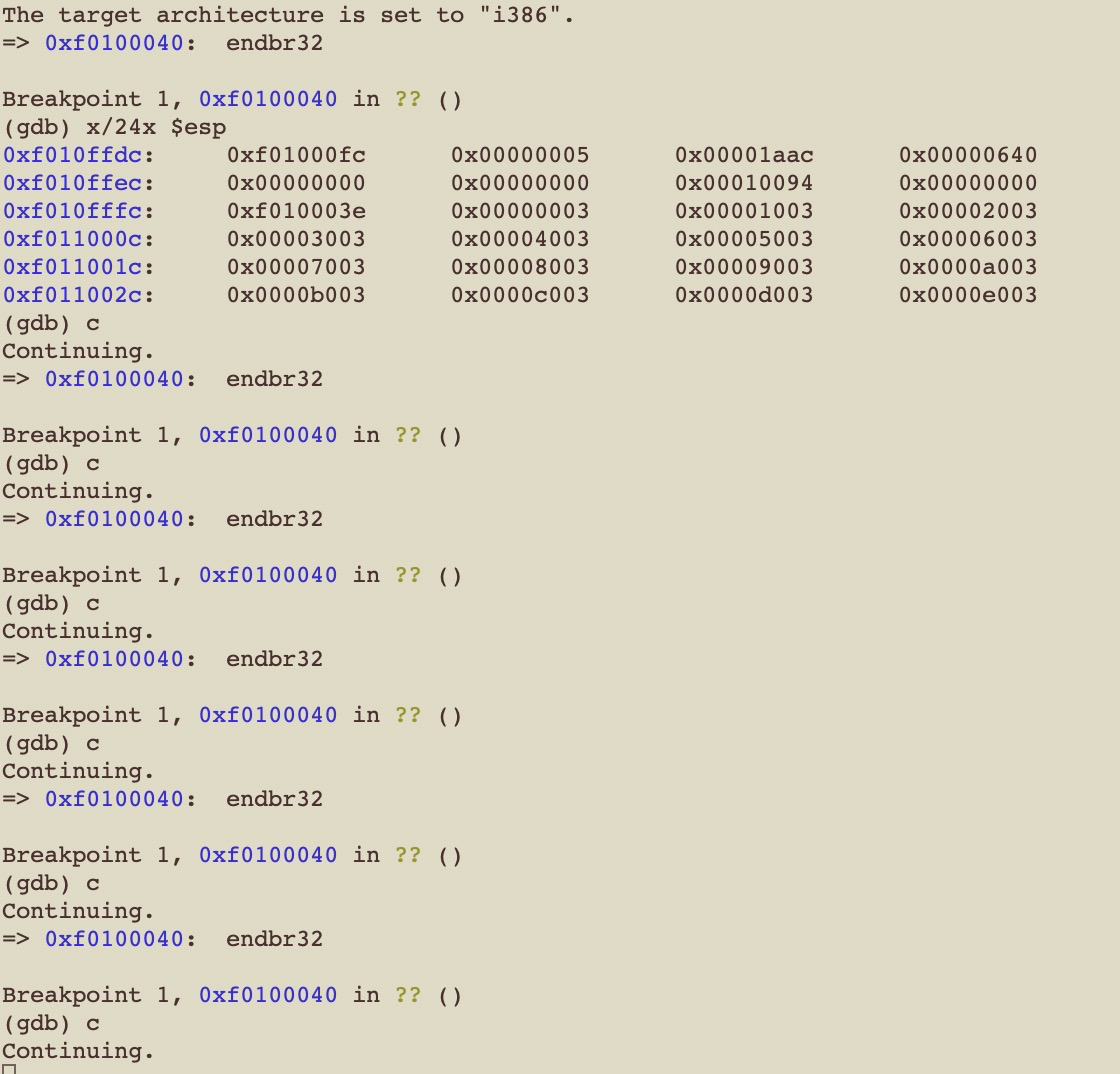
**Through the code as below, the kernel reserve space for its stack**

**movl $(bootstacktop),%esp**

**Ex 10.**

How many 32-bit words does each recursive nesting level of test\_backtrace push on the stack, and what are those words?





**Answer:**

**16 byte is allocated for each stack.**

**Ex11 Code:**

**int**

**mon\_backtrace(int argc, char \*\*argv, struct Trapframe \*tf)**

**{**

**// Your code here.**

**uint32\_t arg[4];**

**uint32\_t\* ebp = (uint32\_t\*)read\_ebp();**

**while(ebp != NULL)**

**{**

**uint32\_t eip = \*(ebp + 1);**

**for(int i = 0; i < 4; i++)**

**arg[i] = \*(ebp + i + 2);**

**cprintf("ebp %08x eip %08x args ", ebp, eip);**

**for(int i = 0; i < 4; i++)**

**cprintf("%08x ", arg[i]);**

**cprintf("\n");**

**ebp = (uint32\_t\*) (\*ebp);**

**}**

**return 0;**

**}**

**Ex 12**

**Add code:**

**kern/kdebug.c**

**stab\_binsearch(stabs, &lline, &rline, N\_SLINE, addr);**

**if(lline <= rline)**

**info -> eip\_line = stabs[lline].n\_desc;**

**else**

**return -1;**

**kern/monitor.c:**

**static struct Command commands[] = {**

**{ "help", "Display this list of commands", mon\_help },**

**{ "kerninfo", "Display information about the kernel", mon\_kerninfo },**

**{ "backtrace", "Display information about the callstack", mon\_backtrace },**

**};**

**mon\_backtrace(int argc, char \*\*argv, struct Trapframe \*tf)**

**{**

**// Your code here.**

**// The data structure to save information about EIP**

**struct Eipdebuginfo info;**

**uint32\_t arg[4];**

**uint32\_t\* ebp = (uint32\_t\*)read\_ebp();**

**while(ebp != NULL)**

**{**

**uint32\_t eip = \*(ebp + 1);**

**for(int i = 0; i < 4; i++)**

**arg[i] = \*(ebp + i + 2);**

**cprintf("ebp %08x eip %08x args ", ebp, eip);**

**for(int i = 0; i < 4; i++)**

**cprintf("%08x ", arg[i]);**

**cprintf("\n");**

**if(debuginfo\_eip(eip, &info) == 0)**

**{**

**cprintf("%s:%d: ", info.eip\_file, info.eip\_line);**

**cprintf("%.\*s", info.eip\_fn\_namelen, info.eip\_fn\_name);**

**cprintf("+%d\n", eip - (uint32\_t)info.eip\_fn\_addr);**

**}**

**else**

**cprintf("Error happened when reading symbol table\n");**

**ebp = (uint32\_t\*) (\*ebp);**

**}**

**return 0;**

**}**

