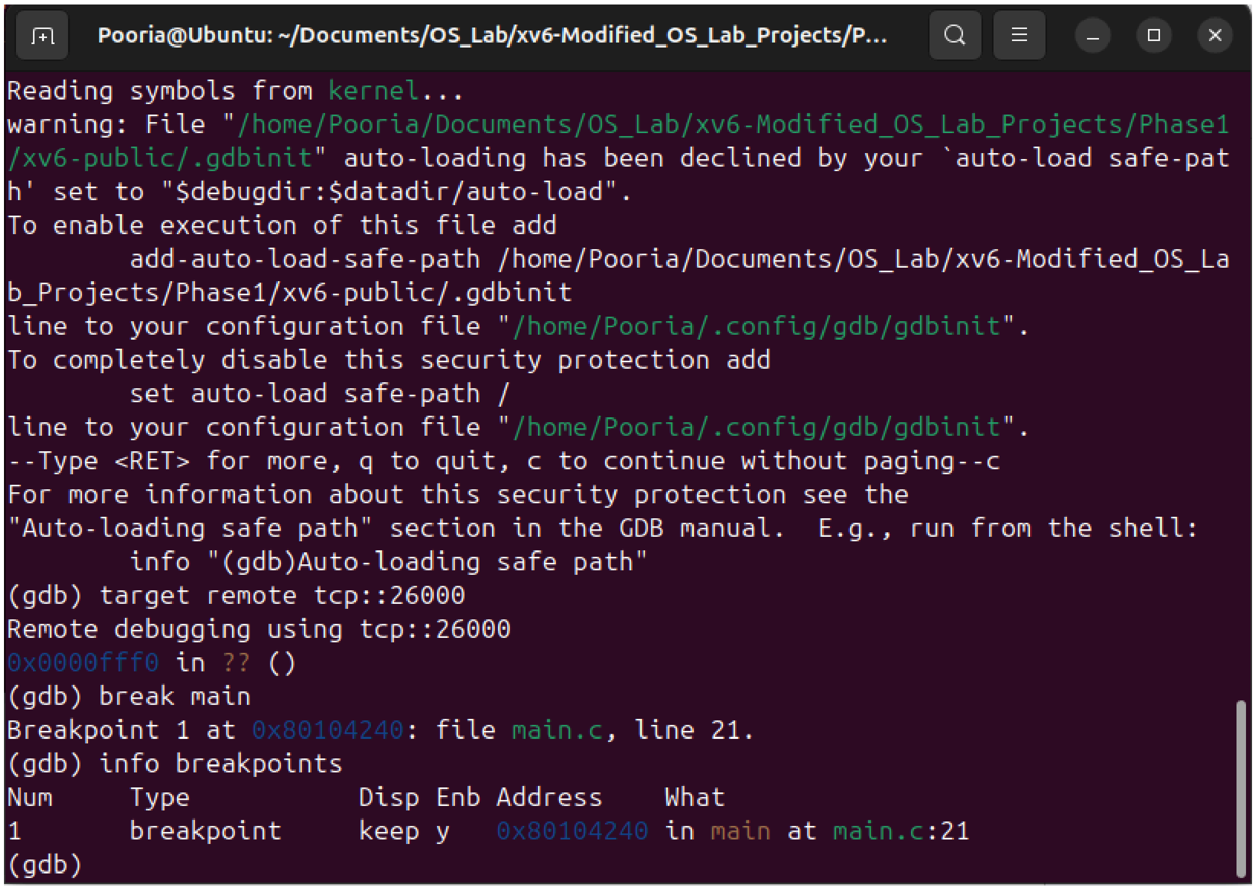
OS Project 1 Debugging Report

Group Members:Pooria Mahdian
Mohammad Taha Majlesi
Alireza Karimi

Answer of the Questions:

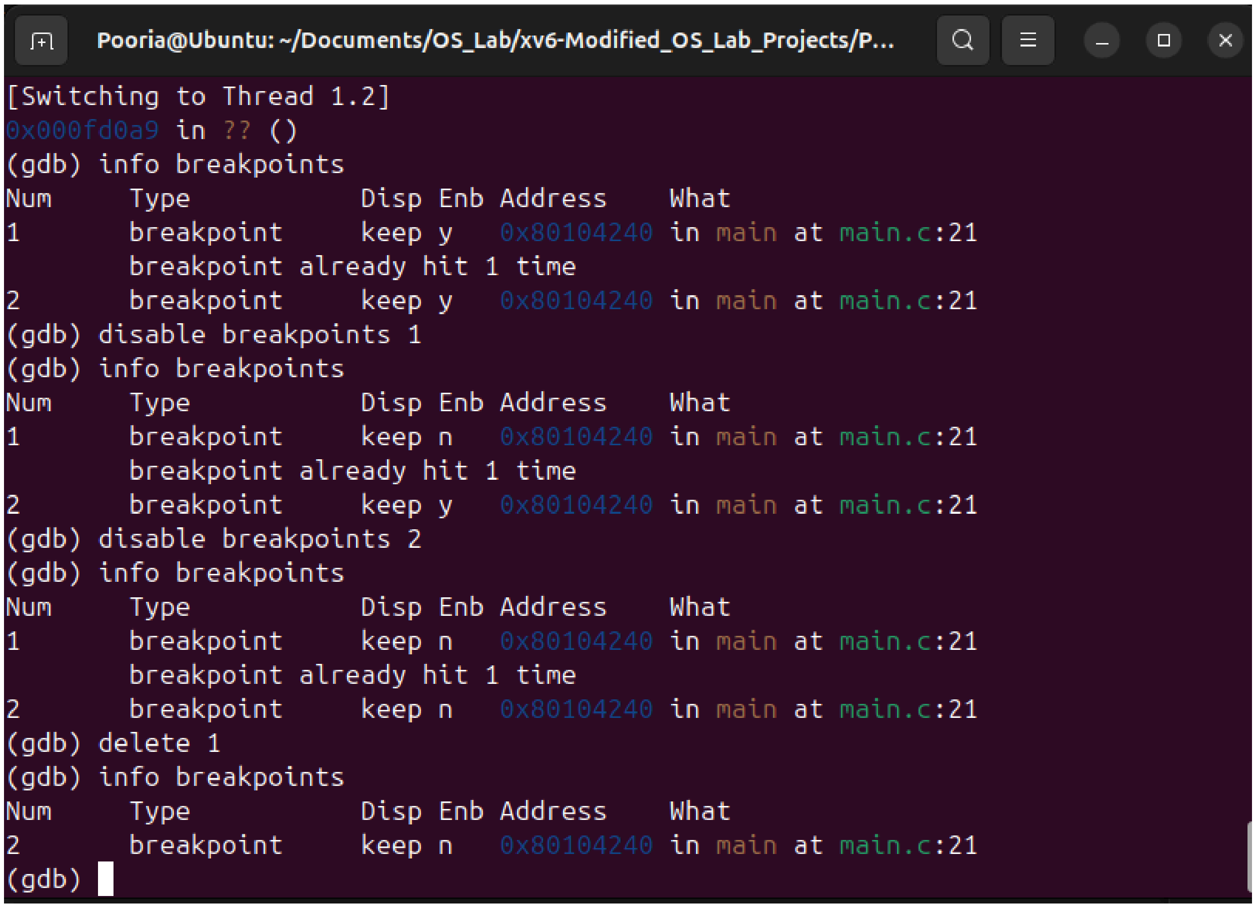
1)There are several ways to see the breakpoints.By using the “info breakpoints” command the breakpoints’ information is printed inside the debugging console.Also using the “save breakpoints” command will save the definition of the breakpoints as a script.



```
Reading symbols from kernel...
warning: File "/home/Pooria/Documents/OS_Lab/xv6-Modified_OS_Lab_Projects/Phase1
/xv6-public/.gdbinit" auto-loading has been declined by your `auto-load safe-pat
h' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/Pooria/Documents/OS_Lab/xv6-Modified_OS_La
b_Projects/Phase1/xv6-public/.gdbinit
line to your configuration file "/home/Pooria/.config/gdb/gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/Pooria/.config/gdb/gdbinit".
--Type <RET> for more, q to quit, c to continue without paging--c
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) break main
Breakpoint 1 at 0x80104240: file main.c, line 21.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x80104240 in main at main.c:21
(gdb)
```

Img 1.”info breakpoints”

2)The “disable breakpoints” command is used to disable some or all of the breakpoints.To do so it receives one or several break point numbers.The “delete” command completely removes the selected breakpoints.This command also receives one or several numbers.

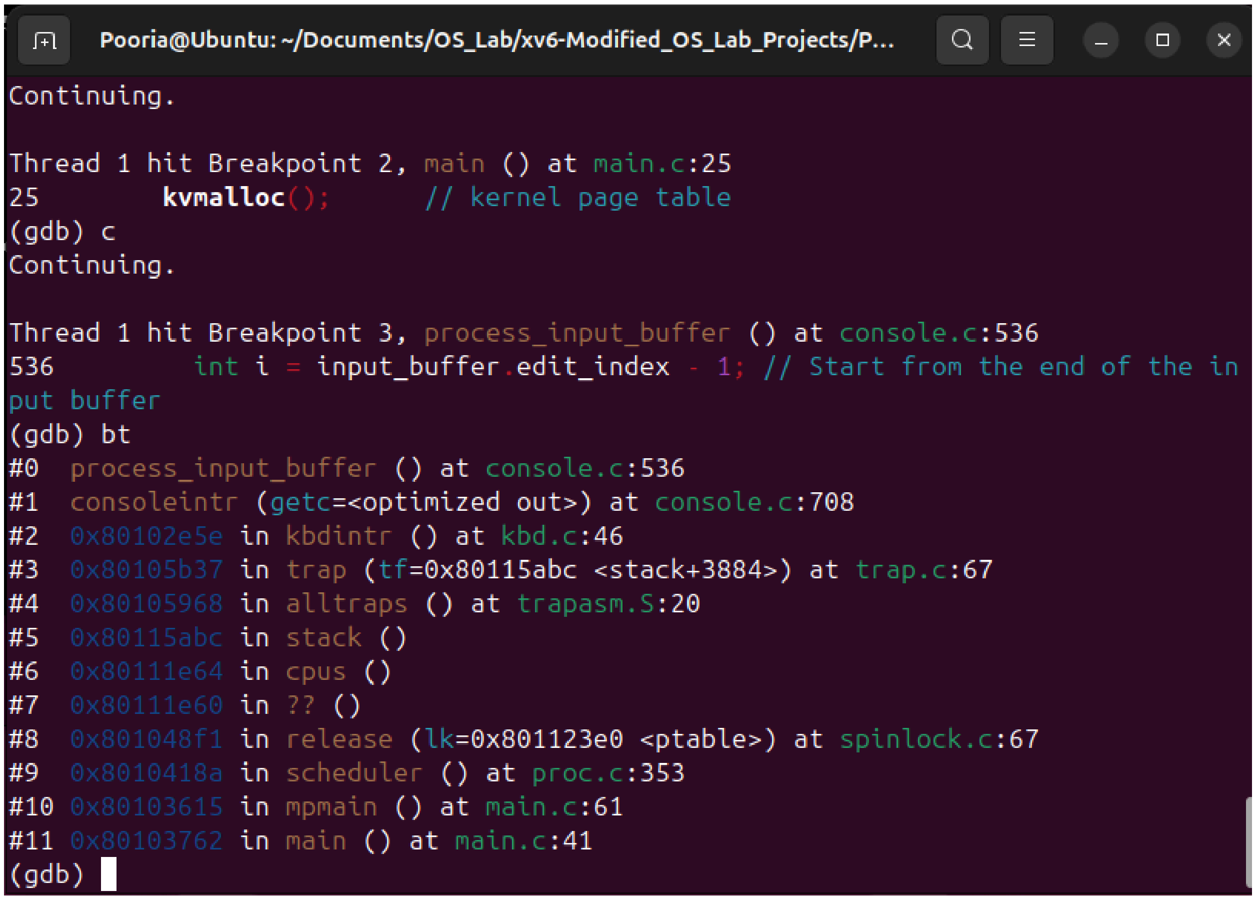


Pooria@Ubuntu: ~/Documents/OS_Lab/xv6-Modified_OS_Lab_Projects/P...

```
[Switching to Thread 1.2]
0x000fd0a9 in ?? ()
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x80104240 in main at main.c:21
        breakpoint already hit 1 time
2       breakpoint     keep y   0x80104240 in main at main.c:21
(gdb) disable breakpoints 1
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep n   0x80104240 in main at main.c:21
        breakpoint already hit 1 time
2       breakpoint     keep y   0x80104240 in main at main.c:21
(gdb) disable breakpoints 2
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep n   0x80104240 in main at main.c:21
        breakpoint already hit 1 time
2       breakpoint     keep n   0x80104240 in main at main.c:21
(gdb) delete 1
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
2       breakpoint     keep n   0x80104240 in main at main.c:21
(gdb) 
```

Img 2.”disable breakpoints” & “delete”

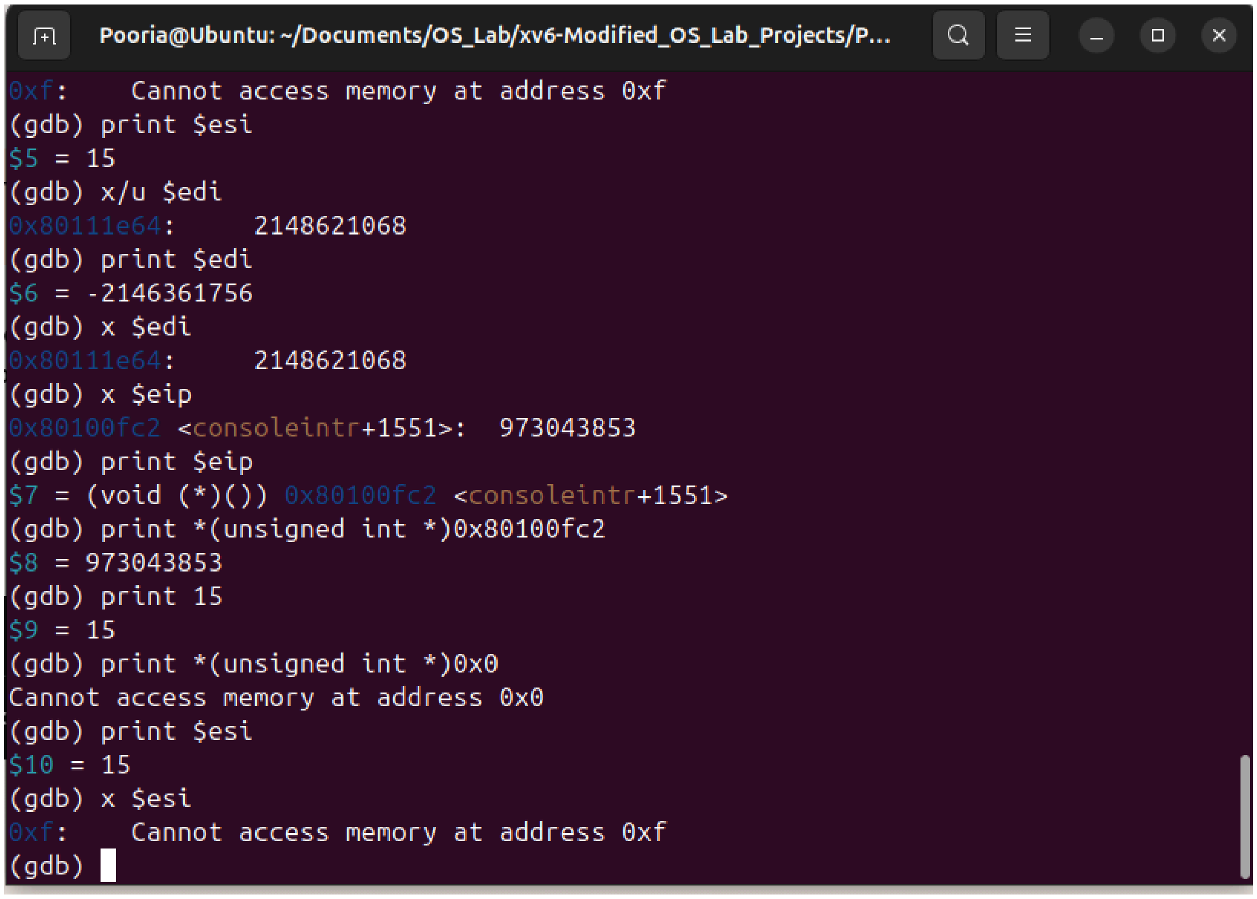
3) bt(Backtrace) provides a list of function calls that led to the current point in the program.For example it prints the sequence of function calls that have been made to reach the current function.



```
Continuing.

Thread 1 hit Breakpoint 2, main () at main.c:25
25          kvmalloc();      // kernel page table
(gdb) c
Continuing.

Thread 1 hit Breakpoint 3, process_input_buffer () at console.c:536
536          int i = input_buffer.edit_index - 1; // Start from the end of the in
put buffer
(gdb) bt
#0  process_input_buffer () at console.c:536
#1  consoleintr (getc=<optimized out>) at console.c:708
#2  0x80102e5e in kbdintr () at kbd.c:46
#3  0x80105b37 in trap (tf=0x80115abc <stack+3884>) at trap.c:67
#4  0x80105968 in alltraps () at trapasm.S:20
#5  0x80115abc in stack ()
#6  0x80111e64 in cpus ()
#7  0x80111e60 in ?? ()
#8  0x801048f1 in release (lk=0x801123e0 <ptable>) at spinlock.c:67
#9  0x8010418a in scheduler () at proc.c:353
#10 0x80103615 in mpmain () at main.c:61
#11 0x80103762 in main () at main.c:41
(gdb) 
```

Img 3."bt" here shows the functions that ran before hitting the breakpoint which was set for console.c:line535(Definition of the function that handles NON=? Inputs and is called process_input_buffer)

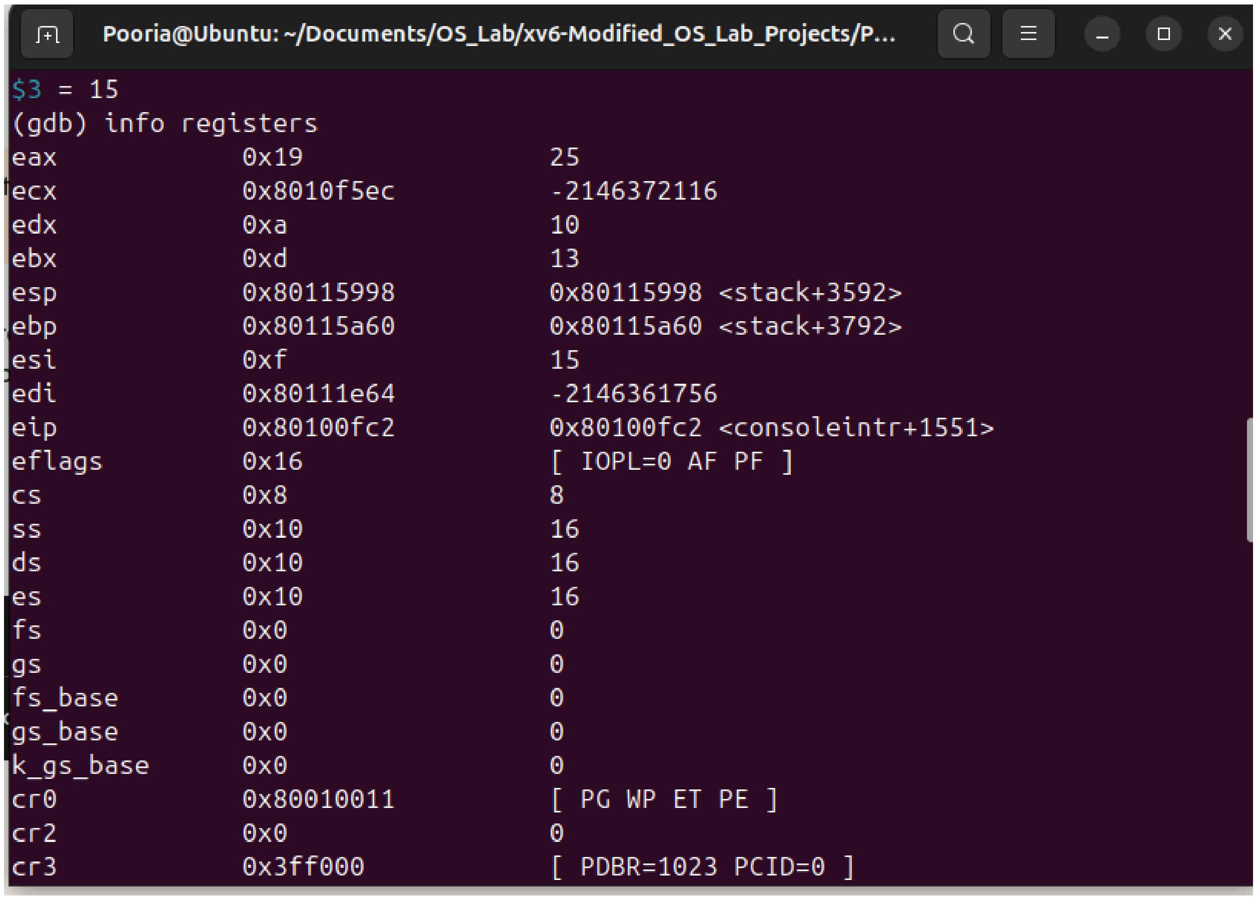
4)Purpose & Usage:"x" command tries to examine memory at the address contained in the given register but "print" simply prints out the value stored in the given register.
Flexibility in output: "x" is used for debugging lower-level memory issues because it allows viewing memory data in a variety of formats but "print" is for general expression evaluation. It also offers many options like limiting output characters,printing array indexes, etc.



```
Pooria@Ubuntu: ~/Documents/OS_Lab/xv6-Modified_OS_Lab_Projects/P...
0xf:    Cannot access memory at address 0xf
(gdb) print $esi
$5 = 15
(gdb) x/u $edi
0x80111e64:     2148621068
(gdb) print $edi
$6 = -2146361756
(gdb) x $edi
0x80111e64:     2148621068
(gdb) x $eip
0x80100fc2 <consoleintr+1551>:  973043853
(gdb) print $eip
$7 = (void (*)()) 0x80100fc2 <consoleintr+1551>
(gdb) print *(unsigned int *)0x80100fc2
$8 = 973043853
(gdb) print 15
$9 = 15
(gdb) print *(unsigned int *)0x0
Cannot access memory at address 0x0
(gdb) print $esi
$10 = 15
(gdb) x $esi
0xf:    Cannot access memory at address 0xf
(gdb) 
```

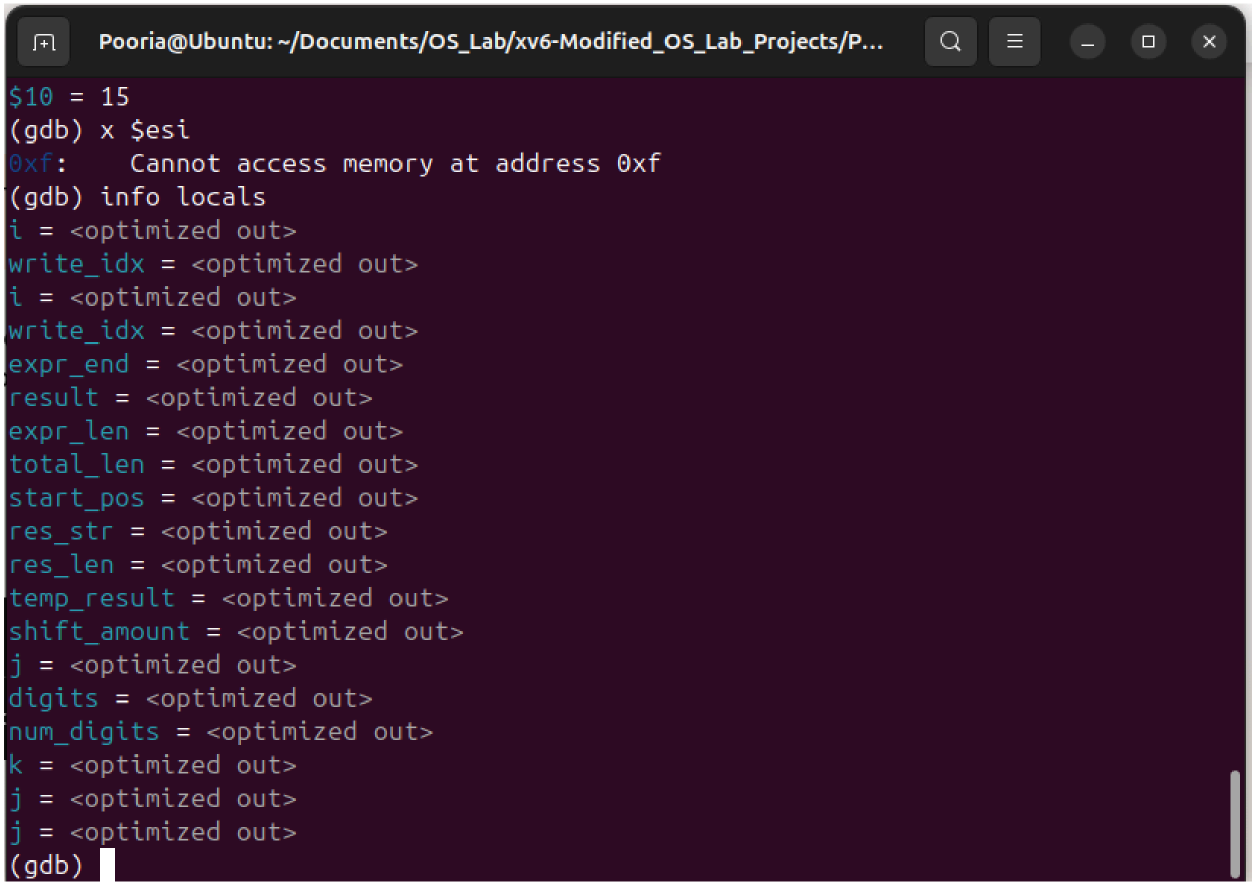
Img 4.Comparison between “x” and “print”

5)”info registers” and “info locals” are the commands for viewing the status of our registers and local variables.


Pooria@Ubuntu: ~/Documents/OS_Lab/xv6-Modified_OS_Lab_Projects/P...

```
$3 = 15
(gdb) info registers
eax            0x19                25
ecx            0x8010f5ec          -2146372116
edx            0xa                 10
ebx            0xd                 13
esp            0x80115998          0x80115998 <stack+3592>
ebp            0x80115a60          0x80115a60 <stack+3792>
esi            0xf                 15
edi            0x80111e64          -2146361756
eip            0x80100fc2          0x80100fc2 <consoleintr+1551>
eflags         0x16                [ IOPL=0 AF PF ]
cs             0x8                 8
ss             0x10                16
ds             0x10                16
es             0x10                16
fs             0x0                 0
gs             0x0                 0
fs_base        0x0                 0
gs_base        0x0                 0
k_gs_base      0x0                 0
cr0            0x80010011          [ PG WP ET PE ]
cr2            0x0                 0
cr3            0x3ff000            [ PDBR=1023 PCID=0 ]
```

Img 5.Output of “info registers”

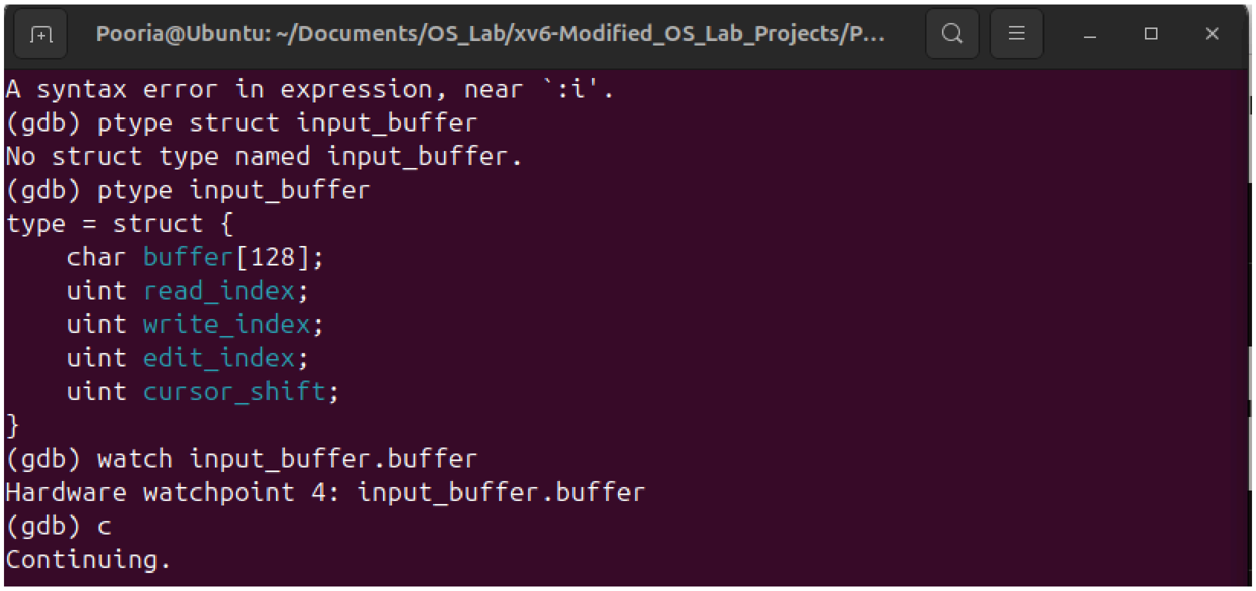

Pooria@Ubuntu: ~/Documents/OS_Lab/xv6-Modified_OS_Lab_Projects/P...

```
$10 = 15
(gdb) x $esi
0xf:    Cannot access memory at address 0xf
(gdb) info locals
i = <optimized out>
write_idx = <optimized out>
i = <optimized out>
write_idx = <optimized out>
expr_end = <optimized out>
result = <optimized out>
expr_len = <optimized out>
total_len = <optimized out>
start_pos = <optimized out>
res_str = <optimized out>
res_len = <optimized out>
temp_result = <optimized out>
shift_amount = <optimized out>
j = <optimized out>
digits = <optimized out>
num_digits = <optimized out>
k = <optimized out>
j = <optimized out>
j = <optimized out>
(gdb)
```

Img 6.Output of “info locals”

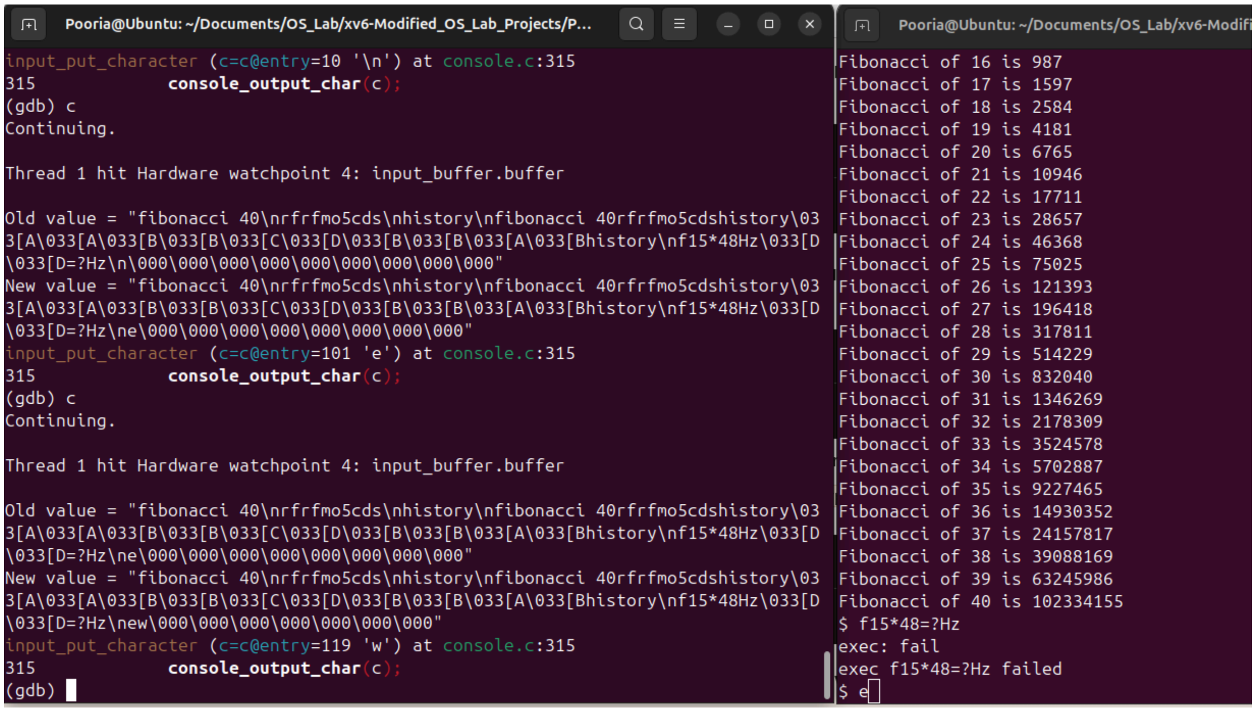
edi(Destination Index Register):This register is used as a destination pointer for string operations.It holds the address of the destination,where the data will be moved or stored.
esi(Source Index Register):It is the same as edi but it is used for sources.

6)By using “ptype struct-name” command we can find and print Typedefs defined in class.And to monitor the changes in the inner variables of the struct we can set a watchpoint on the variable that we want and then gdb will break the program each type the given variable changes and it prints out the old and new values.



```
A syntax error in expression, near `:i'.
(gdb) ptype struct input_buffer
No struct type named input_buffer.
(gdb) ptype input_buffer
type = struct {
    char buffer[128];
    uint read_index;
    uint write_index;
    uint edit_index;
    uint cursor_shift;
}
(gdb) watch input_buffer.buffer
Hardware watchpoint 4: input_buffer.buffer
(gdb) c
Continuing.
```

Img 7.Getting the struct definition and dinner variables

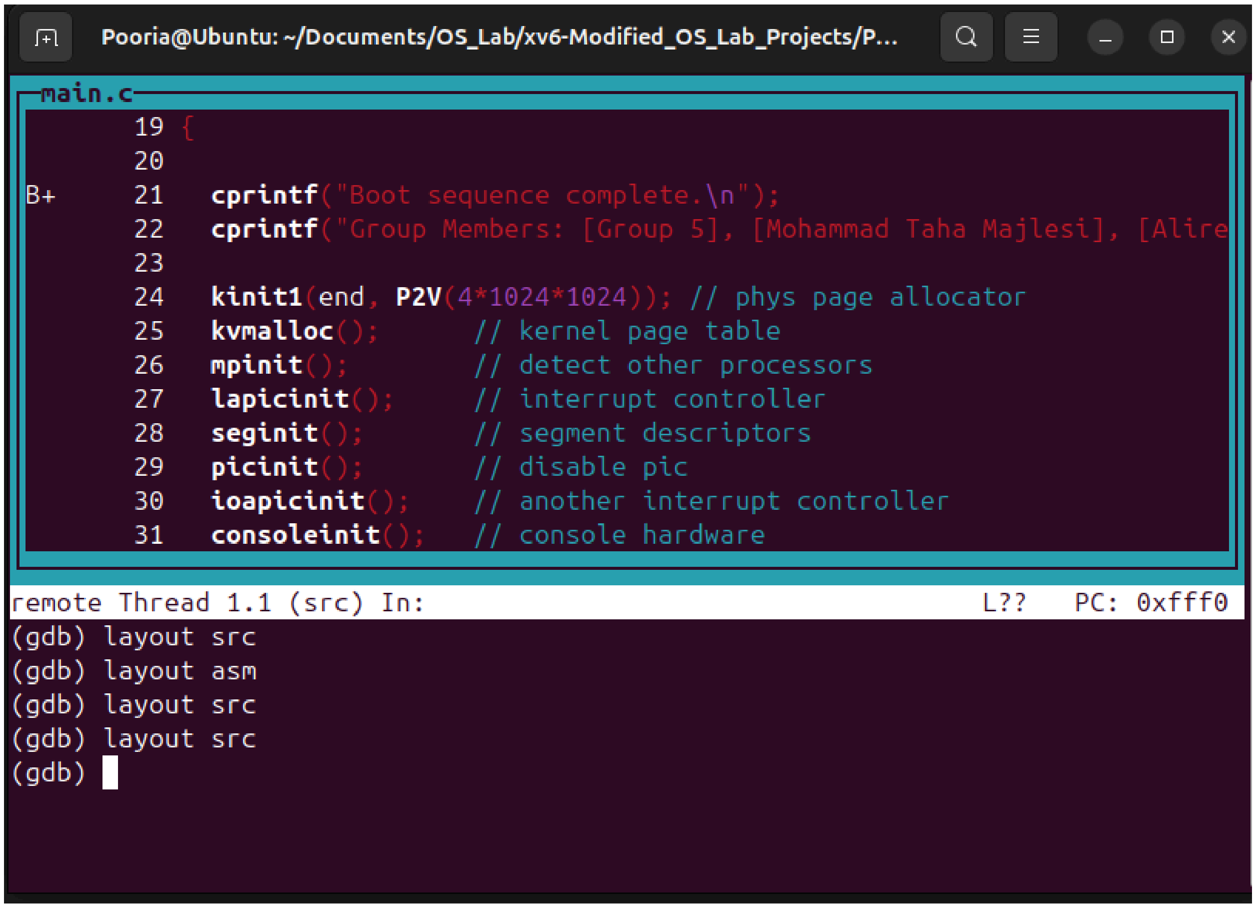


```
input_put_character (c=c@entry=10 '\n') at console.c:315
315             console_output_char(c);
(gdb) c
Continuing.

Thread 1 hit Hardware watchpoint 4: input_buffer.buffer

Old value = "fibonacci 40\nrfrfmo5cds\nhistory\nfibonacci 40rfrfmo5cdshistory\03
3[A\033[A\033[B\033[B\033[C\033[D\033[B\033[B\033[A\033[Bhistory\nf15*48Hz\033[D
\033[D=?Hz\n\000\000\000\000\000\000\000\000\000"
New value = "fibonacci 40\nrfrfmo5cds\nhistory\nfibonacci 40rfrfmo5cdshistory\03
3[A\033[A\033[B\033[B\033[C\033[D\033[B\033[B\033[A\033[Bhistory\nf15*48Hz\033[D
\033[D=?Hz\ne\000\000\000\000\000\000\000\000"
input_put_character (c=c@entry=101 'e') at console.c:315
315             console_output_char(c);
(gdb) c
Continuing.

Thread 1 hit Hardware watchpoint 4: input_buffer.buffer

Old value = "fibonacci 40\nrfrfmo5cds\nhistory\nfibonacci 40rfrfmo5cdshistory\03
3[A\033[A\033[B\033[B\033[C\033[D\033[B\033[B\033[A\033[Bhistory\nf15*48Hz\033[D
\033[D=?Hz\ne\000\000\000\000\000\000\000\000"
New value = "fibonacci 40\nrfrfmo5cds\nhistory\nfibonacci 40rfrfmo5cdshistory\03
3[A\033[A\033[B\033[B\033[C\033[D\033[B\033[B\033[A\033[Bhistory\nf15*48Hz\033[D
\033[D=?Hz\new\000\000\000\000\000\000\000"
input_put_character (c=c@entry=119 'w') at console.c:315
315             console_output_char(c);
(gdb)
```

```
Fibonacci of 16 is 987
Fibonacci of 17 is 1597
Fibonacci of 18 is 2584
Fibonacci of 19 is 4181
Fibonacci of 20 is 6765
Fibonacci of 21 is 10946
Fibonacci of 22 is 17711
Fibonacci of 23 is 28657
Fibonacci of 24 is 46368
Fibonacci of 25 is 75025
Fibonacci of 26 is 121393
Fibonacci of 27 is 196418
Fibonacci of 28 is 317811
Fibonacci of 29 is 514229
Fibonacci of 30 is 832040
Fibonacci of 31 is 1346269
Fibonacci of 32 is 2178309
Fibonacci of 33 is 3524578
Fibonacci of 34 is 5702887
Fibonacci of 35 is 9227465
Fibonacci of 36 is 14930352
Fibonacci of 37 is 24157817
Fibonacci of 38 is 39088169
Fibonacci of 39 is 63245986
Fibonacci of 40 is 102334155
$ f15*48=?Hz
exec: fail
exec f15*48=?Hz failed
$ e
```

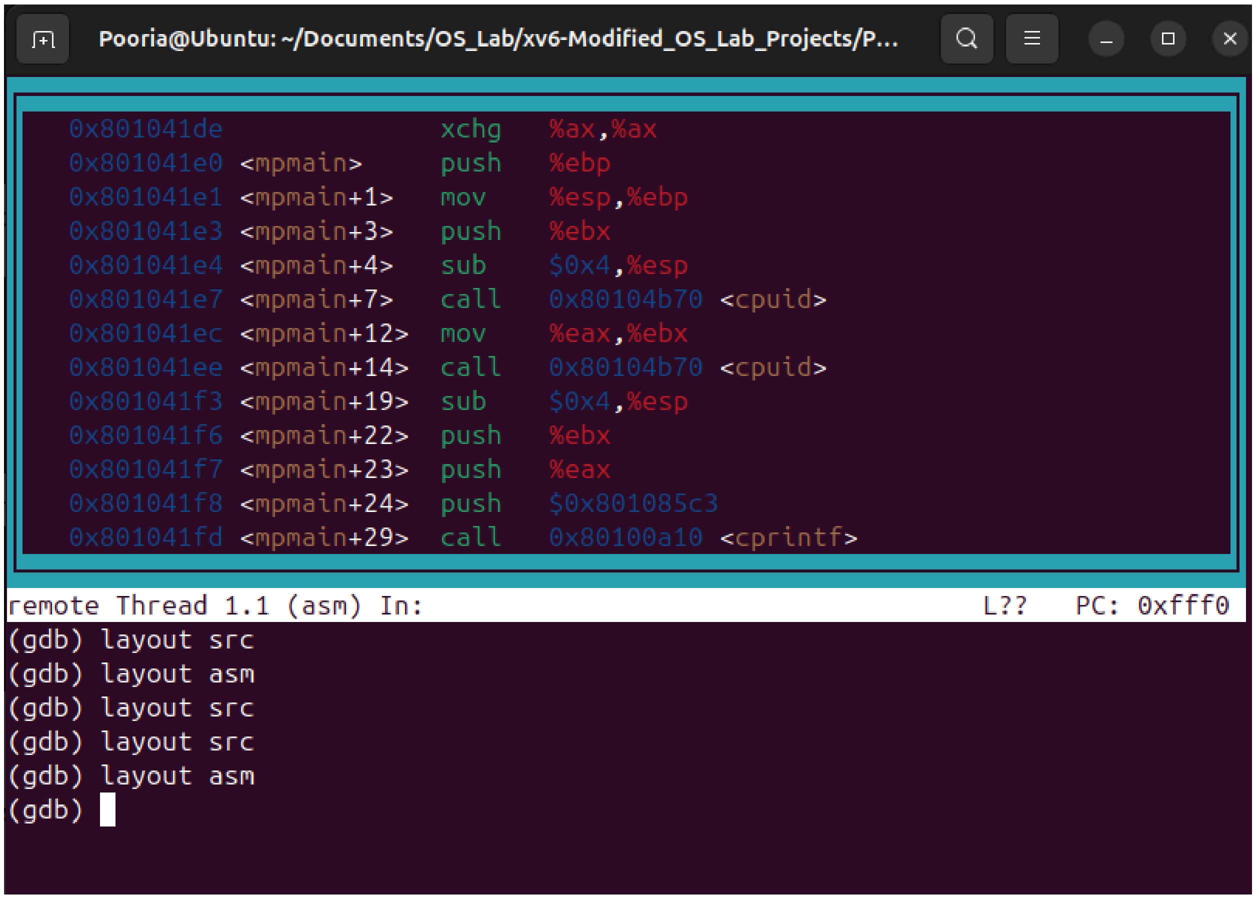
Img 8.Monitoring the changes of our input_buffer which causes the program to stop after typing each character into our console.

7)The "layout src" command displays the source code of the program in a window which allows us to see the high-level code that we are debugging. The "layout asm" command displays the assembly instructions of the program in a window which allows us to see the low-level assembly instructions generated by our code.



```
Pooria@Ubuntu: ~/Documents/OS_Lab/xv6-Modified_OS_Lab_Projects/P...
main.c
{

B+   cprintf("Boot sequence complete.\n");
     cprintf("Group Members: [Group 5], [Mohammad Taha Majlesi], [Alire

     kinit1(end, P2V(4*1024*1024)); // phys page allocator
     kvmalloc();       // kernel page table
     mpinit();         // detect other processors
     lapicinit();      // interrupt controller
     seginit();        // segment descriptors
     picinit();        // disable pic
     ioapicinit();     // another interrupt controller
     consoleinit();    // console hardware
remote Thread 1.1 (src) In:                               L??   PC: 0xfff0
(gdb) layout src
(gdb) layout asm
(gdb) layout src
(gdb) layout src
(gdb)
```

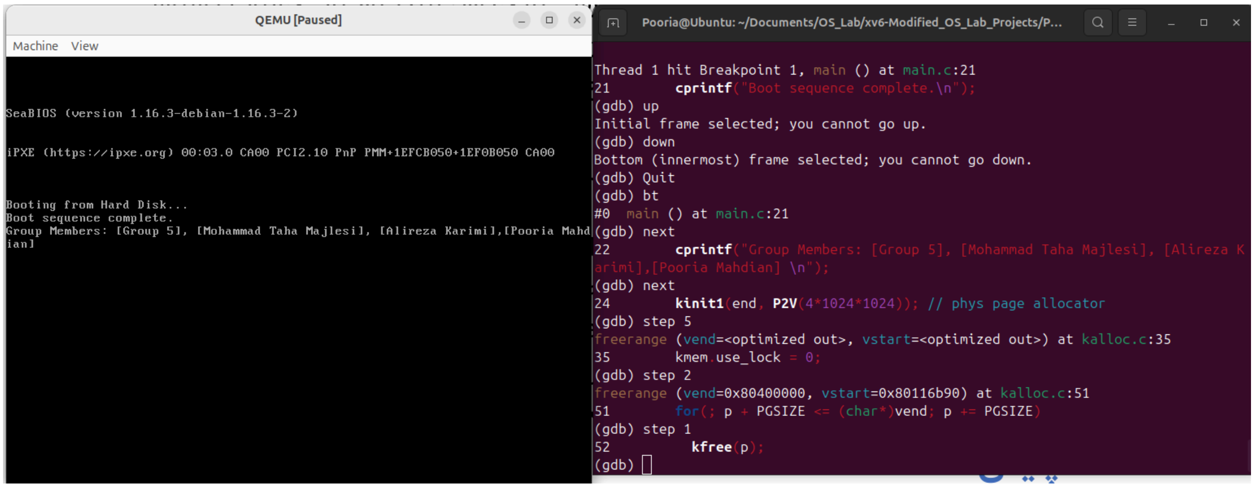
Img 9.”layout src” result



```
Pooria@Ubuntu: ~/Documents/OS_Lab/xv6-Modified_OS_Lab_Projects/P...

   0x801041de                xchg   %ax,%ax
   0x801041e0 <mpmain>       push   %ebp
   0x801041e1 <mpmain+1>     mov    %esp,%ebp
   0x801041e3 <mpmain+3>     push   %ebx
   0x801041e4 <mpmain+4>     sub    $0x4,%esp
   0x801041e7 <mpmain+7>     call   0x80104b70 <cpuid>
   0x801041ec <mpmain+12>    mov    %eax,%ebx
   0x801041ee <mpmain+14>    call   0x80104b70 <cpuid>
   0x801041f3 <mpmain+19>    sub    $0x4,%esp
   0x801041f6 <mpmain+22>    push   %ebx
   0x801041f7 <mpmain+23>    push   %eax
   0x801041f8 <mpmain+24>    push   $0x801085c3
   0x801041fd <mpmain+29>    call   0x80100a10 <cprintf>

remote Thread 1.1 (asm) In:                              L??   PC: 0xfff0
(gdb) layout src
(gdb) layout asm
(gdb) layout src
(gdb) layout src
(gdb) layout asm
(gdb) 
```

Lmg 10."layout asm" result

8)As mentioned in the project documents "next" and "step" are two commands that can be used to move from the breakpoint.The difference is that with "step" we can define the number of steps that we want to take forward inside our C program but next only goes to the next command in the C program.



```
QEMU [Paused]
Machine   View

SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 CA00

Booting from Hard Disk...
Boot sequence complete.
Group Members: [Group 5], [Mohammad Taha Majlesi], [Alireza Karimi],[Pooria Mahd
ian]
```

```
Pooria@Ubuntu: ~/Documents/OS_Lab/xv6-Modified_OS_Lab_Projects/P...

Thread 1 hit Breakpoint 1, main () at main.c:21
21          cprintf("Boot sequence complete.\n");
(gdb) up
Initial frame selected; you cannot go up.
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) Quit
(gdb) bt
#0  main () at main.c:21
(gdb) next
22          cprintf("Group Members: [Group 5], [Mohammad Taha Majlesi], [Alireza K
arimi],[Pooria Mahdian] \n");
(gdb) next
24          kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb) step 5
freerange (vend=<optimized out>, vstart=<optimized out>) at kalloc.c:35
35          kmem.use_lock = 0;
(gdb) step 2
freerange (vend=0x80400000, vstart=0x80116b90) at kalloc.c:51
51          for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
(gdb) step 1
52            kfree(p);
(gdb) 
```

Img 11.Difference of "next" and "step"