

محمد امین توانایی رنجبر 810101396 - سیدعلی تهامی 810101397 - مهدی وجهی 810101558

جواب پرسش یک

1. مدیریت منابع سخت افزاری
2. مدیریت و پاسخ به برنامه های کاربر و اپلیکیشن ها
3. ارتباط سخت افزار با نرم افزار

جواب پرسش سه

`$ (LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBS) -b binary initcode entryother`

این خط کد برنامه های نهایی را لینک میکند و فایل نهایی را میسازد

جواب پرسش چهار

ULIB: فایل های اجرایی هستند که به برنامه های سطح کاربر اجازه میدهند بدون نوشتن دوباره آنها در کد شان بتوانند از آنها استفاده کنند.

UPROGS: برنامه های سطح کاربر هستند.

جواب پرسش نه

چون بعضی جاها که میخواهیم به شدت performance و optimization و سرعت را بالا ببریم ممکن است کامپایلر به شدت efficient عمل نکند و لازم باشد کد هایی به زبان سطح پایین تر زده شود

جواب پرسش نوزده

- سائز مموری (mm->total_vm)
- پوینتر به آخر استک مخصوص پردازش (thread_info->task->stack)
- استیت فعلی پردازش (state)
- پوینتر به پیج تبیل (mm->pgd)
- pid (مشترک)
- پوینتر به فریم ترپ

- اسم پردازش (comn)
- استاتوسی که اگر صفر نباشد پردازش کشته شده است (signal->flags)
- پوینتری به پوشه مشغول (fs->pwd)
- پوینترهایی برای باز کردن فایل در دیسک (files_struct)

جواب پرسش بیست و سوم

بخش انحصاری : تنظیمات مخصوص هر کرنل اختصاصی است که موسوم به segimit است
 بخش مشترک : interrupt controller

چاپ کردن اسامی

```

Machine  View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00

Booting from Hard Disk...
cpu1: starting 1
+ [0:32mMahdi Vajhi (mvajhi@ut.ac.ir) 810101558+ [0m
+ [0:32mMohammad Amin Tavanai Ranjbar (tavanaim@gmail.com) 810101396+ [0m
+ [0:32mSayyed Ali tahami (sayyedali.tahami@ut.ac.ir) 810101397+ [0m
cpu0: starting 0
+ [0:32mMahdi Vajhi (mvajhi@ut.ac.ir) 810101558+ [0m
+ [0:32mMohammad Amin Tavanai Ranjbar (tavanaim@gmail.com) 810101396+ [0m
+ [0:32mSayyed Ali tahami (sayyedali.tahami@ut.ac.ir) 810101397+ [0m
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ _

```

سوالات قابليت كنسول

```
Machine View

Booting from Hard Disk...
cpu1: starting 1
-[0:32mMahdi Vajhi (mvajhi@ut.ac.ir) 810101558+[0m
-[0:32mMohammad Amin Tavanai Ranjbar (tavanaim@gmail.com) 810101396+[0m
-[0:32mSayyed Ali tahami (sayyedali.tahami@ut.ac.ir) 810101397+[0m
cpu0: starting 0
-[0:32mMahdi Vajhi (mvajhi@ut.ac.ir) 810101558+[0m
-[0:32mMohammad Amin Tavanai Ranjbar (tavanaim@gmail.com) 810101396+[0m
-[0:32mSayyed Ali tahami (sayyedali.tahami@ut.ac.ir) 810101397+[0m
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ echo 1

$ echo 2

$ history
  1  echo 1
  2  echo 2
$ aa12/2=?f
exec: fail
exec aa6f failed
$
```

برنامه سطح كاربر

```
Machine View

SeaBIOS (version 1.13.0-lubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00

Booting from Hard Disk...
cpu1: starting 1
<[0:32mMahdi Vajhi (mvajhi@ut.ac.ir) 810101558<[0m
<[0:32mMohammad Amin Tavanai Ranjbar (tavanaim@gmail.com) 810101396<[0m
<[0:32mSayyed Ali tahami (sayyedali.tahami@ut.ac.ir) 810101397<[0m
cpu0: starting 0
<[0:32mMahdi Vajhi (mvajhi@ut.ac.ir) 810101558<[0m
<[0:32mMohammad Amin Tavanai Ranjbar (tavanaim@gmail.com) 810101396<[0m
<[0:32mSayyed Ali tahami (sayyedali.tahami@ut.ac.ir) 810101397<[0m
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ encode QWjfdJDJ wowown KDKDskskd
$ cat result.txt
HNawawUAUA nfnfne BUBUjbjbU
$
```

Q2:

Memory Layout: This file outlines the kernel's memory structure and specifies how object files should be linked together during the compilation process.

Bootloader: These files are crucial for booting xv6, containing the code that loads the kernel into memory and begins its execution.

Low-Level Hardware: These files are designed to interact directly with the hardware, managing block-level disk I/O operations, providing disk interface drivers, and overseeing console input and output.

Pipes: These files implement the pipe mechanism for inter-process communication, allowing processes to share a common buffer so one can write data for another to read.

System Calls: This set of files includes the system call handler, implementing kernel functions associated with each call. System calls enable user programs to perform privileged operations, like file handling and process management.

Locks: These files offer synchronization mechanisms, such as spinlocks and sleeplocks, ensuring safe access to shared resources among multiple threads or processes.

Basic Headers: These files define fundamental data types, constants, and function prototypes that are consistently used throughout the xv6 codebase.

Processes: These files manage process activities in xv6, implementing functions for creating, scheduling, and switching between processes, as well as loading and executing programs.

File System: This group of files implements the file system layer in xv6, managing files, directories, and disk I/O operations. They handle reading and writing files, creating and deleting files, and navigating directories.

String Operations: These files provide utility functions for manipulating strings, covering operations such as copying, concatenating, and comparing strings.

Q13:

The address 0x100000 (or 1MB in decimal) is commonly designated as the starting point for loading the kernel in various operating systems, including xv6. This practice has its roots in the early days of personal computing. The reasons for this convention include:

Memory Layout: In x86-based systems, the memory below 1MB is typically reserved for essential components like BIOS, video memory, and other I/O devices. By loading the kernel above 1MB, it prevents any potential overwriting of these critical areas.

Protected Mode: The x86 processor begins in real mode (16-bit), where it can only access 1MB of memory. To utilize more memory and enable advanced features such as virtual memory and multitasking, the processor must switch to protected mode. Loading the kernel above 1MB ensures it remains accessible after this transition.

Memory Segmentation: In real mode, memory is segmented, with each segment limited to a maximum of 64KB. The address 0x100000 marks the beginning of the first segment that does not overlap with others, making it a suitable location for kernel loading.

Q18:

In the xv6 operating system, each component of the kernel and user commands is stored in memory and defined by a descriptor in the Global Descriptor Table (GDT). This descriptor contains information such as the start of the code, its size, and the Current Privilege Level (CPL).

When an instruction is executed, the GDT is referenced to locate the corresponding code, and the relevant page table is accessed. After converting virtual addresses to physical addresses, the instruction can be executed. During this process, privilege levels are determined through either the CPL or Descriptor Privilege Level (DPL).

Specifically, the `SEG_USER` segment is used in xv6 to distinguish between user space and kernel space. This separation is essential for maintaining system security and stability.

xv6 employs a segmented memory model, where each segment is linked to a specific privilege level. `SEG_USER` represents segments allocated to user space, which operates with lower privileges than kernel space. This design prevents user programs from directly accessing or altering kernel code or data, thereby mitigating the risk of system crashes or security vulnerabilities.

When a user program is executed, the CPU enters user mode, restricting access to memory within the `SEG_USER` segment. Any attempt to reach memory outside this segment, such as kernel memory, triggers a segmentation fault, leading to the termination of the program.

The use of `SEG_USER` for user data and code is a critical mechanism that preserves the integrity and reliability of the system by safeguarding the kernel from unauthorized access or modifications by user programs. This separation between user and kernel spaces is a fundamental principle in operating system design.

Debuging:

GDB:

1- Show breakpoints:

Several Ways to Add Breakpoints

By Function Name: Set a breakpoint at the start of a function.

```
(gdb) break main
```

By Line Number: Specify a particular line in a source file.

```
(gdb) break file.c:42
```

Conditional Breakpoints: Add breakpoints that trigger only when a specified condition is true.

```
(gdb) break my_function if x > 5
```

Address breakpoints:

```
(gdb) break *Address
```

```
(gdb) break factorial:10
Note: breakpoint 1 also set at pc 0x555555551b8.
Breakpoint 2 at 0x555555551b8: file debug.c, line 4.
(gdb) break factorial:20
Note: breakpoints 1 and 2 also set at pc 0x555555551b8.
Breakpoint 3 at 0x555555551b8: file debug.c, line 4.
(gdb) break debug.c:10
Breakpoint 4 at 0x555555551e3: file debug.c, line 11.
(gdb) info breakpoints
```

For example. I add some breakpoint to my code and you can see the results in images

Command for showing the existing breakpoints:

```
info breakpoints
```

In this picture we can see the result of this command.

```
Breakpoint 4 at 0x555555551e3: file debug.c, line 11.
(gdb) info breakpoints
Num      Type             Disp Enb Address                  What
1        breakpoint      keep y   0x0000555555551b8 in factorial at debug.c:4
          breakpoint already hit 2 times
2        breakpoint      keep y   0x0000555555551b8 in factorial at debug.c:4
3        breakpoint      keep y   0x0000555555551b8 in factorial at debug.c:4
4        breakpoint      keep y   0x0000555555551e3 in factorial at debug.c:11
(gdb)
```

2. delete a Breakpoint

For deleting a breakpoint we use this command:

Delete <breakpoint num>

Also we can use this command for deleting a breakpoint from a number

In this picture we can see the result

```
(gdb) info breakpoints
Num      Type           Disp Enb Address                What
1        breakpoint     keep y   0x00005555555551b8 in factorial at debug.c:4
          breakpoint already hit 2 times
2        breakpoint     keep y   0x00005555555551b8 in factorial at debug.c:4
3        breakpoint     keep y   0x00005555555551b8 in factorial at debug.c:4
4        breakpoint     keep y   0x00005555555551e3 in factorial at debug.c:11
(gdb) delete 3
(gdb) info breakpoints
Num      Type           Disp Enb Address                What
1        breakpoint     keep y   0x00005555555551b8 in factorial at debug.c:4
          breakpoint already hit 2 times
2        breakpoint     keep y   0x00005555555551b8 in factorial at debug.c:4
4        breakpoint     keep y   0x00005555555551e3 in factorial at debug.c:11
(gdb)
```

3.\$bt command:

The command bt in GDB stands for "backtrace." It displays the call stack at the point where the program stopped (e.g., when a breakpoint is hit or an error occurs). Each

entry in the backtrace shows a function that was called, along with its parameters and the source file and line number where it was defined.

```

(gdb) delete 3
(gdb) info breakpoints
Num      Type           Disp Enb Address                  What
1        breakpoint     keep y   0x00005555555551b8 in factorial at debug
        breakpoint already hit 2 times
2        breakpoint     keep y   0x00005555555551b8 in factorial at debug
4        breakpoint     keep y   0x00005555555551e3 in factorial at debug
(gdb) bt
#0  factorial (n=9) at debug.c:4
#1  0x00005555555551df in factorial (n=10) at debug.c:9
#2  0x0000555555555239 in main () at debug.c:18
(gdb)

```

#0 factorial (n=9) at debug.c:4

#1 0x00005555555551df in factorial (n=10) at debug.c:9

#2 0x0000555555555239 in main () at debug.c:18

#0: This is the most recent function that was called. The program is currently inside the factorial function, where n is 9. The function is defined in debug.c at line 4.

#1: This shows the function that called the current factorial function. It indicates that factorial was called with n=10 from somewhere in the factorial function itself (likely a recursive call). This call is recorded at line 9 in the same file.

#2: This is the entry point of your program, the main function, which is defined at line 18 in debug.c. It shows the function that originally started the execution.

4.difference between print and x

First we use the help command to find the usage of functions:

```

(gdb) help print
print, inspect, p
Print value of expression EXP.
Usage: print [[OPTION]... --] [/FMT] [EXP]

Options:
  -address [on|off]
    Set printing of addresses.

  -array [on|off]
    Set pretty formatting of arrays.

  -array-indexes [on|off]
    Set printing of array indexes.

  -nibbles [on|off]
    Set whether to print binary values in groups of four bits.

  -characters NUMBER|elements|unlimited
    Set limit on string chars to print.
    "elements" causes the array element limit to be used.
    "unlimited" causes there to be no limit.

  -elements NUMBER|unlimited
    Set limit on array elements to print.
    "unlimited" causes there to be no limit.
    This option also applies to the "print characters"

```



```

but no count or size letter (see 'x' command).
(gdb) help x
quit
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
    t(binary), f(float), a(address), i(instruction), c(char), s(string)
    and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format. If a negative number is specified, memory is
examined backward from the address.

Defaults for format and size letters are those previously used.
Default count is 1. Default address is following last thing printed
with this command or "print".
(gdb) █

```

x Command:

- The x (examine) command is used to examine memory at a specific address. It allows us to specify the format and the number of units to read from memory.
- Syntax: x/Nf where N is the number of units and f is the format (e.g., d for decimal, x for hexadecimal).

print Command:

- The print command is used to evaluate and display the value of a variable or expression. It is primarily used for variables in the context of the program being debugged.
- Syntax: print <variable> or print <expression>.

An example: we print the value of variable n that is equal to 10

```
(gdb) run
Starting program: /home/sayyedali/xv6-public/my_program

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for system-supplied DSO at 0x7ffff7fc3000
Download failed: No route to host. Continuing without separate debug info for s
ystem-supplied DSO at 0x7ffff7fc3000.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Enter a non-negative integer: 10

Breakpoint 1, factorial (n=10) at debug.c:4
4      if (n < 0) {
(gdb) print n
$1 = 10
(gdb) print result
$2 = '\000' <repeats 113 times>
(gdb)
```

Usage of x command: we print value of x by 10 digit that is equal to 55((given input))

```
(gdb) x/10d 0x7fffffffdd3c
0x7fffffffdd3c: 54      -8864      32767      1431654879
0x7fffffffdd4c: 21845    0           0           0
0x7fffffffdd5c: 55      -8832
(gdb) info registers
rax             0x36             54
rbx             0x7fffffffdea8    140737488346792
rcx             0x0              0
rdx             0x0              0
rsi             0x37             55
rdi             0x36             54
rbp             0x7fffffffdd40    0x7fffffffdd40
rsp             0x7fffffffdd30    0x7fffffffdd30
r8              0xa              10
r9              0x0              0
r10             0x7ffff7db1fc0    140737351720896
r11             0x7ffff7e038e0    140737352055008
r12             0x1              1
r13             0x0              0
r14             0x55555557da8     93824992247208
r15             0x7ffff7ffd000     140737354125312
rip             0x555555551c5     0x555555551c5 <factorial+28>
eflags          0x10206           [ PF IF RF ]
cs              0x33             51
ss              0x2b             43
ds              0x0              0
es              0x0              0
fs              0x0              0
gs              0x0              0
fs_base         0x7ffff7fab740    140737353791296
```

To see value of a specific register we can use command `info registers <register_name>`

```
(gdb) info registers rax
rax                0x36                54
(gdb) info locals
```

5.Registers and local variables:

Checking Register Status

To view the current state of the CPU registers in GDB, you can use the `info registers` command:

If we use `info registers` we can see the value of all register and their status

```
(gdb) info registers
rax                0x36                54
rbx                0x7fffffffdea8      140737488346792
rcx                0x0                 0
rdx                0x0                 0
rsi                0x37                55
rdi                0x36                54
rbp                0x7fffffffdd40      0x7fffffffdd40
rsp                0x7fffffffdd30      0x7fffffffdd30
r8                 0xa                 10
r9                 0x0                 0
r10                0x7ffff7db1fc0      140737351720896
r11                0x7ffff7e038e0      140737352055008
r12                0x1                 1
r13                0x0                 0
r14                0x555555557da8      93824992247208
r15                0x7ffff7ffd000      140737354125312
rip                0x5555555551c5      0x5555555551c5 <factorial+28>
eflags             0x10206             [ PF IF RF ]
cs                 0x33                51
ss                 0x2b                43
ds                 0x0                 0
es                 0x0                 0
fs                 0x0                 0
gs                 0x0                 0
fs_base            0x7ffff7fab740      140737353791296
gs_base            0x0                 0
```

If you want to see all local variables in the current stack frame, you can also use the `info locals` command:

```
(gdb) info locals
```

And it's the result, it's equal to 0 because I don't have any local var in my debug.c program.

```
(gdb) info locals  
No locals.
```

x86 Architecture: EDI and ESI Registers:

- In x86 architecture, the EDI (Extended Destination Index) and ESI (Extended Source Index) registers are commonly used for string and memory operations

EDI (Extended Destination Index):

- Primarily used as a pointer for the destination in string operations. When performing operations like MOVS, LODS, and STOS, the EDI register points to the destination buffer where data will be written.

ESI (Extended Source Index):

- Used as a pointer for the source in string operations. It typically points to the source buffer from which data will be read during these operations.

In the context of function calls, EDI and ESI can also hold values that are preserved across function calls according to the calling convention used, making them useful for keeping track of data across different parts of a program.

6.Input structure:

```
(gdb) break console.c:542

Breakpoint 2 at 0x80101470: console.c:542. (3 locations)
(gdb) c

Continuing.
[Switching to Thread 1.1]

Thread 1 hit Breakpoint 2.1, handle_char_input (c=115) at console.c:474
474      return (c == '\r') ? '\n' : c;

(gdb) c

Continuing.

Thread 1 hit Breakpoint 2.1, handle_char_input (c=97) at console.c:474
474      return (c == '\r') ? '\n' : c;

(gdb) ptype input

type = struct {
    char buf[128];
    char history[12][128];
    int last_line_count;
    int history_line;
    int r;
    int w;
    int e;
    int newline_pos;
    int end_pos;
    int current_pos;
}
(gdb)
```

our input structure is defined as follows:

- Buf: it's our buffer storage.
- History: store command's history
- Last_line_count: number of max capacity of history that have filled.
- History_line: it's return index of line that user is watching.
- R: a index to save index of last character we have read from buf
- W: index of the first character that is in the buffer but has not been read.
- E: determines the current index of the buffer which is being written to.
- Newline_pos: it's return the newline_pos
- End_pos: it's return last index position
- Current_pos: it's return current pos

Result In out program:

[illegible]

این برنامه را برای زمانی مقدار بندی میکنیم تا هر کاراکتری که از ورودی ما میخواند را یک دور وارد این فرمت میکند.

در هنگام خواندن هر یک از این ورودی ها، یک دور ریکورد میگیرد و آن را در history ذخیره میکند)) (همانگونه که در شکل مشاهده میشود)) و در نهایت رشته آخر ما را به عنوان ورودی نهایی در نظر میگیرد.

7. TUI:

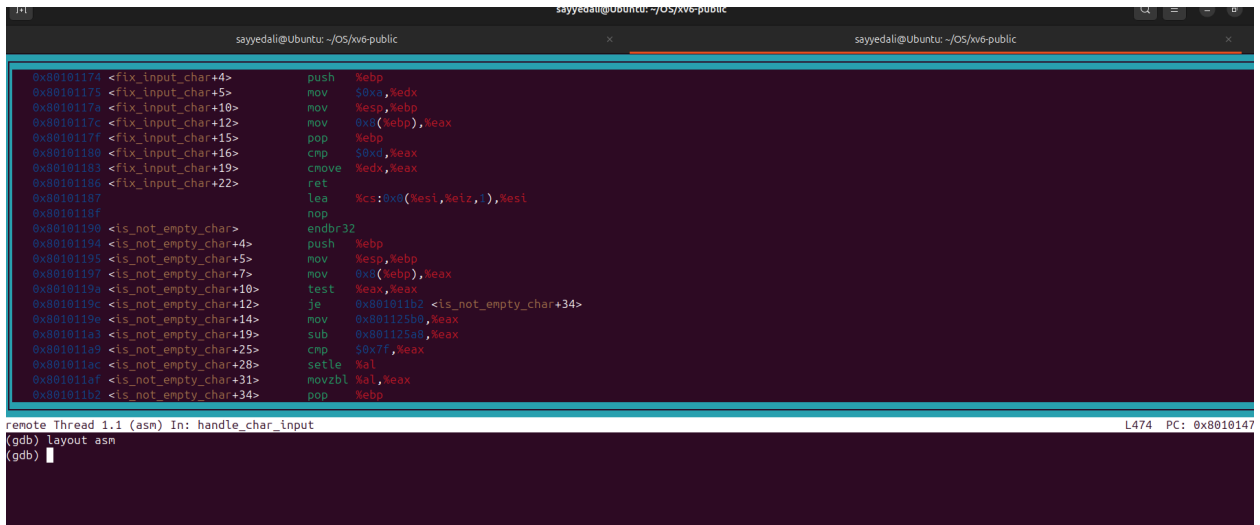
In the TUI mode in GDB, the screen is split in two sectors and source code will be shown, then we can see source code of where the program has been stopped.

layout src is used to show source code.

```
sayyedali@Ubuntu: ~/OS/xv6-public
x
sayyedali@Ubuntu: ~/OS/xv6-public

-console.c
464     execute_command();
465 }
466
467 int is_end_of_line(int c)
468 {
469     return c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF || input.end_pos - input.newline_pos == INPUT_BUF;
470 }
471
472 int fix_input_char(int c)
473 {
474     return (c == '\r') ? '\n' : c;
475 }
476
477 int is_not_empty_char(int c)
478 {
479     return c != 0 && input.e - input.r < INPUT_BUF;
480 }
481
482 void handle_end_line_in_buffer()
483 {
484     input.current_pos = input.end_pos;
485 }
```

Layout asm is used to equivalence assembly code of source code.



```
0x80101174 <fix_input_char+4>      push    %ebp
0x80101175 <fix_input_char+5>      mov     $0xa,%edx
0x80101176 <fix_input_char+10>     mov     %esp,%ebp
0x80101177 <fix_input_char+12>     mov     0xa(%ebp),%eax
0x80101178 <fix_input_char+15>     pop     %ebp
0x80101180 <fix_input_char+16>     cmp     $0xd,%eax
0x80101182 <fix_input_char+19>     cmovbe %edx,%eax
0x80101186 <fix_input_char+22>     ret
0x80101187 <is_not_empty_char>        lea     %cs:0x0(%esi,%eiz,1),%esi
0x8010118f <is_not_empty_char+4>      nop
0x80101190 <is_not_empty_char+5>      endbr32
0x80101194 <is_not_empty_char+7>      push    %ebp
0x80101195 <is_not_empty_char+8>      mov     %esp,%ebp
0x80101197 <is_not_empty_char+10>     mov     0xa(%ebp),%eax
0x80101199 <is_not_empty_char+12>     test    %eax,%eax
0x8010119e <is_not_empty_char+14>     je      0x801011b2 <is_not_empty_char+34>
0x8010119e <is_not_empty_char+14>     mov     0x801125b0,%eax
0x801011a3 <is_not_empty_char+19>     sub     0x801125a0,%eax
0x801011a9 <is_not_empty_char+25>     cmp     $0x7f,%eax
0x801011ac <is_not_empty_char+28>     setle   %al
0x801011af <is_not_empty_char+31>     movzbl  %al,%eax
0x801011b2 <is_not_empty_char+34>     pop     %ebp
```

remote Thread 1.1 (asm) In: handle_char_input L474 PC: 0x8010147
(gdb) Layout asm
(gdb)

8. Movement:

Move between functions in frame stack

- command up is used to jump out of the function to where the function is called from. It has the same functionality as step out.
- command down is used to jump to the next function in the frame stack. It has the same functionality as step in.

