جواب پرسش شماره 1:

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 $(OBJS) -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

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

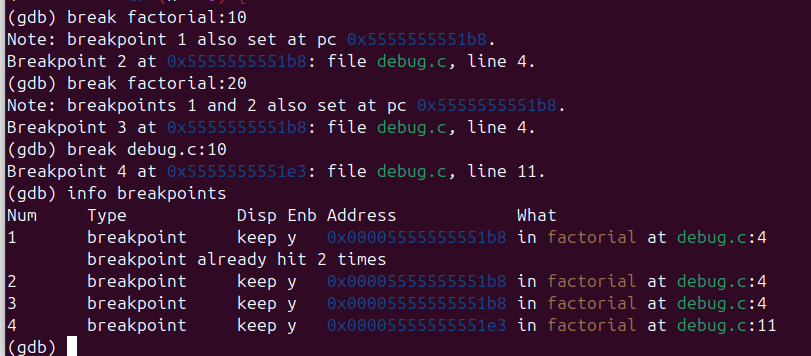
By Line Number: Specify a particular line in a source file.

(gdb) break file.c:42

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

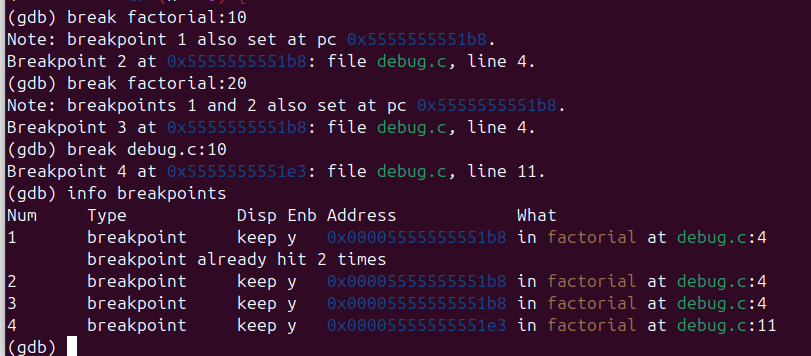
(gdb) break my\_function if x > 5

Address breakpoints:

 (gdb) break \*Address

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.

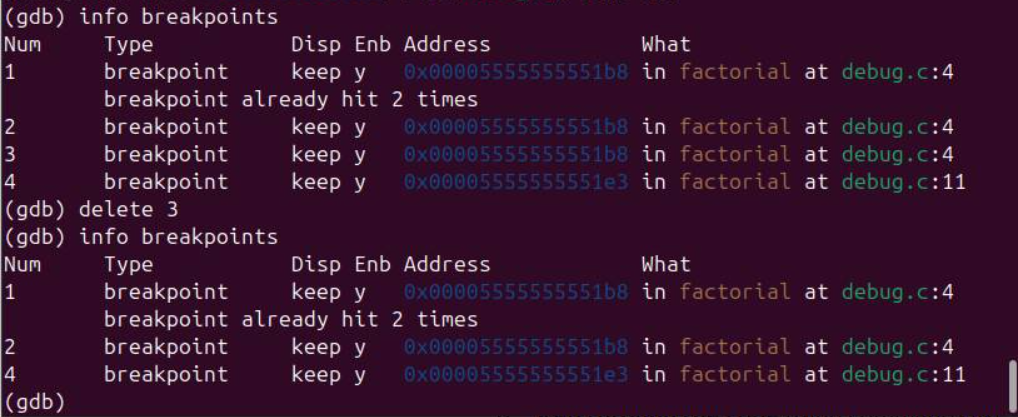
2. delete a Breakpoint

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



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.



#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

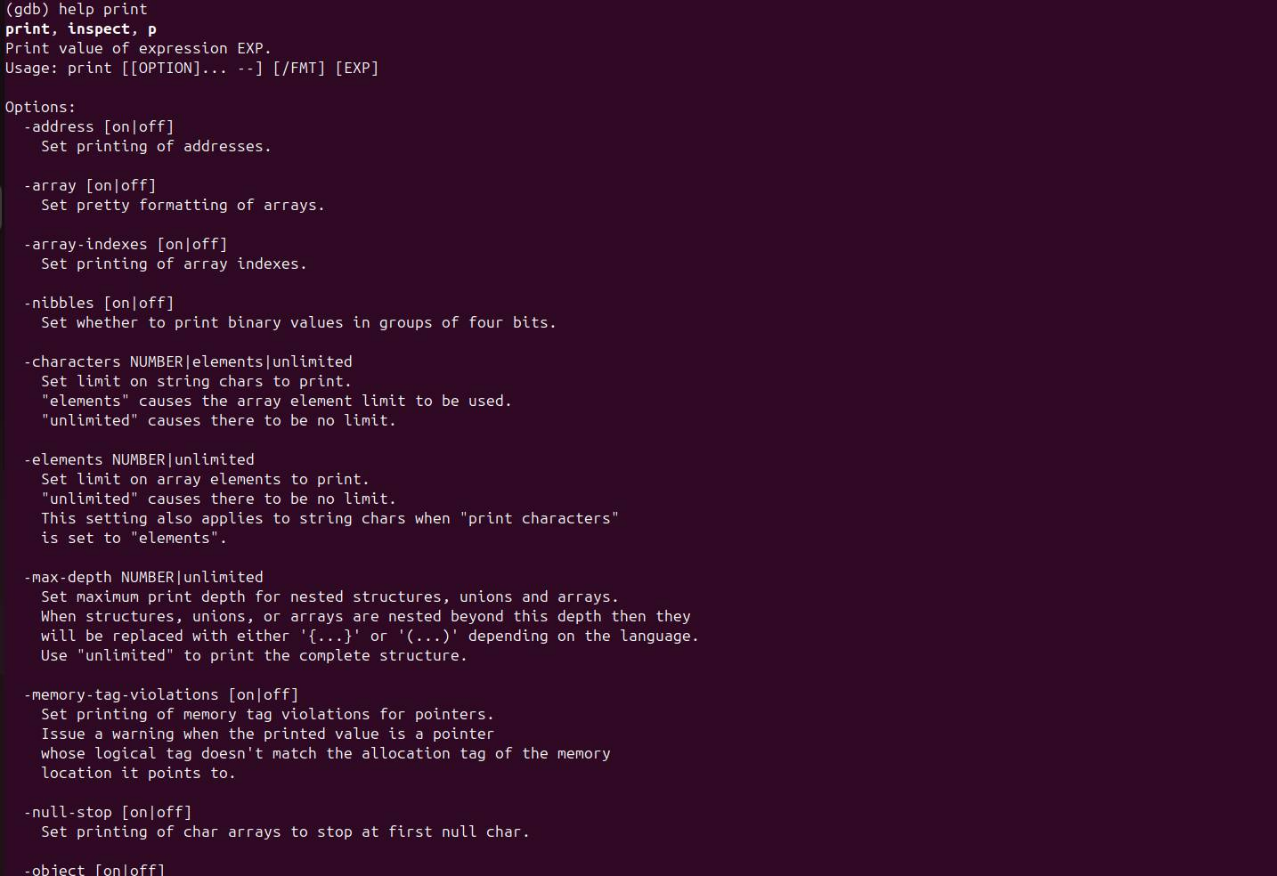
#0: This is the most recent function that was called. The program is currently inside the factorial function, where n is 9. The function is defined in debug.c at line 4.

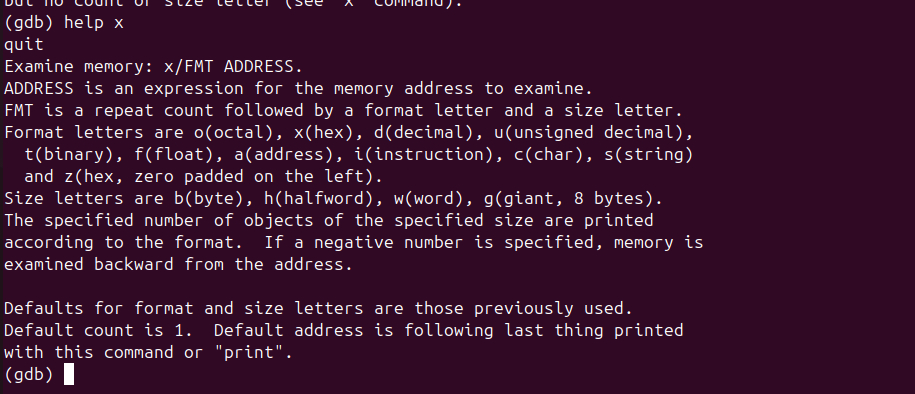
#1: This shows the function that called the current factorial function. It indicates that factorial was called with n=10 from somewhere in the factorial function itself (likely a recursive call). This call is recorded at line 9 in the same file.

#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:



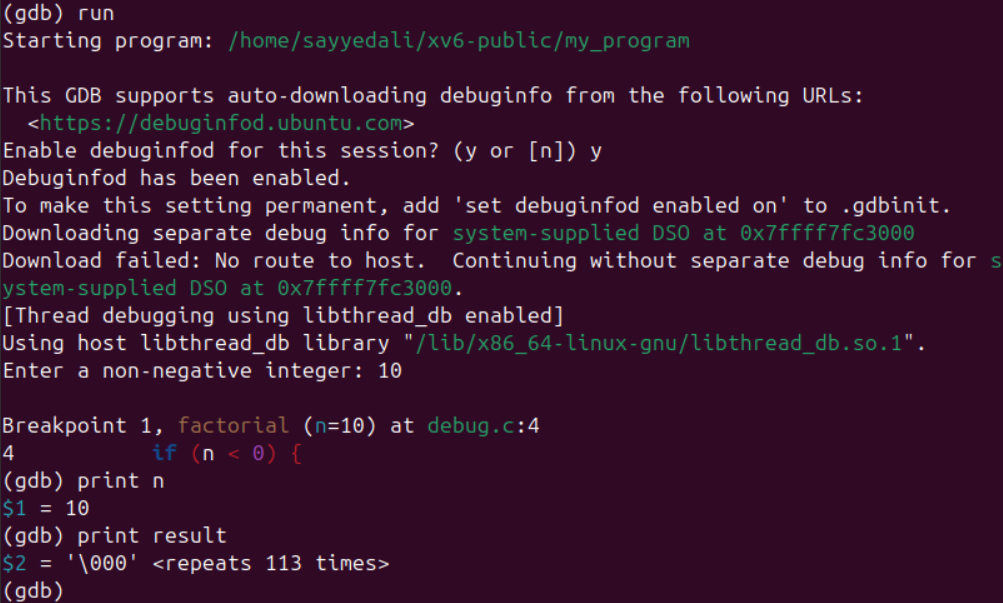


**x Command**:

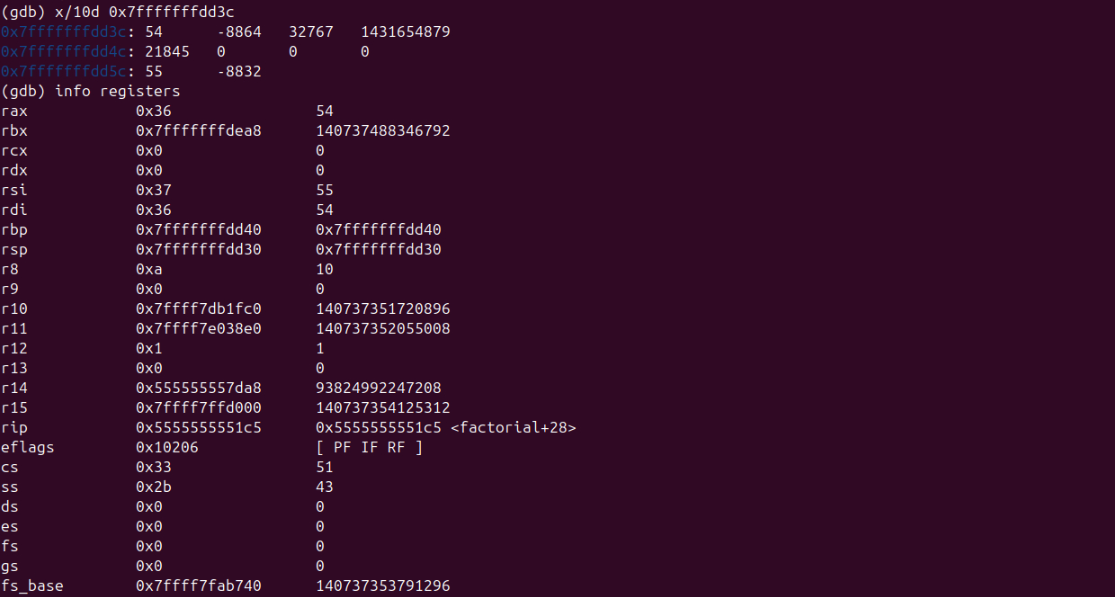
* The x (examine) command is used to examine memory at a specific address. It allows us to specify the format and the number of units to read from memory.
* Syntax: x/Nf where N is the number of units and f is the format (e.g., d for decimal, x for hexadecimal).

**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  


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



To see value of a specific register we can use command info registers <register\_name>

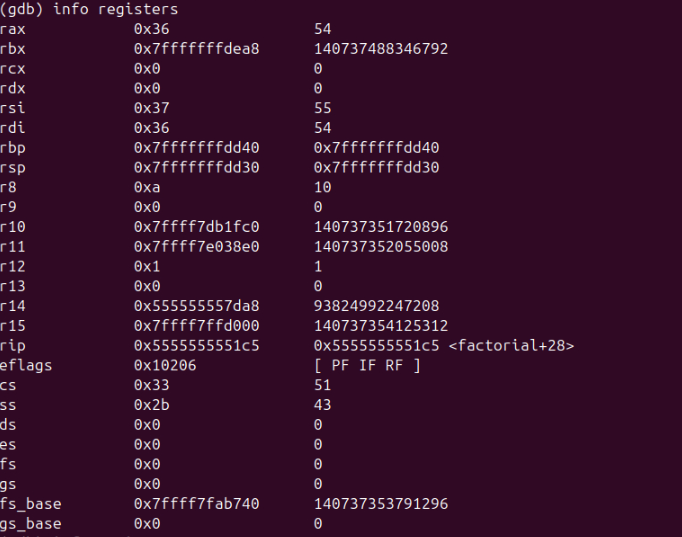


5.Registers and local variables:

**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



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.



**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)**:

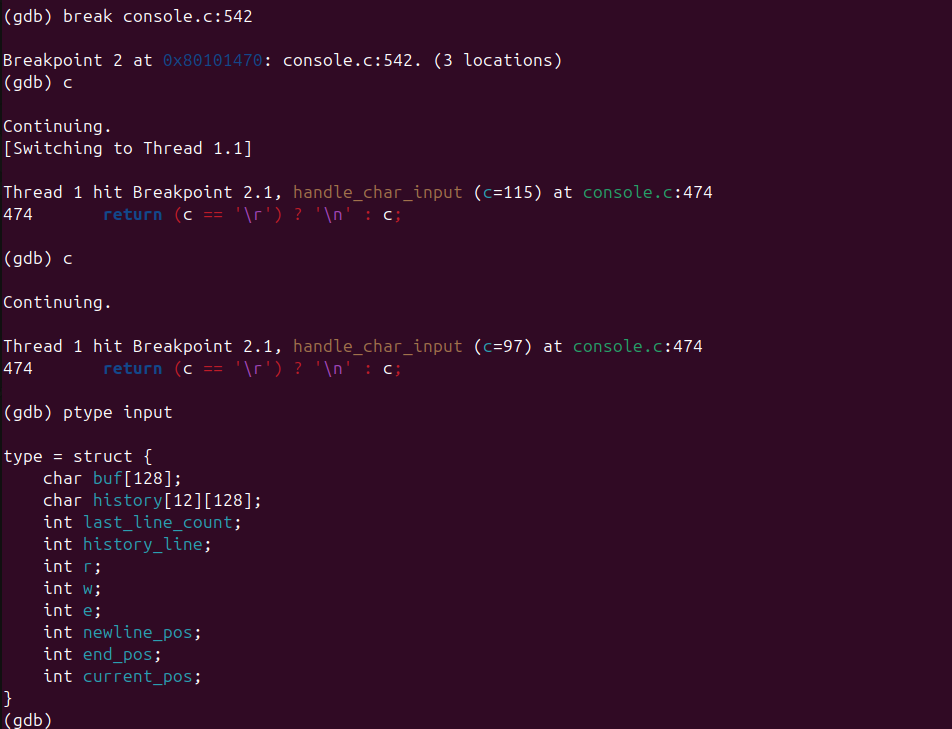
* Primarily used as a pointer for the destination in string operations. When performing operations like MOVS, LODS, and STOS, the EDI register points to the destination buffer where data will be written.

**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.

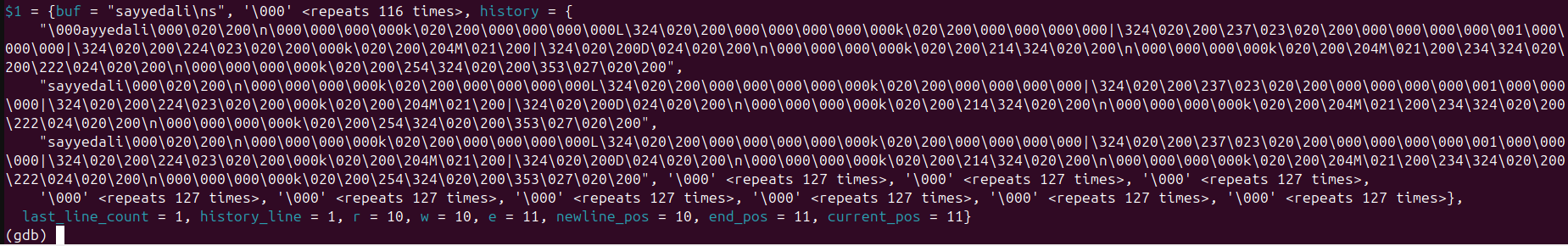
In the context of function calls, EDI and ESI can also hold values that are preserved across function calls according to the calling convention used, making them useful for keeping track of data across different parts of a program.

6.Input structure:



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: 

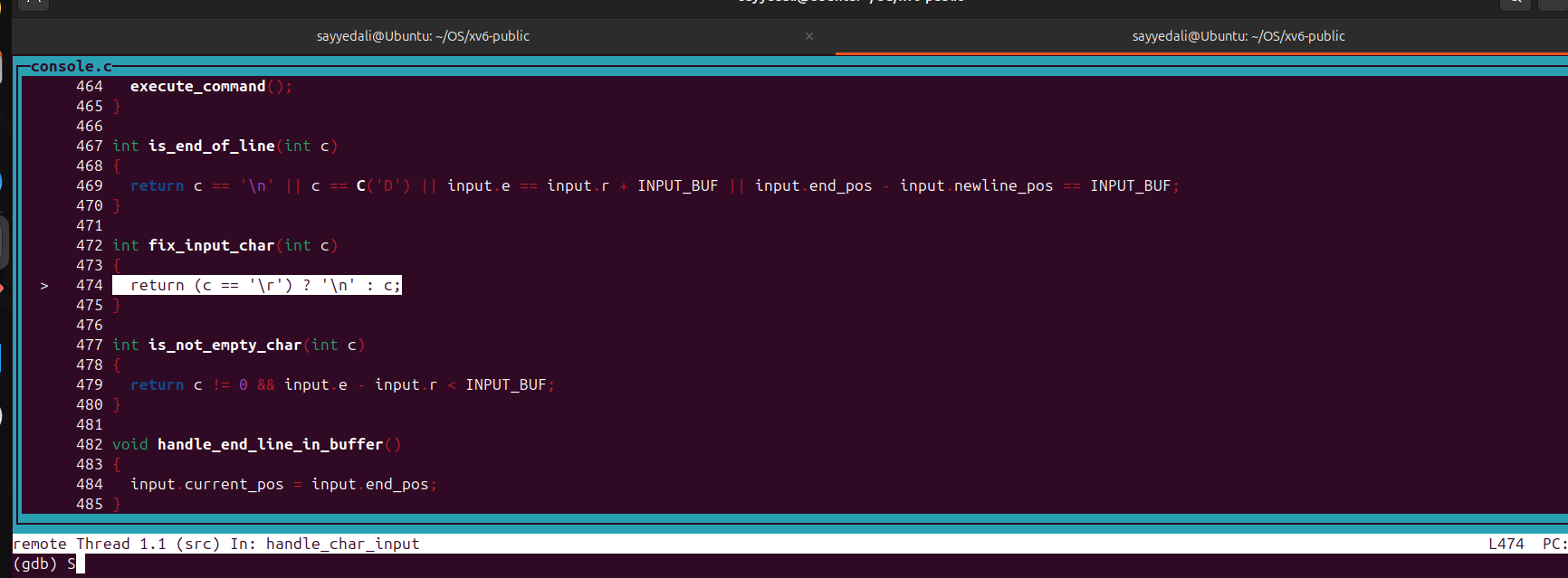
این برنامه را برای زمانی مقدار بندی میکنیم تا هر کاراکتری که از ورودی ما میخواند را یک دور وارد این فرمت میکند.

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.

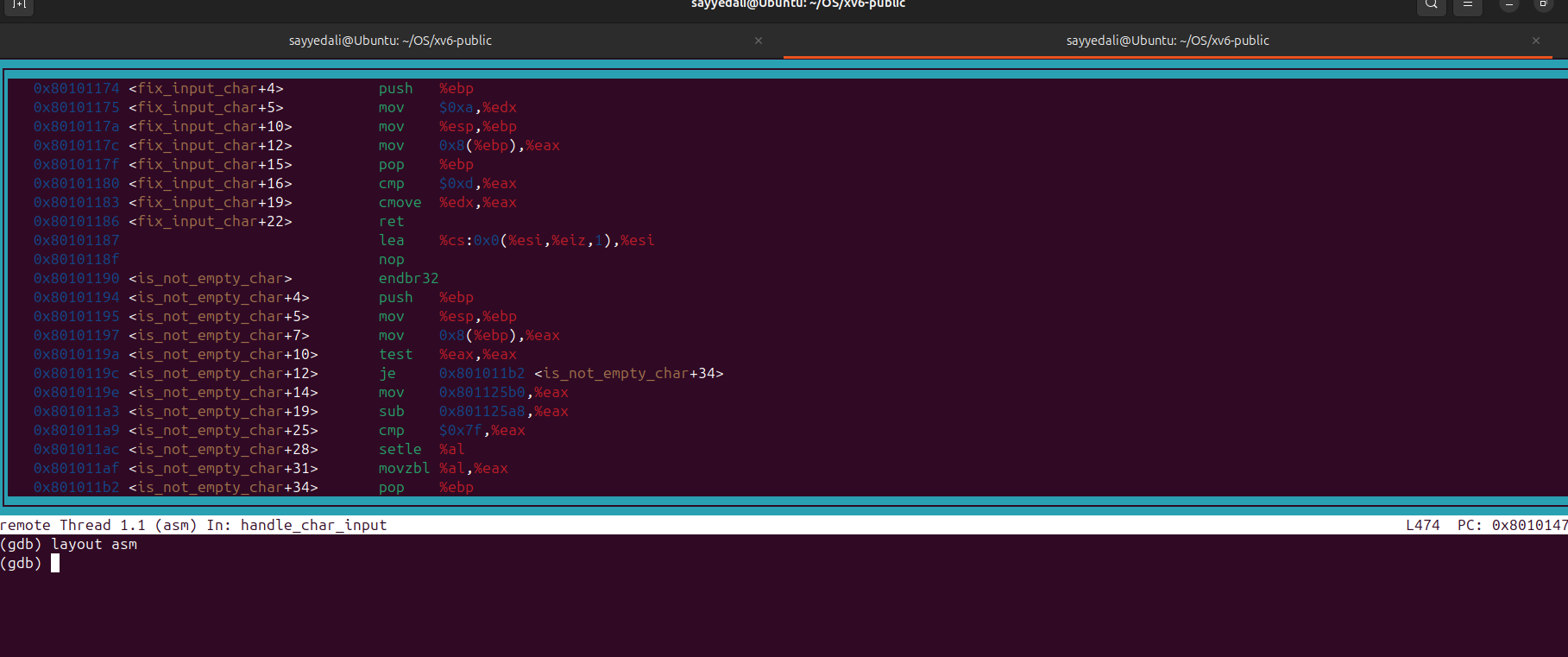
7. TUI:

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.



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



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.