

# Project report

رضا چهرقانی 810101401

امیر نداف فهمیده 810101540

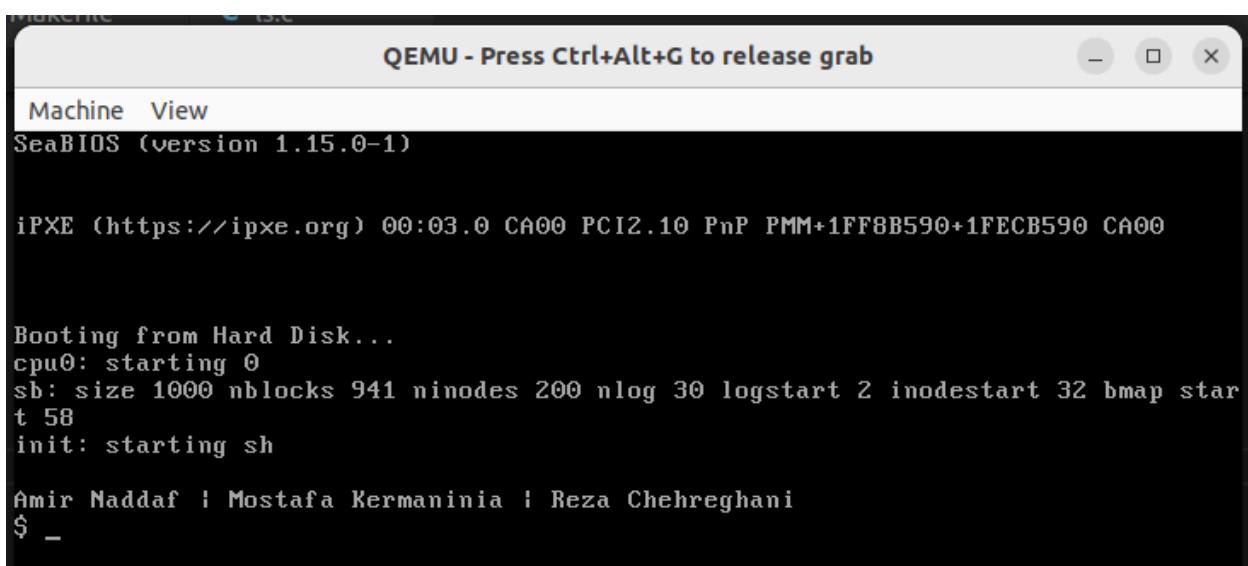
مصطفی کرمانی نیا 810101575

مخزن گیتهاب این پروژه:

<https://github.com/reza-chehreghani/OS-Lab-xv6>

## ● اضافه کردن یک متن به Message Boot

نام اعضای گروه پس از بوت شدن سیستم عامل روی ماشین مجازی Qemu، در انتهای پیام‌های نمایش داده شده در کنسول نشان داده می‌شود:



The screenshot shows a terminal window titled "QEMU - Press Ctrl+Alt+G to release grab". The window has tabs for "Machine" and "View", with "Machine" selected. The title bar also displays "SeaBIOS (version 1.15.0-1)". The terminal output is as follows:

```
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh

Amir Naddaf | Mostafa Kermaninia | Reza Chehreghani
$ -
```

---

## ● اضافه کردن قابلیت های جدید به کنسول

### 1. جلو و عقب رفتن curse

برای پیاده سازی نشانگر، علاوه بر اشاره گر به انتهای بافر، نیاز به یک اشاره گر به موقعیتی که نشانگر در آن قرار دارد، داریم. پس یک عدد دیگر به نام `c` به `input struct` اضافه می کنیم تا موقعیت نشانگر را ذخیره کنیم

```
#define INPUT_BUF 128
struct
{
    char buf[INPUT_BUF];
    uint r; // Read index
    uint w; // Write index
    uint e; // Edit index
    uint c; // Cursor index
} input, copy_input, buf_history[MAX_INDEX_HISTORY];
```

پس حال که موقعیت نشانگر را ذخیره می کنیم، باید با کلید های راست و چپ در بافر ورودی در حال ویرایش به راست و چپ جابه جا شویم. به همین علت در تابع `consoleintr` دو حالت یک برای `left` و یکی برای `right arrow` قرار می دیدم تا در صورت دیدن این ورودی ها در صورت مجاز بودن، موقعیت نشانگر را تغییر دهد.

```
case LEFT_ARROW:  
    if (input.c != input.w)  
    {  
        input.c--;  
        consputc(LEFT_ARROW);  
    }  
    if (is_copying == 1 && copying_index > 0)  
    | copying_index--;  
    break;  
case RIGHT_ARROW:  
    if (input.c != input.e)  
    {  
        input.c++;  
        consputc(RIGHT_ARROW);  
    }  
    if (is_copying == 1 && copying_index < cs_size)  
    | copying_index++;  
    break;
```

همان‌طور که مشاهده می‌کنید، اگر نشانگر در ابتدای رشته‌ی در حال تغییر نباشد، نشانگر با left arrow یک کاراکتر به سمت چپ می‌رود و اگر در انتهای رشته‌ی در حال تغییر نباشد، با right arrow یک کاراکتر به سمت راست می‌رود. همچنین در هر دو حالت تابع consputc را نیز با ورودی مناسب اجرا می‌کند.

حال هنگامی که نشانگر در میانه‌ی دستور در حال ویرایش می‌باشد، در صورت اضافه کردن یا حذف یک کاراکتر باید تمامی کاراکتر‌های سمت چپ آن را یه واحد به راست یا چپ شیفت بدهیم.

هنگامی که یک کاراکتر جدید را می‌خواهیم به بافر اضافه کنیم، اول باید تمامی کاراکتر‌های سمت راست نشانگر را یک واحد به سمت راست شیفت بدهیم و سپس کاراکتر جدید را در محل نشانگر اضافه کنیم.

```
default:
    if (c != 0 && input.e - input.r < INPUT_BUF)
    {
        c = (c == '\r') ? '\n' : c;
        // input.buf[input.e++ % INPUT_BUF] = c;
        if (c != '\n')
        {
            for (uint i = input.e++; i != input.c; i--)
                input.buf[i % INPUT_BUF] = input.buf[(i - 1) % INPUT_BUF];
            input.buf[input.c++ % INPUT_BUF] = c;
            if (is_copying == 1)
            {
                for (uint i = cs_size++; i != copying_index; i--)
                    copied_string[i] = copied_string[i - 1];
                copied_string[copying_index++] = c;
                // copied_string[copying_index] = c;
                // cs_size++;
                // copying_index++;
            }
        }
        else
            input.buf[input.e++ % INPUT_BUF] = c;
    }
```

البته هنگامی که enter زده می‌شود، فرقی نمی‌کند که در کجای متن باشد و باید خط ورودی اجرا شود.  
پس همیشه کاراکتر \n را در انتهای خط قرار می‌دهیم.

در حالتی هم که backspace زده می‌شود، باید تمامی کاراکترهای سمت راست نشانگر یک واحد به سمت چپ شیفت داده بشنود (با این کار کاراکتری که نشانگر به آن اشاره می‌کند خود به خود حذف می‌شود). در نهایت نیز تابع consputc با ورودی backspace صدا زده می‌شود.

```
case '\x7f': // Backspace
    // if(input.e != input.w){
    if (input.c != input.w)
    {
        // input.e--;
        for (uint i = input.c--; i != input.e; i++)
        | input.buf[(i - 1) % INPUT_BUF] = input.buf[i % INPUT_BUF];
        input.e--;
        consputc(BACKSPACE);
    }
    if (is_copying == 1 && copying_index > 0)
    {
        for (uint i = copying_index--; i != cs_size; i++)
        | copied_string[i - 1] = copied_string[i];
        cs_size--;
    }
    break;
```

تا الان تمام کارهایی که انجام دادیم برای درست کردن بافر input بوده است. برای آنکه این تغییرات در صفحه کنسول نیز نمایش داده بشنوند، باید تابع `cgaputc` را نیز ویرایش کنیم.

در این تابع `pos` موقعیت نشانگر را در صفحه تعیین می‌کند. پس با ورودی `left arrow` و `right arrow` موقعیت نشانگر در صفحه یک واحد باید به چپ یا راست تغییر کند.

```
else if (c == LEFT_ARROW)
{
    if (pos > 0)
    | --pos;
}
else if (c == RIGHT_ARROW)
{
    if (pos < 25 * 80 - 1)
    | ++pos;
}
```

البته باید حواسمن باشد که از صفحه خارج نشود.

---

کاراکتر های روی صفحه کنسول با یک آرایه به نام crt به ابعاد صفحه نمایش که برابر  $25 \times 80$  (25 سطر و 80 ستون) نمایش داده می شوند. برای آنکه بدانیم از محلی که نشانگر قرار دارد تا انتهای خط ورودی چند کاراکتر وجود دارد از کد زیر استفاده می کنیم.

```
int rst = input.e - input.c;
```

پس برای اضافه کردن یه حرف در میانه یک خط باید تمام کاراکتر هایی که در بعد از نشانگر قرار دارند یه واحد به راست بروند و کاراکتر جدید در محل نشانگر اضافه شود.

```
else
{
    memmove(crt + pos + 1, crt + pos, sizeof(crt[0]) * rst);
    crt[pos++] = (c & 0xff) | 0x0700; // black on white
}
```

برای پاک کردن یک کاراکتر با ورودی backspace نیز باید کاراکتر های سمت چپ نشانگر یک واحد به چپ شیفت پیدا کنند و کاراکتر آخر حذف شود.

```
else if (c == BACKSPACE)
{
    if (pos > 0)
    {
        memmove(crt + pos - 1, crt + pos, sizeof(crt[0]) * rst);
        crt[--pos + rst] = ' ' | 0x0700;
    }
}
```

## 2. نمایش history و بالا و پایین کردن دستورات

نمایش history و فلش بالا و پایین به صورت جداگانه پیاده سازی شده است. دستور history در فایل sh.c و فلش بالا و پایین در فایل console.c هندل شده اند.

---

ابتدا به سراغ فلش بالا و پایین می‌رویم. برای نگهداری ۱۰ دستور اخیر یک آرایه ۰۱ تایی از بافر و همچین یک متغیر برای کپی کردی بافر کنونی در این ۰۱ بافر ایجاد می‌کنیم.

```
struct
{
    char buf[INPUT_BUF];
    uint r; // Read index
    uint w; // Write index
    uint e; // Edit index
    uint c; // Cursor index
} input, copy_input, buf_history[MAX_INDEX_HISTORY];
```

سپس در تابع `consoleintr` هر زمان که دستوری برای خوانده شدن می‌رود یکی کپی از بافر اصلی را در بافر کپی نگه می‌داریم. همچنین مقدار متغیر `e` را یکی کم می‌کنید تا دیگر `\n` را در نظر نگیرد. متغیر `command_index` نشان دهنده آن است که در حال حاضر بر روی کدام دستور از تاریخچه هستیم. پس هنگامی که دستوری برای اجرا می‌رود یعنی دیگر در تاریخچه قرار نداریم و آن را `invalid` می‌کنیم.

```
if (c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF)
{
    copy_input = input;
    copy_input.e--;
    command_index = -1;
    input.w = input.e;
    input.c = input.e;
    wakeup(&input.r);
}
```

سپس چک می‌کنی که اگر دستوری از طرف کاربر برای اجرا داده شود (یعنی `enter` بزنند) اگر دستور ورودی خالی نباشد آن را در `buf_history` ذخیره می‌کنیم. برای این کار باید ۹ دستور اخیر را یه واحد شیفت دهیم و آخرین دستور وارد شده را در خانه اول `buf_history` کپی کنیم.

```
if (c == '\n' && copy_input.e - copy_input.w != 0)
{
    for (int i = 9; i > 0; i--)
    {
        buf_history[i] = buf_history[i - 1];
    }
    copy_input.c = copy_input.e;
    buf_history[0] = copy_input;
    if (num_saved_commands < MAX_INDEX_HISTORY)
        num_saved_commands++;
}
```

همچنین نشانگر را نیز به انتهای خط می‌بریم. برای آنکه در صورت کمتر بودن دستورات تاریخچه از عدد ۱۰، دیگر به فلش بالا اجازه اجرا ندهیم، برای ۱۰ دستور اول، تعداد را در num\_saved\_commands افزایش داده و بعد از آنها این عدد را بازگردانی کنیم.

حال هنگامی که down arrow یا up arrow زده می‌شود، در صورتی که command\_index در بازه‌ی معتبری باقی بماند (یعنی بین ۰ تا num\_saved\_commands) اجازه اجرا به آنها داده می‌شود. برای اجرا هم در بافر اصلی و هم در صفحه نمایش، نشانگر را به انتهای خط می‌بریم و به تعداد کاراکترهای موجود در خط backspace می‌زنیم تا خط کامل پاک شود. سپس مقدار consputc(buf\_history[command\_index]) را در بافر اصلی می‌ریزیم. در این مرحله بافر اصلی درست شده است اما برای نمایش آن، کاراکترهای آن خط را به ترتیب باتابع consputc بر روی صفحه کنسول نویسیم.

---

```
case UP_ARROW:
    if (command_index < num_saved_commands - 1)
    {
        command_index++;
        while (input.c != input.e)
        {
            input.c++;
            consputc(RIGHT_ARROW);
        }
        for (uint i = input.e; i != input.w; i--)
            consputc(BACKSPACE);
        input = buf_history[command_index];
        for (uint i = input.w; i != input.e; i++)
            if (input.buf[i % INPUT_BUF] != '\n')
                consputc(input.buf[i % INPUT_BUF]);
    }
    break;
```

```
case DOWN_ARROW:
    if (command_index > 0)
    {
        command_index--;
        while (input.c != input.e)
        {
            input.c++;
            consputc(RIGHT_ARROW);
        }
        for (uint i = input.e; i != input.w; i--)
            consputc(BACKSPACE);
        input = buf_history[command_index];
        for (uint i = input.w; i != input.e; i++)
            if (input.buf[i % INPUT_BUF] != '\n')
                consputc(input.buf[i % INPUT_BUF]);
    }
    break;
```

---

برای پیاده سازی دستور history یک فایل history.c می سازیم و آن را به makefile نیز اضافه می کنیم تا جزو دستورات قرار بگیرد. در این تابع انتظار داریم که دستور های تاریخچه که باید نمایش داده شوند به صورت string در ورودی argv تابع قرار داده شوند و در این تابع صرفا آن ها را print کنیم.

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
    for(int i = 1; i < argc; i++)
        printf(1, argv[i]);
    exit();
}
```

اما چگونه دستورات اخیر را ذخیره کنیم و به این تابع به عنوان ورودی بدهیم. برای ذخیره داده ها دوباره در فایل sh.c آرایه ای درست می کنیم تا دستورات اخیر را ذخیره کند.

```
#define HISTORY_SIZE 11          // Number of commands to store
#define CMD_LENGTH 100           // Maximum length of each command

struct {
    char buf[HISTORY_SIZE][CMD_LENGTH];
    uint count;
    uint index;
} history;
```

این دفعه نحوه ذخیره سازی در این آرایه را به صورت cycle پیاده سازی می کنیم که در آن count تعداد دستوراتی است که تا به اینجا وارد شده که اگر کمتر از ۱۰ بود فقط همان تعداد نمایش داده شوند و index اشاره به خانه ای می کند که دستور جدید باید در آن نوشته شود. نکته ای که هست طول این آرایه ۱۱ است.

---

زیرا هنگامی که دستور history وارد می‌شود آن نیز در این آرایه ذخیر خواهد شد اما در این مرحله نباید نمایش داده شود.

برای ذخیر دستورات اخیر در تابع main این فایل، در ابتدای حلقه while که دستور ورودی توسط تابع getcmd گرفته می‌شود، اگر دستور خالی نباشد (یعنی فقط enter نباشد) آن را در بافر history ذخیر کرده. اگر هم تعداد دستورات هنوز به ماکزیمم نرسیده بود history.count را یکی زیاد می‌کنیم.

```
// Read and run input commands.  
while(getcmd(buf, sizeof(buf)) >= 0){  
    if(strlen(buf)>1){  
        strcpy(history.buf[history.index++ % HISTORY_SIZE], buf);  
        if(history.count < HISTORY_SIZE)  
            history.count++;  
    }  
}
```

برای آنکه هنگام اجرای دستور history این تاریخچه‌های ذخیره شده به عنوان ورودی با تابع آن داده شود، قبل از اجرای هر دستور یعنی قبل از صدا شدن تابع exec چک می‌کنیم که آیا دستور ورودی history هست یا نه. اگر history بود مقادیر argv از ۱ تا مаксیمم ۱۰ را پوینتری به دستورات ذخیره شده قرار می‌دهیم.

```
if(strcmp(ecmd->argv[0], "history") == 0)  
    for(uint i=1; i<history.count; i++)  
        ecmd->argv[i]=history.buf[(history.index-i-1) % HISTORY_SIZE];  
  
exec(ecmd->argv[0], ecmd->argv);
```

از آنجایی که خانه اول argv همیشه اسم آن دستور است، پس ما باید خانه‌های ۱ تا مаксیمم ۱۱ را پوینتر به آرایه‌ای ذخیره کننده دستورات اخیر قرار بدیم. اما طول آرایه argv به صورت دیفالت برای ۱۰ قرار داده شده است. برای آنکه به مشکل نخوریم زیرا ما ممکن است ۱۱ خانه بخواهیم، مقدار دیفالت آن را به ۱۱ تغییر می‌دهیم.

```
#define MAXARGS 11

struct execcmd {
    int type;
    char *argv[MAXARGS];
    char *eargv[MAXARGS];
};
```

### 3. عملیات‌های copy & paste

در این بخش ابتدا به یک سری متغیر نیاز داریم تا در آن رشته کپی شده را ذخیره کنیم.

```
151
152     int copying_index = 0;
153     int is_copying = 0;
154
```

متغیر copying\_index نشان‌دهنده ایندکسی از بافر است که در آن نقطه S زده شده که در واقع نقطه شروع کپی است. یک فلگ is\_copying هم داریم که زمانی که یک است یعنی داریم کپی می‌کنیم و زمانی که صفر است چیزی کپی نمی‌شود.

```

case C('S'):
    if (is_copying == 0)
    {
        copying_index = input.c;
        is_copying = 1;
    }
    break;
case C('F'):
    if (is_copying == 1)
    {
        is_copying = 0;
        int end_index = input.c;
        for (int j = copying_index; j < end_index; j++)
        {
            if (input.buf[j % INPUT_BUF] != '\n')
            {
                for (uint i = input.e++; i != input.c; i--)
                    input.buf[i % INPUT_BUF] = input.buf[(i - 1) % INPUT_BUF];
                input.buf[input.c++ % INPUT_BUF] = input.buf[j % INPUT_BUF];
            }
            else
                input.buf[input.e++ % INPUT_BUF] = input.buf[j % INPUT_BUF];
            consputc(input.buf[j % INPUT_BUF]);
        }
    }
}

```

حال نیاز است به تابع `consoleintr` دو کیس جدید اضافه کنیم کیس اول `ctrl + S` است که اگر در حال کپی نباشیم، متغیر های کپی را سمت می‌کند یعنی `copying_index` را به ایندکس کرسر سمت می‌کند و فلگ `is_copying` را یک می‌کند.

کیس بعدی `ctrl + F` است. ابتدا فلگ `is_copying` را صفر می‌کند. سپس نقطه پایان را که همان ایندکس کرسی است که در آن نقطه `ctrl + F` زده شده را ذخیره می‌کنیم. سپس (طبق گفته TA) تمام کاراکتر های بین نقطه شروع (اینده کسی که `ctrl + S` زده شده) تا نقطه پایان (اینده کسی که `ctrl + F` زده شده) را در یک `for` به بافر اصلی اضافه می‌کنیم و در کنسول می‌نویسیم.

با توجه به این تعریف (حرف TA مربوطه در اسکایپ) اگر با عقب بردن کرسر و زدن `F` در ایندکسی قبل از نقطه شروع (اینده `ctrl + S`) هیچ کاراکتری کپی نمی‌شود.

(دقت شود که این بخش کد ما روز آخر تغییر کرد و از حالت کپی پیست لینوکس به کپی پیست گفته شده در گروه اسکایپ تغییر داده شد.)

---

## 4. عملیات‌های ریاضی

برای این بخش در تابع `consoleintr` ، بعد از هندل کردن کارکتر وارد شده یک دور بافر را به وسیله تابع `search_buf`، سرج می‌کنیم تا در صورت وجود عبارت ریاضی به فرمت گفته شده بگردیم.

```
void search_buf()
{
    for (int i = input.w; i < input.e; i++)
    {
        if (input.buf[i % INPUT_BUF] == '?')
        {
            mathexp_finder(i);
        }
    }
}
```

همانطور که مشخص است این تابع به دنبال یک علامت سوال در بافر می‌گردد و اگر پیدا شد، ایندکس مربوطه را به تابع `mathexp_finder` پاس می‌دهد.

```

void mathexp_finder(int qmark_pos)
{
    int isntr_exp = 0;
    int i = qmark_pos - 1;
    if (input.buf[i % INPUT_BUF] == '=')
    {
        i--;
        if (0 <= (int)input.buf[i % INPUT_BUF] - 48 && (int)input.buf[i % INPUT_BUF] - 48 < 10)
        {
            int flag = 0;
            while (0 <= (int)input.buf[i % INPUT_BUF] - 48 && (int)input.buf[i % INPUT_BUF] - 48 < 10)
            {
                i--;
            }
            if (i == input.e - 2)
            {
                flag = 1;
                char op = input.buf[i % INPUT_BUF];
                if (op == '%' || op == '+' || op == '-' || op == '*' || op == '/')
                {
                    i--;
                }
                else
                {
                    flag = 1;
                    int op_index = i + 1;
                    while (0 <= (int)input.buf[i % INPUT_BUF] - 48 && (int)input.buf[i % INPUT_BUF] - 48 < 10)
                    {
                        i--;
                    }
                    if (i == op_index - 1)
                    {
                        flag = 1;
                        if (flag == 1)
                        {
                            isntr_exp = 1;
                            if (isntr_exp == 0)
                            {
                                calculator(i + 1, qmark_pos, op, op_index);
                            }
                        }
                    }
                }
            }
        }
    }
}

```

در این تابع همانطور که از اسمش پیداست به دنبال اگوی ریاضی گفته شده می‌گردیم به این صورت که از علامت سوال به سمت عقب می‌آیم و ابتدا یک مساوی بعد چند رقم عدد سپس یک operator و دوباره چند رقم عدد ببیند، متوجه می‌شود که یک عبارت با الگوی گفته شده پیدا کرده و ایندکس شروع اولین عدد، ایندکس پایان عبارت (علامت سوال) و عملگر و ایندکس آن را به تابع calculator می‌دهد. این تابع یکم طولانی است برای همین در چند عکس توضیح داده می‌شود.

```

void calculator(int s_index, int end_index, char op, int op_index)
{
    int num1 = 0, num2 = 0, pos = 1, fraction = 0, int_ans = 0;
    int f_ans = 0, less_than1 = 0;
    for (int i = s_index; i < op_index; i++)
        num1 = num1 * 10 + input.buf[i % INPUT_BUF] - 48;
    for (int i = op_index + 1; i < end_index - 1; i++)
        num2 = num2 * 10 + input.buf[i % INPUT_BUF] - 48;
    switch (op)
    {
        case '%':
            f_ans = num1 % num2;
            break;
        case '-':
            f_ans = num1 - num2;
            break;
        case '+':
            f_ans = num1 + num2;
            break;
        case '/':
            f_ans = (10 * num1) / num2;
            int_ans = num1 / num2;
            break;
        case '*':
            f_ans = num1 * num2;
            break;
        default:
            break;
    }
}

```

در ابتدای این تابع یک سری متغیر را ست می‌کنیم که به ترتیب دو عدد عبارت، فلگ مثبت بودن، فلگ اعشاری بودن، جواب عدد صحیح، جواب فرکشنال (جلو تر میگم چرا `int` تعریف شده) و فلگ کمتر از یک بودن هستند. سپس در دو حلقه `for` بعدی بر اساس ایندکس‌ها، اعداد را استخراج کرده و با سوییچ کیس بعد از آن جواب عبارت در `f_ans` ریخته می‌شود. در بخش تقسیم که ممکن است پاسخ ما اعشاری باشد چون تنها به یک رقم اعشار نیاز داریم، ۱۰ برابر تقسیم هم ذخیره کردیم و اگر این مقدار با رند شده مقدار تقسیم برابر نباشد می‌فهمیم که اعشار داریم.

```

279
280     if (op == '/')
281     {
282         if (f_ans != 10 * int_ans)
283         {
284             fraction = 1;
285             int_ans = f_ans;
286             if (int_ans < 10)
287                 less_than1 = 1;
288         }
289     }
290     else
291         int_ans = f_ans;
292     if (int_ans < 0)
293     {
294         int_ans = -int_ans;
295         pos--;
296     }

```

در این تیکه هم اگر عملیات تقسیم بود چک می‌کنیم که جواب اعشاری است یا خیر اگر بود فلگ های مورد نظر را مقدار دهی می‌کنیم و رقم‌ها را به صورت عدد در int\_ans ذخیره می‌کنیم.

(اگر عملیات تقسیم بود و اعشاری بود f\_ans شامل رقم‌های مورد نیاز است اما اگر اعشار نداشته باشد خود int\_ans حاوی مقدار صحیح است. در باقی عملیات‌ها هم چون اعشار نداریم f\_ans همان int\_ans است. بعد از آن هم چک می‌کنیم که اگر جواب منفی بود فلگ متاظر را صفر کنیم و مقدار جواب را به مثبت تغییر دهیم ( تنها در عملیات‌منها ممکن است جواب منفی شود).

```

279
280     int bias = input.c - end_index - 1;
281     if (bias > 0)
282     {
283         while (input.c != end_index + 1)
284         {
285             consputc(LEFT_ARROW);
286             input.c--;
287         }
288         for (int i = input.c; i <= end_index; i++)
289         {
290             consputc(RIGHT_ARROW);
291             input.c++;
292         }
293         for (int j = end_index + 1; j > s_index; j--)
294         {
295             for (uint i = input.c--; i != input.e; i++)
296                 input.buf[(i - 1) % INPUT_BUF] = input.buf[i % INPUT_BUF];
297             input.e--;
298             consputc(BACKSPACE);
299         }
300     }

```

در این بخش کد موقعیت کرس را به بعد از علامت سوال تغییر می دهیم به این صورت که ایندکس کرس از `end_index` که همان ایندکس علامت سوال است بیشتر بود، با دادن `LEFT_ARROW` به `consputc` و کم کردن ایندکس کرس، آن را به بعد از علامت سوال می رسانیم. و اگر هم ایندکس کرس کمتر از ایندکس علامت سوال بود با دادن `RIGHT_ARROW` به `consputc` اضافه کردن ایندکس کرس، آن را به بعد از علامت سوال می رسانیم. همه این کار ها را کردیم تا عبارت را پاک کنیم برای این کار به تعداد کارکتر های عبارت ریاضی، `BACKSPACE` می زنیم.

(در صورت نیاز بافر را شیفت می دهیم، همچنین ایندکس `e` را نیز آپدیت می کنیم.)

```

234 void calculator(int s_index, int end_index, char op, int op_index)
299 {
300     if (pos == 0)
301     {
302         for (uint i = input.e++; i != input.c; i--)
303             input.buf[i % INPUT_BUF] = input.buf[(i - 1) % INPUT_BUF];
304         input.buf[input.c++ % INPUT_BUF] = '-';
305         consputc('-');
306     }
307     int num_digits = 0;
308     do
309     {
310         num_digits++;
311         int a = int_ans % 10 + 48;
312         int_ans /= 10;
313         for (uint i = input.e++; i != input.c; i--)
314             input.buf[i % INPUT_BUF] = input.buf[(i - 1) % INPUT_BUF];
315         if (fraction == 1 && num_digits > 1)
316         {
317             fraction = 0;
318             int_ans *= 10;
319             int_ans += a - 48;
320             input.buf[input.c++ % INPUT_BUF] = (int) '.';
321         }
322         else if (less_than1 == 1 && int_ans == 0 && a == (int) '0')
323         {
324             less_than1 = 0;
325             input.buf[input.c++ % INPUT_BUF] = (int) '0';
326         }
327         else
328             input.buf[input.c++ % INPUT_BUF] = a;
329     } while (int_ans != 0 || less_than1);
330 }
```

بعد از تمام این کار ها نوبت به ریختن جواب در بافر است. باید توجه داشت که بعد از پاک کردن عبارت ریاضی کرس روی ایندکس شروع عبارت است و دقیقاً جایی است که باید شروع به نوشتن کند. حال اگر جواب منفی بود ابتدا یک علامت منفی در بافر می اندازیم و آن را در کنسول نمایش می دهیم. سپس از راست عدد رقم به بافر اضافه می کنیم به طوری که عملاً در بافر عدد

---

جواب به صورت بر عکس قرار می گیرد (ابتدا رقم یکان بعد دهگان و ...). همچنین اگر عدد ما اعشاری باشد، '! را در بافر قرار می دهیم یا اگر اعشاری کمتر از 1 باشد، صفر را دستی در بافر قرار می دهیم. تعداد دفعات تکرار این حلقه را در num\_digits ذخیره می کنیم تا بعدا از آن برای درست کردن جای ارقام استفاده کنیم.

```
for (int i = 0; i < num_digits / 2; i++)
{
    int temp = input.buf[(input.c - 1 - i) % INPUT_BUF];
    input.buf[(input.c - 1 - i) % INPUT_BUF] = input.buf[(input.c - (num_digits - i)) % INPUT_BUF];
    input.buf[(input.c - (num_digits - i)) % INPUT_BUF] = temp;
}
if (pos == 0)
| s_index++;
for (int i = 0; i < num_digits; i++)
{
    consputc(input.buf[(i + s_index) % INPUT_BUF]);
}
while (bias > 0)
{
    consputc(RIGHT_ARROW);
    input.c++;
    bias--;
}
}
```

حال که اعداد به صورت بر عکس در بافر ذخیره شده اند، کافی است عدد را به اصطلاح mirror کنیم یعنی رقم آم از آخر را با رقم آم از اول جا به جا کنیم. بعد از این اگر عدد منفی بود، قبل ایک علامت منفی در ایندکس شروع قرار دادیم پس ایندکس شروع جدید که در آن رقم ها ذخیره شده اند یکی بعد از ایندکس قدیمی است. سپس از جایی که رقم های جواب شروع می شوند تا زمانی که تمام رقم ها نوشته شوند، رقم به رقم را به consputc پاس می دهیم تا روی کنسول نمایش داده شوند.

در آخر هم اگر کرسر ما از اول بعد از علامت سوال بود و ما آن را جا به جا کردیم، به مقدار بایاس آن را به جلو می بردیم و RIGHT\_ARROW می زنیم تا ایندکس کرسر و نمایش کرسر در کنسول با هم یکی باشند.

دقت شود که ما در این بخش از کد اعداد چند رقمی به عنوان operand را نیز هندل کردیم همچنین عملیات باقیمانده را نیز پشتیبانی می کند.

## • برنامه سطح کاربر

در این بخش با توجه به شماره دانشجویی‌های اعضای تیم key به صورت زیر محاسبه می‌شود.

$$\text{Key} = (75 + 40 + 1) \bmod 26 = 12$$

سیس برای هر دستور یک فایل کد داریم که عکس و توضیح آن ها در زیر آمده است.

کد دستور encode، اکه به صورت زیر است.

کد به این صورت است که علاوه بر هدرهای `types.h` (که شامل تعریف انواع داده‌های پایه ساختاری است)، `stat.h` (که اطلاعات مربوط به فایل را شامل می‌شود) و `user.h` (که شامل یک سری توابع مثلاً برای باز کردن و بستن فایل و غیره است)، `fcntl.h` هم داریم که شامل تعریف فلگ‌های مرتبط با عملیات فایل است.

سپس دو آرایه از حروف بزرگ و کوچک انگلیسی درست کردیم تا بتوانیم برای هر حرف انکود شده آن را بیابیم. تابع `encode` یک رشته می‌گیرد، آن را انکود می‌کند و رشته انکود شده را برمی‌گرداند. با توجه به کلید تعریف شده، برای هر کاراکتر رشته، اگر عضو کاراکتر های کوچک بود از آرایه الفبای کوچک و اگر کاراکتر بزرگ بود از آرایه الفبای بزرگ، ایندکس آن را به اندازه `key` شیفت می‌دهیم و باقیمانده بر 26 می‌گیریم تا کاراکتر معادل انکود شده را بدهد. دقت شود که اگر کاراکتری از حروف انگلیسی نبود، همان باقی می‌ماند.

```

22
23 int main(int argc, char *argv[])
24 {
25     if (argc <= 1)
26     {
27         printf(1, "There is no text\n");
28         exit();
29     }
30     int fd = 0;
31     if ((fd = open("result.txt", O_CREATE | O_RDWR)) < 0)
32         printf(1, "Couldn't open the file");
33     for (int i = 1; i < argc; i++)
34     {
35         char *c = encode(argv[i]);
36         write(fd, c, strlen(argv[i]));
37         if (i != argc - 1)
38             write(fd, " ", 1);
39         else
40             write(fd, "\n", 1);
41     }
42     close(fd);
43     exit();
44 }
```

در تابع `main` این فایل هم ابتدا چک می‌شود که یک تکستی جلوی دستور `encode` آمده باشد. سپس یک فایل به اسم `result.txt` باز می‌کنیم. فلگ های `O_CREATE` و `O_RDWR` برای این است که فایل را در صورت عدم وجود بسازد و یا در صورت وجود باز کند. سپس با استفاده از `argc` و `argv` متن جلوی دستور `encode` را کلمه به کلمه به تابع `encode` می‌دهیم و رشته کد گذاری شده را در فایل می‌نویسیم. و بعد آن بر اساس اینکه به کلمه آخر رسیدیم یا خیر، اسپیس یا `\n` می‌نویسیم. در آخر هم فایل را می‌بندیم.

دقت شود که در دستور `decode` هم تابع `main` به همین صورت است.

```
char *decode(char *c)
{
    // key = (40 + 75 + 1) mod 26 = 12
    int key = 12;
    for (int i = 0; i < strlen(c); i++)
        if ((int)c[i] >= (int)'a' && (int)c[i] <= (int)'z')
            c[i] = lc_alphabet[((int)c[i] - (int)'a' + 26 - key) % 26];
        else if ((int)c[i] >= (int)'A' && (int)c[i] <= (int)'Z')
            c[i] = uc_alphabet[((int)c[i] - (int)'A' + 26 - key) % 26];
    return c;
}
```

تنها فرق تابع decode با encode در نحوه شیفت دادن است که در تابع encode ایندکس کاراکتر با مقدار key جمع می‌شود اما در تابع decode همانطور که مشاهده می‌کنید مقدار key کم می‌شود تا عملای تاثیر encode خنثی شود. از طرفی چون عملگر باقیمانده روی اعداد منفی کار نمی‌کند. دستی یک  $+26$  اضافه کردیم تا مقدار داخل پرانتز منفی نشود.

حال برای این که دستورات ما بعد از اینکه در کنسول نوشته می‌شوند اجرا شوند، آن‌ها را به اضافه می‌کنیم.

```
167
168     UPROGS=\
169     _cat\
170     _echo\
171     _forktest\
172     _grep\
173     _init\
174     _kill\
175     _ln\
176     _ls\
177     _mkdir\
178     _rm\
179     _sh\
180     _stressfs\
181     _usertests\
182     _wc\
183     _zombie\
184     _history\
185     _encode\
186     _decode\
187
```

```
253     EXTRA=\
254     mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
255     ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c history.c encode.c decode.c\
256     printf.c umalloc.c\
257     README dot-bochssrc *.pl toc.* runoff runoff1 runoff.list\
258     .gdbinit.tmpl gdbutil\
```

که همانطور که دیده می‌شود اسم دو دستور `encode` و `decode` را در لیست `UPROGS` ذخیره می‌شود که در واقع لیستی از برنامه‌هایی که در فضای کاربر ران می‌شود است. علامت `_` نشان‌دهنده فایل میانی است و با زدن دستور در واقع کد واقع در آن فایل اجرا می‌شود. فایل کد‌ها نیز در لیست `EXTRA` اضافه می‌کنیم. یک نمونه از خروجی این کد به صورت زیر است.

```
$ encode This is a simple text.
$ cat result.txt
Ftue ue m euybxq fqjf .
$ decode Ftue ue m euybxq fqjf .
$ cat result.txt
This is a simple text.
$
```

## ● مقدمه‌ای درباره سیستم عامل و xv6

### سوال 1) سه وظیفه اصلی سیستم عامل را نام ببرید.

1. مدیریت و ساده سازی دسترسی به سخت افزار: سیستم عامل وظیفه دارد که سخت افزار کامپیوتر را کنترل و مدیریت کند و جزئیات فنی پیچیده آن را از دید برنامه ها پنهان سازد. به این ترتیب، برنامه ها بدون نیاز به درک مستقیم از سخت افزار، می توانند از منابع مختلف سیستم استفاده کنند.
2. اشتراک گذاری منابع بین برنامه ها: سیستم عامل منابع سخت افزاری مانند CPU، حافظه و دستگاه های ورودی/خروجی را به طور منظم بین برنامه های مختلف به اشتراک می گذارد، تا به نظر برسد که این برنامه ها به صورت همزمان اجرا می شوند، حتی اگر در واقعیت اینگونه نباشد.
3. فراهم کردن بستری امن برای ارتباط و همکاری بین برنامه ها: سیستم عامل راه های ایمن و کنترل شده ای را برای تعامل و تبادل داده میان برنامه های مختلف ارائه می دهد تا بتوانند به صورت مؤثر با هم کار کنند، بدون اینکه امنیت و ثبات سیستم به خطر بیافتد.

### سوال 2) فایل های اصلی سیستم عامل xv6 در صفحه یک کتاب xv6 لیست شده اند. به طور مختصر هر گروه را توضیح دهید. نام پوشه اصلی فایل های هسته سیستم عامل، فایل های سرایнд و فایل سیستم در سیستم عامل لینوکس چیست؟ در مورد محتویات آن مختصرا توضیح دهید.

#### 1. Basic Headers (فایل های سرایнд اصلی)

فایل های سرایнд این دسته شامل تعریفات پایه ای و اساسی سیستم عامل هستند. آن ها ساختار های داده، انواع داده های سفارشی، پارامتر های اصلی سیستم (مانند تعداد ماکسیمم فرآیندها)، ساختار چیدمان حافظه و آدرس قسمت های مختلف در حافظه و توابع مهمی را تعریف می کنند که در بخش های مختلف سیستم عامل استفاده می شوند. همچنین شامل تعریفات مرتبط با معماری x86 و اطلاعات مربوط به واحد مدیریت حافظه (MMU) و فرمت اجرایی فایل های باینری (ELF) و ساختار های مربوط به تاریخ و زمان سیستم می شوند. این فایل ها پایه ای برای ارتباط سیستم عامل با سخت افزار و نرم افزار هستند.

#### 2. Entering xv6 (ورود به xv6)

این دسته از فایل‌ها مربوط به راهاندازی اوپریم سیستم‌عامل است. فایل‌های اس‌بی‌الی و C در این دسته وظیفه دارند که پس از روشن شدن سیستم، پردازنده را آماده‌سازی کنند و سپس هسته سیستم‌عامل را به اجرا درآورند. این مرحله شامل تنظیمات اوپریم CPU، آماده‌سازی حافظه، و شروع اجرای فرآیندهای پایه‌ای هسته است. در سیستم‌های چندپردازنده، فایل‌های مخصوص به راهاندازی پردازنده‌های اضافی نیز در این دسته قرار دارند.

### 3. Locks (قفل‌ها)

فایل‌های این دسته به منظور همگام‌سازی (synchronization) بین بخش‌های مختلف سیستم‌عامل طراحی شده‌اند. قفل‌های چرخشی (spinlocks) و قفل‌های خواب (sleeplocks) به عنوان ابزارهای اصلی برای جلوگیری از دسترسی همزمان به منابع مشترک استفاده می‌شوند. این فایل‌ها ساختارهای داده و توابعی را فراهم می‌کنند که به هسته اجازه می‌دهد تا فرآیندهای چندگانه را به صورت همزمان اما کنترل شده مدیریت کند.

### 4. Processes (فرآیندها)

این دسته از فایل‌ها مسئول مدیریت فرآیندها در سیستم‌عامل هستند. فایل‌های این دسته شامل تعریف و پیاده‌سازی ساختار فرآیندها، تخصیص و آزادسازی حافظه برای آن‌ها، و همچنین مدیریت وضعیت فرآیندها و سوییچ کردن بین فرآیندهای مختلف است. همچنین، این دسته شامل کدهای مربوط به زمان‌بندی (scheduling) فرآیندها و تعاملات آن‌ها با سیستم‌عامل است.

### 5. System Calls (فراخوانی‌های سیستمی)

این دسته شامل فایل‌هایی است که پیاده‌سازی فراخوانی‌های سیستمی را بر عهده دارند. سیستم‌عامل از طریق این فراخوانی‌ها خدماتی مانند ایجاد فرآیندها، مدیریت حافظه، ارتباطات بین فرآیندها و مدیریت فایل‌ها را در اختیار برنامه‌های کاربری قرار می‌دهد. همچنین شامل مدیریت وقفه‌ها و استثنایات (interrupts & exceptions) و پیاده‌سازی توابعی است که به برنامه‌های سطح کاربر اجازه می‌دهد تا از خدمات هسته استفاده کنند.

### 6. File System (سیستم فایل)

این فایل‌ها مربوط به پیاده‌سازی سیستم فایل در xv6 هستند. سیستم فایل شامل مدیریت دیسک، فایل‌ها و دایرکتوری‌ها، بافرهای دیسک، و sleeplock برای کنترل همزمانی در عملیات فایل است. علاوه بر این،

---

این دسته شامل پیاده‌سازی عملیات خواندن و نوشتن فایل‌ها، مدیریت فایل‌ها در سطح هسته و اجرای برنامه‌ها از طریق فایل‌های اجرایی است.

#### 7. Pipes (لوله‌ها)

این دسته شامل پیاده‌سازی مکانیزم لوله‌ها (pipes) است که یک روش ارتباط بین فرآیندها (IPC) است. لوله‌ها به فرآیندها این امکان را می‌دهند که به طور همزمان از طریق یک جریان داده با هم ارتباط برقرار کنند. این فایل‌ها ارتباط داده‌ای بین فرآیندها را در سیستم مدیریت می‌کنند.

#### 8. String Operations (عملیات‌های رشته‌ای)

این فایل‌ها شامل توابعی برای مدیریت و پردازش رشته‌ها (strings) هستند. این توابع شامل عملیات‌هایی مانند کپی کردن، مقایسه، و محاسبه طول رشته‌ها می‌شوند و در بخش‌های مختلف سیستم عامل استفاده می‌شوند.

#### 9. Low-Level Hardware (سخت‌افزار سطح پایین)

این دسته شامل فایل‌هایی است که به سیستم اجازه می‌دهند تا با سخت‌افزار سطح پایین، مانند پردازنده‌های چندگانه و کنترل‌کننده‌های وقفه (APIC)، تعامل کند. این فایل‌ها مسئول پیاده‌سازی تعامل مستقیم با سخت‌افزار و مدیریت وقفه‌ها و منابع سخت‌افزاری هستند که برای عملکرد صحیح سیستم حیاتی‌اند.

#### 10. User-Level (سطح کاربر)

این دسته شامل فایل‌هایی است که به تعامل بین هسته و برنامه‌های سطح کاربر مرتبط است. این فایل‌ها شامل پیاده‌سازی فراخوانی‌های سیستمی سطح کاربر، اجرای شل ساده (که دستورات کاربر را دریافت و اجرا می‌کند)، و کدهای اولیه‌ی سطح کاربر برای شروع فرآیندهای جدید است.

#### 11. Bootloader (بوت‌لودر)

فایل‌های بوتلودر مسئول بارگذاری هسته سیستم عامل از دیسک به حافظه و آغاز اجرای آن هستند. این فایل‌ها شامل کدهای اسمبلی و C برای اجرای فرآیندهای اولیه‌ی بوتلودر و شروع به کار سیستم عامل بعد از روشن شدن سیستم می‌شوند.

#### 12. Link (لينكدهي)

---

این فایل شامل اسکریپت لینکدهی است که ترتیب و نحوه چیدمان قسمت‌های مختلف حافظه در فایل اجرایی هسته را تعیین می‌کند. این فایل مشخص می‌کند که هر بخش از کد و داده‌ها در چه آدرسی قرار گیرد.

#### پوشش‌های مشابه در لینوکس:

##### 1. فایل‌های هسته سیستم‌عامل (Kernel Core)

- مسیر: /usr/src/linux/kernel

- محتویات: این بخش شامل کدهای اصلی مدیریت هسته سیستم‌عامل است. اعمال مهمی مانند مدیریت فرآیندها، زمان‌بندی، مدیریت فرآخوانی‌های سیستمی، و مدیریت حافظه در این دایرکتوری پیاده‌سازی شده‌اند. فایل‌های موجود در این مسیر به نحوه عملکرد کلی هسته و تعاملات آن با سخت‌افزار سیستم و برنامه‌های کاربر مرتب می‌شوند.

##### 2. فایل‌های سرایнд (Header Files)

- مسیر: /usr/src/linux/include/

- محتویات: این دایرکتوری شامل فایل‌های سرایнд است که تعریفات ساختار‌های داده، ثابت‌ها و پروتوتایپ توابع مورد استفاده در هسته لینوکس را فراهم می‌کنند. این فایل‌ها به توسعه‌دهنگان هسته اجازه می‌دهند تا کدهای خود را مطابق با استانداردهای سیستم‌عامل بنویسند. همچنین این پوشش شامل فایل‌های مربوط به معمازی‌های مختلف برای CPU‌های مختلف است که از طریق زیرشاخه‌های خاص در دسترس هستند.

##### 3. فایل‌های سیستم‌فایل (File System)

- مسیر: /usr/src/linux/fs/

- محتویات: این دایرکتوری شامل کدهای مرتبط با سیستم‌فایل لینوکس است. در این دایرکتوری توابعی برای مدیریت فایل‌ها (مانند باز کردن، خواندن، نوشتن و بستن فایل‌ها) و داده‌ساختار‌های ضروری برای مدیریت فایل‌ها تعریف شده‌اند. همچنین سیستم‌های فایل مختلف (مانند NFS، ext4، و FAT) در این مسیر پیاده‌سازی شده‌اند.

## ● کامپایل سیستم عامل xv6

سوال 3) دستور `n make` را اجرا نمایید. کدام دستور، فایل نهایی هسته را می‌سازد؟

دستور `n make` در سیستم‌های یونیکس و لینوکس، دستوری است که اجازه می‌دهد تا بررسی کنید که چه فرمان‌هایی قرار است توسط `make` اجرا شوند، بدون آنکه واقعاً آنها را اجرا کند. پس ما با زدن این دستور و خواندن `makefile` به این نتیجه رسیدیم که این بخش مسئول ساختن فایل نهایی هسته است:

```
ld -m elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o laptic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.o string.o swtch.o syscall.o sysfile.o sysproc.o trapasm.o trap.o uart.o vectors.o vm.o -b binary initcode entryother
```

این دستور هسته‌ی سیستم عامل (`kernel`) را می‌سازد زیرا از ابزار `ld` برای لینک کردن تمام فایل‌های ۰. مرربوط به اجزای هسته (که قبل از طریق کامپایلر به صورت جداگانه کامپایل شده‌اند) استفاده می‌کند (فایل‌های ۰. شامل کدهای مختلف سیستم عامل هستند که همانطور که گفتیم با استفاده از اسکریپت `kernel.ld` ترکیب می‌شوند). خروجی این فرآیند، هسته‌ی نهایی است که توسط بوت‌لودر در زمان بوت سیستم بارگذاری می‌شود.

سوال 4) در `Makefile` متغیرهایی به نام‌های `UPROGS` و `ULIB` تعریف شده است. کاربرد آنها چیست؟

UPROGS: این متغیر شامل لیستی از برنامه‌های سطح کاربر (user programs) است که در محیط کاربر اجرا می‌شوند که در فرآیند بیلد سیستم عامل باید کامپایل شده و در سیستم فایل (`fs.img`) قرار گیرند تا در زمان اجرای سیستم عامل قابل دسترسی باشند. در فایل `Makefile`، این برنامه‌ها معمولاً به صورت زیر تعریف شده‌اند:

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _history\
    _encode\
    _decode\
```

هر یک از این برنامه‌ها، به صورت یک فایل اجرایی در فضای کاربر (مانند sh) مانند ایجاد می‌شود. علامت \_ در ابتداء، نشان‌دهندهٔ فایل‌های میانی است که در نهایت به نام اصلی خود (بدون \_) تغییر نام می‌دهند.

ULIB: این متغیر به کتابخانه‌های سطح کاربر (user libraries) اشاره دارد که برای لینک کردن برنامه‌های کاربری از آن‌ها استفاده می‌شود. در Makefile، کتابخانه‌های کاربری مانند ulib.o, usys.o و سایر فایل‌های مورد نیاز برای اجرای برنامه‌های کاربر در این متغیر تعریف شده‌اند که بعداً توسط ld به فایل‌ها اضافه شده و در کدهای کرنل از این کتابخانه‌ها استفاده می‌شود:

```
ULIB = ulib.o usys.o printf.o umalloc.o
```

پس این object file‌ها شامل توابع کتابخانه‌ای و system call‌هایی هستند که برنامه‌های کاربر به آن‌ها نیاز دارند تا بتوانند با هسته تعامل داشته باشند.

## ● اجرا بر روی شبیه‌ساز QEMU

سوال 5) دستور `make qemu -n` را اجرا نمایید. دو دیسک به عنوان ورودی به شبیه‌ساز داده شده است. محتوای آنها چیست؟ (راهنمایی: این دیسک‌ها حاوی سه خروجی اصلی فرایند بیلد هستند.)

دستور `make qemu -n` به صورت شبیه‌سازی فقط نشان می‌دهد که اگر بخواهید برنامه را در QEMU اجرا کنید، چه مراحلی طی خواهد شد، اما هیچ تغییری اعمال نمی‌شود و برنامه واقعاً اجرا نخواهد شد.

برای شبیه‌سازی سیستم عامل xv6، دو دیسک اصلی به شبیه‌ساز داده می‌شود:

```
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img  
,index=0,media=disk,format=raw -smp 2 -m 512
```

1. **xv6.img**: شامل بوت‌لودر (bootloader) که برای بوت اولیه است و فایل‌های اصلی سیستم عامل که برای بارگذاری سیستم عامل استفاده می‌شوند.

2. **fs.img**: این دیسک حاوی **file system** مجازی است که تمامی فایل‌ها و داده‌های ذخیره شده روی دیسک را در بر می‌گیرد. پس شامل فایل‌ها و داده‌هایی است که سیستم عامل برای عملکرد خود به آن‌ها نیاز دارد. همچنین برنامه‌های جانبی و user programs مثل `cat` و `echo` و `grep` و داده‌های کاربر نیز می‌توانند در این دیسک ذخیره شوند.

این فایل سیستم توسط ابزار `mkfs` ساخته شده است که فایل‌هایی مانند `README` و برنامه‌های کامپایل شده را بر روی این دیسک قرار می‌دهد.

```
./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests  
_wc _zombie _history _encode _decode
```

نهایتاً هم سه خروجی اصلی فرایند بیلد را می‌توان `bootblock` که بوت لودر اولیه است و کرنل سیستم عامل و فایل سیستم که حاوی اطلاعات فایل‌های سیستم و برنامه‌های کاربر است دانست.

## ● اجرای بوت‌لودر

سوال 8) علت استفاده از دستور **objcopy** در حین اجرای عملیات **make** چیست؟

دستور objcopy در جریان عملیات make برای تبدیل فرمت فایل‌های O. يا اجرایی و حذف اطلاعات اضافی مانند هدرها استفاده می‌شود. این کار باعث بهینه‌سازی و کاهش حجم فایل و ایجاد نسخه‌های باینری خام (raw binary) مناسب برای شبیه‌سازها می‌شود. پس objcopy فایل‌ها را برای سازگاری با محیط‌های خاص آماده می‌کند.

- حذف اطلاعات غیرضروری: با استفاده از فلگ -S، بخش‌های غیرضروری (مثل اطلاعات سمبول‌ها) از فایل حذف می‌شوند تا حجم فایل نهایی کمتر شود.
- تبدیل به قالب باینری خام: با استفاده از فلگ -O binary، فایل O. يا اجرایی به یک فایل باینری خام تبدیل می‌شود. این فایل‌ها برای بارگذاری مستقیم در حافظه و اجرای سیستم‌عامل یا بوت‌لودر ضروری هستند.

در makefile ما هم دستور objcopy برای تبدیل فایل‌های O. به فایل‌های باینری خام استفاده شده است.

در اولین مورد، برای ساخت فایل bootblock از فایل bootblock.o، دستور objcopy با فلگ‌های S- و -O binary و -j. text استفاده شده است. فلگ -S باعث حذف اطلاعات اشکال‌زدایی می‌شود، فلگ -O binary خروجی را به فرمت باینری خام تبدیل می‌کند، و فلگ -j. text مشخص می‌کند که فقط بخش text. که شامل کد اجرایی است، استخراج شود.

```
OBJCOPY) -S -O binary -j .text bootblock.o bootblock) $
```

در مورد دوم، برای فایل entryother.o از فایل bootblockother دقيقاً همان دستورات و فلگ‌ها استفاده شده‌اند و هدف مشابه است: تبدیل یک فایل O. به باینری خام که فقط کد اجرایی را شامل شود.

```
OBJCOPY) -S -O binary -j .text bootblockother.o) $  
entryother
```

در مورد سوم، برای فایل initcode.out، از فایل initcode استفاده می‌شود. این بار دستور objcopy با فلگ‌های -S و -O binary اجرا می‌شود، اما فلگ j - به کار نمی‌رود، بنابراین کل محتوای فایل شیء به باینری خام تبدیل می‌شود.

```
OBJCOPY) -S -O binary initcode.out initcode) $
```

**سوال 13) کد bootmain.c هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در آدرس 0x100000 قرار می‌دهد. علت انتخاب این آدرس چیست؟**

آدرس‌های پایین‌تر از 1 مگابایت (مثل 0x7C00 که بوتلودر در آن بارگذاری می‌شود) به دلایل مختلف برای بوتلودر و داده‌های سیستم عامل رزرو شده‌اند. این بخش شامل BIOS و سایر روتین‌های ضروری است که برای راهاندازی سیستم نیاز هستند. بارگذاری هسته در آدرس 0x100000 از تداخل با این بخش‌ها جلوگیری می‌کند.

پس استفاده از آدرس 0x100000 به عنوان نقطه‌ی شروع برای بارگذاری هسته، به این دلیل است که این مقدار بعد از محدوده‌ای قرار دارد که توسط حافظه دستگاه‌های ورودی/خروجی و بخش‌های دیگر استفاده می‌شود، و در عین حال پایین‌تر از محدوده‌های بالاتر است که ممکن است روی برخی سیستم‌های کوچک حافظه‌ی فیزیکی در دسترس نباشد. به عبارت دیگر، انتخاب این آدرس یک نقطه مناسب برای اجرای هسته است که از تداخل با دستگاه‌های I/O جلوگیری می‌کند و در عین حال در دسترس است.

## • اجرای هسته xv6

**سوال 18) علاوه بر صفحه‌بندی در حد ابتدایی از قطعه‌بندی به منظور حفاظت هسته استفاده خواهد شد.**  
این عملیات توسط seginit() انجام می‌گردد. همانطور که ذکر شد، ترجمه قطعه تأثیری بر ترجمه آدرس منطقی نمی‌گذارد. زیرا تمامی قطعه‌ها اعم از کد و داده روی یکدیگر می‌افتد. با این حال برای کد و داده‌های سطح کاربر پرچم SEG\_USER تنظیم شده است. چرا؟ (راهنمایی: علت مربوط به ماهیت دستور العمل‌ها و نه آدرس است).

- در سیستم عامل **xv6**، برای هر فرآیند سطح کاربر از پرچم **SEG\_USER** در بخش‌هایی از کد استفاده می‌شود تا تفاوت‌های سطح دسترسی مشخص شوند. این پرچم برای کد و داده‌های سطح کاربر تنظیم می‌شود، به این دلیل که فرآیندها در حالت کاربر نباید بتوانند به بخش‌های حساس کرنل دسترسی داشته باشند. اما دسترسی به حافظه و منابع اختصاصی کاربر برای اجرای صحیح برنامه‌های کاربر ضروری است.
- علت تنظیم پرچم **SEG\_USER** برای قطعات (**Segments**) مربوط به کد و داده‌های کاربر مربوط به ماهیت دستورالعمل‌ها است. در معماری **x86**، دستورالعمل‌های اجرا شده توسط پردازش‌های کاربر نباید حق دسترسی به حافظه و منابع کرنل را داشته باشند، چراکه این دستورالعمل‌ها در سطح امنیتی پایین‌تری (**CPL=3** Current Privilege Level) نسبت به کرنل (**CPL=0**) اجرا می‌شوند. اگر این پرچم برای کدهای سطح کاربر تنظیم نشود، ممکن است دستورالعمل‌های کاربر به اشتباه اجازه اجرای دستورات حساس کرنل را داشته باشند، که این امر خطرناک است و می‌تواند باعث ایجاد آسیب‌پذیری‌های امنیتی شود.
- بنابراین، ترجمه قطعات به خودی خود بر آدرس‌دهی تأثیری نمی‌گذارد، اما استفاده از این پرچم برای اطمینان از اجرای صحیح دستورات و جلوگیری از دسترسی‌های غیرمجاز ضروری است.

## • اجرای نخستین برنامه سطح کاربر

سوال 19) جهت نگهداری اطلاعات مدیریتی برنامه‌های سطح کاربر ساختاری تحت عنوان **struct proc** (خط ۲۳۳۶) ارائه شده است. اجزای آن را توضیح داده و ساختار معادل آن در سیستم عامل لینوکس را بیابید.

1. **: (state)Process State**

این فیلد مشخص می‌کند که فرآیند در چه وضعیتی است، مثل تخصیص داده شده (**allocated**)، آماده اجرا (**ready to run**), در حال اجرا (**running**), در حال انتظار برای I/O (**waiting for I/O**) یا در حال اتمام (**exiting**) باشد. این حالت‌ها به کرنل می‌گوید که با فرآیند چه کار کند.

---

## :**(pgdir)Page Directory .2**

این فیلد به Page Table فرآیند اشاره می‌کند که آدرس‌های مجازی فرآیند را به آدرس‌های فیزیکی در حافظه ترجمه می‌کند. این ساختار کمک می‌کند که هر فرآیند فضای حافظه مختص خود را داشته باشد و از دسترسی به حافظه سایر فرآیندها جلوگیری شود.

## :**(kstack) Kernel Stack .3**

هر فرآیند یک Kernel Stack دارد که در زمان اجرای فرآیند در حالت کرنل استفاده می‌شود، مثلاً وقتی فرآیند یک System Call انجام می‌دهد یا وقمهای رخ می‌دهد. این پشته برای ذخیره اطلاعات موقت کرنل استفاده می‌شود.

## :**(context)CPU Context .4**

این فیلد وضعیت CPU (مثل ثبات‌ها و آدرس بازگشت) را نگهداری می‌کند تا وقتی که کرنل بخواهد بین فرآیندها Context Switch کند، از این اطلاعات برای بازیابی فرآیند استفاده کند.

## :**(name)Process Name .5**

این فیلد یک رشته است که نام فرآیند را نگهداری می‌کند، که بیشتر برای Debugging مفید است.

## :**(pid) Process ID .6**

این فیلد شناسه منحصر به فرد فرآیند است. هر فرآیند یک PID دارد که کرنل از آن برای شناسایی فرآیندها استفاده می‌کند. مثل شماره شناسنامه برای هر فرآیند عمل می‌کند.

## :**(tf) Trap Frame .7**

زمانی که یک وقه (Interrupt) یا System Call رخ می‌دهد، وضعیت ثبات‌های CPU در Trap Frame ذخیره می‌شود. این کمک می‌کند که فرآیند بعد از بازگشت از وقه به درستی ادامه پیدا کند.

معادل struct proc در لینوکس:

در لینوکس، ساختاری به نام task\_struct وجود دارد که اطلاعات مشابهی را نگهداری می‌کند:

.1. state: وضعیت فرآیند (Running، Sleeping و ...).

.2. mm\_struct: مدیریت فضای حافظه (معادل pgdir در xv6).

.3. stack: اشاره‌گر به Kernel Stack فرآیند.

.4. pid: شناسه فرآیند، مشابه PID در xv6.

.5. sched\_info: اطلاعات مربوط به Scheduling و زمان‌بندی.

لینوکس دارای یک ساختار پیچیده‌تر است زیرا باید قابلیت‌هایی مثل Advanced Multithreading را پشتیبانی کند. اما در کل این شباهت‌ها نشان‌دهنده اصول طراحی مشترک سیستم‌عامل‌ها در نگهداری و مدیریت فرآیندها هستند.

سوال 23) کدام بخش از آماده‌سازی سیستم، بین تمامی هسته‌های پردازنده مشترک و کدام بخش اختصاصی است؟ (از هر کدام یک مورد را با ذکر دلیل توضیح دهید). زمان‌بند روی کدام هسته اجرا می‌شود؟

بخش‌های مشترک در آماده‌سازی سیستم برای تمامی هسته‌های پردازنده:

- جدول صفحات (Page Table): جدول صفحات مسئول مدیریت حافظه و نگاشت آدرس‌های مجازی به آدرس‌های فیزیکی است. تمامی هسته‌های پردازنده از یک جدول صفحه‌ی مشترک برای دسترسی به حافظه استفاده می‌کنند. این باعث می‌شود تا تمامی هسته‌ها به داده‌های یکسانی در حافظه دسترسی داشته باشند و سازگاری بین پردازش‌ها حفظ شود.

- سیستم ورودی/خروجی: مدیریت دستگاه‌های ورودی و خروجی نیز بین تمامی هسته‌ها به اشتراک گذاشته می‌شود. دستگاه‌هایی مثل دیسک و شبکه نیاز دارند که توسط تمامی هسته‌ها به اشتراک گذاشته شوند، زیرا هر هسته ممکن است بخواهد داده‌ای را از این دستگاه‌ها بخواند یا روی آن‌ها بنویسد.

---

- قفل‌های سیستم (Locks): برای جلوگیری از دسترسی همزمان و نامطلوب به داده‌های مشترک بین هسته‌ها، سیستم عامل از قفل‌هایی استفاده می‌کند. این قفل‌ها تضمین می‌کنند که فقط یک هسته در هر لحظه به داده‌های مشترک دسترسی داشته باشد.

### بخش‌های اختصاصی هر هسته:

- مدیریت وقفه‌ها (Interrupt Handling): هر هسته‌ی پردازنده مسئول مدیریت وقفه‌های خود است. برای مثال، کنترل‌کننده محلی وقفه (Local APIC) در هر هسته به صورت مستقل تنظیم می‌شود تا وقفه‌های مربوط به همان هسته را مدیریت کند. این وقفه‌ها می‌توانند شامل وقفه‌های زمان‌سنج یا رویدادهای خاصی باشند که فقط برای آن هسته اتفاق می‌افتد.
- پشتۀ کرنل (Kernel Stack): هر هسته یک پشتۀ اختصاصی دارد که برای اجرای عملیات سیستم عامل استفاده می‌شود. پشتۀ کرنل برای ذخیره‌ی وضعیت اجرایی هسته و فرآیندهای در حال اجرا استفاده می‌شود.

### زمان‌بند (Scheduler):

زمان‌بند، که مسئولیت مدیریت اجرای فرآیندها را بر عهده دارد، به صورت مستقل روی هر هسته اجرا می‌شود. هر هسته دارای یک حلقه زمان‌بندی اختصاصی است که فرآیندهای قابل اجرا (RUNNABLE) را از لیست فرآیندها انتخاب کرده و اجرا می‌کند. این باعث می‌شود که هر هسته بتواند به‌طور مستقل فرآیندها را مدیریت کند و نیازی به زمان‌بندی مرکزی نباشد و فرآیندهای سیستم به صورت متعادل و همزمان بین هسته‌ها توزیع شوند و بار کاری روی یک هسته متمرکز نشود.

### زمان‌بند روی کدام هسته اجرا می‌شود؟

زمان‌بند در همه هسته‌ها اجرا می‌شود. هر هسته زمان‌بند خودش را دارد و می‌تواند به محض اینکه بیکار شود، فرآیندی از صف مشترک فرآیندها برداشته و آن را اجرا کند. این ویژگی باعث می‌شود که فرآیندهای سیستم به صورت متعادل و همزمان بین هسته‌ها توزیع شوند و بار کاری روی یک هسته متمرکز نشود.

---

## • اشکال زدایی

### روند اجرای GDB

(1) برای مشاهده Breakpoint‌ها از چه دستوری استفاده می‌شود؟

برای مشاهده Breakpoint‌ها در GDB، از دستور زیر استفاده می‌شود:

```
$ info breakpoints
```

این دستور تمام Breakpoint‌های فعلی و اطلاعات مربوط به آن‌ها را نشان می‌دهد.

(2) برای حذف یک Breakpoint از چه دستوری و چگونه استفاده می‌شود؟

برای حذف یک Breakpoint، از دستور زیر استفاده می‌شود:

```
$ delete [breakpoint_number]
```

مثلاً برای حذف Breakpoint شماره ۱:

```
$ delete 1
```

برای حذف تمام Breakpoint‌ها، می‌توان از دستور زیر استفاده کرد:

```
$ delete
```

کنترل روند اجرا و دسترسی به حالت سیستم

(3) دستور bt را اجرا کنید. خروجی آن چه چیزی را نشان می‌دهد؟

در GDB، دستور bt که مخفف "backtrace" است، برای نمایش پشته‌ی فرآخوانی‌های فعلی استفاده می‌شود. این دستور نشان می‌دهد که چگونه برنامه از نقطه فعلی اجرای خود به اینجا رسیده است، به عبارتی دیگر، نشان می‌دهد که چه توابعی به ترتیب فرآخوانی شده‌اند تا به نقطه فعلی برسیم.

```
$ bt
```

خروجی این دستور شامل لیستی از توابع فرآخوانی شده به همراه پارامترهای ورودی آن‌ها و مکان‌هایی در کد که این توابع فرآخوانی شده‌اند، می‌باشد.

---

#### 4) دو تفاوت دستورهای `x` و `print` را توضیح دهید. چگونه می‌توان محتوای یک ثبات خاص را چاپ کرد؟

- دستور `x`: برای بررسی محتویات حافظه استفاده می‌شود. این دستور به شما امکان می‌دهد یک آدرس مشخص را با فرمتهای مختلف نمایش دهید. ساختار کلی دستور به شکل زیر است:

```
x/nfu address
```

که در آن `n` تعداد واحدها، `f` فرمت نمایش، و `u` اندازه هر واحد است. به عنوان مثال: `4xw 0x7fffffff000` هگزادسیمال نمایش می‌دهد.

- دستور `print`: برای نمایش مقادیر متغیرها یا عبارات به کار می‌رود. این دستور می‌تواند برای نمایش مقادیر متغیرهای محلی، ثباتها و هر عبارت C که در برنامه استفاده شده، استفاده شود. ساختار کلی دستور به شکل زیر است:

```
print variable_name
```

این دستور مقدار متغیری به نام `variable_name` را نمایش می‌دهد.

برای چاپ محتوای یک ثبات خاص (`register`), می‌توان از دستور `print` استفاده کرد. برای نمایش یک ثبات خاص (مثل `eax`), از دستور `print` به شکل زیر استفاده می‌شود:

```
print $eax
```

علامت `$` برای اشاره به ثبات‌ها استفاده می‌شود.

#### 5) برای نمایش وضعیت ثبات‌ها از چه دستوری استفاده می‌شود؟ متغیرها محلی چطور؟ نتیجه این دستور را در گزارش‌کار خود بیاورید. همچنین در گزارش خود توضیح دهید که در معماری x86 رجیسترهاي `esi` و `edi` نشانگر چه چیزی هستند؟

برای نمایش وضعیت ثبات‌ها در GDB، از دستور `info registers` استفاده می‌شود. این دستور تمام مقادیر فعلی ثبات‌های CPU را نمایش می‌دهد.

برای نمایش متغیرهای محلی در یک تابع خاص، از دستور `info locals` استفاده می‌شود. این دستور تمامی متغیرهای محلی فعلی در تابعی که در آن قرار دارد را نشان می‌دهد.

## قرار دادن Breakpoint در ابتدای تابع :consoleintr ●

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000ffff0 in ?? ()
(gdb) b consoleintr
Breakpoint 1 at 0x80100fc0: file console.c, line 377.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, consoleintr (getc=0x801035e0 <kbdgetc>) at console.c:377
377      acquire(&cons.lock);
(gdb) list
372
373 void consoleintr(int (*getc)(void))
374 {
375     int c, doprocdump = 0;
376
377     acquire(&cons.lock);
378     while ((c = getc()) >= 0)
379     {
380         switch (c)
381     {
```

## وضعیت ثبات‌ها: ●

```
(gdb) info registers
eax          0x1          1
ecx          0x0          0
edx          0x0          0
ebx          0x80116c98    -2146341736
esp          0x80116c40    0x80116c40 <stack+3824>
ebp          0x80116c5c    0x80116c5c <stack+3852>
esi          0x80113020    -2146357216
edi          0x80113024    -2146357212
eip          0x80100fc0    0x80100fc0 <consoleintr>
eflags        0x86        [ IOPL=0 SF 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 ]
cr4          0x10        [ PSE ]
cr8          0x0          0
efer         0x0          [ ]
```

```
xmm0        {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0->repeats 16 times>, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm1        {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0->repeats 16 times>, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm2        {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0->repeats 16 times>, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm3        {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0->repeats 16 times>, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm4        {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0->repeats 16 times>, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm5        {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0->repeats 16 times>, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm6        {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0->repeats 16 times>, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm7        {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0->repeats 16 times>, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
rxcsr      0x1f80        [ IM DM ZM OM UM PM ]
```

- وضعیت متغیرهای محلی:

```
(gdb) info locals  
c = <optimized out>  
doprocdump = 0
```

در معماری x86، رجیسترهاي ESI (Extended Destination Index) و SSI (Source Index) جزو رجیسترهاي عمومي ۳۲ بيتی هستند و کاربردهای متعددی دارند. اما به طور خاص، در دستورات و عملکردهایی که شامل عملیات مربوط به رشته‌ها و انتقال داده‌ها هستند، این رجیسترها نقش مهمی دارند.

- ESI (Source Index): به عنوان شاخص منبع (source index) عمل می‌کند. این رجیستر معمولاً آدرس مبدا (منبع) داده‌هایی که باید خوانده یا پردازش شوند را نگه می‌دارد. در عملیات مربوط به رشته‌ها (مثل MOVS, LODS, SCAS)، رجیستر ESI برای اشاره به آدرس مبدا در حافظه استفاده می‌شود. به طور معمول این عملیات داده‌ها را از آدرسی که در ESI قرار دارد خوانده و به جای دیگری منتقل می‌کنند.
- EDI (Destination Index) (Destination Index): به عنوان شاخص مقصد (destination index) عمل می‌کند. این رجیستر معمولاً آدرس مقصد داده‌ها را نگه می‌دارد. در عملیات مرتبط با انتقال داده‌های رشته‌ای، رجیستر EDI برای اشاره به آدرس مقصد در حافظه استفاده می‌شود. داده‌هایی که از منبع (اشارة شده توسط ESI) خوانده می‌شوند، به آدرسی که EDI نشان می‌دهد، نوشته می‌شوند.

## 6) به کمک استفاده از GDB، درباره ساختار struct input موارد زیر را توضیح دهید:

- توضیح کلی این struct و متغیرهای درونی آن و نقش آنها

ساختار struct input برای مدیریت و نگهداری ورودی‌هایی که از دستگاه‌های ورودی مثل کیبورد به سیستم می‌رسد، استفاده می‌شود. این ساختار به عنوان یک بافر حلقی (circular buffer) طراحی شده است که ورودی‌های کاربر را در خود ذخیره کرده و سپس سیستم می‌تواند به این داده‌ها دسترسی پیدا کند. این ساختار به طور کلی در فایل console.c تعریف شده و شامل متغیرهایی است که برای مدیریت ورودی از جمله کیبورد استفاده می‌شود.

```

181     #define INPUT_BUF 128
182     struct {
183         char buf[INPUT_BUF];
184         uint r; // Read index
185         uint w; // Write index
186         uint e; // Edit index
187     } input;

```

- بک آرایه از نوع `char[INPUT_BUF]`: بک آرایه از نوع `char` و `[INPUT_BUF]` که ورودی‌ها (مثل کلیدهای زدهشده روی کیبورد) را نگه می‌دارد. `INPUT_BUF` به طور پیش‌فرض برابر با ۱۲۸ تعریف شده است، که به معنی این است که این بافر می‌تواند تا ۱۲۸ کاراکتر ورودی را نگهداری کند.
- `r` (اندیس خواندن): این متغیر نشان‌دهنده موقعیت خواندن در بافر است. وقتی سیستم می‌خواهد یک ورودی را پردازش کند، از این اندیس برای خواندن از بافر استفاده می‌شود. با هر خواندن یک ورودی، این اندیس به جلو حرکت می‌کند (به صورت حلقی).
- `w` (اندیس نوشتمن): این متغیر نشان‌دهنده موقعیت نوشتمن در بافر است. وقتی یک ورودی جدید مثل وارد کردن یک دستور دریافت می‌شود، داده در محل اندیس `w` در بافر نوشته می‌شود و سپس `w` به جلو حرکت می‌کند. اگر `w` به انتهای بافر برسد، دوباره به ابتدای بافر بر می‌گردد (حلقوی).
- `e` (اندیس ویرایش): این متغیر موقعیت فعلی در بافر را برای ویرایش یا اصلاح ورودی کاربر نشان می‌دهد. مثلاً زمانی که کاربر در حال تایپ است، `e` به سمت جلو حرکت می‌کند، اما اگر کاربر کلید Backspace را بزند، `e` به عقب حرکت می‌کند و ورودی قبلی حذف می‌شود. این متغیر تنها زمانی که کاربر هنوز کلید Enter را نزدیک نمایند، تغییر می‌کند، یعنی زمانی که کاربر در حال تایپ است و هنوز ورودی ثبت نهایی نشده است.

## ● نحوه و زمان تغییر مقدار متغیرهای درونی (برای مثال، `input.e` در چه حالتی تغییر می‌کند و چه مقداری می‌گیرد)

- متغیر `e` با هر کلید فشردهشده (مثل حروف و اعداد) افزایش پیدا می‌کند، یعنی به سمت جلو حرکت می‌کند. اگر کاربر کلید Backspace را فشار دهد، `e` کاهش می‌یابد، که نشان‌دهنده حذف آخرین کاراکتر وارد شده است. زمانی که کاربر کلید Enter را فشار دهد، ورودی کامل شده و پردازش می‌شود.
- متغیر `w` زمانی تغییر می‌کند که سیستم ورودی تایپ شده را در بافر قرار دهد، یعنی وقتی کاربر کلید Enter را فشار می‌دهد، ورودی تایپ شده در `buf` ذخیره می‌شود و `w` به جلو حرکت می‌کند. این مقدار همچنین با هر ورودی جدید در حال تغییر است و به سمت جلو حرکت می‌کند و همواره به اولین خانه‌ی بعد از آخرین ورودی اشاره می‌کند.

- 
- متغیر ۲ زمانی تغییر می‌کند که هسته سیستم (kernel) می‌خواهد ورودی‌های ذخیر شده در بافر را بخواند. معمولاً این اتفاق وقتی می‌افتد که ورودی کامل شده (مثل بعد از زدن Enter) و سیستم باید آن ورودی را پردازش کند. هر بار که ورودی پردازش می‌شود، ۲ به سمت جلو حرکت می‌کند تا به ورودی بعدی اشاره کند.

## (7) خروجی دستورهای layout asm و layout src در UI TUI چیست؟

: این دستور نمایشی از کد منبع (source code) را در رابط کاربری متى GDB فعال می‌کند. می‌توان خطوط کد منبع فعلی را که در اینجا به زبان C نوشته شده است در بالای صفحه مشاهده و به راحتی مکان‌یابی کرد.

: این دستور نمایشی از کد اسembly (assembly code) را در رابط کاربری متى GDB فعال می‌کند. می‌توان دستور العمل‌های اسembly که کامپایلر gcc از زبان C تولید کرده است را مشاهده نمود.

## (8) برای جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستوراتی استفاده می‌شود؟

- frame <n>

برای جابجایی به یک قاب (فریم) خاص در زنجیره فراخوانی استفاده می‌شود. با این دستور می‌توان به فریم شماره <n> در زنجیره فراخوانی رفت. قاب شماره ۰ قاب فعلی است (تابع فعلی که در آن قرار داریم).

- up <n>

این دستور برای حرکت به سمت قاب‌های بالاتر (یعنی به توابعی که فراخوانی‌کننده تابع فعلی بوده‌اند) در زنجیره فراخوانی استفاده می‌شود. می‌توان با این دستور به قاب بالاتر (caller function) رفت. اگر عددی <n> مشخص شود، به تعداد آن فریم‌ها به سمت بالا حرکت می‌کند.

- down <n>

این دستور برای حرکت به سمت قاب‌های پایین‌تر (یعنی به توابعی که توسط تابع فعلی فراخوانی شده‌اند) استفاده می‌شود. اگر عددی <n> مشخص شود، به تعداد آن فریم‌ها به سمت پایین حرکت می‌کند.

