

گزارش آزمایشگاه سیستم عامل

پروژه شماره ۱

سبحان کوشکی جهرمی و سید محمد جزایری و سید نوید هاشمی

810101496 - 810101399 - 810101549

1) معماری سیستم عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

xv6 یک پیاده سازی جدید از نسخه ششم یونیکس برای سیستم های چند پردازنده x86 است که معماری این سیستم عامل معماری کلی یونیکس است.

معماری سیستم عامل یونیکس (UNIX) یک طراحی لایه ای و ماژولار دارد که به بخش های مختلف تقسیم می شود. این طراحی به یونیکس اجازه می دهد که هم ساده و هم قدرتمند باشد، به طوری که بتواند به راحتی در سیستم های مختلف پیاده سازی و توسعه یابد. یونیکس به دلیل همین معماری ساده و موثر، پایه گذار بسیاری از سیستم های عامل مدرن مانند Linux و macOS بوده است.

معماری کلی یونیکس به چند لایه اصلی تقسیم می شود :

هسته (Kernel) , شل (Shell) , کتابخانه ها (Libraries) , برنامه های کاربردی (User Applications)

فلسفه یونیکس بر این بوده که " ساخت ابزارهای کوچک که یک کار را خوب انجام می دهند " یونیکس از ابتدا برای پشتیبانی از چندین کاربر (multiuser) و اجرای چندین فرآیند به صورت همزمان (multitasking) طراحی شده است. این امکان به کاربران مختلف اجازه می دهد که همزمان از منابع سیستم استفاده کنند.

یونیکس از یک ساختار **سلسله مراتبی** برای سیستم فایل خود استفاده می کند که تمام فایل ها و دایرکتوری ها را در یک ساختار درختی منظم سازمان دهی می کند. فایل ها از طریق مسیریابی و دایرکتوری ها قابل دسترسی هستند.

در کل معماری یونیکس به صورت لایه ای و ماژولار است، با یک هسته قوی که مدیریت منابع و فرآیندها را انجام می دهد و لایه های بالاتری که شامل شل، کتابخانه ها و برنامه های کاربردی هستند. طراحی ماژولار و فلسفه ابزارهای کوچک و تخصصی، آن را به یک سیستم عامل بسیار انعطاف پذیر و کارآمد تبدیل کرده است که پایه بسیاری از سیستم های عامل مدرن است.

از دلایل طراحی این سیستم عامل برای multiprocessor x86 این است که در فایل asm.h , mmu.h , به وضوح این موضوع اشاره شده و کدهایی برای این موضوع زده شده است.

```
//
// assembler macros to create x86 segments
//
#define SEG_NULLASM \
    .word 0, 0; \
    .byte 0, 0, 0, 0

// The 0xC0 means the limit is in 4096-byte units
// and (for executable segments) 32-bit mode.
#define SEG_ASM(type, base, lim) \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
        (0xc0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

#define STA_X 0x8 // Executable segment
#define STA_W 0x2 // Writeable (non-executable segments)
#define STA_R 0x2 // Readable (executable segments)
```

```
// This file contains definitions for the
// x86 memory management unit (MMU).

// Eflags register
#define FL_IF 0x00000200 // Interrupt Enable

// Control Register flags
#define CR0_PE 0x00000001 // Protection Enable
#define CR0_WP 0x00010000 // Write Protect
#define CR0_PG 0x80000000 // Paging

#define CR4_PSE 0x00000010 // Page size extension
```

2) یک پردازنده در سیستم عامل xv6 از چه بخش‌هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازنده‌های مختلف اختصاص می‌دهد؟

در سیستم عامل xv6، یک process (فرآیند) به مجموعه‌ای از منابع و وضعیت‌هایی گفته می‌شود که اجرای یک برنامه را توصیف می‌کند. هر فرآیند در xv6 از بخش‌های مختلفی تشکیل شده است که به آن اجازه می‌دهد از منابع سیستم (مانند پردازنده، حافظه و ورودی/خروجی) استفاده کند. همچنین، سیستم عامل xv6 دارای یک مکانیزم زمان‌بندی است که پردازنده را به فرآیندهای مختلف اختصاص می‌دهد.

بخش‌های یک فرآیند (Process) در xv6:

- 1- کد برنامه (Program Code)
- 2- داده‌ها (Data Segment)
- 3- پشته (Stack)
- 4- هیپ (Heap)
- 5- بلوک کنترل فرآیند (Process Control Block یا PCB)
- 6- فایل‌های باز (Open Files)

در xv6، هر فرآیند به نوبت و برای مدت زمان مشخصی به پردازنده دسترسی پیدا می‌کند. این مدت زمان مشخص به نام تکه زمانی (Time Slice یا Quantum) شناخته می‌شود. زمان‌بند xv6 به صورت تناوبی (دوره‌ای) فرآیندها را انتخاب می‌کند و آنها را برای مدت زمان مشخصی اجرا می‌کند. اگر فرآیند در این مدت زمان تکمیل نشود، پردازنده از آن گرفته شده و به فرآیند بعدی داده می‌شود. این چرخه ادامه می‌یابد تا تمام فرآیندها به طور منصفانه پردازنده را دریافت کنند. در این میان برنامه‌هایی که کامل تمام نشده اند وقتی پردازنده از آن‌ها گرفته می‌شود

روند آن‌ها در حافظه ذخیره می‌شود که در نوبت بعدی از دوباره این فرآیند روندش از جایی که قطع شده است ادامه پیدا کند و به این صورت اطلاعاتی در عملیات از بین نمی‌رود.

3) مفهوم file descriptor در سیستم عامل های مبتنی بر UNIX چیست؟ عملکرد pipe در سیستم عامل xv6 چگونه است و به طور معمول برای چه هدفی استفاده می‌شود؟

File Descriptor یک عدد صحیح است که سیستم عامل برای هر فرآیند ایجاد می‌کند تا به یک فایل یا منبع خاص دسترسی داشته باشد.

وقتی یک فرآیند فایلی را باز می‌کند، سیستم عامل یک عدد کوچک یکتا را به آن تخصیص می‌دهد که به عنوان file descriptor شناخته می‌شود. فرآیند می‌تواند با استفاده از این عدد به فایل دسترسی پیدا کند.

این عدد به توابع مختلف مانند `read()`, `write()`, `close` و غیره به عنوان آرگومان منتقل می‌شود، به طوری که این توابع از طریق آن به فایل مرتبط دسترسی پیدا می‌کنند.

File descriptor های استاندارد:

0: ورودی استاندارد (`stdin`)

1: خروجی استاندارد (`stdout`)

2: خروجی خطا (`stderr`)

این file descriptor ها به صورت پیش فرض برای هر فرآیند باز هستند و برای تعامل با ترمینال و ورودی/خروجی های متنی استفاده می‌شوند.

Pipe یک مکانیزم ارتباطی یک طرفه است که به دو فرآیند اجازه می‌دهد با یکدیگر ارتباط برقرار کنند. این ارتباط معمولاً به این شکل است که یک فرآیند اطلاعاتی را می‌نویسد و فرآیند دیگر آن اطلاعات را می‌خواند.

در xv6، پایپ یک `buffer` حلقوی است که داده‌ها را از یک فرآیند به فرآیند دیگر منتقل می‌کند. معمولاً pipe برای ارتباط والد فرزند استفاده می‌شود. فرآیند والد یک pipe ایجاد می‌کند، سپس با استفاده از `fork()` یک فرآیند فرزند ایجاد می‌کند. فرآیند والد می‌تواند داده‌ها را به pipe بنویسد و فرآیند فرزند آن داده‌ها را از pipe بخواند (یا برعکس).

4) فراخوانی های سیستمی exec و fork چه عملی انجام می‌دهند ؟ از نظر طراحی , ادغام نکردن این دو چه مزیتی دارد؟

fork() یک فراخوان سیستمی است که برای ایجاد یک فرآیند جدید استفاده می‌شود. فرآیند جدید، یک کپی دقیق از فرآیند والد است. وقتی فرآیندی از **fork()** استفاده می‌کند، سیستم‌عامل یک فرآیند فرزند ایجاد می‌کند که دقیقاً مشابه والد است. یعنی: حافظه فرآیند، متغیرهای آن، فایل‌های باز، و وضعیت رجیسترها همگی به فرآیند فرزند کپی می‌شوند. اگر برنامه فرزند برنامه به دلیلی متوقف شود و یا اجرا به طول بینجامد برای برنامه پدر نیز همین اتفاق می‌افتد. پس از اجرای برنامه فرزند به برنامه پدر باز می‌گردیم. **exec()** فراخوان سیستمی است که یک برنامه جدید را در فرآیند فعلی بارگذاری می‌کند و اجرای آن را از سر می‌گیرد. این فراخوانی به یک فرآیند اجازه انتقال به یک برنامه دیگر را می‌دهد. مانند زمانی که یک برنامه نیاز به اجرای برنامه دیگری را دارد. در اثر اجرای این فراخوانی سیستمی کد و داده‌های برنامه قدیمی از حافظه حذف می‌شوند و فضای آدرس پاک می‌شود و کد بارگذاری می‌شود و داده‌های برنامه جدید در فضای آدرس **process** بارگذاری می‌شود. برخلاف تابع **fork** برنامه به caller تابع **exec()** باز نمی‌گردد و برنامه جدید اجرا می‌شود. ادغام نکردن این دو فراخوانی این امکان را می‌دهد که تغییرات لازم را در صورت لزوم پس از فراخوانی **fork** در **file descriptor** انجام دهد و سپس فراخوانی **exec** انجام شود.

اضافه کردن یک متن به Boot Message:

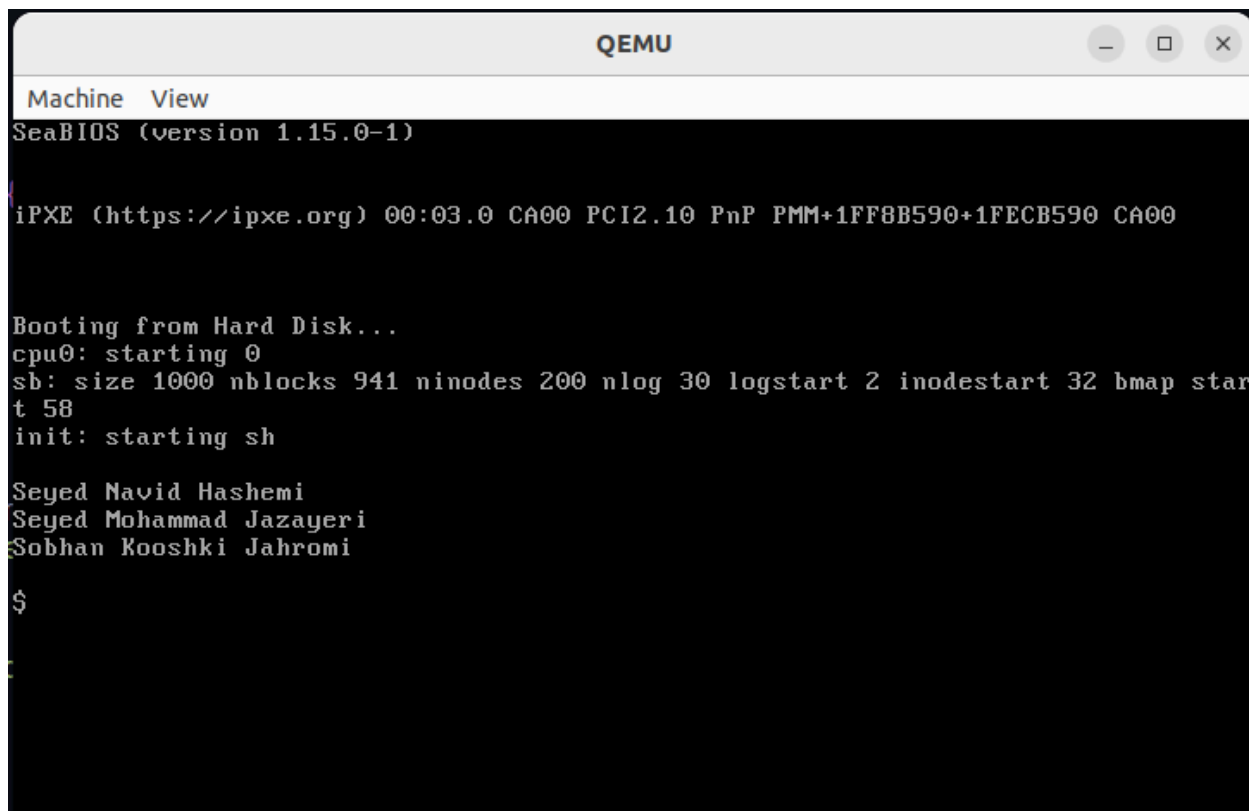
در این قسمت ما در قسمت **init** که در همان اول توسط **boot loader** صدا زده می‌شود ما توسط یک خط کد ساده می‌توانیم هر متنی را در اول **boot** شدن سیستم به آن اضافه کنیم در قطه کد زیر این خط کد را مشاهده می‌کنیم.

```
int
main(void)
{
    int pid, wpid;

    if(open("console", O_RDWR) < 0){
        mknod("console", 1, 1);
        open("console", O_RDWR);
    }
    dup(0); // stdout
    dup(0); // stderr

    for(;;){
        printf(1, "init: starting sh\n");
        printf(1, "\nSeyed Navid Hashemi\nSeyed Mohammad Jazayeri\nSobhan Kooshki Jahromi\n\n");
        pid = fork();
        if(pid < 0){
            printf(1, "init: fork failed\n");
            exit();
        }
    }
}
```

این کد در `init.c` و در تابع `main` قرار داده شده است.
نتیجه انجام این قطعه کد را در خروجی زیر میتوانیم ببینیم.



```
Machine View
SeaBIOS (version 1.15.0-1)

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

Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$
```

اسم اعضای گروه را در ابتدای و در قسمت `boot` مشاهده میکنیم.

کاربر با فشردن کلیدهای `→` و `←` بتواند نشانه گر (`cursor`) را بین کاراکترهای موجود در رشته ورودی کنونی کنسول جلو و عقب ببرد.

در اوایل مرحله باید در `console.c` تابعی که دکمه های کیبورد را میفهمد را پیدا کنیم که این تابع همان `consoleintr` است در اینجا حال کد زیر را اضافه میکنیم تا به `kernel` دکمه های چپ و راست را بفهمانیم.

`LEFT_ARROW` و `RIGHT_ARROW` در واثع کد های اسکی دکمه های بالا و پایین هستند که به ترتیب 228 , 229 هستند.

```

case LEFT_ARROW:
    if((input.e - num_of_left_pressed) > input.w){
        change_cursor_position(0);
        num_of_left_pressed++;
    }
    break;

case RIGHT_ARROW:
    if(num_of_left_pressed > 0){
        change_cursor_position(1);
        num_of_left_pressed--;
    }
    break;

```

دو شرط مهم در تاثیرگذاری این دکمه ها عبارت های زیر هستند:

اگر ما در ابتدای رشته ورودی باشیم (خط 2 کد بالا) نمیتوانیم به سمت چپ برویم. این کار را با استفاده از `input.e` , `input.w` و تعداد دفعات چپ رفتن چک میکنیم.

اگر ما در انتهای رشته باشیم نمیتوانیم به سمت راست برویم این کار را با استفاده از چک کردن تعداد دفعه هایی که به سمت چپ رفته ایم انجام میدهیم.

وقتی در شرط رفتیم تابع `change_cursor_position` را طبق ارگومان خاص اجرا میکنیم

```

void
change_cursor_position(int direction){
    //get cursor position
    int pos;

    // Cursor position: col + 80*row.
    outb(CRTPORT, 14);
    pos = inb(CRTPORT+1) << 8;
    outb(CRTPORT, 15);
    pos |= inb(CRTPORT+1);

    //case
    switch(direction){
        case 0:
            pos -= 1;
            break;
        case 1:
            pos += 1;
            break;
        default:
            break;
    }

    //update
    outb(CRTPORT, 14);
    outb(CRTPORT+1, pos>>8);
    outb(CRTPORT, 15);
    outb(CRTPORT+1, pos);
}

```

در این بخش ما جایگاه cursor را درست میکنیم.

```
//shift_back
void shift_back(int pos){
    for(int i = pos-1; i < pos + num_of_left_pressed; i++){
        crt[i] = crt[i + 1];
    }
}

// push_right
void push_right(int pos){
    for(int i = pos + num_of_left_pressed; i > pos; i--){
        crt[i] = crt[i-1];
    }
}

//-----
```

این بخش که در `cgaputc` جایگاه بقیه کلمات را با توجه به اینکه به سمت چپ رفتیم یا راست درست میکنیم

نتیجه کد های این بخش را در عکس های پایین میبینیم

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
t 58
init: starting sh
(
Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$ we can move the cursor
```

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
t 58
init: starting sh
(
Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$ we can move the cursor
```

در تست بالا تا حرف `a` توانستیم به عقب بیایم و بعد توانستیم به سمت راست برویم و بر روی حرف `۲` بایستیم.

کاربر با وارد کردن کلمه **history** بتواند **10** دستور اخیر وارد شده را به ترتیب در کنسول مشاهده کند و با فشردن کلیدهای \uparrow و \downarrow بتواند بین دستورهای موجود در تاریخچه جابجا شود و مستقیماً در کنسول آنها را مشاهده کند.

ما با تعریف یک ساختار برای نگه داری تاریخچه دستورات ابتدا توانستیم یک ساختار مناسب را برای پیاده سازی دستور **history** و یا فشردن کلیدهای بالا و پایین ایجاد کنیم.

```
struct {
    struct Input hist[NUM_OF_HISTORY_COMMAND];
    struct Input current_command;
    int num_of_cmnd;
    int start_index;
    int num_of_press;
}history_cmnd;
```

سپس در شل با استفاده از کد روبرو توانستیم دستور **history** را به کرنل بفهمانیم این تابع در قسمت **runcmd** در فایل **sh.c** وجود دارد در کیس های **EXEC** این کد را اضافه میکنیم

```
92     case EXEC:
93         ecmd = (struct execcmd*)cmd;
94         if(strcmp(ecmd->argv[0], "history") == 0){
95             exit();
96         }
```

در ادامه بخش های مهم دستورات **history** را آورده ایم

```
void show_current_history(int temp){
    for(int i = temp; i > 0; i--){
        if(input.buf[i - 1] != '\n'){
            consputc(BACKSPACE);
            input.e--;
        }
        input = history_cmnd.current_command;
    }
    for(int j = input.w; j < input.e; j++){
        consputc(input.buf[j]);
    }
}
```

char *command

این قطعه کد برای نشان دادن دستور فعلی که در بافر کامند وجود دارد و نمیخواهیم با نشان دادن بقیه تاریخچه ها از بین برود (مانند لینوکس) گذاشته ایم و سپس با برگشتن به آخرین دستور وارد شده متنی که در بافر بود نمایش داده میشود

```
0
1  int is_history(char* command){
2      for(int i = 0; i < 8; i++){
3          if(command[i] != input.buf[i + input.w]){
4              return 0;
5          }
6      }
7      return 1;
8  }
```

این قسمت بررسی میکنیم که command وارد شده history است یا خیر

```
void handle_up_and_down_arrow(enum Direction dir){
    for(int i = input.e; i > input.w; i--){
        if(input.buf[i - 1] != '\n'){
            consputc(BACKSPACE);
        }
    }
    if(dir == UP){
        input = history_cmnd.hist[history_cmnd.num_of_cmnd - history_cmnd.num_of_press];
        input.e--;
    }
    if(dir == DOWN){
        input = history_cmnd.hist[history_cmnd.num_of_cmnd - history_cmnd.num_of_press];
        input.e--;
    }

    //here is about showing on the console
    for(int i = input.w; i < input.e; i++){
        consputc(input.buf[i]);
    }
}
```

در این قطعه کد هم میبینیم با استفاده از این تابع میتوانیم بین history ها switch کنیم

```

case UP_ARROW:
    for(int i = 0; i < num_of_left_pressed; i++)
        change_cursor_position(1);
    num_of_left_pressed = 0;
    if(currecnt_com == 0){
        history_cmnd.current_command = input;
        currecnt_com = 1;
    }
    if(history_cmnd.num_of_cmnd != 0 && history_cmnd.num_of_press < history_cmnd.num_of_cmnd){
        history_cmnd.num_of_press++;
        handle_up_and_down_arrow(UP);
    }
    break;
case DOWN_ARROW:
    for(int i = 0; i < num_of_left_pressed; i++)
        change_cursor_position(1);
    num_of_left_pressed = 0;
    if(history_cmnd.num_of_cmnd != 0 && history_cmnd.num_of_press > 1){
        history_cmnd.num_of_press--;
        handle_up_and_down_arrow(DOWN);
    }
    else if(history_cmnd.num_of_press == 1){
        history_cmnd.num_of_press = 0;
        int temp = input.e - input.w;
        show_current_history(temp);
        currecnt_com = 0;
    }
}

```

در این بخش هم مانند قسمت اول باید دکمه های بالا و پایین را برای kernel تعریف کنیم این کار را با استفاده از اضافه کردن به کیس consoleintr انجام میدهیم و نتیجه کلی این دستور به صورت زیر است که ما بر روی دستورات user_program انجام دادیم.

```

Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$ user_program encode "a"
$ cat result.txt
"a"
$ user_program encode "u"
$ cat result.txt
"o"
$ user_program decode "o"
$ cat result.txt
"u"
$ history
ute
_program encode "a"
cat result.txt
user_program encode "u"
cat result.txt
user_program decode "o"
cat result.txt
history
$ _

```

در صورت فشردن دستور **Ctrl+S** توسط کاربر، مادامی که دستور **Ctrl+F** زده نشده است، هر کاراکتر حرفی دیگری که در رشته ورودی کنسول وارد می شود ، **copy** شود و با فشردن دستور **Ctrl+F** عینا **paste** شود

```
char coppied_input[128];
int cur_index = 0;
int num_of_left_copy = 0;
int ctrl_s_start = 0;
```

ما برای این دستور چند متغیر **global** تعریف میکنیم در خط اول آمده ایم یک بافر برای خود در نظر گرفته ایم که بعد از اینکه **ctrl+S** زده شد کاراکتر های ورودی را در آن ذخیره کنیم در خط آخر آمده ایم و یک فلگ تعریف کرده ایم که بفهمیم فرایند **copy** شروع شده است

```
break;
case C('S'):
    cur_index = 0;
    ctrl_s_pressed = 1;
    ctrl_s_start = input.e - input.w;
    //is about chacking ctrl s and f
    handle_ctrl_s();

break;
case C('F'):
    handle_ctrl_f();
break;
default:
```

در ای قسمت در تابع **consoleintr** می اییم و کلید های **ctrl+F** , **ctrl+S** را تعریف میکنیم و با این کار حالا میتوانیم فرایند خود را انجام دهیم

```
void handle_ctrl_s(){
    start_ctrl_s = input.e;
    memset(coppied_input, '\0', sizeof(coppied_input));
}
void handle_ctrl_f(){
    if(ctrl_s_pressed){
        // copied_command = input;
        print_copied_command();
    }
}
```

در `handle_ctr_s` ما می‌ایم و ان ارایه `coppied_input` را به `initial` , `null` می‌کنیم تا آماده ریختن کاراکتر ها در ان شویم

```
if(ctrl_s_pressed){
    if(num_of_left_pressed == 0){
        coppied_input[cur_index] = c;
        cur_index++;
    }
    else{
        if(cur_index - num_of_left_pressed >= 0){
            handle_copying();
            coppied_input[cur_index - num_of_left_copy] = c;
            cur_index++;
        }
        else if(num_of_left_copy > 0 && num_of_left_copy != cur_index){
            coppied_input[cur_index - num_of_left_copy] = c;
            cur_index++;
        }
    }
    // cur_index++;
}
```

در این قطعه کد که در قسمت `default` تابع `consoleintr` است می‌ایم و فرایند `copy` از بافر ورودی و در `coppied_input` ریختن را انجام می‌دهیم و با استفاده از شرط هایی که صورت سوال برای ما طرح کرده سعی بر این داریم که کاراکتر هایی که نباید در `coppied_input` ذخیره شوند به داخل این ارایه نیایند
در بخش `handle_ctr_f` به سراغ نوشتن این ارایه بافر در خروجی نمایش داده شده و در بافر اصلی ذخیره می‌کنیم.

```
void handle_shifting(int length, int cursor_pos){
    for (int i = input.e - 1; i >= cursor_pos; i--){
        input.buf[i + length] = input.buf[i];
    }
}

void print_coppied_command(){
    for(int i = 0; i < cur_index; i++){
        consputc(coppied_input[i]);
    }
    if(num_of_left_pressed > 0){
        handle_shifting(cur_index, input.e - num_of_left_pressed);
    }
    int x = input.e;
    for(int i = 0; coppied_input[i] != '\0'; i++){
        input.buf[x - num_of_left_pressed + i] = coppied_input[i];
        input.e++;
    }
}
```

در `print_copied_command` ما این ارایه را در خروجی که میبینیم مینویسیم و سپس در `for` انتها ما در بافر اصلی که `kernel` آن را در کارهای خود میبیند هم وارد میکنیم و ذخیره میکنیم با این چند قطعه کد که در بالا نشان داده ایم میتوانیم مراحل `copy` و `paste` را با موفقیت انجام دهیم.

```
Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$ we are a best
```

در این جا ما قبل از نوشتن `best` کپی را زدیم (`ctrl+S`) و `best` در بافر ذخیره میشود

```
Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$ we are a best best
```

سپس با `ctrl+F` میتوانیم کاراکترهایی که در بافر ذخیره کرده بودیم را روی خروجی ببینیم

در صورتی که رشته ورودی کنسول عبارتهای عددی که دارای الگوی `NON=?` باشند (منظور از `N` ، عدد و منظور از `O` , عملگر است) وارد شود، کل عبارت حذف شده و حاصل عددی آن در همان محل جایگزین شود. برای مثال عبارت `a2+3=?b` تبدیل می شود به `a5b`.

```
default:
//here is about checking operation
if(there_is_question_mark()){
    int qm_index = find_question_mark_index();
    search_for_NON(qm_index); //int type, int current_index, char temp_char
}
```

در تابع `consoleintr` در بخش `default` همیشه بررسی میکنیم که کاراکتر ؟ وارد شده است یا خیر اگر این طور بود میرویم و `index` این کاراکتر را پیدا میکنیم حال به بخش بررسی الگوی `NON` میرویم تا ببینیم که آیا همچین الگویی در قبل از ؟ وجود دارد یا خیر

```
void search_for_NON(int qm_index){
    index_question_mark = qm_index;
    for(int i = 1; i <= 4; i++){
        check_states_question_mark(input.buf[qm_index - i]);
    }
}
```

چون در سوال گفته شده است که ما اعداد تک رقمی را در اینجا باید هندل کنیم پس ما فقط تا 4 تا قبل تر از علامت سوال را چک میکنیم (= و NON)

```
void check_states_question_mark(char c){
    switch (state_of_question_mark)
    {
        case 0:
            if(!is_equal_mark(c)){
                state_of_question_mark = 0;
            }
            else{
                state_of_question_mark = 1;
            }
            break;
        case 1:
            if(is_digit(c)){
                first_digit = c;
                state_of_question_mark = 2;
            }
            else{
                state_of_question_mark = 0;
            }
            break;
        case 2:
            if(is_operand(c)){
                state_of_question_mark = 3;
                operand = c;
            }
            else{
                state_of_question_mark = 0;
            }
            break;
        case 3:
            if(is_digit(c)){
                second_digit = c;
                do_operation();
                state_of_question_mark = 0;
            }
            else{
                state_of_question_mark = 0;
            }
            break;
    }
}
```

در اینجا ما ی استیت ماشین طراحی کرده ایم که می اییم و این الگو را چک میکنیم اگر کامل درست بود حالا به بخش do_operation میرویم و operation مورد نظر را انجام میدهم

```
void do_operation(){
    int first_num = first_digit - '0';
    int second_num = second_digit - '0';
    float result = 0;
    switch (operand)
    {
        case '+':
            result = (float)second_num + (float)first_num;
            break;
        case '-':
            result = (float)second_num - (float)first_num;
            break;
        case '*':
            result = (float)second_num * (float)first_num;
            break;
        case '/':
            result = (float)second_num / (float)first_num;
            break;
        default:
            result = (float)second_num + (float)first_num;
            break;
    }

    // release(&cons.lock);
    // printf("\n\nsecond = %d\n\n", first_num);
    // acquire(&cons.lock);
    // release(&cons.lock);
    // printf("\nresult\n = %d", result);
    // acquire(&cons.lock);
    char result_as_string[INPUT_BUF];
    memset(result_as_string, '\0', sizeof(result_as_string));
    int num_res_digits = int_to_string(result, result_as_string);
    show_result(num_res_digits, result_as_string);
}
```

در این قسمت تمام حالت ها را نوشت ایم و محاسبه انجام میشود و در اخر ان را در تابع show_result نمایش میدهم

```
void show_result(int offset, char* result){
    for(int i = input.e - num_of_left_pressed; i <= index_question_mark; i++){
        change_cursor_position(1);
        num_of_left_pressed--;
    }
    for(int i = 0; i < 5; i++){
        consputc(BACKSPACE);
        input.e--;
    }

    int temp_e = input.e;
    for(int i = 0; i < 5; i++){
        for(int j = index_question_mark + 1; j <= temp_e - 1; j++){
            input.buf[j - 1] = input.buf[j];
        }
    }

    for(int j = offset - 1; j >= 0; j--){
        input.buf[input.e] = result[j];
        consputc(input.buf[input.e]);
        input.e++;
    }
    //here is about clearing arrays
}
```

این قسمت از نمایش به صورت real time است و شما اگر الگوی NON= را داشته باشید و کاراکتر ؟ را وارد کنید به صورت real time می آید و جواب را مینویسد.
نتیج کد در عکس های زیر میبینید

```
Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$ 8/2=
```

```
Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$ 4
```

```
Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$ 4sodf j,5+2=smdf lkjsdf
```

```
Seyed Navid Hashemi
Seyed Mohammad Jazayeri
Sobhan Kooshki Jahromi

$ 4sodf j,7smdf lkjsdf
```

برنامه سطح کاربر:

در این بخش ما به کد زدن یک encoder و یک decoder در سیستم عامل xv6 میپردازیم.
کد را به زبان C با الگوریتم سزار پیاده سازی میکنیم در عکس زیر بخش های مهم کد این الگوریتم به شما نمایش داده شده است.

```
void encode(char *text, int key) {
    int i;
    for (i = 0; text[i] != '\0'; i++) {
        char ch = text[i];
        if (ch >= 'a' && ch <= 'z') {
            text[i] = ((ch - 'a' + key) % 26) + 'a';
        } else if (ch >= 'A' && ch <= 'Z') {
            text[i] = ((ch - 'A' + key) % 26) + 'A';
        }
    }
}
```

این بخش هم انکودر ما و هم بخش دیکودر ما است ما میتوانیم با استفاده از پس دادن key-26 این بخش را مانند decoder کنیم و در نوشتن تابع decoder صرفه جویی کنیم

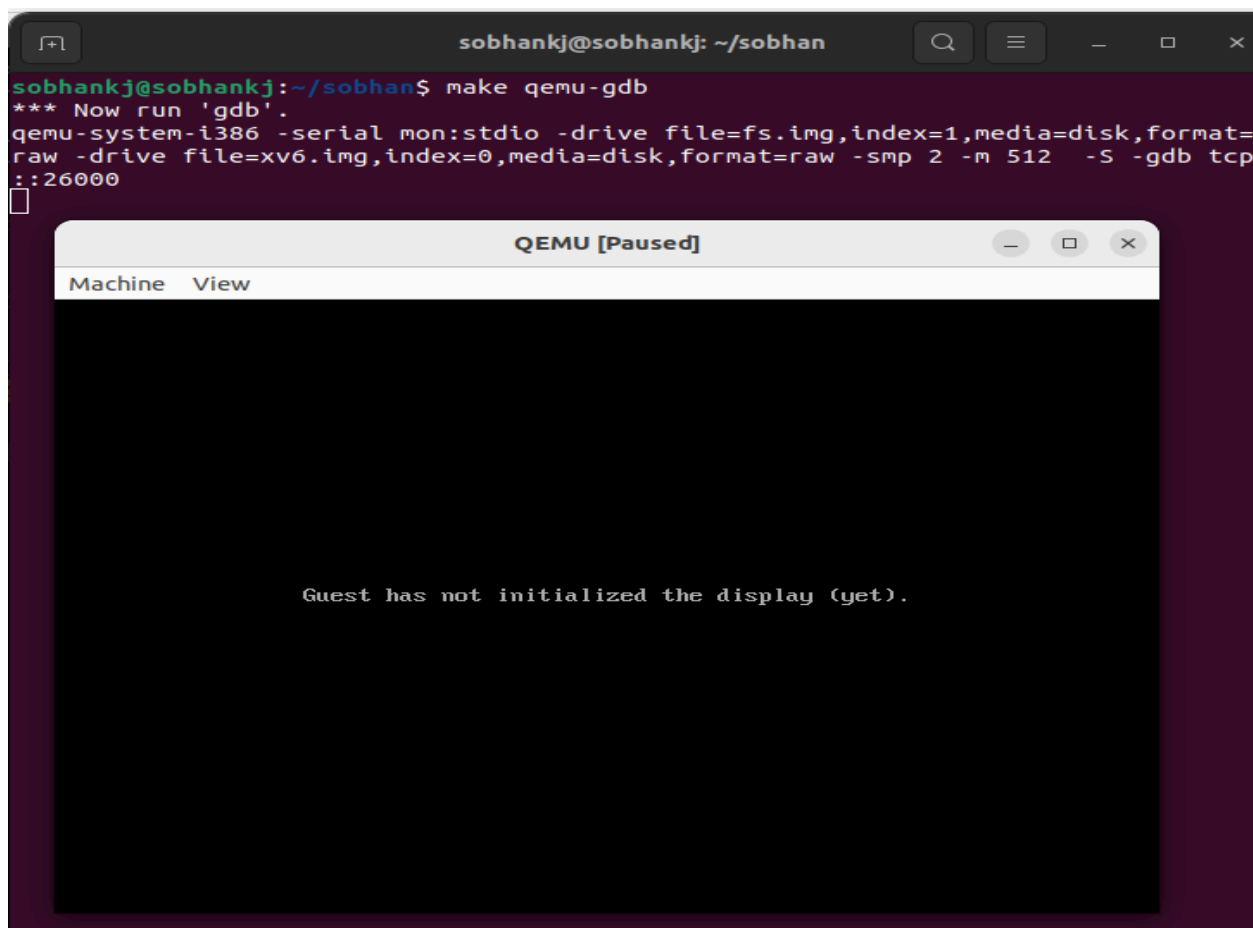
```
char *action = argv[1];
char *text = argv[2];
int key = (argc == 4) ? atoi(argv[3]) : 20; // default key is 20

if (strcmp(action, "encode") == 0) {
    encode(text, key);
} else if (strcmp(action, "decode") == 0) {
    // we use a encode as a decoder by passing a not key to encode function
    encode(text, 26 - key);
} else {
    printf(1, "Invalid action. Use 'encode' or 'decode'.\n");
    exit();
}
```


این هم بخش مین این برنامه که در ابتدا ما `buffer` نوشته شده را میگیریم و بررسی میکنیم تعداد ارگومان های آن را در ارگومان یک نام برنامه در ارگومان دو اسم `encode` یا `decode` را میگیریم و هر کدام از این ها گفته شده بود آن تابع را انجام میدهیم و در ارگومان سوم متن درخواست داده شده را میگیریم. شماره دانشجویی های ما مود به 26 برابر 20 میشود به همین دلیل ما به صورت پیش فرض `key` خود را 20 گذاشته ایم.

اشکال زدایی

در این بخش طبق توضیحاتی که داده شده است می توانیم برنامه های خود را در محیط `gdb` که یک اشکال زدا که مستقل از نوع اشکال بوده و تنها اجرا را ردگیری نموده و اطلاعاتی از حالت سیستم (شامل سخت افزار و نرم افزار) در حین اجرا یا پس از اجرا جهت درک بهتر رفتار برنامه برمی گردانند ، دیباگ کنیم
با دستور `make qemu-gdb` وارد این قسمت `gdb` میشویم



```
sobhankj@sobhankj: ~/sobhan
sobhankj@sobhankj:~/sobhan$ make qemu-gdb
*** Now run 'gdb'.
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 -S -gdb tcp://:26000
```

The screenshot shows a terminal window with the command `make qemu-gdb` executed. The output indicates that the program is now running in `gdb` and provides the command line for `qemu-system-i386`. Below the terminal, a `QEMU [Paused]` window is visible, showing a message: "Guest has not initialized the display (yet)."

و سپس با استفاده از یک ترمینال دیگر و وصل شدن به محیط `gdb` میتوانیم آن برنامه ای را که میخواهیم دیباگ کنیم

```

sobhankj@sobhankj:~/sobhan$ gdb _user_program
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from _user_program...
warning: File "/home/sobhankj/sobhan/.gdbinit" auto-loading has been declined by your 'auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
  add-auto-load-safe-path /home/sobhankj/sobhan/.gdbinit
line to your configuration file "/home/sobhankj/.config/gdb/gdbinit".
To completely disable this security protection add
  set auto-load safe-path /
line to your configuration file "/home/sobhankj/.config/gdb/gdbinit".
For more information about this security protection see the
--Type <RET> for more, q to quit, c to continue without paging--c
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
  info "(gdb)Auto-loading safe path"
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000ffff in ?? ()
(gdb) break 8
Breakpoint 1 at 0x190: file user_program.c, line 8.
(gdb) info break
Num      Type             Disp Enb Address            What
1        breakpoint       keep y   0x000000190 in encode at user_program.c:8
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, encode (text=0x2fe4 "\a", key=20) at user_program.c:8
8      for (i = 0; text[i] != '\0'; i++) {
(gdb) next
10     if (ch >= 'a' && ch <= 'z') {
(gdb)

```

برای مشاهده breakpoint ها از چه دستوری استفاده می شود؟

برای مشاهده breakpoint ها که در GDB گذاشته ایم، از دستور زیر استفاده می شود

info break

این دستور لیستی از تمام breakpoints و watchpoints فعال را نمایش می دهد. اطلاعاتی که این دستور نشان می دهد شامل شماره ی breakpoint، مکان آن در کد، شرایط و وضعیت فعال یا غیرفعال بودن است.

برای حذف یک breakpoint از چه دستوری و چگونه استفاده می شود؟

برای حذف یک breakpoint در GDB، از دستور **delete** استفاده می شود. این دستور به شما امکان می دهد تا یک یا چند breakpoint خاص را حذف کنید

delete <breakpoint-number> حذف یک نقطه

Delete برای حذف همه

دستور زیر را اجرا کنید. خروجی آن چه چیزی را نشان می دهد

دستور **bt** در GDB مخفف **backtrace** است و برای نمایش پشته‌ی فراخوانی‌ها (**call stack**) استفاده می‌شود. این دستور لیستی از توابعی که در حال حاضر در حال اجرا هستند و ترتیب فراخوانی آنها از ابتدا تا جایی که خطایابی (**debugging**) در حال انجام است را نشان می‌دهد.

```
gdb) bt
#0  encode (text=0x2fe4 "\"a\"", key=20) at user_program.c:10
#1  0x00000054 in main (argc=3, argv=0x2fd4) at user_program.c:29
gdb)
```

دو تفاوت دستورهای **x** و **print** را توضیح دهید. چگونه می توان محتوای یک ثبات خاص را چاپ کرد؟

دستور **print** برای چاپ متغیرهای سطح بالا در برنامه استفاده می‌شود. وقتی از **print** استفاده می‌کنی، GDB به نوع داده (مثل **int**، **float**، **char** یا حتی اشاره‌گر (pointer)) توجه می‌کند و مقدار آن را به شیوه‌ای مناسب و خوانا برای شما چاپ می‌کند. دستور **x** برای بررسی محتوای خام حافظه استفاده می‌شود. در اینجا شما باید آدرس حافظه را مستقیماً به GDB بدهید و به آن بگویید که چطور محتوای آن را نمایش دهد. GDB نوع داده‌ای را که در آن آدرس قرار دارد، نمی‌داند، بلکه فقط می‌داند شما می‌خواهید محتویات حافظه را بررسی کنید.

print با متغیرهای سطح بالا کار می‌کند. نیازی نیست آدرس دقیق متغیر را بدانید؛ فقط نام متغیر را وارد کنید، و GDB خودش نوع داده و مقدارش را می‌فهمد و چاپ می‌کند. **x** به محتوای خام حافظه دسترسی دارد. شما باید دقیقاً بگویید در کدام آدرس حافظه می‌خواهید چه چیزی را ببینید، و GDB نمی‌داند که در آن آدرس چه نوع داده‌ای قرار دارد.

برای نمایش وضعیت ثبات‌ها از چه دستوری استفاده می شود؟ متغیرها محلی چطور؟ نتیجه این دستور را در گزارش کار خود بیاورید. همچنین در گزارش خود توضیح دهید که در معماری **x86** رجیسترهای **Esi** و **esi** نشانگر چه چیزی هستند؟

برای نمایش وضعیت **register** ها می توانیم از دستور **info registers** استفاده کنیم و برای متغیرهای محلی از دستور **info locals** استفاده کنیم
رجیستر **esi** اغلب به عنوان شاخص منبع در عملیات‌های پردازشی استفاده می‌شود. به ویژه، در دستورالعمل‌هایی که نیاز به جابجایی داده‌ها دارند (مانند **movs**، **cmps**، **scas**)، این ثبات به عنوان آدرس منبع عمل می‌کند.

ESI در هنگام کپی کردن داده‌ها از یک مکان در حافظه به مکان دیگر یا در عملیات مقایسه‌ای که نیاز به استفاده از یک منبع (Source) دارند، استفاده می‌شود.

Edi اغلب به عنوان شاخص مقصد عمل می‌کند و در دستورالعمل‌هایی که داده‌ها را به یک مکان در حافظه منتقل می‌کنند، مانند **movs**, **stos**, **cmps**، به عنوان آدرس مقصد استفاده می‌شود.

EDI در هنگام کپی کردن داده‌ها به یک مکان خاص در حافظه به کار می‌رود.

```
(gdb) info locals
ch = 34 ''
i = 0
(gdb)
```

```
#1 0x00000054 in main (argc=3, argv=0x2fd4) at user_program.c:29
(gdb) info registers
eax            0x22          34
ecx            0x948        2376
edx            0x2fee       12270
ebx            0x2fe4       12260
esp            0x2f7c       0x2f7c
ebp            0x2f88       0x2f88
esi            0x14         20
edi            0x4ec4ec4f    1321528399
eip            0x1d8        0x1d8 <encode+72>
eflags        0x206        [ IOPL=0 IF PF ]
cs             0x1b        27
ss             0x23        35
ds             0x23        35
es             0x23        35
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            0xdf23000    [ PDBR=57123 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, 0x1f80}, v2_double = {0x0, 0x1f800000000000}, v16_int8 = {0x0 <repeats 12 times>, 0x80, 0x1f, 0x0, 0x0}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x1f80, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x1f80}, v2_int64 = {0x0, 0x1f800000000000}, uint128 = 0x1f800000000000000000000000000000}
mxcsr          0x1f80        [ IM DM ZM OM UM PM ]
(gdb)
```

به کمک استفاده از **GDB** درباره ساختار **struct input** موارد زیر را توضیح دهید:

این استراکت دارای یک آرایه ۱۲۸ تایی از کاراکتر (بافر) و ۳ متغیر است: **e**, **w**, **r**.

هر ۳ این متغیرها نشاندهنده اندیس در بافر هستند. **r** اندیسی است که تا آنجا دستورات اجرا شده و توسط سیستم عامل مدیریت شده اند. **w** اندیسی است که تا آنجا در بافر ذخیره شده است (یعنی مثلاً در هنگام وارد کردن دستور جدید، اندیس اول خط نمایش داده میشود) و پس از وارد کردن

کامند جدید و فشردن اینتر w , r اپدیت میشوند و مقدار e را میگیرند e هم اندیسی که در حال تایپ در آن هستیم را نمایش میدهد (یعنی مثلا با وارد کردن یک کاراکتر جدید در کنسولیک زیاد میشود) در بافر هم که کامند وارد شده ذخیره میشود.

خروجی دستورهای `layout src` و `layout asm` در TUI چیست؟

در GDB، حالت TUI Text User Interface یک محیط کاربری مبتنی بر متن است که امکان مشاهدهی همزمان کد منبع و دیس اسمبلی (`assembly`) را در کنار دستورات GDB فراهم می‌کند. دو دستور `layout src` و `layout asm` در TUI به شما کمک می‌کنند تا به این نماها دسترسی پیدا کنید

زمانی که دستور `layout src` را اجرا می‌کنید، یک پنجره در TUI باز می‌شود که در آن می‌توانید کد منبع (فایل‌های `.cpp`، `c` و غیره) برنامه را همراه با شماره خطوط مشاهده کنید. این نمایش به شما امکان می‌دهد که دقیقا ببینید برنامه در کدام خط از کد منبع متوقف شده است و شما را قادر می‌سازد که کدهای خود را در حین اجرای دستورات GDB مرور کنید.

زمانی که دستور `layout asm` را اجرا می‌کنید، GDB یک پنجره در TUI باز می‌کند که در آن می‌توانید دیس اسمبلی (`assembly`) دستورات پردازشی برنامه را مشاهده کنید. این نمایش دستورات اسمبلی که CPU در حال اجرای آن‌هاست را به شما نشان می‌دهد. می‌توانید کد ماشین و دستورات اسمبلی مربوطه را ببینید و بفهمید که برنامه در سطح پایین چگونه عمل می‌کند.

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

میتوان استفاده کرد. هر دوی این دستورها میتوانند یک عدد به عنوان `down` و `up` از دو دستور ورودی بگیرند (که به صورت پیشفرض ۱ است) که میتوان با استفاده از آن چندین لایه حرکت کرد.

1) سه وظیفه اصلی سیستم عامل را نام ببرید.

1. مدیریت منابع (Resource Management):

سیستم عامل وظیفه مدیریت و تخصیص منابع سخت افزاری مانند پردازنده، حافظه، دستگاه‌های ورودی/خروجی و ذخیره سازی را دارد. این مدیریت شامل تخصیص منابع به فرآیندها و برنامه‌ها و اطمینان از استفاده بهینه از این منابع است.

2. مدیریت فرآیندها (Process Management):

سیستم‌عامل وظیفه مدیریت ایجاد، زمان‌بندی و پایان‌دهی به فرآیندها را دارد. این مدیریت شامل زمان‌بندی اجرای فرآیندها روی پردازنده، تخصیص منابع به هر فرآیند و هماهنگی بین فرآیندهای مختلف است.

3. مدیریت فایل‌ها و سیستم فایل (File and File System Management):

سیستم‌عامل به کاربران و برنامه‌ها امکان دسترسی و مدیریت فایل‌ها و داده‌ها را می‌دهد. این وظیفه شامل سازماندهی، ذخیره‌سازی، بازیابی و دسترسی به داده‌ها روی دستگاه‌های ذخیره‌سازی مانند دیسک‌های سخت است. این وظایف اصلی به سیستم‌عامل اجازه می‌دهد تا به‌صورت موثر و کارآمد بین کاربر، سخت‌افزار و برنامه‌ها واسطه‌گری کند.

2) فایل‌های اصلی سیستم عامل xv6 در صفحه کتاب xv6 لیست شده‌اند. به طور مختصر

هر گروه را توضیح دهید. نام پوشه اصلی فایل‌های هسته سیستم عامل و **Header**

Files و فایل سیستم در سیستم عامل لینوکس چیست؟ در مورد محتویات آن مختصراً

توضیح دهید.

در سیستم‌عامل xv6، فایل‌های اصلی هسته در چند گروه مختلف سازماندهی شده‌اند. هر گروه نقش و وظایف خاصی در اجرای سیستم‌عامل دارند. این گروه‌ها شامل کدهای مرتبط با مدیریت حافظه، مدیریت فرآیندها، فایل سیستم، مدیریت دستگاه‌ها و غیره هستند.

1- **Kernel (هسته):** فایل‌های مرتبط با پیاده‌سازی هسته سیستم‌عامل و ساختارهای داده پایه که سیستم عامل برای مدیریت منابع و فرآیندها استفاده می‌کند.

- **proc.c:** مدیریت فرآیندها (تعریف ساختار فرآیند، زمان‌بندی، و سوئیچینگ بین فرآیندها)

- **trap.c:** مدیریت وقفه‌ها و استثناها (Interrupt Handling)

- **syscall.c:** پیاده‌سازی فراخوان‌های سیستمی

- **vm.c:** مدیریت حافظه مجازی

- **main.c:** نقطه شروع هسته xv6 که تنظیمات اولیه را انجام می‌دهد.

2- **File System (فایل سیستم):** این فایل‌ها مربوط به پیاده‌سازی فایل سیستم و مدیریت فایل‌ها و دایرکتوری‌ها است.

- **fs.c:** پیاده‌سازی فایل سیستم، شامل عملیات باز و بستن فایل‌ها، خواندن و نوشتن داده‌ها

- **inode.c:** مدیریت ساختار inode برای فایل‌ها و دایرکتوری‌ها

- **log.c:** سیستم ثبت وقایع برای فایل سیستم (Log-structured file system)

3- Device Drivers (درایورهای دستگاه): این فایل‌ها مربوط به تعامل با دستگاه‌های سخت‌افزاری مانند دیسک، ترمینال، و تایمر هستند.

- **console.c**: مدیریت ورودی/خروجی ترمینال
- **ide.c**: مدیریت دیسک سخت (دستگاه IDE)
- **uart.c**: مدیریت ارتباط سریال

4- Libraries (کتابخانه‌ها): این گروه شامل کتابخانه‌های پشتیبانی سیستم‌عامل برای توابع کمکی است.

- **string.c**: توابع کمکی مرتبط با مدیریت رشته‌ها
- **printf.c**: پیاده‌سازی تابع **printf**

5- User Programs (برنامه‌های کاربر): برنامه‌هایی که توسط کاربران در فضای کاربر اجرا می‌شوند. این برنامه‌ها از فراخوان‌های سیستمی برای ارتباط با هسته استفاده می‌کنند.

- **sh.c**: پیاده‌سازی شل (Shell) ساده
- **cat.c**: برنامه برای نمایش محتویات فایل‌ها
- **grep.c**: برنامه جستجوی الگو در فایل‌ها

پوشه اصلی فایل‌های هسته در لینوکس:

در سیستم‌های لینوکس، هسته سیستم‌عامل معمولاً در پوشه‌ای به نام **usr/src/linux** قرار دارد.

این پوشه شامل کد منبع هسته لینوکس، ماژول‌ها، و فایل‌های مرتبط با کامپایل هسته است و این پوشه شامل کد منبع هسته است. توسعه‌دهندگان و کاربرانی که می‌خواهند هسته را بازسازی یا ماژول‌های جدید اضافه کنند، از این مسیر استفاده می‌کنند.

پوشه Header Files (فایل‌های سرآیند):

فایل‌های سرآیند (Header Files) که توابع و ساختارهای داده را تعریف می‌کنند، در پوشه **usr/include/** قرار دارند. این پوشه شامل فایل‌های سرآیند (Header Files) است که به برنامه نویسان اجازه می‌دهد از توابع کتابخانه‌ای و سیستم‌عاملی در برنامه‌های خود استفاده کنند.

پوشه فایل سیستم (File System):

فایل‌های سیستم مرتبط با فایل سیستم در لینوکس، معمولاً در مسیر `etc/fstab/` یا `mnt/` و `media/` قرار دارند. این پوشه‌ها برای مدیریت فایل سیستم‌ها و پارتیشن‌ها استفاده می‌شوند.

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

در سیستم عامل `xv6`، اگر دستور `make -n` را اجرا کنید، خروجی‌ای شامل لیستی از دستورات نشان داده می‌شود که برای کامپایل و لینک کردن هسته به کار می‌روند. این دستورات شامل فراخوانی کامپایلر و ابزارهای ساخت مختلف هستند. فایلی که در نهایت هسته را می‌سازد:

فایل نهایی هسته `xv6` معمولاً با نام `kernel` ساخته می‌شود.

(4) در `Makefile` متغیرهایی به نام های `ULIB` و `UPROGS` تعریف شده است. کاربرد آن‌ها چیست؟

در `xv6` این دو متغیر برای تعریف برنامه‌های سطح کاربر و کتابخانه‌های آن مورد استفاده قرار می‌گیرند.

`UPROGS` همان `User Libraries` است و `ULIB` همان `User Libraries` است که به ترتیب برنامه‌های کاربر و کتابخانه‌های کاربر محسوب می‌شود. `UPROGS` برای برنامه‌های سطح کاربر مورد استفاده قرار می‌گیرد که قابلیت اجرا در `xv6` را دارند. این متغیر لیستی از این برنامه‌ها را شامل می‌شود. `ULIB` به کتابخانه‌های سطح کاربر اختصاص یافته است. و درواقع شامل تعدادی از کتابخانه‌های زبان C است. برنامه‌های سطح کاربر نیازمند این هستند که `ULIB` اجرا شود.

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

در اجرای `make qemu -n` معمولاً دو دیسک به `QEMU` معرفی می‌شوند که هر کدام حاوی یکی از خروجی‌های اصلی بیلد سیستم هستند:

1- دیسک اول - هسته سیستم عامل (Kernel Disk): این دیسک حاوی فایل هسته (kernel) سیستم عامل xv6 است که در فرآیند بیلد تولید می‌شود. در واقع، این دیسک شامل کدی است که توسط لینکر (Linker) برای ساخت هسته (با دستوراتی مثل `ld -o kernel`) ایجاد شده است. این فایل معمولاً با نام `kernel` شناخته می‌شود و شامل کدهای اجرایی هسته xv6 است.

2- دیسک دوم - فایل سیستم (File System Disk): این دیسک حاوی فایل سیستم (file system) است که برنامه‌ها و داده‌های ذخیره شده روی دیسک شبیه‌سازی شده QEMU را شامل می‌شود. معمولاً این فایل به نام `fs.img` (فایل سیستم ایمیج) شناخته می‌شود و حاوی ساختار و داده‌های فایل سیستم xv6 است. این فایل سیستم شامل: برنامه‌های کاربر (مانند `sh`, `cat`, `grep`, `init` و غیره) که بعد از بوت شدن هسته توسط کاربر فراخوانی می‌شوند. داده‌ها و فایل‌های دیگر که برای عملکرد سیستم مورد نیاز هستند.

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

دستور `objcopy` یک ابزار مهم در فرآیند بیلد (build process) سیستم عامل‌ها و برنامه‌های نرم‌افزاری است. این دستور بخشی از ابزارهای GNU binutils است و وظیفه دارد فایل‌های شیء (object files) و فایل‌های اجرایی (binary files) را تغییر داده یا به فرمت‌های مختلف تبدیل کند.

تبدیل فرمت فایل‌های اجرایی (Executable Conversion): می‌تواند `objcopy` محتوای فایل‌های اجرایی (ELF (Executable and Linkable Format را به فرمت خام (binary) تبدیل کند که برای شبیه‌سازهایی مانند QEMU قابل استفاده است.

حذف بخش‌های غیرضروری (Stripping Sections): مکان حذف بخش‌های غیرضروری فایل‌های شیء یا اجرایی، مثل بخش‌های اشکال زدایی (debugging sections)، را دارد. این کار حجم فایل اجرایی نهایی را کاهش داده و باعث بهینه شدن آن می‌شود. برای مثال، در xv6 ممکن است از این دستور برای حذف اطلاعات اضافی از هسته سیستم عامل استفاده شود تا هسته نهایی کوچکتر و سبک‌تر شود.

در پروژه‌هایی مانند xv6، استفاده از دستور `objcopy` برای تبدیل فایل اجرایی هسته (kernel) به فرمت باینری خام ضروری است تا شبیه‌ساز QEMU بتواند آن را به‌طور مستقیم بارگذاری کند. همچنین، این دستور در بهینه‌سازی و کوچک‌سازی فایل‌ها، حذف اطلاعات غیرضروری و تبدیل به فرمت‌های قابل استفاده در محیط‌های خاص کمک می‌کند.

13) کد `bootmain.c` هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در

آدرس `0x100000` قرار می دهد. علت انتخاب این آدرس چیست ؟

فضای خالی پایین تر از این آدرس: آدرس های پایین تر از `0x100000` (یک مگابایت) معمولاً برای بوت لودر و برنامه های اولیه مورد نیاز سیستم (مانند بخش های BIOS یا bootblock) رزرو شده اند. این فضای اولیه در سیستم های مبتنی بر معماری x86 به صورت خاص برای راه اندازی سیستم و پردازش های ابتدایی نگه داشته می شود. بنابراین، بارگذاری هسته بالاتر از این فضا انجام می شود تا با بوت لودر تداخلی نداشته باشد.

مدیریت ساده تر حافظه: انتخاب آدرس `0x100000` (که برابر با یک مگابایت است) باعث می شود که هسته در فضایی جدا و متمایز از برنامه های بوت لودر و فضای ابتدایی حافظه (first 1MB) قرار گیرد. این تفکیک به مدیریت بهتر حافظه توسط سیستم عامل کمک می کند.

سازگاری با استانداردهای معماری x86: آدرس `0x100000` یک مکان رایج برای بارگذاری هسته در سیستم های مبتنی بر معماری x86 است. این آدرس در بسیاری از سیستم عامل های قدیمی تر و حتی سیستم عامل های جدید مانند Linux به عنوان محل بارگذاری هسته استفاده می شود، زیرا از معماری حافظه x86 پیروی می کند که اجازه می دهد بوت لودر در فضای زیر یک مگابایت کار کند و سپس هسته در آدرس های بالاتر قرار گیرد.

استفاده از ساختار فایل های ELF: هسته سیستم عامل معمولاً در قالب یک فایل ELF (Executable and Linkable Format) ذخیره می شود. در این فرمت، هدر فایل ELF توسط بوت لودر خوانده می شود و محتوای آن به مکان مشخصی در حافظه (که در اینجا `0x100000` است) کپی می شود. این آدرس هنگام لینک کردن هسته توسط فایل `kernel.ld` تعیین می شود.

18). علاوه بر صفحه بندی در حد ابتدایی از قطعه بندی به منظور حفاظت هسته استفاده خواهد شد. این عملیات توسط `seginit` انجام می گردد. همان طور که ذکر شد، ترجمه قطعه تأثیری بر ترجمه آدرس منطقی نمی گذارد. زیرا تمامی قطعه ها اعم از کد و داده روی یکدیگر می افتند. با این حال برای کد و داده های سطح کاربر پرچم `SEG_USER` تنظیم شده است. چرا؟ (راهنمایی: علت مربوط به ماهیت دستورالعمل ها و نه آدرس است)

محافظت از هسته سیستم عامل:

یکی از اهداف اصلی سیستم عامل، حفاظت از هسته است تا دستورات کاربر نتوانند به حافظه هسته دسترسی داشته باشند یا کدهای هسته را تغییر دهند. با تنظیم پرچم **SEG_USER**، می توان مطمئن شد که دستورات کاربر فقط به بخش های مربوط به فضای کاربر دسترسی دارند و نمی توانند به بخش هایی از حافظه که به هسته اختصاص دارد، دسترسی پیدا کنند. این مکانیزم از اجرای کدهای خطرناک یا مخرب جلوگیری می کند.

تفکیک دسترسی های کد و داده های کاربر از هسته:

در معماری x86، بخش های مختلف حافظه (مانند کد و داده) می توانند با استفاده از پرچم های مختلف محافظت شوند. با تنظیم **SEG_USER**، سیستم مشخص می کند که این بخش ها به سطح کاربر تعلق دارند و اگر پردازنده بخواهد دستوری را از فضای کاربر اجرا کند، باید از محدودیت های دسترسی تبعیت کند. به همین دلیل، دسترسی های سطح کاربر برای داده ها و دستورالعمل ها تنظیم می شوند.

اجرای امن دستورات:

علت اصلی که در سؤال به آن اشاره شده است، به ماهیت دستورالعمل ها برمی گردد. در سیستم های x86، برخی دستورالعمل ها تنها در حالت هسته (kernel mode) قابل اجرا هستند. اگر **SEG_USER** تنظیم نشده باشد، دستورات کاربر می توانند به طور بالقوه به این دستورات حساس دسترسی پیدا کنند، که می تواند امنیت سیستم را به خطر بیاندازد. با تنظیم این پرچم، پردازنده از اجرای دستورات حساس و سیستمی توسط کدهای کاربر جلوگیری می کند.

19) جهت نگهداری اطلاعات مدیریتی برنامه های سطح کاربر ساختاری تحت عنوان struct proc خط 2336 ارائه شده است. اجزای آن را توضیح داده و ساختار معادل آن در سیستم عامل لینوکس را بیابید.

Pgdir : پوینتر متعلق به page table

Name : نام پردازنده مورد استفاده

SZ: سایز حافظه پردازنده

Kstack : پایین استک کرنل که در پردازنده وجود دارد

State: وضعیت پردازنده

Pid: عدد اختصاص داده شده به این پردازنده

Tf: چارچوب trap interrupt برای فراخوانی سیستمی فعلی

Context: برای context switching نگهداری شده است

Chan: در صورت صفر بودن به این معناست که پردازنده موقتاً غیر فعال است.

Killed: اگر صفر نباشد به معنای kill شدن پردازنده های پردازنده است

Ofile: فایل های باز شده

Cwd: نمایانگر پوشه کنونی

ساختار معادل این استراکت در کرنل لینوکس به نام task_struct است.

23) کدام بخش از آماده سازی سیستم، بین تمامی هسته های پردازنده مشترک و کدام بخش اختصاصی است؟ از هر کدام یک مورد را با ذکر دلیل توضیح دهید. زمان بند روی کدام هسته اجرا میشود؟

توابعی که در تابع main وجود دارند به صورت اختصاصی توسط اولین هسته استفاده میشوند، اما توابعی که در تابع npmain() استفاده شده اند به صورت مشترک توسط تمامی هسته ها استفاده میشوند، مانند تابع scheduler() در تابع mpmain صدا زده میشود که این تابع بین تمامی هسته ها مشترک است.

کار هسته اول بوت کردن است هسته اول با کد entry.S به main در فایل main.c میرود همه تابع های آماده سازی سیستم در تابع main صدا زده میشوند هسته های دیگر نیز وارد تابع mpenter میشود.