

# آزمایشگاه سیستم عامل

## گزارش کار پروژه یک

اعضای گروه:

810101604	امیرحسین عارف زاده
810101536	مهدی نایینی
810101413	کیارش خراسانی

---

**Repository :** <https://github.com/Amir-rfz/OS-Lab>

**Latest Commit :** ed2c402c6c2006a1ce6ac8836e7d5580c718aa62

---

## آشنایی با سیستم عامل xv6

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

سیستم عامل xv6 یک سیستم عامل آموزشی مبتنی بر Unix است که بر اساس مستندات آن، نسخه‌ای جدید از Unix V6 محسوب می‌شود. این سیستم عامل برای پردازنده های معماری x86 طراحی و پیاده سازی شده است. به منظور تأیید این ادعا می‌توان به فایل `x86.h` در کدهای xv6 اشاره کرد که از دستورات ویژه پردازنده های x86 استفاده می‌کند. علاوه بر این، فایل `mmu.h` نیز به وضوح نشان می‌دهد که xv6 از واحد مدیریت حافظه (Memory Management Unit) پردازنده های x86 بهره می‌برد.

یکی دیگر از نکات مهم درباره xv6، معماری هسته یکپارچه (monolithic kernel) آن است. به این معنا که تمام اجزای سیستم عامل، از جمله system call ها، در حالت سوپروایزر (kernel mode) اجرا می‌شوند. در مستندات xv6 نیز به این موضوع اشاره شده که تمامی فراخوان های سیستمی در این سیستم عامل تحت حالت هسته (kernel mode) اجرا می شوند، که کنترل کامل هسته بر روی منابع سخت افزاری را تضمین می‌کند.

2. یک پردازنده در سیستم عامل xv6 از چه بخش هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازنده های مختلف اختصاص می دهد؟

هر پردازنده در سیستم عامل xv6 از دو بخش User-space memory و Per-process state private to the kernel تشکیل شده است.

بخش User-space memory شامل 3 بخش است:

- دستورات (instructions): کد برنامه ای است که پردازنده در حال اجرا است و اطلاعات شامل متغیرها، ثابت ها و بقیه اطلاعاتی است که توسط برنامه استفاده می شود
- پشته (stack): قسمتی از حافظه است که برای فراخوانی ها و کنترل متغیرهای محلی استفاده می شود
- اطلاعات (data)

وضعیت خصوصی هر فرآیند در هسته (Per-process state private to the kernel) شامل مجموعه ای از اطلاعات و ساختارهای داده ای است که هسته برای مدیریت و کنترل اجرای فرآیندها نگهداری می کند. این داده ها شامل مواردی مانند شناسه یکتای فرآیند (PID) می شود که به هر فرآیند تعلق می گیرد و آن را از سایر فرآیندها متمایز می کند. همچنین، وضعیت فعلی فرآیند (مانند اجرا، انتظار، یا متوقف) و اولویت آن نیز در این مجموعه نگهداری می شود. به علاوه، شماره گذاری فایل های باز و سایر ساختارهای داده ای مرتبط با هسته نیز جزء این اطلاعات هستند. هسته سیستم عامل مسئول مدیریت این داده ها است و فرآیندها نمی توانند به طور مستقیم به این اطلاعات دسترسی داشته باشند یا آن ها را تغییر دهند.

نحوه ی اختصاص پردازنده به پردازنده های مختلف در xv6 به صورت share-time می باشد. به این صورت که به صورت متوالی و بدون متوجه شدن کاربر، پردازنده های در دسترس را به پردازنده های در دست اجرا اختصاص می دهد و پس از مدتی کوتاه، این پردازنده ها را به پردازنده های دیگری که منتظر اجرا شدن هستند، اختصاص می دهد. و همین روند ادامه پیدا می کند. به صورتی که پردازنده ها به صورت تقریباً موازی با یکدیگر اجرا می شوند. زمانی که یک پردازنده در حال اجرا نیست، سیستم عامل رجیسترهای CPU که حاوی موارد مورد نیاز توسط آن پردازنده بود را در حافظه ذخیره می کند و دوباره زمانی که نوبت به اجرای دوباره ی این پردازنده رسید، آن ها را بازیابی می کند.

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

در سیستم عامل های مبتنی بر یونیکس، هر فرآیند دارای جدولی به نام جدول فایل های باز (file descriptor table) است که در آن فایل هایی که فرآیند باز کرده است، نگهداری می‌شوند. هر فایل دیسکریپتور (fd) عدد صحیحی است که به عنوان نماینده ای برای دسترسی به فایل ها، پایپ ها یا دستگاه های مختلف استفاده می‌شود. هنگامی که یک فایل باز می‌شود، سیستم عامل کمترین فایل دیسکریپتور آزاد را به آن فایل تخصیص می‌دهد و از آن برای خواندن و نوشتن داده ها استفاده می‌شود. برای مثال، فایل دیسکریپتورهای 0، 1 و 2 به ترتیب برای ورودی استاندارد (stdin)، خروجی استاندارد (stdout) و خطای استاندارد (stderr) رزرو شده‌اند.

این روش به فرآیندها اجازه می‌دهد که بدون توجه به نوع فایل (مانند فایل معمولی، پایپ یا دستگاه)، با استفاده از یک واسط واحد یعنی فایل دیسکریپتور با منابع مختلف تعامل کنند. در سیستم عامل xv6، پایپ‌ها امکان ارتباط بین فرآیندها را فراهم می‌کنند و این قابلیت به آن‌ها اجازه می‌دهد تا داده‌ها را بین یکدیگر منتقل کنند. برای ایجاد پایپ، از فراخوان سیستمی pipe استفاده می‌شود که دو فایل دیسکریپتور برمی‌گرداند: یکی برای خواندن (read end) و دیگری برای نوشتن (write end). علاوه بر این، پایپ‌ها به همگام سازی اجرای فرآیندها کمک می‌کنند.

در xv6، پایپ ها به صورت مسدودکننده عمل می‌کنند؛ به این معنی که اگر فرآیندی قصد داشته باشد از پایپی که داده‌ای در آن وجود ندارد بخواند، منتظر می‌ماند تا داده وارد شود. به همین ترتیب، اگر فرآیندی بخواهد در پایپی که پر است بنویسد، تا آزاد شدن فضای کافی، مسدود می‌شود. این ویژگی باعث همگام سازی بهتر فرآیندها می‌شود.

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

• فراخوانی سیستمی fork :

تابع fork برای ایجاد یک پردازش جدید استفاده می‌شود. این تابع، نسخه‌ای از پردازشی که آن را فراخوانی کرده (پردازش والد) تولید می‌کند. منظور از کپی کردن این است که داده‌ها و دستورات پردازش والد (parent) در حافظه پردازش جدید (child) کپی می‌شوند. اگرچه در لحظه ایجاد پردازش فرزند، داده‌ها مانند متغیرها و رجیسترها با پردازش والد یکسان هستند، اما این دو پردازش حافظه‌ای مجزا خواهند داشت. به همین دلیل، تغییر یک متغیر در پردازش والد، هیچ تأثیری روی پردازش فرزند نخواهد داشت و بالعکس. پس از ایجاد پردازش فرزند، پردازش والد به تابع fork بازمی‌گردد، که این امکان را فراهم می‌کند تا هر دو پردازش به طور همزمان اجرا شوند.

مقدار بازگشتی از تابع fork، شناسه پردازش (PID) فرزند است. این مقدار در پردازش والد به تابع فراخوان بازمی‌گردد. نقطه شروع پردازش فرزند نیز همان جایی است که تابع fork در آن فراخوانی شده است، با این تفاوت که در پردازش فرزند مقدار بازگشتی fork برابر 0 است.

مقدار بازگشتی fork می‌تواند سه حالت داشته باشد که هرکدام نشان‌دهنده وضعیت خاصی هستند:

- (1)  $pid = 0$  : در پردازش فرزند هستیم.
- (2)  $pid > 0$  : در پردازش پدر هستیم و مقدار pid درواقع شناسه پردازش فرزند است.
- (3)  $pid < 0$  : در هنگام اجرای پردازش فرزند به اروری خورده ایم و ایجاد نشده است.

قطعه کد زیر را بعنوان مثال در نظر بگیرید:

```
int pid = fork();
if(pid > 0){
    printf("parent: child=%d\n", pid);
    pid = wait();
    printf("child %d is done\n", pid);
} else if(pid == 0){
    printf("child: exiting\n");
    exit();
} else {
    printf("fork error\n");
}
```

فراخوانی سیستمی wait باعث میشود پردازش پدر منتظر پایان یافتن پردازش فرزند می‌شود و سپس کار خود را ادامه می‌دهد. خروجی این تابع، pid پردازش پایان یافته است. اگر پردازش فعلی هیچ پردازش فرزندی نداشته باشد، خروجی این تابع 1- خواهد بود. در حقیقت اتفاقی که در سیستم عامل می‌افتد تا حدی در تکه کد بالا توضیح داده شده است.

#### • فراخوانی سیستمی exec :

تابع exec حافظه پردازش فعلی را با یک حافظه جدید که در آن یک برنامه با فایل ELF لود شده است، جایگزین می‌کند، در حالی که جدول فایل های باز (file table) پردازش اولیه حفظ می‌شود. اگر exec موفقیت آمیز باشد، دیگر به پردازش قبلی باز نخواهد گشت و برنامه جدید در حافظه جدید اجرا می‌شود. برنامه جدید نیز در یک نقطه خاص با استفاده از تابع exit اجرای خود را خاتمه می‌دهد. این تابع دو ورودی می‌پذیرد: ورودی اول نام فایل برنامه است و ورودی دوم آرگومان های برنامه هستند.

مثال زیر نحوه استفاده از این تابع را نشان می‌دهد:

```
char* args[] = { "ls", "-l", "/home", NULL }; // NULL برای خاتمه آرگومان ها ضروری است
```

```
exec("/bin/ls", args);
```

```
printf("Exec failed!\n");
```

یکی از دلایل اصلی عدم ادغام توابع fork و exec جلوگیری از ساخت پردازش های بی مصرف است که بلافاصله توسط exec جایگزین می‌شوند. در شرایط عادی، این دو تابع به طور متوالی فراخوانی می‌شوند. اگر این دو ادغام شوند، علاوه بر ایجاد پردازش های غیرضروری و اشغال حافظه بیشتر، مدیریت آرگومان های توابع نیز پیچیده تر خواهد شد.

مزیت مهم جدا نگه داشتن این دو تابع در شرایطی مانند تغییر مسیر ورودی و خروجی (I/O redirection) قابل مشاهده است. زمانی که کاربر یک برنامه را در shell اجرا می‌کند، مراحل پشت صحنه به این شکل است:

1. ابتدا دستور تایپ شده توسط کاربر خوانده می‌شود.
2. با استفاده از تابع fork، یک پردازش جدید ایجاد می‌شود.
3. در پردازش فرزند، تابع exec فراخوانی می‌شود تا برنامه درخواست شده توسط کاربر اجرا شود.
4. پردازش والد منتظر اتمام پردازش فرزند می‌ماند (تابع wait).
5. پس از اتمام پردازش فرزند، shell به حالت آماده باش برای دریافت دستور جدید بازمی‌گردد.

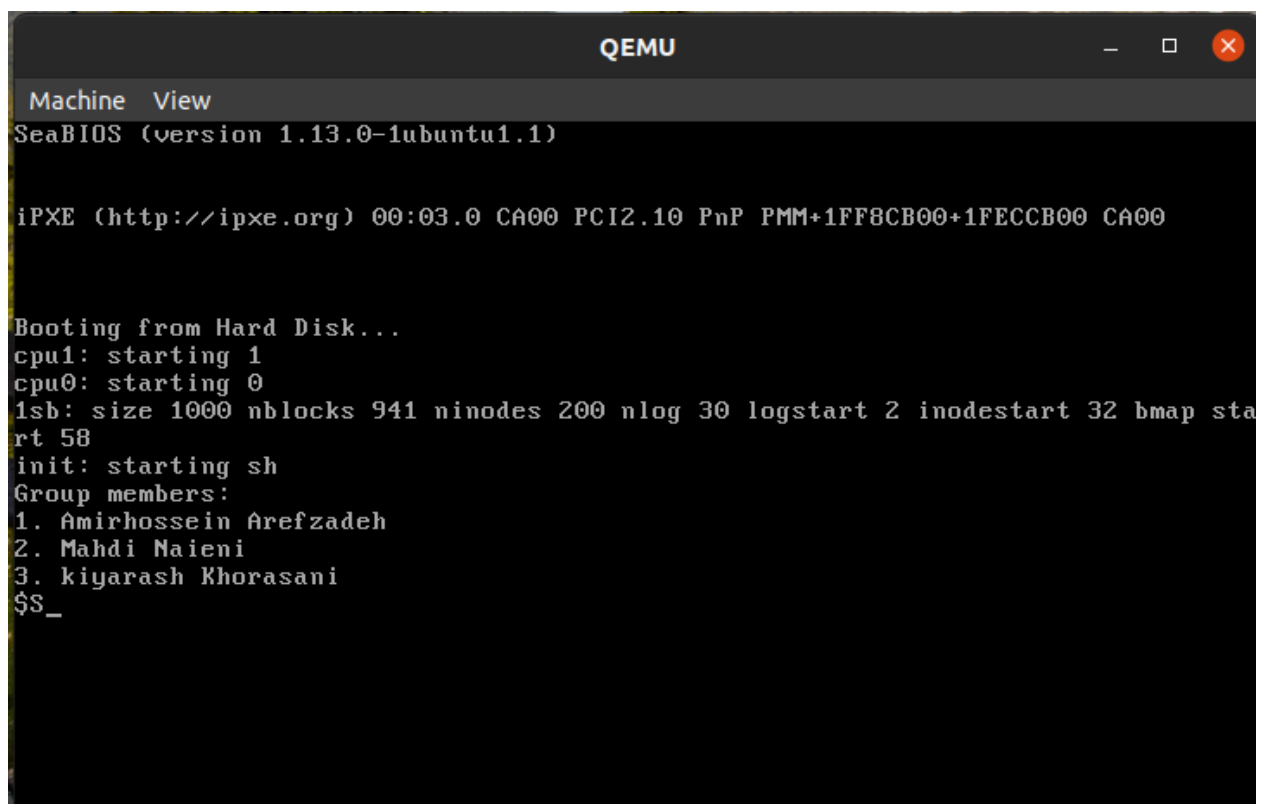
اکنون مشخص است چرا این دو فراخوان سیستمی ادغام نشده اند. به این دلیل که shell می‌تواند ابتدا یک پردازش فرزند را با استفاده از fork ایجاد کند، سپس با استفاده از توابع open، dup، close و exec تغییراتی در ورودی‌ها و خروجی‌های استاندارد فایل دیسکریپتورها اعمال کند و در نهایت exec را فراخوانی کند. در این روش، هیچ تغییری در برنامه در حال اجرا نیاز نیست. اگر این دو تابع ادغام شده بودند، احتمالاً shell به طراحی پیچیده تری نیاز داشت تا تغییر مسیر ورودی/خروجی را مدیریت کند یا برنامه‌های اجراشده باید خودشان متوجه نحوه مدیریت این تغییرات می‌شدند.

## اضافه کردن یک متن به Boot Message

برای نشان دادن نام اعضای گروه پس از بوت شدن سیستم عامل کافیهست که عناوین و متن مربوطه را با دستور printf به فایل init.c اضافه کنیم:

```
for(;;){
    printf(1, "init: starting sh\n");
    printf(1, "Group members:\n");
    printf(1, "1. Amirhossein Arefzadeh\n");
    printf(1, "2. Mahdi Naieni\n");
    printf(1, "3. kiyarash Khorasani\n");
    pid = fork();
    if(pid < 0){
        printf(1, "init: fork failed\n");
        exit();
    }
    if(pid == 0){
        exec("sh", argv);
        printf(1, "init: exec sh failed\n");
        exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
        printf(1, "zombie!\n");
}
```

نمونه خروجی:



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

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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
lsb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
rt 58
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiyarash Khorasani
$$_
```

## اضافه کردن چند قابلیت به کنسول xv6

### 1. اضافه کردن دستور ← و →:

برای پیاده سازی این دستور در کد مربوط به سیستم عامل xv6 یک متغیر global به نام num\_of\_backs قرار داده شده است که این متغیر نشان دهنده تعداد کاراکترهایی است که به عقب بازگشته ایم:

```
static void backwardCursor(){
    int pos;

    // get cursor position
    outb(CRTPORT, 14);
    pos = inb(CRTPORT+1) << 8;
    outb(CRTPORT, 15);
    pos |= inb(CRTPORT+1);

    // move back
    if(crt[pos - 2] != ('$' | 0x0700))
        pos--;

    // reset cursor
    outb(CRTPORT, 14);
    outb(CRTPORT+1, pos>>8);
    outb(CRTPORT, 15);
    outb(CRTPORT+1, pos);
}
```

```
static void forwardCursor(){
    int pos;

    // get cursor position
    outb(CRTPORT, 14);
    pos = inb(CRTPORT+1) << 8;
    outb(CRTPORT, 15);
    pos |= inb(CRTPORT+1);

    // move forward
    pos++;

    // reset cursor
    outb(CRTPORT, 14);
    outb(CRTPORT+1, pos>>8);
    outb(CRTPORT, 15);
    outb(CRTPORT+1, pos);
}
```

همچنین قسمت مربوط به BACKSPACE هم که مرتبط با دستور ← و → هست باید اصلاح شود.

```
else if(c == BACKSPACE){
    for (int i = pos - 1 ; i < pos + num_of_backs ; i++)
        crt[i] = crt[i + 1];

    if(pos > 0) --pos;
}
else{
    for (int i = pos + num_of_backs; i > pos ; i--)
        crt[i] = crt[i - 1];
    crt[pos++] = (c&0xff) | 0x0700; // black on white
}
```



در ادامه دستورات `shiftright()` و `shiftleft()` را برای اینکه بافر به درستی کاراکتر ها را در جای خود ذخیره کند، اضافه می می کنیم تا به هنگام زدن دستورات `→` و `←` بافر به مشکل برنخورد.

```
static void shiftright(char *buf)
{
    for (int i = input.e; i > input.e - num_of_backs; i--)
    {
        buf[(i) % INPUT_BUF] = buf[(i - 1) % INPUT_BUF]; // Shift elements to the right
    }
}

static void shiftleft(char *buf)
{
    for (int i = input.e - num_of_backs - 1; i < input.e; i++)
    {
        buf[(i) % INPUT_BUF] = buf[(i + 1) % INPUT_BUF]; // Shift elements to the right
    }
    input.buf[input.e] = ' ';
}
```

در پایان نیز دستورات `→` و `←` را به برنامه خود در بخش `consoleintr()` اضافه می کنیم.

```
case KEY_LF: // Cursor Backward
    if((input.e - num_of_backs) > input.w){
        backwardCursor();
        num_of_backs++;
    }
    if((saved_input.e - num_of_backs_saved) > saved_input.w && is_copy == 1){
        num_of_backs_saved++;
    }
    break;
case KEY_RT: // Cursor Backward
    if(num_of_backs > 0){
        forwardCursor();
        num_of_backs--;
    }
    if(num_of_backs_saved > 0 && is_copy == 1){
        num_of_backs_saved--;
    }
    break;
```

نمونه خروجی:

```
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ echo 123 hello!
```

## 2. اضافه کردن دستور history:

برای اضافه کردن history یک struct به نام inputs می سازیم و در آن ده input اخیر را با استفاده از کدی که در بخش زیر آمده ذخیره میکنیم.

```
struct {
    struct Input history[HISTORY_SIZE];
    int curent;
    int end;
    int size;
    int match;
} inputs;
```

با استفاده از کد زیر هنگامی که کاربر enter را می زند، دستور اخیر در inputs ذخیره می شود.

```
if((c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF) && 1){
    inputs.history[inputs.end++ % HISTORY_SIZE] = input;
    inputs.curent = inputs.end;
    if (inputs.size < 10)
        inputs.size++;
    input.w = input.e;
    wakeup(&input.r);
}
```

همچنین با استفاده از کد زیر می توانیم input را نمایش دهیم.

```
void display_command(){
    for(int i = (input.w); i < input.e; i++){
        consputc(input.buf[i]);
    }
}
```

سپس هر زمان که دستور history وارد شد، آنگاه ده دستور اخیری که ذخیره کرده بودیم را به ترتیب با استفاده از display\_command() نمایش می دهیم.

```
if(c == '\n') {
    num_of_backs = 0;
    num_of_backs_saved = 0;
    match_history = 0;
    if((input.e - input.w) >= 7){
        if (match_history == 0) {
            match_history = 1;
            char *history_cmd = "history";
            for(int i=input.w, j=0; i< input.e; i++, j++) {
                if(input.buf[i] != history_cmd[j] && j <= 6)
                    match_history = 0;
                else if (j >= 7 && input.buf[i] != ' ')
                    match_history = 0;
            }
            copy_input = input;
            if(match_history == 1){
                if (inputs.size != 0)
                    consputc('\n');
                for(int i=0; i<inputs.size; i++){
                    input = inputs.history[(inputs.end - inputs.size + i) % HISTORY_SIZE];
                    input.buf[--input.e] = '\0';
                    display_command();
                    if (i != inputs.size - 1)
                        consputc('\n');
                }
                input = copy_input;
            }
        }
    }
}
```

نمونه خروجی:

```
$ history
echo 1
echo 2
echo 3
echo 4
echo 5
echo 6
echo 7
echo 8
echo 9
echo 10
$ _
```

### 3. اضافه کردن دستور arrow up & arrow down:

برای اینکه بتوانیم دستور جدید را نمایش دهیم باید ابتدا دستور قبلی را پاک کنیم، برای انجام این کار از کد زیر برای پاک کردن خط استفاده می کنیم.

```
void display_clear()
{
    for (int i = 0; i < num_of_backs; i++)
        forwardCursor();
    num_of_backs = 0;
    int end = input.e;
    while (end != input.w && input.buf[(end - 1) % INPUT_BUF] != '\n')
    {
        end--;
        consputc(BACKSPACE);
    }
}
```

پس از اینکه دستور قبلی را پاک کردیم با استفاده از تابع های arrowup() و arrowdown() می توانیم در history ای که از قبل ساخته ایم جابجا شویم و دستور مورد نظر را با استفاده از تابع display\_command() نمایش دهیم.

```
static void arrowup()
{
    if (inputs.curent == inputs.end)
    {
        inputs.history[inputs.end % HISTORY_SIZE] = input;
    }
    display_clear();
    input = inputs.history[--inputs.curent % HISTORY_SIZE];
    input.buf[--input.e] = '\0';
    display_command();
}

static void arrowdown()
{
    if (inputs.curent < inputs.end)
    {
        display_clear();
        input = inputs.history[++inputs.curent % HISTORY_SIZE];
        if (input.e != input.w && inputs.curent != inputs.end)
            input.buf[--input.e] = '\0';
        display_command();
    }
}
```

در نهایت تابع های `arrowup()` و `arrowdown()` را به برنامه خود در بخش `consoleintr()` اضافه میکنیم.

```
switch(c){  
case KEY_UP:  
    if (inputs.size && inputs.end - inputs.curent < inputs.size)  
        arrowup();  
    break;  
case KEY_DN:  
    arrowdown();  
    break;  
}
```

نمونه خروجی:

```
$ history  
echo 1  
echo 2  
echo 3  
echo 4  
echo 5  
echo 6  
echo 7  
echo 8  
echo 9  
echo 10  
$ echo 6_
```

#### 4. اضافه کردن دستور Ctrl+S و Ctrl+F :

وقتی که Ctrl+S وارد شد برای اینکه بتوانیم کاراکتر های وارد شده را ذخیره کنیم یک saved\_input مانند input می سازیم تا کاراکتر های وارد شده را در بافر آن ذخیره کنیم و تابع های مرتبط با input را برای saved\_input هم پیاده سازی می کنیم. علاوه بر آن یک متغیر global به نام num\_of\_backs\_saved تعریف می کنیم که نشان دهنده تعداد کاراکترهایی است که به عقب برگشته ایم. همچنین متغیر های is\_copy و copy\_is\_end را برای تشخیص زمان شروع و پایان عملیات کپی اضافه می کنیم. در ادامه کد مربوط به دستور Ctrl+S و Ctrl+F آمده است.

```
case C('S'):  
    saved_input.e = saved_input.w;  
    num_of_backs_saved = 0;  
    is_copy = 1;  
    copy_is_end = 0;  
    break;  
case C('F'):  
    if (copy_is_end == 0) {  
        copy_is_end = 1;  
        is_copy = 0;  
        break;  
    }  
    is_copy = 0;  
    display_saved_command();  
    for(int i = (saved_input.w); i < saved_input.e; i++){  
        input.buf[(input.e++ - num_of_backs) % INPUT_BUF] = saved_input.buf[i];  
    }  
    break;
```

با استفاده از کد زیر کاراکتر های ورودی را به هنگام کپی در در بافر saved\_input ذخیره می کنیم.

```
if(is_copy == 1)  
    saved_input.buf[(saved_input.e++ - num_of_backs_saved) % INPUT_BUF] = c;
```

نمونه خروجی:

```
init: starting sh  
Group members:  
1. Amirhossein Arefzadeh  
2. Mahdi Naieni  
3. kiarash Khorasani  
$ echo 123 123 123 123 123_
```

## 5. اضافه کردن دستورهایی Operators:

با استفاده از کد زیر در صورتی که رشته ی ورودی کنسول عبارتی به شکل الگوی  $NON=?$  باشد (منظور از N، عدد و منظور از 0، عملگر است) را به کل حذف کرده و پس از انجام محاسبات، حاصل عددی آن را در همان محل جایگزین می کند. برای این تمرین ما عملگر های جمع، تفریق، ضرب، تقسیم و باقی مانده را تعریف کرده ایم و در صورت نیاز می توان عملگر های بیشتری را نیز اضافه کرد.

```
int get_num(int index_num, int index_end ) {
    int num = 0;
    for (int i = index_num ; i <= index_end ; i++) {
        num *= 10;
        num += (input.buf[i] - '0');
    }
    return num;
}

float get_answer(int index_num1, int index_num2, int edit_place, int is_num1_neg) {
    int num1 = get_num(index_num1, index_num2-2);
    int num2 = get_num(index_num2, edit_place-3);
    float float_ans = 0;
    if (is_num1_neg == 1)
        num1 = 0-num1;
    if (input.buf[index_num2 - 1] == '+')
        float_ans = num1 + num2;
    else if (input.buf[index_num2 - 1] == '-')
        float_ans = num1 - num2;
    else if (input.buf[index_num2 - 1] == '*')
        float_ans = num1 * num2;
    else if (input.buf[index_num2 - 1] == '%')
        float_ans = num1 % num2;
    else if (input.buf[index_num2 - 1] == '/')
        float_ans = (float)num1 / num2;
    return float_ans;
}
```

```
void remove_equation(int remove_num) {
    for(int i=0;i<remove_num;i++){
        if(input.e != input.w && input.e - input.w > num_of_backs){
            if (num_of_backs > 0)
                shiftright(input.buf);
            input.e--;
            consputc(BACKSPACE);
        }
        if(saved_input.e != saved_input.w && saved_input.e - saved_input.w > num_of_backs_saved && is_copy == 1) {
            if (num_of_backs_saved > 0)
                shiftright_saved(saved_input.buf);
            saved_input.e--;
        }
    }
}

void print_number(int ans) {
    if (ans <= 0)
        return;
    print_number(ans / 10);
    print_char('0' + (ans % 10));
}

void print_answer(float float_ans, int index_num2) {
    char float_part = '0';
    if (float_ans < 0) {
        print_char('-');
        float_ans = 0-float_ans;
    }
    if (input.buf[index_num2 - 1] == '/') {
        int temp = float_ans * 10;
        float_part = '0' + (temp % 10);
    }
    if (float_ans < 1)
        print_char('0');
    else
        print_number((int)float_ans);
    if (input.buf[index_num2 - 1] == '/') {
        print_char('.');
        print_char(float_part);
    }
}
```

```

if(input.e >= 5) {
    int edit_place = input.e - num_of_backs;
    int match_equation = 0;
    int state_machine = 0;
    int is_num1_neg = 0;
    int index_num1 = 0;
    int index_num2 = 0;

    if (is_digit_number(input.buf[edit_place-3]) && input.buf[edit_place-2] == '=' && input.buf[edit_place-1] == '?') {
        for (int i = edit_place - 3 ; i >= input.w ; i--){
            if (state_machine == 0){
                if (is_digit_number(input.buf[i]))
                    continue;
                else if (is_operator(input.buf[i])){
                    state_machine = 1;
                    index_num2 = i+1;
                }
                else
                    break;
            }
            else if (state_machine == 1 && is_digit_number(input.buf[index_num2-2])){
                match_equation = 1;
                index_num1 = i;
                if (is_digit_number(input.buf[i]))
                    continue;
                else if (input.buf[i] == '-') {
                    index_num1 = i+1;
                    is_num1_neg = 1;
                    break;
                }
                else {
                    index_num1 = i+1;
                    break;
                }
            }
        }
    }

    if (match_equation == 1) {
        float float_ans = get_answer(index_num1, index_num2, edit_place, is_num1_neg);
        int num_remove = edit_place - index_num1 + is_num1_neg;
        remove_equation(num_remove);
        print_answer(float_ans, index_num2);
    }
}

```

نمونه خروجی:

```

init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ ans = 125*450=_

```

```

init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ ans = 56250_

```



## اجرا و پیاده سازی یک برنامه سطح کاربر

برای این کار دو برنامه به زبان C به نام های encode.c و decode.c میسازیم و کد خود را در آن جا مینویسیم. سپس این برنامه را باید به برنامه های سطح کاربر اضافه کنیم که برای اینکار نیاز است که تغییریاتی در MakeFile اعمال کنیم:

key = mod 26 جمع دو رقم سمت راست شماره دانشجویی اعضای گروه )

$$\text{key} = 4 + 36 + 13 \bmod 26 = 1$$

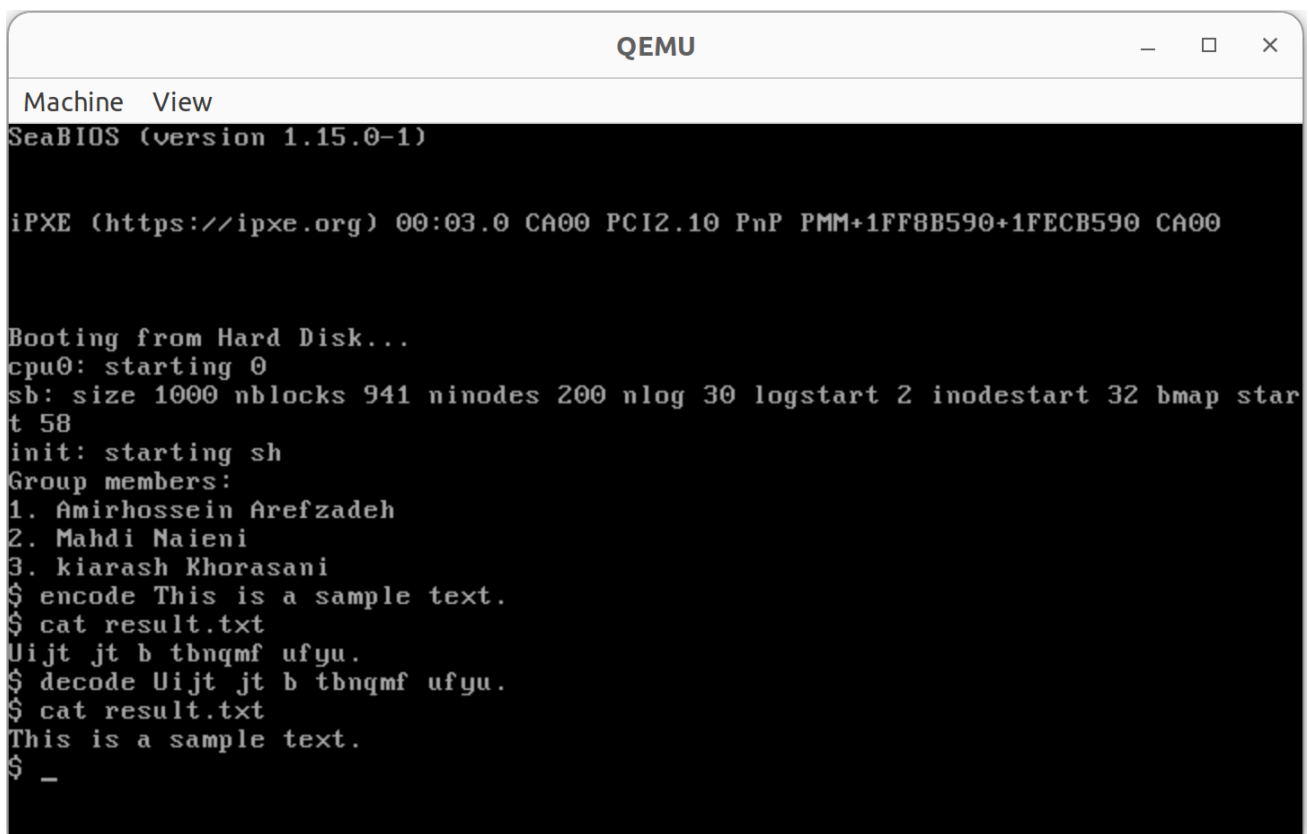
```
C encode.c > ...
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4  #include "fcntl.h"
5
6  const int KEY = 1;
7
8  void CalculateEncode(const char *word1, char *result) {
9      int i = 0;
10     while (word1[i] != '\0') {
11         char c = word1[i];
12         if (c >= 'a' && c <= 'z') {
13             result[i] = ((c - 'a' + KEY) % 26) + 'a';
14         }
15         else if (c >= 'A' && c <= 'Z') {
16             result[i] = ((c - 'A' + KEY) % 26) + 'A';
17         }
18         else {
19             result[i] = c;
20         }
21         i++;
22     }
23     result[i] = '\0'; // Null-terminate the result string
24 }
25
26 int main(int argc, char *argv[]) {
27     if (argc < 2) {
28         printf(1, "Usage: encode <text_to_encode>\n");
29         exit();
30     }
31
32     int fd = open("result.txt", O_CREATE | O_RDWR);
33     if (fd < 0) {
34         printf(1, "Can't create or open the file\n");
35         exit();
36     }
37
38     char result[512];
39
40     for (int i = 1; i < argc; i++) {
41         CalculateEncode(argv[i], result);
42         write(fd, result, strlen(result));
43
44         // Add a space between words
45         if (i < argc - 1) {
46             write(fd, " ", 1);
47         }
48     }
49
50     char diff[2];
51     diff[0] = '\n';
52     diff[1] = '\0';
53     write(fd, diff, 1);
54
55     close(fd);
56     exit();
57 }
```

```
C decode.c > ...
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4  #include "fcntl.h"
5
6  const int KEY = 1;
7
8  void CalculateDecode(const char *word1, char *result) {
9      int i = 0;
10     while (word1[i] != '\0') {
11         char c = word1[i];
12         if (c >= 'a' && c <= 'z') {
13             result[i] = ((c - 'a' - KEY + 26) % 26) + 'a';
14         }
15         else if (c >= 'A' && c <= 'Z') {
16             result[i] = ((c - 'A' - KEY + 26) % 26) + 'A';
17         }
18         else {
19             result[i] = c;
20         }
21         i++;
22     }
23     result[i] = '\0'; // Null-terminate the result string
24 }
25
26 int main(int argc, char *argv[]) {
27     if (argc < 2) {
28         printf(1, "Usage: decode <text_to_encode>\n");
29         exit();
30     }
31
32     int fd = open("result.txt", O_CREATE | O_RDWR);
33     if (fd < 0) {
34         printf(1, "Can't create or open the file\n");
35         exit();
36     }
37
38     char result[512];
39
40     for (int i = 1; i < argc; i++) {
41         CalculateDecode(argv[i], result);
42         write(fd, result, strlen(result));
43
44         // Add a space between words
45         if (i < argc - 1) {
46             write(fd, " ", 1);
47         }
48     }
49
50     char diff[2];
51     diff[0] = '\n';
52     diff[1] = '\0';
53     write(fd, diff, 1);
54
55     close(fd);
56     exit();
57 }
```

برنامه های نوشته شده را به متغیر های PROGS در MakeFile اضافه میکنیم .

```
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     _userstests\
182     _wc\
183     _zombie\
184     _encode\
185     _decode\
186
187 fs.img: mkfs README $(UPROGS)
188     ./mkfs fs.img README $(UPROGS)
189
```

نمونه اجرای دستورها با key = 1 :



```
QEMU
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 star
t 58
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ encode This is a sample text.
$ cat result.txt
Uijt jt b tbnqmf ufyu.
$ decode Uijt jt b tbnqmf ufyu.
$ cat result.txt
This is a sample text.
$ _
```

## مقدمه ای درباره سیستم عامل xv6

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

1. **process management**: سیستم عامل ساخت و خاتمه و توقف و scheduling هر process رو انجام میدهد. مطمئن میشود هر process به اندازه کافی منابع و زمان cpu دریافت میکند و انجام چند کار به صورت همروند رو مدیریت میکند. و process ها رو همگام میکند (synchronize) و ارتباط بین اونها رو برقرار میکند.

2. **memory management**: کار های مربوط به تخصیص حافظه به هر process در صورت نیاز و آزادسازی آن بعد از اتمام کار رو انجام میدهد. سعی میکند استفاده از مموری رو بهینه کند و virtual memory را مدیریت میکند. جابه جایی بین physical memory و disk storage رو انجام میدهد.

3. **I/O system management**: عملیات های I/O مثل ارتباط سخت افزار هایی مثل کیبورد، موس، پرینتر و حافظه های دستگاه رو مدیریت میکند. انتقال اطلاعات بین سخت افزار و نرم افزار را مدیریت میکند.

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 سرایند 5 و فایل سیستم در سیستم عامل لینوکس چیست؟ در مورد محتویات آن مختصراً توضیح دهید.

در xv6، فایل های اصلی هسته سیستم عامل به چند گروه دسته بندی شده اند که هر گروه مربوط به بخش های مختلف سیستم عامل است. این دسته بندی ها و توضیحات مختصری از هر گروه عبارتند از:

#### ● سرایندهای پایه:

شامل فایل های سرایند که ساختارهای داده، ثابت ها و توابع عمومی سیستم عامل را تعریف می کنند.

فایل های مرتبط: types.h, param.h, memlayout.h, defs.h.

#### ● فراخوان های سیستمی:

این گروه شامل فایل هایی است که فراخوان های سیستمی را مدیریت و پیاده سازی می کنند.

فایل های مرتبط: syscall.h, syscall.c, sysproc.c.

## • ورودی xv6:

این فایل‌ها مربوط به نقطه ورود xv6 هستند، یعنی شروع بوت سیستم عامل و اجرا شدن آن.

فایل‌های مرتبط: `entry.S, main.c`.

## • قفل‌ها:

فایل‌هایی که پیاده‌سازی قفل‌های مورد نیاز برای همزمانی بین پردازش‌ها را شامل می‌شوند.

فایل‌های مرتبط: `spinlock.h, spinlock.c`.

## • مدیریت پردازش‌ها:

مدیریت پردازش‌ها شامل مدیریت حافظه مجازی و زمان‌بندی پردازش‌ها در این گروه قرار دارد.

فایل‌های مرتبط: `vm.c, proc.h, proc.c`.

## • سیستم فایل:

این گروه شامل فایل‌هایی است که سیستم فایل و مدیریت ذخیره‌سازی دیسک را پیاده‌سازی می‌کنند.

فایل‌های مرتبط: `buf.h, fs.h, file.h, fs.c`.

## • لوله‌ها:

فایل‌هایی که ارتباط بین پردازش‌ها با استفاده از `pipe` را مدیریت می‌کنند.

فایل مرتبط: `pipe.c`.

## • عملیات‌های سطح پایین سخت‌افزاری:

این فایل‌ها مسئول مدیریت ارتباطات سخت‌افزاری و دستگاه‌ها هستند.

فایل‌های مرتبط: `mp.h, lapic.c, ioapic.c`.

## • کاربر سطحی:

این فایل ها مربوط به برنامه های سطح کاربر و نحوه ارتباط آنها با هسته سیستم عامل هستند.

فایل های مرتبط: `initcode.S`, `usys.S`, `init.c`, `sh.c`.

## • بوت لودر:

فایل هایی که مربوط به بوت شدن سیستم عامل و انتقال کنترل به هسته هستند.

فایل های مرتبط: `bootasm.S`, `bootmain.c`.

## • لینک:

این فایل ها مسئول لینک کردن بخش های مختلف هسته و سیستم عامل هستند.

فایل مرتبط: `kernel.ld`.

## نام پوشه اصلی فایل های هسته سیستم عامل:

- در سیستم عامل های مبتنی بر یونیکس مانند لینوکس، پوشه اصلی فایل های هسته سیستم عامل معمولاً در مسیر `usr/src/linux/` قرار دارد. این مسیر شامل تمامی سورس کدهای هسته سیستم عامل است.

## فایل های سرایند اصلی:

1. `types.h`: ساختارهای داده پایه مانند `pid_t`, `size_t` و غیره را تعریف می کند.
2. `param.h`: شامل تعاریف عمومی برای پیکربندی سیستم مانند اندازه های بافر، نسخه سیستم عامل و غیره است.
3. `defs.h`: شامل توابع و تعاریف اساسی سیستم است.
4. `mman.h`: شامل تعاریف و توابع مرتبط با مدیریت حافظه است.
5. `elf.h`: شامل ساختارهای مرتبط با فایل های باینری ELF است.

## فایل سیستم در سیستم عامل لینوکس:

- پوشه /etc: شامل فایل های پیکربندی سیستم است که توسط برنامه ها و سرویس های سیستم عامل استفاده می شوند.
- پوشه /proc: یک فایل سیستم مجازی است که اطلاعات لحظه ای سیستم، مانند پردازش ها، حافظه و وضعیت سخت افزار را نشان می دهد.
- پوشه /dev: شامل فایل های دستگاه ها است که ارتباط با دستگاه های سخت افزاری مانند دیسک ها و پرینترها را فراهم می کند.

این فایل ها و پوشه ها برای مدیریت هسته سیستم عامل و تعامل با سخت افزار و نرم افزارهای مختلف ضروری هستند.

## کامپایل سیستم عامل xv6

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

`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 which`

فایل های اصلی سیستم عامل `fs.img` و `xv6.img` و `kernel` است که با دستور آخر با استفاده از این فایل ها `core` سیستم عامل ساخته میشوند. که در بالا آورده شده اند.

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

`UPROGS` یا همان `user programs` شامل لیستی از برنامه های سطح کاربر است که در هنگام ساخت و کامپایل `xv6` کامپایل شده و به برنامه قابل اجرا توسط `OS` تبدیل می شوند. به طور مثال دستور `cat` یا `grep` یا `echo` عضوی از این برنامه ها است. این فایل ها عموماً در `user directory` سیستم عامل `xv6` قرار دارند

`ULIB` یا همان `user libraries` شامل تعدادی کتابخانه زبان `c` است که در اجرای برنامه ها (یا سیستم عامل) به آنها نیاز است. به همین دلیل در هنگام ساخت فایل `object` برای قسمت های زیادی به عنوان `dependency` اضافه میشود. پس این کتابخانه ها باید ساخته و به برنامه کاربرد لینک شوند تا بتوان آنها را اجرا نمود. کتابخانه ها عموماً از کدهای قابل استفاده و توابعی که در اکثر برنامه ها استفاده می شوند تشکیل شده اند و در `ulib directory` سورس کد `xv6` قرار دارند.

وقتی makefile اجرا می شود ابتدا کتابخانه های ulib کامپایل شده و به فایل آجکت تبدیل می شوند. این آجکت ها با برنامه های کاربر در uprogs لینک شده و تبدیل به فایل قابل اجرا می شوند که کاربرد میتواند در xv6 آن ها را اجرا کند.

## اجرای شبیه ساز بر روی QEMU

5. دستور `qemu make -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.)  
`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`

دیسک اول fs.img است. file system image شامل ساختار file system از جمله directories و فایل ها و metadata است. به طور کل میتوان گفت که fs.img دارای فایل هایی و directory structures است که user space رو در xv6 میسازد. در index 1 از QEMU میبینیم که Mount شده است. که شامل user program, files, و بقیه اطلاعاتی هست که جزئی از فایل سیستم است و user با آنها ارتباط برقرار میکند.

دیسک دوم xv6.img یا همان kernel image است. که شامل کد کامپایل شده کرنل xv6 است و در index 0 از QEMU می بینیم که Mount شده است. به طور کل این دیسک شامل kernel space است. که یعنی کد های هسته سیستم عامل که کار رسیدگی به process management و memory management و فانکشنالیتی های دیگر سطح پایین سیستم است.

## مراحل بوت سیستم عامل xv6

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

1. برای binary conversion: این کاربردی است که در xv6 makefile هم استفاده شده و در این مورد برای تبدیل فایل های ELF به باینری دودویی از آن استفاده می شود. دلیل این کار این است که بسیاری از سیستم ها از جمله bootloader ها انتظار دارند که kernel به فرمت raw binary باشید و `objcopy` برای این کار به ما کمک میکند که بتوانیم هنگام boot فایل هایی که raw binary شده اند را مستقیم وارد memory بکنیم.
2. برای section manipulation: برای اینکه بخش خاصی از یک فایل object را copy کنیم مانند بخش txt یا data. که حجم اطلاعات کمتر شده و optimize تر میشود.

3. برای stripping Unnecessary Data: برای اینکه قسمت سمبول های دیباگ رو strip کنیم و اطلاعات غیر ضروری را از فایل object حذف کنیم و حجم image نهایی را کم کنیم.
4. برای creating symbol files: این فایل ها برای آنالیز مرحله کامپایل و یا دیباگ کردن به درد خواهند خورد و میتوان از آنها استفاده کرد.

### 13. کد bootmain.c هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در آدرس 100000x0 قرار میدهد. علت انتخاب این آدرس چیست؟

برای این کار چند دلیل وجود دارد. و دلیل اول بحث های تاریخی است. منظور این است که در سیستم عامل های زیادی در گذشته مثل نسخه های اولیه linux از این ساختار استفاده شده بود و دلیلش هم ایجاد یک جدایی کامل بین قسمت های BIOS (و یا در سیستم های قدیمی تر DOS) با قسمت kernel است که cpu را وارد یک حالت protected میکند که از فایل های BIOS و bootblock محافظت می کند.

از طرفی 0x100000 برابر با یک مگابایت است که یعنی یک مگابایت اول برای فایل های bootloader از code و stack space و Bios رزرو خواهد شد که چون یک مگابایت انتخاب شده فضای کافی برای فایل های ذکر شده وجود خواهد داشت و جدایی کاملی بین kernel و bootblock ایجاد خواهد شد.

از طرفی چون ۱ مگابایت رند است کار های حافظه برای jump یا پیدا کردن کرنل راحت تر است.

همانطور که گفته شد، کد های bootloader به فضای های اضافی برای فضای stack یا حافظه ای برای فایل های موقت نیاز دارد و ۱ مگابایت فضای کافی ای است که اطمینان حاصل میکند فایل های kernel از فایل های BIOS routine و system-function های سطح پایین دیگر، جدا است و هیچ کدام در فایل های دیگری overwrite انجام نمی دهند.

## اجرای هسته xv6

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. همانطور که ذکر شد، ترجمه قطعه تاثیری بر ترجمه آدرس منطقی نمیگذارد. زیرا تمامی قطعه ها اعم از کد و داده روی یکدیگر می افتند. با این حال برای کد و داده های سطح کاربر پرچم USER\_SEG تنظیم شده است. چرا؟ (راهنمایی: علت مربوط به ماهیت دستورالعمل ها و نه آدرس است.)

برای این کار چند دلیل داریم:



۱. برای اینکه مطمئن شویم دستورات مشخصی که در حالت user\_mode اجرا می‌شوند اجازه دسترسی یا تغییر حافظه کرنل را ندارند. و در واقع به لایه امنیتی دیگر و stability ایجاد میکند.

۲. برای اینکه نشان دهیم user mode با kernel mode متفاوت است. این مورد برای محافظت از کرنل در مقابل دسترسی غیر مجاز user-level لازم است.

۳. رفتار بعضی از دستورات با توجه به اینکه ما در user mode هستیم یا kernel mode متفاوت است. وقتی فلگ user\_seg را برای user-level فعال میکنیم، xv6 می‌تواند مطمئن شود که کدهایی که کاربر وارد میکند نمیتواند به دستورات خاصی که سطح دسترسی بالاتر (kernel mode) لازم دارند را اجرا کند.

۴. درسته که segmentation روی ترجمه آدرس های منطقی تاثیری ندارد چون کد و دیتا overlap دارند، ولی وجود فلگ USER\_SEG از اجرای قوانین دسترسی به حافظه اطمینان حاصل میکند و از دسترسی مستقیم user code به kernel space جلوگیری میکند.

## اجرای نخستین برنامه سطح کاربر

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

```
37 // Per-process state
38 struct proc {
39     uint sz; // Size of process memory (bytes)
40     pde_t* pgdir; // Page table
41     char *kstack; // Bottom of kernel stack for this process
42     enum procstate state; // Process state
43     int pid; // Process ID
44     struct proc *parent; // Parent process
45     struct trapframe *tf; // Trap frame for current syscall
46     struct context *context; // swtch() here to run process
47     void *chan; // If non-zero, sleeping on chan
48     int killed; // If non-zero, have been killed
49     struct file *ofile[NOFILE]; // Open files
50     struct inode *cwd; // Current directory
51     char name[16]; // Process name (debugging)
52 };
53
54 // Process memory is laid out contiguously, low addresses first:
55 // text
56 // original data and bss
57 // fixed-size stack
58 // expandable heap
```

این struct در فایل proc.h تعریف شده و اجزای آن شامل:

1. `unit sz`: حجمی که پردازش اشغال کرده را به بایت بیان میکند.
2. `pde_t* pdgir`: پوینتری به `page Table` است. که همان جزئی است که نگاشتی بین حافظه مجازی و فیزیکی ایجاد می‌کند.
3. `char *kstack`: هر پردازش برای ذخیره سازی حالت کنونی خود به یک `stack` در `kernel` نیاز دارد. این پوینتر به پایین ترین خانه آن `stack` اشاره میکند.
4. `enum procstate state`: حالات مختلف پردازش را مشخص میکند که با توجه به `procstate` شامل `UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE` است.
5. `int pid`: هر پردازش یک `id` دارد و این `id` خاص این پردازش است.
6. `struct proc *parent`: اشاره به پردازش `parent` دارد. وقتی تابع `fork` یا `exec` صدا میشود یک پردازش `child` و `parent` ایجاد میشود. این اشاره به پردازش `parent` دارد.
7. `struct trapframe *tf`: اشاره گری به `trap frame` است که مسئولیت هندل کردن `trap` ها و اجرای `syscall` ها و ذخیره وضعیت پردازش قبل از اجرای `syscall` برای ادامه پردازش از همان حالت را دارد.
8. `struct context *context`: این ساختار قبل از `switch` کردن به پردازش دیگر با تابع `switch` محتوای رجیستر ها را ذخیره میکند تا بتوان از ادامه پردازش را ادامه داد.
9. `void *chan`: اگر مقداری غیر صفر داشته باشد یعنی توسط تابعی مانند `wait` متوقف شده و در حالت `sleep` است.
10. `int killed`: مقدار غیر صفر یعنی پردازش `kill` شده است.
11. `struct file *ofile[NOFILE]`: آرایه ای به پوینتر فایل های باز شده. وقتی به فایل باز میشود اشاره گری به ابتدای آن فایل اشاره میکند و ذخیره میشود و `index` آن خانه به عنوان `file descriptor` بازگردانده می‌شود.
12. `struct node* cwd`: نمایانگر دایرکتوری جاری فرایند.
13. `char name`: نام فرایند که برای اشکال زدایی استفاده می‌شود

معادل این `struct` در هسته لینوکس:

<https://github.com/torvalds/linux/blob/master/include/linux/sched.h>

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؟

در انتهای فایل entry.s، شرایطی برای اجرای کد C فراهم شده است تا تابع main فراخوانی شود. این تابع در هسته‌ای که سیستم‌عامل راه‌اندازی شده، اجرا می‌شود. هسته‌های دیگر از طریق entryother.s به تابع mpenter منتقل می‌شوند، که در اینجا چهار تابع مشخص فراخوانی می‌شود؛ این توابع همچنین در تابع main نیز فراخوانی شده‌اند. به عنوان نمونه، تابع switchkvm در کد kvmalloc که در main وجود دارد، مورد استفاده قرار گرفته است. در مجموع، 18 تابع در تابع main به کار رفته‌اند. از این تعداد، چهار تابع در mpenter بین تمامی هسته‌ها مشترک بوده و 14 تابع دیگر مختص هر هسته هستند.

مشترک‌ها:

switchkvm,seginit,lapicinit,mpmain

اختصاصی‌ها:

kinitl,kvmalloc(setupkvm),mpinit,picinit,ioapicinit,consoleinit,uartinit,pinit,tvinit,binit,fileinit,ideinit,startothers,kinit2,userinit

زمان‌بند در تابع mpmain اجرا می‌شود و بین تمامی هسته‌های پردازنده مشترک است. هر هسته یک زمان‌بند اختصاصی دارد که پس از تنظیمات اولیه، آن را به کار می‌گیرد.

```

// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    picinit(); // disable pic
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
    pinit(); // process table
    tvinit(); // trap vectors
    binit(); // buffer cache
    fileinit(); // file table
    ideinit(); // disk
    startothers(); // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
    userinit(); // first user process
    mpmain(); // finish this processor's setup
}

// Other CPUs jump here from entryother.S.
static void
mpenter(void)
{
    switchkvm();
    seginit();
    lapicinit();
    mpmain();
}

// Common CPU setup code.
static void
mpmain(void)
{
    cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
    idtinit(); // load idt register
    xchg(&(mycpu()->started), 1); // tell startothers() we're up
    scheduler(); // start running processes
}

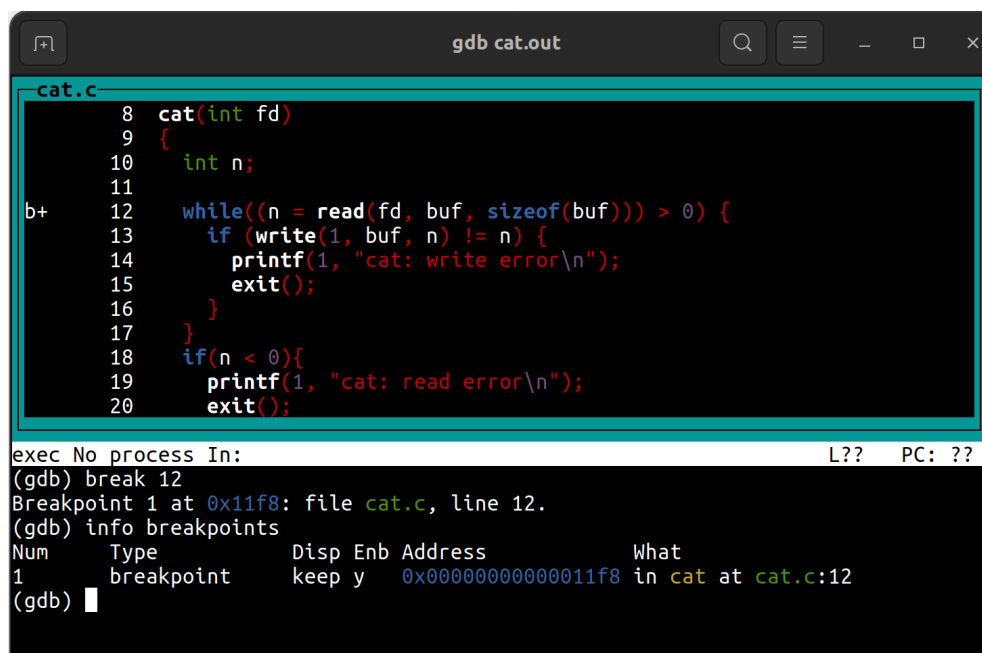
```

# اشکال زدایی

## روند اجرای GDB

1. برای مشاهده breakpoint ها از چه دستوری استفاده می‌شود؟

برای این کار کافی است که از دستور `info breakpoint` استفاده کنیم.



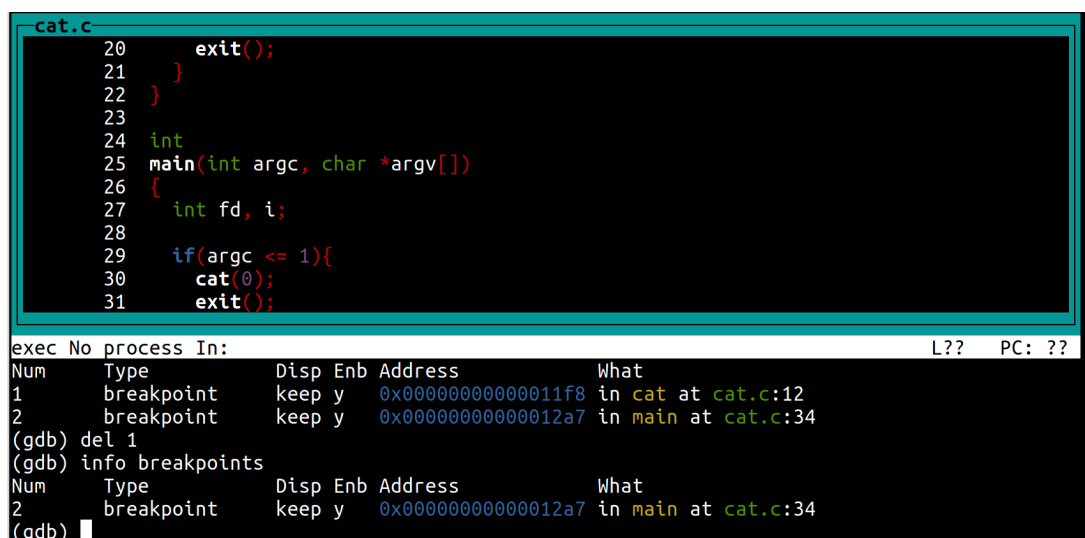
```
cat.c
8 cat(int fd)
9 {
10     int n;
11
12     while((n = read(fd, buf, sizeof(buf))) > 0) {
13         if (write(1, buf, n) != n) {
14             printf(1, "cat: write error\n");
15             exit();
16         }
17     }
18     if(n < 0){
19         printf(1, "cat: read error\n");
20         exit();
21     }
22 }
```

```
exec No process In: L?? PC: ??
(gdb) break 12
Breakpoint 1 at 0x11f8: file cat.c, line 12.
(gdb) info breakpoints
Num    Type         Disp Enb Address          What
1      breakpoint   keep y  0x00000000000011f8 in cat at cat.c:12
(gdb)
```

2. برای حذف یک breakpoint از چه دستوری و چگونه استفاده می‌شود؟

با استفاده از دستور `del <breakpoint_number>` به صورت

برای مثال اگر فرض کنیم breakpoint های ما در ابتدا به صورت زیر است و سپس با وارد کردن دستور `del 1` خواهیم داشت:



```
cat.c
20     exit();
21 }
22 }
23
24 int
25 main(int argc, char *argv[])
26 {
27     int fd, i;
28
29     if(argc <= 1){
30         cat(0);
31         exit();
32     }
33 }
```

```
exec No process In: L?? PC: ??
Num    Type         Disp Enb Address          What
1      breakpoint   keep y  0x00000000000011f8 in cat at cat.c:12
2      breakpoint   keep y  0x00000000000012a7 in main at cat.c:34
(gdb) del 1
(gdb) info breakpoints
Num    Type         Disp Enb Address          What
2      breakpoint   keep y  0x00000000000012a7 in main at cat.c:34
(gdb)
```

همانطور که میبینیم breakpoint شماره یک حذف شد.

## کنترل روند اجرا و دسترسی به حالت سیستم

### 3. خروجی دستور bt چه چیزی را نشان می‌دهد؟

دستور bt که مخفف backtrace است call stack برنامه در لحظه کنونی را نشان می‌دهد. هر تابع که صدا زده می‌شود یک stack frame مخصوص به خودش را می‌گیرد که متغیرهای محلی و آدرس بازگشت و غیره در آن ذخیره می‌شود. خروجی این دستور در هر خط یک frame stack را نشان می‌دهد که به ترتیب از درونی‌ترین به بیرونی‌ترین است. می‌توان به این دستور مقدار n را داد که n بیرونی‌ترین لایه‌ها را نشان دهد. می‌توان به این دستور مقدار -n را داد که n درونی‌ترین لایه‌ها را نشان دهد.

```
cat.c
7 void
8 cat(int fd)
9 {
10     int n;
11
12     while((n = read(fd, buf, sizeof(buf))) > 0) {
13         if (write(1, buf, n) != n) {
14             printf(1, "cat: write error\n");
15             exit();
16         }
17     }
18     if(n < 0){

B+>

multi-thre Thread 0x7ffff7d797 In: cat L12 PC: 0x5555555551f8
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, cat (fd=0) at cat.c:12
(gdb) bt
#0 cat (fd=0) at cat.c:12
#1 0x00005555555552a2 in main (argc=1, argv=0x7fffffdd58) at cat.c:30
(gdb)
```

4. دو تفاوت دستور های x و print را توضیح دهید. چگونه می‌توان محتوای یک ثابت خاص را چاپ کرد؟

در دستور print مقدار متغیری که به عنوان آرگومان به آن داده شده نمایش داده می شود.

```
cat.c
7 void
8 cat(int fd)
9 {
10     int n;
11
12     while((n = read(fd, buf, sizeof(buf))) > 0) {
13         if (write(1, buf, n) != n) {
14             printf(1, "cat: write error\n");
15             exit();
16         }
17     }
18     if(n < 0){
19         printf(1, "cat: read error\n");
20         exit();
21     }
22 }
```

multi-thre Thread 0x7ffff7d797 In: cat L12 PC: 0x5555555551f8  
Start it from the beginning? (y or n) yStarting program: /home/amir/Desktop/OS-Lab1/cat.out  
[Thread debugging using libthread\_db enabled]  
Using host libthread\_db library "/lib/x86\_64-linux-gnu/libthread\_db.so.1".  
Breakpoint 1, cat (fd=0) at cat.c:12  
(gdb) print n  
\$1 = 0  
(gdb)

در دستور x مقدار خانه ای از حافظه که به عنوان آرگومان به آن داده شده نمایش داده می شود.

```
cat.c
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 char buf[512];
6
7 void
8 cat(int fd)
9 {
10     int n;
11
12     while((n = read(fd, buf, sizeof(buf))) > 0) {
13         if (write(1, buf, n) != n) {
14             printf(1, "cat: write error\n");
15             exit();
16         }
17     }
18     if(n < 0){
19         printf(1, "cat: read error\n");
20         exit();
21     }
22 }
```

multi-thre Thread 0x7ffff7d797 In: cat L12 PC: 0x5555555551f8  
(gdb) info breakpoints  
Num Type Disp Enb Address What  
1 breakpoint keep y 0x00005555555551f8 in cat at cat.c:12  
breakpoint already hit 1 time  
2 breakpoint keep y 0x00005555555552b0 in main at cat.c:35  
(gdb) x buf  
0x5555555558040 <buf>: 0x00000000  
(gdb)

5. برای نمایش وضعیت ثبات ها از چه دستوری استفاده می شود؟ متغیرهای محلی چگونه؟ در معماری x86 رجیستر های edi و esi نشانگر چه چیزی هستند؟

با استفاده از دستور info registers می توان وضعیت ثباتها را مشاهده کرد

```
gdb cat.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, cat (fd=0) at cat.c:12
12      while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) info registers
rax             0x5555555527f          93824992236159
rbx             0x0                  0
rcx             0x555555557d98        93824992247192
rdx             0x7fffffffdd68        140737488346472
rsi             0x7fffffffdd58        140737488346456
rdi             0x0                  0
rbp             0x7fffffffddc10        0x7fffffffddc10
rsp             0x7fffffffdbf0        0x7fffffffdbf0
r8              0x7ffff7f97f10        140737353711376
r9              0x7ffff7fc9040        140737353912384
r10             0x7ffff7fc3908        140737353890056
r11             0x7ffff7fde660        140737353999968
r12             0x7fffffffdd58        140737488346456
r13             0x5555555527f          93824992236159
r14             0x555555557d98        93824992247192
r15             0x7ffff7ffd040        140737354125376
rip             0x5555555551f8        0x5555555551f8 <cat+15>
eflags          0x206                [ PF IF ]
cs              0x33                51
ss              0x2b                43
ds              0x0                  0
es              0x0                  0
--Type <RET> for more, q to quit, c to continue without paging--
```

با استفاده از دستور info locals می توان وضعیت متغیر های محلی را مشاهده کرد

```
(gdb) list
7      void
8      cat(int fd)
9      {
10         int n;
11
12         while((n = read(fd, buf, sizeof(buf))) > 0) {
13             if (write(1, buf, n) != n) {
14                 printf(1, "cat: write error\n");
15                 exit();
16             }
(gdb) info locals
n = 0
(gdb)
```

رجیسترهای esi و edi دو مورد از ثباتهای همه منظوره موجود در معماری هستند. آنها رجیستر های 32 بیتی هستند.



esi: اساسا به عنوان source index برای عملیات رشته ها مانند کپی کردن یا مقایسه رشته ها استفاده می شود. همچنین معمولا عنوان نشانگر داده های منبع در عملیات data manipulation استفاده می شود.

edi: به عنوان destination index برای عملیات رشته ای عمل می کند که در آن داده ها نوشته یا اصلاح می شوند. مشابه edi , esi اغلب به عنوان یک اشاره گر به داده های مقصد یا آدرس مقصد برای دستورات data manipulation استفاده می شود.

## 6. به کمک استفاده از gdb ساختار struct input را توضیح دهید.

این استراکچر در فایل c.console تعریف شده است و برای ورودی command line کنسول سیستم عامل استفاده می شود.

با دستور ptype input به محتویات این متغیر دسترسی می یابیم

```
(gdb) ptype input
type = struct Input {
    char buf[128];
    uint r;
    uint w;
    uint e;
}
(gdb) █
```

این struct دارای 4 متغیر است:

- buf: آرایه ای به سائز 128 است که محل ذخیره خط ورودی است.
- e : نشان دهنده محل کنونی کرسر در خط است.
- w : محل شروع خط را نشان می دهد.
- r : برای خواندن buf بعد از زدن enter استفاده می شود.

w زمانی که دستوری ثبت شود (اینتر بخورد) به انتهای بافر یا همان خطی که اجرا شده می رود.

e زمانی که حرفی پاک شود یکی به عقب میاید و وقتی حرفی نوشته شود به جلو میرود.

r وقتی دستوری اجرا می شود او از مکان خود به انتهای دستور میاید .

در ابتدا که چیزی در ورودی نوشته نشده است مقدار متغیر های input به شکل زیر است:

```
(gdb) print input
$1 = {buf = '\000' <repeats 127 times>, r = 0, w = 0, e = 0}
```

وارد کردن عبارت Amir و سپس زدن enter ابتدا میبینیم که مقدار w برابر e است.

```
(gdb) print input
$5 = {buf = "Amir", '\000' <repeats 123 times>, r = 0, w = 0, e = 4}
```

اجرای کلمه Amir (زدن enter) که میبینیم که مقدار w برابر e است.

```
(gdb) print input
$6 = {buf = "Amir\n", '\000' <repeats 122 times>, r = 5, w = 5, e = 5}
```

سپس عبارت salam را در ورودی می‌نویسیم و خواهیم دید که به مقدار e پنج واحد اضافه می‌شود (به اندازه طول ورودی جدید)

```
(gdb) print input
$7 = {buf = "Amir\nsalam", '\000' <repeats 117 times>, r = 5, w = 5, e = 10}
```

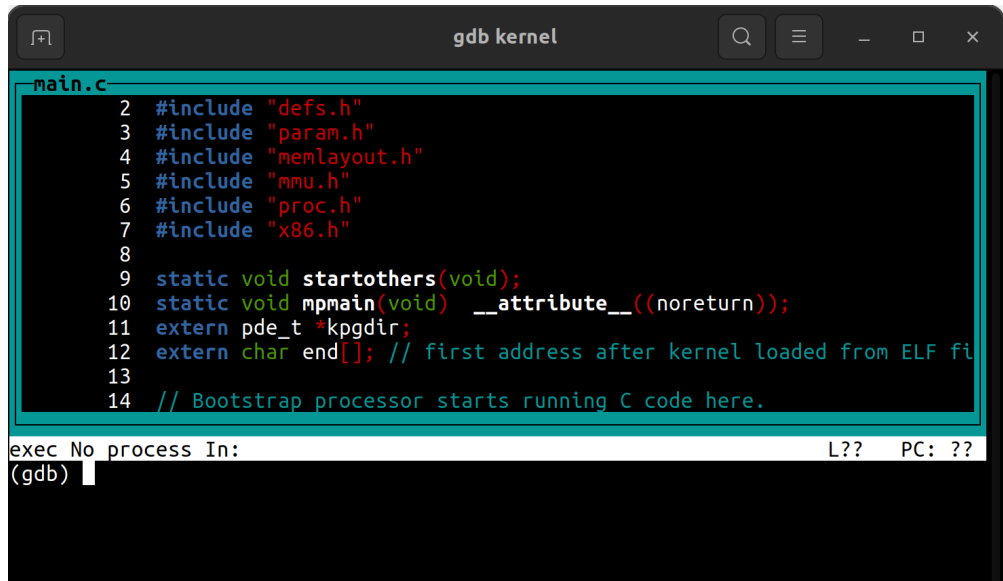
حذف کاراکتر ا باعث میشود همانطور که میبینیم بافر ورودی هم تغییر میکند و همچنین e هم یک واحد کم میشود

```
(gdb) print input
$9 = {buf = "Amir\nsaam\000 ", '\000' <repeats 116 times>, r = 5, w = 5, e = 9}
```

## اشکال زدایی در سطح کد اسمبلی

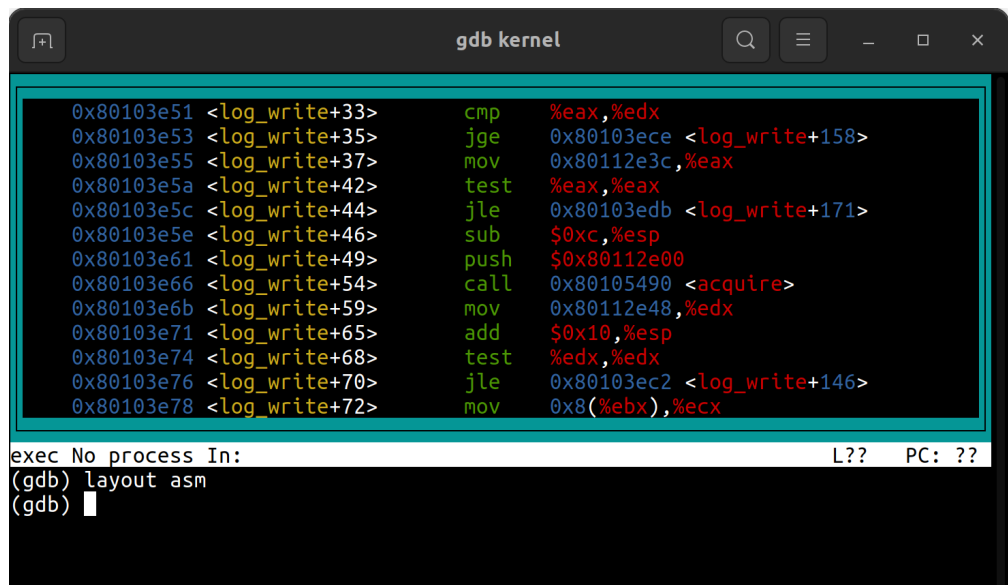
### 7. خروجی دستور های layout src و layout asm در TUI چیست؟

با وارد کردن دستور layout src میتوانیم کد c مربوطه را هنگام توقف و دیباگ در محیط TUI ببینیم.



```
gdb kernel
main.c
2 #include "defs.h"
3 #include "param.h"
4 #include "memlayout.h"
5 #include "mmu.h"
6 #include "proc.h"
7 #include "x86.h"
8
9 static void startothers(void);
10 static void mpmain(void) __attribute__((noreturn));
11 extern pde_t *kpgdir;
12 extern char end[]; // first address after kernel loaded from ELF fi
13
14 // Bootstrap processor starts running C code here.
15
16 exec No process In: L?? PC: ??
(gdb)
```

با وارد کردن دستور layout asm می‌توانیم کد اسمبلی را هنگام توقف و دیباگ در محیط TUI ببینیم.



```
gdb kernel
0x80103e51 <log_write+33>    cmp     %eax,%edx
0x80103e53 <log_write+35>    jge     0x80103ece <log_write+158>
0x80103e55 <log_write+37>    mov     0x80112e3c,%eax
0x80103e5a <log_write+42>    test    %eax,%eax
0x80103e5c <log_write+44>    jle     0x80103edb <log_write+171>
0x80103e5e <log_write+46>    sub     $0xc,%esp
0x80103e61 <log_write+49>    push    $0x80112e00
0x80103e66 <log_write+54>    call    0x80105490 <acquire>
0x80103e6b <log_write+59>    mov     0x80112e48,%edx
0x80103e71 <log_write+65>    add     $0x10,%esp
0x80103e74 <log_write+68>    test    %edx,%edx
0x80103e76 <log_write+70>    jle     0x80103ec2 <log_write+146>
0x80103e78 <log_write+72>    mov     0x8(%ebx),%ecx
17
18 exec No process In: L?? PC: ??
(gdb) layout asm
(gdb)
```

8. برای جا به جایی میان توابع زنجیره ای فراخوانی جاری (نقطه توقف) از چه دستور هایی استفاده می‌شود؟

دیدیم که پشته فراخوانی را با استفاده از دستور bt یا backtrace ببینیم. حال پس از آن برای جابه جایی میان توابع زنجیره فراخوانی می‌توان از دستور up و down استفاده کرد. برای مثال اگر پشته فراخوانی مانند تصویر زیر باشد:

```
gdb kernel

proc.c
289     pid = p->pid;
290     kfree(p->kstack);
291     p->kstack = 0;
292     freevm(p->pgdir);
293     p->pid = 0;
294     p->parent = 0;
b+ 295     p->name[0] = 0;
296     p->killed = 0;
297     p->state = UNUSED;
298     release(&ptable.lock);
299     return pid;
300 }
301 }

remote Thread 1.1 In: mycpu                                L48  PC: 0x801047f1
#0  mycpu () at proc.c:48
#1  0x8010535e in pushcli () at spinlock.c:113
#2  0x8010543d in holding (lock=0x80113480 <ptable>) at spinlock.c:93
#3  release (lk=0x80113480 <ptable>) at spinlock.c:49
#4  0x80104bb1 in scheduler () at proc.c:353
#5  0x80103f2f in mpmain () at main.c:57
#6  0x8010407c in main () at main.c:37
(gdb)
```

با وارد کردن دستور up به تابع pushcli در خط 113 فایل spinlock.c می‌رویم.

```
gdb kernel

spinlock.c
105 pushcli(void)
106 {
107     int eflags;
108
109     eflags = readeflags();
110     cli();
111     if(mycpu()->ncli == 0)
112         mycpu()->intena = eflags & FL_IF;
> 113     mycpu()->ncli += 1;
114 }
115
116 void
117 popcli(void)

remote Thread 1.1 In: pushcli                                L113  PC: 0x8010535e
#2  0x8010543d in holding (lock=0x80113480 <ptable>) at spinlock.c:93
#3  release (lk=0x80113480 <ptable>) at spinlock.c:49
#4  0x80104bb1 in scheduler () at proc.c:353
#5  0x80103f2f in mpmain () at main.c:57
#6  0x8010407c in main () at main.c:37
(gdb) up
#1  0x8010535e in pushcli () at spinlock.c:113
(gdb)
```