

# آزمایشگاه سیستم عامل

## پروژه یک

اعضای گروه:

الهه خداوردی - 810100132

فرشته باقری - 810100089

عاطفه میرزاخانی - 810100220

**Repository:** <https://github.com/elahekhodaverdi/Operating-System-Lab-Projects>

**Latest Commit:** 7931e3db341b8c1a811bbe2b484e1fd515528520

## آشنایی با سیستم عامل xv6

### 1. معماری سیستم عامل 6xv چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

سیستم عامل 6xv یک سیستم عامل آموزشی Unix-Based است که طبق داکيومنت 6xv ، یک re-implementation از 6Unix V است و این سیستم عامل مبتنی بر پردازنده های 86x نوشته شده است. در فایل x86.h نیز از دستورات پردازنده 86x استفاده شده است.

سیستم عامل unix از سه بخش اصلی kernel, shell و user applications تشکیل شده است که 6xv هم همین روال را دارد و به طور کلی در اجرای پردازش ها از روش unix تبعیت می کند.

That operating system, xv6, provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie's Unix operating system, as well as mimicking Unix's internal design.

Xv6 runs on Intel 80386 or later ("x86") processors on a PC platform, and much of its low-level functionality (for example, its process implementation) is x86-specific.

معماری هسته 6xv به صورت یکپارچه (monolithic) است یعنی کل سیستم عامل در حالت سوپروایزر اجرا می شود.

One possibility is that the entire operating system resides in the kernel, so that the implementations of all system calls run in kernel mode. This organization is called a monolithic kernel. In this organization the entire operating system runs with full hardware privilege.

## 2. یک پردازنده در سیستم عامل 6xv 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 از دو بخش User-space memory و Per-process state private to the kernel تشکیل شده است. بخش User-space memory شامل دستورات (instructions)، اطلاعات (data) و پشته (stack) است. دستورات، 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 و پشته قسمتی از حافظه است که برای فراخوانی ها و کنترل متغیرهای محلی استفاده می شود. این بخش از پردازنده مختص پردازنده بوده و توسط بقیه پردازنده ها قابل دسترسی نیست. Per-process state private to the kernel شامل اطلاعات و ساختارهای داده ای است که توسط هسته نگهداری می شوند تا اجرای یک فرایند را مدیریت و کنترل کنند. این شامل داده هایی مانند شناسه فرایند (PID) است که هر فرایند را به صورت یکتا شناسایی می کند، همچنین شامل سایر اطلاعات مربوط به فرایند مانند وضعیت اجرای فعلی (اجرا، انتظار و غیره)، اولویت فرایند، شماره گذاری فایل و سایر ساختارهای داده ای مرتبط با هسته است. این وضعیت فرایند به صورت خاص توسط هسته مدیریت و دسترسی دارد و به صورت مستقیم توسط خود فرایند قابل دسترسی یا تغییر نیست.

این سیستم عامل به طور کلی پردازنده را به طریق time-share به پردازنده های مختلف اختصاص می دهد. پردازنده بین پردازنده های قابل اجرا (که منتظر اجرا شدن هستند) جا به جا می شود

## 3. مفهوم file descriptor در سیستم عامل مبتنی بر UNIX چیست؟ عملکرد pipe در سیستم عامل 6xv چگونه است و به طور معمول برای چه هدفی استفاده می شود؟

هر پردازنده یک آرایه خصوصی به نام ofile دارد که در آن اشاره گر ها به فایل هایی که باز کرده است وجود دارند. هنگامی که یک پردازنده فایلی را باز می کند index ای از ofile که پوینتر به آن فایل در آن ذخیره شده باز گردانده می شود و پردازنده با استفاده از این عدد می تواند در فایل بنویسد یا از آن بخواند.

هنگامی که یک پردازنده دستور fopen را می دهد، kernel کوچک ترین fd که UNUSED است را به فایل مورد نظر اختصاص داده و آن را برمی گرداند. هنگامی که یوزر fclose را اجرا می کند سیستم fd مورد نظر را برای استفاده مجدد آزاد می کند.

Fd ها می توانند به فایل های عادی، پایپ یا device file اشاره کنند. در این حالت یوزر فارغ از اینکه تایپ فایلش چیست صرفا با یک fd کار می کند و به نوعی مانند یک interface عمل می کند.

در هر پروسه، file descriptor های اول تا سوم به ترتیب مربوط به فایل های زیر هستند:

Standard input

Standard output

Standard error

پایپ ها یک مکانیزم برای ارتباط بین پردازش های مختلف (inter-process communication) است که اجازه می دهد دو پردازش با تبادل داده ها با هم ارتباط بگیرند. با استفاده از پایپ ها می توان output یک پردازش را به input دیگری وصل کنیم.

پایپ ها توسط sysCall pipe() ایجاد می شوند و دو fd یکی برای read end و دیگری برای write end بر می گردانند.

عملکرد پایپ ها باعث Synchronization اجرای پردازش ها نیز می شود.

برای مثال وقتی پردازش پدر تابع fork را صدا می زند، یک child process ایجاد می شود که این دو میتوانند از طریق پایپ با هم ارتباط بگیرند. هر پردازش سری که نمیخواهد استفاده کند را می بندد تا در هر زمان فقط read end یا write end برای آن پردازش باز باشد. سپس پدر می تواند چیزی در پایپ بنویسد و فرزند از read end آن را بخواند.

لازم به ذکر است که عملکرد های پایپ blocking هستند یعنی اگر پردازش ای بخواهد از پایپ خالی چیزی بخواند تا زمانی که دیتایی وجود نداشته باشد پردازش بلاک خواهد شد. برای نوشتن در پایپ پر هم همینطور است.

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

تابع fork برای ایجاد یک process جدید استفاده می شود. درواقع این تابع یک نسخه کپی از پردازش ای می سازد که این تابع را صدا زده است. منظور از کپی این است که دیتا و دستورات پردازش فعلی در حافظه پردازش جدید (child) کپی می شوند. با وجود اینکه در لحظه ایجاد پردازش فرزند، داده های آن (متغیرها و رجیسترها) با پردازش پدر یکسان هستند، اما درواقع این دو پردازش حافظه جداگانه ای خواهند داشت و تغییر یک متغیر در پردازش پدر، آن متغیر در پردازش فرزند را تغییر نمی دهد. پردازش پدر پس از ایجاد پردازش فرزند، به caller تابع fork بازمی گردد که امکان اجرای همزمان دو پردازش را فراهم می سازد. مقدار return

شده از تابع fork نیز pid پردازش فرزند خواهد بود. نقطه شروع پردازش فرزند نیز دقیقاً همان caller تابع fork است، با این تفاوت که مقدار خروجی این تابع عدد 0 خواهد بود. پس اگر با استفاده از قطعه کد `int pid = fork();` یک پردازش جدید درست کنیم، یکی از حالت‌های زیر برای pid رخ می‌دهد:

- `pid = 0`: در پردازش فرزند هستیم.
- `pid > 0`: در پردازش پدر هستیم و مقدار pid در واقع شناسه پردازش فرزند است.
- `pid < 0`: در زمان اجرای تابع fork و پردازش جدید خطایی رخ داده و پردازش فرزند ایجاد نشده است. اگر پس از fork کردن از تابع `wait()` استفاده شود، 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.

```
int pid = fork();

if (pid == 0) {
    printf("This is child process.\n");
    printf("Child process is exiting...\n");
    exit(0);
}

else if (pid > 0) {
    printf("This is parent process, child's PID = %d.\n", pid);
    printf("Waiting for child process to exit...\n");
    wait();
    printf("Child process exited.\n");
}

else {
    printf("Fork failed!\n");
}
```

در حقیقت اتفاقی که در سیستم‌عامل می‌افتد تا حدی در تکه کد بالا توضیح داده شده است.

تابع exec حافظه پردازش فعلی را با یک حافظه جدید که در آن یک برنامه با فایل ELF لود شده است، جایگزین می‌کند، اما file table اولیه را هم حفظ می‌کند. درواقع exec() راهی برای اجرای یک برنامه در پردازش فعلی است. برخلاف تابع fork()، برنامه به caller تابع exec() بازمی‌گردد و برنامه جدید اجرا می‌شود، مگر اینکه در زمان اجرای این تابع خطایی رخ دهد. برنامه جدید اجرا شده در یک نقطه‌ای با استفاده از تابع exit اجرای پردازش را خاتمه می‌دهد. تابع exec دو پارامتر ورودی دارد که پارامتر اول نام فایل برنامه و پارامتر دوم آرایه آرگومان‌های ورودی برنامه است.

قطعه کد زیر مثالی از اجرای این تابع را نشان می‌دهد:

```
char* args[] = { "ls", "-l", "/home", NULL }; // NULL is required
```

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

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

در حقیقت ادغام نکردن این دو تابع از ساختن پردازش‌های بی‌مصرف و جایگزین شدن سریع آنها توسط exec جلوگیری می‌کند. در حالت عادی توابع fork و exec پشت سر هم اجرا می‌شوند. اگر این دو ادغام شوند، علاوه بر پردازش‌های اضافه و میزان حافظه زیادی که اشغال می‌شود، مدیریت آرگومان‌های توابع هم دشوار می‌شود. مزیت ادغام نکردن این دو تابع در زمان I/O redirection خودش را نشان می‌دهد. زمانی که کاربر در shell

یک برنامه را اجرا می‌کند، کاری که در پشت صحنه انجام می‌شود به شرح زیر است:

1. ابتدا دستور تایپ شده توسط کاربر در ترمینال را می‌خواند.
2. با استفاده از تابع fork یک پردازش جدید ایجاد می‌کند.
3. در پردازش فرزند با استفاده از تابع exec برنامه درخواست شده توسط کاربر را جایگزین پردازش فعلی (فرزند) می‌کند.
4. در پردازش پدر برای اتمام کار پردازش فرزند wait می‌کند.
5. پس از اتمام پردازش فرزند به main بازمی‌گردد و منتظر دستور جدید می‌شود.

زمانی که کاربر برای یک دستور از redirection استفاده می‌کند، تغییرات لازم در file descriptor ها پس از fork و پیش از exec و در پردازش فرزند انجام می‌شود.

قطعه کد زیر این مورد را به شکل ساده نشان می‌دهد (فرض کنید دستور اجرا شده cat < in.txt است):

```
char* args = { "cat", NULL };
```

```
int pid = fork();
```

```
if (pid == 0) {
```

```

close(0); // close stdin

open("in.txt", O_RDONLY); // open in.txt for reading (fd: 0)

exec("/bin/cat", args);

printf("Exec failed!\n");

}

else if (pid > 0) {

    wait();

    printf("Child process has exited.\n");

}

else {

    printf("The fork failed!\n");

}

```

در صورتی که این دو تابع ادغام شوند، یا باید حالت‌های redirection به‌عنوان پارامتر به تابع forkexec پاس داده شوند که هندل کردن این حالت در دسرهاى خودش را دارد و یا اینکه shell پیش از اجرای این تابع، file descriptor خود را تغییر دهد و بعد از اتمام کار این تابع نیز به حالت قبل برگرداند و یا در بدترین حالت، هندل کردن redirection را در هر برنامه مانند cat پیاده‌سازی کنیم.

## 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 #29:\n");
    printf(1, "Elaheh Khodaverdi\n");
    printf(1, "Fereshteh Bagheri\n");
    printf(1, "Atefeh Mirzakhani\n");
    pid = fork();
}

```

```

cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 in
t 58
init: starting sh
Group #29:
Elaheh Khodaverdi
Fereshteh Bagheri
Atefeh Mirzakhani
$

```

## اضافه کردن چند قابلیت به کنسول xv6

نکته: برای پیاده‌سازی دستورهای داده شده در کد مربوط به سیستم عامل xv6 یک متغیر global به نام back\_count قرار داده شده است که نشان دهنده تعداد کاراکترهایی است که به عقب بازگشته ایم:

1. اضافه کردن دستور Ctrl+B

```

case C('B'): // Cursor Backward
    if ((input.e - back_count) > input.w)
        backwardCursor();
    break;

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] != ((' '$' & 0xff) | 0x0700))
        pos--;

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

```

2. اضافه کردن دستور Ctrl+F:

```

case C('F'):
    if (back_count > 0)
        forwardCursor();
    break;

```

```

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);
    back_count--;
}

```

3. اضافه کردن دستور Ctrl+L :

```

static void clrscreen()
{
    for (int i = 0; i < back_count; i++)
        forwardCursor();
    back_count = 0;
    int pos;

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

    while (pos)
    {
        conputc(BACKSPACE);
        pos--;
    }
    input.e = input.w = input.r = 0;
    conputc('$');
    conputc(' ');
    pos += 2;
    // reset cursor
    outb(CRTPORT, 14);
    outb(CRTPORT + 1, pos >> 8);
    outb(CRTPORT, 15);
    outb(CRTPORT + 1, pos);
}

void consoleintr(int (*reset)(void))

```



4. اضافه کردن دستور `:arrow up`:

```
case 226:
    if (inputs.size && inputs.end - inputs.cur < inputs.size)
        arrowup();
    break;

static void arrowup()
{
    if (inputs.cur == inputs.end)
    {
        inputs.history[inputs.end % INPUT_HISTORY] = input;
    }
    displayclear();
    input = inputs.history[--inputs.cur % INPUT_HISTORY];
    input.buf[--input.e] = '\0';
    displaylastcommand();
}
```

5. اضافه کردن دستور `:arrow down`:

```
case 227:
    if (inputs.size && inputs.end - inputs.cur > 0)
        arrowdown();
    break;
```

```
static void arrowdown()
{
    if (inputs.cur < inputs.end)
    {
        displayclear();
        input = inputs.history[++inputs.cur % INPUT_HISTORY];
        if (input.e != input.w && inputs.cur != inputs.end)
            input.buf[--input.e] = '\0';
        displaylastcommand();
    }
}
```

## اجرا و پیاده سازی یک برنامه سطح کاربر

```
Htereh MirZakhan1
$ strdiff apple banana
$ cat strdiff_result.txt
100011
$
```

## کامپایل سیستم عامل xv6

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

UPROGS -> user programs

این متغیر شامل لیستی از برنامه های سطح کاربر است که در هنگام ساخت و کامپایل 6xv کامپایل شده و به برنامه قابل اجرا توسط OS تبدیل می شوند. این فایل ها عموماً در user directory سیستم عامل 6xv قرار دارند.

ULIB -> user libraries

این متغیر شامل تعدادی کتابخانه زبان C است که در اجرای برنامه ها (یا سیستم عامل) به آنها نیاز است. 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 6xv قرار دارند.

وقتی makefile اجرا می شود ابتدا کتابخانه های ulib کامپایل شده و به فایل آبجکت تبدیل می شوند. این آبجکت ها با برنامه های کاربر در uprogs لینک شده و تبدیل به فایل قابل اجرا می شوند که کاربرد میتواند در 6xv آن ها را اجرا کند.

## مراحل بوت سیستم عامل xv6

11. برنامه های کامپایل شده در قالب فایل های دودویی نگهداری می شوند. فایل مربوط به بوت نیز دودویی است. نوع این فایل دودویی چیست؟ تفاوت این نوع فایل دودویی با دیگر فایل های دودویی کد xv6 چیست؟ چرا از این نوع فایل دودویی استفاده شده است؟ این فایل را به زبان قابل فهم انسان اسمبلی تبدیل نمایید. (راهنمایی: از ابزار objdump استفاده کنید. باید بخشی از آن مشابه فایل bootasm.S باشد).

در	قسمتی	از	Makefile	مشاهده	می کنیم:
----	-------	----	----------	--------	----------

```
bootblock: bootasm.S bootmain.c
$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
$(OBJDUMP) -S bootblock.o > bootblock.asm
$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
```

دو فایل bootmain.c و bootasm.S که به ترتیب به زبان های C و assembly هستند کامپایل شده و فایل های bootmain.o و bootasm.o که دارای پسوند O هستند ساخته می شوند و این فایل ها توسط دستور LD با هم لینک می شوند و فایل bootblock.o ساخته می شود.

bootblock.o در آدرس خاصی شروع به اجرا شدن می‌کند، بنابراین در هنگام ساخته شدن آن از فلگ 0x7Ctext 00T- استفاده شده است که آدرس بخش text. فایل خروجی را مشخص می‌کند. فلگ e start- هم بیانگر آن است که نقطه شروع برنامه لیبل start در bootasm.S است.

بعد از ساخت bootblock.o، با استفاده از دستور OBJCOPY که در سوال بعد توضیح داده می‌شود فایل bootblock ساخته می‌شود که نوع آن باینری خام است. اما دیگر فایل‌های دودویی از فرمت ELF هستند.

ELF برای تعریف فرمت فایل‌های اجرایی در 6xv استفاده می‌شود. هنگامی که یک برنامه C را در 6xv کامپایل کنیم، یک فایل اجرایی ELF تولید می‌کند که سیستم عامل می‌تواند بارگذاری و اجرا کند. فایل‌های ELF دارای یک هدر هستند که حاوی اطلاعات حیاتی در مورد فایل اجرایی است، مانند نقطه ورود (جایی که برنامه باید اجرا شود)، بخش‌های برنامه، و جزئیات مربوط به پیوند پویا. فایل‌های ELF دارای section هایی نیز هستند که می‌توان به text. (که حاوی کد اصلی برنامه است)، data. (حاوی داده‌های اولیه و متغیرهای static و global) و bss. (حاوی متغیرهایی که مقداردی اولیه ندارند). rodata. (حاوی داده‌های read-only است که در طول اجرای برنامه نمی‌توان آنها را تغییر داد). اشاره کرد.

بنابراین فایل bootblock از فرمت باینری خام است و ELF نیست و هدر و بخش‌هایی مانند data. و غیره را ندارد و بخش اصلی آن text. است.

از آنجا که فایل باینری خام فقط بخش text را دارد پس حجم آن کم می‌باشد (حداکثر 512 بایت) و به همین منظور از این نوع برای فایل بوت استفاده می‌شود زیرا اولین فایلی که برای بوت شدن اجرا می‌شود نباید حجم زیادی داشته باشد.

همچنین فایل ELF نمی‌تواند مستقیماً توسط CPU اجرا شود زیرا هسته سیستم عامل آن را می‌شناسد پس تا زمانی که هسته اجرا نشده است فایل ELF نمی‌تواند اجرا شود و می‌بایست به فایل باینری خام تبدیل شود که توسط CPU شناخته شده است.

برای تبدیل bootblock به زبان اسمبلی از دستور زیر استفاده می‌کنیم:

```
objdump -D -b binary -m i386 -M addr16,data16 bootblock
```

-D: برای disassemble کردن باینری بکار می‌رود.

-b binary: نوع فایل باینری را مشخص می‌کند که ما آن را باینری خام در نظر گرفتیم.

-m i386: معماری اسمبلی را مشخص می‌کند که آن را i386 گذاشتیم.

16M addr16,data: برای تعیین option های disassembler استفاده می شود. در این مورد، disassembler باید کد را با استفاده از آدرس دهی 16 بیتی (16addr) و عملوندهای داده 16 بیتی (16data) در نظر بگیرد. (این قسمت برای شبیه شدن کد خروجی به bootasm.S اضافه شده است.)

```
atefeh@atefeh-VirtualBox:~/xv6-public$ objdump -D -b binary -m i386 -M addr16,data16 bootblock
bootblock:      file format binary

Disassembly of section .data:

00000000 <.data>:
 0:  fa                cli
 1:  31 c0             xor    %ax,%ax
 3:  8e d8             mov    %ax,%ds
 5:  8e c0             mov    %ax,%es
 7:  8e d0             mov    %ax,%ss
 9:  e4 64             in     $0x64,%al
 b:  a8 02             test   $0x2,%al
 d:  75 fa             jne    0x9
 f:  b0 d1             mov    $0xd1,%al
11:  e6 64             out    %al,$0x64
13:  e4 64             in     $0x64,%al
15:  a8 02             test   $0x2,%al
17:  75 fa             jne    0x13
19:  b0 df             mov    $0xdf,%al
1b:  e6 60             out    %al,$0x60
1d:  0f 01 16 78 7c    lgdtw  0x7c78
22:  0f 20 c0          mov    %cr0,%eax
25:  66 83 c8 01       or     $0x1,%eax
29:  0f 22 c0          mov    %eax,%cr0
2c:  ea 31 7c 08 00    ljmp   $0x8,$0x7c31
31:  66 b8 10 00 8e d8 mov    $0xd88e0010,%eax
37:  8e c0             mov    %ax,%es
39:  8e d0             mov    %ax,%ss
3b:  66 b8 00 00 8e e0 mov    $0xe08e0000,%eax
41:  8e e8             mov    %ax,%gs
43:  bc 00 7c          mov    $0x7c00,%sp
46:  00 00             add    %al,(%bx,%si)
48:  e8 fc 00          call   0x147
4b:  00 00             add    %al,(%bx,%si)
4d:  66 b8 00 8a 66 89 mov    $0x89668a00,%eax
53:  c2 66 ef          ret     $0xef66
56:  66 b8 e0 8a 66 ef mov    $0xef668ae0,%eax
5c:  eb fe             jmp     0x5c
5e:  66 90             xchg   %eax,%eax
...
```

## 11. علت استفاده از دستور objcopy در حین اجرای عملیات make چیست؟

در makefile سیستم عامل xv6 از دستور objcopy برای کپی کردن یک فایل آبجکت در آبجکت دیگر یا تبدیل فایل های باینری کامپایل شده به فایل باینری خام استفاده کرد. در طول فرایند make سورس کد xv6 کامپایل می شود و نتیجه آن یک فایل آبجکت برای هر سورس است. سپس این فایل ها با هم لینک می شوند و یک فایل دودویی قابل اجرا با فرمت ELF ایجاد می شود.

بعد از این مرحله از دستور objcopy برای تبدیل فایل ELF به فایل دودویی خام استفاده می شود که این فایل یک binary image برای سیستم عامل ایجاد می کند که این تصویر در طی فرآیند بوت بر روی حافظه لود می شود و توسط سخت افزار اجرا می شود.

با استفاده از این دستور فایل Makefile اطمینان حاصل می کند که کد کامپایل شده 6xv به یک تصویر دودویی تبدیل شده است که می توان آن را مستقیماً بارگذاری و اجرا کرد. این امر فرآیند بوت را ساده می کند و به سیستم عامل امکان اجرا بهینه روی سخت افزار هدف را می دهد.

### **13. بوت سیستم توسط فایل های bootasb.s و bootmain.c صورت میگیرد. چرا تنها از کد C استفاده نشده است؟**

در سیستم عامل 6xv، فرآیند بوت سیستم شامل دو فایل اصلی است. دلیل استفاده از هر دو فایل C و فایل اسمبلی برای مدیریت جنبه های مختلف فرآیند بوت به صورت کارآمد است.

زبان اسمبلی سطح پایین است و کنترل مستقیم بر سخت افزار را فراهم می کند.

این فایل مسئول تنظیم محیط اولیه است، مانند تنظیم استک، فعال سازی حالت محافظت شده و انتقال از حالت واقعی به حالت محافظت شده است. کد اسمبلی برای انجام عملیات سطح پایین استفاده می شود که نیاز به کنترل دقیق بر سخت افزار دارد، مانند پیکربندی جدول بردارهای وقفه و مقداردهی واحد مدیریت حافظه (MMU).

یکی از این کارها وارد شدن به protected mode است که فقط با کد اسمبلی قابل اجراست و با کد c نمیتواند از real mode به protected mode رفت.

زبان abstraction، C، سطح بالاتری دارد و برای منطق پیچیده و مدیریت داده مناسب تر است و مسئول بارگذاری بقیه سیستم عامل به حافظه و شروع اجرای آن است. کد C برای مدیریت عملیات سیستم فایل، خواندن تصویر هسته از دیسک و انجام وظایف سطح بالاتر مورد نیاز برای بوت سیستم عامل استفاده می شود.

با استفاده از ترکیبی از کد اسمبلی و C، فرآیند بوت می تواند از کنترل سطح پایین ارائه شده توسط زبان اسمبلی بهره برداری کند و در عین حال از انتزاعات و قابلیت های سطح بالاتر ارائه شده توسط C بهره مند شود. این امر باعث می شود فرآیند بوت در 6xv به صورت کارآمد و قابل تنظیم باشد.

### **14. یک ثبات عام منظوره، یک ثبات قطعه، یک ثبات وضعیت و یک ثبات کنترلی در معماری x86 را نام برده و وظیفه هر یک را به طور مختصر توضیح دهید.**

ثبات عام منظوره: پردازنده 86x دارای 8 رجیستر عام منظوره است که یکی از آنها Accumulator register یا همان AX است که برای ذخیره خروجی محاسبات ALU استفاده می شود. در هر مرحله مقدار محاسبه شده در ALU در این رجیستر ذخیره شده و در جای دیگر به عنوان ورودی استفاده می شود.

سایر general-purpose register ها به شرح زیر هستند:

Counter register (CX). Used in shift/rotate instructions and loops.

Data register (DX). Used in arithmetic operations and I/O operations.

Base register (BX). Used as a pointer to data (located in segment register DS, when in segmented mode).

Stack Pointer register (SP). Pointer to the top of the stack.

Stack Base Pointer register (BP). Used to point to the base of the stack.

Source Index register (SI). Used as a pointer to a source in stream operations.

Destination Index register (DI). Used as a pointer to a destination in stream operations.

ثبات قطعه: این پردازنده 6 ثبات قطعه (segment register) دارد که به شرح زیر می باشند:

Stack Segment (SS). Pointer to the stack ('S' stands for 'Stack').

Code Segment (CS). Pointer to the code ('C' stands for 'Code').

Data Segment (DS). Pointer to the data ('D' stands for 'Data').

Extra Segment (ES). Pointer to extra data ('E' stands for 'Extra').

F Segment (FS). Pointer to more extra data ('F' comes after 'E').

G Segment (GS). Pointer to still more extra data ('G' comes after 'F').

برای مثال ثبات پشته برای ذخیره اطلاعات مربوط به قطعه ای است که از آن پشته برای فراخوانی استفاده می شود. ثبات کنترلی: EFLAGS register یک رجیستر 32 بیتی است که یک رشته از 0 و 1 است که هر کدام یک flag برای یک وضعیت می باشند و میتوانند درست یا غلط باشند. این فلگ ها نشان دهنده وضعیت اعمال منطقی و محاسباتی یا محدودیت های اعمال شده بر عملیات فعلی پردازنده هستند.

FLAGS -> 16-bit

EFLAGS -> 32-bit

RFLAGS -> 64-bit

Intel x86 FLAGS register						
0=	1=	Category	Description	Abbreviation	Mask	# Bit
FLAGS						
NC(No Carry)	CY(Carry)	Status	Carry flag	CF	0x0001	0
			Reserved, always 1 in EFLAGS		0x0002	1
PO(Parity Odd)	PE(Parity Even)	Status	Parity flag	PF	0x0004	2
			Reserved		0x0008	3
NA(No Auxiliary Carry)	AC(Auxiliary Carry)	Status	Adjust flag	AF	0x0010	4
			Reserved		0x0020	5
NZ(Not Zero)	ZR(Zero)	Status	Zero flag	ZF	0x0040	6
PL(Positive)	NG(Negative)	Status	Sign flag	SF	0x0080	7
		Control	Trap flag (single step)	TF	0x0100	8
DI(Disable Interrupt)	EI(Enable Interrupt)	Control	Interrupt enable flag	IF	0x0200	9
UP(Up)	DN(Down)	Control	Direction flag	DF	0x0400	10
NV(Not Overflow)	OV(Overflow)	Status	Overflow flag	OF	0x0800	11
		System	I/O privilege level (286+ only) always 1 on 8086 and 186	IOPL	0x3000	12-13
		System	Nested task flag (286+ only) always 1 on 8086 and 186	NT	0x4000	14
			Reserved always 1 on 8086 and 186 always 0 on later models		0x8000	15
EFLAGS						
		System	Resume flag (386+ only)	RF	0x0001 0000	16
		System	Virtual 8086 mode flag (386+ only)	VM	0x0002 0000	17
		System	Alignment check (486SX+ only)	AC	0x0004 0000	18
		System	Virtual interrupt flag (Pentium+)	VIF	0x0008 0000	19
		System	Virtual interrupt pending (Pentium+)	VIP	0x0010 0000	20
		System	Able to use CPUID instruction (Pentium+)	ID	0x0020 0000	21
		System	Reserved		0xFFC0 0000	22-31
RFLAGS						
			Reserved		...0xFFFF FFFF 0000 0000...	32-63

ثبات های کنترلی: این ثبات ها در رفتار کلی پردازنده یا دستگاه های مرتبط به آن نقش دارند برای مثال رجیستر OCR است که نشان دهنده تغییرات و کنترل های مختلف در رفتار کلی پردازنده است.

## 18. کد معادل S.entry در هسته لینوکس را بیابید.

<https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry.S>

## 19. چرا این آدرس فیزیکی است؟

در فرآیند بوت، جدول صفحه مسئول نگاشت آدرس‌های مجازی به آدرس‌های فیزیکی است. اما در مراحل ابتدایی بوت، سیستم حافظه مجازی هنوز تنظیم نشده است. بنابراین، استفاده از یک آدرس فیزیکی برای جدول صفحه به سیستم امکان می‌دهد تا نگاشت‌های اولیه را برقرار کرده و خود سیستم حافظه مجازی را تنظیم کند.

حافظه مجازی برای نگاشت به page table نیاز دارد و اگر آدرس آن نیز مجازی بود یک حلقه dependency بی نهایت به وجود می‌آید پس حتما آدرس جدول صفحه باید فیزیکی باشد.

علاوه بر صفحه بندی در حد ابتدایی از قطعه بندی به منظور حفاظت هسته استفاده خواهد شد. این عملیات توسط (`seginit`) انجام می‌گردد. همانطور که ذکر شد، ترجمه قطعه تاثیری بر ترجمه آدرس منطقی نمی‌گذارد. زیرا تمامی قطعه ها اعم از کد و داده روی یکدیگر می‌افتند. با این حال برای کد و داده های سطح کاربر پرچم `USER_SEG` تنظیم شده است. چرا؟ (راهنمایی: علت مربوط به ماهیت دستورات عمل‌ها و نه آدرس است.) در `6xv`، هر دو بخش هسته و کاربر دارای توصیف گرهایی در جدول توصیفگر سراسری (`GDT`) هستند. این توصیفگرها حاوی اطلاعاتی در مورد بخش ها هستند، مانند آدرس شروع، اندازه و سطح دسترسی.

هنگام اجرای یک دستورالعمل، قطعه مربوطه (اعم از کد یا داده) ابتدا از طریق توصیفگر آن در `GDT` قرار می‌گیرد. توصیفگرهای هسته و کاربر می‌توانند سطوح دسترسی متفاوتی داشته باشند، حتی اگر به حافظه فیزیکی یکسانی اشاره کنند. سطح دسترسی، امتیاز مورد نیاز برای اجرای دستورالعمل را تعیین می‌کند.

در طی این فرآیند، سطح امتیاز فعلی (`CPL`) بر اساس سطح دسترسی مشخص شده در توصیفگر تعیین می‌شود. اگر `CPL` کمتر از سطح دسترسی مورد نیاز باشد، دستور نمی‌تواند اجرا شود. برای مثال، برخی دستورالعمل‌های ممتاز ممکن است به سطح دسترسی بالاتری نسبت به آنچه برای کد سطح کاربر مجاز است نیاز داشته باشند.

وقتی `USER_SEG` تنظیم می‌شود، به این معنی است که کدهای سطح کاربر و بخش‌های داده مجوزهای محدودی دارند و از اجرای برخی عملیات ممتاز برنامه‌های سطح کاربر که می‌تواند به یکپارچگی سیستم آسیب برساند، جلوگیری کند. در مقابل، کدهای سطح `kernel` و بخش‌های داده معمولاً مجوزهای آزادتری دارند زیرا نیاز به تعامل با سخت‌افزار و مدیریت سیستم دارند.



تنظیم پرچم USER\_SEG برای بخش‌های سطح کاربر کمک می‌کند تا اطمینان حاصل شود که فرآیندهای کاربر نمی‌توانند دستورالعمل‌های ممتاز را اجرا کنند یا به بخش‌های محدود حافظه دسترسی پیدا کنند، و امنیت و پایداری سیستم عامل را افزایش می‌دهد.

قطعه‌بندی در 6xv در تابع seginit و در تکه کد زیر انجام می‌شود که در فایل vm.c وجود دارد :

```
// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
    struct cpu *c;

    // Map "logical" addresses to virtual addresses using identity map.
    // Cannot share a CODE descriptor for both kernel and user
    // because it would have to have DPL_USR, but the CPU forbids
    // an interrupt from CPL=0 to DPL=3.
    c = &cpus[cuid()];
    c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
    c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
    c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
    c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
    lgdt(c->gdt, sizeof(c->gdt));
}
```

تعریف SEG نیز به صورت زیر می‌باشد که در فایل mmu.h وجود دارد:

```
// Normal segment
#define SEG(type, base, lim, dpl) (struct segdesc) \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
#define SEG16(type, base, lim, dpl) (struct segdesc) \
{ (lim) & 0xffff, (uint)(base) & 0xffff, \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
  (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
#endif
```

## اجرای نخستین برنامه سطح کاربر

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

این ساختار که در فایل `proc.h` تعریف شده از اجزای زیر تشکیل شده است:

**uint sz:** حجم حافظه اشغال شده توسط پردازش به بایت

**pde\_t\* pgdir:** پوینتر به page table directory entry

Page table در هر پردازش یک نگاشت بین حافظه مجازی و فیزیکی ایجاد می کند.

**char \*kstack:** هر پردازش در kernel نیاز به یک stack جداگانه برای ذخیره حالتش دارد. این پوینتر به پایین ترین خانه kernel stack که به این پردازش اختصاص دارد اشاره می کند.

**enum procstate state :** وضعیت پردازش را مشخص می کند.

حالات ممکن: UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE

**int pid :** یک عدد یکتا برای هر پردازش با عنوان process identifier

**struct proc \*parent :** اشاره گر به پردازش parent. وقتی پردازش پدر تابع fork را call کرده است این پردازش با همان حالت ها و مموری و ... ایجاد شده است.

**struct trapframe \*tf :** اشاره گر به trap frame.

Trap frame آرگومان های لازم برای هندل کردن trap ها و اجرای sysCall ها را فراهم می کند و وضعیت پردازش را قبل از اجرای sysCall ذخیره می کند تا از همان حالت ادامه دهد.

**struct context \*context :** اشاره گر به ساختار context

این struct هنگام suspend شدن پردازش و سوییچ کردن به دیگری با تابع swtch محتوای رجیستر ها را ذخیره می کند تا دوباره بتواند از همان جای قبلی ادامه یابد. این ساختار شامل اجزای زیر می باشد:

edi: Destination index, for string operations

esi: Source index, for string operations

ebx: Base index, for use with arrays

ebp: Stack Base Pointer, for holding the address of the current stack frame

eip: Instruction Pointer, points to instruction to be executed

**void \*chan:** اگر مقداری غیر صفر داشته باشد یعنی پردازش در حالت sleeping است و برای انجام کاری wait می کند.

**void \*killed:** اگر مقدار غیر صفر داشته باشد یعنی پردازش kill شده است.

**struct file \*ofile :** آرایه ای پوینتر به فایل های باز شده توسط پردازش. هنگامی که فایلی باز می شود یک پوینتر در اولین خانه خالی این آرایه به آن فایل ذخیره می شود و index آن خانه به عنوان file descriptor باز گردانده می شود.

current working directory : **struct inode \*cwd**

**char name :** نام پردازش برای debug

معادل این struct در هسته لینوکس:

استراکت task\_struct

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

27. 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 می شود؟

در انتهای entry.s امکان اجرای کد C هسته فراهم می شود تا در انتها تابع main صدا زده شود و این تابع در هسته ای که سیستم عامل بوت کرده است فراخوانی می شود. هسته های دیگر از طریق

entryother.s به تابع mpenter می روند که در این تابع 4 تابع فراخوانی می شود که این چهار تابع در تابع main نیز فراخوانی شده اند. (تابع switchkvm در تابع kvmalloc که در main است فراخوانی شده است.)

به طور کلی در main 18 تابع فراخوانی شده اند که آن 4 تابع فراخوانی شده در mpenter بین تمامی هسته های پردازنده مشترک هستند و 14 تابع دیگر اختصاصی هستند.

مشترک ها: switchkvm, seginit, lapicinit, mpmain

اختصاصی ها:

kinit1,kvmalloc(setupkvm),mpinit,picinit,ioapicinit,consoleinit,

uartinit,pinit,tvinit,binit,fileinit,ideinit,startothers,kinit2,userinit

زمانبند یا scheduler در تابع mpmain صدا زده می‌شود و بین تمامی هسته های پردازنده مشترک است و هر پردازنده scheduler خودش را دارد و بعد از setup کردن خودش آن را فراخوانی می‌کند.

```
// 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
}
```

که از توضیحات قبل از تعریف scheduler قابل برداشت است:

```
// Per-CPU process scheduler.  
// Each CPU calls scheduler() after setting itself up.
```

## 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، info break استفاده کنیم.

```
(gdb) b console.c:317  
Breakpoint 1 at 0x80100bdf: file console.c, line 318.  
(gdb) b console.c:369  
Breakpoint 2 at 0x80100ce0: file console.c, line 373.  
(gdb) info breakpoint  
Num      Type      Disp Enb Address      What  
1        breakpoint keep y  0x80100bdf in arrowdown at console.c:318  
2        breakpoint keep y  0x80100ce0 in consoleintr at console.c:373
```

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

برای حذف یک breakpoint میتوان ابتدا با دستور گفته شده در قسمت قبل شماره breakpoint را بدست آورد و سپس با استفاده از دستور del به صورت del <breakpoint\_number> breakpoint، delخواه را حذف کرد.

برای مثال اگر فرض کنیم breakpoint های ما در ابتدا به صورت زیر است:

```
(gdb) info break  
Num      Type      Disp Enb Address      What  
1        breakpoint keep y  0x80100cd5 in arrowdown at console.c:319  
2        breakpoint keep y  0x80100d80 in consoleintr at console.c:373
```

سپس با وارد کردن دستور del 1 خواهیم داشت:

```
(gdb) del 1  
(gdb) info break  
Num      Type      Disp Enb Address      What  
2        breakpoint keep y  0x80100d80 in consoleintr at console.c:373
```

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؟

با فراخوانی هر تابع، آن یک stack frame برای خود در نظر می‌گیرد و مکان فراخوانی‌ها (یا همان بازگشت‌ها) و متغیرهای محلی را ذخیره می‌کند.

bt (backtrace) دستوری است که به وسیله آن در لحظه توقف برنامه می‌توان call stack برنامه را در لحظه توقف دید. در واقع آن خلاصه‌ای از مسیری است که برنامه برای رسیدن به آن نقطه طی کرده است.

خط اول مربوط frame مربوط به آخرین جایی است که برنامه در آن متوقف شده و در هر خط بعدی هر کدام صدازنده frame بالایی‌اش است.

bt [option] ... [qualifier]... [count]

backtrace [option] ... [qualifier]... [count]

: [count]

با وارد کردن دستور bt به تنهایی، تمام call stack چاپ می‌شود. برای کنترل تعداد frame های چاپ شده می‌توان از bt n و bt -n استفاده کرد.

bt n : frame n درونی‌تر را چاپ می‌کند.

bt -n : frame n بیرونی‌تر را چاپ می‌کند.

مثال [option]:

با وارد کردن bt -full در هر خط مقدار متغیرهای محلی نیز چاپ می‌شود.

در مثال زیر با قرار دادن breakpoint روی خط 8 فایل cat.c پس از وارد کردن دستور bt می‌بینیم که ابتدا در main در خط 39 تابع cat فراخوانی می‌شود و سپس به تابع cat در خط 12 رفته و در آن تابع برنامه متوقف می‌شود.

```
(gdb) b cat.c:12
Breakpoint 2 at 0x93: file cat.c, line 12.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, cat (fd=3) at cat.c:12
12     while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) bt
#0  cat (fd=3) at cat.c:12
#1  0x00000054 in main (argc=2, argv=0x2fe4) at cat.c:39
(gdb) █
```

مثالی دیگر:

```
(gdb) break console.c:136
Breakpoint 3 at 0x80100436: file console.c, line 146.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 3, cgaputc (c=c@entry=99) at console.c:146
146     if (c == '\n')
(gdb) bt
#0  cgaputc (c=c@entry=99) at console.c:146
#1  0x80100817 in consputc (c=99) at console.c:198
#2  consputc (c=99) at console.c:181
#3  cprintf (fmt=<optimized out>) at console.c:75
#4  0x801037c2 in mpmain () at main.c:54
#5  0x8010392c in main () at main.c:37
```

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

نحوه دریافت ورودی این دو دستور و نیز نحوه نمایش اطلاعات آن‌ها با هم متفاوت است. با استفاده از دستور **print** (به اختصار **p**) می‌توان مقدار یک متغیر (variable) یا یک عبارت دلخواه (arbitrary expression) را چاپ کرد.

در هر دو دستور می‌توانیم فرمت خروجی را به صورت **FMT** / در آرگومان ها مشخص کنیم.

**دستور x** : همانطور که گفته شده این دستور محتویات یک خانه حافظه را نمایش می‌دهد و به اشکال زیر قابل استفاده است:

**x** [*Address expression*]

**x** / [*Format*] [*Address expression*]

**x** / [*Length*][*Format*] [*Address expression*]

برای مشخص کردن **format** می‌توانیم از موارد زیر استفاده کنیم:

- o - octal
- x - hexadecimal
- d - decimal
- u - unsigned decimal
- t - binary
- f - floating point
- a - address
- c - char
- s - string
- i - instruction

**دستور print:** مقدار متغیر داده شده را چاپ خواهد کرد و برای مثال میتوان به شکل زیر از آن استفاده کرد:

**print** [*Expression*]

**print** [*First element*]*@*[*Element count*]

**print** */*[*Format*] [*Expression*]

برای مثال میتوان مقدار متغیر *i* را با وارد کردن دستور *print i* مشاهده کرد. همچنین می‌توانیم آرایه‌ها را با وارد کردن *\*array@length* چاپ کنیم که در آن *array* نام آرایه و *length* طولی از آرایه که مایل به چاپ آن هستیم را نشان می‌دهند.

```
Thread 1 hit Breakpoint 1, shiftleft (buf=0x801104a0 <input> "ls") at console.c:29
299      for (int i = input.e - back_count - 1; i < input.e; i++)
(gdb) print input.e
$6 = 2
(gdb) print &input.e
$7 = (uint *) 0x80110528 <input+136>
(gdb) x 0x80110528
0x80110528 <input+136>: 0x00000002
(gdb) p *input.buf@128
$8 = "ls", '\000' <repeats 125 times>
```

در ادامه باید عنوان کرد که برای مشاهده محتوای همه ثبات‌ها میتوان از دستور *info registers* استفاده کرد که میتوان از کوتاه شده آن به صورت *i r* استفاده کرد.

اگر بخواهیم تنها محتویات یک ثبات خاص را ببینیم از دستور *info registers <register\_name>* استفاده می‌کنیم که محتویات ثبات *<register\_name>* را نشان خواهد داد.



```
(gdb) info registers eax
eax                0x2                2
```

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؟

همانطور که پیش تر گفته شد میتوان برای دیدن محتویات همه رجیستر ها از دستور info registers یا r استفاده کرد.

```
(gdb) i r
eax                0x2                2
ecx                0x1                1
edx                0x0                0
ebx                0x1                1
esp                0x801169a4         0x801169a4 <stack+3764>
ebp                0x801169dc         0x801169dc <stack+3820>
esi                0x80112dc0         -2146357824
edi                0x80112dc4         -2146357820
eip                0x8010110d         0x8010110d <consoleIntr+1645>
eflags             0x2                [ IOPL=0 ]
cs                 0x8                8
ss                 0x10             16
ds                 0x10             16
es                 0x10             16
fs                 0x0                0
gs                 0x0                0
fs_base            0x0                0
gs_base            0x0                0
k_gs_base          0x0                0
cr0                0x80010011         [ PG WP ET PE ]
cr2                0x0                0
cr3                0x3ff000         [ PDBR=1023 PCID=0 ]
cr4                0x10             [ PSE ]
cr8                0x0                0
efer               0x0                [ ]
xmm0               {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}}
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}}
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}}
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}}
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}}
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}}
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}}
xmm7               {v4_float = {0x0, 0x0, 0x0, 0x1f80}, v2_double = {0x0, 0x1f80000000000000}, v16_int8 = {0x0 <repeats 16 times>, 0x80, 0x1f, 0x0, 0x0}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x1f80, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x1f80}, v2_int64 = {0x0, 0x1f80000000000000}, uint128 = 0x1f80000000000000000000000000000000}
--Type <RET> for more, q to quit, c to continue without paging--
```

برای دیدن متغیر های محلی نیز می توان از دستور info local استفاده کرد.

برای مثال متغیر های محلی مربوط به تابع consoleIntr پس از وارد کردن Ctrl+U به شکل زیر است:

```
(gdb) info local
c = 21
doprocDump = 0
```

در معماری x86، تعدادی عملیات وجود دارد که فقط با استفاده از رجیسترهای DI و SI قابل انجام هستند. این عملیات عبارتند از:

REP STOSB

REP MOVSB

که به ترتیب عملیات‌های ذخیره‌سازی تکراری (= جمع)، بارگیری و اسکن می‌باشند. شما SI و DI را برای اشاره به یک یا هر دو عملوند تنظیم می‌کنید، شاید یک شمارنده را در CX قرار دهید و سپس عملیات را انجام می‌دهید. این عملیات‌ها بر روی یک دسته از بایت‌ها به صورت همزمان عمل می‌کنند و نوعی حالت خودکار به CPU اعمال می‌کنند. زیرا شما حلقه‌ها را به صورت صریح کد نمی‌کنید.

رجیستر EDI (Extended Destination Index) معمولاً برای نشان دادن آدرس مقصد در عملیات‌های حلقه (مانند دستورات LOOP و REP) استفاده می‌شود. این رجیستر در عملیات‌هایی که نیاز به تکرار عملیات بر روی داده‌ها دارند، مورد استفاده قرار می‌گیرد.

رجیستر ESI (Extended Source Index) نیز برای نشان دادن آدرس منبع در عملیات‌های جابجایی داده‌ها (مانند دستورات MOVSB و CMPSB) استفاده می‌شود. این رجیستر در عملیات‌هایی که نیاز به انتقال داده‌ها از یک مکان به مکان دیگر دارند، مورد استفاده قرار می‌گیرد. آنها کار خود را به طور معمول به طور کارآمدتری انجام می‌دهند تا یک حلقه به صورت دستی کدنویسی شده.

به طور خلاصه، رجیستر EDI برای نشان دادن آدرس مقصد و رجیستر ESI برای نشان دادن آدرس منبع در عملیات‌های حلقه و جابجایی داده‌ها استفاده می‌شوند.

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

نکته: این توضیحات ابتدا با فرض کد داده شده (لینک گیت هاب قرار داده شده) توضیح داده می‌شود و سپس توضیحات تغییرات داده شده در کد خود گروه نیز داده می‌شود.

```
struct {
    char buf[INPUT_BUF];
    uint r; // Read index
    uint w; // Write index
    uint e; // Edit index
} input;
```

این unnamed struct در فایل console.c قرار داده شده و از یک instance آن به نام input استفاده می‌شود.

این struct دارای 4 متغیر است:

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

برای نشان دادن تغییرات مربوط به اطلاعات ایندکس‌ها (e,w,r) دو breakpoint در consoleintr قرار داده شد.

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

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

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

```
(gdb) print input
$3 = {buf = "elahe\n", '\000' <repeats 121 times>, r = 0, w = 6, e = 6}
```

سپس پس از خوانده شدن خط ( در اخر switch case در اخر قسمت default) خواهیم دید که مقدار r نیز برابر مقدار w می‌شود. در واقع r یکی یکی زیاد می‌شود و ورودی جدید رو میخواند تا به انتها(تا جایی که در آن خط ورودی نوشته شده بود) برسد.

```
(gdb) p input
$7 = {buf = "elahe\n", '\000' <repeats 121 times>, r = 6, w = 6, e = 6}
```

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

```
(gdb) p input
$2 = {buf = "elahe\nop", '\000' <repeats 119 times>, r = 6, w = 6, e = 8}
```

همچنین پس از زدن هر backspace نیز مقدار e یکی کم می‌شود.

در واقع input.e نشان دهنده جایی است که ما مینویسیم و با اضافه کردن هر حرف جدید یکی به آن اضافه می‌شود و با زدن هر backspace نیز یکی از مقدار آن کم می‌شود.

Input.w نیز در هر لحظه نشان دهنده ابتدای خطی است که در حال نوشتن هستیم. پس از وارد کردن هر enter مقدار آن برابر input.e می‌شود(می‌توان گفت به سر خط می‌رود).

Input.r نیز برای خواندن بافر استفاده می‌شود. در هر مرحله پس از زدن enter خط قبلی نوشته شده را میخواند.

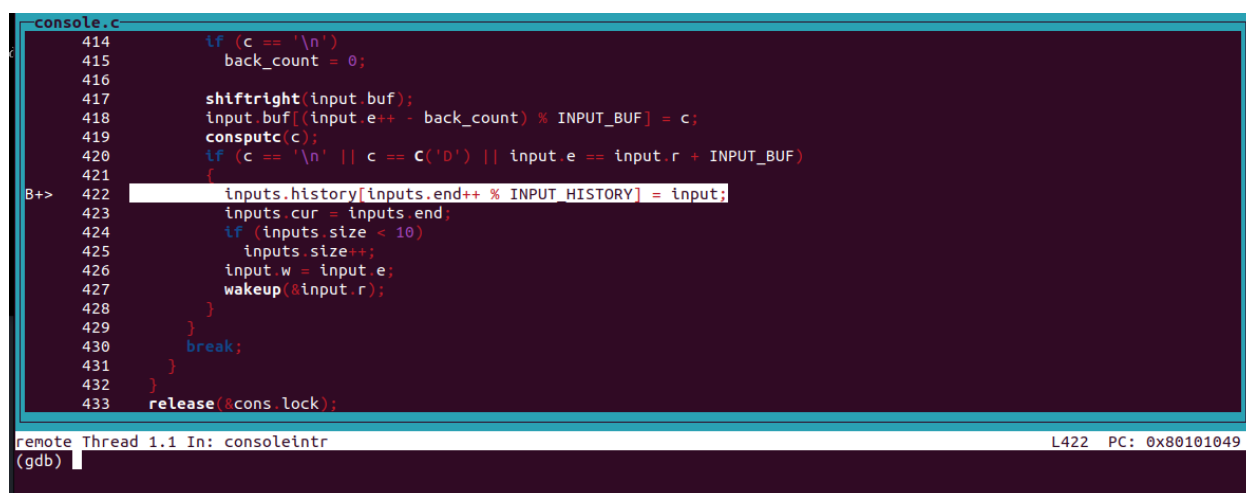
باید عنوان کرد که در کد مربوط به پروژه آزمایشگاه به دلیل نیاز به پیاده‌سازی قابلیت های عنوان شده ما در کد خود input.e را جایی که مینویسیم در نظر نگرفته و بلکه آن را انتهای خطی که مینویسیم در نظر گرفته‌ایم.

برای پیاده‌سازی قابلیت ها، input.e در کد ما نشان دهنده آخر خط و برای پیدا کردن جایی که قرار داریم و در حال ویرایش آن نقطه هستیم از متغیر back\_count استفاده کردیم که نشان دهنده تعداد خانه هایی است که از آخر خط به عقب بازگشته ایم. همچنین به علت استفاده از این struct در جای دیگر ما آن را Input نامگذاری کردیم.

## 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.



```
console.c
414     if (c == '\n')
415         back_count = 0;
416
417     shiftright(input.buf);
418     input.buf[(input.e++ - back_count) % INPUT_BUF] = c;
419     consputc(c);
420     if (c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF)
421     {
422         inputs.history[inputs.end++ % INPUT_HISTORY] = input;
423         inputs.cur = inputs.end;
424         if (inputs.size < 10)
425             inputs.size++;
426         input.w = input.e;
427         wakeup(&input.r);
428     }
429 }
430 break;
431 }
432 }
433 release(&cons.lock);

remote Thread 1.1 In: consoleIntr
(gdb) L422 PC: 0x80101049
```

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

```

0x80100c0e <consoleintr+366>  cmp    $0xa,%ebx
0x80100c11 <consoleintr+369>  je     0x80100ff0 <consoleintr+1360>
0x80100c17 <consoleintr+375>  cmp    $0xd,%ebx
0x80100c1a <consoleintr+378>  je     0x80100ff0 <consoleintr+1360>
0x80100c20 <consoleintr+384>  mov    0x8011052c,%edi
0x80100c26 <consoleintr+390>  mov    %eax,%ecx
0x80100c28 <consoleintr+392>  mov    %bl,-0x28(%ebp)
0x80100c2b <consoleintr+395>  mov    %eax,%edx
0x80100c2d <consoleintr+397>  sub    %edi,%ecx
0x80100c2f <consoleintr+399>  cmp    %ecx,%eax
0x80100c31 <consoleintr+401>  jbe    0x80100c78 <consoleintr+472>
0x80100c33 <consoleintr+403>  mov    %ebx,%esi
0x80100c35 <consoleintr+405>  lea    0x0(%esi),%esi
0x80100c38 <consoleintr+408>  mov    %edx,%eax
0x80100c3a <consoleintr+410>  sub    $0x1,%edx
0x80100c3d <consoleintr+413>  mov    %edx,%ebx
0x80100c3f <consoleintr+415>  sar    $0x1f,%ebx
0x80100c42 <consoleintr+418>  shr    $0x19,%ebx
0x80100c45 <consoleintr+421>  lea    (%edx,%ebx,1),%ecx
0x80100c48 <consoleintr+424>  and    $0x7f,%ecx

remote Thread 1.1 In: consoleintr
(gdb) layout asm
(gdb)

```

میتوان با وارد کردن layout split می‌توان هر دو کد سورس و کد اسمبلی را همزمان دید.

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

همانطور که در سوال های بالاتر دیدیم میتوانیم 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 که به ترتیب به چند تابع بالاتر و چند تابع پایین تر می‌روند. میتوان تعداد توابعی که به بالا یا پایین می‌رویم را به صورت <n> up و <n> down مشخص کنیم. در صورت مشخص نکردن تعداد مقدار default برای آن 1 در نظر گرفته می‌شود.

برای مثال اگر پشته فراخوانی مانند تصویر زیر باشد:

```

(gdb) break console.c:136
Breakpoint 3 at 0x80100436: file console.c, line 146.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 3, cgaputc (c=c@entry=99) at console.c:146
146         if (c == '\n')
(gdb) bt
#0  cgaputc (c=c@entry=99) at console.c:146
#1  0x80100817 in consputc (c=99) at console.c:198
#2  consputc (c=99) at console.c:181
#3  cprintf (fmt=<optimized out>) at console.c:75
#4  0x801037c2 in mpmain () at main.c:54
#5  0x8010392c in main () at main.c:37

```

با وارد کردن دستور up به تابع consputc در خط 198 فایل console.c می‌رویم.

## پیکربندی هسته لینوکس (امتیازی)

پس از نصب ابونتو روی VMware با استفاده از دستور عمل مربوط به این [لینک](#) یک نسخه نزدیک به هسته قبلی خود دانلود و نصب کردیم و پس از انجام مراحل با دستور `uname -a` نسخه جدید را دیدیم که نسخه 5.15.136 است.

```
fereshte@ubuntu:~$ uname -a
Linux ubuntu 5.15.136 #2 SMP Sun Oct 22 05:18:05 PDT 2023 x86_64 x86_64 x86_64 GNU/Linux
fereshte@ubuntu:~$
```

سپس با ساختن یک فایل به نام `group.c` کد مربوط به چاپ اعضای گروه را در آن گذاشته و برای آن یک `makefile` در همان جا ایجاد می‌کنیم. کد مربوط به هر فایل به صورت زیر است:

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3
4 MODULE_LICENSE("GPL");
5
6 int init_module(void)
7 {
8     printk(KERN_INFO "Group 29:\n- Ferehshte Bagheri : 810100089\n- Elahe Khodaverdi : 810100132\n- Atefe Mirzakhani : 810100220\n");
9     return 0;
10 }
11
12 void cleanup_module(void) {}
```

```
1  obj-m += group.o
2  all:
3      make -C /lib/modules/5.15.136/build M=$(PWD) modules
```

پس از وارد کردن دستور `make` در ترمینال از دستور `sudo insmod group1.ko` استفاده می‌کنیم. در نهایت اگر دستور `dmesg` را اجرا کنیم، می‌توانیم اسم اعضای گروه را در انتهای خروجی این دستور مشاهده کنیم:

```
fereshte@ubuntu:~$ make
make -C /lib/modules/5.15.136/build M=/home/fereshte modules
make[1]: Entering directory '/home/fereshte/Downloads/linux-5.15.136'
CC [M] /home/fereshte/group.o
MODPOST /home/fereshte/Module.symvers
CC [M] /home/fereshte/group.mod.o
LD [M] /home/fereshte/group.ko
BTF [M] /home/fereshte/group.ko
make[1]: Leaving directory '/home/fereshte/Downloads/linux-5.15.136'
fereshte@ubuntu:~$ ss
```

```
missing - tainting kernel
[ 1803.187927] Group 29:
- Ferehshte Bagheri : 810100089
- Elahe Khodaverdi : 810100132
- Atefe Mirzakhani : 810100220
fereshte@ubuntu:~$
```