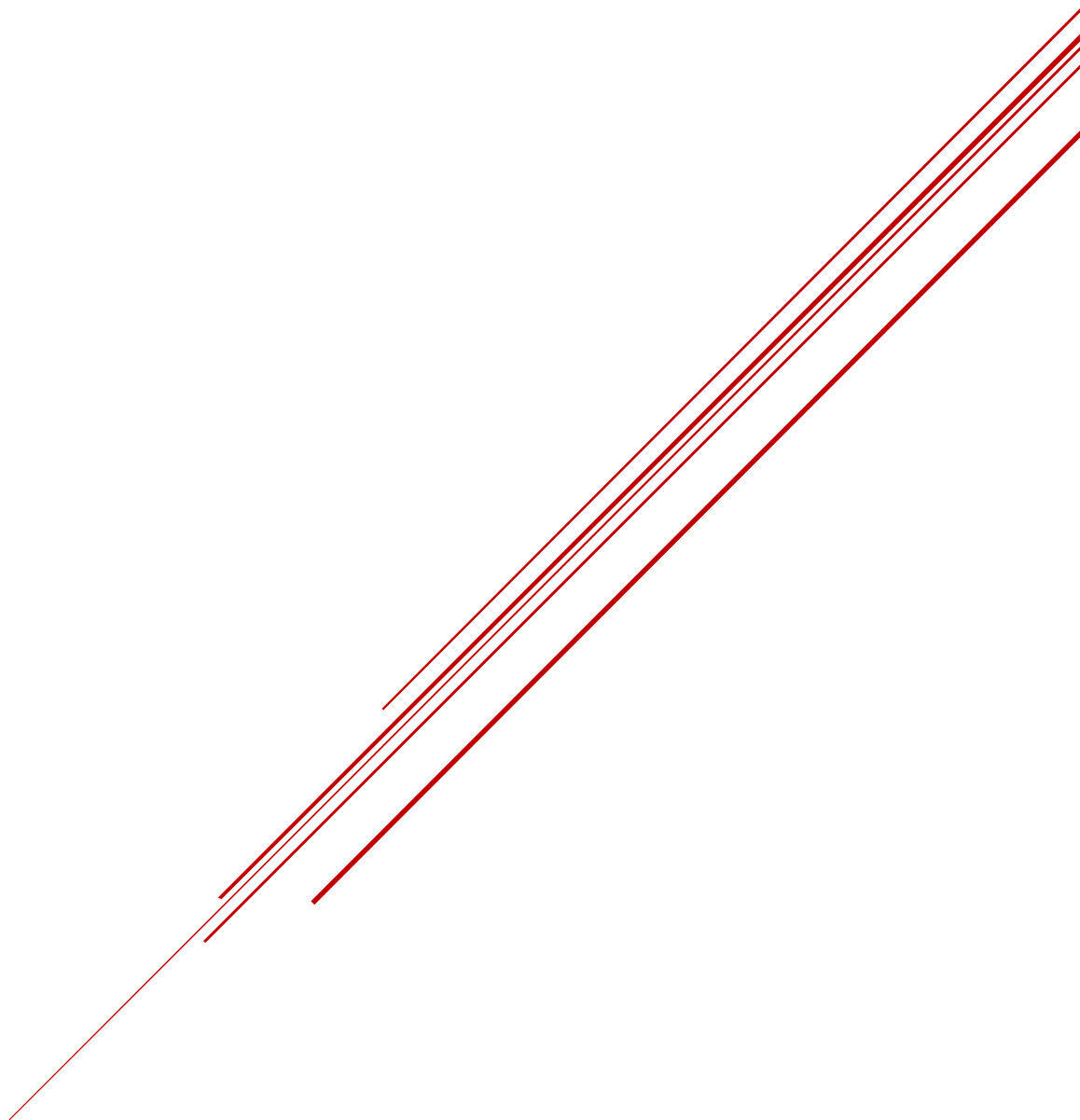


# آزمایشگاه سیستم عامل

تمرین کامپیوتری ۱



اعضای گروه:

علی حمزه پور - ۸۱۰۱۰۰۱۳۹

نرگس سادات سیدحائری - ۸۱۰۱۰۰۱۶۵

مینا شیرازی - ۸۱۰۱۰۰۲۵۰

## سوال یک: معماری سیستم عامل xv6 چیست؟ چه دلیلی در دفاع از نظر خود دارید؟

با توجه به فصل اول کتاب، xv6 به عنوان یک هسته سیستم عامل مونولیتیک پیاده سازی شده است، که از بیشتر سیستم های عامل یونیکس الگوبرداری کرده است. به همین دلیل، در xv6، رابط هسته (Kernel) معادل رابط سیستم عامل است، و هسته سیستم عامل تمام وظایف و سرویس های مربوط به سیستم عامل را پیاده سازی می کند. با این حال، از آنجا که xv6 تعداد کمتری از خدمات سیستمی را ارائه می دهد، هسته آن کوچک تر از برخی از میکرو هسته ها (Microkernels) است. به عبارت دیگر، در xv6، تمام عملکردهای سیستم عامل، از جمله مدیریت حافظه، مدیریت فایل ها، ورود و خروج داده ها و متغیرهای سیستمی و غیره، به صورت یکپارچه در هسته پیاده سازی شده است. این معماری رابطه مستقیمی بین هسته و وظایف سیستمی دارد.

## سوال دو: یک پردازش در سیستم عامل xv6 از چه بخش هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازش های مختلف اختصاص می دهد؟

یک پردازش در سیستم عامل xv6 شامل حافظه ای در فضای کاربری برای اجرای برنامه ها و ذخیره داده ها و استک است. xv6 قادر به تقسیم زمانی بین این ها است؛ به این معنی که به طور خودکار، پردازنده های مختلف را بین مجموعه ای از پردازش ها که منتظر اجرا هستند، تقسیم می کند. وقتی یک پردازش در حال اجرا نیست، xv6 اطلاعات مهمی مانند موقعیت فعلی آن در اجرای وظیفه اش را ذخیره می کند تا بتواند از همان نقطه ادامه دهد تا وقتی دوباره به اجرا درآید. هر پردازش با یک شناسه پردازش یا "pid" شناخته می شود. به این ترتیب، سیستم عامل می تواند پردازش های مختلف را مدیریت کرده و کنترل کند.

## سوال چهار: فراخوانی های سیستمی exec و fork چه عملی انجام می دهند؟ از نظر طراحی، ادغام نکردن این دو چه مزیتی دارد؟

یک پردازش ممکن است با استفاده از فراخوان سیستمی fork یک پردازش جدید ایجاد کند. Fork یک پردازش جدید را ایجاد می کند که به آن "پردازش فرزند" (child process) گفته می شود و دقیقاً همان محتوای حافظه ای را دارد که پردازش فراخواننده، که به آن "پردازش والد" (parent proces) گفته می شود، دارد. در نهایت، فراخوان سیستمی fork از هر دو پردازش فراخواننده و فرزند باز می گردد. در پردازش والد، فراخوان fork شناسه پردازش فرزند را برمی گرداند؛ و در پردازش فرزند، صفر را برمی گرداند.

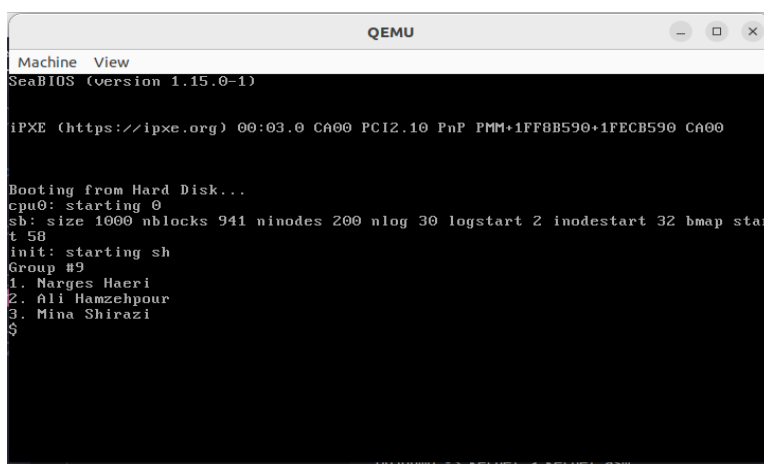
فراخوانی سیستمی exec حافظه پردازش فراخواننده را با یک تصویر حافظه جدید که از یک فایل در فایل سیستم بارگیری شده است، جایگزین می کند. این فایل باید یک فرمت خاص داشته باشد که مشخص می کند کدام قسمت از فایل دستورات را نگه می دارد، کدام قسمت داده ها را، از کجا باید اجرا آغاز شود و موارد مشابه. xv6 از فرمت ELF استفاده می کند بنابراین وقتی فراخوان exec موفقیت آمیز باشد، به برنامه فراخواننده بازگردانده نمی شود؛ به جای آن، دستورات بارگیری شده از فایل از نقطه ورودی اعلام شده در هدر ELF شروع به اجرا

می‌کنند. به عبارت دیگر، با استفاده از `exec`، می‌توان یک برنامه موجود در یک فایل جایگزین برنامه فعلی پردازش کرد تا پردازش جدید با کدها و داده‌های موجود در فایل ادامه یابد.

اگر `fork` و `exec` جدا باشند، شل (`shell`) می‌تواند یک فرزند (`child`) را ایجاد کند و در آن از توابع `open`، `close` و `dup` برای تغییر ورودی و خروجی استاندارد استفاده کند، و سپس `exec` را انجام دهد. در این روش، هیچ تغییری در برنامه‌ای که قرار است اجرا شود لازم نیست. اگر `fork` و `exec` به یک فراخوان سیستمی ترکیب شوند، به یک دستگاه دیگر (احتمالاً پیچیده‌تر) برای تغییر مسیر ورودی و خروجی توجه بیشتری نیاز دارد یا برنامه باید خود بفهمد که چگونه ورودی و خروجی را تغییر دهد.

## اضافه کردن یک متن به boot message:

صرفاً با اضافه کردن یک دستور چاپ به `init.c` (که موقع بوت برای راه‌اندازی فضای کاربر اجرا می‌شود) می‌توان نام اعضا گروه را هنگام بوت مشاهده کرد:



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

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

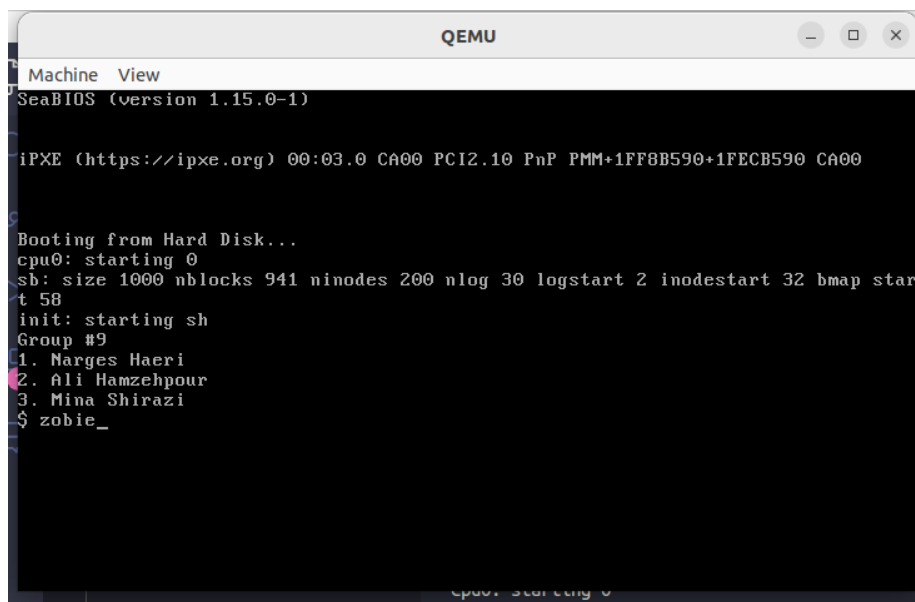
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 50
init: starting sh
Group #9
1. Narges Haeri
2. Ali Hamzehpour
3. Mina Shirazi
$
```

## اضافه کردن چند قابلیت به کنسول xv6

- **دستور `ctrl+B`:** برای پیاده‌سازی این دستور، به ساختار `input` یک متغیر دیگر اضافه کردیم تا آخرین اندیسی که در بافر نوشتیم را نشان دهد. با وجود این متغیر حالا می‌توانیم `input.e` را به عقب برگردانیم و اندیس نقطه‌ی آخر را از دست ندهیم. زمان نوشتن یک کاراکتر عادی ابتدا چک می‌کنیم اگر `input.e` با `input.end` برابر نبود، `input.buf` را از `input.e` تا `input.end` به سمت راست شیفت می‌دهیم تا برای کاراکتر جدید جا پیدا شود. (همچنین در تابع `cgaputc` که کاراکتر را در کنسول نمایش می‌دهد) موقع فعال شدن `ctrl+B` متغیر `pos` را (که نشان‌دهنده‌ی `cursor` کنسول است) یکی کم می‌کند و هنگام اضافه کردن یک کاراکتر عادی هم `crt` را از `pos` تا انتها شیفت می‌دهیم. (فاصله‌ی بین `pos` تا آخرین کاراکتر برابر فاصله‌ی `input.e` و `input.end` است.) همچنین زمانی که در وسط متن هستیم و می‌خواهیم `backspace` بزنیم، باید

کاراکترهای جلوتر از کرسر را به سمت چپ شیفت بدهیم. این کار را هم برای crt و هم برای input.buf انجام می‌دهیم. همچنین یک شرط هم قرار دادیم که اگر به input.w رسیدیم دیگر عقب‌تر نرویم. (چون به ابتدای خط رسیدیم).

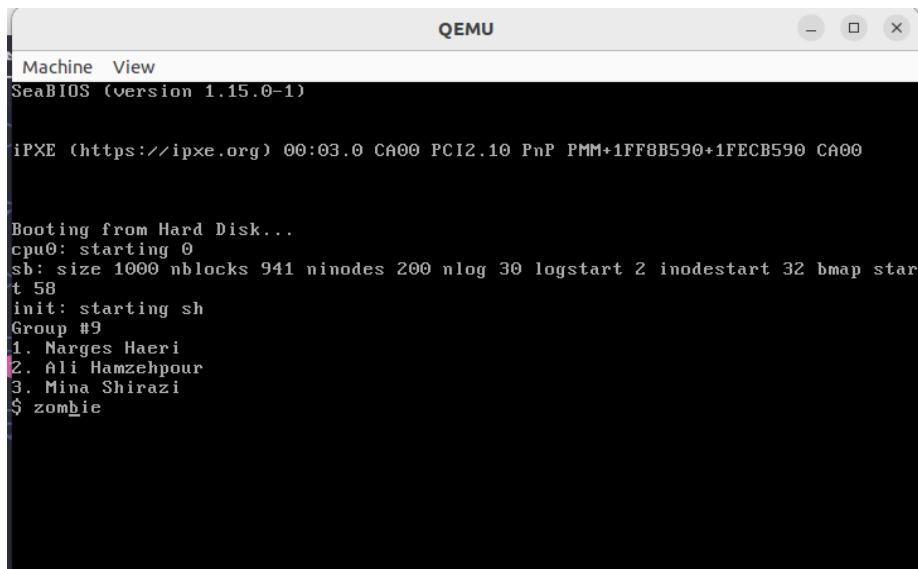
در شکل زیر ابتدا عبارت zombie را می‌نویسیم سپس به عقب برمی‌گردیم و کاراکتر m را جایگزین می‌کنیم.



```
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 #9
1. Narges Haeri
2. Ali Hamzehpour
3. Mina Shirazi
$ zombie_
```



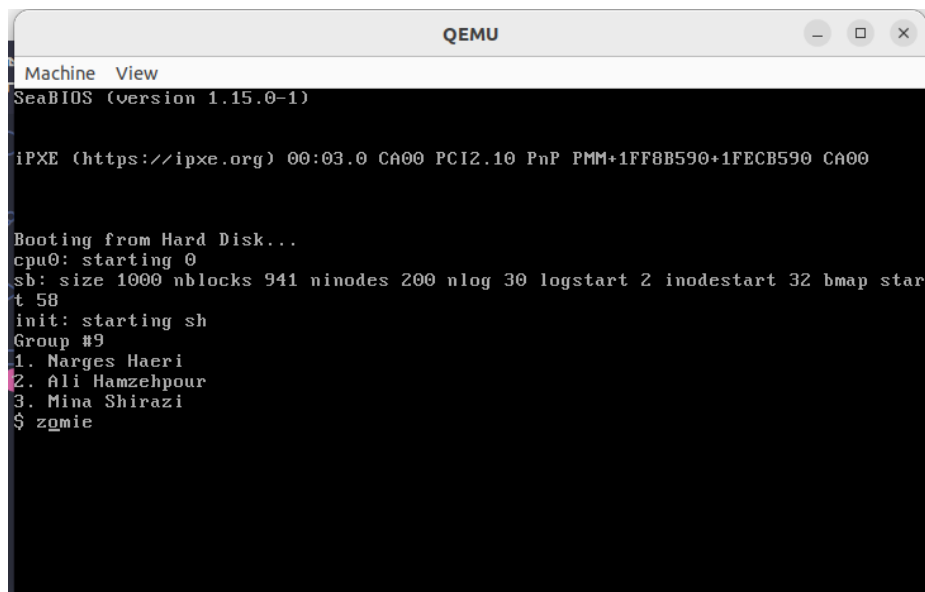
```
Machine View
SeaBIOS (version 1.15.0-1)

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 #9
1. Narges Haeri
2. Ali Hamzehpour
3. Mina Shirazi
$ zombie
```

- **دستور `ctrl + F`:** دقیقا مانند دستور قبلی پیاده‌سازی می‌شود و صرفا کرسر را باید به جای عقب به جلو ببریم. همچنین یک شرط می‌گذاریم که اگر به `input.end` رسید دیگر جلوتر نرود.

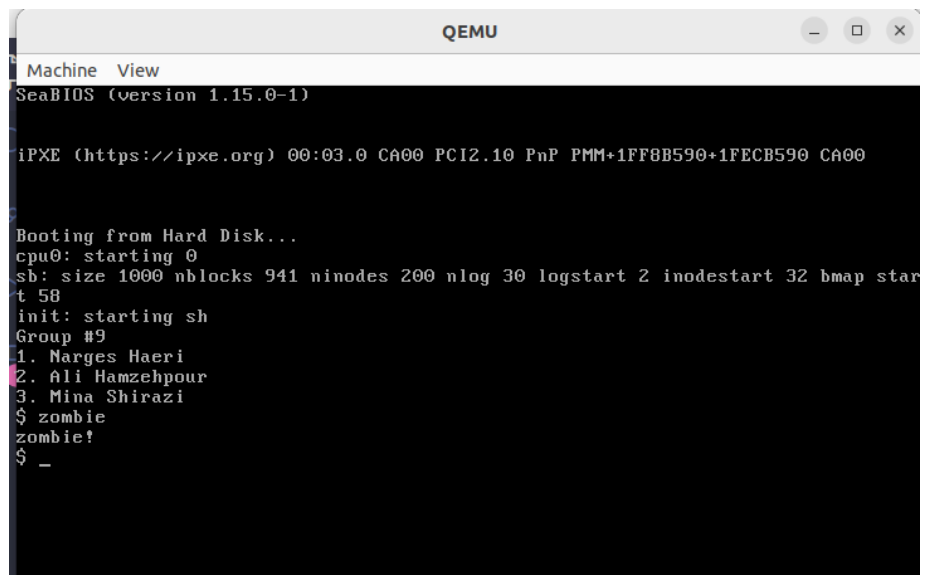
در شکل زیر ابتدا `zombie` را می‌نویسیم سپس به عقب می‌رویم و دوباره به جلو می‌رویم و `b` را اضافه می‌کنیم:



```
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 #9
1. Narges Haeri
2. Ali Hamzehpour
3. Mina Shirazi
$ zombie
```

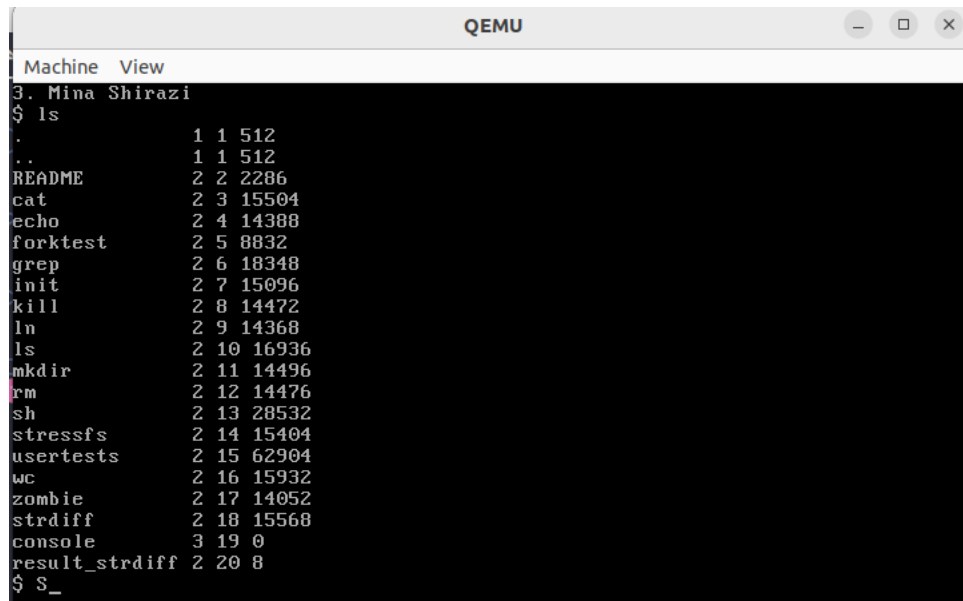


```
Machine View
SeaBIOS (version 1.15.0-1)

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

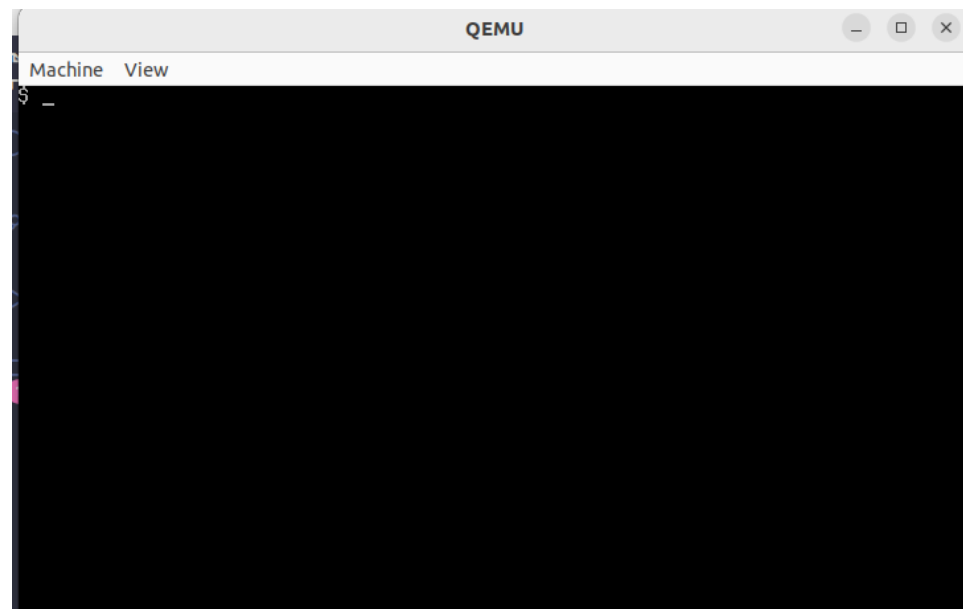
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #9
1. Narges Haeri
2. Ali Hamzehpour
3. Mina Shirazi
$ zombie
zombie!
$ -
```

- **دستور `ctrl+L`:** برای پیاده‌سازی این دستور که صفحه را پاک می‌کند، صرفاً `input.w` و `input.e` و `input.end` را با هم برابر می‌کنیم. تمام اعضای `crt` را هم برابر `blank` قرار داده و بعد یک `$` و یک `space` به کنسول اضافه می‌کنیم. برای مثال در شکل زیر، پاک‌کردن صفحه را نشان دادیم:



The screenshot shows a QEMU terminal window with a title bar that says "QEMU". Inside the terminal, the prompt is "3. Mina Shirazi" and the command "ls" has been entered. The output is a long list of files and directories with their permissions, owner, group, size, and name. The files listed are: ., .., README, cat, echo, forktest, grep, init, kill, ln, ls, mkdir, rm, sh, stressfs, usertests, wc, zombie, strdiff, console, and result\_strdiff. The prompt is now "\$ S\_".

```
Machine View
3. Mina Shirazi
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 15504
echo      2 4 14388
forktest  2 5 8832
grep      2 6 18348
init      2 7 15096
kill      2 8 14472
ln        2 9 14368
ls        2 10 16936
mkdir     2 11 14496
rm        2 12 14476
sh        2 13 28532
stressfs  2 14 15404
usertests 2 15 62904
wc        2 16 15932
zombie    2 17 14052
strdiff   2 18 15568
console   3 19 0
result_strdiff 2 20 8
$ S_
```



The screenshot shows the same QEMU terminal window after pressing `ctrl+L`. The screen is now blank, with only the prompt "\$ \_" visible at the top left. The title bar still says "QEMU".

```
Machine View
$ _
```

- دستور **arrow up** و **arrow down**: برای پیاده‌سازی این ویژگی، یک ساختار تعریف کردیم که محتوای دستورهای استفاده‌شده‌ی قبلی را در خود نگه می‌دارد:

```
#define MAX_HISTORY 10
#define MAX_HISTORY_PLUS (MAX_HISTORY + 1)

struct {
    char recent_cmds[MAX_HISTORY_PLUS][INPUT_BUF];
    ushort first;
    ushort cur;
    ushort size;
} cmd_history;
```

این ساختار شامل یک آرایه‌ی دوبعدیست که دستورهای قبلی را نگه می‌دارد و اندیس دستور اول، دستور فعلی و تعداد دستورهای استفاده‌شده نگه‌داشته شده‌است.

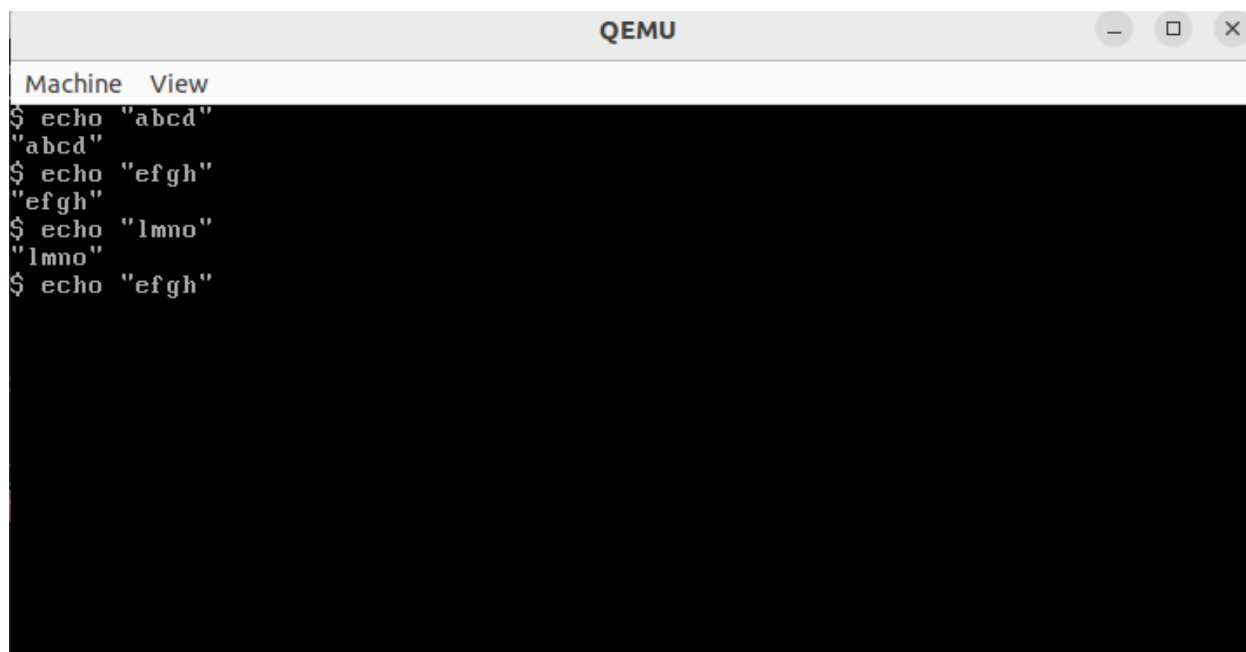
وقتی از **arrow down** استفاده می‌شود، متغیر **cur** کم می‌شود و دستور متناظر با آن اندیس نشان داده می‌شود و وقتی از **arrow up** استفاده می‌شود هم متغیر **cur** یکی زیاد می‌شود و دستور متناظر با آن باز نمایش داده می‌شود. همچنین برای هر کدام از این حالات شرط‌هایی گذاشته شده که از اولین دستور و آخرین دستور گذر نکنیم و در این مرزها متوقف شویم:



The screenshot shows a QEMU terminal window with a title bar containing the text 'QEMU' and standard window controls. The terminal displays a command history list. At the top, there are two columns: 'Machine' and 'View'. Below these, the following commands and their outputs are listed:

```
$ echo "abcd"
"abcd"
$ echo "efgh"
"efgh"
$ echo "lmno"
"lmno"
$
```

دو بار استفاده از دستور `:arrow up`:



The screenshot shows a terminal window titled "QEMU". The prompt is "\$". The first command entered is `echo "abcd"`, and the output "abcd" is displayed on the next line. The second command entered is `echo "efgh"`, and the output "efgh" is displayed on the next line. The prompt "\$" is visible at the start of each line.

```
Machine View
$ echo "abcd"
"abcd"
$ echo "efgh"
"efgh"
$ echo "lmno"
"lmno"
$ echo "efgh"
```

۳ بار استفاده از `:arrow up` و بعد استفاده از دستور `:arrow down`:



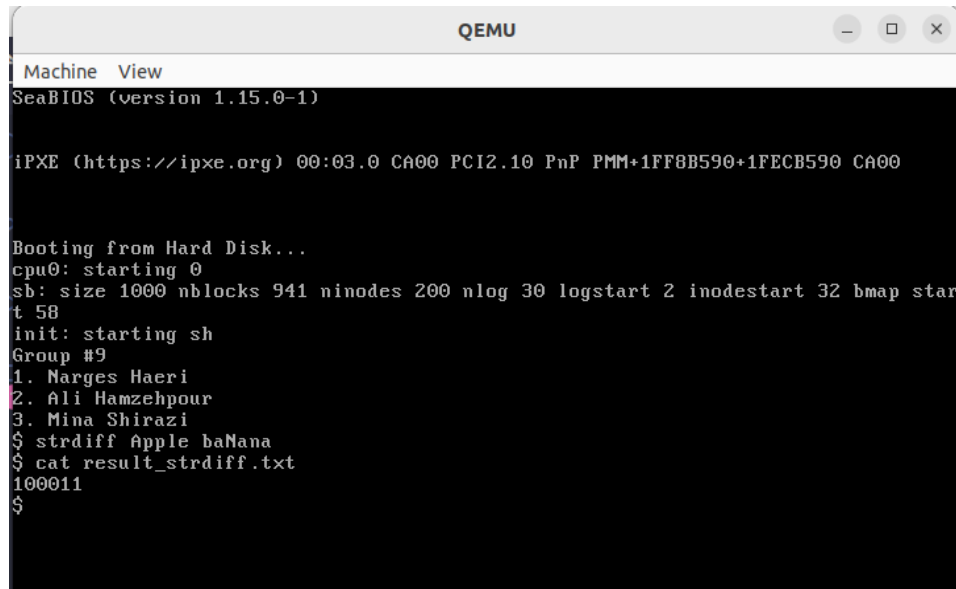
The screenshot shows a terminal window titled "QEMU". The prompt is "\$". The first command entered is `echo "abcd"`, and the output "abcd" is displayed on the next line. The second command entered is `echo "efgh"`, and the output "efgh" is displayed on the next line. The third command entered is `echo "lmno"`, and the output "lmno" is displayed on the next line. The fourth command entered is `echo "efgh"`, and the output "efgh" is displayed on the next line. The fifth command entered is `echo "lmno"`, and the output "lmno" is displayed on the next line. The prompt "\$" is visible at the start of each line.

```
Machine View
$ echo "abcd"
"abcd"
$ echo "efgh"
"efgh"
$ echo "lmno"
"lmno"
$ echo "efgh"
"efgh"
$ echo "lmno"
```



## اجرای برنامه‌ی سطح کاربر:

در این بخش منطق برنامه را در فایل `strdiff.c` پیاده‌سازی کردیم و سپس `_strdiff` را به متغیر `UPROGS` در `Makefile` اضافه کردیم تا به برنامه‌های سطح کاربر در فرآیند کامپایل اضافه شود:



```
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 #9
1. Narges Haeri
2. Ali Hamzeshpour
3. Mina Shirazi
$ strdiff Apple baNana
$ cat result_strdiff.txt
100011
$
```

**سوال هشت: در `Makefile` متغیرهایی به نام‌های `UPROGS` و `ULIB` تعریف شده است. کاربرد آنها چیست؟**

در سیستم عامل `xv6`، متغیر "`UPROGS`" که مختصر `user program` است، در فایل `Makefile` لیستی از نام‌های برنامه‌های کاربری است که باید در هنگام کامپایل `xv6` ساخته شوند. این برنامه‌ها برنامه‌هایی هستند که کاربران می‌توانند در سیستم `xv6` اجرا کنند. وقتی دستور "`make`" را برای کامپایل `xv6` اجرا می‌کنیم، `Makefile` از متغیر "`UPROGS`" برای تعیین برنامه‌های کاربری که باید به عنوان بخشی از سیستم عامل کامپایل و لینک شوند، استفاده می‌کند.

متغیر "`ULIB`" که مختصر `user libraries` است، برای مشخص کردن کتابخانه سطح کاربری استفاده می‌شود که برنامه‌های سطح کاربری در طی فرآیند کامپایل به آن لینک می‌شوند. این متغیر، کتابخانه `C` را برای برنامه‌های کاربری در `xv6` تعیین می‌کند. فایل "`ULIB`" حاوی توابع و ابزارهایی است که برنامه‌های کاربری می‌توانند از آنها برای انجام کارهای مختلف و دسترسی به توابع استاندارد کتابخانه `C` استفاده کنند.

سوال یازده: برنامه های کامپایل شده در قالب فایل های دودویی نگهداری میشوند. 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.

در سیستم عامل xv6، فایل های باینری آبجکت (object files)، از فرمت ELF (به عبارت دیگر Executable Linkable Format) پیروی می کنند.

در فایل های ELF، بخش ها (Sections) نقش مهمی ایفا می کنند. بخش ها اطلاعات مختلف را در فایل های اجرایی و کتابخانه ها ذخیره می کنند و به لینکر (linker) و لودر (loader) کمک می کنند تا کد و داده ها را به درستی ادغام و بارگذاری کنند. هر بخش در ELF یک نوع خاص دارد که تعیین کننده وظیفه اش است. برخی از نوع های معمول بخش ها شامل "text." (برای کد اجرایی)، "data." (برای داده های اجرایی)، "rodata." (برای داده های تنها خواندنی) و "bss." (برای داده های اولیه با مقدار صفر) هستند. این نوع ها به لینکر و لودر اطلاع می دهند که چگونه با هر بخش برخورد کنند.

حال دستور `objdump -h bootblock.o` را اجرا می کنیم:

```
ali@ali-virtual-machine:~/projects/Operating-System-Lab/xv6-public$ objdump -h bootblock.o
bootblock.o:      file format elf32-i386

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
 0 .text          000001c3  00007c00  00007c00  00000074  2**2
   CONTENTS, ALLOC, LOAD, CODE
 1 .eh_frame      000000b0  00007dc4  00007dc4  00000238  2**2
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .comment       0000002b  00000000  00000000  000002e8  2**0
   CONTENTS, READONLY
 3 .debug_aranges 00000040  00000000  00000000  00000318  2**3
   CONTENTS, READONLY, DEBUGGING, OCTETS
 4 .debug_info    00000505  00000000  00000000  00000358  2**0
   CONTENTS, READONLY, DEBUGGING, OCTETS
 5 .debug_abbrev  0000023c  00000000  00000000  00000add  2**0
   CONTENTS, READONLY, DEBUGGING, OCTETS
 6 .debug_line    00000283  00000000  00000000  00000b19  2**0
   CONTENTS, READONLY, DEBUGGING, OCTETS
 7 .debug_str     00000221  00000000  00000000  00000d9c  2**0
   CONTENTS, READONLY, DEBUGGING, OCTETS
 8 .debug_line_str 0000005c  00000000  00000000  00000fb0  2**0
   CONTENTS, READONLY, DEBUGGING, OCTETS
 9 .debug_loclists 0000018d  00000000  00000000  00001019  2**0
   CONTENTS, READONLY, DEBUGGING, OCTETS
10 .debug_rnglists 00000033  00000000  00000000  000011a6  2**0
   CONTENTS, READONLY, DEBUGGING, OCTETS
```

همانطور که در تصویر مشاهده می شود زمانی که این دستور را اجرا می کنیم، لیستی از هدرهای بخش ها در فایل آبجکت مشخص شده نمایش داده می شود. این اطلاعات می تواند برای درک نحوه سازماندهی و ساختار فایل آبجکت مفید باشد، اگر `bootblock.o` را با فایل های آبجکت دیگر مقایسه کنیم، متوجه می شویم که بخش های `data.` و ... را ندارد و فقط بخش `text.` را دارد.

حال دستور زیر را اجرا می کنیم:

`objcopy -S -O binary -j .text bootblock.o bootblock`

زمانی که این دستور را اجرا می‌کنیم، کد ماشینی خام از بخش "text." فایل آجکت ورودی "bootblock.o" استخراج شده و در یک فایل باینری جدید با نام "bootblock" ذخیره می‌شود. این فایل باینری می‌تواند به طور مستقیم توسط سخت‌افزار کامپیوتر بارگذاری و اجرا شود و بنابراین برای استفاده به عنوان یک بوت‌سکتور یا برنامه قابل بوت مورد استفاده قرار می‌گیرد. این جمله به این معناست که فایل "bootblock" با فرمت ELF (برخلاف بقیه فایل‌های باینری سیستم عامل xV6) مطابقتی ندارد و هیچ هدری در خود نمی‌گیرد. این فایل شامل کد اجرایی خام بدون هیچ اطلاعات اضافی‌ای می‌باشد. در واقع، این فایل تنها حاوی کد ماشینی خام (raw executable code) برای اجرا می‌باشد و هیچ ساختار یا اطلاعات اضافی ندارد. بنابراین نوع فایل دودویی مربوط به بوت raw binary می‌باشد.

دلیل اصلی این که چرا فایل "bootblock" به عنوان بوت‌سکتور در سیستم عامل xV6 از فرمت ELF استفاده نمی‌کند این است که وقتی که بوت‌سکتور اجرا می‌شود، هسته سیستم عامل هنوز اجرا نشده است و تنها پردازنده مرکزی (CPU) دارای کنترل است. CPU نمی‌تواند فرمت ELF را تشخیص دهد و قادر به خواندن آن نیست. بنابراین، برای بوت‌سکتور، تنها کدهای ماشینی خام به CPU داده می‌شود. همچنین، یک دلیل دیگر برای استفاده از کدهای ماشینی خام این است که اندازه فایل باینری کاهش می‌یابد. با استخراج تنها بخش "text." از فایل "bootblock"، حجم آن کمتر می‌شود و در ۵۱۰ بایت جا می‌گیرد. این امر دارای اهمیت ویژه برای بوت‌سکتورها است چرا که باید در ۵۱۲ بایت جا شوند تا توسط BIOS به درستی بارگذاری شوند.

بنابراین، از دلایل مهم این انتخاب استفاده از کد ماشینی خام برای بوت‌سکتور، عدم وجود وابستگی به هسته سیستم عامل و کاهش اندازه فایل برای اجرای موفقیت‌آمیز در محیط بوت کامپیوتر است.

برای تبدیل bootblock به اسمبلی، دستور زیر را اجرا می‌کنیم:

```
ali@ali-virtual-machine:~/projects/Operating-System-Lab/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
```

همانطور که در تصویر مشاهده می‌شود ابتدای خروجی بسیار مشابه با bootasm.S است.

## سوال دوازده : علت استفاده از دستور objcopy در حین اجرای عملیات make چیست؟

"Objcopy" در فرآیند "make" در xv6 برای اطمینان از اینکه مؤلفه‌های ضروری مانند بوت‌لودر و هسته در یک فرمتی باشند که به طور مستقیم توسط سخت‌افزار کامپیوتر قابل اجرا باشند، استفاده می‌شود. این فرآیند شامل حذف هدرهای ELF و تبدیل فایل‌های باینری به فرمت باینری خام برای اجرای مستقیم توسط سخت‌افزار و همچنین کاهش اندازه فایلها (بدلیل محدودیت اندازه بوت لودر) و سادگی ساختار آنها است. این اقدام ضروری است تا بوت‌لودر و هسته بتوانند به درستی بارگذاری و اجرا شوند.

## سوال چهارده: 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:** ثبات عام منظوره برای ذخیره داده‌های موقت داخل میکروپروسسور استفاده می‌شوند. میکروپروسسور ۸۰۸۶، ۸ عدد رجیستر عام منظوره دارد. از این رجیسترها می‌توان به رجیستر شمارنده اشاره کرد. این به عنوان رجیستر شمارنده count register شناخته می‌شود. ۱۶ بیت آن به دو رجیستر ۸ بیتی تقسیم می‌شود، CH و CL، که اجازه اجرای دستورات ۸ بیتی را نیز می‌دهد. این به عنوان یک شمارنده برای حلقه‌ها عمل می‌کند و توسعه حلقه‌های برنامه را تسهیل می‌دهد. دستورات شیفت/چرخش و مدیریت رشته هر دو اجازه استفاده از count register به عنوان یک شمارنده را می‌دهند.

**ثبات قطعه:** در مورد ۸۰۸۶، چهار رجیستر قطعه وجود دارد: es، ds، cs و ss. این‌ها به ترتیب نمایانگر Code Segment (رجیستر برش کد)، Data Segment (رجیستر برش داده)، Extra Segment (رجیستر برش اضافی) و Stack Segment (رجیستر برش پشته) هستند. این رجیسترها همگی ۱۶ بیتی هستند و وظیفه انتخاب بلوک‌های (برش‌های) حافظه اصلی را دارند. به عبارت دیگر، یک رجیستر برش (مانند cs) به ابتدای یک برش در حافظه اشاره می‌کند. همان طور که گفته شده یکی از اینها cs است که به برشی از حافظه اشاره می‌کند که شامل دستورات ماشینی در حال اجرا می‌باشد. با وجود محدودیت ۶۴ کیلوبایتی برش در ۸۰۸۶، برنامه‌هایی که با این محدودیت در تداخل هستند می‌توانند بیشتر از ۶۴ کیلوبایت باشند. میتوان برش‌های مختلفی از کد را در حافظه قرار داده شود. از آنجا که می‌توان مقدار رجیستر cs را تغییر دهید، می‌توانید به برش جدیدی از کد منتقل شد و دستورات موجود در آنجا را اجرا کرد.

**ثبات وضعیت:** در معماری میکروپروسسور ۸۰۸۶، ویژگی‌های "وضعیتی" خاصی وجود ندارد، مشابه ویژگی‌های معمولاً در برخی میکروپروسسورها. به جای آن، پردازنده ۸۰۸۶ از مجموعه‌ای از پرچم‌ها در رجیستر FLAGS (یا همان رجیستر وضعیت یا رجیستر پرچم) برای نمایش نتایج عملیات‌های مختلف و کنترل جریان برنامه استفاده می‌کند. یک نمونه از این پرچم‌ها پرچم DF است:

**پرچم جهت (DF):** این پرچم توسط برخی دستورهای در تعامل با رشته‌ها استفاده می‌شود. هنگامی که تنظیم شود، باعث می‌شود که عملیات‌های رشته به طور خودکار اندیس‌های رشته (SI و DI) را کاهش دهند. وقتی پاک می‌شود، اندیس‌ها به طور خودکار افزایش می‌یابند.

این پرچم‌ها برای انجام پرش‌های شرطی و تصمیم‌گیری در داخل برنامه‌ها استفاده می‌شوند. برنامه‌نویسان می‌توانند این پرچم‌ها را با استفاده از دستورات پرش شرطی برای ایجاد منطق بر اساس نتایج مختلف عملیات‌ها تست و کنترل کنند. رجیستر FLAGS که این پرچم‌ها را نگهداری می‌کند، یک رجیستر ۱۶ بیتی است که هر پرچم یک بیت آن است.

**ثبات کنترلی:** پردازنده‌های مبتنی بر معماری اینتل دارای مجموعه‌ای از ثبت‌های کنترلی هستند که برای پیکربندی پردازنده در زمان اجرا (مانند تعویض بین حالت‌های اجرا) استفاده می‌شوند. این ثبت‌ها در معماری x86 به عرض ۳۲ بیت و در معماری AMD64 (حالت بلند) به عرض ۶۴ بیت هستند.

شش رجیستر کنترلی و یک رجیستر توانایی توسعه (EFER) وجود دارند

- CR0: این رجیستر شامل انواع پرچم‌های کنترلی است که عملکرد اصلی پردازنده را تغییر می‌دهند.
- CR1: این رجیستر برای استفاده در آینده احتفاظ شده است.
- CR2: این رجیستر شامل آدرس خطای صفحه (Page Fault Linear Address) در هنگام رخ دادن خطای صفحه است.

### سوال هجده: کد معادل entry.S در هسته لینوکس را بیابید.

کد معادل entry.S برای معماری x86 در هسته لینوکس یک بخش مشترک و یک بخش مختص ۳۲ بیتی و ۶۴ بیتی دارد.

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

برای ۳۲ بیت: [https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry\\_32.S](https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry_32.S)

برای ۶۴ بیت: [https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry\\_64.S](https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry_64.S)

### سوال نوزده: چرا این آدرس فیزیکی است؟

استفاده از آدرس فیزیکی از ترجمه آدرس مجازی به آدرس فیزیکی برای دسترسی به جدول صفحه ضروری است؛ زیرا استفاده از آدرس مجازی برای آن باعث ایجاد حلقه‌های بی‌پایان می‌شود و امکان دسترسی به جدول را ناممکن می‌کند. از دلایل دیگر، ایجاد جدایی و امنیت در سیستم کامپیوتر است، که اجازه می‌دهد هر پردازنده مجموعه جداگانه‌ای از جداول صفحه داشته باشد و از دسترسی مستقیم به حافظه فیزیکی پردازنده‌های دیگر

جلوگیری کند. به علاوه، استفاده از آدرس فیزیکی از ترجمه آدرس مجازی به آدرس فیزیکی برای نرم افزار یک انتزاع فراهم می کند و مدیریت انعطاف پذیر حافظه و کنترل های امنیتی قوی را فراهم می کند.

**سوال بیست و دو :** علاوه بر صفحه بندی در حد ابتدایی از قطعه بندی به منظور حفاظت هسته استفاده خواهد شد. این عملیات توسط `seinit()` انجام میگردد. همانطور که ذکر شد، ترجمه قطعه تأثیری 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؟

در `xv6`، ترجمه آدرس ها برای کد و داده های سطح کاربر از ترجمه آدرس های هسته متفاوت است. تنظیم پرچم `"USER_SEG"` به این معناست که آدرس های کد و داده های سطح کاربر با سطح دسترسی محدودتری ترجمه می شوند تا از دسترسی غیرمجاز به مناطق حیاتی هسته جلوگیری شود.

**سوال بیست و سه:** 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.

```
struct proc {
    uint sz;                // Size of process memory (bytes)
    pde_t* pgdir;           // Page table
    char *kstack;           // Bottom of kernel stack for this process
    enum procstate state;   // Process state
    volatile int pid;       // Process ID
    struct proc *parent;    // Parent process
    struct trapframe *tf;   // Trap frame for the current interrupt
    struct context *context; // switch() here to run process
    void *chan;             // If non-zero, sleeping on chan
    int killed;             // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;       // Current directory
    char name[16];          // Process name (debugging)
};
```

در زمینه سیستم عامل `xv6`، این ساختار (`struct`) ویژگی ها و داده های مرتبط با یک پردازش را تعریف می کند. حالا به تفسیر هر متغیر در داخل این ساختار می پردازیم:

۱. `"sz"` (اندازه حافظه پردازش): این متغیر اندازه حافظه پردازش را به بایت نگه می دارد. این متغیر مقدار حافظه اختصاص یافته به پردازش برای کد و داده های آن را نمایان می سازد.
۲. `"pgdir"` (جدول صفحه): `"pgdir"` یک اشاره گر به جدول صفحه برای پردازش است. جدول صفحه برای مدیریت نگاشت حافظه مجازی به حافظه فیزیکی پردازش استفاده می شود.

۳. "kstack" (پشته هسته): این اشاره‌گر به پایین پشته هسته برای پردازش است. هر پردازش دارای پشته هسته خود است که برای ذخیره داده‌ها و اطلاعات تماس تابع در هنگام وقوع اجرا در حالت هسته مورد استفاده قرار می‌گیرد.
۴. "state" (وضعیت پردازش): "state" یک شمارگان (enumeration) است که وضعیت فعلی پردازش را نمایان می‌کند. وضعیت‌های ممکن شامل UNUSED، EMBRYO، SLEEPING، RUNNING و ZOMBIE هستند و وضعیت کنونی پردازش را توصیف می‌کنند.
۵. "pid" (شناسه پردازش): این یک مقدار عددی است که به عنوان یک شناسه یکتا برای پردازش در سیستم عامل عنوان می‌شود.
۶. "parent" (پردازش والد): "parent" اشاره‌گری به پردازش والد پردازش جاری است. این کمک می‌کند تا ارتباط والد-فرزند بین پردازش‌ها را تعیین کند.
۷. "tf" (چارچوب تله): "tf" یک اشاره‌گر به چارچوب تله است که اطلاعات بحرانی در مورد وضعیت ثبت‌های CPU در زمان وقوع نشانه یا استثناء را نگه می‌دارد. در مواقع context switching به کار می‌رود.
۸. "context" (زمینه): این اشاره‌گر به ساختار زمینه مورد استفاده برای context switching است، که به سیستم عامل اجازه می‌دهد وضعیت اجرایی پردازش را ذخیره و بازیابی کند.
۹. "chan" (کانال): "chan" یک فیلد است که نشان می‌دهد پردازش منتظر چه چیزی می‌باشد. این ممکن است مرجعی به یک شیء همگام‌سازی مانند سمافور یا قفل باشد.
۱۰. "killed": این یک پرچم عددی است که به مقدار غیرصفر تنظیم می‌شود اگر پردازش kill شده باشد، به معنای این است که باید terminate شود.
۱۱. "ofile" (فایل‌های باز): "ofile" یک آرایه از اشاره‌گرها به ساختارهای فایل باز مرتبط با پردازش است. این به پردازش اجازه می‌دهد تا پیگیری فایل‌هایی که باز کرده است را انجام دهد.
۱۲. "cwd" (پوشه کنونی): "cwd" یک اشاره‌گر به inode مربوط به پوشه کاری فعلی پردازش است.
۱۳. "name" (نام پردازش): "name" یک آرایه کاراکتری است که نام پردازش را برای اهداف اشکال‌زدایی و شناسایی پردازش نگه می‌دارد.

معادل این struct در هسته لینوکس:

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

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

بخش‌هایی که مشترک هستند:

- switchkvm
- seginit
- lapicinit
- mpmain

برای مثال تابع mpmain برای راه‌اندازی و آماده شدن هر پردازنده اجرا می‌شود و دیتا استراکچرها، کرنل استک و پوینترهای هر پردازنده را درست می‌کند، همچنین این تابع، زمان‌بند (scheduler) را هم صدا می‌زند. پس با این اوصاف باید بین پردازنده‌ها مشترک باشد.

بخش‌هایی که تنها به صورت اختصاصی و در هسته‌ی اول اجرا می‌شوند:

- kinit1
- kvmalloc
- mpinit
- picinit
- ioapicinit
- consoleinit
- uartinit
- pinit
- tvinit
- binit
- fileinit
- ideinit
- startothers
- kinit2
- userinit

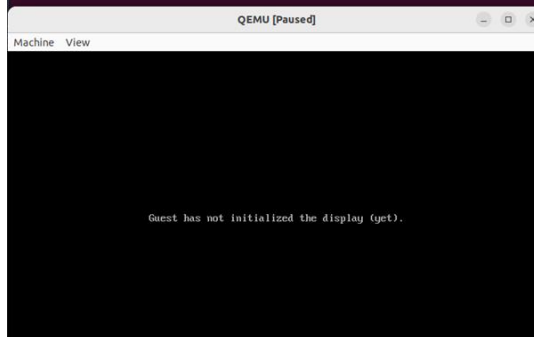
برای مثال تابع userinit برای بالاآوردن اولین برنامه‌ی سطح کاربر بعد راه‌اندازی کرنل است و فقط توسط پردازنده‌ی اول اجرا می‌شود



## GDB

همانطور که در توضیحات آزمایش آمده سیستم عامل را با حالت gdb بوت می کنیم و در ترمینالی دیگر gdb را به آن وصل می کنیم:

```
ali@ali-virtual-machine: ~/projects/Operating-System-Lab/xv6-publi$ make qemu-gdb
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S -gdb tcp::26000
```



```
ali@ali-virtual-machine: ~/projects/Operating-System-Lab/xv6-publi$ gdb kernel
GNU gdb (Ubuntu 12.1-0ubuntu1-22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel...
* target remote localhost:25000
s.gdbinit:24: Error in sourced command file:
localhost:25000: Connection timed out.
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
The target architecture is set to "i386".
[f000:ffff] 0xffff0: jmp $0x3630,$0xf000e05b
0x0000ffff in ?? ()
(gdb)
```

در این حالت دیباگر تنها روی کد سطح هسته کنترل دارد. برای مثال می توان روی تابع exec برکپوینت قرار داد تا وقتی اولین پردها بعد پردهای init شروع به کار کند، برنامه متوقف شود:

```
Remote debugging using tcp::26000
The target architecture is set to "i386".
[f000:ffff] 0xffff0: jmp $0x3630,$0xf000e05b
0x0000ffff in ?? ()
(gdb) break exec
Breakpoint 1 at 0x801013c0: file exec.c, line 20.
(gdb) continue
Continuing.
The target architecture is set to "i386".
=> 0x801013c0 <exec>: push %ebp

Thread 1 hit Breakpoint 1, exec (path=0x1c "/init", argv=0x8dffffed0) at exec.c:20
20 struct proc *curproc = myproc();
(gdb)
```

اما برای دیباگ برنامه‌ی سطح کاربر باید به دیباگر گفته‌شود که روی کد سطح کاربر کنترل انجام دهد. قبل از این کار برک‌پوینت قرارداده شده روی exec را حذف می‌کنیم.

### سوال ۱) برای مشاهده‌ی برک‌پوینت‌ها از چه دستوری استفاده می‌شود؟

با استفاده از دستور "info break" مانند شکل زیر می‌توان برک‌پوینت‌ها را مشاهده کرد:

```
no breakpoint at 1.  
(gdb) info break  
Num      Type      Disp Enb Address      What  
1        breakpoint keep y   0x801013c0 in exec at exec.c:20  
breakpoint already hit 1 time  
(gdb) delete 1
```

### سوال ۲) برای حذف برک‌پوینت از چه دستوری و چگونه استفاده می‌شود؟

با استفاده از دستور "delete breakpoint\_num" می‌توان یک برک‌پوینت با شماره‌ی مشخص را حذف کرد. برای مثال برای حذف برک‌پوینت شکل قبل اینگونه عمل می‌کنیم:

```
(gdb) delete 1  
(gdb) info break  
No breakpoints or watchpoints.  
(gdb)
```

همانطور که می‌بینید پس از این دستور دیگر برک‌پوینتی نداریم.

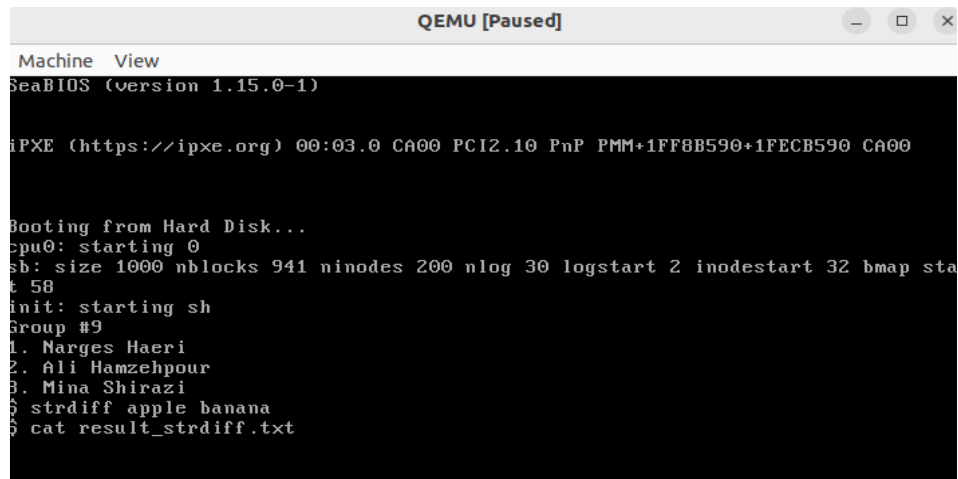
حالا دیباگر را به کد سطح کاربر متصل می‌کنیم. برای این کار از دستور file استفاده می‌کنیم:

```
(gdb) file _cat  
A program is being debugged already.  
Are you sure you want to change the file? (y or n) y  
Load new symbol table from "_cat"? (y or n) y  
Reading symbols from _cat...  
(gdb)
```

حالا دیباگر به برنامه‌ی سطح کاربر دسترسی دارد و می‌توان در آن کد برک‌پوینت قرار داد:

```
(gdb) break cat.c:12
Breakpoint 2 at 0x93: file cat.c, line 12.
(gdb) info break
Num      Type           Disp Enb Address      What
2        breakpoint    keep y   0x00000093   in cat at cat.c:12
(gdb)
```

اجرا را ادامه می‌دهیم و در سیستم‌عامل از دستور cat استفاده می‌کنیم تا برک‌پوینت فعال شود:



```
QEMU [Paused]
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 sta
t 50
init: starting sh
Group #9
1. Narges Haeri
2. Ali Hamzhepour
3. Mina Shirazi
5 strdiff apple banana
$ cat result_strdiff.txt
```

```
(gdb) continue
Continuing.
The target architecture is set to "i8086".
[ 1b: 93] 0x243 <gets+19>: push $0x1

Thread 1 hit Breakpoint 2, cat (fd=3) at cat.c:12
12 while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb)
```

در این حالت می‌توان با اجرای دستورهای step، next و finish برنامه را گام‌به‌گام اجرا کرد:

- دستور next: به خط بعدی اجرا شده می‌رود اما وارد کد توابع دیگر نمی‌شود.

```
(gdb) next
[ 1b: a0] 0x250 <gets+32>: test %eax,%eax
13 if (write(1, buf, n) != n) {
(gdb)
```

- دستور step: به خط بعدی اجرا شده می‌رود و در صورت نیاز وارد کد توابع دیگر هم می‌شود.

```
12     while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) step
[ 1b: 37b] 0x52b <printf+107>: je 0x638 <printf+376>
read () at usys.S:15
15     SYSCALL(read)
(gdb) █
```

- دستور finish: تابع فعلی را تا زمانی که به اتمام برسد اجرا می‌کند.

```
Thread 1 hit breakpoint 2, cat (fd=3) at cat.c:12
12     while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) finish
Run till exit from #0 cat (fd=3) at cat.c:12
[ 1b: 54] 0x204 <strchr+20>: je 0x22c
main (argc=2, argv=0x2fdc) at cat.c:40
40     close(fd);
(gdb) █
```

### سوال ۳) خروجی دستور "bt" چیست؟

این دستور که مخفف backtrace است، نشان می‌دهد که در برنامه‌ی فعلی چه توالی از توابع صدا زده شده تا به لحظه‌ی فعلی برسیم. برای مثال شکل زیر بیان می‌کند که ابتدا در تابع main فایل cat.c قرار داشتیم و در آن تابع cat صدا زده شده و بعد به خط فعلی رسیدیم.

```
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=0x2fdc) at cat.c:39
(gdb) █
```

### سوال ۴) دو تفاوت دستورهای x و print را توضیح دهید. چگونه می‌توان یک رجیستر خاص را چاپ کرد؟

با استفاده از دستور print می‌توان مقدار یک متغیر در لحظه‌ی کنونی را چاپ کرد. از دستور x برای مشاهده‌ی محتویات یک خانه‌ی حافظه می‌توان استفاده کرد. همچنین در دستور x می‌توان فرمت چاپ محتوای حافظه را هم مشخص کرد.

در حالت کلی از دستور print برای بررسی مقدار متغیرها و عبارات در لقطه‌ی فعلی استفاده می‌شود درحالی‌که از x برای بررسی محتوای خام حافظه و بررسی آن در فرمت‌های مختلف استفاده می‌شود.

```
(gdb) print fd
$2 = 3
(gdb) x/d &fd
0x2f90: 3
(gdb)
```

برای چاپ کردن محتوای یک رجیستر خاص هم از دستور "info registers register\_num" می‌توان استفاده کرد:

```
(gdb) info registers cx
cx          0x2fd4          12244
(gdb)
```

سوال ۵) برای نمایش وضعیت رجیسترها و متغیرهای محلی از چه دستوری استفاده می‌شود؟ همچنین توضیح دهید در معماری x86 رجیسترهای edi و esi نشانگر چه چیزی هستند.

```
(gdb) info registers
eax          0x3          3
ecx          0x2fd4          12244
edx          0xbfac          49068
ebx          0x2fe4          12260
esp          0x2f88          0x2f88
ebp          0x2f88          0x2f88
esi          0x2          2
edi          0x3          3
eip          0x93          0x93 <cat+3>
eflags       0x206          [ IOPL=0 IF PF ]
gs           0x1b          27
ss           0x23          35
ds           0x23          35
es           0x23          35
fs           0x0          0
gs           0x0          0
fs_base      0x0          0
gs_base      0x0          0
k_gs_base    0x0          0
cr0          0x80010011          [ PG WP ET PE ]
cr2          0x0          0
cr3          0xdf13000          [ POBDR=57107 PCID=0 ]
cr4          0x10          [ PSE ]
cr8          0x0          0
efer         0x0          [ ]
xmm0         {v4_float = (0x0, 0x0, 0x0, 0x0), v2_double = (0x0, 0x0), v16_int8 = (0x0 <repe
ats 16 times>), v8_int16 = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0), v4_int32 = (0x0, 0x0, 0x0
, 0x0), v2_int64 = (0x0, 0x0), uint128 = 0x0}
xmm1         {v4_float = (0x0, 0x0, 0x0, 0x0), v2_double = (0x0, 0x0), v16_int8 = (0x0 <repe
ats 16 times>), v8_int16 = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0), v4_int32 = (0x0, 0x0, 0x0
, 0x0), v2_int64 = (0x0, 0x0), uint128 = 0x0}
xmm2         {v4_float = (0x0, 0x0, 0x0, 0x0), v2_double = (0x0, 0x0), v16_int8 = (0x0 <repe
ats 16 times>), v8_int16 = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0), v4_int32 = (0x0, 0x0, 0x0
, 0x0), v2_int64 = (0x0, 0x0), uint128 = 0x0}
xmm3         {v4_float = (0x0, 0x0, 0x0, 0x0), v2_double = (0x0, 0x0), v16_int8 = (0x0 <repe
ats 16 times>), v8_int16 = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0), v4_int32 = (0x0, 0x0, 0x0
, 0x0), v2_int64 = (0x0, 0x0), uint128 = 0x0}
xmm4         {v4_float = (0x0, 0x0, 0x0, 0x0), v2_double = (0x0, 0x0), v16_int8 = (0x0 <repe
ats 16 times>), v8_int16 = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0), v4_int32 = (0x0, 0x0, 0x0
, 0x0), v2_int64 = (0x0, 0x0), uint128 = 0x0}
xmm5         {v4_float = (0x0, 0x0, 0x0, 0x0), v2_double = (0x0, 0x0), v16_int8 = (0x0 <repe
ats 16 times>), v8_int16 = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0), v4_int32 = (0x0, 0x0, 0x0
, 0x0), v2_int64 = (0x0, 0x0), uint128 = 0x0}
xmm6         {v4_float = (0x0, 0x0, 0x0, 0x0), v2_double = (0x0, 0x0), v16_int8 = (0x0 <repe
ats 16 times>), v8_int16 = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0), v4_int32 = (0x0, 0x0, 0x0
, 0x0), v2_int64 = (0x0, 0x0), uint128 = 0x0}
xmm7         {v4_float = (0x0, 0x0, 0x0, 0x0), v2_double = (0x0, 0x0), v16_int8 = (0x0 <repe
ats 16 times>), v8_int16 = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0), v4_int32 = (0x0, 0x0, 0x0
, 0x0), v2_int64 = (0x0, 0x0), uint128 = 0x0}
xmm8         {v4_float = (0x0, 0x0, 0x0, 0x0), v2_double = (0x0, 0x0), v16_int8 = (0x0 <repe
ats 16 times>), v8_int16 = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0), v4_int32 = (0x0, 0x0, 0x0
, 0x0), v2_int64 = (0x0, 0x0), uint128 = 0x0}
```

```
(gdb) info locals
n = <optimized out>
```

با استفاده از دستورهای info locals و info registers می‌توان وضعیت رجیسترها و متغیرهای محلی را مشاهده کرد:

- رجیستر edi مخفف extended destination index بوده و به عنوان اندیس مقصد در عملیات‌های مربوط به رشته استفاده می‌شود.
- رجیستر esi هم مخفف extended source index بوده و به عنوان اندیس مبدا در عملیات‌های مربوط به رشته استفاده می‌شود.

**سوال ۶) با استفاده از GDB محتویات متغیر input را بررسی کنید و نحوه و زمان تغییر آن را توضیح دهید.**

متغیر input یک متغیر global در فایل console.c است که وظیفه‌ی آن ذخیره کردن محتویات دستور فعلی کاربر است.

با استفاده از دستور ptype می‌توانیم تعریف آن را بررسی کنیم:

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

این ساختار شامل متغیرهای زیر است:

- متغیر buf: کاراکترهای دستور در این بافر ذخیره می‌شود.
- متغیر r: برای خواندن بافر از آن استفاده می‌شود.
- متغیر w: نشان‌دهنده‌ی اندیس اولین کاراکتر دستور جدید در بافر است.
- متغیر e: نشان‌دهنده‌ی اندیس مکانی‌ست که کرسر قرار دارد و در آن قرار است بنویسیم (اختصار یافته‌ی edit)
- متغیر end: یک متغیر کمکی‌ست که ما به ساختار اضافه کردیم و اندیس انتهای دستور فعلی را در بافر نشان می‌دهد.

حالا در خطی از فایل console.c که کاراکتر "\n" بررسی می‌شود برک‌پوینت می‌گذاریم (سیستم عامل و دیباگر را قبل از این مرحله ری‌استارت کردیم):

```
(gdb) break console.c:374
Breakpoint 4 at 0x80100d27: file console.c, line 374.
```

حالا قبل از اینکه دستور بعدی را وارد کنیم، با استفاده از ctrl+c برنامه را متوقف می‌کنیم و محتویات input را چاپ می‌کنیم:

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

همانطور که می‌بینید در ابتدا بافر خالی‌ست و مقادیر دیگر هم صفر هستند.

برنامه رو ادامه می‌دهیم و دستور ls را وارد کنسول می‌کنیم و اینتر را می‌زنیم. برنامه متوقف می‌شود و باز input را چاپ می‌کنیم:

```
Thread 1 hit Breakpoint 2, consoleintr (getc=0x80103000 <kbdgetc>) at console.c:374
374         if(c == '\n' || c == C('D') || input.end == input.r+INPUT_BUF){
(gdb) print input
$2 = {buf = "ls", '\000' <repeats 125 times>, r = 0, w = 0, e = 2, end = 2}
(gdb)
```

بافر عبارت ls را در خود دارد و مقادیر e و end به ۲ تغییر کرده‌اند (که همان طول طول دستور است). با استفاده از next چند خط در کد جلو می‌رویم و باز input را چاپ می‌کنیم:

```
(gdb) next
=> 0x80101096 <consoleintr+1414>:    mov     %eax,0x8011042c
375         input.buf[input.end++ % INPUT_BUF] = c;
(gdb) next
=> 0x801010a7 <consoleintr+1431>:    call    0x80100a20 <push_command_to_history>
376         push_command_to_history();
(gdb) next
=> 0x801010ac <consoleintr+1436>:    sub     $0xc,%esp
377         input.e = input.end;
(gdb) next
=> 0x801010be <consoleintr+1454>:    mov     %eax,0x80110424
378         input.w = input.e;
(gdb) next
=> 0x801010c3 <consoleintr+1459>:    call    0x80104a10 <wakeup>
379         wakeup(&input.r);
(gdb) print input
$3 = {buf = "ls\n", '\000' <repeats 124 times>, r = 0, w = 3, e = 3, end = 3}
(gdb)
```

در این مرحله "\n" هم به بافر اضافه شده و مقادیر w,e و end به ۳ رسیدند اما r همچنان صفر است. دلیل آن این است که در مرحله‌ی خواندن و اجرای دستور r را تا w جلو می‌بریم و دستور را می‌خوانیم. باز continue را می‌زنیم و این بار دستور zombie را وارد می‌کنیم:

```
Thread 1 hit Breakpoint 3, consoleintr (getc=0x80103000 <kbdgetc>) at console.c:375
375      input.buf[input.end++ % INPUT_BUF] = c;
(gdb) print input
$7 = {buf = "ls\nzombie", '\000' <repeats 118 times>, r = 3, w = 3, e = 9, end = 9}
(gdb)
```

مقدار ۲ برابر ۳ شده. این به این معنیست که در این بین دستور ls با استفاده از ۲ از روی بافر خوانده شده. باز دستور zombie را وارد می‌کنیم ولی این بار قبل از اینتر کرسر را به عقب می‌بریم:

```
(gdb) print input
$8 = {buf = "ls\nzombie\nzombie", '\000' <repeats 111 times>, r = 10, w = 10, e = 13, end = 16}
(gdb)
```

در این حالت e با end یکی نیست چون هنگام رفتن به خط بعد جای کرسر آخر خط نبوده است. پس در حالت کلی با تغییراتی که در کد دادیم، e همیشه اندیس جاییست که کرسر قرار دارد، end اندیس انتهای خط است، w هم اندیس ابتدای خط و r هم اندیس ابتدای خط است، اما نسبت به w دیرتر آپدیت می‌شود تا در فرآیند خواندن دستور به کار بیاید. در پایان هم وقتی دستور نوشته و خوانده شد، تمامی مقادیر یکی می‌شوند:

```
(gdb) continue
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
=> 0x80104211 <mycpu+17>:      mov     0x80112ca4,%esi
mycpu () at proc.c:48
48      for (i = 0; i < ncpu; ++i) {
(gdb) print input
$9 = {buf = "ls\nzombie\nzombie\n", '\000' <repeats 110 times>, r = 17, w = 17, e = 17, end = 17}
(gdb)
```

## سوال ۷) خروجی دستورهایی layout src و layout asm در tui چیست؟

دستور layout src سورس کد در حالت دیباگ را به ما نشان می‌دهد و دستور layout asm سورس اسمبلی همان را به ما نشان می‌دهد.:



ali@ali-virtual-machine: ~/projects/Operating-System-Lab/xv6-public

```

proc.c
32 return mycpu()-cpus;
33 }
34 // Must be called with interrupts disabled to avoid the caller being
35 // rescheduled between reading lapicid and running through the loop.
36 struct cpu*
37 mycpu(void)
38 {
39     int apicid, i;
40     if(readeflags() & FL_IF)
41         panic("mycpu called with interrupts enabled\n");
42     apicid = lapicid();
43     // APIC IDs are not guaranteed to be contiguous. Maybe we should have
44     // a reverse map, or reserve a register to store &cpus[i].
45     for (i = 0; i < ncpu; ++i)
46         if (cpus[i].apicid == apicid)
47             return &cpus[i];
48     panic("unknown apicid\n");
49 }
50 // Disable interrupts so that we are not rescheduled
51 // while reading proc from the cpu structure
52 struct proc*
53 myproc(void) {
54     struct cpu *c;
55     struct proc *p;

```

remote Thread 1.1 In: mycpu
L48 PC: 0x80104211
(gdb) layout src
(gdb)

ali@ali-virtual-machine: ~/projects/Operating-System-Lab/xv6-public

```

0x80104260 <cpu0d> push %ebp
0x80104261 <cpu0d+1> mov %esp,%ebp
0x80104263 <cpu0d+3> sub $0x8,%esp
0x80104266 <cpu0d+6> call 0x80104200 <mycpu>
0x8010426b <cpu0d+11> leave
0x8010426c <cpu0d+12> sub $0x80112cc0,%eax
0x80104271 <cpu0d+17> sar $0x4,%eax
0x80104274 <cpu0d+20> imul $0xba2e8ba3,%eax,%eax
0x8010427a <cpu0d+26> ret
0x8010427b lea 0x0(%esi,%eiz,1),%esi
0x8010427f nop
0x80104280 <myproc> push %ebp
0x80104281 <myproc+1> mov %esp,%ebp
0x80104283 <myproc+3> push %ebx
0x80104284 <myproc+4> sub $0x4,%esp
0x80104287 <myproc+7> call 0x80104d00 <pushcli>
0x8010428c <myproc+12> call 0x80104200 <mycpu>
0x80104291 <myproc+17> mov 0xac(%eax),%ebx
0x80104297 <myproc+23> call 0x80104db0 <popcli>
0x8010429c <myproc+28> mov %ebx,%eax
0x8010429e <myproc+30> mov -0x4(%ebp),%ebx
0x801042a1 <myproc+33> leave
0x801042a2 <myproc+34> ret
0x801042a3 lea 0x0(%esi,%eiz,1),%esi
0x801042aa lea 0x0(%esi),%esi
0x801042b0 <userinit> push %ebp
0x801042b1 <userinit+1> mov %esp,%ebp
0x801042b3 <userinit+3> push %ebx
0x801042b4 <userinit+4> sub $0x4,%esp

```

remote Thread 1.1 In: mycpu
L48 PC: 0x80104211
(gdb) layout src
(gdb) layout asm
(gdb)

سوال ۸) برای جابه‌جایی میان توابع زنجیره فراخوانی جاری از چه دستورهای استفاده می‌شود؟

remote Thread 1.1 In: popcli
L121 PC: 0x80104dc2
(gdb) layout asm
(gdb) bt
#0 mycpu () at proc.c:48
#1 0x80104dc2 in popcli () at spinlock.c:121
#2 0x80104e89 in holding (lock=0x80113240 <ptable>) at spinlock.c:95
#3 release (lk=0x80113240 <ptable>) at spinlock.c:49
#4 0x801045d1 in scheduler () at proc.c:353
#5 0x8010394f in mpmain () at main.c:57
#6 0x80103a9c in main () at main.c:37
(gdb) up
#1 0x80104dc2 in popcli () at spinlock.c:121
(gdb) up
#2 0x80104e89 in holding (lock=0x80113240 <ptable>) at spinlock.c:95
(gdb) down
#1 0x80104dc2 in popcli () at spinlock.c:121
(gdb)

از دستور bt برای دیدن این زنجیره و از دستورهای up و down برای جابه‌جایی در این زنجیره استفاده می‌شود.

## هسته لینوکس (امتیازی)

به دلیل جلوگیری از خطاهای پیش‌بینی نشده و سادگی کار از فایل پیکربندی پیش‌فرض سیستم عامل استفاده کردیم.

تصویر زیر نسخه‌ی هسته قبل از نصب هسته‌ی جدید را نشان می‌دهد (از اوبونتو ۲۲/۰۴ در vmware استفاده شد)

```
test@test-virtual-machine:~$ uname -a
Linux test-virtual-machine 5.15.0-43-generic #46-Ubuntu SMP Tue Jul 12 10:30:17
UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
test@test-virtual-machine:~$
```

نسخه‌ی هسته در ابتدا ۵/۱۵/۰ بود و بعد از کامپایل و نصب کرنل جدید به ۵/۱۵/۱۳۵ ارتقا پیدا کرد:

```
test@test-virtual-machine: ~
test@test-virtual-machine:~$ uname -a
Linux test-virtual-machine 5.15.135 #1 SMP Thu Oct 19 21:12:17 +0330 2023 x86_64
x86_64 x86_64 GNU/Linux
test@test-virtual-machine:~$
```

همچنین فایل جدیدی به اسم group۹.c ساخته شد که در آن از دستور printk برای چاپ اعضای گروه استفاده کردیم تا در دستور dmesg نمایش دهد. سپس این فایل را بعد از make، با استفاده از دستور sudo insmod group۹.ko آن را به ماژول‌های کرنل اضافه کردیم.

کد و میک‌فایل:

```
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");

int init_module(void){
    printk(KERN_INFO "Group 9:\n1- Narges Haeri\n2- Ali Hamzehpour\n3-Mina Shirazi");
    return 0;
}

void cleanup_module(void) {}
```

```
CONFIG_MODULE_SIG=n
obj-m += group9.o

all:
    make -C /lib/modules/5.15.135/build M=$(PWD) modules

~
```

نمایش اعضای گروه در یکی از پیام‌های dmesg:

```
[ 1645.717495] group9: loading out-of-tree module taints kernel.
[ 1645.717551] group9: module verification failed: signature and/or required key missing - tainting kernel
[ 1645.718411] Group 9:
                1- Narges Haeri
                2- Ali Hamzeshpour
                3-Mina Shirazi
```