



گزارش پروژه اول آزمایشگاه سیستم عامل

به تدریس دکتر کارگهی

محمد امین یوسفی

مبینا مهرآذر

متین نبی زاده

پاییز 1402

آشنایی با سیستم عامل xv6

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

در ابتدا، گزارش را با ذکر نکاتی کلی از ساختار و معماری این سیستم عامل، آغاز می‌کنیم. معماری این سیستم عامل مانند بسیاری از سیستم عامل‌های دیگر به شکل چندکاره¹ است. معماری اصلی xv6 از چندین بخش ساختار هسته²، پردازنده مرکزی و فرآیندهای سیستم عامل، سیستم فایل‌ها، مدیریت ورودی و خروجی و مدیریت حافظه تشکیل شده است. مدیریت ورودی و خروجی به منظور ارتباط شامل توانایی ارتباط با دستگاه‌های ورودی و خروجی (نظیر کیبورد یا صفحه نمایش)، می‌باشد. سیستم فایل Unix، توانایی ایجاد، خواندن، نوشتن، حذف فایل‌ها و دایرکتوری‌ها را ایجاد می‌کند. مدیریت حافظه، مکانیزم حفاظت از حافظه، Paging و ... فراهم می‌کند. در این معماری، هسته، وظیفه مدیریت منابع سیستم و اجرای سرویس‌های سیستمی را بر دوش می‌کشد. پردازنده مرکزی، مدیریت چندین فرآیند را به صورت همزمان انجام می‌دهد.

سیستم عامل xv6 یک سیستم عامل پیشرفت داده شده به هدف آموزش می‌باشد که توسط گروهی از دانشجویان دانشگاه MIT مشابه معماری یونیکس ویرایش ششم، ساخته شده است. این سیستم عامل در اصل برای یک مینی-کامپیوتر به اسم PDP-11 ساخته شده بود؛ اما بر خلاف سیستم عامل unix 6، این سیستم عامل با x86 سازگار است. ساختار سیستم عامل xv6 از نوع یکپارچه (Monolithic) می‌باشد؛ یعنی حول هسته‌ای یکپارچه، بنا شده است.

¹ Multitasking

² Kernel

از جمله دلایلی که می‌شود برای این قضیه ارائه داد، کد داخل فایل x86.h می‌باشد که دستورات داخل این فایل مشابه دستورات پردازنده‌ی x86 است. همچنین، طبق اسناد xv6-rev11، فایل mmu.h شامل بخش مدیریت حافظه (Memory Management Unit) مربوط به x86 می‌باشد. همین‌طور فایل asm.h نیز شامل کد assembler macros برای ساخت segment های x86 می‌باشد.

- یک پردازنده در سیستم‌عامل xv6 از چه بخش‌هایی تشکیل شده است؟ این سیستم‌عامل به طور کلی، چگونه پردازنده را به پردازنده‌های مختلف، اختصاص می‌دهد؟

یک پردازنده در xv6 از دو بخش تشکیل شده است: بخشی که فقط برای هسته (Kernel) قابل رویت است که "وضعیت پردازنده" نام دارد. بخش دیگر، حافظه فضای کاربری (User Space Memory) است که شامل دستورات آن، استک و داده‌ها می‌شود. هسته سیستم عامل xv6 مانند هسته بقیه پردازنده‌ها به هر پردازنده، یک شناسه یکتا (Process Identifier) یا PID اختصاص می‌دهد و با یک فراخوانی سیستم، با دستور getpid() شناسه یکتای پردازنده کنونی را دریافت می‌کند.

- نمونه هایی از فراخوانی این دستور در فایل usertest.c آمده:

```
427 void
428 mem(void)
429 {
430     void *m1, *m2;
431     int pid, ppid;
432
433     printf(1, "mem test\n");
434     ppid = getpid();
435     if((pid = fork()) == 0){
436         m1 = 0;
437         while((m2 = malloc(10001)) != 0){
438             *(char**)m2 = m1;
439             m1 = m2;
440         }
441         while(m1){
442             m2 = *(char**)m1;
443             free(m1);
444             m1 = m2;
445         }
446         m1 = malloc(1024*20);
447         if(m1 == 0){
448             printf(1, "couldn't allocate mem?!\n");
449             kill(ppid);
450
451         if(c != a){
452             printf(stdout, "sbrk downsize failed, a %x c %x\n", a, c);
453             exit();
454         }
455
456         // can we read the kernel's memory?
457         for(a = (char*)(KERNBASE); a < (char*)(KERNBASE+
458             ppid = getpid();
459             pid = fork();
460             if(pid < 0){
461                 printf(stdout, "fork failed\n");
462                 exit();
463             }
464             if(pid == 0){
465                 printf(stdout, "oops could read %x = %x\n", a, *a);
466                 kill(ppid);
467                 exit();
468             }
469             wait();
470         }
471
472         // if we run the system out of memory, does it cl
473         // failed allocation?
```

اختصاص پردازنده توسط xv6 انجام می‌شود که روش Time Sharing به صورت Transparent می‌باشد. از این روش برای Multiprogramming استفاده می‌شود. این تقسیم زمانی به گونه ای انجام می‌شود که برای کاربر نامحسوس باشد و زمانی که پردازنده بعد از تمام شدن سهمیه زمانی دسترسی به CPU، از حالت اجرا توسط پردازنده خارج می‌شود و CPU از آن گرفته می‌شود تا پردازنده بعدی به حالت اجرا دربیاید، سیستم عامل xv6 رجیسترهای موجود در CPU که دیتای لازم برای اجرای پردازنده مذکور را حاوی هستند را ذخیره میکند تا این رجیسترها آماده ذخیره اطلاعات برای پردازنده بعدی شوند و وقتی دوباره این پردازنده به حالت اجرا درمی‌آید این مقادیر به این رجیسترها برگردانده شوند. در این فرایند برنامه‌ها به شکل همروند به اجرا در می‌آیند به گونه ای که کاربر این Time Sharing را احساس نمی‌کند.

• فراخوانی‌های سیستمی exec و fork چه عملی را انجام می‌دهند؟

فراخوانی سیستمی fork یک پردازنده جدید تولید می‌کند؛ یعنی با توجه به مفهوم تولید هر پردازنده توسط یک parent، با صدا شدن دستور fork، همه حافظه فضای کاربری (User Space Memory) مرتبط با parent را کپی می‌کند و در حافظه پردازنده جدید، وارد می‌کند.³ از جمله این اطلاعات، می‌توان به متن⁴، داده⁵، پشته⁶ و هرم⁷، اشاره کرد. اما این رابطه فرزند و والد باعث نمی‌شود که بعد از تغییر داده ذخیره شده در یک رجیستر یا متغیر در یک پردازنده، آن مقدار در دیگری هم تغییر یابد. علت آن هم این است که این اطلاعات در حافظه‌های جداگانه‌ای ذخیره می‌شوند.

پردازنده پدر، به عنوان parent پردازنده فرزند، بعد از ایجاد فرزند خود به caller تابع fork برمی‌گردد و در این نقطه، دو پردازنده به شکل همزمان انجام می‌شوند و یک پردازنده separate and duplicate ایجاد می‌کند.

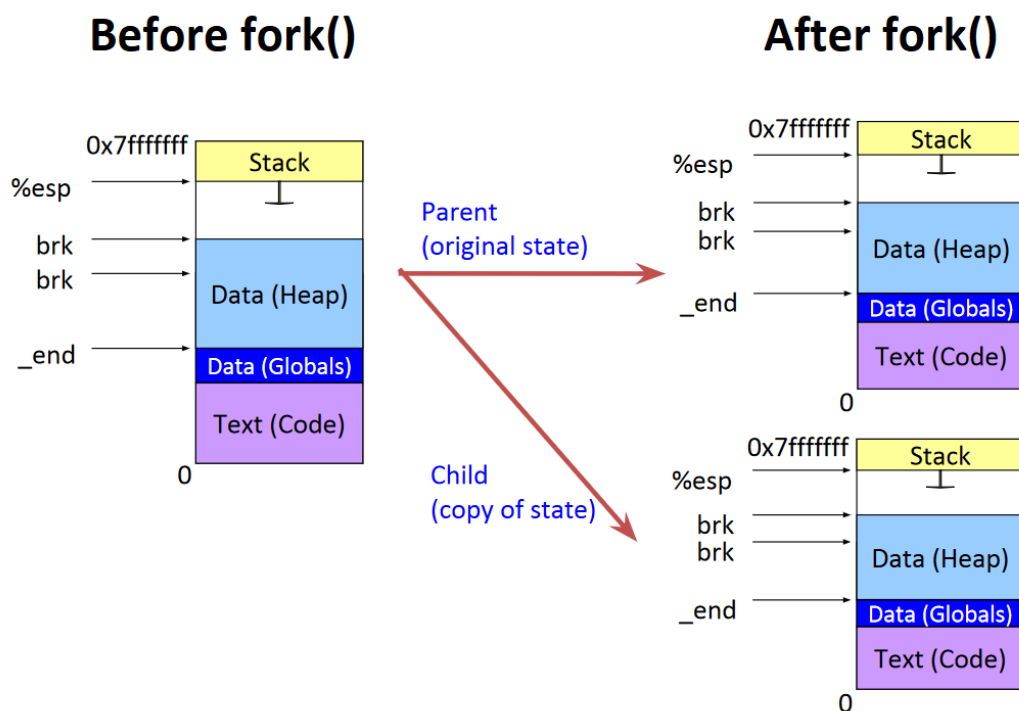
³ در واقع، بدین معنا است که این فراخوان سیستمی، همه دیتا و دستورات پردازنده فعلی را در حافظه پردازنده جدید، کپی می‌کند.

⁴ Text

⁵ Data

⁶ Stack

⁷ Heap



تابع `fork`، مقداری بازگشتی معادل با PID پردازۀ فرزند خروجی می‌دهد. پردازۀ فرزند نیز پس از ایجاد شدن، به caller تابع `fork` یا به عبارتی نقطه‌ی شروع خود برمی‌گردد. مقدار `return` شده تابع `fork` پس از ایجاد یک پردازۀ جدید برای والد و فرزند متفاوت است:

اگر خروجی 0 بود یعنی در پردازۀ فرزند هستیم، و اگر مقداری بزرگتر از 0 داشت یعنی شناسۀ پردازۀ فرزند را خروجی داده و در پردازۀ پدر هستیم. اگر هم خروجی منفی بود، یعنی در فرایند تولید پردازۀ فرزند به یک خطا برخوردیم و به درستی ایجاد نشده است.

- دو استفاده تابع fork به شرح زیر میباشد:

(الف) تولید یک برنامه موازی شامل چندین پردازش؛ مانند یک web server که بر روی هر HTTP request، یک پردازش جدید fork می‌کند.

(ب) انجام عمل launch برای یک برنامه جدید به کمک توابع نظیر exec؛ مانند پوسته⁸ لینوکس، پردازش ls را fork می‌کند.

حال اگر در کد، بعد fork از wait استفاده شده باشد، پردازش پدر صبر می‌کند تا پردازش فرزند پایان یابد و سپس، اجرایش از سر گرفته می‌شود. اگر پردازش فعلی فرزند نداشته باشد، خروجی مقدار 1- خواهد بود.

تابع exec حافظه پردازش خواننده را با یک تصویر حافظه جدید از یک فایل در فایل سیستم پر می‌کند. فرمت این فایل نشان‌دهنده سیاست نگه‌داری اطلاعات و دستورات در آن می‌باشد. این اطلاعات شامل نقطه شروع برنامه و داده‌های متناظر با برنامه می‌باشد. برنامه در نهایت به محل فراخوانی تابع exec بر نمی‌گردد بلکه یک برنامه موجود در یک فایل جایگزین برنامه فعلی پردازش می‌شود تا پردازش جدید با کدها و داده‌های موجود در فایل ادامه یابد.

به عبارتی در ادامه برنامه جدید، اجرا می‌شود؛ مگر اینکه میان اجرای این برنامه، خطایی رخ دهد که در این صورت، یعنی بعد از رخ دادن خطا، با exit اجرای پردازش خاتمه می‌یابد. لازم به ذکر است که پارامتر اول تابع exec اسم فایل برنامه و پارامتر دوم آرگومان‌های ورودی برنامه خواهد بود.

⁸ Shell

• از نظر طراحی، ادغام نکردن این دو، چه مزیتی دارد؟

در زمان تغییر مسیر⁹ ورودی/خروجی¹⁰، اگر کاربر در پوسته، برنامه‌ای را اجرا کند، در پشت صحنه و به دور از نظر کاربر، فرایند زیر رخ می‌دهد:

- دستور موجود در ترمینال خوانده می‌شود.
- با تابع fork، یک پردازش جدید تولید می‌شود.
- در پردازش فرزند، با تابع exec، برنامه وارد شده کاربر جایگزین پردازش فعلی (پردازش فرزند) می‌شود.
- پردازش پدر تا اتمام پردازش فرزند، wait می‌کند.
- با اتمام پردازش فرزند، به main باز می‌گردد و منتظر دستور جدید می‌ماند.

اگر کاربر برای دستوری از تغییر مسیر استفاده کرد، تغییرات در file descriptor ها بعد از fork و پیش از exec، در پردازش فرزند انجام می‌شود. اگر قرار بود این دو تابع، در یک تابع فرضی spawn() تلفیق شوند، رویه¹¹ بسیار پیچیده‌تری برای پوسته، نیاز بود تا قادر باشد ورودی‌ها و خروجی‌های استاندارد را به درستی، ارجاع دهد. همچنین، در این صورت، اگر کارکرد جدیدی به پیاده‌سازی افزوده می‌شد، این رویه، به تغییر نیاز می‌داشت. همینطور اگر این دو دستور جدا باشند، شل می‌تواند با فراخوانی توابع open و close و dup در فرزند تولید شده ورودی و خروجی استاندارد را تغییر دهد و در ادامه exec را اعمال کند. بنابر این برنامه‌ای که قرار است اجرا شود تغییر اضافه‌ای در این حالت نمی‌خواهد. و اگر ادغام صورت می‌گرفت باید هزینه‌ای مثل دستگاه جداگانه برای تغییر مسیر ورودی و خروجی بپردازیم یا این وظیفه را برعهده برنامه بگذاریم که معقول نیست.

⁹ Redirection

¹⁰ I/O

¹¹ Interface

اضافه کردن یک متن به Boot Message

برای اضافه کردن متن، باید فایل `init.c` را کمی تغییر دهیم. می‌دانیم که فایل `init`، شامل کدی است که هنگام `boot` شدن سیستم عامل `xv6` اجرا می‌شود. برای بازکردن یک فایل در لینوکس از یک `Text Editor` استفاده می‌کنیم. انواع مختلفی از `text editor` ها در لینوکس وجود دارد که یک نمونه‌ی پرکاربرد آن `nano` می‌باشد. با دستور زیر، فایل `init.c` را با `nano` باز می‌کنیم:

```
mobina@ubuntu:~/Downloads/xv6-public-master$ nano init.c
```

پس از فشردن دکمه `Enter`، فایل `init.c` با این `Text Editor` باز می‌شود. حال به کمک کلیدهای `pgUp` و `pgDn`، می‌توانیم در طول این صفحه حرکت کنیم و به داخل تابع `main` بیایم. در این تابع حلقه‌ای را می‌بینیم که بی‌نهایت بار اجرا می‌شود. آرگومان این حلقه `::` می‌باشد که به این معنی است که `initialize` نشده است و هیچ شرطی ندارد و هیچ متغیری در این حلقه افزایش نمی‌یابد. این همان حلقه‌ای است که بی‌نهایت بار اجرا می‌شود تا اینکه یک `break` در مسیر اجرای کد دیده شود و از حلقه به بیرون بپرد.

- این فایل را به شکل زیر تغییر می‌دهیم:

```
GNU nano 4.8                               init.c                               Modified
// init: The initial user-level program

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

char *argv[] = { "sh", 0 };

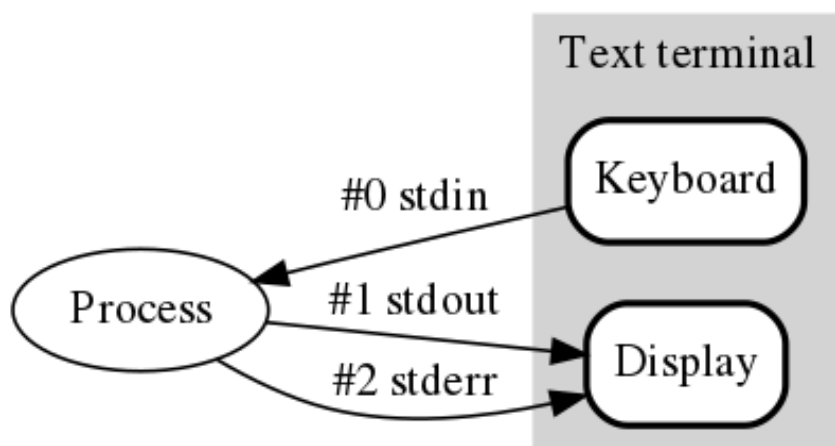
int
main(void)
{
    int pld, wpid;

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

    for(;;){
        printf(1, "init: starting sh\n");
        printf(1, "Group 10:\n");
        printf(1, "1. Matin Nabizadeh\n");
        printf(1, "2. Mobina Mehrazar\n");
        printf(1, "3. Mohammad Amin Yousefi\n");
        pid = fork();
        if(pid < 0){
            printf(1, "init: fork failed\n");
            exit();
        }
        if(pid == 0){
            exec("sh", argv);
            printf(1, "init: exec sh failed\n");
            exit();
        }
        while((wpid=wait()) >= 0 && wpid != pid)
            printf(1, "zombie!\n");
    }
}
```

- توجه شود که تغییرات اعمال شده، شامل اضافه کردن دستورهای زیر بوده است:

```
printf(1, "Group #10:\n");
printf(1, "1. Matin Nabizadeh\n");
printf(1, "2. Mobina Mehrazar\n");
printf(1, "3. Mohammad Amin Yousefi\n");
```



علت استفاده از مقدار 1 به عنوان آرگومان اول تابع printf این است که تعیین کنیم خروجی در کجا چاپ شود؛ بدین صورت که مقدار 1 متناظر با Standard Output می‌باشد که خروجی را به کنسول هدایت می‌کند.

- همچنین، با استفاده از دکمه‌های `^O` یا همان `Write Out` که عملکردی مشابه `save` کردن فایل را دارد، می‌توانیم صفحه‌ی زیر را مشاهده کنیم:

```
}
while((wpid=wait()) >= 0 && wpid != pid)
    printf(1, "zombie!\n");
}
}

File Name to Write: init.c
^G Get Help      M-D DOS Format      M-A Append          M-B Backup File
^C Cancel        M-M Mac Format      M-P Prepend         ^I To Files
```

همان طور که مشخص است، نام فایل به درستی در این قسمت، نوشته شده است. در ادامه، پس از فشردن کلید `Enter`، با `^X` از `nano text editor` خارج می‌شویم. حال به کمک دستور `cat`، محتوای این فایل را در صفحه‌ی ترمینال مشاهده می‌کنیم که به درستی وارد شده‌است.

```
mobina@ubuntu:~/Downloads/xv6$ cat init.c
// init: The initial user-level program

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

char *argv[] = { "sh", 0 };

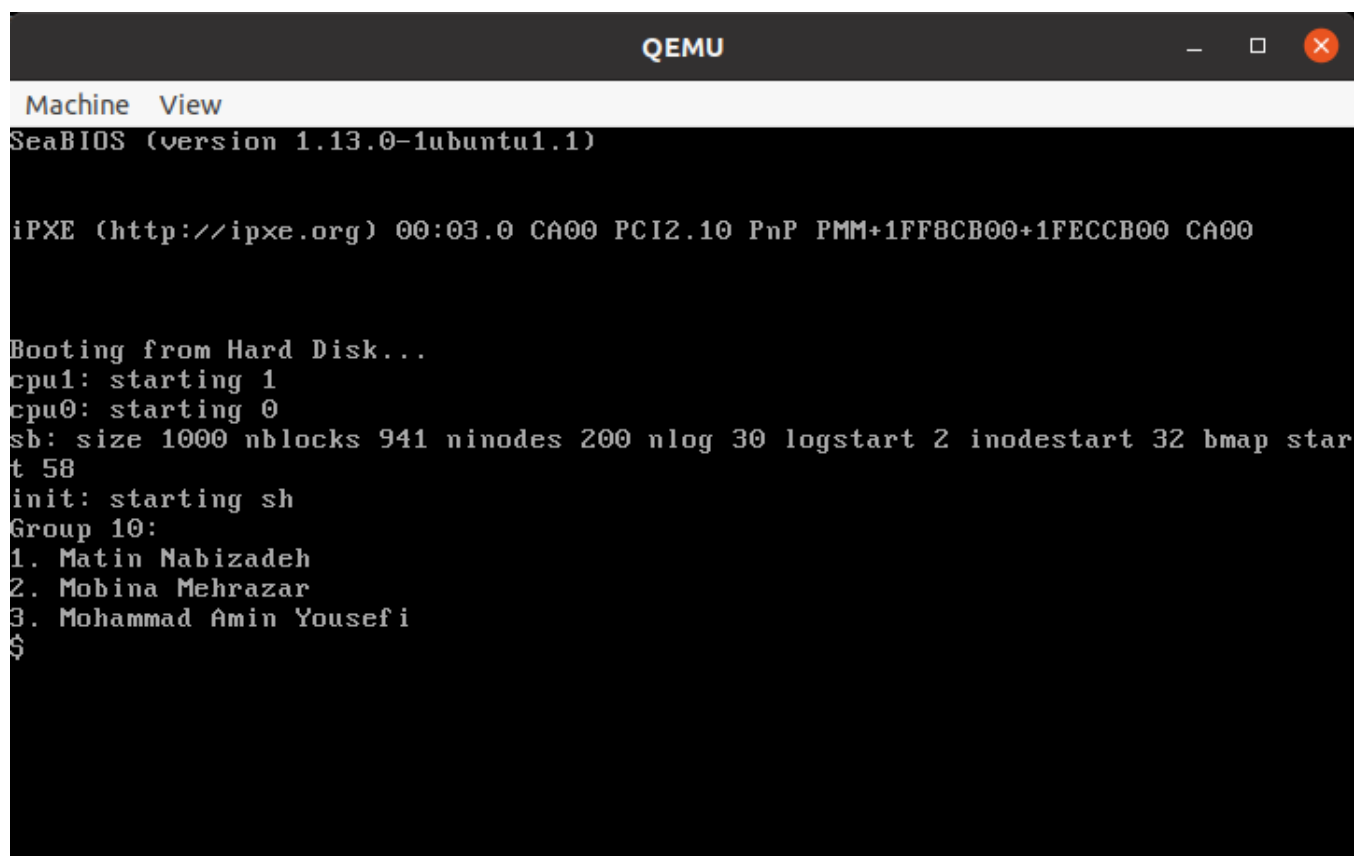
int
main(void)
{
    int pid, wpid;

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

    for(;;){
        printf(1, "init: starting sh\n");
        printf(1, "Group 10:\n");
        printf(1, "1. Matin Nabizadeh\n");
        printf(1, "2. Mobina Mehrazar\n");
        printf(1, "3. Mohammad Amin Yousefi\n");
        pid = fork();
        if(pid < 0){
            printf(1, "init: fork failed\n");
            exit();
        }
        if(pid == 0){
            exec("sh", argv);
            printf(1, "init: exec sh failed\n");
            exit();
        }
        while((wpid=wait()) >= 0 && wpid != pid)
            printf(1, "zombie!\n");
    }
}
```

همان‌طور که از نتایج برمی‌آید، تغییرات مد نظرمان، به درستی اعمال شده‌اند.

حال با دستور `make clean`، فایل‌های موجود در شکل زیر با تمامی پسوندها را پاک کرده و با دستور `make`، تمامی فایل‌ها را مجدداً کامپایل می‌کنیم. در ادامه، با دستور `make` سیستم عامل `xv6` را بوت می‌کنیم:



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

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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group 10:
1. Matin Nabizadeh
2. Mobina Mehrazar
3. Mohammad Amin Yousefi
$
```

اضافه کردن چند قابلیت به کنسول xv6

- در ابتدا راجع به عملکرد چندین تابع در فایل console.c توضیح می‌دهیم.

```
cgaputc
```

عملکرد کلی این تابع آپدیت کردن صفحه نمایش و رسیدگی به حرکت های Cursor و scrolling ها می‌باشد. آرگومان ورودی همان کاراکتری است که قرار است نمایان شود.

```
consoleintr
```

این تابع به interrupt های مرتبط با console input رسیدگی میکند.getc یک پوینتر به فانکشن می‌باشد که یک کاراکتر از input source مثل keyboard یا serial port دریافت میکند. console lock در ابتدا acquire میشود تا یک دسترسی انحصاری در طول process به آن پیدا کند. یک حلقه در این تابع است که به ازای هر کاراکتری که از input source دریافت میکند، رسیدگی کند. بعد از رسیدگی به کاراکتر های ورودی console.lock را یک مرتبه release میکند تا بقیه process ها هم به console lock دسترسی داشته باشند.

consoleread

این تابع، کاراکتر ها را از روی بافر ورودی کنسول می‌خواند و به کاراکتر هایی نظیر EOF و newline رسیدگی می‌کند. اگر بافر خالی بود هم صبر می‌کند¹².

consolewrite

این تابع، کاراکتر هارا از بافر به خروجی کنسول با فراخوانی تابع consputc به ازای هر کاراکتر انجام می‌دهد.

- در ادامه، نحوه پیاده‌سازی قابلیت‌های ذکر شده در صورت پروژه را شرح می‌دهیم. در دستور های Ctrl + F و Ctrl + B، از متغیر input.iterator استفاده می‌کنیم که نحوه عملکرد آن در بخش GDB به طور کامل توضیح داده شده است. نحوه عملکرد این دو دستور به این گونه است که در ابتدا امکان اجرای دستورات، بررسی می‌شود. اگر این امکان وجود داشت، بر حسب نیاز، بافرها را به عقب یا جلو شیفست می‌دهیم.

¹² با استفاده از تابع wait

- **Ctrl + B**

هنگامی که کاربر، کلیدهای Ctrl + B را می‌فشارد، یک کاراکتر کنترلی ویژه¹³ ایجاد می‌شود. این کاراکتر، در اصل، یک ثابت از پیش تعریف شده است که در هنگام فشردن توأمان با کلید Ctrl، با کد اَسکی حرف B، متناظر است. تابع consoleintr، فشردن شدن ترکیب کلیدها را ضمن مقایسه کاراکتر دریافت شده ('c') با مقدار C('B')، بررسی می‌کند. اگر این کلیدها فشرده شوند و ترکیب آن‌ها تشخیص داده شود، از طریق input.iterator، بررسی می‌شود که آیا اشاره‌گر در ابتدای بافر ورودی (input.w) هست، یا خیر. اگر نبود، تابع، موقعیت اشاره‌گر را یک واحد به عقب برمی‌گرداند. قطعه کد زیر، نحوه پیاده‌سازی این بررسی را نمایش می‌دهد:

```
case C('B'):  
    if (input.iterator != input.w)  
    {  
        input.iterator--;  
        consputc(C('B'));  
    }  
    break;
```

اگر دستور Ctrl + B وارد شود و سپس کاراکتر جدیدی وارد شود، ابتدا input.buff و crt را به سمت جلو شیفت می‌دهیم و در نهایت، در جای خالی ایجاد شده، کاراکتر جدید را می‌نویسیم. اگر BACKSPACE به جای کاراکتر جدید وارد شود نیز، به سمت عقب شیفت می‌دهیم.

¹³ در اینجا، C('B')

- Ctrl + F

هنگامی که کاربر، کلیدهای Ctrl + F را می‌فشارد، یک کاراکتر کنترلی ویژه¹⁴ ایجاد می‌شود. این کاراکتر، در اصل، یک ثابت از پیش تعریف شده است که در هنگام فشردن توأمان با کلید Ctrl، با کد اَسکی حرف F، متناظر است. تابع `consoleintr`، فشردن ترکیب کلیدها را ضمن مقایسه کاراکتر دریافت شده ('c') با مقدار C('F')، بررسی می‌کند. اگر این کلیدها فشرده شوند و ترکیب آن‌ها تشخیص داده شود، از طریق `input.iterator`، بررسی می‌شود که آیا اشاره‌گر در انتهای بافر ورودی (`input.e`) هست، یا خیر. اگر نبود، تابع، موقعیت اشاره‌گر را یک واحد به جلو می‌برد. قطعه کد زیر، نحوه پیاده‌سازی این بررسی را نمایش می‌دهد:

```
case C('F'):
    if (input.iterator != input.e)
    {
        input.iterator++;
        consputc(C('F'));
    }
    break;
```

اگر ابتدا دستور Ctrl + F و سپس کاراکتر جدیدی وارد شود، ابتدا `input.buff` و `crt` را به سمت جلو شیفت می‌دهیم و در نهایت در جای خالی ایجاد شده حرف جدید را می‌نویسیم. اگر BACKSPACE به جای کاراکتر جدید وارد شده بود، به سمت عقب شیفت می‌دهیم.

¹⁴ در اینجا، C('F')

- روند اجرای دو دستور ذکر شده:

ابتدا عبارت test را وارد می‌کنیم و سپس دوبار کلید Ctrl + B را می‌فشاریم:

```
SeaBIOS (version 1.13.0-1ubuntu1.1)

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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ test
```

- حال، حرف t را وارد می‌کنیم:

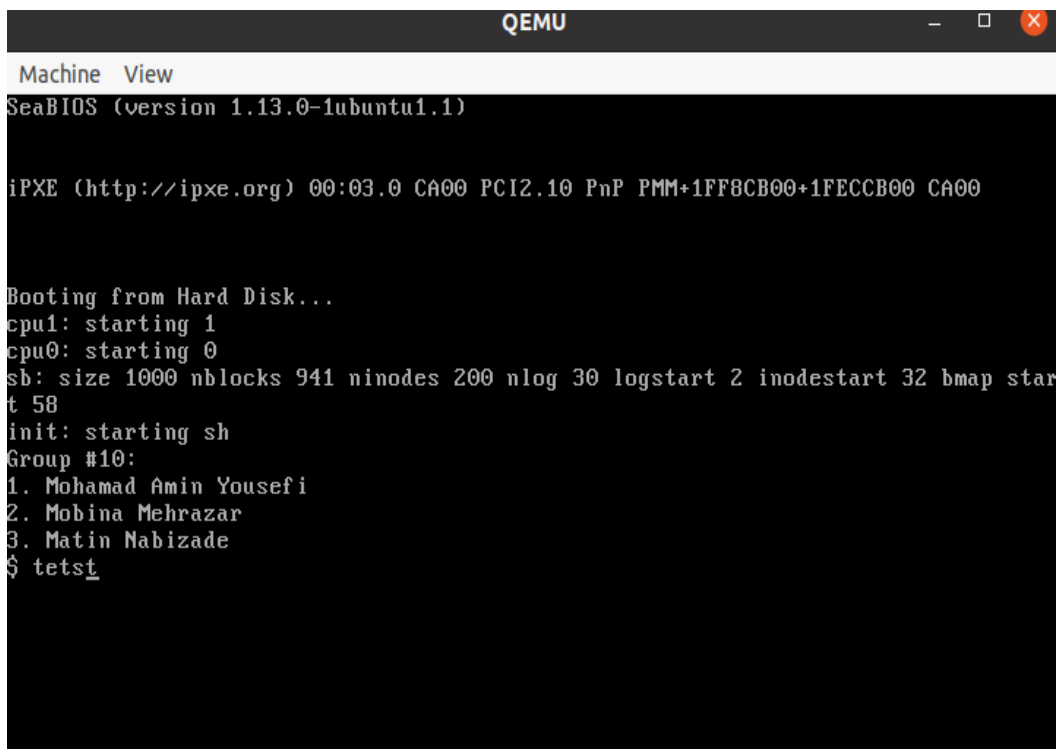
```
QEMU

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

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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ tetst
```


- و با فشردن Ctrl + F، می‌بینیم که:

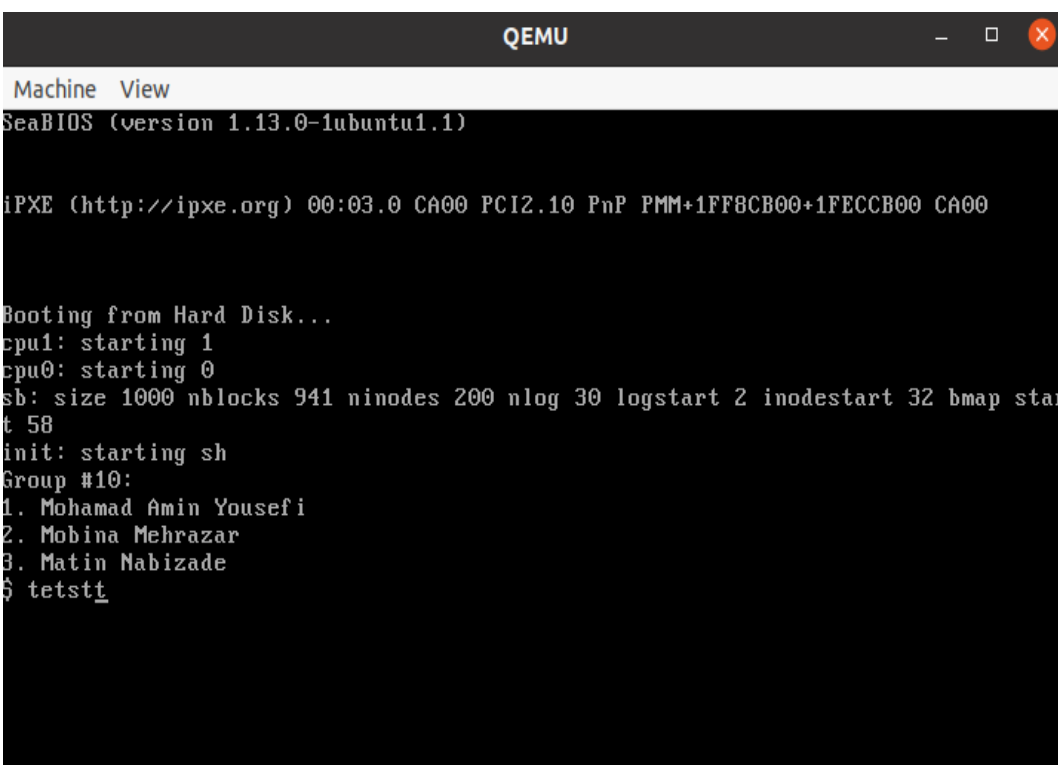


```
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ tetst
```

- و در نهایت، مجدداً حرف t را وارد می‌کنیم:



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

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

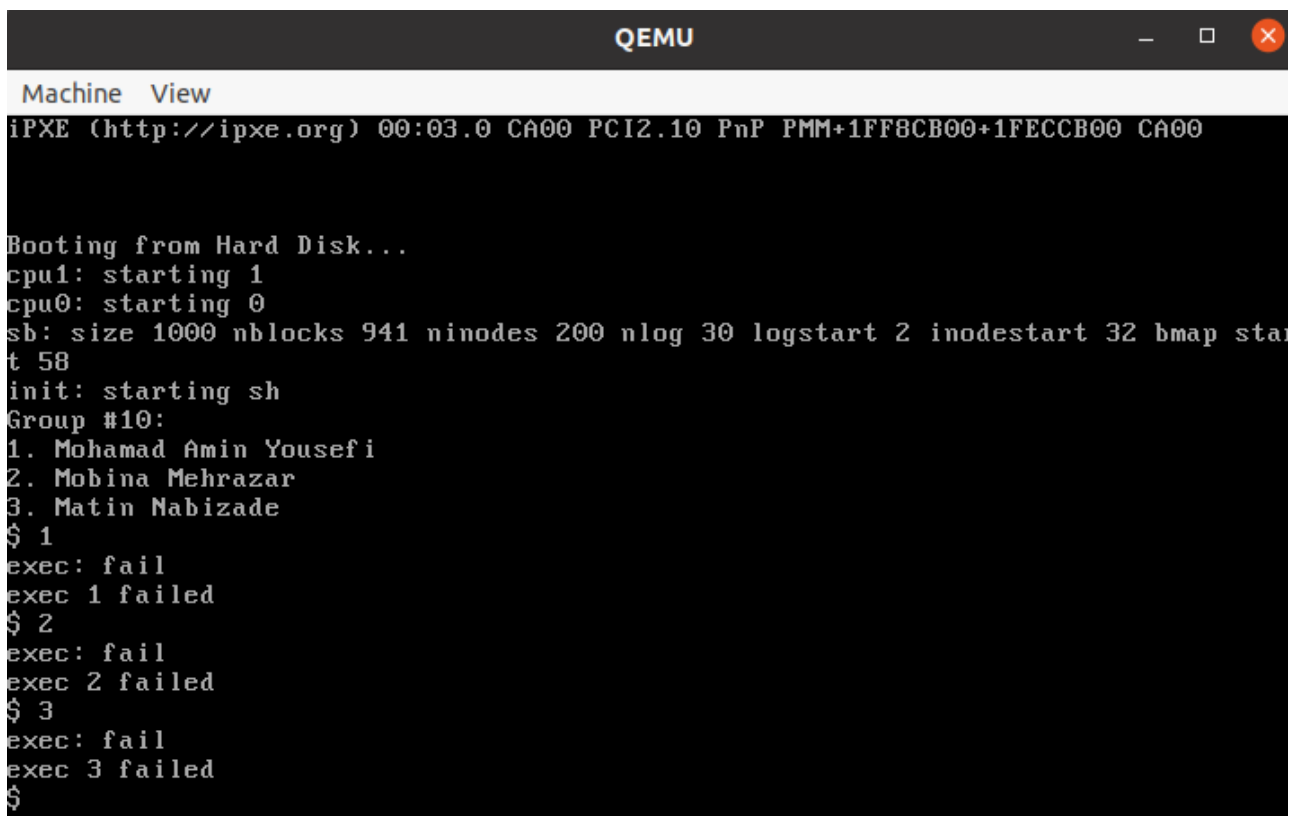
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ tetst
```

- Ctrl + L

ابتدا بافر ورودی را با دستور kill line پاک می‌کنیم. سپس، کنسول خروجی را با دستور consputc(clean) پاک کرده و در آن یک کاراکتر \$ می‌نویسیم و با اضافه کردن کاراکتر space به آن، ترمینال را آماده دریافت فرمان جدید می‌کنیم.

```
392     case C('L'):
393         kill_line();
394         consputc(CLEAN);
395         consputc('$');
396         consputc(' ');
397         break;
398
```

- ابتدا دستورات تست را وارد می‌کنیم:



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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
t 58
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ 1
exec: fail
exec 1 failed
$ 2
exec: fail
exec 2 failed
$ 3
exec: fail
exec 3 failed
$
```

- حال، `ctrl + L` را می‌فشاریم:



- **Arrow Up and Arrow Down**

برای `Arrow Up` و `Arrow Down`، یک داده‌ساختار جدید که یک صف حلقوی را برای ما به ارمغان می‌آورد، تشکیل می‌دهیم. این صف را `queue` می‌نامیم که از ۱۰ کلمه ۱۲۸ کاراکتری (سایز بافر ورودی)، تشکیل شده است. `queue.head`، نشان‌دهنده اولین دستور از 10 دستور اخیر است که در `queue` ذخیره شده‌اند. مقدار موجود در `queue.to_write`، اندیسی از `queue` که دستور جدید باید در آن نوشته شود را نشان می‌دهد. از `queue.index` نیز برای حرکت روی `queue` استفاده می‌شود که در حالت پیش‌فرض، برابر با `queue.to_write` می‌باشد. توجه شود که این متغیرها به صورت `uint` تعریف شده‌اند و مقدار ابتدایی آنها، صفر می‌باشد.

```

40 struct
41 {
42     char arr[QUEUE_SIZE][INPUT_BUF];
43     uint index;
44     uint size;
45     uint head;
46     uint to_write;
47 }queue;
48

```

- پس از هربار فشردن Enter یا Ctrl + D، دستور ورودی در queue توسط تابع زیر ذخیره می‌شود:

```

448     if (c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF)
449     {
450         save_in_queue();
451         input.w = input.e;
452         wakeup(&input.r);
453     }

```

```

304 void save_in_queue()
305 {
306     if(queue.size == QUEUE_SIZE)
307     {
308         for(int i = input.w; i <= (input.e - 1); i++)
309         {
310             queue.arr[queue.head][i - input.w] = input.buf[i % INPUT_BUF];
311         }
312         queue.head++;
313         queue.head %= QUEUE_SIZE;
314     }
315     else
316     {
317         for(int i = input.w; i <= (input.e - 1); i++)
318         {
319             queue.arr[queue.to_write][i - input.w] = input.buf[i % INPUT_BUF];
320         }
321         queue.size++;
322     }
323     queue.to_write++;
324     queue.to_write %= QUEUE_SIZE;
325     queue.index = queue.to_write;
326     up_arrow_check = 1;
327 }
328

```

در صورتی که مقادیر up_arrow_check و down_arrow_check برابر با 1 باشند، می‌توانیم عمل مربوطه را انجام دهیم.

همچنین، برای دستور ذکر شده (Arrow Up)، پس از ذخیره دستور، ابتدا داخل تابع consoleintr، خط فعلی را پاک می‌کنیم و سپس، بررسی می‌کنیم که اگر مقدار queue.index صفر بود، آن را برابر 10 قرار دهیم تا پس از کم شدن، برابر با 9 شود. توجه شود که در صورت اجرای این دستور، اجرای دستور Arrow Down هم ممکن می‌شود.

```
399     case UP_ARROW:
400         if (up_arrow_check && queue.size != 0)
401         {
402             kill_line();
403             if (queue.index == 0)
404                 queue.index = QUEUE_SIZE;
405             queue.index--;
406             write_from_queue();
407             down_arrow_check = 1;
408             if(queue.index == queue.head)
409                 up_arrow_check = 0;
410         }
411         break;
```

Arrow Down

```
291 void write_from_queue()
292 {
293     int i = input.w ;
294     while(queue.arr[queue.index][i - input.w] != '\n')
295     {
296         input.buf[i % INPUT_BUF] = queue.arr[queue.index][i - input.w];
297         consputc(queue.arr[queue.index][i - input.w]);
298         input.e++;
299         i++;
300     }
301     input.iterator = input.e;
302 }
303
```

```

413     case DOWN_ARROW:
414         if (down_arrow_check && ((queue.index + 1) % QUEUE_SIZE) != queue.to_write && queue.size != 0)
415         {
416             queue.index++;
417             queue.index %= QUEUE_SIZE;
418             up_arrow_check = 1;
419             kill_line();
420             write_from_queue();
421             if(queue.index + 1 == queue.to_write)
422                 down_arrow_check = 0;
423         }
424     else
425     {
426         queue.index = queue.to_write;
427         down_arrow_check = 0;
428         kill_line();
429     }
430     break;

```

حال، شیوهٔ اجرای این دستور را نمایش می‌دهیم.

- ابتدا ۳ دستور وارد می‌کنیم و سپس کلید Arrow Up را دو بار می‌فشاریم:

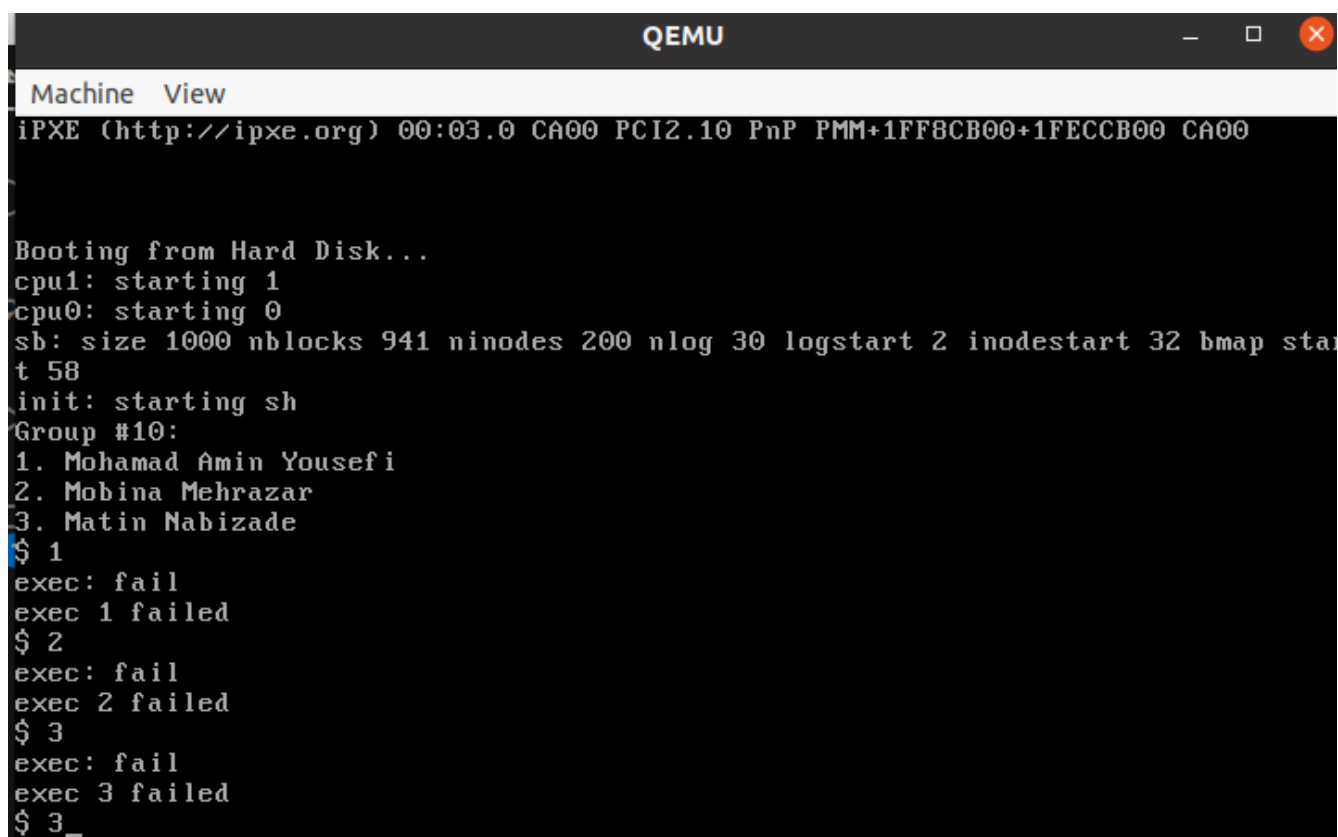
```

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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ 1
exec: fail
exec 1 failed
$ 2
exec: fail
exec 2 failed
$ 3
exec: fail
exec 3 failed
$ 1_

```

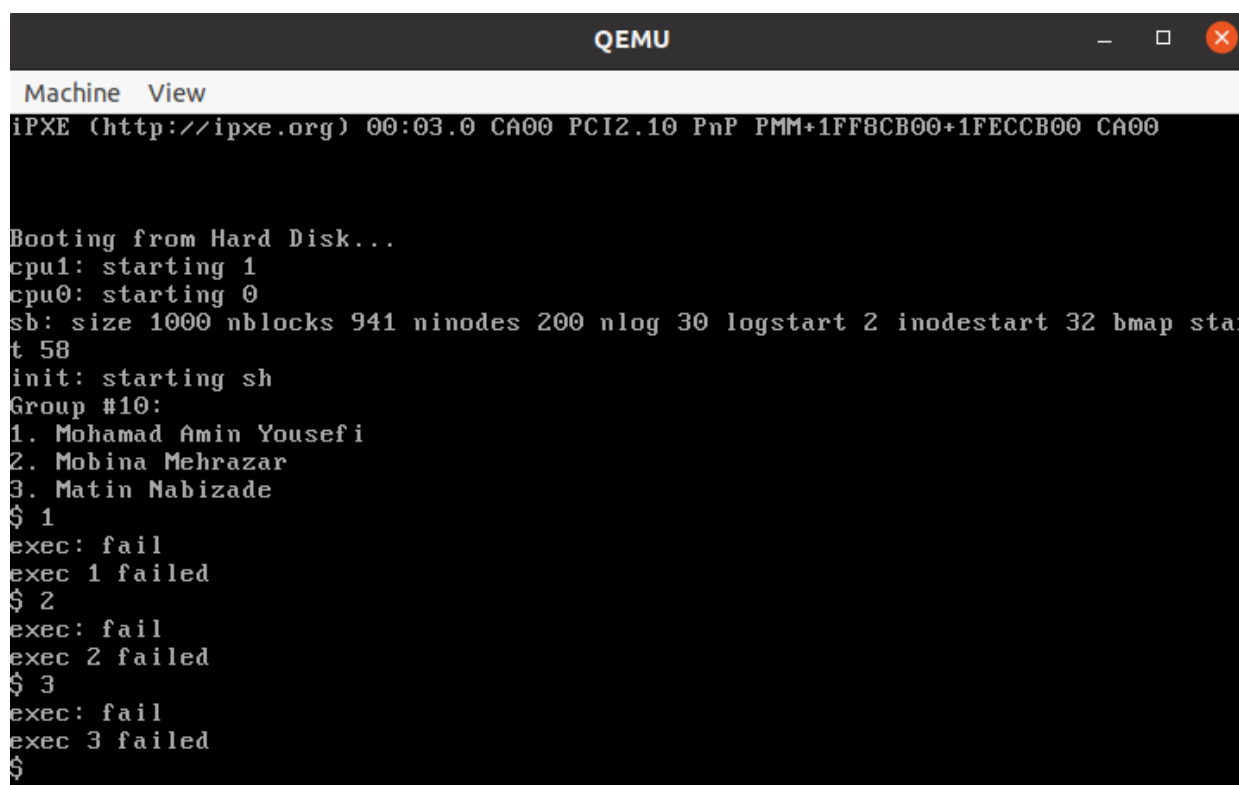
- در ادامه، ضمن دوبار فشردن Arrow Down:



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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
t 58
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ 1
exec: fail
exec 1 failed
$ 2
exec: fail
exec 2 failed
$ 3
exec: fail
exec 3 failed
$ 3_
```

- و در نهایت، محض پاک شدن کامل خط، یک بار دیگر فشردن Arrow Down:



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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
t 58
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ 1
exec: fail
exec 1 failed
$ 2
exec: fail
exec 2 failed
$ 3
exec: fail
exec 3 failed
$
```

اجرا و پیاده‌سازی یک برنامه در سطح کاربر

```
167
168 UPROGS=\
169   _cat\
170   _echo\
171   _forktest\
172   _grep\
173   _init\
174   _kill\
175   _ln\
176   _ls\
177   _mkdir\
178   _rm\
179   _sh\
180   _stressfs\
181   _usertests\
182   _wc\
183   _zombie\
184   _strdiff\
185
186 fs.img: mkfs README $(UPROGS)
187   ./mkfs fs.img README $(UPROGS)
188
```

برنامه strdiff.c نوشته شده و با دستور زیر در makefile به بقیه برنامه های سطح کاربر سیستم عامل اضافه شده است:

متغیر UPROGS، لیستی از برنامه های سطح کاربر را با یک زیرخط¹⁵ (یا همان _) در ابتدای آنها، ذخیره می‌کند.

با وارد کردن دستور زیر، برنامه برای دو کلمه زیر اجرا می‌شود:

```
strdiff apple banana
```

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

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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group 10:
1. Matin Nabizadeh
2. Mobina Mehrazar
3. Mohammad Amin Yousefi
$ strdiff apple banana
$ cat strdiff_result.txt
100011
$ _
```

- با وارد کردن دستور cat و اسم فایل مربوطه، محتوای فایل در صفحه ترمینال نوشته می‌شود:

¹⁵ Underscore

کامپایل سیستم عامل xv6

- در Makefile، متغیرهایی با نام‌های UPROGS و ULIB تعریف شده است. کاربرد آن‌ها، چیست؟

متغیر UPROGS¹⁶ برای لیست کردن برنامه های سطح کاربر و متغیر ULIB برای لیست کردن آجکت های کتابخانه های سطح کاربر¹⁷ استفاده می‌شود. برنامه‌های لیست شده که به آن‌ها اشاره شد، در فضای کاربر¹⁸ اجرا می‌شوند. در واقع هنگام ساخت و کامپایل xv6، این برنامه‌ها هم کامپایل می‌شوند و در نهایت، به فایل‌هایی که توسط سیستم‌عامل قابل اجرا هستند، تبدیل شده‌اند. از جمله این برنامه‌ها، می‌توان به cat یا echo اشاره کرد. نمونه دیگری از این برنامه‌ها، strdiff می‌باشد که در بخش های بالا، اجرا شدن این برنامه سطح کاربر را نشان داده‌ایم.

متغیر ULIB¹⁹، عبارت است از کتابخانه های سرح کاربر. این کتابخانه‌ها، توابع و روتین‌هایی به زبان C را شامل می‌شوند که برای اجرا شدن برنامه هایی در این سیستم‌عامل که در آن‌ها، از توابع این کتابخانه‌ها استفاده شده باشد، باید کامپایل شوند تا هنگام ساختن برنامه‌های وابسته به هم لینک شوند. این فایل ها در قوانین makefile به عنوان پیش‌نیاز برنامه‌های وابسته به آن‌ها قرار می‌گیرند و با دستور ld به فایل های اجرایی می‌پیوندند. این متغیر object file های مربوطه را ذخیره می‌کند. دو نمونه از آن، شامل ulib.o یا usys.o یا printf.o می‌باشد.

¹⁶ User Programs

¹⁷ User-Level Library Objects

¹⁸ User Space

¹⁹ User Library

از جمله توابع موجود در فایل‌های ULIB می‌توان به strcpy، malloc، printf و ... اشاره کرد. این اسامی با یک زیرخط در ابتدایشان آمده‌اند همگی یک target با پیش‌نیازهای object file هدف و متغیر ULIB دارند. این target های موجود در UPROGS باعث ساخت آجکت فایل‌های برنامه کاربر، اجرا شدن target های ULIB می‌شوند. بعد از این مرحله دستور ld اجرا می‌شود. این دستور فایل‌های مورد نیاز را پیوند می‌دهد تا یک فایل قابل اجرا تولید کند.

- لازم به ذکر است object file های مربوط به هر برنامه توسط یک قانون درونی makefile ساخته می‌شوند که به صورت صریح در makefile نوشته نشده‌اند.

مراحل بوت سیستم عامل xv6

(1) اجرای بوت لودر

- برنامه‌های کامپایل شده در قالب فایل‌های دودویی نگهداری می‌شوند. فایل مربوط به بوت نیز دودویی است. نوع این فایل دودویی چیست؟ تفاوت این نوع فایل دودویی با دیگر فایل‌های دودویی کد xv6 چیست؟ چرا از این نوع فایل دودویی استفاده شده است؟

- نوع این فایل بوت دودویی، ELF²⁰ است. فایل elf.h در میان کدهای xv6 وجود دارد. در این فایل (که به زبان C است)، دو استراکت elfhdr و proghdr را داریم. هر ELF شامل یک یا چندین بخش (Segment) مرتبط با runtime است. یک یا چند بخش sections که اطلاعات مختلف را در executable file و libraryها را حاوی است و در فرایند link و load نقش مهمی دارد. بخش Segment اشاره میکند که این ELF File چگونه در حافظه map شوند یا دیتای مربوط به آن را دارند. بخشی از Segment به کد قابل اجرا اشاره میکند. حال در یک dynamically linked binary سگمنت جداگانه‌ای داریم که مشخص میکند کدام کتابخانه به اشتراک گذاشته شده²¹ باید لود شود و این را با xv6 در میان می‌گذارد. دستور objdump یک برنامه است که اطلاعات مربوط به object file ها را نشان می‌دهد.

```
-h|--section-headers|--headers
```

²⁰ Executable and Linkable Format

²¹ Shared Library

- در ادامه این دستور، h- داریم که خلاصه ای از Section Header های مرتبط با object file مربوطه است. نمونه هایی از section های ELF در قسمت زیر، آمده است:

.txt	شامل دستورات قابل اجرای برنامه است.
.data	داده های مقداردهی شده را ذخیره می کند؛ مانند متغیرهای گلوبال.
.rodata	داده های از نوع readonly را شامل می شود؛ مانند string literal.
.bss	داده های بدون مقداردهی و به فرمت آدرس و اندازه آن داده ساختار ذخیره می شوند.

با اعمال دستور زیر برای فایل های باینری (یا همان *.o)²² از جمله bootblock.o، متوجه می شویم که این فایل باینری و بقیه فایل های باینری، از نوع ELF هستند:

```
objdump -h bootblock.o
```

- در خط اول خروجی، عبارت زیر را می بینیم:

```
bootblock.o      file format elf32-i386
```

```
mobina@ubuntu:~/Downloads/xv6$ objdump -h bootblock.o
bootblock.o:      file format elf32-i386

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
 0 .text          000001d3  00007c00  00007c00  00000074  2**2
 1 .eh_frame      000000b0  00007dd4  00007dd4  00000248  2**2
 2 .comment       0000002b  00000000  00000000  000002f8  2**0
 3 .debug_aranges 00000040  00000000  00000000  00000328  2**3
 4 .debug_info    000005d2  00000000  00000000  00000368  2**0
 5 .debug_abbrev  0000022c  00000000  00000000  0000093a  2**0
 6 .debug_line    0000029a  00000000  00000000  00000b66  2**0
 7 .debug_str     00000223  00000000  00000000  00000e00  2**0
 8 .debug_loc     000002bb  00000000  00000000  00001023  2**0
 9 .debug_ranges  00000078  00000000  00000000  000012de  2**0
```

که بدین معنا است که فرمت این فایل، از نوع ELF و برای معماری 32 بیتی x86 می باشد. در ادامه، section های مختلف ELF دیده می شود.

²² فایل های با پسوند .o توسط لینکر، استفاده می شوند.

این لیست، لیستی از headerهای بخش‌ها در object file مشخص شده می‌باشد. ضمن مقایسهٔ o.bootblock با بقیه Object File ها، می‌بینیم که بقیهٔ بخش‌های data و... را ندارد و فقط شامل بخش text، به عنوان بخش اصلی، می‌شوند.

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

این دستور کد ماشینی را از بخش txt. از object file ورودی یعنی آرگومان دوم را استخراج کرده و وارد یک فایل باینری به نام bootblock می‌کند. این فایل با دیگر فایل‌های باینری xv6 تفاوت دارد و در اصل، کد قابل اجرای خالص یا raw executable code بدون هیچ اطلاعات و ساختار اضافه‌ای است و مستقیماً توسط سخت افزار قابلیت اجرا دارد. بنابراین هیچ headerای ندارد و این باعث تفاوت با فرمت بقیه فایل‌های xv6 که فرمت ELF دارند می‌شود. بنابراین نوع فایل دودویی بوت از raw binary می‌باشد.

دلیل استفاده نکردن از ELF برای bootblock، این است که فرمت ELF را CPU نمیشناسد بلکه هستهٔ سیستم عامل می‌شناسد؛ وقتی هسته هنوز اجرا نشده است (اجرای بوت سکتور)، نمی‌توان فرمت ELF تحویل داد.

یک دلیل دیگر هم کم کردن حجم فایل است. با استخراج بخش text. فایل bootblock.o، حجم آن کاهش یافته و بالاجبار، در 510 بایت جای می‌گیرد و این به بارگذاری درست BIOS کمک می‌کند.

- این فایل را به زبان قابل فهم انسان (اسمبلی)، تبدیل نمایید.

- برای تبدیل bootblock به اسمبلی، از دستور زیر استفاده می‌کنیم:

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

از آنجا که bootblock، دودویی خام است و هیچ هدری برای مشخص کردن معماری‌اش ندارد، آنها را باید به صورت دستی به objdump بدهیم. در ادامه، پرچم‌هایی که استفاده شده‌اند را به اختصار، در جدول زیر، توضیح داده‌ایم:

-D	دودویی را ²³ disassemble می‌کنیم.
-b binary	نوع فایل را دودویی خام در نظر می‌گیریم.
-m i386	معماری اسمبلی فایل را مشخص می‌کنیم.
-M addr16, data16	آدرس‌ها و داده‌ها را 16-بیت در نظر می‌گیریم.

²³ به فرآیند تبدیل زبان ماشین به زبان اسمبلی، Disassemble می‌گویند.

- از آنجا که وقتی BIOS سکتور بوت را لود می‌کند در حالت واقعی هستیم، CPU در حالت 16 بیتی قرار دارد و اسمبلی 16-بیت نیز استفاده شده است، هنگام disassemble کردن هم می‌گوییم که آدرس‌ها و داده‌ها را 16-بیت در نظر بگیرد. با مشاهده خروجی این دستور، می‌بینیم که ابتدای آن، بسیار شبیه به bootasm.S است:

```
bootblock:      file format binary
```

```
Disassembly of section .data:
```

```
00007c00 <.data>:
```

```
7c00:      fa      cli
7c01:      31 c0     xor     %ax,%ax
7c03:      8e d8     mov     %ax,%ds
7c05:      8e c0     mov     %ax,%es
7c07:      8e d0     mov     %ax,%ss
7c09:      e4 64     in      $0x64,%al
7c0b:      a8 02     test    $0x2,%al
7c0d:      75 fa      jne     0x7c09
7c0f:      b0 d1     mov     $0xd1,%al
7c11:      e6 64     out     %al,$0x64
7c13:      e4 64     in      $0x64,%al
7c15:      a8 02     test    $0x2,%al
7c17:      75 fa      jne     0x7c13
7c19:      b0 df     mov     $0xdf,%al
7c1b:      e6 60     out     %al,$0x60
7c1d:      0f 01 16 78 7c lgdtw   0x7c78
7c22:      0f 20 c0     mov     %cr0,%eax
7c25:      66 83 c8 01     or      $0x1,%eax
7c29:      0f 22 c0     mov     %eax,%cr0
7c2c:      ea 31 7c 08 00 ljmp    $0x8,$0x7c31
7c31:      66 b8 10 00 8e d8 mov     $0xd8e0010,%eax
7c37:      8e c0     mov     %ax,%es
7c39:      8e d0     mov     %ax,%ss
7c3b:      66 b8 00 00 8e e0 mov     $0xe08e0000,%eax
7c41:      8e e8     mov     %ax,%gs
7c43:      bc 00 7c     mov     $0x7c00,%sp
7c46:      00 00     add     %al,(%bx,%si)
7c48:      e8 f0 00     call    0x7d3b
```

- علت استفاده از دستور objcopy حین اجرای عملیات make، چیست؟

از این دستور برای کپی کردن سکشن‌های به‌خصوصی مثل .txt از آبجکت فایل‌ها و تبدیل آنها به فرمت باینری استفاده شده است. هدف از انجام این کار، ساختن بخش‌های سیستم عامل xv6 برای disk image می‌باشد.

حال، به بررسی مواردی که در makefile از این دستور استفاده شده است، می‌پردازیم:

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

- از این دستور، برای کپی کردن سکشن .txt از آبجکت فایل bootblock.o و خروجی دادن آن به عنوان یک فایل باینری به اسم bootblock استفاده می‌شود.

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

- از این دستور برای کپی کردن سکشن .txt از آبجکت فایل bootblockother.o و خروجی دادن آن به عنوان یک فایل باینری به اسم entryother استفاده می‌شود.

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

- با استفاده از این دستور، محتوای آبجکت فایل input.out به داخل فایل باینری initcode کپی می‌شود.

همان‌طور که مشاهده می‌شود، فرمت فایل ورودی و خروجی یکسان نیست. دستور objcopy از کتابخانهٔ BFD استفاده می‌کند و تمامی فرمت‌های موجود در خود را پشتیبانی می‌کند.

همان‌طور که در دستورهای اشاره شده می‌بینیم، در این دستور آپشن‌هایی داریم که هر یک در ادامه، شرح داده می‌شوند:

-S

- این آپشن، اطلاعات مربوط به debug و symbol را در عملیات کپی‌برداری، حذف می‌کند. یا به عبارتی حذف هدرهای ELF.

- 0

- با استفاده از این آپشن، فرمت فایل مقصد را تعیین می‌کنیم. همان‌طور که واضح است، این آپشن در هر سه قطعه کد، به binary تنظیم شده است. سیستم عامل، تمایزی بین فایل دودویی خام²⁴ و فایل متنی²⁵، قائل نمی‌شود و همگی به شکل جریانی از بایت‌ها در نظر گرفته می‌شوند.

-j

- از این آپشن برای خارج کردن section به‌خصوصی از آبجکت فایل ورودی استفاده می‌شود. در ادامه، راجع به .txt section مربوط به آبجکت فایل توضیح داده شده است. این section عموماً کد قابل اجرای برنامه را شامل می‌شود. اتمام ساخت فایل کرنل با لینک کردن همه آبجکت فایل‌های ذخیره شده در متغیر OBJS و entry.o و فایل‌های باینری initcode، bootblock و entry.other انجام می‌شود.

²⁴ Raw Binary File

²⁵ Text File

در واقع برای اطمینان از اینکه مؤلفه‌های ضروری مانند bootloader و هسته در یک فرمتی باشند که مستقیماً توسط سخت‌افزار کامپیوتر قابل اجرا باشند، استفاده میشود. هدف اجرا و بارگذاری درست بوتلودر و هسته است.

• 14. یک ثبات عام منظوره، 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-منظوره²⁶: x86 مدرن 8 ثبات 32 بیتی از این نوع دارد²⁷ و می‌توان به EAX و EBX و ECX و EDX و ESI و EDI اشاره کرد. این ثبات‌ها برای ذخیره متغیرها در طول محاسبات استفاده می‌شوند. انواع دیگر آن مثل EBP و ESP اشاره‌گرهایی به کف و انتهای پشته، ذخیره می‌کنند. این آدرس‌های ذخیره شده، مجازی²⁸ هستند. ثبات EIP همان شمارنده برنامه²⁹ است و آدرس مجازی دستور بعدی CPU را ذخیره می‌کند. برای مثال، در توضیح ثبات عام-منظوره EAX، می‌توان گفت که بیشتر به عنوان محل ذخیره نتیجه محاسبات ریاضی و منطقی مورد استفاده واقع می‌شود. همچنین، این ثبات، برای مقادیر بازگشتی از فراخوانی توابع، اولویت اول را دارد. لازم به ذکر است که 16 بیت کم‌ارزش‌تر این ثبات، می‌توانند به طور مستقل، مورد دسترسی واقع شوند.

ثبات قطعه³⁰: از جمله این رده از ثبات‌ها، می‌توان به CS و DS و ES و FS و GS و SS اشاره کرد. اشاره‌گرهایی به سگمنت‌های مختلف حافظه پردازش (مثل code segment یا data segment) stack ذخیره می‌کند. این ثبات‌ها در تغییر بین حالت کاربر و حالت هسته، عوض می‌شوند. برای مثال، در توضیح ثبات قطعه CS³¹، می‌توان گفت که این ثبات، انتخاب‌کننده سگمنت را برای سگمنت کد، نگهداری می‌کند و در واقع،

²⁶ General-Purpose Register

²⁷ لازم به ذکر است که رجیسترها، تنها به نوع 32 بیتی محدود نمی‌شوند.

²⁸ Virtual

²⁹ Program Counter

³⁰ Segment Register

³¹ Code-Segment Register

به محلی از حافظه اشاره می‌کند که کد قابل اجرای برنامه، ذخیره شده است. هنگامی که یک برنامه در حال اجرا است نیز، این ثبات به سگمنت گدی که CPU از آن دستوراتش را می‌خواند، اشاره می‌کند. در مجموع، چنین می‌توان گفت که وجود این ثبات، برای اجرای دستورات برنامه‌ها و همچنین، حفظ جریان آن‌ها، حیاتی است.

ثبات **کنترلی**³²: از انواع آن، می‌توان به **CR0** اشاره کرد که این نوع ثبات، یک ثبات 32-بیت است که حالات و شرایط اجرایی متنوعی از پردازنده را مدیریت می‌کند. این ثبات همچنین حالت عملیاتی (حالت واقعی³³، حالت محافظت‌شده³⁴ یا حالت 8086 مجازی³⁵) را مدیریت کرده عملکرد اساسی و مهمی در فعال یا غیرفعال کردن خصوصیات پردازنده، دارد. نوعی دیگر از ثبات‌های کنترلی، **CR3** است که آدرس page table مربوط به پردازنده در حال اجرا را ذخیره می‌کند.

ثبات **وضعیت**: اطلاعات متعددی را در CPU ذخیره می‌کند؛ برای مثال، ثبات **FLAGS** حالت فعلی پردازنده را مشخص می‌کند. این ثبات در انواع پردازنده‌ها، نام‌های متفاوتی دارد.

- در پردازنده 16-بیتی، آن را **FLAGS** گویند.
- در پردازنده 32-بیتی، **EFLAGS** نام دارد.
- در پردازنده 64-بیتی، به آن، **RFLAGS** می‌گویند.

هر بیت این رجیسترها یک flag از یک وضعیت می‌باشد. و این flag میتواند حالت درست یا غلط باشد. وضعیت اعمال منطقی و محاسباتی و محدودیت های عملیات فعلی پردازنده در این flag ها ذخیره می‌شوند. برای مثال، می‌توان به ثبات‌های وضعیت صفر، نقلی، سرریز، علامت، جهت و اخلال، اشاره کرد. ثبات پرچم صفر، تعیین می‌کند که آیا نتیجه یک محاسبه به خصوص، برابر با صفر شده است یا خیر. ثبات پرچم باقی‌مانده، مشخص می‌کند که آیا رقم نقلی به دست‌آمده از پرارزش‌ترین بیت موجود در یک عملیات

³² Control Register

³³ Real Mode

³⁴ Protected Mode

³⁵ Virtual 8086 Mode

محاسباتی، یک هست یا خیر. به همین صورت، اگر مقدار ثبات پرچم سرریز، برابر با 1 باشد، نشان از به وقوع پیوستن یک سرریز محاسباتی است. همچنین، ثبات پرچم علامت، مشخص می‌کند که نتیجه یک عملیات حسابی، مثبت است یا منفی. اگر مقدار آن برابر با 1 باشد، بدان معنا است که این نتیجه، منفی است. در غیر این صورت، نتیجه محاسبه مذکور، مثبت است.

• کد معادل entry.S در هسته لینوکس را بیابید.

- کد بخش مشترک:

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

- کد مختص 32 بیتی:

https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry_32.S

- کد مختص 64 بیتی:

https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry_64.S

• همانطور که اشاره شد، کد معادل، یک بخش مشترک و یک بخش اختصاصی دارد.

2) اجرای هسته xv6

• 19 چرا این آدرس، فیزیکی است؟

جدولی که برای نگاشت آدرس مجازی مورد استفاده به آدرسی فیزیکی استفاده می‌شود، خود باید در حافظه فیزیکی قرار گیرد چرا که اگر آدرس این جدول به صورت مجازی ذخیره شده باشد، باید ابتدا آدرس فیزیکی آن را پیدا کنیم و این آدرس از خودش به دست می‌آید(!) و در یک حلقه بی‌نهایت می‌افتیم که در این صورت، عملاً هرگز این جدول حاصل نمی‌شود. حال اگر جدولی تعریف کنیم که این مشکل را حل کند و آدرس فیزیکی جدول اولیه را بدهد، بعد از اجرای این دستور باید به این حلقه پایان دهیم و پایان دادن به اجرای دستور هم نیازمند آدرس فیزیکی است و این، خود باعث تناقض می‌شود. بنابراین راه حل این است که آدرس این جدول نگاشت در ابتدا به صورت فیزیکی ذخیره شود و یک آدرس فیزیکی مخصوص داشته باشد. این جدایی و امنیت ایجاد شده در سیستم، که محدودیت دسترسی مستقیم به حافظه فیزیکی پردازنده‌های دیگر است، از مزیت‌ها می‌باشد.

• چرا برای کد و داده‌های سطح کاربر، پرچم SEG_USER تنظیم شده است؟

برای حفاظت از هسته، علاوه بر صفحه‌بندی، قطعه‌بندی نیز انجام می‌شود.

کد زیر در فایل vm.c (تابع seginit)، شیوهٔ قطعه‌بندی در xv6 را نشان می‌دهد:

```
13 // Set up CPU's kernel segment descriptors.
14 // Run once on entry on each CPU.
15 void
16 seginit(void)
17 {
18     struct cpu *c;
19
20     // Map "logical" addresses to virtual addresses using identity map.
21     // Cannot share a CODE descriptor for both kernel and user
22     // because it would have to have DPL_USR, but the CPU forbids
23     // an interrupt from CPL=0 to DPL=3.
24     c = &cpus[cuid()];
25     c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
26     c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
27     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
28     c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
29     lgdt(c->gdt, sizeof(c->gdt));
30 }
31
```

- همچنین برای تعریف SEG در فایل mmu.h، داریم:

```
42 // Normal segment
43 #define SEG(type, base, lim, dpl) (struct segdesc) \
44 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
45   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
46   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
47 #define SEG16(type, base, lim, dpl) (struct segdesc) \
48 { (lim) & 0xffff, (uint)(base) & 0xffff, \
49   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
50   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
51 #endif
```

از آنجایی که ترجمه آدرس‌ها برای کد و داده‌های سطح کاربر از ترجمه آدرس‌های هسته متفاوت است، این پرچم برای ترجمه در سطح محدود روی آدرس‌های کد و داده‌های سطح کاربر اعمال می‌شود. بدین شکل دسترسی غیرمجاز به سطح کاربری حساس شکل نمی‌گیرد.

3) اجرای نخستین برنامه سطح کاربر

- جهت نگهداری اطلاعات مدیریتی برنامه‌های سطح کاربر، ساختاری تحت عنوان **struct proc** (در خط 2336) ارائه شده است. اجزای آن را توضیح دهید.

این ساختار، اطلاعات مدیریتی برنامه‌های سطح کاربر (یا همان وضعیت هر پردازش) را در خود ذخیره می‌کند که هرکدام را در بخش زیر، به تفصیل، توضیح می‌دهیم.

- این ساختار، 13 متغیر به شرح زیر دارد:

متغیر	توضیحات
unit sz	حجم حافظهٔ مربوط به پردازش را بر حسب بایت، ذخیره می‌کند.
pde ³⁶ _t* pgdir	اشاره‌گری به page table مربوط به پردازش است.
char *kstack	پوینتری به کرنل استک ذخیره می‌کند.
enum procstate state	وضعیت پردازش را ذخیره می‌کند ³⁷ .
int pid	شناسهٔ یکتای مربوط به پردازش است. این عدد، برای هر پردازش، یکتا می‌باشد.
struct proc *parent	پوینتری به ساختار proc مربوط به والد پردازش کنونی است. این والد به کمک تابع fork پردازش کنونی را می‌سازد ³⁸ .
struct trapframe *tf	پوینتری به trap frame مربوط به فراخوانی سیستمی کنونی است. این پوینتر، وضعیت اجرای برنامه در هنگام اجرای فراخوانی سیستمی را ذخیره می‌کند.
struct context *context	پوینتری به struct context که برای context switching به کار می‌رود. مقادیر رجیسترها را در این فرایند نگهداری می‌کند. switch کردن بین پردازش‌ها با تابع اسمبلی انجام می‌شود.

³⁶ Page Directory Entry

³⁷ انواع این وضعیت‌ها به شکل UNUSED و EMBRYO و SLEEPING و RUNNABLE و RUNNING و ZOMBIE می‌باشد.

³⁸ در سوال یک گزارش، به طور مفصل توضیح داده شده است.

<code>void *chan</code> ³⁹	اگر مقدارش صفر نبود، بدین معنا است که پردازش در چنل به خصوصی در حالت sleeping قرار دارد؛ یعنی برای کاری wait می‌کند.
<code>int killed</code>	مقدار غیر صفر آن، بدین معنا است که پردازش، kill شده است.
<code>struct file *ofile[NOFILE]</code>	یک آرایه از پوینترهایی به فایل‌های باز شده توسط پردازش است.
<code>struct inode *cwd</code>	مقدار working directory کنونی پردازش را مشخص می‌کند.
<code>char name[16]</code>	نام استفاده شده پردازش در روند اشکال‌زدایی را ذخیره می‌کند.

- در خط 2340، متغیری مشاهده می‌شود که به منظور اشاره به پشته هسته، تعریف شده است. از پیش، می‌دانیم که وضعیت ثبات، حین فراخوانی توابع، بر پشته کاربر، ذخیره می‌شود تا در آینده، بتوان آن را بازیابی کرد یا از سر گرفت. متناظراً، هنگامی که زمان اجرای کد هسته فرا می‌رسد، چارچوب CPU بر پشته هسته، ذخیره می‌شود. یعنی در اصل، به ازای هر پردازش، پشته‌ای مجزا (به غیر از پشته مختص به کاربر)، در قسمتی مجزا از فضای حافظه، در نظر گرفته می‌شود.⁴⁰ دلیل این موضوع هم آن است که سیستم عامل، به پشته کاربر، اعتماد نداشته و با این کار، خود را از اتکای بر پشته کاربر، آزاد می‌سازد.

char *kstack

- همچنین، در خط 2348، تعریف آرایه‌ای را می‌بینیم که در اصل، آرایه‌ای از اشاره‌گرها به فایل‌های باز است که هر عنصر این آرایه، حاوی اطلاعاتی در مورد فایل مذکور می‌باشد. هنگامی که کاربر فایلی را باز می‌کند، یک مدخل جدید به این آرایه، افزوده می‌شود و اندیس مدخل گفته شده، به عنوان یک توصیف‌گر فایل⁴¹

³⁹ همان Channel است؛ برای مثال، کانال خط ورودی کنسول.

⁴⁰ این قسمت از حافظه، توسط کد معمولی کاربر، قابل دسترسی نیست.

به کاربر، پاس داده می‌شود. لازم به ذکر است که سه خانه اول این آرایه، به طور پیش‌فرض، اشغال می‌شوند که به ترتیب، از 0 تا 2، مربوط به ورودی استاندارد، خروجی و خطا هستند. واضح است که فایل‌هایی که بعداً باز می‌شوند، خانه‌های پس از اندیس 2 را به خود، اختصاص می‌دهند.

```
struct file *ofile[NOFILE]
```

- می‌دانیم که هر دستور یا داده موجود در تصویر حافظه یک پردازنده (اعم از کد / داده، پشته، هرم و ...)، آدرسی مختص به خود، دارد. همچنین، از پیش، می‌دانستیم که آدرس‌های مجازی از 0 شروع می‌شوند اما آدرس‌های فیزیکی، ممکن است متفاوت باشند. در خط 2339، متغیر دیگری مشاهده می‌شود که حاوی یک نگاشت میان آدرس‌های فیزیکی و مجازی مذکور است.

```
pde_t* pgdir
```

• همچنین، ساختار معادل آن در سیستم عامل لینوکس را بیابید.

- در نهایت، بایستی گفت که ساختار معادل آن در سیستم عامل لینوکس، ساختار task_struct (خط 743) در فایل linux/sched.h است.

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

- کدام بخش از آماده سازی سیستم بین تمامی هسته های پردازنده مشترک و کدام بخش، اختصاصی است؟

فرایند بوت سیستم عامل توسط هسته اول انجام می شود. کد نوشته شده به زبان اسمبلی در فایل entity.S به سمت تابع main در فایل main.c هدایت می کند. آماده سازی سیستم توسط توابع نوشته شده در این تابع انجام می شود. بقیه ی هسته ها توسط دستورات موجود در کد entryother.S وارد تابع mpenter می شود. چهار تابع بین همه هسته ها مشترک اند و در این تابع جای می گیرند و بین همگی هسته ها مشترک اند.

```
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
}
```

- لازم به ذکر است که تابع اول (یعنی switchkvm) مستقیماً با هسته اول در ارتباط نیست. این تابع، در mpenter صدا زده می شود و مستقیماً در main وجود ندارد.

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

- همچنین، تابع kvmalloc به شرح زیر است:

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
    kpgdir = setupkvm();
    switchkvm();
}
```

دستور اول تابع، یک page table در این هسته ایجاد می‌کند:

```
// Set up kernel part of a page table
pde_t*
setupkvm(void)
// ... The rest of the code
```

این page table توسط هسته اول شکل می‌گیرد و در روند اجرای برنامه به آن مراجعه می‌کند. از آنجایی که آماده‌سازی سیستم عامل xv6 توسط همگی هسته‌ها تقسیم می‌شود، بخش‌هایی که در ادامه می‌آیند، مشترک می‌باشند:

switchkvm / seginit / lapicinit/ mpmain

بخش‌های اختصاصی هسته‌ی اول به شرح زیر می‌باشد:

kinit1 / kvmalloc(setupkvm) / mpinit / picinit / ioapicinit / kinit2 / userinit
consoleinit uartinit / pinit / tvinit / binit / fileinit / ideinit / startothers

علت اینکه برخی موارد فقط توسط هسته اول اجرا می‌شود، ماهیت آن دستور می‌باشد؛ مثلاً startothers ، دستور به کار آوردن بقیه هسته‌ها می‌باشد.

شناسایی دیسک توسط پردازنده اول با تابع ideinit در فایل memide.c فقط یکبار و توسط هسته اول انجام می‌شود. از جمله توابع مشترک بین همه هسته‌ها switchkvm می‌باشد.

```
// Switch h/w page table register to the kernel-only page table,  
// for when no process is running.  
void  
switchkvm(void)  
{  
    lcr3(V2P(kpgdir));    // switch to the kernel page table  
}
```

که وظیفه ذخیره‌کردن page table ایجاد شده توسط هسته اول را در رجیسترهای مربوط به هر هسته را دارد که توسط همگی هسته‌ها باید انجام شود.

```
// 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 کد setup همگی هسته‌ها است.

زمان‌بندی هر پردازنده یکتا است؛ پس باید برای همگی آنها کد زمان‌بند توسط تابع scheduler اجرا شود و خود این تابع در تابعی دیگر به نام mpmain صدا زده می‌شود.

```
// 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
}
```

و این تابع بین همگی هسته‌ها مشترک می‌باشد.

اشکال زدایی

همانند توضیحات داده شده عمل می‌کنیم و دو ترمینال در آدرس فایل‌ها باز می‌کنیم. سپس در یکی از ترمینال‌ها، دستور زیر:

```
make qemu-gdb
```

و در دیگری، دستور:

```
gdb kernel
```

را اجرا می‌کنیم و در نهایت، دستور:

```
target remote tcp::26000
```

را وارد می‌کنیم. حال، ترمینال‌های ما، آماده اشکال زدایی با استفاده از GDB هستند. در ابتدا GDB روی حالت kernel قرار دارد. برای اشکال زدایی یک برنامه در سطح کاربر (مانند cat)، از دستور:

```
file _cat
```

یا برای strdiff، از:

```
file _strdiff
```

استفاده می‌کنیم.

روند اجرای GDB

(1) برای مشاهده Breakpoint ها، از چه دستوری استفاده می‌شود؟

برای مشاهده Breakpoint ها از دستور `info breakpoints` استفاده می‌شود.

```
(gdb) file _cat
Load new symbol table from "_cat"? (y or n) y
Reading symbols from _cat...
(gdb) break cat.c:12
Breakpoint 1 at 0x97: file cat.c, line 12.
(gdb) info breakpoint
```

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x00000097	in cat at cat.c:12

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

برای حذف یک Breakpoint، از دستور زیر، استفاده می‌شود:

```
del <breakpoint_number>
```

این دستور مخفف دستور `delete` می‌باشد. همچنین با اجرای دستور `clear`، همه breakpoint ها پاک می‌شوند.

```
(gdb) info breakpoint
```

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x00000097	in cat at cat.c:12
2	breakpoint	keep	y	0x000000fe	in cat at cat.c:20
3	breakpoint	keep	y	0x0000007d	in main at cat.c:30

```
(gdb) del 2
(gdb) info breakpoint
```

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x00000097	in cat at cat.c:12
3	breakpoint	keep	y	0x0000007d	in main at cat.c:30

کنترل روند اجرا و دسترسی به حالت سیستم

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

با هر بار فراخوانی هر تابع، یک frame پشته به آن تخصیص پیدا می‌کند که در این frame، اطلاعات مربوط به آن فراخوانی مانند return address و ... نمایش داده می‌شود. دستور bt یا همان backtrace، وضعیت frame ها را داخل پشته نشان می‌دهد که خط اول آن به معنی بالاترین frame (آخرین تابع فراخوانی شده) و پایین‌ترین خط آن به معنی پایین‌ترین frame (اولین تابع فراخوانی شده) در پشته می‌باشد.

```
(gdb) bt
#0 main (argc=-81915917, argv=0x4244c8d) at strdiff.c:74
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, main (argc=-81915917, argv=0x4244c8d)
    at strdiff.c:85
85      write(fd, "\n", 1);
(gdb) bt
#0 main (argc=-81915917, argv=0x4244c8d) at strdiff.c:85
(gdb) step
strdiff (fd=6464, w1=0x64 <main+100>, w2=0x0 <main>) at strdiff.c:32
32      {
(gdb) bt
#0 strdiff (fd=6464, w1=0x64 <main+100>, w2=0x0 <main>) at strdiff.c:32
#1 0x00000082 in main (argc=-81915917, argv=0x4244c8d) at strdiff.c:85
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

دستور bt می‌تواند آرگومان های <integer> و <integer>- داشته باشد که <integer> bt به تعداد <integer> از پایین پشته و <integer>- از بالای پشته را نمایش می‌دهد.

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

- دستور زیر، برای نمایش یک متغیر استفاده می‌شود:

```
print <variable_number>
```

- کمک⁴² دستور:

```
(gdb) help print
Print value of expression EXP.
Usage: print [[OPTION]... --] [/FMT] [EXP]

Options:
  -address [on|off]
    Set printing of addresses.

  -array [on|off]
    Set pretty formatting of arrays.

  -array-indexes [on|off]
    Set printing of array indexes.

  -elements NUMBER|unlimited
    Set limit on string chars or array elements to print.
    "unlimited" causes there to be no limit.

  -max-depth NUMBER|unlimited
    Set maximum print depth for nested structures, unions and arrays.
    When structures, unions, or arrays are nested beyond this depth then they
    will be replaced with either '{...}' or '(...)' depending on the language.
    Use "unlimited" to print the complete structure.

  -null-stop [on|off]
    Set printing of char arrays to stop at first null char.

  -object [on|off]
    Set printing of C++ virtual function tables.

  -pretty [on|off]
    Set pretty formatting of structures.

  -raw-values [on|off]
    Set whether to print values in raw form.
    If set, values are printed in raw form, bypassing any
    pretty-printers for that value.

  -repeats NUMBER|unlimited
    Set threshold for repeated print elements.
    "unlimited" causes all elements to be individually printed.

  -static-members [on|off]
    Set printing of C++ static members.
```

- دستور زیر، محتویات خانه‌ای از حافظه که آدرس آن به عنوان آرگومان داده می‌شود را نمایش می‌دهد:

```
x <memory_address>
```

- help دستور:

```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
    t(binary), f(float), a(address), i(instruction), c(char), s(string)
    and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format. If a negative number is specified, memory is
examined backward from the address.

Defaults for format and size letters are those previously used.
Default count is 1. Default address is following last thing printed
with this command or "print".
```

- نمونه اجرای دستورات:

```
(gdb) print fd
$1 = 6464
(gdb) print w1
$2 = (const struct word *) 0x64 <main+100>
(gdb) x 0x64
0x64 <main+100>:      0xc0850000
(gdb) x/c 0x64
0x64 <main+100>:      0 '\000'
```

- برای مشاهده یک ثبات خاص، از دستور زیر (که به عنوان آرگومان، نام ثبات را دریافت می‌کند) استفاده می‌کنیم:

info registers

- که با اجرای آن، خواهیم دید:

```
(gdb) info register edi
edi                0x0
(gdb) info register esi
esi                0x0
```

5) برای نمایش وضعیت ثبات‌ها از چه دستوری استفاده می‌شود؟ متغیرهای محلی چگونه؟ نتیجه این دستور را در گزارش‌کار خود بیاورید. همچنین، در گزارش خود توضیح دهید که در معماری x86، رجیسترهای edi و esi نشانگر چه چیزی هستند؟

برای مشاهده وضعیت همهٔ ثبات‌ها، از دستور `info registers` استفاده می‌کنیم:

```

(gdsh) Info registers
eax          0x0          0
ecx          0x3fec       16364
edx          0x0          0
ebx          0x0          0
esp          0x3fbc       0x3fbc
ebp          0x3fd8       0x3fd8
esi          0x0          0
edi          0x0          0
ebp          0x110        0x110 <strdlff>
eflags       0x212        [ IDPL=0 IF AF ]
cs           0x1b         27
ss           0x23         35
ds           0x23         35
es           0x23         35
fs           0x0          0
gs           0x0          0
fs_base      0x0          0
gs_base      0x0          0
kgs_base     0x0          0
cr0          0x80010011    [ PG WP ET PE ]
cr2          0x0          0
cr3          0xdf73000     [ PDBR=0 PCID=0 ]
cr4          0x10         [ PSE ]
cr8          0x0          0
efer         0x0          [ ]
xmm0         [v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = {0x0}]
xmm1         [v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = {0x0}]
xmm2         [v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = {0x0}]
xmm3         [v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = {0x0}]
xmm4         [v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = {0x0}]
xmm5         [v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = {0x0}]
xmm6         [v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = {0x0}]
xmm7         [v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 12 times>, 0x80, 0x1f, 0x0, 0x0}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x1f}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x1f8000000000}, uint128 = 0x1f80000000000000000000000000000000]
XCR3CR4      0x1f80      [ 1M 2M 3M 4M 5M 6M ]
XCR3CR5      0x1f80      [ 1M 2M 3M 4M 5M 6M ]

```

همچنین برای مشاهده متغیرهای محلی، از دستور `info locals` استفاده می‌شود:

```
(gdb) info locals
size = <optimized out>
bit = <optimized out>
bit_char = 0 '\000'
```

- توجه شود که دستور بالا، حین اشکال زدایی برنامه `strdiff` اجرا شده است.

- رجیستر ⁴³`edi`: یک رجیستر ۳۲ بیتی است که به عنوان ثبات مقصد، در عملیات‌های مربوط به رشته ⁴⁴، مورد استفاده قرار می‌گیرد.
- رجیستر ⁴⁵`esi`: یک رجیستر ۳۲ بیتی است که به عنوان ثبات مبدأ، در عملیات‌های مربوط به رشته، مورد استفاده قرار می‌گیرد.

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

- توضیح کلی این `struct` و متغیرهای درونی آن و نقش آن‌ها
- نحوه و زمان تغییر مقدار متغیرهای درونی (برای مثال، این که `input.e` در چه حالتی تغییر می‌کند و چه مقداری می‌گیرد)

این داده ساختار برای دریافت ورودی از `console` استفاده می‌شود که تعریف آن در `console.c` وجود دارد. بعد از تعریف، یک نمونه از آن گرفته شده است که `input` نام دارد. `input.buffer` در واقع `buffer` ورودی است که تمام ورودی‌های کیبورد (به جز برخی دستورات خاص) در این `buffer` (که اندازه آن ۱۲۸ کاراکتر می‌باشد)، ذخیره می‌شود.

⁴³ Extended Destination Index

⁴⁴ String Process

⁴⁵ Extended Source Index

متغیر input.r برای خواندن از ابتدای input.w استفاده می‌شود. input.w نشان دهندهٔ خانهٔ ابتدایی دستور فعلی روی buffer است. همچنین input.e نشان دهندهٔ خانهٔ پس از آخرین کاراکتر دستور فعلی روی buffer می‌باشد. محل input.e در واقع محل نوشته شدن کاراکتر بعدی است که روی ورودی می‌آید.

برای نمایش تغییرات input، ابتدا یک breakpoint روی فایل console.c قرار می‌دهیم و سپس، حالت پایهٔ input را نمایش می‌دهیم:

```
(gdb) break console.c:440
Breakpoint 1 at 0x80100e08: file console.c, line 443.
(gdb) print input
$1 = {buf = '\000' <repeats 127 times>, r = 0, w = 0, e = 0}
(gdb)
```

حال عبارت "test1" را وارد می‌کنیم و کلید Enter را می‌فشاریم:

```
(gdb) print input
$2 = {buf = "test1\n", '\000' <repeats 121 times>, r = 6, w = 6, e = 6}
(gdb)
```

سپس، عبارت "test2" را وارد می‌کنیم اما پیش از فشردن کلید Enter، وضعیت ورودی را نمایش می‌دهیم:

```
(gdb) print input
$3 = {buf = "test1\ntest2", '\000' <repeats 116 times>, r = 6, w = 6, e = 11}
(gdb)
```

در نهایت، کلید Enter را می‌فشاریم:

```
(gdb) print input
$6 = {buf = "test1\ntest2\n", '\000' <repeats 115 times>, r = 12, w = 12, e = 12}
(gdb)
```

اشکال زدایی در سطح کد اسمبلی

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

برای نمایش کارکرد این دستورات، برنامه cat_ اشکال زدایی می‌شود که به همین منظور، داخل فایل cat.c، یک breakpoint قرار داده شده است و با استفاده از دستور continue، به محل breakpoint می‌رویم:

```
(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) break 30
Breakpoint 1 at 0x7d: file cat.c, line 30.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, main (argc=-81915917, argv=0x4244c8d) at cat.c:30
30      cat(0);
(gdb)
```

در TUI، با استفاده از دستور `layout src` کد برنامه‌ای که در حال اشکال‌زدایی می‌باشد، نمایش داده می‌شود:

```
cat.c
25     main(int argc, char *argv[])
26     {
27         int fd, i;
28
29         if(argc <= 1){
B+>30             cat(0);
31             exit();
32         }
33
34         for(i = 1; i < argc; i++){
35             if((fd = open(argv[i], 0)) < 0){
36                 printf(1, "cat: cannot open %s\n", argv[i]);
37                 exit();
38             }
39             if(fdup2(fd, 1) < 0)
40                 continue;
41             close(fd);
42             if(read(fd, buf, sizeof(buf)) < 0)
43                 continue;
44             if(buf[0] < 0)
45                 continue;
46             if(buf[0] < 255)
47                 write(1, buf, buf[0]);
48             else
49                 write(1, buf, 255);
50             if(buf[0] > 0)
51                 write(1, "\n", 1);
52         }
53     }
54 }
```

remote Thread 1.1 In: main L30 PC: 0x7d

(gdb) layout src

(gdb)

- همچنین، از دستور:

layout **asm**

برای نمایش کد اسمبلی برنامه‌ای که در حال اشکال زدایی می‌باشد، استفاده می‌شود.

```
B+> 0x7d <main+125> call 0x110 <strcpy>
0x82 <main+130> add $0x10,%esp
0x85 <main+133> test %eax,%eax
0x87 <main+135> js 0x9d <cat+13>
0x89 <main+137> cmpb $0x63,0x1940
0x90 <cat> jne 0x58 <main+88>
0x92 <cat+2> cmpb $0x64,0x1941
0x99 <cat+9> jne 0x58 <main+88>
0x9b <cat+11> jmp 0x48 <main+72>
0x9d <cat+13> call 0xd93
0xa2 <cat+18> sub $0xc,%esp
0xa5 <cat+21> push $0x1940
0xaa <cat+26> call 0xbb0 <buf+48>
```

```
remote Thread 1.1 In: main L30 PC: 0x7d
(gdb) layout src
(gdb) layout asm
(gdb)
```

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

همانطور که پیش‌تر گفته شد، می‌توانیم از دستور bt (و یا where) برای مشاهدهٔ call stack استفاده کنیم. برای بالا یا پایین رفتن در پشته، به ترتیب از دستورات زیر استفاده می‌شود:

```
<up <integer  
<down <integer
```

که در آن‌ها، <integer>، یک عدد طبیعی است و تعداد حرکت در پشته را مشخص می‌کند و مقدار پیش‌فرض آن، ۱ می‌باشد.

• دستور up:

```
strdiff.c
75      initialize_word(&w2, argv[2], 2);
76
77      int fd;
78      const char *filename = "strdiff_result.txt";
79      if ((fd = open(filename, O_CREATE | O_WRONLY)) < 0)
80      {
81          printf(1, "strdiff: cannot create strdiff_result.txt\n");
82          exit();
83      }
84      strdiff(fd, &w1, &w2);
B->85      write(fd, "\n", 1);
86      close(fd);
87
88      exit();
89      }
```

```
remote Thread 1.1 In: main
(gdb) where
#0  main (argc=3, argv=0x2fd8) at strdiff.c:85
(gdb) step
write () at usys.S:16
(gdb) where
#0  write () at usys.S:16
#1  0x00000086 in main (argc=3, argv=0x2fd8) at strdiff.c:85
(gdb) up 1
#1  0x00000086 in main (argc=3, argv=0x2fd8) at strdiff.c:85
(gdb)
```


• دستور down:

```
usys.S
6      name: \
7      movl $SYS_ ## name, %eax; \
8      int $T_SYSCALL; \
9      ret
10
11      SYSCALL(fork)
12      SYSCALL(exit)
13      SYSCALL(wait)
14      SYSCALL(pipe)
15      SYSCALL(read)
>16     SYSCALL(write)
17      SYSCALL(close)
18      SYSCALL(kill)
19      SYSCALL(exec)
20      SYSCALL(open)
21      SYSCALL(mknod)
22      SYSCALL(unlink)
23      SYSCALL(fstat)
24      SYSCALL(link)
25      SYSCALL(mkdir)
26      SYSCALL(chdir)
27      SYSCALL(dup)
28      SYSCALL(getpid)

remote Thread 1.1 In: write
(gdb) where
#0  main (argc=3, argv=0x2fd8) at strdiff.c:85
(gdb) step
write () at usys.S:16
(gdb) where
#0  write () at usys.S:16
#1  0x00000086 in main (argc=3, argv=0x2fd8) at strdiff.c:85
(gdb) up 1
#1  0x00000086 in main (argc=3, argv=0x2fd8) at strdiff.c:85
(gdb) down 1
#0  write () at usys.S:16
(gdb)
```

پیکر بندی و ساختن هسته لینوکس

در ابتدا فایل پیکربندی پیش فرض سیستم عامل لینوکس را دانلود کرده ایم و به کمک دستور زیر، نسخه هسته قبل انجام این بخش را چاپ می کنیم:

```
mobina@ubuntu: ~/Downloads/linux-5.15.136
mobina@ubuntu:~/Downloads/linux-5.15.136$ uname -a
Linux ubuntu 5.15.0-86-generic #96~20.04.1-Ubuntu SMP Thu Sep 21 13:23:37 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
mobina@ubuntu:~/Downloads/linux-5.15.136$
```

نسخه قبل فرایند 5.15.0 می باشد و برای نزدیکی ورژن جدید به آن، نسخه 5.15.136 را انتخاب می کنیم. برای کم کردن هزینه (از نظر حجم هسته و زمان کامپایل)، از دستور `make defconfig` و پیکربندی پیش فرض استفاده شده است. ضمن جایگزینی هسته جدید با قبلی، به کمک دستور زیر، ورژن هسته جدید را مشاهده می کنیم:

```
mobina@ubuntu:~/Downloads$ uname -rs
Linux 5.15.136
mobina@ubuntu:~/Downloads$
```

پس این فرآیند، فایل به نام `group10.c` ساخته می شود:

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

MODULE_LICENSE("");

int init_module(void)
{
    printk(KERN_INFO "Group #10:\nMobina Mehrazar : 810100216\nAmin Yousefi : 810100236\nMatin Nabizadeh : 810100223\n");
    return 0;
}

void cleanup_module(void) {}
```

در این قطعه کد، دستور printk داریم که به کمک آن، نام اعضای گروه را چاپ می‌کنیم. نام اعضای گروه در دستور dmesg نمایش داده می‌شود. با make کردن و سپس به دستور sudo insmod group10.ko به ماژول‌های هسته اضافه می‌کنیم.

- دستورات داخل makefile به شرح زیر است:

```
obj-m += group10.o  
all:  
    make -C /lib/modules/5.15.136/build M=$(PWD) modules
```

با اجرای آخرین دستور، در شکل زیر نام اعضای گروه را در صفحه ترمینال، مشاهده می‌کنیم:

```
mobina@ubuntu:~/Downloads$ make  
make -C /lib/modules/5.15.136/build M=/home/mobina/Downloads modules  
make[1]: Entering directory '/home/mobina/Downloads/linux-5.15.136'  
make[1]: Leaving directory '/home/mobina/Downloads/linux-5.15.136'  
mobina@ubuntu:~/Downloads$ sudo insmod group10.ko  
insmod: ERROR: could not insert module group10.ko: File exists  
mobina@ubuntu:~/Downloads$ dmesg | tail -5  
[ 701.884305] Disabling lock debugging due to kernel taint  
[ 701.884650] Group #10:  
                Mobina Mehrazar : 810100216  
                Amin Yousefi    : 810100236  
                Matin Nabizadeh  : 810100223  
mobina@ubuntu:~/Downloads$ dmesg >> out.txt  
mobina@ubuntu:~/Downloads$
```

- Github Hash: c35c8a887b103869c1e775fd79c7287ad712ac81