

## آشنایی با سیستم عامل XV6

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

سه وظیفه اصلی سیستم عامل عبارت اند از:

مدیریت منابع (Resource Management):

مدیریت و تخصیص منابع سخت افزاری سیستم مانند پردازنده (CPU)، حافظه (RAM)، دستگاه های ورودی/خروجی (I/O) و فایل ها بین برنامه های مختلف.

مدیریت فایل ها و سیستم ورودی/خروجی (File and I/O Management):

فراهم کردن راهی استاندارد برای دسترسی برنامه ها به فایل ها و دستگاه ها بدون نیاز به درگیری مستقیم با جزئیات سخت افزار.

مدیریت فرآیندها (Process Management):

ایجاد، زمان بندی، اجرا، و پایان دادن به فرآیندها، و هماهنگ سازی و ارتباط بین آن ها.

### 2. آیا وجود سیستم عامل در تمام دستگاه ها الزامی است؟ چرا؟ در چه شرایطی استفاده از سیستم عامل لازم است؟

نه، وجود سیستم عامل در تمام دستگاه ها الزامی نیست. دلیل آن این است که در برخی دستگاه های ساده مانند میکروکنترلرها یا سیستم های جاسازی شده (مانند ترموستات ها، ساعت های دیجیتال یا کنترلرهای صنعتی کوچک)، برنامه ها می توانند مستقیماً روی سخت افزار اجرا شوند بدون نیاز به لایه واسط سیستم عامل. این رویکرد ساده تر، کم حجم تر و کارآمدتر است، زیرا منابع محدود هستند و نیازی به مدیریت پیچیده منابع یا چندوظیفگی نیست.

با این حال، استفاده از سیستم عامل در شرایطی لازم است که:

- نیاز به مدیریت منابع پیچیده مانند پردازنده، حافظه، دستگاه‌های ورودی/خروجی و شبکه وجود داشته باشد (مثل کامپیوترهای شخصی، سرورها یا گوشی‌های هوشمند).

- سیستم باید از چند وظیفگی، امنیت، یا اجرای همزمان چندین برنامه پشتیبانی کند.

- رابط کاربری (UI) یا API برای توسعه‌دهندگان لازم باشد تا برنامه‌نویسی آسان‌تر شود.

- دستگاه با محیط‌های پویا و کاربران متعدد سروکار داشته باشد، مانند سیستم‌های عامل عمومی (ویندوز، لینوکس)

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

با توجه به فصل اول کتاب xv6 به عنوان یک هسته سیستم‌عامل مونولیتیک پیاده‌سازی شده است، که از بیشتر سیستم‌های عامل بونیکس الگوبرداری کرده است. به همین دلیل، در xv6، رابط هسته (Kernel) معادل رابط سیستم‌عامل است، و هسته سیستم‌عامل تمام وظایف و سرویس‌های مربوط به سیستم‌عامل را پیاده‌سازی می‌کند. با این حال، از آنجا که xv6 تعداد کمتری از خدمات سیستمی را ارائه می‌دهد، هسته آن کوچک‌تر از برخی از میکروهسته‌ها (Microkernels) است. به عبارت دیگر، در xv6، تمام عملکردهای سیستم‌عامل، از جمله مدیریت حافظه، مدیریت فایل‌ها، ورود و خروج داده‌ها و متغیرهای سیستمی و غیره، به صورت یکپارچه در هسته پیاده‌سازی شده است. این معماری رابطه مستقیمی بین هسته و وظایف سیستمی دارد.

### 4. سیستم عامل xv6 یک سیستم تک وظیفه ای است یا چند وظیفه ای؟

دلایل چندوظیفه‌ای بودن xv6: پشتیبانی از فرآیندها : xv6 به چندین فرآیند اجازه می‌دهد تا به طور همزمان در حافظه بارگذاری شوند.

زمان‌بندی (Scheduling): هسته xv6 دارای یک زمان‌بند (Scheduler) است که به طور مداوم بین فرآیندهای آماده اجرا جابجا می‌شود.

مکانیزم تعویض متن (Context Switching): هنگامی که زمان‌بند تصمیم می‌گیرد یک فرآیند دیگر را اجرا کند، وضعیت CPU (محتویات رجیسترها) فرآیند فعلی ذخیره و وضعیت فرآیند جدید بارگذاری می‌شود. این کار باعث ایجاد توهم اجرای همزمان فرآیندها می‌شود.

سیستم‌کال‌ها (System Calls): وجود system call هایی مانند fork (برای ایجاد فرآیند جدید)، wait (برای انتظار والد برای پایان فرزند) و exit (برای خاتمه فرآیند) مستقیماً از قابلیت چندوظیفه‌ای پشتیبانی می‌کنند.

### 5. همانطور که می‌دانید به طور کلی چندوظیفگی تعمیمی است از حالت چند برنامه‌گی، چه

#### تفاوتی میان یک برنامه و یک پردازنده وجود دارد؟

تفاوت برنامه (Program) و پردازش (Process):

1. برنامه (Program):

یک موجودیت ایستا (Static) است.

مجموعه‌ای از دستورالعمل‌ها و داده‌ها است که روی دیسک ذخیره شده است (مثل یک فایل اجرایی).

تا زمانی که اجرا نشود، در حافظه اصلی بارگذاری نمی‌شود و هیچ فعالیتی ندارد.

2. پردازش (Process):

یک موجودیت پویا (Dynamic) است.

یک نمونه در حال اجرا (Instance) از یک برنامه است.

در حافظه اصلی بارگذاری شده و منابعی مانند CPU، حافظه، فایل‌ها و ... به آن تخصیص داده می‌شود.

دارای وضعیت اجرا (State) مانند در حال اجرا، آماده، منتظر و ... است.

ارتباط چندوظیفگی (Multitasking) و چندبرنامگی (Multiprogramming):

چندبرنامگی به معنی حضور چندین پردازش در حافظه اصلی به طور همزمان است.

چندوظیفگی شکل پیشرفته‌ای از چندبرنامگی است که در آن CPU بین پردازش‌های مختلف به سرعت جابجا می‌شود و به کاربر این توهم را می‌دهد که پردازش‌ها به طور همزمان در حال اجرا هستند.

## 6. ساختار یک پردازنده در سیستم عامل xv6 از چه بخش‌هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازنده‌های مختلف اختصاص می‌دهد؟

یک پردازش در سیستم عامل xv6 از بخش‌های زیر تشکیل شده است:

1. User-space-memory که شامل instructions و data و stack است.

2. state هر پردازش که به طور خصوصی در اختیار kernel قرار دارد

یک پردازش در این سیستم عامل از یک بخش که فقط برای هسته قابل دسترسی است و بخش دیگر که شامل دستورات و داده‌ها و stack و ... است تشکیل شده است.

این سیستم عامل با استفاده از روش time-sharing به صورت transparent پردازنده‌ها را مدیریت می‌کند. می‌دانیم روش time-sharing برای multiprogramming استفاده می‌شود. در این روش اگر برنامه‌ای در حال اجرا در صورتی که سهمیه زمانی استفاده‌اش از CPU به پایان برسد، دسترسی CPU از این process گرفته شده و محتوای رجیسترهای آن در memory ذخیره شده و سپس process بعدی اجرا می‌شود و CPU به آن تخصیص داده می‌شود. وقتی دوباره نوبت به این process رسید محتوا از memory به رجیسترها بازمی‌گردد. با این روش همه process‌ها به صورت هم‌روند رو به پیشرفت هستند. هسته سیستم عامل برای رهگیری درست هر process به آن یک کد به نام pid اختصاص می‌دهد.

7) مفهوم file descriptor در سیستم عامل‌های مبتنی بر UNIX چیست؟ عملکرد pipe

سیستم عامل xv6 چگونه است و به طور معمول برای چه هدفی استفاده می‌شود؟

File Descriptor یک عدد صحیح غیر منفی است که به عنوان یک ارجاع (Handle) برای یک شیء I/O باز شده استفاده می‌شود. این مفهوم در هسته سیستم‌عامل‌های شبه-یونیکس (مانند xv6) به این صورت کار می‌کند:

هر پردازش یک جدول فایل توصیف‌کننده‌ها مخصوص به خود دارد

سه File Descriptor استاندارد همیشه وجود دارند:

0: stdin (ورودی استاندارد)

1: stdout (خروجی استاندارد)

2: stderr (خطای استاندارد)

- هرگاه یک فایل باز می‌شود، یا یک سوکت ایجاد می‌شود، یک File Descriptor جدید به آن اختصاص داده می‌شود

عملکرد Pipe در xv6 و کاربرد آن:

Pipe در xv6 یک کانال ارتباطی یک طرفه بین دو پردازش است که به صورت زیر عمل می‌کند:

کاربردهای اصلی Pipe:

1. ارتباط بین پردازشی (IPC): برقراری ارتباط بین پردازش والد و فرزند

2. زنجیره کردن دستورات: مانند استفاده از ``` در shell برای اتصال خروجی یک دستور به ورودی دستور دیگر

3. هماهنگی بین پردازش‌ها: برای همگام‌سازی عملیات پردازش‌های مختلف

## 8. فراخوانی‌های سیستمی exec و fork در سیستم‌عامل xv6 چه عملیاتی را انجام

می‌دهند؟ 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 می‌کند که به آن "پردازش فرزند" (child process) گفته می‌شود و دقیقاً همان محتوای حافظه‌ای را دارد که پردازش فراخواننده، که به آن "پردازش والد" (parent process) گفته می‌شود، دارد. در نهایت، فراخوان سیستمی fork از هر دو پردازش فراخواننده و فرزند باز می‌گردد. در پردازش والد، فراخوان fork شناسه پردازش فرزند را برمی‌گرداند؛ و در پردازش فرزند، صفر را برمی‌گرداند.

فراخوانی سیستمی exec حافظه پردازش فراخواننده را با یک تصویر حافظه جدید که از یک فایل در فایل سیستم بارگیری شده است، جایگزین می‌کند. این فایل باید یک فرصت خاص داشته باشد که مشخص می‌کند کدام قسمت از فایل دستورات را نگه می‌دارد، کدام قسمت داده‌ها را، از کجا باید اجرا آغاز شود و موارد مشابه. xv1 از فرصت ELF استفاده می‌کند بنابراین وقتی فراخوان exec موفقیت‌آمیز باشد، به برنامه فراخواننده بازگردانده نمی‌شود؛ به جای آن، دستورات بارگیری شده از فایل از نقطه ورودی اعلام شده در هدر ELF شروع به اجرا کنند. به عبارت دیگر، با استفاده از

exec می‌توان یک برنامه موجود در یک فایل جایگزین برنامه فعلی پردازش می‌کرد تا پردازش جدید با کدها و داده‌های موجود در فایل ادامه یابد.

اگر fork و exec جدا باشند شل می‌تواند یک فرزند ایجاد کند و در آن از توابع open, close, 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 exec و fork 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.

## 9. دستور make -n را اجرا نمایید. کدام دستور نهایی فایل هسته را می‌سازد؟

دستوری که فایل نهایی هسته (kernel) را می‌سازد، معمولاً یک دستور لینکر (ld) است که:

تمام فایل‌های object (مانند main.o, vm.o, proc.o, ...) را به هم لینک می‌کند

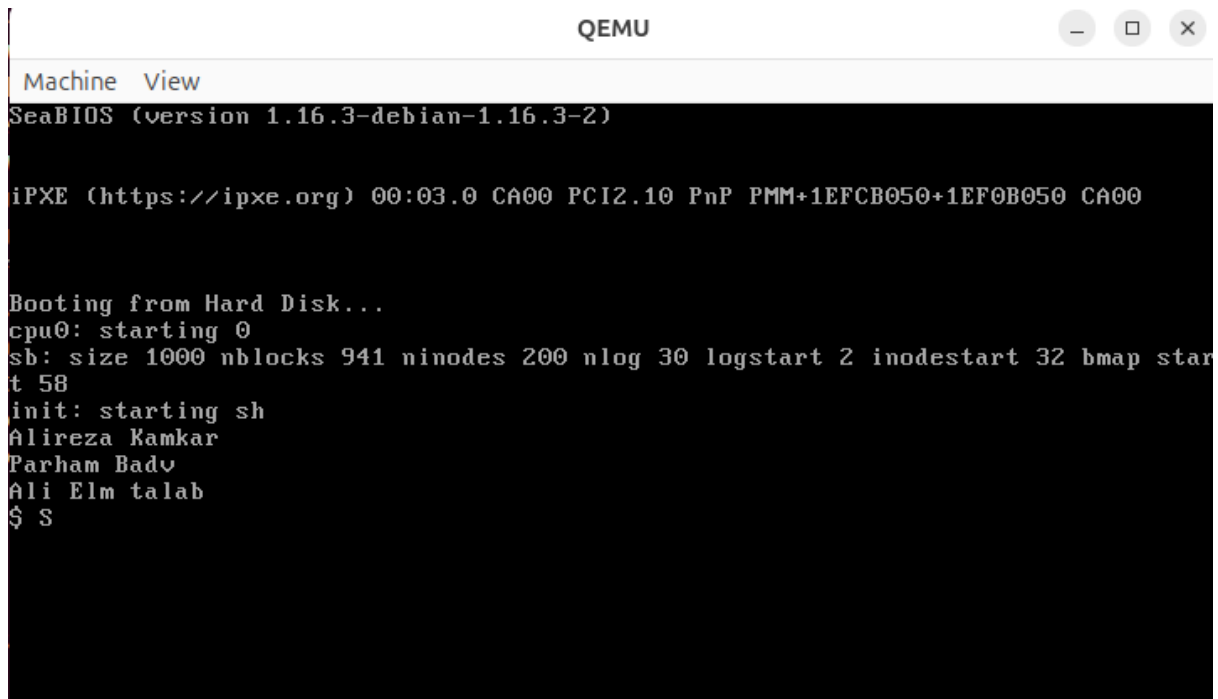
آدرس شروع را روی 0x100000 تنظیم می‌کند

فایل خروجی kernel را تولید می‌کند

## اضافه کردن یک متن به Boot Message

برای نمایش دادن نام اعضای گروه پس از بوت شدن سیستم عامل کافایت که اسامی را با دستور printf به فایل init.c اضافه می‌کند.

```
22  for(;;){
23      printf(1, "init: starting sh\n");
24      printf(1, "Alireza Kamkar\n");
25      printf(1, "Parham Badv\n");
26      printf(1, "Ali Elm talab\n");
27      pid = fork();
28      if(pid < 0){
29          printf(1, "init: fork failed\n");
```



```
QEMU
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Alireza Kamkar
Parham Badv
Ali Elm talab
$ S
```

اضافه کردن چند قابلیت به کنسول:

## 1. قابلیت جا به جایی مکان نما:

دو تابع برای مدیریت مکان نما به عقب و جلو در console.c ایجاد میکنیم سپس دستورات shiftLeft و shiftRight را برای اینکه بافر به درستی کاراکترها را در جای خود ذخیره کند، اضافه میکنیم. همچنین قسمت مربوط به BACKSPACE هم که مرتبط با ← و → را اصلاح میکنیم. در آخر نیز دستورات ← و → را به بخش consoleintr اضافه میکنیم.

```

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

    // move back
    if(crt[pos - 2] != ('$' | 0x0700))
        pos--;
    // reset cursor
    outb(CRTPORT, 14);
    outb(CRTPORT+1, pos>>8);
    outb(CRTPORT, 15);
    outb(CRTPORT+1, pos);
}

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

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

```

```

static void shiftRight(char *buf)
{
    for (int i = input.e; i > input.e - numBack; i--)
    {
        buf[(i) % INPUT_BUF] = buf[(i - 1) % INPUT_BUF];
    }
}

static void shiftLeft(char *buf)
{
    for (int i = input.e - numBack - 1; i < input.e; i++)
    {
        buf[(i) % INPUT_BUF] = buf[(i + 1) % INPUT_BUF];
    }
    input.buf[input.e] = ' ';
}

```

```

case LEFT_ARROW:
    if((input.e - numBack) > input.w){
        backCursor();
        numBack++;
    }
    if((input.e - numBackSaved) > input.w && is_copy == 1){
        numBackSaved++;
    }
    break;
case RIGHT_ARROW:
    if(numBack > 0){
        forwardCursor();
        numBack--;
    }
    if(numBackSaved > 0 && is_copy == 1){
        numBackSaved--;
    }
    break;

```





```
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Alireza Kamkar
Parham Badv
Ali Elm talab
$ ddege
```

برای Ctrl+A و Ctrl+D ابتدا دو تابع برای پیدا کردن ابتدا کلمه بعدی و فعلی مینویسیم و سپس در بخش consoleintr اضافه میکنیم.

```

static int find_next_word_start() {
    int current_pos = input.e - numBack;
    char *buf = input.buf;

    if(current_pos >= input.e) return 0;

    while(current_pos < input.e && buf[current_pos % INPUT_BUF] == ' ') {
        current_pos++;
    }
    while(current_pos < input.e && buf[current_pos % INPUT_BUF] != ' ') {
        current_pos++;
    }
    while(current_pos < input.e && buf[current_pos % INPUT_BUF] == ' ') {
        current_pos++;
    }

    return current_pos - (input.e - numBack);
}

static int find_prev_word_start() {
    int current_pos = input.e - numBack - 1;
    char *buf = input.buf;

    if(current_pos < (int)input.w) return 0;

    while(current_pos >= (int)input.w && buf[current_pos % INPUT_BUF] == ' ') {
        current_pos--;
    }
    while(current_pos >= (int)input.w && buf[current_pos % INPUT_BUF] != ' ') {
        current_pos--;
    }

    return (input.e - numBack) - (current_pos + 1);
}

```

```

case CTRL_D: // Move to beginning of next word
    if(numBack > 0) {
        int move = find_next_word_start();
        if(move > 0 && move <= numBack) {
            for(int i = 0; i < move; i++) {
                forwardCursor();
                numBack--;
            }
        }
    }
    break;
case CTRL_A: // Move to beginning of current/previous word
    if(input.e - numBack > input.w) {
        int move;

        if(numBack == 0 || input.buf[(input.e - numBack) % INPUT_BUF] == ' ') {
            move = find_prev_word_start();
        } else {
            move = 0;
            int temp_pos = numBack;
            while(temp_pos > 0 && input.buf[(input.e - temp_pos) % INPUT_BUF] != ' ') {
                move++;
                temp_pos--;
            }
        }

        if(move > 0) {
            for(int i = 0; i < move; i++) {
                backCursor();
                numBack++;
            }
        }
    }
    break;

```

## 2. قابلیت حذف آخرین کاراکتر وارد شده:

مانند قبلی یک تابع اضافه میکنیم و در consoleintr هم اضافه میکنیم.

```

static void delete_last_char() {
    if(input.e > input.w) {
        input.e--;
        consputc(BACKSPACE);

        if(numBack > 0) {
            numBack--;
        }
    }
}

```

```

case CTRL_Z: // Delete last character (by time)
    delete_last_char();
    break;
default:
    if(c != 0 && input.e-input.r < INPUT_BUF){
        c = (c == '\r') ? '\n' : c;
        input.buf[input.e++ % INPUT_BUF] = c;
        consputc(c);
        if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
            input.w = input.e;
            wakeup(&input.r);
            numBack = 0; // reset cursor after enter
        }
    }
    break;
}

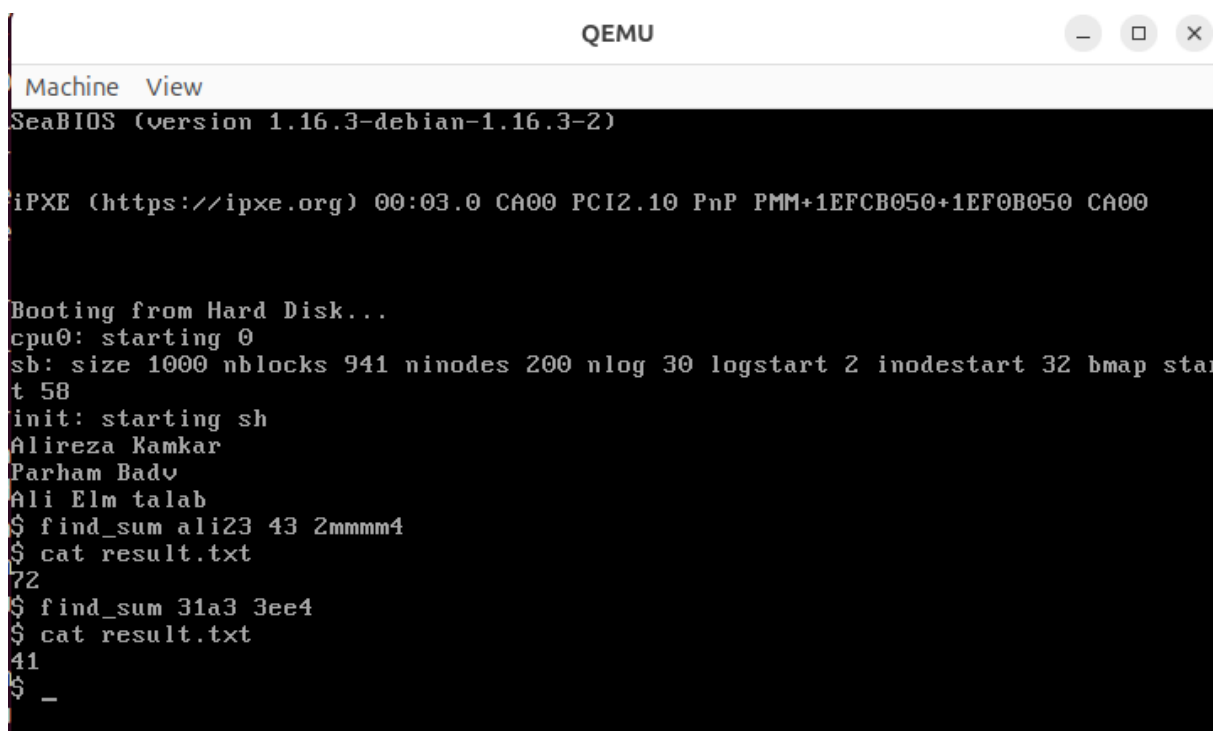
```

## قابلیت های شما:

کلیدی برای پاک کردن کل عبارت پرینت شده و کلید بالا و پایین برای رفتن به کاراکتر بعدی و قبلی از نظر ترتیب مثلا وقتی c باشد با فشردن دکمه بالا به d تبدیل شود و با فشردن دکمه پایین به b تبدیل شود

## برنامه سطح کاربر:

برای این کار یک برنامه به زبان C به نام find\_sum.c میسازیم و کد خود را در آن جا می نویسیم. سپس این برنامه را باید به برنامه های سطح کاربر اضافه کنیم که برای این کار باید تغییراتی در MakeFile اعمال کنیم.



```

QEMU
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
t 58
init: starting sh
Alireza Kamkar
Parham Badv
Ali Elm talab
$ find_sum ali23 43 2mmmm4
$ cat result.txt
72
$ find_sum 31a3 3ee4
$ cat result.txt
41
$ _

```

برنامه نوشته شده را به متغیرهای EXTRA و UPROGS در MakeFile اضافه میکنیم.

```
UPROGS=\
_cat\
_echo\
_forktest\
_grep\
_init\
_kill\
_ln\
_ls\
_mkdir\
_rm\
_sh\
_stressfs\
_usertests\
_wc\
_zombie\
_find_sum\
```

## 10. در Makefile متغیرهایی به نام های ULIB و UPROGS تعریف شده است. کاربرد آنها چیست؟

در سیستم عامل xv6، متغیر "UPROGS" که مختصر user program است، در فایل Makefile لیستی از نام های برنامه های کاربری است که باید در هنگام کامپایل xv6 ساخته شوند. این برنامه ها برنامه هایی هستند که کاربران می توانند در سیستم xv6 اجرا کنند. وقتی دستور "make" را برای کامپایل xv6 اجرا می کنیم، Makefile از متغیر "UPROGS" برای تعیین برنامه های کاربری که باید به عنوان بخشی از سیستم عامل کامپایل و لینک شوند، استفاده می کند.

متغیر "ULIB" که مختصر user libraries است، برای مشخص کردن کتابخانه سطح کاربری استفاده می شود که برنامه های سطح کاربری در طی فرآیند کامپایل به آن لینک می شوند. این متغیر، کتابخانه C را برای برنامه های کاربری در xv6 تعیین می کند. فایل "ULIB" حاوی توابع و ابزارهایی است که برنامه های کاربری می توانند از آنها برای انجام کارهای مختلف و دسترسی به توابع استاندارد کتابخانه C استفاده کنند.

## 11. اگر به فایل های موجود در xv6 دقت کنید، می بینید که فایل مربوط به دستور cd، برخلاف دستورات مانند ls و cat، وجود ندارد و این دستور در سطح کاربر اجرا نمی شود.

## توضیح دهید که این دستور cd در کجا اجرا می‌شود. به نظر شما دلیل این تفاوت میان دستور cd و دستورات دیگر مثل ls و cat چیست؟

محل اجرای cd: دستور cd در پوسته (shell) به صورت built-in اجرا می‌شود، نه به عنوان یک برنامه مستقل سطح کاربر.

دلیل تفاوت:

تغییر وضعیت فرآیند والد:

وقتی cd اجرا می‌شود، باید دایرکتوری جاری پروسه پوسته (shell parent) را تغییر دهد.

اگر cd به عنوان برنامه مستقل اجرا شود، فقط دایرکتوری جاری پروسه فرزند خودش را تغییر می‌دهد.

پس از پایان اجرای برنامه فرزند، این تغییر از بین می‌رود و پوسته در دایرکتوری قبلی باقی می‌ماند.

محدودیت fork/exec:

در مدل fork/exec، هر برنامه جدید یک کپی از فرآیند والد است.

تغییرات محیطی در فرآیند فرزند بر والد تأثیر نمی‌گذارد.

بنابراین cd باید مستقیماً در پوسته اجرا شود تا محیط پوسته تغییر کند.

## قابلیت تکمیل خودکار

ابتدا آرایه‌ای از دستورات قابل تکمیل تعریف کردیم.

```
static char *commands[] = {
    "ls", "ln", "wc", "cat", "echo", "grep", "kill", "mkdir", "rm",
    "sh", "ps", "forktest", "init", "date", "touch", "find_sum"
};
static int ncommands = sizeof(commands) / sizeof(commands[0]);
```

سپس چند توابع کمکی در console.c اضافه کردیم.

```

static int get_word_start() {
    int abs_end = (int)input.e;
    int abs_cursor = abs_end - numBack; // absolute index where cursor is (before char at abs_cursor)
    int abs_start = abs_cursor;
    // move left until start or space or input.w
    while(abs_start > (int)input.w && input.buf[(abs_start-1) % INPUT_BUF] != ' ')
        abs_start--;
    return abs_start;
}

// helper to copy prefix into local buffer; returns length
static int copy_prefix(char *dst) {
    int abs_start = get_word_start();
    int abs_cursor = (int)input.e - numBack;
    int len = abs_cursor - abs_start;
    if(len <= 0) {
        dst[0] = '\0';
        return 0;
    }
    for(int i = 0; i < len; i++) {
        dst[i] = input.buf[(abs_start + i) % INPUT_BUF];
    }
    dst[len] = '\0';
    return len;
}

// helper: insert a character ch at cursor position (i.e. at abs_cursor) and update input.e
static void insert_char_at_cursor(char ch) {
    int abs_cursor = (int)input.e - numBack;
    // shift buffer content right by 1 from abs_cursor..input.e-1
    for(int i = (int)input.e; i > abs_cursor; i--) {
        input.buf[i % INPUT_BUF] = input.buf[(i-1) % INPUT_BUF];
    }
    input.buf[abs_cursor % INPUT_BUF] = ch;
    input.e++;
    // visually output char at cursor; cgaputc uses numBack to shift visible chars right
    consputc((int)ch);
}

```

درون switch(c)، یک case جدید برای Tab اضافه شده است:

```

case KEY_TAB: {
    // Autocomplete handling
    char prefix[INPUT_BUF];
    int prefix_len = copy_prefix(prefix); // returns 0 if empty
    int matches = 0;
    int match_idx = -1;

    // find matches in commands[]
    for(int i = 0; i < ncommands; i++) {
        if(prefix_len == 0) {
            // empty prefix matches all
            matches++;
            match_idx = i; // last match index; if matches==1 later will be used
        } else {
            // compare prefix
            int ok = 1;
            for(int j = 0; j < prefix_len; j++) {
                if(commands[i][j] == '\0' || commands[i][j] != prefix[j]) {
                    ok = 0;
                    break;
                }
            }
            if(ok) {
                matches++;
                match_idx = i;
            }
        }
    }
}
}

```

## 12. در سکتور نخست دیسک قابل بوت، محتوای چه فایلی قرار دارد؟

در سکتور نخست دیسک قابل بوت، محتوای فایل `bootblock` قرار دارد. این فایل از کامپایل و لینک فایل‌های `bootmain.c` و `bootasm.S` ایجاد می‌شود.

```

ld -m elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock

```

مراحل ایجاد bootblock:

1. کامپایل: `bootmain.c` و `bootasm.S` به فایل‌های object کامپایل می‌شوند

2. لینک: فایل‌های object با آدرس شروع `0x7C00` لینک می‌شوند



3. تبدیل به باینری: با `objcopy` به فرمت باینری خالص تبدیل می‌شود

4. امضا: با `sign.pl` امضا شده و در سکتور اول دیسک قرار می‌گیرد

وظیفه bootblock:

اولین کدی است که BIOS از دیسک می‌خواند

در آدرس `0x7C00` حافظه بارگذاری می‌شود

پردازنده را از Real Mode به Protected Mode منتقل می‌کند

هسته سیستم‌عامل (kernel) را از دیسک بارگذاری می‌کند

کنترل را به هسته منتقل می‌کند.

**13. 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 که 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، فایل‌های باینری آبجکت (object files) از فرمت ELF (Executable Linkable Format) پیروی می‌کنند.

در فایل‌های ELF بخش‌ها (Sections) نقش مهمی ایفا می‌کنند. بخش‌ها اطلاعات مختلف را در فایل‌های اجرایی و کتابخانه‌ها ذخیره می‌کنند و به لینکر (linker) و لودر (loader) کمک می‌کنند تا کد و داده‌ها را به درستی ادغام و بارگذاری کنند. هر بخش در ELF یک نوع خاص دارد که تعیین کننده وظیفه‌اش است. برخی از نوع‌های معمول بخش‌ها شامل "text" (برای کد اجرایی)، "data" (برای داده‌های اجرایی)، "rodata" (برای داده‌های تنها خواندنی) و "bss" (برای داده‌های اولیه با مقدار صفر) هستند. این نوع‌ها به لینکر و لودر اطلاع می‌دهند که چگونه با هر بخش برخورد کنند.

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

```
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	000000be	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	00000585	00000000	00000000	00000358 2**0
			CONTENTS, READONLY, DEBUGGING, OCTETS		
5	.debug_abbrev	0000023c	00000000	00000000	000008dd 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	00000fbd 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 می‌شود زمانی که این دستور را اجرا می‌کنیم، لیستی از هدرهای بخش‌ها در فایل آبجکت مشخص شده نمایش داده می‌شود. این اطلاعات می‌تواند برای درک نحوه سازماندهی و ساختار فایل آبجکت مفید باشد، اگر bootblock.o را با فایل‌های آبجکت دیگر مقایسه کنیم، متوجه می‌شویم که بخش‌های data و ... را ندارد و فقط بخش text را دارد.

حال دستور زیر را اجرا می‌کنیم:

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

زمانی که این دستور را اجرا می‌کنیم، کد ماشینی خام از بخش ".text" فایل آبجکت 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) مطابقت ندارد و هیچ هدری در خود نمی‌گیرد. این فایل شامل کد اجرایی خام بدون هیچ اطلاعات اضافی 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 510 بایت جا می‌گیرد. این امر دارای اهمیت ویژه برای بوت‌سکتورها است چرا که باید در 512 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 سیستم‌عامل و کاهش اندازه فایل برای اجرای موفقیت‌آمیز در محیط بوت کامپیوتر است.

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

```
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 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
```

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

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، از "objcopy" 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 بوت‌لودر) و سادگی ساختار آنها است. این اقدام ضروری است تا بوت‌لودر و هسته بتوانند به درستی بارگذاری و اجرا شوند.

## 15. در فایل‌های موجود در XV6 مشاهده می‌شود که بوت سیستم توسط فایل‌های bootmain.c و bootasm.S صورت می‌گیرد. چرا تنها از کد C استفاده نشده است؟

دلیل اصلی: نیاز به عملیات سطح پایین سخت‌افزار. کد C نمی‌تواند کارهای ضروری زیر را در مرحله اولیه بوت انجام دهد:

1. غیرفعال کردن وقفه‌ها

2. تنظیم رجیسترهای سگمنت

### 3. فعال کردن A20 Line

- دسترسی مستقیم به پورت‌های I/O فقط در اسمبلی ممکن است

### 4. انتقال به Protected Mode

- کار با رجیسترهای کنترل (CR0) فقط در اسمبلی ممکن است

### 5. پرش به کد 32 بیتی

- تغییر کد سگمنت نیاز به دستور پرش خاص دارد

اسمبلی محیط را آماده می‌کند و سپس کنترل را به کد C می‌سپارد. بدون اسمبلی، کد C نمی‌تواند در محیط Real Mode اجرا شود زیرا کتابخانه استاندارد C و runtime environment هنوز بارگذاری نشده‌اند.

## 16. یک ثابت عام منظوره، یک ثابت قطعه، یک ثابت وضعیت و یک ثابت 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 8086، 8 عدد رجیستر عام منظوره دارد. از این رجیسترها می‌توان به رجیستر شمارنده اشاره کرد. این به عنوان رجیستر شمارنده count register شناخته می‌شود. 16 بیت آن به دو رجیستر 8 بیتی تقسیم می‌شود، CH و CL که اجازه اجرای دستورات 8 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 8086، چهار رجیستر قطعه وجود دارد: cs، ds، es 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 16 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. همانطور 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 64 کیلوبایت برش در 8086، برنامه‌هایی که با این محدودیت در تداخل هستند می‌توانند بیشتر از 64 کیلوبایت باشند. می‌توان برش‌های مختلفی از کد را در حافظه قرار داد. از آنجا که می‌توان مقدار رجیستر cs را تغییر داد، می‌توانید به برش جدیدی از کد منتقل شده و دستورات موجود در آنجا را اجرا کرد.

ثبات وضعیت: در معماری میکروپروسسور 8086، ویژگی‌های "وضعیتی" خاصی وجود ندارد، مشابه ویژگی‌های معمول در برخی میکروپروسسورها. به جای آن، پردازنده 8086 از مجموعه‌ای از پرچم‌ها در رجیستر FLAGS (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 16 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 32 بیت و در معماری AMD64 (حالت بلند) به عرض 64 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.

**17. پردازنده‌های x86 دارای مدهای مختلفی هستند. هنگام بوت، این پردازنده‌ها در مد حقیقی قرار دارند؛ مدی که سیستم‌عامل اماس‌داس (MS-DOS) در آن اجرا می‌شد. یک نقص اصلی این مد را بیان نمایید. آیا در پردازنده‌های دیگر مانند RISC-V یا ARM نیز مدها به همین شکل هستند یا خیر؟ توضیح دهید.**

مشکلات Real Mode:

1. دسترسی مستقیم به تمام حافظه:

هر برنامه می‌تواند به هر آدرس حافظه دسترسی داشته باشد

برنامه‌های کاربر می‌توانند حافظه کرنل را تغییر دهند

هیچ جداسازی بین برنامه‌های مختلف وجود ندارد

2. عدم مدیریت سطوح دسترسی:

همه کدها با بالاترین سطح امتیاز اجرا می‌شوند

هیچ تفکیکی بین کد کاربر و کد سیستمی وجود ندارد

3. محدودیت آدرس‌دهی:

فقط 1MB حافظه قابل آدرس‌دهی است

استفاده از سگمنت: آفست برای آدرس‌دهی پیچیده است

4. عدم پشتیبانی از حافظه مجازی:

هیچ مکانیزمی برای صفحه‌بندی (Paging) وجود ندارد

برنامه‌ها مستقیماً به حافظه فیزیکی دسترسی دارند

مقایسه با RISC-V و ARM:

RISC-V:

تفاوت: از ابتدا در حالت Machine با حفاظت کامل شروع می‌شود

مزیت: نیاز به تغییر حالت مانند x86 ندارد

کاربرد: M-mode برای کرنل، S-mode برای سیستم‌عامل، U-mode برای کاربر

ARM:

تفاوت: دارای سطوح دسترسی از ابتدا

مزیت: اجرای امن کد کاربر در EL0 با محدودیت دسترسی

- کاربرد: EL1 برای کرنل، EL0 برای برنامه‌های کاربر

این تفاوت طراحی نشان می‌دهد که x86 برای سازگاری با سیستم‌های قدیمی طراحی شده، در حالی که RISC-V و ARM با معماری‌های امن‌تر و مدرن‌تر توسعه یافته‌اند.

**18. یکی دیگر از مدهای مهم، مد حفاظت‌شده می‌باشد. وظیفه‌ی اصلی این مود چیست؟**

**پردازنده‌ها در چه زمانی در این مود قرار می‌گیرند؟**

وظایف اصلی Protected Mode:

1. حفاظت حافظه (Memory Protection)

- جداسازی حافظه: هر پردازنده فقط به حافظه مخصوص خود دسترسی دارد

- کنترل دسترسی: تعیین سطوح read/write/execute برای هر بخش حافظه

- پیشگیری از تداخل: جلوگیری از دسترسی برنامه‌های کاربر به حافظه کرنل

2. سطوح امتیاز (Privilege Levels)

- Ring 0: سطح کرنل - دسترسی کامل به سخت‌افزار

- Ring 3: سطح کاربر - دسترسی محدود

- کنترل دستورات: دستورات حساس فقط در Ring 0 قابل اجرا هستند

3. مدیریت حافظه پیشرفته

- Segmentation: تقسیم حافظه به سگمنت‌های مجزا

- Paging: پشتیبانی از حافظه مجازی با صفحه‌بندی

- آدرس‌دهی 32-bit: دسترسی به 4GB حافظه

4. مدیریت وقفه‌های امن

- Interrupt Descriptor Table: مدیریت کنترل‌شده وقفه‌ها

- Gate Mechanisms: انتقال امن بین سطوح امتیاز

زمان قرارگیری در Protected Mode:

- پس از اجرای bootasm.S و قبل از فراخوانی bootmain.c

- قبل از بارگذاری هسته در حافظه

- در مرحله اولیه بوت توسط بوت‌لودر

**19. کد bootmain.c هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در آدرس**

**0x100000 قرار می‌دهد. علت انتخاب این آدرس چیست؟**

برای این کار چند دلیل وجود دارد. دلیل اول بحث‌های تاریخی است. منظور این است که در سیستم‌عامل‌های زیادی در گذشته مثل نسخه‌های اولیه linux از این ساختار استفاده شده بود و دلیلش هم ایجاد یک جدایی کامل بین قسمت‌های BIOS (و یا در سیستم‌های قدیمی‌تر DOS) با قسمت kernel است که CPU را وارد یک حالت protected می‌کند که از فایل‌های BIOS و bootblock محافظت می‌کند.

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

از طرفی چون 1 مگابایت رند است کارهای حافظه برای jump یا پیدا کردن کرنل راحت‌تر است.

همانطور که گفته شد، کدهای bootloader به فضای اضافی برای فضای stack یا حافظه‌ای برای فایل‌های موقت نیاز دارد و 1 مگابایت فضای کافی‌ای است که اطمینان حاصل می‌کند فایل‌های kernel از فایل‌های BIOS routine و system-function های سطح پایین دیگر، جدا است و هیچ کدام در فایل‌های دیگری overwrite انجام نمی‌دهند.

## اشکال زدایی

### 20. برای مشاهده Breakpoint ها از چه دستوری استفاده می‌شود؟

از دستور info breakpoints استفاده می‌شود.

### 21. 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.

### 22. دستور bt را اجرا کنید. خروجی آن چه چیزی را نشان می‌دهد؟

در GDB دستور bt که مخفف "backtrace" است، برای نمایش پشته‌ی فراخوان‌های فعلی استفاده می‌شود. این دستور نشان می‌دهد که چگونه برنامه از نقطه فعلی اجرای خود به اینجا رسیده است، به عبارت دیگر، نشان می‌دهد که چه توابعی به ترتیب فراخوان شده‌اند تا به نقطه فعلی برسیم.

خروجی این دستور شامل لیستی از توابع فراخوان شده به همراه پارامترهای ورودی آن‌ها و مکان‌هایی در کد که این توابع فراخوان شده‌اند، می‌باشد.

### 23. 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 محتوای 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 لحظه‌ی فعلی استفاده می‌شود درحالی‌که از x برای بررسی محتوای خام حافظه و بررسی آن در فرمت‌های مختلف استفاده می‌شود.

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

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

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

```
(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 ]
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
fs_gs_base   0x0         0
cr0          0x80010011   [ PG WP ET PE ]
cr2          0x0         0
cr3          0xdf13000   [ PDBR=57107 PCID=0 ]
cr4          0x10        [ PSE ]
cr8          0x0         0
rfer         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 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 <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}
```

با استفاده از دستورهای info registers و info locals می‌توان 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 مخفف Extended Source Index بوده و به عنوان اندیس مبدأ در عملیات‌های مربوط به رشته استفاده می‌شود.

## 25. 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.
- برای نمایش تغییرات input ابتدا یک breakpoint روی فایل console.c قرار می‌دهیم و سپس حالت پایه 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)
```

## 26. خروجی دستورهای layout asm و layout src 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:

```
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
41     if(readeflags() & FL_IF)
42         panic("mycpu: called with interrupts enabled\n");
43     apicid = lapicid();
44     // APIC IDs are not guaranteed to be contiguous. Maybe we should have
45     // a reverse map, or reserve a register to store &cpus[i].
46     for (i = 0; i < ncpu; ++i) {
47         if (cpus[i].apicid == apicid)
48             return &cpus[i];
49     }
50     panic("unknown apicid\n");
51 }
52 // Disable interrupts so that we are not rescheduled
53 // while reading proc from the cpu structure
54 struct proc*
55 myproc(void) {
56     struct cpu *c;
57     struct proc *p;
58 }
59
60
```

```
0x00104208 <cpuId>    push    %ebp
0x00104209 <cpuId+1>   mov     %esp, %ebp
0x0010420a <cpuId+3>    sub     $0x8, %esp
0x0010420b <cpuId+6>    call   0x00104200 <mycpu>
0x0010420c <cpuId+11>   leave   %esp
0x0010420d <cpuId+12>   sub     $0x00112cc0, %eax
0x0010420e <cpuId+17>   sar     $0x4, %eax
0x0010420f <cpuId+20>   imul    $0xba2e8ba3, %eax, %eax
0x00104210 <cpuId+26>   ret
0x00104211 <cpuId+26>   lea     0x0(%esi, %eiz, 1), %esi
0x00104212 <myproc>    push    %ebp
0x00104213 <myproc+1>   mov     %esp, %ebp
0x00104214 <myproc+3>   push    %ebx
0x00104215 <myproc+4>   sub     $0x4, %esp
0x00104216 <myproc+7>   call   0x00104d00 <pushcli>
0x00104217 <myproc+12>  call   0x00104200 <mycpu>
0x00104218 <myproc+17>  mov     0x4(%eax), %ebx
0x00104219 <myproc+23>  call   0x00104d00 <popcli>
0x0010421a <myproc+28>  mov     %ebx, %eax
0x0010421b <myproc+30>  mov     -0x4(%ebp), %ebx
0x0010421c <myproc+33>  leave   %ebp
0x0010421d <myproc+34>  ret
0x0010421e <myproc+34>  lea     0x0(%esi, %eiz, 1), %esi
0x0010421f <myproc+34>  lea     0x0(%esi), %esi
0x00104220 <userinit>   push    %ebp
0x00104221 <userinit+1>  mov     %esp, %ebp
0x00104222 <userinit+3>  push    %ebx
0x00104223 <userinit+4>  sub     $0x4, %esp
```

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

دیدیم که پشته فراخوانی را با استفاده از دستور backtrace یا bt می‌بینیم. حال پس از آن برای جابه‌جایی میان توابع زنجیره فراخوانی می‌توان از دستور up و down استفاده کرد.

```
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)
```



