به نام خدا

دانشگاه تهران
دانشکده مهندسی برق و کامپیوتر

تمرین کامپیوتری ۱

آزمایشگاه سیستم‌عامل

دکتر کارگهی

اعضای گروه:

- **علی قنبری** **۸۱۰۱۹۹۴۷۳**
- **بهراد علمی** **۸۱۰۱۹۹۵۵۷**
- **بیتا نصیری** **۸۱۰۱۹۹۵۰۴**

نیمسال اول ۰۳-۱۴۰۲

آشنایی با سیستم‌عامل xv6

۱. معماری سیستم‌عامل xv6

معماری سیستم‌عامل xv6 از نوع x86 است.

شاید یکی از ابتدایی‌ترین دلایلی که می‌توان آورد، این باشد که معماری سیستم‌عامل xv6 برگرفته شده از Unix v6 است که مبتنی بر معماری x86 می‌باشد. (Unix v6 توسط Dennis Ritchi و Ken Thompson به منظور اهداف آموزشی بنا شد.) علاوه بر آن در دفاع از این نظر می‌توان به فایل x86.h اشاره کرد که از دستورات assembly مختص پردازنده‌های x86 استفاده شده‌است. در فایل trap.h نیز می‌توانیم مشاهده کنیم که trapهای مختص معماری x86 پیاده‌سازی شده‌اند. در دیگر فایل‌های "basic headers" نظیر asm.h و mmu.h نیز می‌توان اشاراتی به معماری x86 مشاهده کرد.

از طرفی مطابق با صفحه ۱۷ مستند xv6 (کتاب book-rev11):

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

۲. بخش‌های پردازه[1] و چگونگی اختصاص پردازنده به پردازه‌های مختلف

یک پردازه در xv6 از دو بخش تشکیل می‌شود:

۱. حافظه فظای کاربری یا user-space که شامل دستورات، داده‌ها و استک است. (دستورات محاسبات برنامه را انجام می‌دهند، داده‌ها بخش‌هایی هستند که محاسبات روی آنها انجام می‌شود و استک بخشی است که فرایند فراخوانی‌های برنامه در آن مدیریت می‌شود.)

۲. وضعیت پردازه (per-process state) که فقط برای هسته قابل رؤیت است.

روش مورد استفاده در xv6 برای مدیریت پردازه‌های مختلف، time-share است. در این حالت سیستم‌عامل به‌صورت نامحسوس پردازنده‌ها را به پردازه‌های مختلف که منتظر اجرا هستند، اختصاص می‌دهد. هر وقت یک پردازه قرار است از اجرا توسط پردازنده خارج شود، سیستم‌عامل registerهای cpu که حاوی مقادیر مورد نیاز آن پردازه بوده را ذخیره می‌کند تا در زمان بازگشت مجدد به آن پردازه، بتواند آنها را بازیابی کند.

هسته xv6 به هر پردازه یک شناسه یکتا[2] اختصاص می‌دهد که با استفاده از system call ء `getpid()` می‌توان مقدار آن را برای پردازه کنونی دریافت کرد.

[1] Process
[2] Process Identifier (PID)

تکمیلی:

اجزای تعریف کننده وضعیت پردازه (per-process state) را در فایل‌های xv6 می‌توان مشاهده کرد:

proc.h

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

`struct proc` در سیستم‌عامل xv6 نشان‌دهنده بلاک کنترل فرایند[1] است و فیلدهای مختلفی دارد که اطلاعات مهم در مورد پردازه را ذخیره می‌کند. توضیح راجع به فیلدهای این ساختار[2] در بخش‌های بعدی داده شده‌است.

۳. مفهوم file descriptor و عملکرد pipe در سیستم‌عامل xv6

file descriptor در واقع یک عدد صحیح کوچک نامنفی است که به یک شی تحت مدیریت kernel اشاره می‌کند و به کمک آن می‌توان از یک فایل خواند یا در آن نوشت.

هر پردازه یک فضای خصوصی برای نگهداری file descriptorهایش دارد که از 0 آغاز می‌شود. طبق قرارداد، مقدار 0 برای stdin[3]، مقدار 1 برای stdout[4] و مقدار 2 برای stderr[5] تعریف شده است. با توجه به اینکه

[1] Process Control Block (PCB)
[2] Struct
[3] Standard Input
[4] Standard Output
[5] Standard Error

file descriptor می‌تواند مربوط به یک file، یک device یا pipe باشد، سیستم‌عامل با استفاده از پیاده‌سازی file descriptorها به این شکل، توانسته است یک interface انتزاعی برای هرکدام از این موارد ایجاد کند و همه آنها را به یک شکل ببیند. (جهت آشنایی بیشتر با مفهوم file descriptor می‌توانید این فیلم را از یوتیوب مشاهده کنید.)

pipe یک kernel buffer کوچک است که برای ارتباط بین پردازه‌ها استفاده می‌شود. در واقع به کمک این عملگر می‌توانیم stdout یک پردازه را به stdin یک پردازه دیگر متصل کنیم.

عملکرد pipe در سیستم‌عامل xv6 به این صورت است که ابتدا به کمک تابع `pipe`، دو file descriptor که به هم متصل هستند ایجاد می‌کند. سپس برای پردازه سمت چپ ابتدا بخش قابل خواندن pipe را می‌بندد و سپس بخش قابل نوشتن آن را به‌عنوان stdout برای این پردازه قرار می‌دهد و دستور را اجرا می‌کند. برای پردازه سمت راست ابتدا بخش قابل نوشتن pipe را می‌بندد و سپس بخش قابل خواندن آن را به‌عنوان stdin در نظر می‌گیرد و درنهایت این دستور را هم اجرا می‌کند. سپس منتظر می‌ماند تا هر دو دستور خاتمه یابند. ممکن است دستور سمت راست pipe شامل دستوراتی باشد که در خود آنها نیز از pipe استفاده شده است. در این صورت درختی از دستورات اجرا می‌شوند. لازم به ذکر است که پردازه سمت راست تا زمانی که stdin آن به EOF[1] نرسیده باشد، منتظر داده جدید می‌ماند.

به pipe می‌توان به دید یک فایل مجازی نگاه کرد که خود به خود از حافظه پاک می‌شود و استفاده از آن راحت‌تر از ایجاد یک فایل است. (برای آشنایی بیشتر با عملکرد pipe می‌توانید این فیلم را از یوتیوب مشاهده کنید.)

۴. توابع fork و exec

تابع `fork` برای ایجاد یک process جدید استفاده می‌شود. درواقع این تابع یک نسخه کپی از پردازه‌ای می‌سازد که این تابع را صدا زده است. منظور از کپی این است که دیتا و دستورات پردازه فعلی در حافظه پردازه جدید (child) کپی می‌شوند. با وجود اینکه در لحظه ایجاد پردازه فرزند، داده‌های آن (متغیرها و رجیسترها) با پردازه پدر یکسان هستند، اما درواقع این دو پردازه حافظه جداگانه‌ای خواهند داشت و تغییر یک متغیر در پردازه پدر، آن متغیر در پردازه فرزند را تغییر نمی‌دهد. پردازه پدر پس از ایجاد پردازه فرزند، به caller تابع `fork` بازمی‌گردد که امکان اجرای همزمان دو پردازه را فراهم می‌سازد. مقدار return شده از تابع `fork` نیز pid پردازه فرزند خواهد بود. نقطه شروع پردازه فرزند نیز دقیقا همان caller تابع `fork` است، با این تفاوت که که مقدار خروجی این تابع عدد 0 خواهد بود.

پس اگر با استفاده از قطعه کد `int pid = fork();` یک پردازه جدید درست کنیم، یکی از حالت‌های زیر برای pid رخ می‌دهد:

[1] End Of File

- pid = 0: در پردازه فرزند هستیم.
- pid > 0: در پردازه پدر هستیم و مقدار pid درواقع شناسه پردازه فرزند است.
- pid < 0: در زمان اجرای تابع fork و پردازه جدید خطایی رخ داده و پردازه فرزند ایجاد نشده‌است.

اگر پس از fork کردن از تابع `wait()` استفاده شود، پردازه پدر منتظر پایان یافتن پردازه فرزند می‌شود و سپس کار خود را ادامه می‌دهد. خروجی این تابع، pid پردازه پایان یافته است. اگر پردازه فعلی هیچ پردازه فرزندی نداشته باشد، خروجی این تابع 1- خواهد بود.

قطعه کد زیر مثالی برای استفاده از تابع `fork` را نشان می‌دهد:

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

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

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

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

```
char* args[] = { "ls", "-l", "/home", NULL }; // NULL is required
exec("/bin/ls", args);
printf("Exec failed!\n");
```

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

۱. ابتدا دستور تایپ شده توسط کاربر در ترمینال را می‌خواند.
۲. با استفاده از تابع `fork` یک پردازه جدید ایجاد می‌کند.
۳. در پردازه فرزند با استفاده از تابع `exec` برنامه درخواست شده توسط کاربر را جایگزین پردازه فعلی (فرزند) می‌کند.
۴. در پردازه پدر برای اتمام کار پردازه فرزند `wait` می‌کند.
۵. پس از اتمام پردازه فرزند به main بازمی‌گردد و منتظر دستور جدید می‌شود.

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

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

```
char* args = { "cat", NULL };
int pid = fork();
if (pid == 0) {
	close(0); // close stdin
	open("in.txt", O_RDONLY); // open in.txt for reading (fd: 0)
	exec("/bin/cat", args);
	printf("Exec failed!\n");
}
else if (pid > 0) {
	wait();
	printf("Child process has exited.\n");
}
else {
	printf("The fork failed!\n");
}
```

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

اضافه کردن یک متن به Boot Message

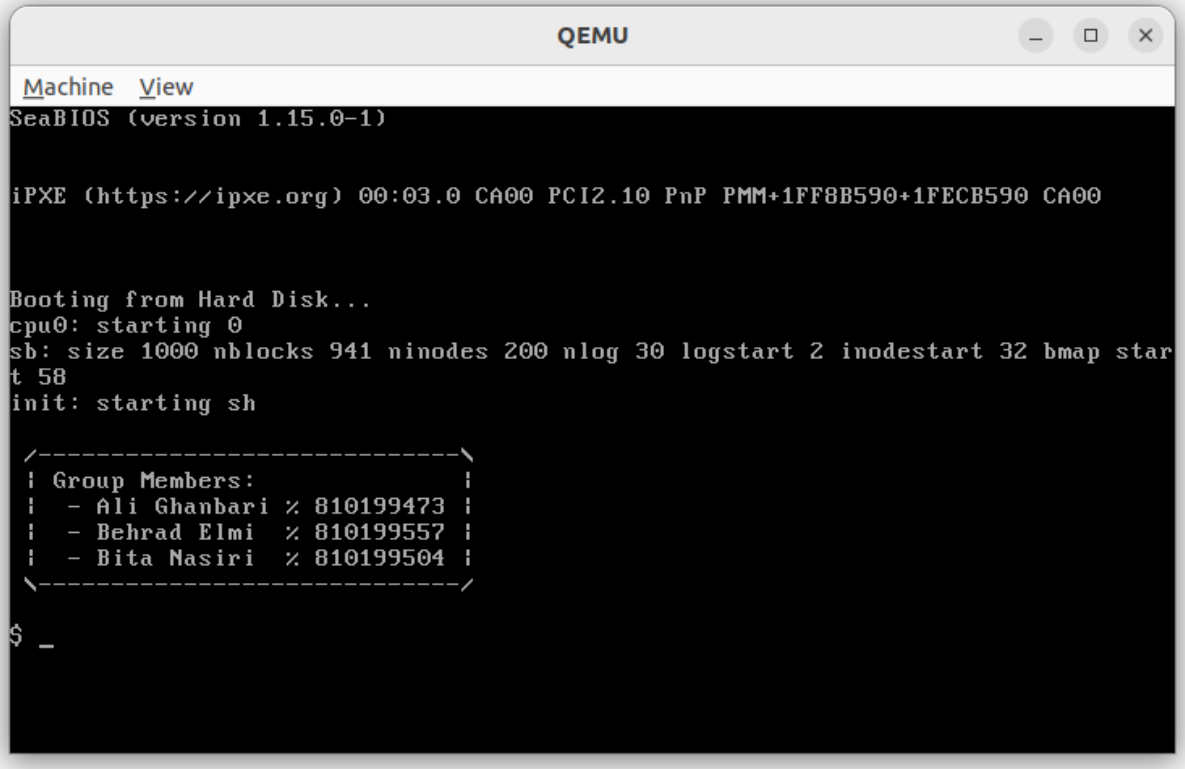
این کار با افزودن `printf` در فایل init.c میسر می‌شود:

init.c

```
printf(1, "\n");
printf(1, " /----------------------------\\\n");
printf(1, " | Group Members:              |\n");
printf(1, " |  - Ali Ghanbari %% 810199473 |\n");
printf(1, " |  - Behrad Elmi  %% 810199557 |\n");
printf(1, " |  - Bita Nasiri  %% 810199504 |\n");
printf(1, " \\----------------------------/\n");
printf(1, "\n");
```

لازم به ذکر است که مقدار آرگومان اول `printf` را 1 قرار دادیم، زیرا می‌خواهیم بر stdout بنویسیم.

پس از بوت شدن سیستم‌عامل نام اعضای گروه بدین صورت نمایش داده می‌شود:



اضافه کردن چند قابلیت به کنسول xv6

برای افزودن قابلیت به کنسول xv6 باید در فایل console.c تغییراتی اعمال کنیم.

در ابتدا مقداری refactor و تغییر روی کد اصلی انجام می‌دهیم تا پیاده‌سازی قابلیت‌های خواسته شده، ساده‌تر گردد.

برای آن‌که قابلیت‌های `struct input` بیشتر شود و خوانایی کد افزایش یابد، یک فیلد `len` به آن می‌افزاییم که طول بافر ورودی را نگه دارد و با افزودن یا حذف کاراکتر از بافر نیز مقدار آن را در ادامه برنامه آپدیت می‌کنیم:

console.c

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

تابع `get_cursor_position` را برای دسترسی به موقعیت کنونی cursor تعریف می‌کنیم:

console.c

```
static int
get_cursor_position()
{
    // Cursor position: col + 80*row.
    outb(CRTPORT, 14);
    int pos = inb(CRTPORT + 1) << 8;
    outb(CRTPORT, 15);
    pos |= inb(CRTPORT + 1);
    return pos;
}
```

برای تغییر موقعیت cursor، تابع `change_cursor_position` را به این شکل تعریف و استفاده می‌کنیم:

console.c
```
static void
change_cursor_position(int pos)
{
    outb(CRTPORT, 14);
    outb(CRTPORT + 1, pos >> 8);
    outb(CRTPORT, 15);
    outb(CRTPORT + 1, pos);
}
```

حال سراغ پیاده‌سازی قابلیت‌های خواسته شده می‌رویم:

۱ و ۲. دستورهای] + shift و [+ shift برای رفتن به ابتدا و انتهای خط

تمایز اینکه در حال نوشتن بر ابتدای خط هستیم یا انتهای خط را با متغیر `cursor_mode` مشخص کرده و آن را به صورت global تعریف می‌کنیم (حالت default، نوشتن بر انتهای خط در نظر گرفته شده است.):

console.c
```
#define CURSOR_RIGHT_MODE 0
#define CURSOR_LEFT_MODE  1
static ushort cursor_mode = CURSOR_RIGHT_MODE;
```

درصورت جابجایی نشانه‌گر از انتها به ابتدای خط (یا برعکس) متغیر `cursor_mode` آپدیت شده و برنامه عملکرد متفاوتی تحت اثر این مسئله از خود نشان می‌دهد.

کلید میانبر shift + x را اینگونه تعریف می‌کنیم:

console.c
```
#define S(x)  ((x) + ' ')  // Shift-x
```

دقت کنید کاراکتر space با استفاده از کد ascii بدست آمده است. (فشردن کلید shift به همراه یک کاراکتر، کاراکتر جدیدی می‌سازد که با کاراکتر قبلی 32 واحد اختلاف دارد و 32 معادل کد ascii کاراکتر space است.)

با صدا کردن تابع زیر در switch case تابع `consoleintr` به ازای حالت‌های `S('[')` و `S(']')`، می‌توانیم این دو دستور را به ترمینال اضافه کنیم:

console.c

```
static void
set_cursor(ushort mode)
{
    if (mode == CURSOR_RIGHT_MODE) {
        switch_end();
        cursor_mode = CURSOR_RIGHT_MODE;
        input.e = input.len;
    }
    else if (mode == CURSOR_LEFT_MODE) {
        switch_begin();
        cursor_mode = CURSOR_LEFT_MODE;
        input.e = input.w;
    }
}
```

همان‌طور که مشاهده می‌کنید به ازای modeهای مختلف، موقعیت cursor را به انتها یا ابتدای خط منتقل کرده و متغیرهای `cursor_mode` و `input.e` (محل کنونی cursor در کنسول) را آپدیت می‌کنیم.

توابع کمکی `switch_begin` و `switch_end` اینگونه تعریف می‌شوند:

console.c

```
static void
switch_begin()
{
    int pos = get_cursor_position();
    if (crt[pos - 2] == ('$' | 0x0700)) // is cursor at first of line?
        return;
    int change = pos % 80 - 2;
    input.e -= change;
    change_cursor_position(pos - change);
}
```

اگر عبارت شرطی `crt[pos-2] == ('$'|0x0700)`، true برگرداند، به این مفهوم است که cursor در ابتدای خط قرار دارد و نیاز مجدد به جابجایی آن در راستای اجرای دستور `shift + [` وجود ندارد.

console.c

```
static void
switch_end()
{
    int pos = get_cursor_position();
    int change = pos % 80 - 2 - input.len;
    change_cursor_position(pos - change);
}
```

از آنجا که برای تایپ کاراکترها در ترمینال حالت‌بندی کرده‌ایم و بین دو حالت `cursor_mode` تمایز قائل شده‌ایم، نیاز است در default ء switch case یاد شده واقع در تابع `consoleintr` تغییراتی اعمال کنیم تا بافر ورودی در حالت نوشتن بر ابتدای خط، متناظر با نمایشگر کنسول آپدیت شود. نیز بعد از تایپ هر کاراکتر ورودی (به جز Backspace) بایستی مقدار متغیر `input.len` را یک واحد افزایش دهیم.

console.c

```
default:
    if (c != 0 && input.e - input.r < INPUT_BUF) {
        c = (c == '\r') ? '\n' : c;
        input.len++;
        if (cursor_mode == CURSOR_LEFT_MODE)
            shift_right_buffer();
        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);
        }
    }
    break;
```

Changes

تابع `shift_right_buffer`، کاراکترهای بافر را از `input.e` تا `input.len` یک واحد به راست شیفت می‌دهد تا جا برای کاراکتر جدید وارد شده در حالت `CURSOR_LEFT_MODE` باز شود.

لازم به ذکر است اگر در حالت `CURSOR_LEFT_MODE` باشیم، درصورت تایپ کاراکتر Backspace دچار مشکل می‌شویم، زیرا به‌جای افزودن کاراکتر در یک index خاص، بایستی یک کاراکتر از بافر حذف کنیم. (که

با یک واحد شیفت دادن بافر به سمت چپ میسر می‌شود.) همچنین باید حواسمان باشد که از متغیرهای `input.len` و `input.e` یک واحد بکاهیم تا آپدیت باقی بمانند.

برای هندل کردن موارد گفته‌شده تابعی تحت عنوان `backspace` تعریف کردیم.

console.c

```
static void
backspace()
{
    int pos = get_cursor_position();
    if (crt[pos - 2] != ('$' | 0x0700) &&
        input.buf[(input.e - 1) % INPUT_BUF] != '\n') {
        if (cursor_mode == CURSOR_LEFT_MODE)
            shift_left_buffer(input.e);
        input.e--;
        input.len--;
        consputc(BACKSPACE);
    }
}
```

همانطور که اشاره شد، در حالت `CURSOR_LEFT_MODE` نیاز به شیفت دادن بافر به سمت چپ وجود داشت. لذا تابعی با نام `shift_left_buffer` تعریف کردیم که از index ورودی به آن (`input.e` یا همان موقیت کنونی cursor در کنسول) تا `input.len` را یک واحد به سمت چپ شیفت می‌دهد.

یکی از قابلیت‌هایی که به‌صورت پیش‌فرض در کنسول xv6 وجود داشت، دستور ctrl + u است. عملکرد آن به این صورت است که پس از فشردن کلید، کل خط ورودی در کنسول را پاک می‌کند. برای اجرای صحیح این دستور و نیز همخوانی آن با قابلیت‌هایی که ما به کنسول xv6 افزودیم، بایستی تغییراتی در آن بوجود آوریم. تابع `kill_line` را به همین منظور تعریف می‌کنیم. به این صورت که ابتدا به حالت default ورودی دادن به کنسول یا همان `CURSOR_RIGHT_MODE` سوییچ می‌کنیم، سپس تابع `backspace` را آنقدر صدا می‌زنیم تا به نقطه شروع خط برسیم و بعد return می‌کنیم.

console.c

```
static void
kill_line()
{
    set_cursor(CURSOR_RIGHT_MODE);
    while (1) {
        int pos = get_cursor_position();
        if (crt[pos - 2] == ('$' | 0x0700) ||
            input.buf[(input.e - 1) % INPUT_BUF] == '\n') {
            break;
        }
        backspace();
    }
}
```

۳. دستور ctrl + w برای پاک کردن کلمه قبل از نشانه‌گر

پیاده‌سازی این قابلیت در قالب دو حلقه while صورت گرفته است. بدین‌صورت که در حلقه while اول از موقعیت cursor تا زمانی‌که کاراکتر space وجود داشته باشد، تابع backspace (که خود تعریف کردیم) را فرا می‌خوانیم. پس انتظار داریم بعد از اجرای حلقه while اول، spaceهای قبل cursor پاک شده باشد و cursor به کاراکتری غیر از space اشاره کند. عملکرد حلقه while دوم بدین صورت است که در آن آنقدر backspace را فرا می‌خوانیم تا به space یا نقطه شروع خط برسیم. (یعنی کلمه قبل از cursor را پاک کرده و space را به‌عنوان seperator بین دو کلمه می‌بینیم.)

دقت کنید متغیرهای input.e و input.len در داخل تابع backspace پس از هر بار فراخوانی آپدیت می‌شوند و در طول برنامه و توابع دیگری که تعریف کردیم نیز این موضوع همواره مورد بررسی است.

همانند قابلیت‌های قبلی، با صدا کردن تابع زیر در switch case تابع consoleintr به ازای حالت C('W')، می‌توان این قابلیت را به ترمینال اضافه کرد:

console.c

```
static void
delete_last_word()
{
    int pos = get_cursor_position();
    if (crt[pos - 2] == ('$' | 0x0700) || // is cursor at first of line?
        input.buf[(input.e - 1) % INPUT_BUF] == '\n') {
        return;
    }

    // remove spaces after last word
    while (input.buf[(input.e - 1) % INPUT_BUF] == ' ')
        backspace();

    // remove last word
    while (1) {
        int pos = get_cursor_position();
        if (crt[pos - 2] == ('$' | 0x0700) ||
            input.buf[(input.e - 1) % INPUT_BUF] == ' ') {
            break;
        }
        backspace();
    }
}
```

اجرا و پیاده‌سازی یک برنامه سطح کاربر

در ابتدا باید کد برنامه را بزنیم. الگوریتم که مشخص است، اما از آنجا که کد را برای سیستم‌عامل می‌نویسیم، باید نکاتی را در نظر بگیریم. به عنوان مثال ممکن است برخی از توابع استاندارد زبان C را در اختیار نداشته باشیم و یا بعضا header fileی با نام مشابه وجود داشته باشد. باید برنامه را با توجه به این موارد توسعه بدهیم.

تابع `printf` در اینجا عملکردی مشابه با تابع شناخته‌شده در C دارد، با این تفاوت که یک آرگومان دیگر به عنوان شناسه محلی نیز می‌گیرد. منظور از شناسه محلی، اندیس پردازه است که در کدام آدرس (stdout یا stderr) بنویسد. همچنین این شناسه نباید لزوما به خروجی‌های ترمینال منتهی شود و می‌تواند مربوط به یک فایل باشد و در فایل عملیات نوشتن را انجام دهد. در کد تابع `printf` می‌توان مشاهده کرد که در مراحل پایین‌تر، فراخوانی سیستمی `write` را صدا می‌زند و فقط رابط بهتری را در اختیارمان قرار می‌دهد.

تابع `atoi` بصورت پیش‌فرض وجود دارد و برای ما قابل استفاده است، اما تابع با قابلیت مشابه تحت عنوان `custom_atoi` تعریف کردیم که بتوانیم خطاهای برنامه را بهتر کنترل کنیم و نیز کاراکترهای غیر عددی را تشخیص دهیم:

mmm.c

```
int
custom_atoi(const char* str)
{
    int sign = 1;
    int unsigned_num = 0;

    if (*str == '-') {
        sign = -1;
        ++str;
    }
    else if (*str == '+')
        ++str;

    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] >= '0' && str[i] <= '9')
            unsigned_num = unsigned_num * 10 + (str[i] - '0');
        else {
            printf(2, "Error: Wrong number format.\n");
            exit();
        }
    }
    return sign * unsigned_num;
}
```

کد کامل این بخش در فایل mmm.c قابل مشاهده است.

قبل از کامپایل و اجرای سیستم‌عامل، برای اینکه برنامه ما در سطح کاربر، در سیستم‌عامل در دسترس قرار گیرد، بایستی برنامه را به متغیر UPROGS در Makefile با فرمت _sourceName بیفزاییم. قسمت UPROGS برنامه را در فایل‌سیستم قرار می‌دهد (که بعدا به qemu داده می‌شود.) تا برنامه بارگذاری شده و در دسترس قرار گیرد و کاربر بتواند آن دستورها را اجرا نماید.

Makefile

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

حال کافیست سیستم‌عامل را کامپایل کرده و در qemu اجرا کنیم. برنامه ما آماده اجرا شدن می‌باشد:

```
$ mmm
usage: mmm <array_of_numbers>
$ mmm 1 2 c
Error: Wrong number format.
$ mmm 1 2 3 4 5 6 7 8
Error: Please ensure that you enter no more than 7 numbers.
$ mmm 8 2 8 4 2 3
$ cat mmm_result.txt
4 3 2
$
```

همان‌طور که مشاهده می‌کنید، توانستیم errorهای مختلف را در پیاده‌سازی‌مان هندل کنیم. در نهایت پس از دادن ورودی با فرمت مورد نظر، خروجی در فایل `mmm_result.txt` ریخته می‌شود که با دستور `cat` می‌توانیم محتوای آن را مشاهده کنیم. (دقت کنید دستور `cat` به‌صورت default در xv6 تعریف شده‌است.)

مقدمه‌ای درباره سیستم‌عامل xv6

۵. سه وظیفه اصلی سیستم‌عامل

۱. واسطی است میان سخت‌افزار و نرم‌افزارها (کاربران)
۲. مدیریت منابع (لایه‌های زیرین)
۳. مدیریت کاربران و برنامه‌های کاربردی (لایه‌های بالایی)

۶. بخش‌های سیستم‌عامل xv6

بخش‌های سیستم‌عامل xv6 به شرح زیراند:

- **Basic Headers:**

تعاریف متغیرها و defineهای اولیه و header fileها و بعضی تعریف typeها، همینطور بعضی توابع اولیه برای تبدیل assembly به C در این بخش قرار دارند.

- types.h: شامل typedefهای مورد نیاز
- param.h و memlayout.h و asm.h: حاوی define بعضی مقادیر ثابت
- defs.h: تعریف چند struct و توابع
- x86.h: توابعی برای استفاده از دستورات assembly در معماری x86
- mmu.h: بعضی structها، مقادیر define شده برای مدیریت حافظه
- elf.h و date.h

- **Entering xv6:**

اجرای اولیه سیستم‌عامل که ابتدا بیشتر به زبان assembly در entry.S است و فراهم کردن امکانات لازم نظیر allocate کردن حافظه‌های اولیه در main.c توسط این بخش هندل می‌شوند.

- main.c: نقطه شروع سیستم است و سیستم از اینجا شروع به اجرا می‌کند.
- entry.S: kernel از اینجا شروع به کار می‌کند و دستورات assembly این بخش برنامه را به بخش اجرای کد C منتقل می‌کنند.
- entryother.c

- **Locks:**

مدیریت همروندی با گذاشتن قفل روی نوشتن یا خواندن از فایل هنگامی‌که همروندی ممکن است مشکل‌زا باشد.

پیاده‌سازی این بخش در دو فایل spinlock.h و spinlock.c موجود است. امکاناتی برای گرفتن و رها کردن lock در این بخش درنظر گرفته شده‌است. (توابع acquire و release)

- **Processes:**

این بخش وظیفه اختصاص‌دادن حافظه فیزیکی به پردازه‌ها، مدیریت پردازه‌ها و قابلیت context switching را بر عهده دارد.

- swtch.S: در این بخش قابلیت context switching پیاده‌سازی شده است به این صورت که وضعیت فعلی registerها ذخیره می‌شوند تا دوباره بعدا برای اجرا بتوانند بازیابی شوند.
- proc.c و proc.h: قابلیت‌های مربوط به ایجاد و مدیریت پردازه‌ها. پیاده‌سازی `fork` در این بخش انجام شده است.
- vm.c
- kalloc.c: در این بخش پیاده‌سازی نحوه اختصاص یافتن حافظه فیزیکی به پردازه‌ها انجام شده است.

- **System Calls:**

در این بخش trapها و system callها تعریف شده‌اند تا بتوان از آنها استفاده کرد.

- traps.c و traps.h: انواع trapها و عدد متناظر آنها تعریف شده‌اند. همچنین توابع مربوط به trapها نیز در این بخش پیاده‌سازی شده‌اند.
- syscall.c و syscall.h: عدد متناظر با system callها و توابع مرتبط پیاده‌سازی شده‌اند.

- **File System:**

هدف یک فایل‌سیستم، organize کردن و ذخیره‌کردن داده‌هاست. معمولاً file systemها به اشتراک‌گذاری داده‌ها را میان userها و applicationها پشتیبانی می‌کنند. فایل‌سیستم xv6 از شش لایه تشکیل می‌شود:

- پایینی ترین لایه از طریق buffer cache بلوک‌هایی را از روی IDE Disk می‌خواند و می‌نویسد که تضمین می‌کند حداکثر یک kernel process در هر لحظه می‌تواند داده فایل-سیستمی ذخیره‌شده در یک block را تغییر دهد.
- لایه دوم به لایه‌های بالاتر اجازه می‌دهد که آپدیت‌هایی را بر blockهای بسیاری در یک transaction انجام دهد تا تضمین کند همه blockها اتوماتیک آپدیت می‌شوند.
- لایه سوم فایل‌های بی‌نام provide می‌کند که هرکدام با یک inode و دنباله‌ای از blockها شامل داده‌های فایل نمایش داده می‌شوند.

- لایه چهارم directoryها را به‌عنوان inode خاص که محتویاتش دنباله‌ای از entryهای directory است که هرکدام یک اسم و reference به inode است.
- لایه پنجم سلسله مراتب path nameها (مثل usr/rtm/xv6/fs.c) را با استفاده از ساختاری بازگشتی تامین می‌کند.
- لایه آخر خیلی از منابع unix (مثل files، devices، pipes و ...) را به کمک فایل systeminterface انتزاع‌سازی می‌کند و کار را برای application programmerها ساده‌تر می‌کند.

بعضی از فایل‌های آن به نام‌های زیر هستند که اعمال بالا را مدیریت می‌کنند:

- fs.c: روتین‌های low level مربوط به file system را داراست.
- log.c: حداکثر یک transaction در لحظه را مدیریت می‌کند.

- **Pipes:**

در این بخش `struct pipe` تعریف شده است و توابعی برای عملیات خواندن و نوشتن برای آن پیاده‌سازی شده است. به‌طور کلی pipe برای این استفاده می‌شود که پردازه‌ها بتوانند بر روی pipe بنویسند یا از آن بخوانند و بتوانند با هم ارتباط برقرار کنند.

- **String Operations:**

توابع کمکی لازم برای کار کردن با string.

- **Low-level Hardware:**

شامل کدهای سطح پایین برای ارتباط با کیبورد و cpuها و هندل کردن interruptهای سخت‌افزاری و IO[1] در کنسول است.

- mp.h: تعاریف مربوط به فایل mp.c
- mp.c: پیاده‌سازی پشتیبانی multi processor
- lapic.c: مدیریت interruptهای داخلی (غیر I/O)
- ioapic.c: مدیریت interruptهای سخت‌افزار برای یک سیستم SMP
- kbd.h و kbd.c: تعریف ثابت‌های دکمه‌های کیبورد
- console.c: کدهای ورودی و خروجی. ورودی از طریق کیبورد یا سریال پورت است و خروجی در صفحه و سریال پورت نوشته می‌شود.
- Uart.c: سریال پورت intel 8250

[1] Input/Output

- **User-level:**

در این بخش، اولین برنامه سطح کاربر اجرا می‌شود و امکاناتی نظیر shell اجرایی می‌شوند.

- initcode.S: کدهای asm برای اجرای برنامه سطح کاربر init.
- usys.S: تعریف system callها در سطح کاربر.
- init.c: فایل اولیه‌ای که سمت کاربر اجرا می‌شود.
- sh.c: توابع و تعریف‌ها برای اجرای دستورات در shell.

- **Boot Loader:**

این بخش عملیات‌های لازم را برای boot شدن سیستم را انجام می‌دهد.

- bootasm.S: کد assembly برای load شدن کد BIOS از اولین sector حافظه و منتقل‌کردن اجرا به کد C.
- bootmain.c: توابع مرتبط برای عملیات‌های boot.

- **Link:**

یک linker script برای JOS kernel است.

نام پوشه اصلی هسته لینوکس، kernel است و کدهای اصلی هسته در آن موجود است.
فایل‌های هسته در لینوکس: https://github.com/torvalds/linux/tree/master/kernel

فایل‌های header لینوکس در پوشه /usr/src هستند و مشابه header ها در زبان C، هدفشان تعریف یک رابط کاربری یا interface بین kernel space وuser space ، همین‌طور بین اجزای داخلی هسته است.
فایل‌های سرایند[1] در لینوکس: https://github.com/torvalds/linux/tree/master/include

فایل‌سیستم مجموعه‌ای logical از فایل‌های روی یک partition است و ساختاری درختی دارد. فایل‌های فایل‌سیستم در سیستم‌عامل لینوکس از root اصلی یا همان / شروع می‌شوند.
فایل‌های فایل‌سیستم در لینوکس: https://github.com/torvalds/linux/tree/master/fs

[1] Header Files

کامپایل سیستم‌عامل xv6

۷. اجرای دستور make -n و کدام دستور فایل نهایی را می‌سازد؟

۸. متغیرهای UPROGS و ULIB در Makefile

متغیر UPROGS:

UPROGS معادل User Programs است و یک لیست از برنامه‌های کاربر دارد که در هنگام ساخت و کامپایل xv6، این برنامه‌ها نیز کامپایل و تبدیل به فایل‌های قابل اجرا توسط سیستم‌عامل می‌شوند و می‌توان آنها را در shell فراخوانی کرد. نام هر یک از این برنامه‌ها به‌صورت _fileName (اسامی که یک underscore ابتدایشان دارند.) در این لیست قرار گرفته است. تمام اسامی به این فرمت یک هدف[1] با پیشنیاز[2]های فایل آبجکت هدف (fileName.o) و متغیر ULIB دارد. بنابراین هدف‌های موجود در UPROGS منجر به ساخت فایل آبجکت برنامه‌های کاربر، اجرا شدن هدف‌های مربوط به ULIB و درنهایت اجرای دستور ld می‌شود. دستور ld برای پیوند[3] فایل‌های مورد نیاز و تولید یک فایل قابل اجرا مورد استفاده قرار می‌گیرد. علاوه‌بر آن فایل‌های آبجکت مربوط به هر برنامه (fileName.o) توسط یک قانون درونی[4] Makefile ساخته می‌شوند و به‌صورت صریح در Makefile نوشته نشده‌اند.

متغیر ULIB:

ULIB معادل User Libraries است و شامل تعدادی از کتابخانه‌های زبان C می‌باشد. در بسیاری از کدهای xv6 توابع این کتابخانه‌ها استفاده شده‌اند و برای اجرایشان به کامپایل این فایل‌ها نیاز داریم، برای مثال برنامه‌های سطح کاربر نیازمند کامپایل فایل‌های ULIB می‌باشند. بنابراین همانطور که در بخش قبل نیز گفته شد، فایل‌های ULIB به‌عنوان پیشنیاز در قوانین قرار گرفته‌اند و درنهایت توسط دستور ld به فایل‌های اجرایی پیوند می‌شوند. ULIB عبارت است از فایل‌های ulib.o، usys.o، printf.o و umalloc.o. فایل‌های ULIB توابعی مانند printf، strcmp، strcpy و malloc را شامل می‌شوند.

اجرا بر روی شبیه‌ساز QEMU

۹. محتوای دو دیسک ورودی QEMU

[1] Target
[2] Prerequisite
[3] Link
[4] Built-in implicit rule

مراحل بوت سیستم‌عامل xv6

اجرای بوت‌لودر

۱۰. محتوای سکتور نخست دیسک قابل بوت

اولین کامندهای اجرا شونده توسط Makefile شامل کامپایل کردن object fileهای bootmain.c و bootasm.S، پیوند زدن این دو و تولید bootblock.o، objcopy کردن بخش text. فایل bootblock.o به فایل bootblock و درنهایت داده شدن به اسکریپت sign.pl برای اضافه کردن 2 بایت boot signature به bootblock است.

در سکتور نخست (512 بایت اول) دیسک قابل بوت، محتوای فایل bootblock قرار دارد.

۱۱. مقایسه فایل باینری بوت با بقیه فایل‌های باینری xv6 و تبدیل آن به اسمبلی

همه فایل‌های باینری آبجکت xv6 در فرمت ELF هستند. این فرمت مخفف Executable and Linkable Format یا Executable Linkable Format می‌باشد که فرمتی استاندارد برای فایل‌های executable و object codeها و shared libraryها و core dumpها می‌باشد که برای اولین بار، برای مشخص کردن رابط باینری (ABI[1]) در سیستم‌عامل یونیکس استفاده شد. فرمت ELF انعطاف پذیر و توسعه پذیر و cross platform می‌باشد. انعطاف پذیری به این معنا که از چندین نوع cpu، معماری ماشین و سیستم‌عامل پشتیبانی می‌کند و توسعه پذیری به این معنا که هر فایل بسته به قسمت‌های مورد نیاز به‌طور متفاوتی ساخته می‌شود که این می‌تواند همه آنها را هندل کند. برای مثال endiannessهای (ترتیب و توالی بایت‌های یک کلمه از داده‌های دیجیتال در حافظه کامپیوتر) مختلف و اندازه آدرس‌های مختلفی را نیز پشتیبانی می‌کند. این موارد باعث شده آن را توسط بسیاری از سیستم‌عامل‌های مختلف بر روی پلتفرم سخت افزاری مختلف مورد استفاده قرار دهند.

این فرمت باینری از بخش‌های مختلفی تشکیل شده است. در ابتدای آن هدرهایی شامل اطلاعات لود شدن فایل نوشته شده است و سپس چند section دارد که هر کدام حجمی از کد یا داده‌اند که در آدرس مشخصی از حافظه لود می‌شوند.

فرمت فایل ELF برای انواع object fileها یعنی relocatable (فایل‌های o. که توسط linker استفاده می‌شوند)، executable و shared objectها تعریف شده است.

دو هدر ELF Header و Program Header در فایل elf.h به زبان C تعریف شده‌اند.

[1] Application Binary Interface

در ELF Header بخشی به نام e_entry وجود دارد که آدرس نقطه ورود برنامه را مشخص می‌کند.
از sectionهای ELF می‌توان به موارد زیر اشاره کرد:

- `.text` شامل دستورات قابل اجرای برنامه است با دسترسی به قابلیت خواندن و اجرا و فقط یک بار بارگذاری می‌شود زیرا محتویات تغییر نمی‌کند.
- `.rodata` حاوی داده‌های read-only از جمله string literalها در زبان C است.
- `.data` شامل داده‌های مقداردهی شده با دسترسی خواندن و نوشتن مانند برخی متغیرهای گلوبال است.
- `.bss` شامل داده‌های مقداردهی نشده است که چون داده‌ای وجود ندارد، فقط آدرس و اندازه‌اش در فایل ذخیره می‌شود.

با استفاده از دستور `objdump -h bootblock.o` می‌توانیم نوع فایل باینری (که مانند بقیه فایل‌های باینری xv6 به فرمت elf32-i386 است) و در ادامه خروجی دستور، sectionهای ELF را مشاهده کنیم.
بوت‌لودر پس از لود شدن در آدرس ثابت 0x7C00 توسط پردازنده اجرا می‌شود تا کرنل را اجرا کند. در اینجا تنها اطلاعات مهم، کدی است که قرار است اجرا بشود. با مقایسه bootblock.o با بقیه object fileها می‌بینیم که بخش‌های data. و غیره را ندارد و بخش اصلی‌اش فقط text. است.
از آنجا که bootblock.o در آدرس خاصی شروع به اجرا شدن می‌کند، در هنگام ساخته شدنش از فلگ `-Ttext 0x7C00` استفاده شده است که آدرس بخش text. فایل خروجی را مشخص می‌کند. فلگ -e start هم می‌گوید که نقطه شروع برنامه لیبل start در bootasm.S است.
خود فایلی که در سکتور بوت قرار دارد یعنی bootblock با استفاده از دستورِ

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

(و اضافه کردن boot signature) تولید می‌شود. این فلگ‌های objcopy در بخش بعدی توضیح داده شده‌اند.
این دستور محتویات بخش text. را به‌صورت raw binary به فایل bootblock می‌ریزد. این یعنی فایل bootblock از فرمت ELF پیروی نمی‌کند و هیچ هدری هم ندارد. این فایل با دیگر فایل‌های باینری xv6 تفاوت دارد و کد قابل اجرای خالص بدون هیچ اطلاعات اضافه‌ای است.
پس نوع فایل دودویی مربوط به بوت raw binary است (که در حالت کلی چیز مشخصی نیست) و اینجا همان محتویات بخش text. (instructionهای قابل اجرا بر روی معماری x86) می‌باشد. یعنی با بقیه فایل‌های باینری از آنجا که به فرمت ELF نیست، تفاوت دارد.
دلیل استفاده نکردن از ELF برای bootblock این است که فرمت ELF را هسته سیستم‌عامل می‌داند و نه cpu. پس وقتی‌که هسته هنوز اجرا نشده، نمی‌توان فرمت ELF را خواند. اگر BIOS فایل bootblock.o را

برای بوت شدن به cpu می‌داد، از آنجا که cpu هدرهای ELF را نمی‌شناسد، همه محتوای فایل را به دید instructionها نگاه کرده و برداشت اشتباهی از آن می‌کند. پس باید فقط دستورات خالص را به cpu داد.
یک دلیل دیگر هم کم کردن حجم فایل است. با استخراج بخش .text فایل bootblock.o، حجم آن کاهش یافته و در 512 بایت جا می‌گیرد.
برای تبدیل bootblock به اسمبلی، از دستور زیر استفاده می‌کنیم:

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

از آنجا که bootblock باینری خام است و هیچ هدری برای مشخص کردن معماری‌اش ندارد، آنها را باید دستی به objdump بدهیم. فلگ‌هایی که استفاده شده:

- `-D`: برای disassemble کردن باینری.
- `-b binary`: نوع فایل را raw binary در نظر می‌گیریم.
- `-m i386`: معماری اسمبلی فایل را مشخص می‌کنیم.
- `-M addr16,data16`: از آنجا که وقتی BIOS سکتور بوت را لود می‌کند در real mode هستیم و cpu در حالت 16 بیت است، اسمبلی 16 بیت نیز استفاده شده است. پس هنگام disassemble کردن هم می‌گوییم که آدرس‌ها و داده‌ها را 16 بیت در نظر گیرد. (در 16 بیت از رجیستر ax، cx و مشابه آن استفاده شده‌است که در bootasm.S نیز همین‌گونه است، اما اگر این قسمت را قرار نمی‌دادیم آن را 32 بیت درنظر می‌گرفت و رجیسترها به‌صورت eax و ecx (extended cx) درنظر گرفته می‌شدند که مشابه کد مورد نظر ما نبود.)

می‌توانیم با استفاده از فلگ `--adjust-vma=0x7C00` آدرس شروع قرار گرفتن اسمبلی خروجی در حافظه را تغییر بدهیم که مانند واقعیت از آدرس 0x7C00 شروع بشود.
با مشاهده خروجی دستورِ

```
objdump -D -b binary -m i386 -M addr16,data16 --adjust-vma=0x7C00 bootblock
```

می‌بینیم که ابتدای آن بسیار مشابه با 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    $0xd88e0010,%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
    7c4b:       00 00                   add    %al,(%bx,%si)
    7c4d:       66 b8 00 8a 66 89       mov    $0x89668a00,%eax
```

۱۲. علت استفاده از objcopy در هنگام make

این دستور همان‌طور که از نامش پیداست، میتواند محتویات یک فایل object را در یک فایل object دیگر کپی کند. برای این کار نیازی نیست فرمت فایل ورودی با فرمت فایل مقصد یکسان باشد. با توجه به این‌که این برنامه کار ترجمه فایل را با استفاده از کتابخانه GNU BFD انجام می‌دهد، تمامی فرمت‌های موجود در این کتابخانه پشتیبانی می‌شوند و امکان تبدیل بین آن‌ها وجود دارد. این‌که objcopy دقیقا چه کاری انجام می‌دهد توسط کاربر و دستوری که در ترمینال می‌نویسد مشخص می‌شود، اما به‌طور کلی objcopy فایل‌های موقتی

تشکیل می‌دهد تا بتواند ترجمه‌هایش را انجام دهد و سپس آن‌ها را پاک می‌کند. آپشن‌هایی از این دستور که در Makefile مربوط به xv6 استفاده شده‌اند به‌طور خلاصه در بخش زیر توضیح داده شده‌است:

- `-S` : در صورت استفاده از این آپشن، اطلاعات مربوط به symbol table و relocation records در فایل مقصد حذف می‌شوند. داده‌های symbol table نام و مکان متغیرها و فرایندهایی را ذخیره می‌کنند که ممکن است در فایل‌های object دیگر از آن‌ها استفاده شده باشد. داده‌های relocation records نیز اطلاعاتی در مورد آدرس‌هایی از فایل object ذخیره می‌کند که در هنگام ساخت فایل مشخص نبوده و نیاز است در ادامه توسط linker مقداردهی شوند. این آدرس‌ها می‌توانند مربوط به متغیرها و توابعی باشند که در فایل‌های دیگر تعریف شده‌اند و در خود فایل وجود ندارند. در این حالت linker در زمان لینک کردن فایل‌ها، این آدرس‌ها را مقداردهی می‌کند.
- `-O` : این آپشن نوع فرمت فایل مقصد را نشان می‌دهد. برای مثال با استفاده از آپشن `-O binary` فایل تولید شده از نوع raw binary خواهد بود. این نوع فایل‌ها به فرمت خاصی نوشته نشده‌اند. از جمله این فایل‌ها می‌توان به فایل‌های memory dump اشاره کرد.
- `-j` : با استفاده از این آپشن می‌توانیم تنها بخشی از فایل object را به فایل جدید کپی کنیم.

در Makefile در چند بخش از دستور objcopy استفاده شده است:

۱. در bootblock پس از لینک شدن bootmain.o و bootasm.o در فایلی به نام bootblock.o، محتویات بخش .text این فایل را در یک فایل raw binary به نام bootblock کپی می‌کند. سپس این فایل را به اسکریپت sign.pl می‌دهد که ابتدا سایز فایل را بررسی می‌کند که بیشتر از 510 بایت نباشد و سپس 2 بایت 0x55 و 0xAA که boot signature اند را به انتهای فایل اضافه می‌کند.
۲. در entryother محتویات بخش .text فایل bootblockother.o را در یک فایل raw binary به نام entryother کپی می‌کند.
۳. در initcode محتویات فایل initcode.out در یک فایل raw binary به نام initcode کپی می‌شود.

در نهایت با لینک شدن فایل‌های entry.o و فایل‌های object که در متغیر OBJS تعریف شده‌اند و فایل‌های باینری initcode و entryother که پیش‌تر با استفاده از دستور objcopy ساخته شدند، فایل kernel ساخته می‌شود.

۱۳. چرا برای بوت کردن فقط از فایل C استفاده نشده و اسمبلی هم هست؟

اجرای کدهای زبان اسمبلی معمولا سریع‌تر و کم حجم‌تر است. هم‌چنین گاهی نیاز است که یک دسترسی سطح سیستم داشته باشیم که در آن صورت کد زبان C به تنهایی کافی نیست.
یک نمونه از این کارها وارد شدن به protected mode است.
وقتی که BIOS کد سکتور بوت را لود می‌کند، پردازنده x86 در real mode اجرا می‌شود. در این حالت آدرس دهی حافظه همیشه فیزیکی است، پردازنده 16 بیت است و فقط 1 مگابایت حافظه داریم.
برای اینکه بتوانیم از پردازنده 32 بیت استفاده کنیم و تا 4 گیگابایت حافظه داشته باشیم، باید وارد protected mode بشویم که این کار فقط در اسمبلی (با 1 کردن بیت اول Control Register 0) ممکن است.

۱۴. وظیفه ثبات[1]های xv6

ثبات عام منظوره[2]:

پردازنده‌های x86 دارای 8 ثبات عام منظوره هستند که می‌توانند به‌عنوان متغیر در زبان C قرار بگیرند. نام همگی این ثبات‌ها به دلیل 32 بیت بودنشان با حرف e که مخفف extended است، شروع می‌شود.
این ثبات‌ها عبارتند از:

- EAX (Accumulator Register): برای عملیات حسابی و منطقی استفاده می‌شود. همچنین برای ذخیره مقادیر بازگشتی تابع نیز مورد استفاده قرار می‌گیرد.
- EBX (Base Register): به‌عنوان یک اشاره‌گر پایه برای دسترسی به حافظه و ذخیره داده‌ها استفاده می‌شود.
- ECX (Counter Register): برای کنترل حلقه و عملیات‌های شمارشی استفاده می‌شود.
- EDX (Data Register): برای عملیات‌های محاسباتی و IO مورد استفاده قرار می‌گیرد.
- ESI (Source Index Register): آدرس حافظه‌ای را در خود نگه می‌دارد که برای خواندن داده‌ها از memory در طول عملیات رشته استفاده می‌شود.
- EDI (Destination Index Register): آدرس حافظه‌ای را در خود نگه می‌دارد که برای نوشتن داده‌ها در memory در طول عملیات رشته استفاده می‌شود.
- ESP (Stack Pointer Register): برای اشاره به بالای stack استفاده می‌شود و در عملیات پشته و فراخوانی تابع به کار برده می‌شود.

[1] Register
[2] General Purpose Register

- EBP (Base Pointer Register): به‌عنوان یک اشاره‌گر پایه برای دسترسی به پارامترهای تابع و متغیرهای محلی استفاده می‌شود.

ثبات قطعه[1]:

پردازنده‌های x86 دارای 6 ثبات قطعه هستند. به‌طور کلی وظیفه این نوع ثبات نگهداری آدرس stack، data و کد است.

این ثبات‌ها عبارتند از:

- CS (Code Segment): برای تعیین محل حافظهٔ کد در حال اجرا استفاده می‌شود.
- DS (Data Segment): هنگام دسترسی به متغیرها یا داده‌ها، از این ثبات برای محاسبه آدرس حافظه مؤثر استفاده می‌شود.
- SS (Stack Segment): برای مدیریت پشته ضروری است. ذخیره متغیرهای محلی، اطلاعات فراخوانی تابع و آدرس‌های برگشتی از موارد استفاده این ثبات محسوب می‌شود.
- ES (Extra Segment): این ثبات یک قطعه اضافی برای ذخیره‌سازی داده‌ها در صورت لزوم فراهم می‌کند.
- FS (Additional Segment): این ثبات قطعه دیگری است که می‌تواند برای ذخیره‌سازی داده‌های اضافی استفاده شود.
- GS (Additional Segment): مشابه ثبات FS، می‌تواند برای اهداف ذخیره‌سازی اطلاعات اضافی مورد استفاده قرار گیرد.

ثبات وضعیت[2]:

ثبات FLAGS، ثبات وضعیتی است که نشان‌دهنده حالت فعلی پردازنده است. این ثبات مخصوص پردازنده‌های 16 بیتی است. EFLAGS و RFLAGS ثبات‌های مشابه برای پردازنده‌های 32 بیتی و 64 بیتی می‌باشند. هر بیت از این ثبات نشان‌دهنده یک پرچم برای یک وضعیت می‌باشد که می‌تواند حالت درست یا غلط داشته باشد. این پرچم‌ها نشان‌دهنده وضعیت اعمال منطقی و محاسباتی یا محدودیت‌های اعمال شده بر عملیات فعلی پردازنده هستند. واضح است که عملکرد این پرچم‌ها به تعداد بیت‌های رجیستر و معماری پردازنده بستگی دارد. ثبات FLAGS برای پردازنده Intel x86 به شرح زیر می‌باشد:

[1] Segment Register
[2] Status Register

Intel x86 FLAGS register			
بیت	مخفف	توضیح	دسته‌بندی
0	CF	Carry Flag	وضعیت
1		رزرو شده	
2	PF	Parity Flag	وضعیت
3		رزرو شده	
4	AF	Adjust Flag	وضعیت
5		رزرو شده	
6	ZF	Zero Flag	وضعیت
7	SF	Sign Flag	وضعیت
8	TF	Trap Flag	کنترل
9	IF	Interrupt enable Flag	کنترل
10	DF	Direction Flag	کنترل
11	OF	Overflow Flag	وضعیت
12-13	IOPL	سطح دسترسی ورودی خروجی	سیستم
14	NT	پرچم فعالیت تو در تو	سیستم
15		رزرو شده	

ثبات کنترلی[1]:

ثبات کنترلی رفتارکلی cpu یا سایر دستگاه‌های دیجیتال را تغییر می‌دهد یا کنترل می‌کند. از وظایف ثبات‌های کنترلی می‌توان به کنترل interrupt، تغییر حالت آدرس‌دهی، کنترل صفحه‌بندی (paging) و کنترل کمک‌پردازنده اشاره کرد.

از این دسته ثبات‌ها می‌توان به CR0 اشاره کرد که در پردازنده‌های 32 بیتی مانند i386 و بالاتر استفاده می‌شود. بیت‌های این ثبات نشان‌دهنده تغییرات و کنترل‌های مختلفی در رفتار کلی پردازنده هستند که به شرح زیر می‌باشند:

[1] Control Register

نام	مخفف	بیت
Protected Mode Enable	PE	0
Monitor co-processor	MP	1
Emulation	EM	2
Task switched	TS	3
Extension type	ET	4
Numeric error	NE	5
Write protect	WP	16
Alignment mask	AM	18
Not-write through	NW	29
Cache disabled	CD	30
Paging	PG	31

info registers:

برای اجرای این دستور، ابتدا باید وارد محیط GDB شویم که با این دستور امکان پذیر است:

```
make qemu-nox-gdb
```

حال با زدن کلید Ctrl+A و سپس C به ترمینال qemu رفته و دستور `info registers` را وارد می‌کنیم:

```
(qemu) info registers
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000663
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00000000
EIP=0000fff0 EFL=00000002 [-------] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 00000000 0000ffff 00009300
CS =f000 ffff0000 0000ffff 00009b00
SS =0000 00000000 0000ffff 00009300
DS =0000 00000000 0000ffff 00009300
FS =0000 00000000 0000ffff 00009300
GS =0000 00000000 0000ffff 00009300
LDT=0000 00000000 0000ffff 00008200
TR =0000 00000000 0000ffff 00008b00
GDT=     00000000 0000ffff
IDT=     00000000 0000ffff
CR0=60000010 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=0000000000000000 0000000000000000 XMM01=0000000000000000 0000000000000000
XMM02=0000000000000000 0000000000000000 XMM03=0000000000000000 0000000000000000
XMM04=0000000000000000 0000000000000000 XMM05=0000000000000000 0000000000000000
XMM06=0000000000000000 0000000000000000 XMM07=0000000000000000 0000000000000000
```

با تکرار همان دکمه‌ها می‌توانیم به ترمینال xv6 بازگردیم.

۱۵. نقش اصلی Real mode در پردازنده x86

Real mode در پردازنده x86 یک حالت عملیاتی است که در آن آدرس‌ها به طور مستقیم به مکان‌های حقیقی در حافظه اشاره می‌کنند. این حالت توسط تمامی پردازنده‌های سازگار با x86 استفاده می‌شود. در real mode، فضای آدرسی حافظه به صورت segment-based مشخص می‌شود و امکان دسترسی مستقیم به حافظه، آدرس‌های ورودی/خروجی و سخت‌افزارهای پریفرال وجود دارد. اما این حالت از حفاظت حافظه، چند وظیفه‌ای و سطوح دسترسی کد پشتیبانی نمی‌کند.

نقش اصلی real mode در پردازنده x86 عبارت است از:

- به عنوان حالت اولیه پردازنده x86 بعد از ریست شدن استفاده می‌شود.
- برای اجرای سیستم‌عامل‌های قدیمی مانند DOS استفاده می‌شود.
- برای اجرای برنامه‌هایی که برای پردازنده‌های قدیمی‌تر x86 نوشته شده‌اند، استفاده می‌شود.

در real mode، آدرس‌ها به صورت segment:offset مشخص می‌شوند، که segment اشاره‌گری به یک بلاک 64KB از حافظه است و offset نشانگر آدرس مطلق درون آن بلاک است. برای مثال، آدرس فیزیکی 0x12345 در real mode به صورت segment:offset نشان داده می‌شود، مانند 0x1234:0x5. این ساختار آدرس‌دهی به real mode امکان دسترسی به 1 مگابایت حافظه را فراهم می‌کند.

برای تغییر از real mode به protected mode در پردازنده x86، باید مراحل زیر را انجام داد:

- تنظیم فلش‌های معماری پردازنده برای فعال کردن protected mode.
- تنظیم مقادیر مربوط به مکان حافظه و رجیسترهای مهم (مانند رجیستر CS) برای ورود به protected mode.

همچنین، در حالت real mode امکان استفاده از محافظ حافظه، چند وظیفه‌ای بودن و سطوح دسترسی کد وجود ندارد. بنابراین، برنامه‌نویسان باید خودشان به صورت دستی محافظت از حافظه و مدیریت وظایف را ایجاد کنند.

۱۶. آدرس‌دهی حافظه در Real mode

حافظه در real mode یک توالی خطی از بایت‌هاست که می‌تواند آزادانه با هر آدرس 20 بیتی که از 16 بیت آدرس segment و 4 بیت آدرس offset تشکیل شده باشد، آدرس‌دهی شود. برنامه می‌تواند فارغ از اینکه در چه بخشی از حافظه قرار دارد، به هر نقطه دسترسی پیدا کند و در آن بخواند یا بنویسد.

نهایتا آدرس ما به‌صورت زیر محاسبه می‌شود:

PhysicalAddress = Segment * 16 + Offset

۱۷. کد bootmain.c چرا هسته را در آدرس 0x100000 قرار می‌دهد؟

۱۸. کد معادل entry.S در هسته لینوکس

کد معادل entry.S برای معماری x86 در هسته لینوکس:
https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry.S
که برای 32 بیت و 64 بیت مجزا است.

اجرای هسته xv6

۱۹. دلیل فیزیکی بودن آدرس page table

این آدرس نباید مجازی باشد، زیرا برای دسترسی به آن باید آدرس مجازی‌اش را به آدرس فیزیکی تبدیل کنیم و برای این کار نیازمند جدول ذکر شده هستیم و دسترسی به جدول نیز با داشتن آدرس آن میسر است که به‌صورت مجازی ذخیره می‌شود. برای پیدا کردن آدرس فیزیکی‌اش به خودش نیاز خواهیم داشت و حلقه بی‌نهایتی بوجود می‌آید که این حالت باعث ایجاد تناقض می‌شود و هیچ وقت نمی‌توانیم به این جدول دسترسی پیدا کنیم. در صورتی که بخواهیم از یک جدول دیگر برای پیدا کردن آدرس فیزیکی این جدول استفاده کنیم، در نهایت نیاز به یک آدرس فیزیکی برای پایان دادن به حلقه خواهیم داشت. در نتیجه آدرس دسترسی به این جدول به‌صورت فیزیکی (در رجیستر CR3) ذخیره می‌شود.

۲۰. توضیح توابع entry.S و تابع معادل در هسته لینوکس

۲۱. توضیح مختصری راجع به محتوای فضای آدرس مجازی هسته

۲۲. چرا برای کد و داده‌های سطح کاربر پرچم SEG_USER تنظیم شده است؟

قطعه‌بندی در xv6 در تابع `seginit` و در تکه کد زیر انجام می‌شود:

vm.c
```
c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
```

علاوه بر آن تعریف SEG به‌صورت زیر می‌باشد:

mmu.h
```
#define SEG(type, base, lim, dpl) (struct segdesc)    \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
```

بنابراین همان‌طور که مشخص است (و در توضیحات آزمایش نیز آمده است) تمام قطعه‌های هسته و کاربر یک بخش از حافظه را در اختیار دارند. هر یک از این قطعه‌ها با یک descriptor در GDT[1] مشخص شده که این descriptor شامل اطلاعاتی مانند آدرس شروع قطعه، اندازه قطعه و سطح دسترسی قطعه می‌باشد.

برای خواندن یک دستورالعمل، ابتدا قطعه آن از طریق descriptor اش یافت می‌شود (که در اینجا قطعه کدِ descriptor هسته و کاربر یکسان اند) و سپس صفحه مربوط به آن پس از طی مراحل مربوطه پیدا می‌شود. پس از این مراحل و تبدیل آدرس منطقی به آدرس فیزیکی، دستورالعمل از حافظه خوانده شده و اجرا می‌شود. موضوعی که در این مرحله باید به آن دقت کرد، سطح دسترسی مورد نیاز یک دستور برای اجرای آن است. هنگامی که مکان قطعه از روی descriptor قطعه مشخص می‌شود، سطح دسترسی فعلی یا همان CPL[2] نیز از روی سطح دسترسی descriptor یا همان DPL[3] مشخص می‌شود. بدین گونه از طریق DPL متفاوت می‌توان سطح دسترسی فعلی دستورالعمل‌ها را نیز تعیین کرد؛ حتی اگر این descriptorها قطعات یکسانی از حافظه را تعریف کنند.

برای مثال دستورالعمل IN، وظیفه خواندن یک بایت از پورت را دارد و این عمل نیازمند این است که سطح دسترسی فعلی مقداری ممتازتر از سطح دسترسی ورودی/خروجی داشته باشد (سطح دسترسی ورودی/خروجی در رجیستر وضعیت FLAG مشخص شده است) که این مقدار در لینوکس برابر صفر است؛ مقدار دسترسی فعلی (CPL) برابر مقدار سطح دسترسی descriptor (DPL) قطعه‌ای است که کد مربوط به این دستور العمل در آن قرار گرفته است و اگر این دستورالعمل در قطعه کاربر قرار گرفته باشد قابل اجرا نخواهد بود؛ چرا که قطعه مربوط به کد کاربر، سطح دسترسی‌ش برابر DPL_USER یا همان 3 (کمترین میزان دسترسی) است.

بنابراین با وجود اینکه هر دو بخش کاربر و هسته به قطعات یکسانی دسترسی دارند، اما سطح دسترسی متفاوتی داشته و کاربر هر دستورالعملی را نمی‌تواند اجرا کند.

[1] Global Descriptor Table
[2] Current Privilege Level
[3] Descriptor Privilege Level

اجرای نخستین برنامه سطح کاربر

۲۳. اجزای struct proc و معادل آن در لینوکس

این struct که برای ذخیره وضعیت هر پردازه به کار می‌رود، در فایل proc.h تعریف شده و 13 متغیر در آن قرار دارد:

- `sz`: حجم و اندازه حافظه گرفته شده توسط پردازه که به واحد بایت است.
- `pgdir`: پوینتر به page table پردازه است. (pde: page directory entry)
- `kstack`: پوینتر به انتهای kernel stack پردازه است. kernel stack قسمتی از kernel space است و نه user space و برای اجرای syscallها از برنامه استفاده می‌شود.
- `state`: این `enum` وضعیت پردازه را مشخص می‌کند و می‌تواند به حالت‌های procstate یعنی UNUSED، EMBRYO، SLEEPING، RUNNABLE، RUNNING و ZOMBIE باشد.
- `pid`: pid (process ID) یک شناسه یکتا است که به هر فرایند در حال اجرا اختصاص داده می‌شود. این شناسه برای تمایز دادن یک فرایند از دیگری استفاده می‌شود.
- `parent`: پوینتر به پردازه والد (پردازه سازنده پردازه کنونی توسط تابع `fork`) است.
- `tf`: پوینتر به trap frame برای ذخیره وضعیت اجرای برنامه در هنگام اجرای یک syscall.
- `context`: پوینتر به `struct context` است که مقادیر رجیسترهای مورد نیاز برای context switching را نگه می‌دارد. با استفاده از تابع `swtch` (که با اسمبلی تعریف شده) می‌توان به یک پردازه switch کرد.
- `chan`: اگر مقدار آن 0 نباشد، پردازه به خواب می‌رود. (برای کاری wait می‌کند.)
  اینجا chan به معنای channel است و چنل‌های متعددی از جمله چنل خط ورودی کنسول داریم.
- `killed`: اگر مقدار آن 0 نباشد، یعنی پردازه kill شده‌است.
- `ofile`: آرایه‌ای از پوینترها به فایل‌های باز شده توسط پردازه است. (حداکثر تعداد فایل‌های باز در هر پردازه، به اندازه ثابت `NOFILE` می‌باشد.)
  هروقت کاربر یک فایل را باز می‌کند، یک entry جدید به آرایه افزوده می‌شود که ایندکس آن entry به عنوان file descriptor به کاربر برگردانده می‌شود. (زمانی که کاربر می‌خواهد فایلی را بخواند یا بنویسد، از این file descriptorها برای رجوع استفاده می‌کند.)
  همچنین در این آرایه سه خانه اول برای stdin، stdout و stderr کنار گذاشته شده‌اند.
- `cwd`: این متغیر current working directory را مشخص می‌کند.

- `name`: نام پردازه برای اشکال زدایی.

معادل این struct در لینوکس، در لینک زیر و در استراکت `task_struct` قرار دارد:

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

استراکت `task_struct` نیز مانند استراکت `proc`، توصیف‌گر پردازه می‌باشد و همه چیزهایی که یک هسته باید در مورد یک پردازه بداند را در خودش جای می‌دهد.

۲۴. چرا به خواب رفتن در کد مدیریت‌کننده سیستم‌عامل مشکل‌ساز است؟

۲۵. تفاوت فضای آدرس هسته با فضای آدرس توسط kvmalloc

۲۶. تفاوت فضای آدرس inituvm با فضای آدرس کاربر در کد مدیریت سیستم

۲۷. کدام بخش از آماده‌سازی سیستم بین تمامی هسته‌های پردازنده مشترک و کدام بخش اختصاصی است؟

آماده‌سازی سیستم توسط دو تابع `main` و `mpenter` (که خود دارای توابع متعددی هستند) واقع در فایل main.c انجام می‌شود:

main.c

```
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
}
```

main.c

```
static void
mpenter(void)
{
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
```

هسته اول فرایند بوت را انجام می‌دهد و توسط کد entry.S وارد تابع main می‌شود. تمامی توابع آماده‌سازی سیستم که در این تابع فراخوانده شده‌اند توسط این هسته اجرا می‌شوند. هسته‌های دیگر از طریق کد entryother.S وارد تابع mpenter می‌شوند. با توجه به اینکه این تابع در تابع main نیز فراخوانی می‌شود، می‌توان گفت این 4 تابع بین تمامی هسته‌ها مشترک خواهند بود.

یکی از این توابع به نام switchkvm به صورت مستقیم با هسته اول مشترک نیست. این تابع در mpenter صدا زده می‌شود در صورتی که در تابع main وجود ندارد. در واقع تابع kvmalloc که در main صدا زده می‌شود به صورت زیر است:

vm.c

```
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

خط اول تابع یک page table برای کرنل ایجاد می‌کند که این مورد توسط هسته اول انجام می‌پذیرد. پس از آن باید هسته به این page table سوییچ کند که این کار در تمامی هسته‌ها انجام می‌پذیرد. بخش‌هایی از آماده‌سازی سیستم که در تمام هسته‌ها مشترک هستند به شرح زیر است:

- switchkvm
- seginit
- lapicinit
- mpmain

همچنین بخش‌هایی که تنها در هسته اول (به صورت اختصاصی) اجرا می‌شوند به شرح زیر است:

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

- `kinit2`
- `userinit`

از موارد اختصاصی هسته اول می‌توان به تابع `startothers` اشاره کرد که واضح است فقط پردازنده اول نیاز است بقیه پردازنده‌ها را start کند و نیازی نیست هر پردازنده در زمان بالا آمدن این کار را انجام دهد. یا برای مثال زمانی که پردازنده اول به کمک تابع `ideinit` دیسک را شناسایی می‌کند، نیازی نیست بقیه پردازنده‌ها این کار را انجام دهند.

از طرفی همه پردازنده‌ها باید آدرس page table که توسط پردازنده اول ایجاد شده را در رجیستر خود ذخیره کنند در نتیجه تابع `switchkvm` بین همه آن‌ها مشترک است. همچنین، همه پردازنده‌ها باید کار خود را شروع کنند و آماده اجرای برنامه‌ها شوند که این مورد توسط تابع `mpmain` انجام می‌پذیرد. در نتیجه این تابع هم بین تمام پردازنده‌ها مشترک خواهد بود.

زمان‌بند که توسط تابع `scheduler` انجام می‌پذیرد در تابع `mpmain` صدا زده می‌شود که این تابع بین تمامی هسته‌ها مشترک است. این مورد از کامنت‌های داکیومنت تابع ذکر شده نیز قابل برداشت است:

proc.c

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

هر پردازنده scheduler مربوط به خودش را خواهد داشت و در نتیجه این تابع بین تمامی پردازنده‌ها مشترک است.

۲۸. برنامه معادل initcode.S در هسته لینوکس

اشکال زدایی

۱. دستور مشاهده breakpointها

با استفاده از دستور `make qemu-gdb` سیستم‌عامل را به‌صورتی بوت می‌کنیم که قابلیت اتصال اشکال زدا به آن وجود داشته باشد. سپس در ترمینال دیگر دستور `gdb _cat` را جهت اشکال زدایی بخش سطح کاربر وارد می‌کنیم. نهایتا با وارد کردن دستور `target remote tcp::26000` اتصال به سیستم‌عامل صورت خواهد گرفت.

با استفاده از convenience variable ء `$bpnum` می‌توانیم تعداد breakpointها را مشاهده کنیم. در صورتی که تمام اطلاعات مربوط به breakpointها را نیاز داشته باشیم، می‌توانیم از دستور `info break` یا `info breakpoints` کمک بگیریم.

شایان ذکر است می‌توان عددی هم به‌عنوان آرگومان به این دستور داد تا اطلاعات همان breakpoint را برای ما نمایش دهد.

```
(gdb) break cat.c:12
Breakpoint 1 at 0x93: file cat.c, line 12.
(gdb) break cat.c:14
Breakpoint 2 at 0xdc: file cat.c, line 14.
(gdb) b *0xf0
Breakpoint 3 at 0xf0: file cat.c, line 19.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00000093 in cat at cat.c:12
2       breakpoint     keep y   0x000000dc in cat at cat.c:14
3       breakpoint     keep y   0x000000f0 in cat at cat.c:19
(gdb) info breakpoints 2
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x000000dc in cat at cat.c:14
(gdb) p $bpnum
$1 = 3
```

۲. دستور حذف یک breakpoint

برای حذف یک breakpoint می‌توان از دو روش زیر استفاده کرد:

- دستور `del <breakpoint_number>`
- دستور `clear <file_name>:<line_number>`

مقدار `breakpoint_number` را می‌توان با استفاده از دستور `info breakpoints` یا `info break` مشاهده کرد.

در ادامه دو تا از breakpointهایی که گذاشتیم را حذف می‌کنیم:

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00000093 in cat at cat.c:12
2       breakpoint     keep y   0x000000dc in cat at cat.c:14
3       breakpoint     keep y   0x000000f0 in cat at cat.c:19
(gdb) del 1
(gdb) clear cat.c:19
Deleted breakpoint 3
(gdb) info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x000000dc in cat at cat.c:14
```

کنترل روند اجرا و دسترسی به حالت سیستم

۳. خروجی bt

دستور `bt` که مخفف backtrace است call stack برنامه در لحظه کنونی (در حین متوقف بودن روند اجرای برنامه) را نشان می‌دهد.

هر تابع که صدا زده می‌شود یک stack frame مخصوص به خودش را می‌گیرد که متغیرهای محلی و آدرس بازگشت و غیره در آن قرار دارند.

خروجی این دستور در هر خط یک stack frame را نشان می‌دهد که به ترتیب از درونی‌ترین frame که در آن قرار داریم شروع می‌شود.

می‌توان با دستور `bt n` که n یک عدد است فقط n فریم درونی‌تر را نشان داد و با دستور `bt -n` فقط n فریم بیرونی‌تر را نشان داد.

برای استفاده از این دستور می‌توان از کلیدواژه‌های مختلفی استفاده کرد از جمله:

`bt, backtrace, where, info stack`

در مثال زیر، در خط 15 فایل wc.c یک breakpoint گذاشته شده است. این خط کد، داخل تابعی به نام `wc` قرار دارد که از داخل تابع `main` ورودی برنامه wc صدا می‌شود.

پس از اجرای کامند `wc README` در ترمینال xv6 مشاهده می‌کنیم که روی خط 15 متوقف شده و دستور `bt` به طور صحیح call stack را نشان می‌دهد.

```
(gdb) break 15
Breakpoint 1 at 0xa0: file wc.c, line 15.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, wc (fd=3, name=0x2ff4 "README") at wc.c:15
15        while((n = read(fd, buf, sizeof(buf))) > 0){
(gdb) bt
#0  wc (fd=3, name=0x2ff4 "README") at wc.c:15
#1  0x00000056 in main (argc=2, argv=0x2fe8) at wc.c:50
```

۴. تفاوت دستورهای x و print

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

```
(gdb) help print
print, inspect, p
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.

  -memory-tag-violations [on|off]
    Set printing of memory tag violations for pointers.
    Issue a warning when the printed value is a pointer
    whose logical tag doesn't match the allocation tag of the memory
    location it points to.

  -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.
```

با استفاده از دستور x می‌توان محتویات یک خانه حافظه را چاپ کرد. بدیهی‌ست که این دستور، آدرس خانه حافظه را به‌عنوان آرگومان می‌گیرد.

```
(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".
```

لازم به ذکر است که هر دو دستور ذکر شده می‌توانند فرمت خروجی را به‌صورت /FMT در آرگومان‌های ورودی خود دریافت کنند.

در مثال زیر، پس از دستور cat mmm_result.txt متغیر fd چاپ می‌شود. برای پیدا کردن آدرس این متغیر نیز از دستور print &fd استفاده شده‌است:

```
(gdb) break 12
Breakpoint 1 at 0x93: file cat.c, line 12.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=3) at cat.c:12
12          while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) print fd
$1 = 3
(gdb) print &fd
$2 = (int *) 0x2f90
(gdb) x 0x2f90
0x2f90: 0x00000003
(gdb) x/d 0x2f90
0x2f90: 3
```

همچنین برای مشاهده مقدار یک ثبات خاص می‌توان از دستور زیر استفاده کرد:

```
info registers <reg_name>
```

```
(gdb) info registers eax
eax            0x3                 3
(gdb) info registers ebx
ebx            0x2fe8              12264
```

۵. نمایش وضعیت ثبات‌ها و متغیرهای محلی؛ رجیسترهای edi و esi

برای نمایش وضعیت ثبات‌ها از دستور `info registers` یا مخفف آن یعنی `i r` می‌توان استفاده کرد:

```
(gdb) info registers
eax            0x0                 0
ecx            0x0                 0
edx            0x663               1635
ebx            0x0                 0
esp            0x0                 0x0
ebp            0x0                 0x0
esi            0x0                 0
edi            0x0                 0
eip            0xfff0              0xfff0
eflags         0x2                 [ IOPL=0 ]
cs             0xf000              61440
ss             0x0                 0
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
fs_base        0x0                 0
gs_base        0x0                 0
k_gs_base      0x0                 0
cr0            0x60000010          [ CD NW ET ]
cr2            0x0                 0
cr3            0x0                 [ PDBR=0 PCID=0 ]
cr4            0x0                 [ ]
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_int3
2 = {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_int3
2 = {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_int3
2 = {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_int3
2 = {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_int3
2 = {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_int3
2 = {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_int3
2 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
--Type <RET> for more, q to quit, c to continue without paging--
```

برای مشاهده متغیرهای محلی نیز می‌توان از دستور `info locals` استفاده کرد. خروجی این دستور برای اشکال زدایی فایل cat.c به‌صورت زیر می‌باشد:

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

همچنین اگر بخواهیم وضعیت تمام متغیرها (حتی غیر localها) را ببینیم، از دستور `info variables` استفاده می‌کنیم:

```
(gdb) info variables
All defined variables:

File cat.c:
5:      char buf[512];

File umalloc.c:
21:     static Header base;
22:     static Header *freep;

Non-debugging symbols:
0x00000880  digits
0x00000b64  __bss_start
0x00000b64  _edata
0x00000d8c  _end
```

ثبات SI مخفف Source Index بوده و برای اشاره به یک مبدا در عملیات stream به کار می‌رود. DI نیز مخفف Destination Index بوده و برای اشاره به یک مقصد در عملیات stream به کار می‌رود. E در ابتدای اسامی این ثبات‌ها به معنی Extended بوده و در حالت 32 بیت به کار می‌رود. SI به‌عنوان نشانگر داده و به‌عنوان مبدا در برخی عملیات مربوط به رشته‌ها استفاده می‌شود. DI نیز به‌عنوان نشانگر داده و مقصد برخی عملیات مربوط به رشته‌ها استفاده می‌شود.

رجیسترهای edi و esi هر دو رجیسترهای عام منظوره هستند.

۶. ساختار struct input

این struct در فایل console.c تعریف شده است و برای خط ورودی کنسول سیستم عامل استفاده می‌شود. این struct در کد چنین تعریف شده‌است (توجه داشته باشید که متغیر `len` را ما در بخش قابلیت‌های کنسول افزوده بودیم و به‌صورت default وجود نداشت):

console.c

```
#define INPUT_BUF 128
struct {
  char buf[INPUT_BUF];
  uint r;   // Read index
  uint w;   // Write index
  uint e;   // Edit index
  uint len; // Buffer length
} input;
```

یعنی از یک instance به نام input از یک unnamed struct استفاده می‌شود. این را در GDB هم می‌توان با کامند `ptype` برای پرینت کردن تایپ یک متغیر مشاهده کرد:

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

متغیرهای استراکت:

- `buf`: این آرایه بافر و محل ذخیره خط ورودی است که اندازه آن حداکثر 128 کاراکتر است.
- `r`: برای خواندن `buf` استفاده می‌شود و از اندیس `w` قبلی شروع می‌کند.

زمانی که در حال خواندن از بافر هستیم (تابع `consoleread`)، به ازای هر کاراکتری که از بافر خوانده می‌شود، اندیس `r` یک خانه در آرایه به جلو می‌رود تا زمانی که دیگر کاراکتری برای خواندن از بافر نباشد.

- `w`: محل شروع نوشتن خط ورودی کنونی در `buf` است.
- `e`: محل کنونی cursor در خط ورودی را نگه می‌دارد.
- `len`: اندازه بافر (تعداد کاراکترهای آرایه `buf`) در این متغیر ذخیره می‌شود.

نحوه تغییر این متغیرها را با یک مثال می‌بینیم:

یک breakpoint روی تابع `consoleinit` می‌گذاریم تا بتوانیم مقادیر اولیه متغیرها را پرینت کنیم:

```
(gdb) b consoleinit
Breakpoint 1 at 0x80100f90: file console.c, line 431.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, consoleinit () at console.c:431
431       initlock(&cons.lock, "console");
(gdb) print input
$1 = {buf = '\000' <repeats 127 times>, r = 0, w = 0, e = 0, len = 0}
```

حال breakpoint دیگری در تابع `consoleintr` و در انتهای بخش default (انتهای بدنهٔ شرطی که چک می‌کند Enter یا ctrl+d زده شده‌است یا cursor از `buf` فراتر رفته‌است)، می‌گذاریم و پس از continue عبارت test را در ترمینال xv6 وارد می‌کنیم:

```
(gdb) break console.c:363
Breakpoint 2 at 0x80100f01: file console.c, line 363.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, consoleintr (getc=0x80102c20 <kbdgetc>) at console.c:363
363               wakeup(&input.r);
(gdb) print input
$1 = {buf = "test\n", '\000' <repeats 122 times>, r = 0, w = 5, e = 5, len = 5}
```

می‌بینیم که ورودی در `buf` قرار گرفته و متغیر `e` به 5 تغییر یافته که مکان بعد از آخرین حرف `buf` است.

حال دوباره continue کرده و دستی (با ctrl+c) روند اجرا را متوقف می‌کنیم:

```
(gdb) continue
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
mycpu () at proc.c:48
48        for (i = 0; i < ncpu; ++i) {
(gdb) print input
$2 = {buf = "test\n", '\000' <repeats 122 times>, r = 5, w = 5, e = 5, len = 5}
```

می‌بینیم که مقدار r به همان مقدار w رسیده است. یعنی از w قبلی (که 0 بود) شروع کرده و به w کنونی می‌رسد تا کل خط را بخواند. (با گذاشتن یک watchpoint می‌توان دقیق‌تر بررسی کرد که r یکی یکی جلو می‌رود.)
این بار عبارت another را در ترمینال xv6 وارد می‌کنیم:

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, consoleintr (getc=0x80102c20 <kbdgetc>) at console.c:363
363               wakeup(&input.r);
(gdb) print input
$3 = {buf = "test\nanother\n", '\000' <repeats 114 times>, r = 5, w = 13, e = 13, len = 13}
```

باز هم به آخر buf رفته و e هم در ابتدای خط ورودی جدید است پس با w برابر است.
اگر برنامه را continue و سپس متوقف کنیم، می‌بینیم که r به w می‌رسد:

```
(gdb) continue
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
mycpu () at proc.c:48
48        for (i = 0; i < ncpu; ++i) {
(gdb) print input
$4 = {buf = "test\nanother\n", '\000' <repeats 114 times>, r = 13, w = 13, e = 13, len = 13}
```

حال continue کرده و عبارت xyz را در ترمینال xv6 می‌نویسیم ولی Enter نمی‌زنیم و دستی برنامه را متوقف می‌کنیم:

```
(gdb) continue
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
mycpu () at proc.c:48
48        for (i = 0; i < ncpu; ++i) {
(gdb) print input
$5 = {buf = "test\nanother\nxyz", '\000' <repeats 111 times>, r = 13, w = 13, e = 16, len = 16}
```

طبق انتظار متغیر e جلو رفته است. اگر کاراکتر آخر را از ترمینال xv6 پاک کنیم:

```
(gdb) continue
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
mycpu () at proc.c:48
48        for (i = 0; i < ncpu; ++i) {
(gdb) print input
$6 = {buf = "test\nanother\nxyz", '\000' <repeats 111 times>, r = 13, w = 13, e = 15, len = 15}
```

می‌بینیم که e یک واحد به عقب بر می‌گردد.

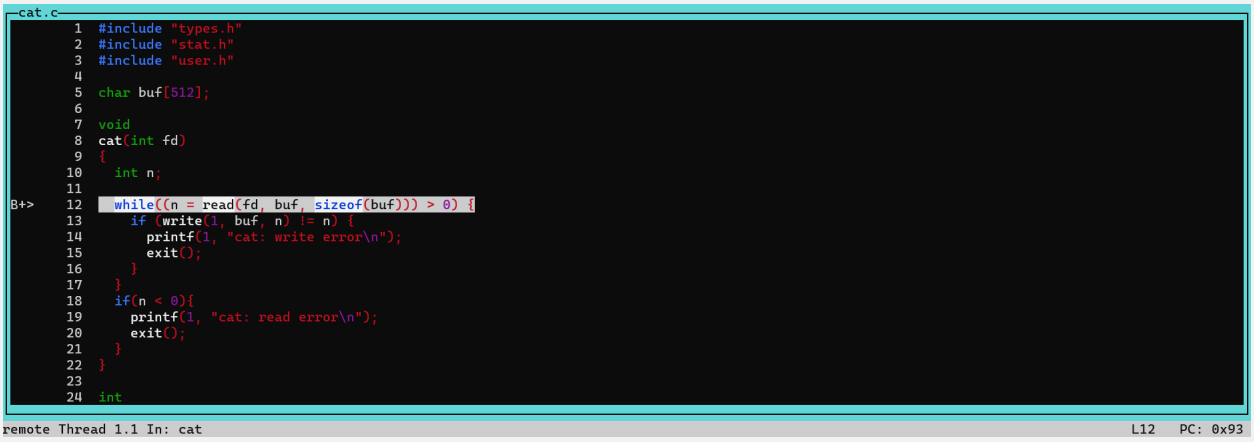
توجه که هر حرکت cursor خود 3 کاراکتر در این بافر می‌ریزد و مقدار e را افزایش می‌دهد، حتی اگر رو به عقب باشد.

اشکال زدایی در سطح کد اسمبلی

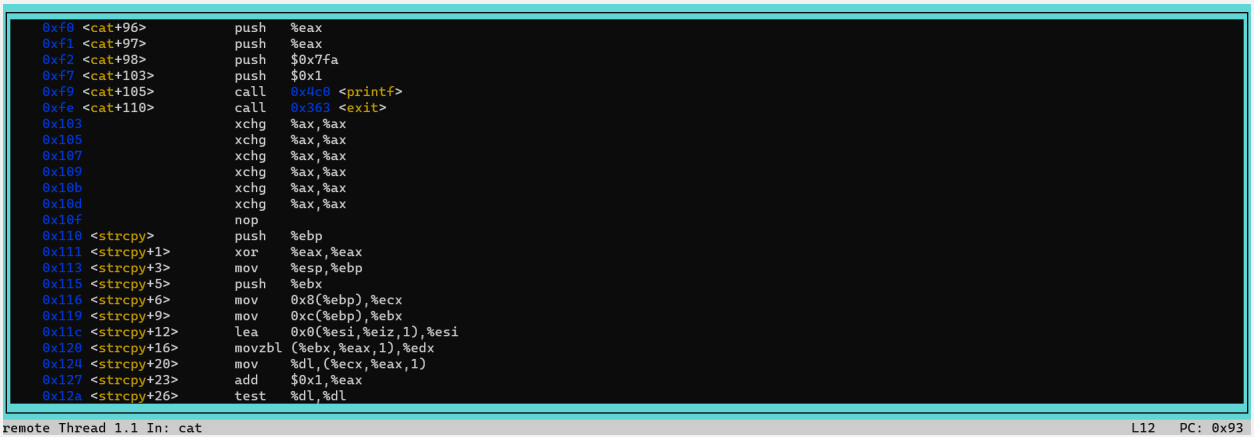
۷. خروجی دستورهای layout src و layout asm در TUI

در محیط TUI با استفاده از دستور layout src می‌توان کد سورس برنامه را در حال دیباگ نمایش داد.

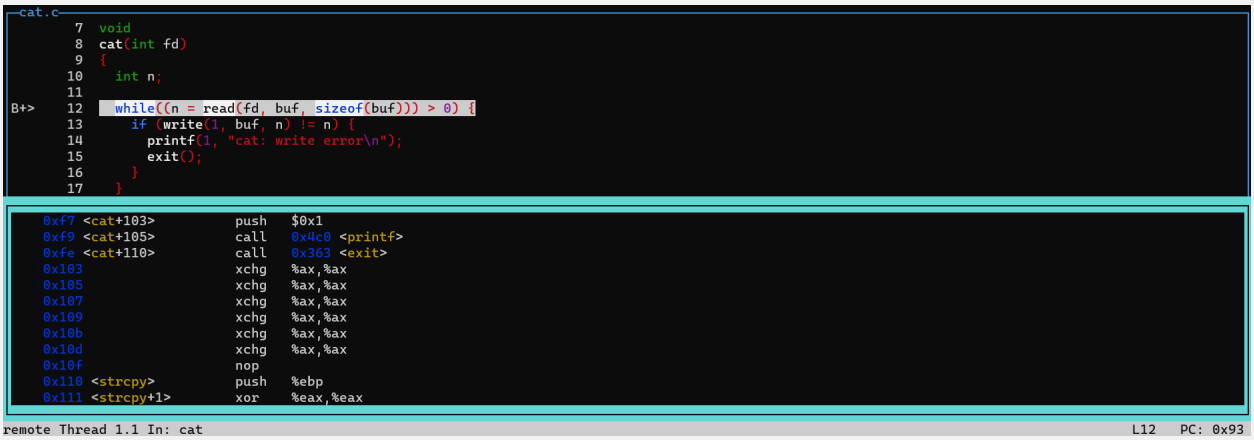
در اینجا ما در خط 12 فایل cat.c یک breakpoint گذاشتیم و با ابزار GDB می‌خواهیم کد سورس (و در ادامه اسمبلی) آن را در قسمتی که breakpoint نهادیم، مشاهده کنیم.



با استفاده از دستور layout asm می‌توانیم کد اسمبلی برنامه در حال دیباگ را مشاهده کنیم:



در نهایت با استفاده از دستور layout split می‌توانیم کد سورس برنامه و اسمبلی آن را به‌طور همزمان مشاهده کنیم:



۸. دستورهای جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف)

برای وضعیت مشاهده پشته فراخوانی فعلی می‌توان از دستور `where` یا `backtrace` در محیط کاربری TUI استفاده کرد. در اینجا یک breakpoint در خط 48 فایل proc.c گذاشته‌ایم و پس از توقف اجرا در این نقطه، با استفاده از دستور `where` پشته فراخوانی را مشاهده می‌کنیم:

```
(gdb) break proc.c:48
Breakpoint 1 at 0x80103e31: file proc.c, line 48.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, mycpu () at proc.c:48
48          for (i = 0; i < ncpu; ++i) {
(gdb) where
#0  mycpu () at proc.c:48
#1  0x80103e8b in cpuid () at proc.c:32
#2  0x80106e7b in seginit () at vm.c:24
#3  0x801035c5 in main () at main.c:24
```

برای حرکت در پشته فراخوانی می‌توان از دستورات زیر استفاده کرد:

- `up <n>` یا `u <n>`: به کمک این دستور می‌توانیم n تا frame به سمت بالای stack برویم. (در صورتی که n را مشخص نکنیم، دستور به‌صورت default با n=1 اجرا می‌شود.)
- `down <n>` یا `d <n>`: به کمک این دستور می‌توانیم n تا frame به سمت پایین stack برویم. (در صورتی که n را مشخص نکنیم، دستور به‌صورت default با n=1 اجرا می‌شود.)
- `frame [frame_spec]`: دستور frame در GDB برای انتخاب یک فریم خاص از stack برای بررسی استفاده می‌شود و این امکان را به ما می‌دهد تا از طریق فریم‌های استک برنامه خود حرکت کنیم. به‌جای `frame_spec` می‌توانیم موارد مختلفی قرار دهیم:

- frame level: شماره فریمی از stack که می‌خواهیم به آن برویم را می‌توانیم مشخص کنیم. درونی‌ترین فریم (فریم در حال اجرا) دارای level ء 0 است و فریمی که آن را فراخوانی کرده، دارای level ء 1 است.
- stack address: آدرس فریمی از stack که می‌خواهیم به آن برویم. این آدرس را می‌توان از خروجی دستور `info frame` بدست آورد.
- function name: فریم stack که متعلق به تابع مورد نظر هست را یافته و برای ما نمایش می‌دهد. اگر چند فریم متعلق به تابع باشد، داخلی‌ترین آن‌ها انتخاب می‌شود.
- non-backtrace frame: می‌توانیم از یک stack address و یک program counter اختیاری برای مشاهده فریمی که بخشی از backtrace ء GDB نیست، به این صورت استفاده کنیم: `frame <stack_addr> <pc_addr>`

نمونه‌ای از استفادهٔ این دستورات در زیر نمایش داده شده‌است:

```
(gdb) up 2
#2  0x80106e7b in seginit () at vm.c:24
24          c = &cpus[cpuid()];
(gdb) down
#1  0x80103e8b in cpuid () at proc.c:32
32          return mycpu()-cpus;
(gdb) frame 0
#0  mycpu () at proc.c:48
48          for (i = 0; i < ncpu; ++i) {
(gdb) frame 1
#1  0x80103e8b in cpuid () at proc.c:32
32          return mycpu()-cpus;
```

پیکربندی و ساختن هسته لینوکس (امتیازی)

پس از نصب ubuntu 22.04 روی VMware Workstation Pro، دستور `uname -a` نشان می‌دهد که ورژن هسته لینوکس در این نسخه 6.2.0 است:

```
ali@funlife:~$ uname -a
Linux funlife 6.2.0-33-generic #33~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Thu Sep  7 10:33:52
UTC 2 x86_64 x86_64 x86_64 GNU/Linux
```

برای نمایش اسم اعضای گروه در دستور dmesg یک فایل C و یک Makefile می‌نویسیم:

new_module.c

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

MODULE_LICENSE("GPL");

int init_module(void)
{
    printk(KERN_INFO "Group Members:\n    - Ali Ghanbari\n    - Behrad
Elmi\n    - Bita Nasiri\n");
    return 0;
}

void cleanup_module(void) {}
```

خط چهارم از فایل new_module.c لایسنس و استاندارد را مشخص می‌کند. همچنین تابع `init_module` هنگام صدا زدن module فراخوانده می‌شود و تابع `cleanup_module` حین خروج از module فراخوانده خواهد شد.

Makefile

```
obj-m += new_module.o

all:
	make -C /lib/modules/$(shell uname -r)/build M="$(PWD)" modules
```

پس از اجرای دستور make، یک فایل با نام new_module.ko ایجاد می‌شود. پس از آن، از دستور `sudo insmod new_module.ko` استفاده می‌کنیم تا module به هسته افزوده شود. (برای حذف module نیز می‌توانیم از دستور `sudo rmmod new_module` استفاده کنیم.)

همان‌طور که می‌بینید module با موفقیت به هسته اضافه شده است:

```
ali@funlife:~$ lsmod
Module                  Size  Used by
new_module             16384  0
bnep                   32768  2
intel_rapl_msr         20480  0
intel_rapl_common      40960  1 intel_rapl_msr
crct10dif_pclmul       16384  1
polyval_clmulni        16384  0
polyval_generic        16384  1 polyval_clmulni
ghash_clmulni_intel    16384  0
sha512_ssse3           53248  0
aesni_intel           397312  0
crypto_simd            20480  1 aesni_intel
vmw_balloon            28672  0
```

در نهایت اگر دستور `sudo dmesg` را اجرا کنیم، می‌توانیم اسم اعضای گروه را در انتهای خروجی مشاهده کنیم:

```
[  191.602286] new_module: loading out-of-tree module taints kernel.
[  191.602401] new_module: module verification failed: signature and/or required key missing - tainting kernel
[  191.603680] Group Members:
                    - Ali Ghanbari
                    - Behrad Elmi
                    - Bita Nasiri
```