

به نام خدا

گزارش آزمایشگاه سیستم عامل: پروژه اول

شهاب شرافت - متین نوریان - یاسمن عموجعفری

810102480 - 810102529 - 810102467

گروه: 14

Github repository :

[shahabsherafat/OS: Operating Systems course projects](https://github.com/shahabsherafat/OS: Operating Systems course projects)

Last commit hash: 27f31bca5b4e1e95b67e05d514f6faeaa9f22db8

آشنایی با سیستم عامل و xv6

(1) سه وظیفه اصلی سیستم عامل را نام ببرید.

۱. Resource Management

سیستم عامل مسئول مدیریت منابعی مثل CPU، memory و I/O devices. باید این منابع رو بین process های مختلف به صورت عادلانه تقسیم کنه تا همه بتونن بدون تداخل ازشون استفاده کنن.

۲. Protection and Isolation

یکی از کارهای مهم سیستم عامله که process ها رو از هم و از kernel جدا نگه داره. این جداسازی باعث می شه اگر یه برنامه خطا کرد، بقیه برنامه ها و خود سیستم دچار مشکل نشن.

۳. Communication and Interaction

سیستم عامل ابزارهایی برای ارتباط بین process ها و همچنین بین کاربر و سیستم فراهم می کنه. این کار معمولا از طریق file system یا pipes انجام می شه.

۲) آیا وجود سیستم‌عامل در تمام دستگاه‌ها الزامی است؟ چرا؟ در چه شرایطی استفاده از سیستم‌عامل لازم است؟

نه، وجود سیستم‌عامل در همه دستگاه‌ها الزامی نیست. بعضی دستگاه‌ها مثل وسایل ساده‌ی دیجیتال (مثلا ماشین‌حساب یا کنترل تلویزیون) برنامه‌ی مشخص و ثابتی دارند و نیازی به Operating System ندارند.

اما در سیستم‌هایی که چند برنامه باید به‌صورت هم‌زمان اجرا بشن یا منابع بین چند process تقسیم بشن (مثل کامپیوتر یا گوشی هوشمند)، وجود سیستم‌عامل ضروریه تا مدیریت منابع، زمان‌بندی و امنیت برقرار بشه.

3) معماری سیستم‌عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

سیستم‌عامل XV6 یک پیاده‌سازی آموزشی از نسخه‌ی کلاسیک UNIX v6 است که برای معماری‌های x86 و RISC-V طراحی شده است. این سیستم‌عامل از نوع Monolithic Kernel بوده و ساختاری شبیه به سیستم‌عامل‌های قدیمی یونیکس دارد. در این نوع معماری، بیشتر قسمت‌های اصلی سیستم‌عامل مانند process management، file system، system calls، و user-level programs همگی درون هسته قرار دارند و به صورت مستقیم با هم در ارتباط‌اند.

دلیل استفاده از این نوع معماری در XV6 سادگی طراحی و فهم آن است. چون هدف اصلی این سیستم‌عامل آموزشی است، طراحان آن سعی کرده‌اند ساختار اصلی UNIX v6 را حفظ کنند تا دانشجویان بتوانند نحوه‌ی عملکرد اجزای مختلف سیستم‌عامل را به صورت واقعی مشاهده کنند.

همچنین در مستندات XV6 اشاره شده که فایل‌هایی مثل mmu.h و asm.h برای مدیریت حافظه و دستورات سطح پایین مربوط به معماری x86 و RISC-V به‌کار رفته‌اند. این موضوع نشان می‌دهد که سیستم‌عامل در سطح پایین به سخت‌افزار دسترسی مستقیم دارد، که از ویژگی‌های معماری Monolithic محسوب می‌شود.

(4) سیستم عامل xv6 یک سیستم تک وظیفه ای است یا چندوظیفه ای؟

XV6 یک سیستم multi-tasking است؛ یعنی می‌تونه چند process رو به‌طور هم‌زمان (در ظاهر) اجرا کنه. این کار با scheduler انجام می‌شه که به هر process نوبت اجرا می‌ده و باعث می‌شه چند برنامه به صورت هم‌زمان فعال به نظر برسد.

(5) همان‌طور که می‌دانید به‌طور کلی چندوظیفگی تعمیمی است از حالت چند برنامه‌گی. چه تفاوتی میان یک برنامه و یک پردازش وجود دارد؟

Program یک مجموعه دستور نوشته‌شده و ذخیره‌شده در حافظه‌ی پایدار (مثل فایل اجرایی) است، ولی process نمونه‌ی در حال اجرای آن برنامه در حافظه است. به عبارت دیگه، وقتی برنامه اجرا می‌شه، سیستم‌عامل برایش یک process ایجاد می‌کنه که شامل کد، داده، و فضای stack مخصوص به خودش است.

(6) ساختار یک پردازش در سیستم‌عامل XV6 از چه بخش‌هایی تشکیل شده است؟ این سیستم‌عامل به‌طور کلی چگونه پردازنده را به پردازش‌های مختلف اختصاص می‌دهد؟ در XV6 هر process شامل قسمت‌های زیر است:

- Code segment یا همان بخش دستوراتالعمل‌ها
- Data segment شامل داده‌های برنامه
- Stack برای نگهداری داده‌های موقت و فراخوانی توابع
- ساختار مدیریتی proc که اطلاعاتی مانند شناسه پردازش (PID)، وضعیت (state) و اشاره‌گر به فایل‌ها را نگه می‌دارد

سیستم‌عامل با استفاده از scheduler، پردازنده را بین process‌های مختلف تقسیم می‌کند. روش مورد استفاده معمولاً round-robin است، یعنی هر پردازش برای مدت کوتاهی روی CPU اجرا می‌شود و سپس نوبت به پردازش بعدی می‌رسد

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

سیستم‌عامل XV6 چگونه است و به طور معمول برای چه هدفی استفاده می‌شود؟

در سیستم‌عامل‌های مبتنی بر UNIX، هر فایل یا منبع ورودی و خروجی (مثل فایل، terminal، یا حتی pipe) با یک عدد مشخص شناخته می‌شود که به آن file descriptor گفته می‌شود. به عبارت ساده، file descriptor واسطه‌ای است بین process و منبعی که می‌خواهد با آن کار کند. هر process در XV6 جدولی دارد که شماره‌ی هر file descriptor را به شیء مربوط به آن (مثل فایل یا لوله) متصل می‌کند. برای مثال، شماره‌های ۰، ۱ و ۲ به ترتیب برای stdin، stdout و stderr رزرو شده‌اند.

اما pipe نوعی مکانیزم ارتباطی است که برای رد و بدل کردن داده بین دو process استفاده می‌شود. این ابزار از دو سر تشکیل شده:

- write end (سمت نوشتن)
- read end (سمت خواندن)

هر داده‌ای که از سمت write end نوشته شود، از سمت read end توسط process دیگر خوانده می‌شود. در واقع، pipe مثل یک کانال ارتباطی عمل می‌کند که اطلاعات رو از یک process به process دیگر منتقل می‌کند بدون اینکه نیاز به فایل موقت یا ذخیره‌سازی روی دیسک باشد.

در XV6 هم از pipe برای inter-process communication (IPC) استفاده می‌شود. مثلاً وقتی در shell دستوری مثل `grep txt | ls` اجرا می‌کنی، خروجی دستور `ls` از طریق pipe به ورودی دستور `grep` فرستاده می‌شود.

8) فراخوانی‌های سیستمی fork و exec در سیستم‌عامل XV6 چه عملیاتی انجام می‌دهند؟

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

در سیستم‌عامل XV6، دو فراخوانی سیستمی مهم به نام‌های `fork()` و `exec()` برای ایجاد و اجرای process های جدید استفاده می‌شوند. هرکدام از این دو فراخوانی وظیفه‌ی خاص خودش رو دارن و جدا بودنشون باعث انعطاف بیشتر سیستم می‌شه.

فراخوانی `fork()` باعث می‌شود سیستم‌عامل به `process` جدید بسازه که کپی دقیقی از `process` والد هست. این `process` جدید (فرزند) همون کد و داده‌های والد رو داره، ولی به شناسه‌ی مخصوص به خودش (PID) می‌گیره. از اینجا به بعد، هر دو `process` (والد و فرزند) به‌طور مستقل می‌تونن ادامه بدن.

والد: همون `process` اصلی است که `fork()` رو فراخوانی کرده.

فرزند: `process` جدیدی است که بعد از `fork()` ایجاد می‌شود و کپی دقیقی از والد است. هر دو `process` از این به بعد به‌طور مستقل از هم اجرا می‌شوند. مثلاً والد می‌تواند ادامه کار خودش رو انجام بده و فرزند هم به‌طور مستقل از آن کارهای خودش رو پیگیری کند.

بعد از `fork()`، معمولاً از `exec()` استفاده می‌شود. فراخوانی `exec()` برنامه‌ی فعلی رو با یه برنامه‌ی جدید جایگزین می‌کنه. یعنی محتوای حافظه‌ی `process` پاک می‌شه و فایل اجرایی جدید داخلش لود می‌شه. در نتیجه، `process` قبلی دیگه همون برنامه‌ی قبلی نیست و از این به بعد برنامه‌ی جدید رو اجرا می‌کنه.

دلیل اینکه این دو تابع جدا طراحی شدن اینه که بین اجرای `fork()` و `exec()` برنامه بتونه تنظیمات خاصی انجام بده. مثلاً ممکنه قبل از اجرای برنامه‌ی جدید، بخواد بعضی `file descriptor` ها رو ببندد یا مسیر ورودی و خروجی رو تغییر بده. اگه این دو تابع یکی بودن، این نوع کنترل روی محیط اجرا از بین می‌رفت.

اجرا و برپایی سیستم‌عامل xv6 روی شبیه‌ساز Qemu

(9) دستور `make -n` را اجرا نمایید. کدام دستور، فایل نهایی هسته را می‌سازد؟

```
yasaman@Ubuntu-24:~/Desktop/OS/CA1/codes$ make -n
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m
32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -fno
-pic -O -nostdinc -I. -c bootmain.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m
32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -fno
-pic -nostdinc -I. -c bootasm.S
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
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m
32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c
-o bio.o bio.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m
32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c
-o console.o console.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m
32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c
-o exec.o exec.c
```

```
ld -m elf_i386 -N -e start -Ttext 0 -o initcode.out initcode.o
objcopy -S -O binary initcode.out initcode
objdump -S initcode.o > initcode.asm
ld -m elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.
o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.
o proc.o sleeplock.o spinlock.o string.o swtch.o syscall.o sysfile.o sysproc.
o trapasm.o trap.o uart.o vectors.o vm.o -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
vasaman@Ubuntu-24:~/Desktop/OS/CA1/codes$
```

دستور `make -n` در واقع یک دستور شبیه‌سازی است که به شما نشان می‌دهد اگر دستور `make` اجرا شود، چه کارهایی انجام خواهد شد. این دستور، هیچ‌کدام از دستورات را در واقع اجرا نمی‌کند، بلکه تنها به شما لیستی از دستورات مورد نیاز برای ساخت پروژه نمایش می‌دهد. این امکان به شما کمک می‌کند تا بتوانید بدون اجرای واقعی دستور `make`، روند کار را بررسی کنید.

در این مثال، دستور `make -n` نشان می‌دهد که چگونه فایل‌های مختلف در پروژه `XV6` کامپایل و لینک می‌شوند تا در نهایت فایل هسته (`Kernel`) ساخته شود.

در فرایند کامپایل `XV6`، پس از این‌که تمام فایل‌های کد منبع به فایل‌های شیء (O) تبدیل شدند، باید این فایل‌ها با هم ترکیب شوند تا هسته‌ی سیستم‌عامل نهایی ساخته شود. این مرحله در واقع توسط دستور `linker` انجام می‌شود که فایل‌های شیء را به یک فایل اجرایی واحد تبدیل می‌کند و مکان دقیق این داده‌ها و توابع را در حافظه پیدا می‌کند تا `process` بتواند آن‌ها را به درستی اجرا کند.

دستورهای لایه شده در عکس قبل این کار را برای ما انجام می‌دهد.

1. ld: این دستور linker است که وظیفه‌ی ترکیب فایل‌های شیء به یک فایل نهایی را بر عهده دارد. در اینجا، فایل‌های شیء مختلفی مثل `entry.o`, `bio.o`, `console.o` و غیره که از قبل توسط دستور `make` کامپایل شده‌اند، به هم لینک می‌شوند.
2. `T kernel.ld`: این گزینه به `ld` می‌گوید که از فایل `kernel.ld` برای انجام لینکینگ استفاده کند. فایل `kernel.ld` یک اسکریپت است که تنظیمات مختلف برای چگونگی ترکیب و سازماندهی فایل‌ها در نهایت هسته را مشخص می‌کند. این تنظیمات شامل نقاط شروع (entry points)، نحوه قرارگیری بخش‌های مختلف (مثل کد و داده‌ها)، و آدرس‌های حافظه برای هسته است.
3. `kernel.o`: با این دستور، نام فایل نهایی که از ترکیب فایل‌های شیء به دست می‌آید، به `kernel` تغییر می‌کند. این فایل، همان هسته‌ی سیستم‌عامل `XV6` است.
4. `other.b: binary initcode entryother`: این قسمت از دستور به `ld` می‌گوید که بخش‌هایی از فایل‌های دیگر مانند `initcode` و `entryother` نیز در این مرحله به هسته اضافه شوند.

در نهایت، فایل `kernel` که به این روش ساخته می‌شود، همان هسته یا `Kernel` سیستم‌عامل است که می‌تواند در محیط‌های مجازی مانند `QEMU` یا به صورت مستقیم روی سخت‌افزار اجرا شود.

اضافه کردن یک متن به Boot Message

```
sh.c  x  console.c  init.c  x
init.c > main(void)
1 // init: The initial user-level program
2
3 #include "types.h"
4 #include "stat.h"
5 #include "user.h"
6 #include "fcntl.h"
7
8 char *argv[] = { "sh", 0 };
9
10 int
11 main(void)
12 {
13     int pid, wpid;
14
15     if(open("console", O_RDWR) < 0){
16         mknod("console", 1, 1);
17         open("console", O_RDWR);
18     }
19     dup(0); // stdout
20     dup(0); // stderr
21
22     for(;;){
23         printf(1, "init: starting sh\n");
24         printf(1, "Group members:\nMatin Nourian\nMohammad Shahab Sherafat\nYasaman Amou Jafar\n");
25         pid = fork();
26         if(pid < 0){
27             printf(1, "init: fork failed\n");
28             exit();
29         }
30         if(pid == 0){
31             exec("sh", argv);
32             printf(1, "init: exec sh failed\n");
33             exit();
34         }
35         while((wpid=wait()) >= 0 && wpid != pid)
36             printf(1, "zombie!\n");
37     }
38 }
```

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group members:
Matin Nourian
Yasaman Amou Jafary
Mohammad Shahab Sherafat
$ _
```


اضافه کردن قابلیت‌های جدید به کنسول

همانطور که خواسته شده بود، قابلیت ها را اضافه کردیم.

قابلیت جابه‌جایی مکان‌نما

برای این قابلیت، به تابع `consoleintr` دو کیس جدید برای چپ و راست اضافه کردیم و همچنین توابع `consputc` و `cgaputc` را تغییر دادیم که این قابلیت را پوشش دهند. همچنین به `struct input` یک `attribute` جدید اضافه کردیم به نام `real_end` که شمارنده پایان واقعی بافر فعلی را ذخیره کند. قبلا `input.e` همان پایان واقعی بود ولی الان فرق می‌کند.

```
// Move left
case KEY_LF:
    if (has_selection()) ...

    if (input.e > input.w)
    {
        input.e--;
        consputc(KEY_LF, 0);
    }
    break;

// Move right
case KEY_RT:
    if (has_selection()) ...

    if (input.e < input.real_end)
    {
        char ch = input.buf[input.e % INPUT_BUF];
        consputc(KEY_RT, ch);
        input.e++;
    }

    break;
```

قابلیت حذف آخرین کاراکتر وارد شده

برای این امر به struct input یک آرایه جدید به نام insert_order اضافه کردیم که ترتیب زمانی کاراکتر های اضافه شده را ذخیره کند و هر وقت که Ctrl+Z زدیم، آخرین کاراکتر از نظر زمانی را پاک می‌کند.

```
int insert_order[INPUT_BUF];
```

```
int max_t = -1, idx = -1;
// find index of last inserted char
for (uint i = input.w; i < input.real_end; i++)
{
    int t = input.insert_order[i % INPUT_BUF];
    if (t > max_t)
    {
        max_t = t;
        idx = (int)i;
    }
}
```

قابلیت select و copy - paste

برای این دستورات، چند attribute جدید به struct input اضافه کردیم. ابتدا sel_a و sel_b را اضافه کردیم که بتوانیم پوزیشن‌های سلکت کردن را ذخیره کنیم. سپس یک آرایه clip هم اضافه کردیم که اگر کاربر از Ctrl+c استفاده کرد، بتوانیم رشته ای که کپی شده است را در کلیپ‌بورد ذخیره سازی کنیم. همچنین تابع has_selection را هم برای این‌که بدانیم هم اکنون سلکشنی انجام شده یا نه زدیم و در حالت های مختلفی که کاربر ممکن بود پس از سلکت کردن کاری انجام دهد، از آن استفاده کردیم.

```
char clip[INPUT_BUF];
```

```
int sel_a;
int sel_b;
```

```
static inline int has_selection(void)
{
    return (input.sel_a >= 0 && input.sel_b >= 0 && input.sel_a != input.sel_b);
}
```

قابلیت‌های شما:

1. استفاده از کلیدهای جهت‌دار (Arrow Keys)

↑) Arrow Up:

این کلید برای بالا رفتن در تاریخچه دستورات استفاده می‌شود. با فشردن Arrow Up می‌توانید دستورات قبلی که وارد کرده‌اید را مرور کنید و دوباره اجرا کنید.

↓) Arrow Down:

بعد از استفاده از Arrow Up، برای برگشت به دستورهای جدیدتر یا به آخرین دستور می‌توانید از Arrow Down استفاده کنید.

2. پاک کردن صفحه ترمینال (Clear Screen)

Ctrl + L: این ترکیب کلیدی باعث می‌شود که صفحه ترمینال شما پاک شود و همه دستورات قبلی از دید پنهان شوند.

clear: می‌توانید دستور clear را هم وارد کنید تا صفحه ترمینال پاک شود

3. جستجو در تاریخچه دستورات (Search History)

برای جستجوی سریع در تاریخچه دستورات می‌توانید از Ctrl+R استفاده کنید. این دستور به شما این امکان را می‌دهد که دستورات قبلی را سریع‌تر پیدا کنید.

برنامه سطح کاربر

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

کاربرد UPROGS: مخفف User Programs است. یعنی برنامه های سطح کاربر که هرکدامشان در قالب یک فایل هستند را به makefile معرفی می کنیم. برنامه های مثل sh، ls و cat برنامه های سطح کاربر هستند.

کاربرد ULIBS: این یکی به آبجکت کتابخانه هایی که کد کاربر قرار است از آنها استفاده کند اشاره می کند. منظور از کتابخانه همان API هایی هست که برای ارتباط با kernel استفاده می شود. Systemcall ها از این دسته ارتباطات محسوب می شوند. سیستم کال هایی مثل printf و umalloc از جمله این کتابخانه ها هستند.

(11) اگر به فایل های موجود در 6xv دقت کنید، می بینید که فایلی مربوط به دستور cd، برخلاف دستوراتی مانند ls و cat، وجود ندارد و این دستور در سطح کاربر اجرا نمیشود. 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؟

دستور cd یک دستور داخلی برنامه سطح کاربر shell است؛ لذا در همان sh.c هندل می شود. دلیل این امر این است که این دستور برای تغییر دایرکتوری کاری (CWD) استفاده می شود؛ CWD جزوی از process state است و فقط خود parent process باید آن را تغییر دهد. در غیر این صورت اگر بخواهیم cd را مثل دیگر دستورات پیاده سازی کنیم، تغییر دایرکتوری روی خود پردازش shell اعمال نشده و پس از اتمام کار cd به پوشه قبلی برمی گردد. چراکه child نمی تواند process state را برای parent تغییر دهد و این کار باید در خود برنامه shell انجام شود.

قابلیت تکمیل خودکار

برای پیاده سازی این قابلیت ابتدا در فایل console.c در تابع consoleintr، با دریافت TAB، تابع wakeup را صدا میزنیم که باعث فراخوانی consoleread میشود. ولی قبل از فراخوانی wakeup حالت console که شامل مکان کرسر، مکان خواندن و مکان نوشتن را ذخیره میکنیم و console را در حالت TAB قرار میدهیم. پس از اتمام consoleread، برخی از متغیرهایی که ذخیره کرده بودیم را بازیابی میکنیم.

```
//Autocompletion
case '\t':
    input.temp_e = input.e;
    input.buf[input.e++ % INPUT_BUF] = '\t';

    input.temp_r = input.r;
    input.temp_w = input.w;
    input.temp_real_end = input.real_end;

    input.w = input.e;
    // input.real_end = input.e;
    input.is_tab_mode = 1;
    wakeup(&input.r);
    input.buf[input.e % INPUT_BUF] = '\0';
    input.e--;
    input.real_end = input.temp_real_end;

    break;
```

```

int consoleread(struct inode *ip, char *dst, int n)
{
    uint target;
    int c;

    iunlock(ip);
    target = n;
    acquire(&cons.lock);

    while (n > 0)
    {
        while (input.r == input.w)
        {
            if (myproc()->killed)
            {
                release(&cons.lock);
                ilock(ip);
                return -1;
            }
            sleep(&input.r, &cons.lock);
        }

        c = input.buf[input.r++ % INPUT_BUF];
        *dst++ = c;
        --n;

        if (c == '\n' || c == '\t')
            break;
    }

    if (c == '\t')
    {
        input.w = input.temp_w;
        input.r = input.temp_r;
    }
}

```

پس از خواندن کنسول، آن را در shell دریافت میکنیم. تابع gets را طوری تغییر میدهیم که نسبت به TAB حساس باشد و با آن مانند ENTER برخورد کند. سپس در shell بررسی میکنیم که آیا بافر شامل TAB میشود یا نه. در صورتی که شامل TAB بود، بررسی میکنیم که رشته درون بافر پیشوند کدام یک از دستورها میتواند باشد. برای حالتی که فقط پیشوند یک دستور باشد، تابع autocomplete صدا میشود. در صورتی که پیشوند بیش از یک دستور باشد، آن را لیست میکند و در غیر اینصورت هیچ تغییری نمیدهد و فقط return میکند. در هر حالت بافر را در یک متغیر temp ذخیره میکنیم که با بافر جدید مقایسه شود. این کار برای هندل کردن دوبار TAB زدن است که در صورتی که بافر تغییر کرد، flag مربوط به آن را صفر کند تا برای دفعات بعدی دستور ها را لیست نکند.

```

if (pressed_tab) {
    int instruction_num = 0;
    char **instructions = get_matching_instructions(buf, &instruction_num);

    if (instruction_num == 0) {
        TAPPRESS = 1;

        if (temp_buf)
            memmove(temp_buf, buf, nbuf);
        return 3;
    }

    else if (instruction_num == 1) {
        auto_complete(instructions[0], buf);
        TAPPRESS = 1;
        if (temp_buf)
            memmove(temp_buf, buf, nbuf);
        return 3;
    }

    else {

        if (has_multiple_choice) {

            print_instructions(instructions, instruction_num, buf);
            done_printing = 1;

            if (temp_buf)
                memmove(temp_buf, buf, nbuf);
            return 3;
        }

        else {

            has_multiple_choice = 1;

            if (temp_buf)
                memmove(temp_buf, buf, nbuf);
            return 3;
        }
    }
}

```

حال در توابع autocomple و printinstructions تابع printf را اجرا میکنیم تا دستور ها را کامل کند که خودش به ازای هر کاراکتری که میخواهد چاپ کند، consolewrite را صدا میکند.

تغییر بعدی در تابع consolewrite اعمال شده تا در صورتی که در حالت TAB قرار داریم، علاوه بر چاپ کردن حروف، آنها را از بافر shell به بافر کنسول منتقل کند. با انتقال هر کاراکتر از بافر اگر در حالت TAB بودیم، کرسر را به اندازه کاراکتر های اضافه شده جلو میبریم. حال برای اینکه به consolewrite بفهمانیم که تمام کاراکتر ها منتقل شده اند، در هربار اخر printf یک کاراکتر TAB قرار میدهیم تا با رسیدن به آن کنسول از حالت TAB خارج شود.

```

int consolewrite(struct inode *ip, char *buf, int n)
{
    int i;

    iunlock(ip);
    acquire(&cons.lock);

    if (!input.is_tab_mode)
    {
        // Normal state of echo
        for (i = 0; i < n; i++)
            consputc(buf[i] & 0xff, 0);
    }

    else
    {
        // Write from the cursor point not the beggining
        uint start = input.e;
        for (i = 0; i < n; i++)
        {
            if (buf[i] == TAB)
            {
                input.has_enter = 0;
                input.is_tab_mode = 0;
                break;
            }

            if (buf[i] == ENTER || buf[i] == NEW_LINE)
            {
                input.e = input.temp_e;
                input.has_enter = 1;
                consputc(buf[i] & 0xff, 0);
                input.buf[(start + i) % INPUT_BUF] = buf[i];
                break;
            }
        }
    }
}

```

```

    consputc(buf[i] & 0xff, 0);
    input.buf[(start + i) % INPUT_BUF] = buf[i];
}

if (input.is_tab_mode && !input.has_enter)
    input.e += n; // cursor moves right as much as the length of new order is.

if (input.real_end < input.e)
    input.real_end = input.e;
}

release(&cons.lock);
ilock(ip);

return n;

```

برای حالتی که در چندین دستور را می‌خواهیم پرینت کنیم، مکان کرسر نباید تغییر کند به همین دلیل یک بولین `has_enter` نیز تعریف کردیم که در صورتی که در انتهای هر دستور `ENTER`

وجود داشت مکان کرسر را به جای قبلی اش برگرداند . و در آخر که تمام دستورات ممکن را پرینت کردیم ، یک کاراکتر TAB میفرستیم که نشان دهیم پرینت کردن دستورات تمام شده و فلگ ها صفر شوند.

مراحل بوت سیستم عامل xv6

اجرای بوت لودر

(12) در سکتور نخست دیسک قابل بوت، محتویات چه فایلی قرار دارد؟

```
o string.o switch.o syscall.o sysfile.o sysproc.o trapasm.o trap.o uart.o vect
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
yasaman@Ubuntu-24:~/Desktop/OS/CA1/codes$
```

```
yasaman@ubuntu-24:~/Desktop/OS/CA1/codes$ make -n
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer -f
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer -f
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
```

در سیستم عامل xv6، سکتور نخست دیسک قابل بوت معمولاً شامل بوت لودر است که به نام bootblock شناخته می شود. این فایل مسئول بارگذاری سیستم عامل به حافظه است. در فرآیند بوت، سیستم عامل xv6 این فایل را از دیسک بارگذاری کرده و به حافظه منتقل می کند.

در کدهایی که در make برای کامپایل سیستم عامل xv6 اجرا می شود، به طور خاص این قسمت وجود دارد:

```
ld -m elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o
bootasm.o bootmain.o
```

در این دستور، از `ld` (لینکر) برای ساخت فایل `bootblock.o` استفاده می‌شود که باید در سکتور نخست دیسک قرار گیرد. کد `0x7C00` به `Ttext` به طور خاص نشان می‌دهد که فایل `bootblock` باید در آدرس `0x7C00` در حافظه بارگذاری شود، که این آدرس به طور سنتی برای بوتلودر در سیستم‌های قابل بوت استفاده می‌شود.

پس از کامپایل شدن، فایل `bootblock.o` به فرمت باینری تبدیل می‌شود و در نهایت به سکتور نخست دیسک منتقل می‌شود.

دستور زیر این کار را انجام می‌دهد:

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

در اینجا، `objcopy` برای تبدیل فایل `bootblock.o` به فرمت `binary` استفاده می‌شود، که آماده است تا در سکتور نخست دیسک قرار بگیرد.

سپس، با استفاده از دستور `dd`، فایل `bootblock` به دیسک منتقل می‌شود:

```
dd if=bootblock of=xv6.img conv=notrunc
```

این دستور باعث می‌شود که فایل `bootblock` در سکتور نخست دیسک در دیسک ایمج (مثل `xv6.img`) قرار گیرد. بنابراین، سیستم‌عامل `XV6` می‌تواند بوت شود.

`Bootblock` در واقع اولین قطعه کد اجرایی است که در هنگام بوت شدن سیستم از دیسک خوانده می‌شود. این کد به طور خاص برای قرار گرفتن در اولین سکتور دیسک طراحی شده و مسئول شروع فرآیند بارگذاری سیستم‌عامل به حافظه است.

وظیفه‌ی `Bootblock`:

وقتی سیستم روشن می‌شود، `BIOS` (یا در سیستم‌های مدرن‌تر `UEFI`) اولین مرحله از فرآیند بوت را آغاز می‌کند. اولین کاری که `BIOS` انجام می‌دهد، شناسایی دستگاه‌های قابل بوت است و سپس به اولین سکتور دیسک مراجعه می‌کند تا بوتلودر (`bootloader`) را بارگذاری کند.

Bootloader اولین کدی است که از دیسک خوانده می‌شود و وظیفه‌اش بارگذاری هسته (Kernel) سیستم‌عامل است. در سیستم‌عامل XV6، این کد به نام bootblock شناخته می‌شود.

ساختار و عملکرد Bootblock

1. قرار گرفتن در اولین سکتور دیسک:
Bootblock باید در اولین سکتور دیسک قرار گیرد. این سکتور اولین جایی است که سیستم پس از روشن شدن به آن مراجعه می‌کند.
2. بارگذاری فایل‌های بوت:
پس از اینکه bootblock بارگذاری شد، مسئولیت بارگذاری فایل‌های بعدی، مانند bootmain.c و bootasm.S را بر عهده دارد. این فایل‌ها به ترتیب فرآیند بوت را ادامه می‌دهند.
3. بارگذاری سیستم‌عامل:
در نهایت، پس از انجام مراحل ابتدایی، کنترل به هسته اصلی سیستم‌عامل منتقل می‌شود و سیستم‌عامل شروع به کار می‌کند.
چرا Bootblock در آدرس 0x7C00 قرار می‌گیرد؟
آدرس 0x7C00 یک آدرس خاص است که برای بوت‌لودر در سیستم‌های قدیمی‌تر رزرو شده است. هنگام بوت شدن، BIOS به طور پیش‌فرض فایل بوت را از این آدرس در حافظه بارگذاری می‌کند. بنابراین، برای اطمینان از اینکه سیستم‌عامل به درستی بوت شود، کد bootblock باید در این آدرس قرار گیرد.

13 برنامه‌های کامپایل شده در قالب فایل‌های دودویی نگهداری می‌شوند. فایل مربوط به بوت نیز دودویی است. نوع این فایل دودویی چیست؟ چرا از این نوع فایل دودویی استفاده

شده است؟ تفاوت این نوع فایل دودویی با دیگر فایل های دودویی کد xv6 چیست؟ این فایل را به زبان قابل فهم انسان(اسمبلی) تبدیل نمایید.

فایل دودویی یا Binary File در واقع فایلی است که داده‌های آن به صورت مستقیم و در قالب صفر و یک ذخیره می‌شوند، نه به صورت متنی. فایل‌های باینری معمولاً برای اجرا در سطح سخت‌افزار (مثل CPU یا BIOS) استفاده می‌شوند.

در سیستم‌عامل XV6، فایل bootblock یکی از همین فایل‌های دودویی است که به صورت مستقیم توسط BIOS بارگذاری می‌شود. این فایل در فرآیند کامپایل از فایل bootblock.o ساخته می‌شود و با دستور زیر به فرمت باینری تبدیل می‌شود:

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

فایل برنامه وقتی کامپایل می‌شود (مثلاً `bootmain.c` یا `bootasm.S`) تبدیل می‌شود به یک فایل شیء (object file) مثل `bootblock.o`.
داخل این فایل چند بخش (section) مختلف وجود دارد که هر کدام کار خودشان را دارند. مهم‌ترین‌ها اینان:

`.text` → شامل کدهای اجرایی برنامه (همون دستورهایی که CPU اجرا می‌کنه)

`.data` → شامل داده‌های ثابت یا متغیرهایی که مقدار اولیه دارند

`.bss` → شامل متغیرهایی که مقدار اولیه ندارند

`.symtab` و `.debug` → اطلاعاتی برای اشکال‌زدایی (debugging) و نمادها (symbols) که برای اجرای واقعی نیازی نیستن

در واقع، `bootblock.o` ابتدا یک فایل شیء (Object File) است که حاوی کد ماشین و متادیتا است، اما با دستور `objcopy` فقط بخش کد قابل اجرا (یعنی بخش `.text`) از آن جدا شده و به فایل `bootblock` تبدیل می‌شود که فقط شامل کد اجرایی است.

دلیل اصلی استفاده از فایل باینری (binary) در اینجا این است که BIOS در زمان روشن شدن سیستم، فقط می‌تواند اطلاعات را به صورت مستقیم از سکتور اول دیسک (Boot Sector) بخواند، نه فایل‌های سطح بالا مثل ELF یا Object file.

پس فایل bootblock باید دقیقاً به فرمتی باشد که BIOS بتواند آن را مستقیماً از دیسک بخواند و در حافظه بارگذاری کند.

به همین خاطر، این فایل از آدرس فیزیکی 0x7C00 در حافظه شروع به اجرا می‌کند (که در خروجی objdump هم دیده می‌شود):

00007c00 <start>:

این آدرس، همان آدرس استاندارد است که BIOS محتویات سکتور اول دیسک را در آن کپی می‌کند و اجرا را از آن‌جا آغاز می‌کند.

```
sh.c  console.c  bootblock.asm x  init.c
bootblock.asm
1
2  bootblock.o:      file format elf32-i386
3
4
5  Disassembly of section .text:
6
7  00007c00 <start>:
8  # with %cs=0 %ip=7c00.
9
10 .code16                # Assemble for 16-bit mode
11 .globl start
12 start:
13     cli                # BIOS enabled interrupts; disable
14     | 7c00: fa          cli
15
16     # Zero data segment registers DS, ES, and SS.
17     xorw    %ax,%ax    # Set %ax to zero
18     | 7c01: 31 c0      xor    %eax,%eax
19     movw    %ax,%ds    # -> Data Segment
20     | 7c03: 8e d8      mov    %eax,%ds
21     movw    %ax,%es    # -> Extra Segment
22     | 7c05: 8e c0      mov    %eax,%es
23     movw    %ax,%ss    # -> Stack Segment
24     | 7c07: 8e d0      mov    %eax,%ss
25
26 00007c09 <seta20.1>:
27
28     # Physical address line A20 is tied to zero so that the first PCs
29     # with 2 MB would run software that assumed 1 MB. Undo that.
30 seta20.1:
31     inb     $0x64,%al    # Wait for not busy
32     | 7c09: e4 64      in     $0x64,%al
33     testb   $0x2,%al
34     | 7c0b: a8 02      test    $0x2,%al
35     jnz     seta20.1
36     | 7c0d: 75 fa      jne     7c09 <seta20.1>
37
38     movb    $0xd1,%al    # 0xd1 -> port 0x64
39     | 7c0f: b0 d1      mov     $0xd1,%al
40     outb    %al,$0x64
41     | 7c11: e6 64      out     %al,$0x64
```

برای مشاهده‌ی محتوای فایل bootblock و تبدیل آن به زبان اسمبلی، از دستور زیر استفاده می‌کنیم:

```
objdump -S bootblock.o > bootblock.asm
```

خروجی این دستور بخشی از محتوای کد اسمبلی فایل را نشان می‌دهد. برای مثال بخشی از آن در خروجی به صورت زیر است:

```
00007c00 <start>:
    cli                                # BIOS enabled interrupts; disable
    xorw    %ax,%ax                    # Set %ax to zero
    movw    %ax,%ds                    # -> Data Segment
    movw    %ax,%es                    # -> Extra Segment
    movw    %ax,%ss                    # -> Stack Segment
```

در این بخش، Bootloader کارهای اولیه مثل غیر فعال کردن وقفه‌ها (`cli`) و آماده کردن رجیسترهای حافظه‌ای را انجام می‌دهد. سپس وارد مرحله‌ی تغییر به Protected Mode می‌شود:

```
lgdt    gdt_desc
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0
```

در این قسمت، Bootloader جدول GDT را بارگذاری کرده و CPU را از حالت ۱۶ بیتی (Real Mode) به ۳۲ بیتی (Protected Mode) منتقل می‌کند تا بتواند کدهای کرنل را اجرا کند.

در نهایت با دستور `ljmp` کنترل را به تابع `bootmain` (در فایل `bootmain.c`) می‌دهد:

```
ljmp    $(SEG_KCODE<<3), $start32
call    bootmain
```

از اینجا به بعد اجرای سیستم‌عامل وارد بخش C می‌شود.

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

در مرحله‌ی ساخت سیستم‌عامل `XV6`، بعد از اینکه فایل‌های مربوط به `Bootloader` کامپایل می‌شن، خروجی به صورت یک فایل شیء (object file) با فرمت `ELF` ساخته میشه که اسمش `bootblock.o` هست.

اما `BIOS` نمی‌تونه این نوع فایل رو بخونه، چون فرمت `ELF` اطلاعات اضافه مثل جدول سمبل‌ها، هدرها و بخش‌های مربوط به دیباگ رو داره. `BIOS` فقط می‌تونه مستقیماً ۵۱۲ بایت اول از دیسک رو بخونه و اون رو در آدرس `0x7C00` حافظه اجرا کنه.

برای همین، از دستور `objcopy` استفاده میشه تا فقط بخش اجرایی (یعنی `text`) از `bootblock.o` جدا بشه و به صورت یک فایل `binary` خالص ذخیره بشه. در واقع `objcopy` فایل `ELF` رو تبدیل می‌کنه به چیزی که فقط شامل کدهای قابل اجرا باشه و برای `BIOS` قابل خوندن باشه.

پس درواقع، `objcopy` باعث میشه فایل `bootblock` به صورت یک باینری ساده و آماده‌ی بارگذاری در سکتور اول دیسک ساخته بشه. این فایل بعداً با دستور `sign.pl` امضا میشه (با کد `0x55AA` در انتها) تا `BIOS` اون رو به عنوان یک سکتور قابل بوت تشخیص بده.

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

در `XV6`، فایل بوت از دو قسمت اصلی تشکیل شده: `bootasm.S` و `bootmain.c`. قسمت اول (`bootasm.S`) با زبان اسمبلی نوشته شده و قسمت دوم (`bootmain.c`) با

زبان C. علت این ترکیب اینه که در مراحل اولیه‌ی بوت، هنوز سیستم در حالت ابتدایی (real mode) هست و هیچ کدوم از امکانات معمولی زبان C مثل استک، سگمنت‌ها، یا دسترسی به حافظه‌ی مجازی آماده نیست.

نحوه آماده‌سازی فضای اجرایی برای C در کد اسمبلی (bootasm.S):

در سیستم‌عامل XV6، برای اینکه کد C به درستی اجرا بشه، ابتدا باید کد بوت (که با Assembly نوشته شده) کارهای اولیه را انجام بده تا محیط مناسب برای اجرای کد C آماده بشه. این کارها بیشتر شامل تنظیمات پایه‌ای سخت‌افزاری و انتقال از real mode (که سیستم در ابتدا در اون قرار داره) به 32 protected mode بیتی هست که برای اجرای کد C ضروریه.

1. غیرفعال کردن وقفه‌ها (cli)

یکی از اولین کارهایی که Assembly انجام می‌ده اینه که تمام وقفه‌ها رو غیرفعال می‌کنه. چرا؟ چون در real mode اجازه داده میشه که وقفه‌ها به صورت خودکار رخ بدن و ممکنه موجب مشکلاتی در اجرای کد بشه.

با این کار، سیستم می‌تونه بدون هیچ وقفه‌ای ادامه پیدا کنه و کنترل به کدهای بعدی داده بشه.

2. تنظیم رجیسترهای سگمنت (DS, ES, SS)

در real mode، ما به سگمنت‌ها نیاز داریم تا حافظه رو به درستی مدیریت کنیم. در اینجا، کد Assembly از registers استفاده می‌کنه تا Extra Segment (ES)، Data Segment (DS) و Stack Segment (SS) رو به درستی تنظیم کنه.

این تنظیمات باعث می‌شه که DS, ES, SS به آدرس‌های صفر (۰) تنظیم بشن تا کدهای بعدی بتونند از فضای حافظه به درستی استفاده کنند. به عبارت ساده، این کار باعث میشه که C runtime از فضای مورد نظر برای داده‌ها و استک استفاده کنه.

3. فعال کردن A20 Line

در real mode، سیستم فقط می‌تونه به ۱ مگابایت اول حافظه دسترسی داشته باشه. اما برای اینکه سیستم بتونه به حافظه بیشتر از ۱ مگابایت دسترسی پیدا کنه، باید A20 line رو فعال کنه. این خط A20 در سیستم‌های قدیمی‌تر (با حافظه 16 بیتی) غیرفعال بود.

4. انتقال از Real Mode به Protected Mode

بعد از اینکه تنظیمات اولیه انجام شد، حالا وقتشه که به 32 bit protected mode بیتی بریم. Protected mode این امکان رو می‌ده که بتونیم از حافظه بیشتر و امکانات جدیدتر پردازنده استفاده کنیم. این کار نیاز به بارگذاری جدول GDT و تنظیم CR0 داره.

در اینجا، کد Assembly جدول GDT رو بارگذاری می‌کنه و سپس CR0 رو تنظیم می‌کنه تا اجازه بده که سیستم به protected mode منتقل بشه. بعد از این کار، ما می‌تونیم از حافظه مجازی و دسترسی‌های سطح بالاتر استفاده کنیم.

5. پرش به کد 32 بیتی (Jump to 32-bit Mode)

زمانی که protected mode آماده شد، کد باید به بخش 32-bit از کرنل پرش کنه. برای این کار از دستور `ljmp` استفاده میشه که CPU رو به حالت 32-bit می‌برد.

این کار باعث میشه که اجرای سیستم‌عامل به حالت 32-bit بره و حالا فضای کافی برای اجرای کدهای C فراهم بشه.

6. شروع اجرای کد C

پس از اینکه محیط برای اجرای C آماده شد، کد به تابع `start32` می‌ره که در آن ابتدا registers و فضای داده برای C تنظیم میشه و سپس `bootmain.c` از اینجا اجرا میشه.

بعد از انجام این کارها، CPU آماده‌ی اجرای کدهای سطح بالاتری میشه و از اینجا به بعد، کنترل به فایل `bootmain.c` داده میشه. این بخش با زبان C نوشته شده و وظیفه‌ی بارگذاری فایل کرنل (kernel) از روی دیسک رو داره.

اگر این کارها فقط با زبان C نوشته می‌شد، برنامه در همان ابتدای بوت قادر به اجرا نبود، چون هنوز محیط مورد نیاز برای اجرای C وجود نداشت. اسمبلی اینجا کمک می‌کند که محیط پایه ساخته بشه تا بعد از اون، بقیه‌ی مراحل به زبان C انجام بشه.

16) یک ثبات عمومی، یک ثبات قطعه، یک ثبات وضعیت و یک ثبات کنترل در معماری x86 را نام برده و وظیفه هر یک را به طور مختصر توضیح دهید.

در معماری x86، پردازنده از انواع مختلفی از ثبات‌ها استفاده می‌کند. این ثبات‌ها مسئول ذخیره و پردازش اطلاعات در داخل پردازنده هستند. به طور کلی، می‌توانیم این ثبات‌ها را در دسته‌های مختلف قرار بدیم: ثبات‌های عمومی، ثبات‌های وضعیت، ثبات‌های کنترل.

1. ثبات‌های عمومی (General Purpose Registers):

مثلاً EAX یکی از مهم‌ترین ثبات‌های عمومی است. از این نوع ثبات‌ها برای انجام محاسبات و نگهداری داده‌ها در حین اجرای برنامه استفاده می‌شود. به عنوان مثال، وقتی پردازنده می‌خواهد دو عدد را با هم جمع کند، ممکن است عدد اول را در EAX و عدد دوم را در EBX قرار دهد و نتیجه را دوباره در EAX ذخیره کند. وظیفه: انجام عملیات ریاضی و منطقی و نگهداری موقت داده‌ها.

2. ثبات قطعه (Segment Register):

مثلاً Code Segment (CS) یکی از ثبات‌های قطعه است. این ثبات مشخص می‌کند که کد برنامه در کجای حافظه قرار دارد و پردازنده باید از چه محدوده‌ای دستورات را اجرا کند. وظیفه: مشخص کردن محل بخش‌های مختلف برنامه در حافظه مثل کد، داده یا پشته (stack).

3. ثبات وضعیت (Status Register):

مثلاً EFLAGS یا FLAGS ثبات وضعیت پردازنده است. این ثبات شامل چندین پرچم (flag) است که هرکدام وضعیت خاصی از پردازنده را نشان می‌دهند، مثل اینکه نتیجه آخرین محاسبه

صفر شده یا overflow رخ داده است.
وظیفه: نگهداری وضعیت فعلی پردازنده و نتایج عملیات منطقی و حسابی

4. ثبات کنترلی (Control Register):

مثلاً CR0 یکی از مهم‌ترین ثبات‌های کنترلی است. این ثبات برای تنظیم حالت‌های کاری پردازنده استفاده می‌شود، مثل فعال کردن Protected Mode یا کنترل کش (Cache). بدون مقدار دهی درست به این ثبات، سیستم‌عامل نمی‌تواند از امکانات پیشرفته حافظه استفاده کند.

وظیفه: کنترل رفتار و حالت‌های کاری پردازنده (مثل تغییر از real mode به protected mode).

در معماری x86، برای اینکه بدونیم پردازنده در هر لحظه در چه وضعیتی قرار دارد و محتویات registerها چیه، می‌تونیم از ابزارهایی مثل gdb و qemu استفاده کنیم.

وقتی سیستم‌عامل xv6 در حال اجراست، با استفاده از gdb می‌تونیم وارد حالت دیباگ بشیم و با دستور زیر وضعیت تمام ثبات‌ها رو ببینیم:

```
(gdb) info registers
```

این دستور مقدار همه‌ی registerهای مهم مثل EAX, EBX, ECX, EDX, EIP, ESP و بقیه رو نشون میده. با دیدن این مقادیر می‌فهمیم پردازنده دقیقاً در چه نقطه‌ای از اجرای برنامه قرار داره.

اگر داخل محیط qemu هستیم، می‌تونیم با زدن ترکیب کلید Ctrl + A و بعد C وارد ترمینال qemu monitor بشیم. اونجا هم با دستور `info registers` می‌تونیم وضعیت فعلی registerها رو مشاهده کنیم. برای برگشت به اجرای عادی xv6 هم کافیه دوباره همین ترکیب کلیدها رو تکرار کنیم.

17) پردازنده‌های x86 دارای مدهای مختلفی هستند. هنگام بوت، این پردازنده‌ها در real mode قرار دارند؛ مدی که سیستم عامل ام‌اس‌داس (MS-DOS) در آن اجرا می‌شدند. یک نقص اصلی این مد را بیان کنید. آیا در پردازنده‌های دیگر مانند ARM یا RISC-V نیز چنین مدهایی وجود دارد؟ توضیح دهید.

پردازنده‌های x86 وقتی سیستم روشن می‌شود، در حالت real mode شروع به کار می‌کنند. این حالت به مدل خیلی ساده و قدیمی که از زمان MS-DOS باقی مانده. توی real mode، پردازنده فقط به حدود ۱ مگابایت حافظه دسترسی دارد و آدرس‌دهی هم به صورت segment:offset انجام می‌شود، یعنی آدرس‌ها به دو قسمت تقسیم می‌شوند.

نکته‌ی مهم این است که توی real mode هیچ سیستم حفاظتی وجود ندارد. یعنی اگر به برنامه اشتباه کنه و به یه بخش از حافظه که نباید دست بزنه، ممکنه کل سیستم از کار بیفته. به خاطر همین محدودیت‌ها، real mode فقط برای شروع کار سیستم و اجرای کدهای اولیه (مثل bootasm.S) استفاده می‌شود. بعد از اون، سیستم به کمک کدهایی که توی فایل bootasm.S هست، وارد protected mode می‌شود تا بتونه از قابلیت‌هایی مثل virtual memory و multi-tasking استفاده کنه.

در مورد معماری‌های دیگه مثل ARM یا RISC-V، این حالت ساده‌ی real mode وجود نداره. این پردازنده‌ها از اول در حالت‌های پیشرفته‌تری بوت می‌شوند. مثلاً در ARM معمولاً سیستم در Supervisor mode یا EL3 اجرا می‌شود و در RISC-V هم حالت اولیه Machine mode ((M-mode هست. این حالت‌ها از همون اول امکان کنترل کامل سیستم و مدیریت حافظه رو دارن، بنابراین نیازی به تغییر حالت مثل x86 نیست.

پس در واقع، real mode به حالت ساده و قدیمی برای شروع سیستم توی x86 هست، ولی در معماری‌های جدیدتر، پردازنده از همون اول با حالت‌های کامل‌تر و امن‌تر شروع به کار می‌کنه.

18) یکی دیگر در از مدهای مهم، مد حفاظت شده می‌باشد. وظیفه‌ی اصلی این مود چیست؟ پردازنده‌ها در چه زمانی در این مود قرار میگیرند؟

Protected Mode در x86 حالت اجرایی‌ای است که سه قابلیت را فعال می‌کند: آدرس‌دهی ۳۲بیتی خطی یا همان linear addressing و segmentation با privilege levels (حلقه‌های ۰ تا ۳) و امکان paging برای حافظه مجازی حفاظت‌شده. وظیفه‌ی اصلی این مد، اجرای ایمن کد در حضور مرزهای حفاظت است. دسترسی هر حلقه (Ring 0/3) و هر بخش (code/data) را کنترل می‌کند، و وقتی paging را هم روشن کنیم، هر صفحه حافظه read/write/exec مستقل و جداگانه سیاست‌گذاری می‌شود. Protected Mode یعنی سخت‌افزار امکان enforce کردن مرزهای حافظه و سطح دسترسی را در اختیار OS می‌گذارد. بدون آن، OS نمی‌تواند امنیت و پایداری معقولی ارائه کند.

وظایف اصلی آن عبارتند از: 1. حفاظت و تفکیک حافظه: هر پردازنده فقط به صفحات/سگمنت‌های خودش دسترسی دارد؛ دسترسی خارج از محدوده یا نوشتن روی ناحیه‌های ممنوع خطا می‌دهد. این همان چیزی است که جلوی خراب‌کردن کل سیستم توسط یک برنامه‌ی معیوب را می‌گیرد. در خود جزوه هم بلافاصله بعد از ورود به این مد، نگاشت آدرس منطقی برنامه به آدرس فیزیکی از طریق جدول‌های صفحه را توضیح داده است. 2. تفکیک سطوح امتیاز (Ringها): هسته روی Ring0 و برنامه‌های کاربر روی Ring3 اجرا می‌شوند؛ دستورهای خاص سخت‌افزار فقط در سطح کرنل مجازند. (همان چیزی که باعث می‌شود یک برنامه‌ی کاربر نتواند مستقیم به دستگاه‌ها/کنترل‌رجیسترها دست بزند). 3. آدرس‌دهی پیشرفته: آدرس منطقی → آدرس خطی، و در صورت فعال‌بودن صفحه‌بندی، آدرس خطی → آدرس فیزیکی (Paging) است. 4. مدیریت امن وقفه/استثنا: به کمک IDT و Gateها انتقال کنترل از/به کرنل با رعایت سطح امتیاز انجام می‌شود. 5. پشتیبانی از چندوظیفگی: سخت‌افزار x86 سازوکار TSS و سوئیچ وظیفه را دارد (هرچند اغلب OSهای جدید، Context Switch را نرم‌افزاری انجام می‌دهند).

پردازنده چه زمانی وارد Protected Mode می‌شود – پس از ریست، CPU در Real Mode با آدرس‌دهی ۱۶-بیتی شروع می‌کند. BIOS boot sector را لود می‌کند و کنترل را به bootloader می‌سپارد. در xv6 این bootloader در فایل bootasm.S ابتدا پیش‌نیازها را فراهم می‌کند. بوت‌لودر بعد از شروع در Real Mode با انجام یک سری مراحل لود کردن وارد Protected Mode می‌شود و آنجا DS/ES/SS و استک موقت را تنظیم می‌کند. از آن‌به‌بعد، هم کرنل و هم برنامه‌های کاربر در Protected Mode باقی می‌مانند. جابه‌جایی بین کاربر/هسته با وقفه، استثنا یا فراخوان سیستمی انجام می‌شود، اما خروج از Protected Mode لازم نیست. در واقع بازگشت به Real Mode بعد از بالا آمدن OS تقریباً انجام نمی‌شود. (مگر در بوت‌لودرهای قدیمی برای استفاده از BIOS).

نکته مهم این است که بدانیم که Protected Mode با Paging یکی نیست: می‌توانیم Protected Mode را بدون فعال‌کردن Paging داشته باشیم (فقط با Segmentation). اگرچه که در اکثر OS‌های امروزی هر دو فعال‌اند.

19) کد bootmain.c، هسته را با شروع از یک سکتور پس از سکتور بوت، خوانده و در آدرس 100000x0 قرار می‌دهد. علت انتخاب این آدرس چیست؟ چرا این آدرس از 0 شروع نشده است؟

bootmain.c پس از آن‌که bootasm.S پردازنده را به Protected Mode و 32بیتی درآورده، هدر ELF هسته را در 0x10000 می‌خواند، سپس هر program segment را دقیقاً در نشانی فیزیکی‌ای که داخل هدر تعیین شده (paddr) می‌گذارد و در پایان به entry می‌پرد؛ در همین کد می‌بینیم که خواندن دیسک از 1 sector آغاز می‌شود و paddr هر سگمنت از خود ELF برداشته می‌شود، نه این‌که صفر یا هر نشانی دلخواه دیگری باشد.

این‌که چرا 0x00100000؟ دو دلیل اصلی دارد. نخست، این نشانی مرز کلاسیک آغاز extended memory در معماری PC است و از ناحیه‌های پرریسک حافظه پایین (low memory) دور می‌ماند: ابتدای حافظه برای ساختارهای BIOS و خود bootloader استفاده شده و کرنل اگر در پایین حافظه بارگذاری شود ممکن است همان کدی را که هنوز در حال اجراست یا بافرهای بوت را تخریب کند. انتخاب 1MB این ریسک را حذف می‌کند و فضای کافی و یکپارچه برای بارگذاری

کرنل می‌دهد. دوم، خود paddr هسته در زمان لینک (kernel.ld) روی 0x00100000 تنظیم شده است؛ بنابراین bootmain.c موظف است سگمنت‌ها را دقیقاً در همین آدرس‌های فیزیکی بریزد و بعد به entry بپردازد. می‌دانیم که paddr کرنل در لینک‌اسکریپت تعیین می‌شود و به همین دلیل بارگذاری از 0x00100000 انجام می‌شود، نه از ۰.

از نظر روند بوت نیز منطقی است: بوت‌سکتور در سکتور ۰ قرار دارد و loader باید هسته را یک سکتور بعد از روی دیسک بخواند. readseg دقیقاً همین را با 1+offset انجام می‌دهد. پس دنباله کلی این است: sector 1 ← خواندن هدر ELF در 0x10000 ← کپی هر سگمنت در paddrهای تعیین‌شده (با شروع متعارف از 0x00100000) ← پرش به entry. این طراحی باعث می‌شود گذار بعدی هسته به paging و نگاشت‌های مجازی کرنل ساده و بدون جابه‌جایی ثانویه باشد (پس از Protected Mode، نگاشت آدرس‌ها به وسیله جدول‌های صفحه برقرار می‌شود).

اشکال زدایی

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

از دستورات `info breakpoints` یا `info b` استفاده میکنیم. این دستورات همه breakpoint ها را با جزئیات کامل، اعم از آنها که روی یک تابع یا آنها که روی یک خانه از حافظه هستند را نشان میدهند. ممکن است برای یک تابع چند تا breakpoint مختلف ایجاد شود. چرا که هر تابع ممکن است چند جای متفاوت خوانده شود.

```
shahab@shahab-VMware-Virtual-Platform: ~/Desktop/OS/OS/CA1/codes
(gdb) info breakpoints
Arguments must be numbers or '$' variables.
(gdb) info breakpoints
Num      Type           Disp Enb Address      What
1        breakpoint    keep y   0x80104380 in main at main.c:20
2        breakpoint    keep y   <MULTIPLE>
2.1      breakpoint    keep y   0x801005d0 in consputc at console.c:234
2.2      breakpoint    keep y   0x8010086d in consputc at console.c:917
2.3      breakpoint    keep y   0x80100901 in consputc at console.c:93
2.4      breakpoint    keep y   0x80100965 in consputc at console.c:227
2.5      breakpoint    keep y   0x801009ac in consputc at console.c:227
2.6      breakpoint    keep y   0x801009d9 in consputc at console.c:227
2.7      breakpoint    keep y   0x80100a01 in consputc at console.c:227
2.8      breakpoint    keep y   0x80100a50 in consputc at console.c:227
2.9      breakpoint    keep y   0x80100a9e in consputc at console.c:227
2.10     breakpoint    keep y   0x80100ada in consputc at console.c:227
2.11     breakpoint    keep y   0x80100b03 in consputc at console.c:227
2.12     breakpoint    keep y   0x80100b3d in consputc at console.c:227
2.13     breakpoint    keep y   0x80100b90 in consputc at console.c:227
2.14     breakpoint    keep y   0x80100cd0 in consputc at console.c:227
2.15     breakpoint    keep y   0x80100cef in consputc at console.c:227
2.16     breakpoint    keep y   0x80100d0e in consputc at console.c:227
2.17     breakpoint    keep y   0x80100d2b in consputc at console.c:227
2.18     breakpoint    keep y   0x80100db5 in consputc at console.c:227
```

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

از تابع `delete` استفاده میکنیم. اگر بدون آرگومان صدایش بزنیم، کل breakpoint ها پاک میشوند و اگر بخواهیم میتوانیم Num مربوط به breakpoint موردنظرمان را به آن پاس میدهیم.

۲۲) دستور bt را اجرا کنید. خروجی آن چه چیزی را نشان میدهد؟

وقتی داخل اجرای برنامه gdb به breakpoint برسد، برنامه متوقف شده و در ترمینال gdb میتوانیم دستور بدهیم. اگر دستور bt را وارد کنیم، که مخفف backtrace است، مسیر اجرای برنامه از اول تا جایی که به آن breakpoint رسیدیم را چاپ میکند. برای مثال در تابع consoleintr که مسئول گرفتن کاراکتر هاست یک breakpoint قرار دادیم و وقتی در ترمینال qemu یک کاراکتر وارد کردیم و این تابع خوانده شد، برنامه متوقف شد و در ترمینال gdb دستور bt را دادیم. مسیر خوانده شدن تابع consoleintr از اول قابل مشاهده است.

```
Thread 1 hit Breakpoint 7, consoleintr (getc=0x80103a30 <kbdgetc>)
  at console.c:452
452      acquire(&cons.lock);
(gdb) bt
#0  consoleintr (getc=0x80103a30 <kbdgetc>) at console.c:452
#1  0x80103b20 in kbdintr () at kbd.c:49
#2  0x80106e35 in trap (tf=0x801166b8 <stack+3912>) at trap.c:67
#3  0x80106b9f in alltraps () at trapasm.S:20
#4  0x801166b8 in stack ()
#5  0x80112a44 in cpus ()
#6  0x80112a40 in ?? ()
#7  0x8010435f in mpmain () at main.c:57
#8  0x801044ac in main () at main.c:37
(gdb)
```

درواقع این تابع پشته فراخوانی (Call Stack) را تابه اینجا چاپ می کند.

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

با استفاده از دستور print، با همان سینتکس زبان C می توان مقدار یک متغیر یا پوینتر را چاپ کرد.

```
(gdb) print *kpgdir
$2 = 0
```

ولی با دستور x (مخفف examine memory) می توان دقیقاً به محتوای آدرس حافظه یا یک متغیر دسترسی پیدا کرد. فرمت استفاده از این دستور به شکل زیر است:

Copy code

x/[count][format][size] address

پارامتر	معنی
count	تعداد واحدهایی که می خوای ببینی (مثلاً 4, 8, ...)
format	رشته = s، کاراکتر = c، دهدهی = d، هگز = x: نوع نمایش
size	بایت = g، بایت = w، بایت = h، بایت = b: اندازه هر واحد

در مثال زیر به متغیر input.r مربوط به struct input را چاپ میکنیم:

```
(gdb) x/1dw &input.r  
0x8010ff00 <input+128>: 31
```

یعنی یک واحد ۴بایتی را به صورت دسیمال چاپ میکند.

نکته: در تابع x باید آدرس متغیر مورد نظر را پاس داد نه خود متغیر را.

نکته: در این توابع باید متغیرهایی که پاس می‌دهیم local همان تابعی باشند که روی آن break انجام شده است. اگر نباشند ارور می‌دهد.

برای چاپ کردن مقدار یک register خاص، می‌توان از متغیر \$ استفاده کرد.

برای مثال، eax نام رجیستری است که در xv6 برای انجام عملیات محاسباتی، ذخیره مقدار بازگشتی توابع و ... استفاده می‌شود.

```
(gdb) print $eax  
$9 = 1
```

(۲۴) برای نمایش وضعیت ثبات ها از چه دستوری استفاده می شود؟ برای متغیرهای محلی چگونه؟ نتیجه این دستور را در گزارش کار خود بیاورید. همچنین در گزارش خود توضیح دهید که در معماری 86x رجیستر های edi و esi نشانگر چه چیزی هستند؟ برای نمایش وضعیت register ها از info registers استفاده می‌کنیم:

```
(gdb) info registers  
eax            0x1            1  
ecx            0x0            0  
edx            0x0            0  
ebx            0x801166b8      -2146343240  
esp            0x80116660      0x80116660 <stack+3824>  
ebp            0x8011667c      0x8011667c <stack+3852>  
esi            0x80112a40      -2146358720  
edi            0x80112a44      -2146358716  
eip            0x80100e50      0x80100e50 <consoleintr>  
eflags         0x82          [ IOPL=0 SF ]  
cs             0x8            8  
ss             0x10           16  
ds             0x10           16  
es             0x10           16
```

برای متغیر های محلی از info locals:

```
(gdb) info locals  
  
c = <optimized out>  
doprocdump = 0
```

رجیستر های ESI (Source Index) و EDI (Destination Index) برای آدرس دهی به داده ها در حافظه استفاده می شوند. یعنی CPU با استفاده از این دوتا داده را از یک خانه حافظه می خواند

```
mov esi, src
```

```
mov edi, dst
```

و در جایی دیگر می نویسد.

(۲۵) به کمک استفاده از GDB، درباره ساختار input struct توضیح دهید.

برای این کار می توانیم از دستور ptype استفاده کنیم. این دستور نشان می دهد که input چیست و چه المان هایی دارد:

```
(gdb) ptype input  
  
type = struct {  
    char buf[128];  
    uint r;  
    uint w;  
    uint e;  
    uint real_end;  
    int insert_order[128];  
    int current_time;  
    int sel_a;  
    int sel_b;  
    char clip[128];  
    int clip_len;  
    int temp_r;  
    int temp_w;  
}
```

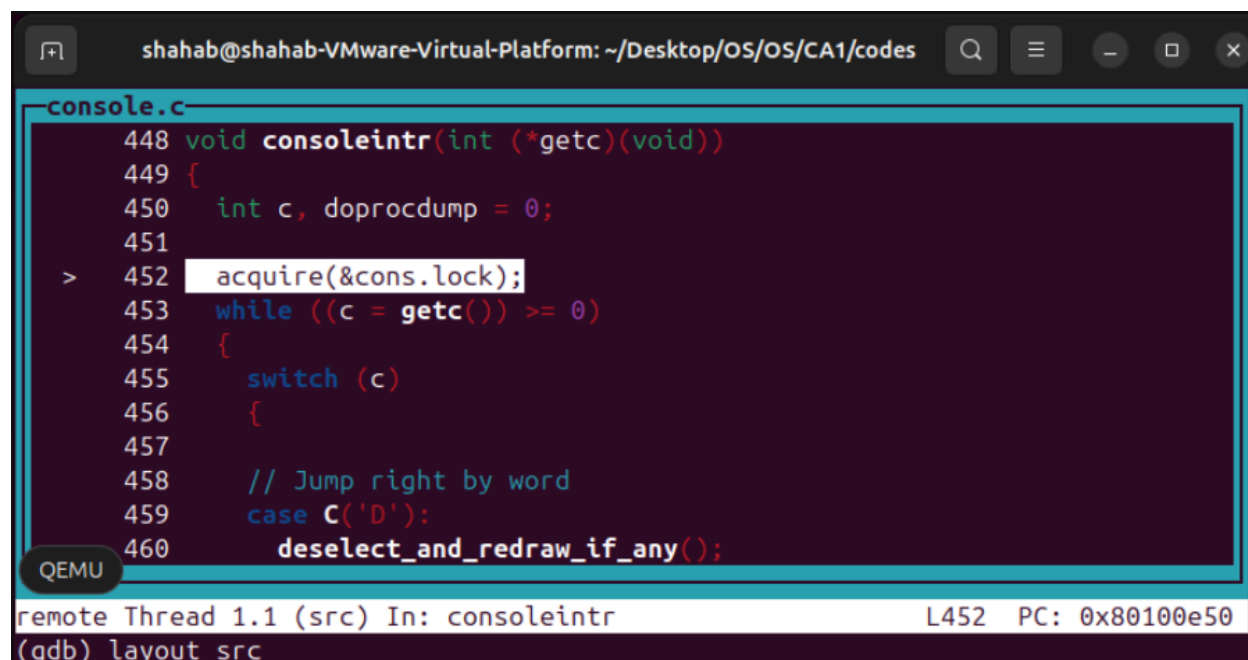
برای کاوش تغییرات یک متغیر از یک struct، می توان از دستور watch استفاده کرد و هر جا که مثلا input.e تغییر کرد، آن تغییر را مشاهده کرد. برای مثال، وقتی یک بار کلید چپ را فشار می دهیم، بدین صورت break می شود و نشان می دهد که چگونه تغییر انجام شده:

```
Thread 1 hit Hardware watchpoint 8: input.e  
  
Old value = 32  
New value = 31  
consoleintr (getc=0x80103a30 <kbdgetc>) at console.c:605  
605                 consputc(KEY_LF, 0);
```

که مشاهده می‌کنیم وقتی کلید چپ را می‌زنیم کرسر ادیت به سمت چپ می‌رود و یکی کم می‌شود.

۲۶) خروجی دستورات src layout و asm layout در TUI چیست؟

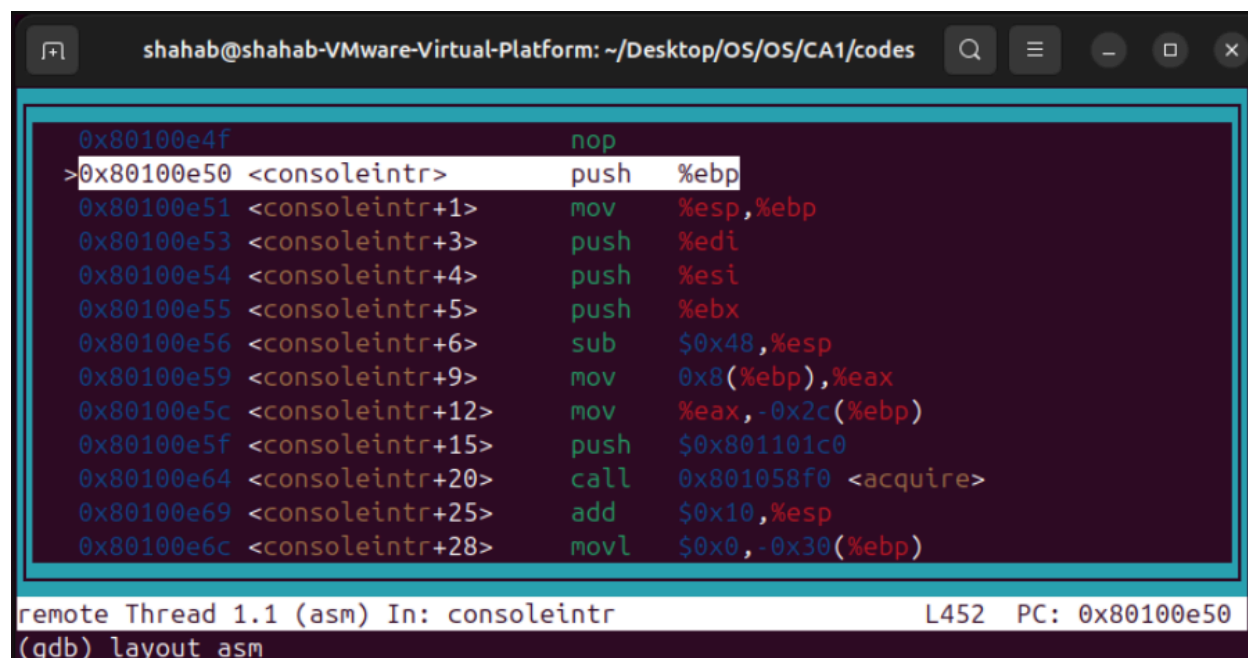
با زدن دستور src layout کد منبع، یعنی کد C سیستم‌عامل مستقیم روی صفحه می‌رود.



```
shahab@shahab-VMware-Virtual-Platform: ~/Desktop/OS/OS/CA1/codes
console.c
448 void consoleintr(int (*getc)(void))
449 {
450     int c, doprocdump = 0;
451
> 452     acquire(&cons.lock);
453     while ((c = getc()) >= 0)
454     {
455         switch (c)
456         {
457
458             // Jump right by word
459             case C('D'):
460                 deselect_and_redraw_if_any();
QEMU

remote Thread 1.1 (src) In: consoleintr L452 PC: 0x80100e50
(gdb) layout src
```

ولی با دستور asm layout کد اسمبلی مربوط به نقطه breakpoint روی صفحه می‌رود.



```
shahab@shahab-VMware-Virtual-Platform: ~/Desktop/OS/OS/CA1/codes
0x80100e4f nop
>0x80100e50 <consoleintr> push %ebp
0x80100e51 <consoleintr+1> mov %esp,%ebp
0x80100e53 <consoleintr+3> push %edi
0x80100e54 <consoleintr+4> push %esi
0x80100e55 <consoleintr+5> push %ebx
0x80100e56 <consoleintr+6> sub $0x48,%esp
0x80100e59 <consoleintr+9> mov 0x8(%ebp),%eax
0x80100e5c <consoleintr+12> mov %eax,-0x2c(%ebp)
0x80100e5f <consoleintr+15> push $0x801101c0
0x80100e64 <consoleintr+20> call 0x801058f0 <acquire>
0x80100e69 <consoleintr+25> add $0x10,%esp
0x80100e6c <consoleintr+28> movl $0x0,-0x30(%ebp)

remote Thread 1.1 (asm) In: consoleintr L452 PC: 0x80100e50
(gdb) layout asm
```

۲۷) برای جابه‌جایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستوراتی استفاده می‌شود؟

برای دیدن دستور قبلی از up و برای دیدن دستور بعدی از down استفاده میکنیم. یعنی frame دستورات از زیاد به کم چیده شده است؛ دستور جدیدتر frame کمتر دارد تا برسد به نقطه breakpoint که فریم برابر با ۰ دارد.

```
#3 0x80106b9f in alltraps () at trapasm.S:20
20      call trap
(gdb) down
#2 0x80106e35 in trap (tf=0x801166b8 <stack+3912>) at trap.c:67
67      kbdintr();
(gdb) down
#1 0x80103b20 in kbdintr () at kbd.c:49
49      consoleintr(kbdgetc);
(gdb) down
#0 consoleintr (getc=0x80103a30 <kbdgetc>) at console.c:452
452      acquire(&cons.lock);
```

همچنین می‌توان با خود دستور n frame به همان جایی که می‌خواهیم پرید و دستور موردنظر را دید.