

آزمایشگاه سیستم عامل

تمرین کامپیووتری 1

اعضای گروه :

محمد سوری - 810100249

صفورا علوی پناه - 810100254

ریحانه حاجبی - 810100116

1. معماری سیستم عامل xv6

با استناد به فصل اول کتاب، xv6 را به عنوان یک هسته سیستم عامل مونولیتیک میداند و پیاده سازی شده است، که از اکثر سیستم عامل های یونیکس الگوبرداری شده است. که به همین دلیل، در xv6 رابط kernel معادل رابط سیستم عامل است، و هسته سیستم عامل تمام وظایف و سرویسها مربوط به سیستم عامل را پیاده سازی میکند. با این حال، از آنجا که xv6 تعداد کمتری از خدمات سیستمی را ارائه میدهد، هسته آن کوچکتر از بخشی از میکرو هسته ها است. به عبارت دیگر، در xv6، تمام عملکردهای سیستم عامل، از جمله مدیریت حافظه، مدیریت فایل ها، ورود و خروج داده ها و متغیرهای سیستمی و غیره، به صورت یکپارچه در هسته پیاده سازی شده است. این معماری رابطه مستقیمی بین هسته و وظایف سیستمی دارد.

2. بخش های پردازه در xv6 و چگونگی اختصاص پردازنده به پردازه های مختلف

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

3. مفهوم file descriptor در سیستم عامل های UNIX و عملکرد pipe در xv6

مفهوم file descriptor در این سیستم عامل‌ها در واقع به یک عدد اشاره می‌کند که به کمک آن می‌تواند از یک فایل بخواند یا در آن بنویسد. به ازای هر پردازه یک جدول برای نگهداری file descriptor ها وجود دارد که باعث می‌شود این مقادیر برای پردازه‌ها به صورت خصوصی باشد و برای هر کدام از آن‌ها از مقدار 0 آغاز شود. طبق قرارداد، مقدار 0 برای `stdin`, مقدار 1 برای `stdout` و مقدار 2 برای `stderr` تعریف شده است. با توجه به اینکه file descriptor می‌تواند مربوط به یک فایل، یک دستگاه یا pipe باشد، سیستم عامل با استفاده از پیاده‌سازی file descriptor ها به این شکل، توانسته است یک interface انتزاعی برای هر کدام از این موارد ایجاد کند و همه آن‌ها را به یک شکل ببینند.

عملگر pipe 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 سیستم عامل xv6 به این صورت است که ابتدا به کمک تابع `(pipe)`, دو که به هم متصل هستند ایجاد می‌کند. سپس برای پردازه سمت چپ ابتدا بخش قابل خواندن پایپ را می‌بندد و سپس بخش قابل نوشتن آن را به عنوان `stdout` برای این پردازه قرار می‌دهد و دستور را اجرا می‌کند. برای پردازه سمت راست ابتدا بخش قابل نوشتن پایپ را می‌بندد و سپس بخش قابل خواندن آن را به عنوان `stdin` در نظر می‌گیرد و در نهایت این دستور را هم اجرا می‌کند. سپس منتظر می‌ماند تا هر 2 دستور خاتمه یابند. ممکن است دستور سمت راست پایپ شامل دستوراتی باشد که در

خود آنها نیز از پایپ استفاده شده است. در این صورت، درختی از دستورات اجرا می‌شوند. لازم به ذکر است که پردازه سمت راست تا زمانی که `stdin` به `end of file` آن به نرسیده باشد، منتظر داده جدید می‌ماند.

4. وظایف `fork` و `Exec`

یک پردازه ممکن است با استفاده از فراخوان سیستمی `fork` یک پردازه جدید ایجاد کند. `Fork` یک پردازه جدید را ایجاد می‌کند که به آن "پردازه فرزند" (child process) گفته می‌شود و دقیقاً همان محتوای حافظه‌ای را دارد که پردازه فراخواننده، که به آن "پردازه والد" (parent process) گفته می‌شود، دارد. در نهایت، فراخوان سیستمی `fork` از هر دو پردازه فراخواننده و فرزند بازمی‌گردد. در پردازه والد، فراخوان `fork` شناسه پردازه فرزند را برمی‌گرداند؛ و در پردازه فرزند، صفر را برمی‌گرداند.

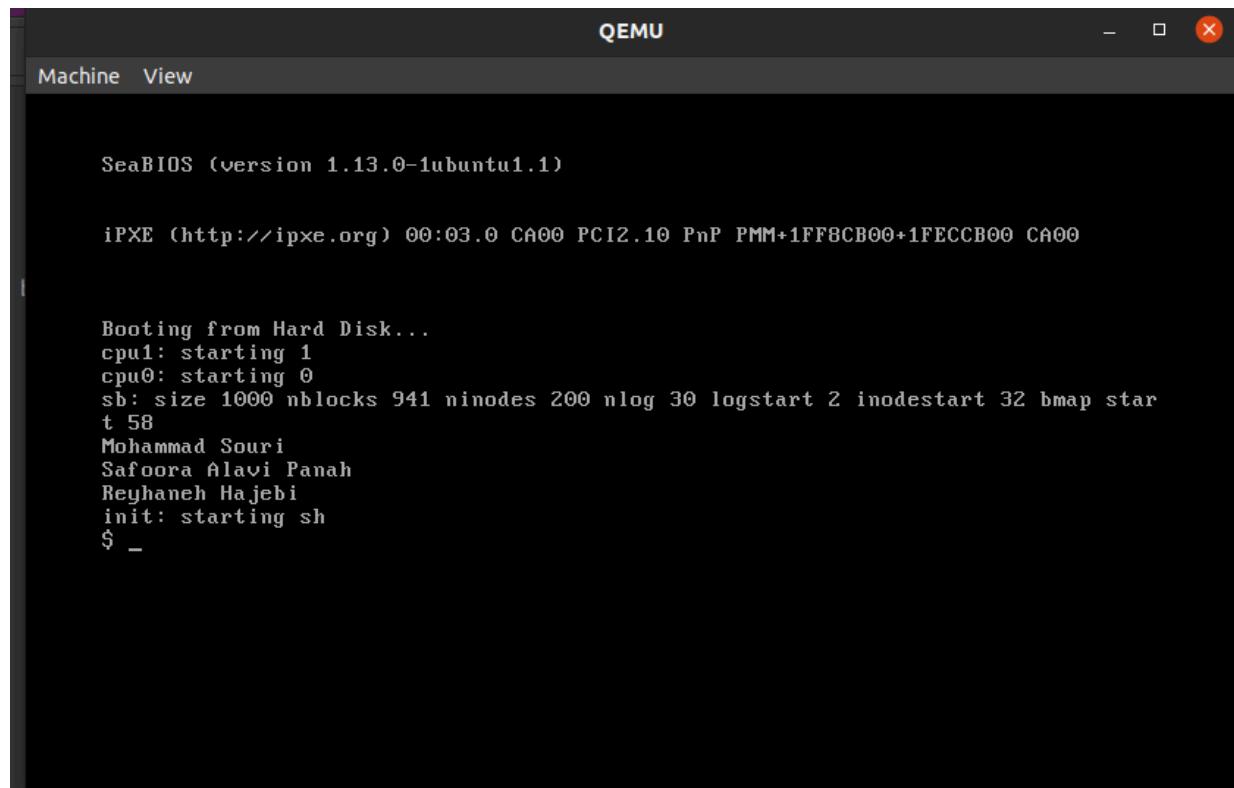
فراخوانی سیستمی `exec` حافظه پردازه فراخواننده را با یک تصویر حافظه جدید که از یک فایل در فایل سیستم بارگیری شده است، جایگزین می‌کند. این فایل باید یک فرمت خاص داشته باشد که مشخص می‌کند کدام قسمت از فایل دستورات را نگه میدارد، کدام قسمت داده‌ها را، از کجا باید اجرا آغاز شود و موارد مشابه. ELF از فرمت `xv6` استفاده می‌کند بنابراین وقتی فراخوان `exec` موفقیت آمیز باشد، به برنامه فراخواننده بازگردانده نمی‌شود؛ به جای آن، دستورات بارگیری شده از فایل از نقطه ورودی اعلام شده در هدر ELF شروع به اجرا می‌کنند. به عبارت دیگر، با استفاده از `exec`، می‌توان یک برنامه موجود

در یک فایل جایگزین برنامه فعلی پردازه کرد تا پردازه جدید با کدها و داده های موجود در فایل ادامه یابد.

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

اضافه کردن یک متن به Boot message

برای اینکار صرفا نیاز به اضافه کردن یک دستور printf به فایل init.c است که به عنوان مثال می توانید اضافه شدن نام اعضای گروه به Boot message را ببینید :



The screenshot shows a terminal window titled "QEMU" with the command "Machine View" selected. The terminal displays the following boot messages:

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

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

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
Mohammad Souri
Safoora Alavi Panah
Reyhaneh Hajebi
init: starting sh
$ _
```

اضافه کردن چند قابلیت به کنسول 6 xv6

- دستور `c shift + C` : در تعریف این دستور آمده است که به مقدار `k` از `input.buf` را کپی کند و نگه دارد.

برای این کار یک بافر اضافه به اسم `copiedBuffer` با سایز `k` در نظر میگیریم و هنگام کپی کردن در آن کپی را انجام میدهیم.

```
case 'C':  
    memset(copiedBuffer, 0, K);  
    for (int i = copyPos; i < eIndex; i++)  
        copiedBuffer[i - copyPos] = input.buf[i % INPUT_BUF];  
    pasteSize = eIndex - copyPos;  
    break;
```

برای کپی کردن کار ساده ای داریم و از موقعیت `k` تا عقب تر تا `input.e` را در بافر `copiedBuffer` میریزیم.

و برای مرحله ای که نیاز به `paste` شدن این حروف بشود `pasteSize` را آپدیت میکنیم تا مقدار درستی `paste` شود.

- دستور **x**: این دستور کاملاً شبیه به دستور **c** است و تنها تفاوت این دو این است که در دستور **cut** به مقداری که کپی انجام می‌شود باید از **input.buf** پاک شود.

```

for (int i = 0; i < pasteSize; i++)
{
    input.e--;
    consputc(BACKSPACE);
}
break;

```

- دستور **v**: این دستور برای **paste** کردن مقادیر داخل **copiedBuffer** در نظر گرفته شده است و مقادیر آن را در کنسول مینویسد.

```

case 'V':
    for (int i = 0; i < pasteSize; i++)
    {
        input.buf[input.e++ % INPUT_BUF] = copiedBuffer[i];
        consputc(copiedBuffer[i]);
    }
    break;

```

دستور Ctrl + e: در تعریف این دستور چهار موضوع ذکر شده بود:

1. اضافه شدن تمام اعداد به مقدار k و در صورت فراتر رفتن از مقدار 9 به ترتیب به حروف A, B, C ... تغییر پیدا کنند.
2. تغییر حروف کوچک به حروف بزرگ.
3. تغییر حروف بزرگ به کوچک.
4. حذف تمام علامت های غیر حروف و غیر اعداد مانند: #, @, !, %, . . .

برای انجام این کار در طی یک loop input.buf پیمایش میکنیم و سپس نسبت به جنس حروف در این آرایه تصمیم میگیریم چه عملی بر روی آن انجام بگیرد.

```
void convertCmd(int wIndex)
{
    while (wIndex < input.e)
    {
        if (input.buf[wIndex] <= '9' && input.buf[wIndex] >= '0')
            input.buf[wIndex] = incNum(input.buf[wIndex]);
        else if (input.buf[wIndex] <= 'Z' && input.buf[wIndex] >= 'A')
            input.buf[wIndex] = makeSml(input.buf[wIndex]);
        else if (input.buf[wIndex] <= 'z' && input.buf[wIndex] >= 'a')
            input.buf[wIndex] = makeCap(input.buf[wIndex]);
        else
        {
            for (int i = wIndex; i < input.e - 1; i++)
                input.buf[i] = input.buf[i + 1];
            input.e--;
            wIndex -= 1;
        }
        consputc(BACKSPACE);
        wIndex++;
    }
}
```

اما در قسمت چهارم و حذف علامات غیر حرف و عدد باید یک چیزی را در نظر بگیریم که وقتی چیزی را حذف میکنیم باید شیفت انجام شود و وقتی این موضوع انجام میشود برای میس نشدن یکی از حروف در طی این شیفت wIndex را یکی کم میکنیم تا چیزی میس نشود.

حال در این زیر شاهد توابع تمامی این convert ها هستیم:

```
char incNum(char num)
{
    int incedNum = num + K;
    if (incedNum > '9')
        incedNum = incedNum - '9' + 'A' - 1;
    return incedNum;
}

char makeSml(char c)
{
    return 'a' + c - 'A';
}

char makeCap(char c)
{
    return 'A' + c - 'a';
}
```

- دستور `tab`: در تعریف این دستور آمده است که در صورت فشردن دکمه `tab` دستوری که در کنسول نوشته شده است در صورت امکان کامل شود.

برای این کار باید 10 دستور اخیری که در کنسول نوشته شده است را ذخیره کنیم و برای مطابقت دادن دستور موجود در کنسول تلاش کنیم.

به منظور انجام این کار آرایه ای دو بعدی به اندازه [128][10] در نظر گرفته شده است با نام `history` است.

حال برای تطبیق و پیشنهاد دادن بهترین گزینه از توابع زیر استفاده میکنیم:

```

void updateCmd()
{
    cmdSize = input.e - input.w;
    memmove(command, input.buf + input.w, cmdSize);
}

findSuggestion(char cmd[INPUT_BUF], int size)
{
    int idx = -1;
    for (int i = 0; i < MAX_HIST; i++)
        if (!strcmp(cmd, history[i], size))
            idx = i;
    return idx;
}

void suggestCmd()
{
    if (!histUsed)
    {
        updateCmd();
        int suggestedCmdIdx = findSuggestion(command, cmdSize);
        if (suggestedCmdIdx != -1)
        {
            histUsed = 1;
            consclear();
            consputs(history[suggestedCmdIdx]);
        }
    }
    else
        return;
}

```

در انجام دستور tab اول از همه تابع suggestCmd فراخوانده می شود که در اول آن این موضوع که آیا تا الان دستور tab انجام شده است یا خیر مورد بررسی قرار میگیرد که در صورتی که یک دور انجام شد دیگر اجازه این فراخوانی داده نمیشود. حال در صورت اولین فراخوانی تابع updateCmd کال می شود که در آن صرفا اینپوت داخل بافر را در یک بافر جداگانه برای بررسی و جست و جو قرار میدهد. سپس در مرحله بعد تابع findSuggestion فراخواند سرچ را انجام میدهد و اندیس آخرین دستور که بیشترین تطابق را با اینپوت داشته باشد به ما برミگرداند و در صورت پیدا شدن دستور و دریافت اندیس قابل استنادی histUsed را برابر با ۱ قرار میدهیم و بعد از آن کنسول را کاملاً پاک میکنیم و بعد از آن کامند پیشنهاد شده از history را در کنسول چاپ میکنیم. یکی دیگه از نیاز های این دستور ساختن و جمع آوری history است. برای این کار تابع makeHist را کال می کنیم و مورد استفاده قرار میدهیم.

```
void makeHist()
{
    if (input.e - input.w == 1)
        return;
    memset(history[historyQueueIdx], 0, INPUT_BUF);
    memmove(history[historyQueueIdx], input.buf + input.w, input.e - input.w - 1);
    historyQueueIdx = (historyQueueIdx + 1) % MAX_HIST;
    histUsed = 0;
    memset(command, 0, INPUT_BUF);
}
```

در این تابع تمامی preprocess های مورد نیاز برای انجام دستور tab را انجام میدهیم. در وهله اول خانه بعدی آرایه history را باmemset خالی و آماده میکنیم سپس توسط memmove محتويات آخری که در input.buf بوده و enter زده شده را در آن خانه

`histUsed`. سپس `historyQueueldx` را برای `history` بعدی آماده میکند. و در آخر `tab` استفاده کنیم.

برنامه سطح کاربر :SDVAR

در این برنامه قرار بر این است که آرایه ای از اعداد با اندازه حداقلی 7 تحویل گرفته شود و با این اندازه میانگین به دو قسمت تقسیم شود و سپس برای قسمتی که از میانگین کمتر هستند انحراف معیار و برای اعداد بزرگتر واریانس گرفته شود و در نهایت در فایل sdvar_result.txt نوشته شود. کدهای این اعمال را من توانید جلوتر مشاهده بفرمایید:

```
114 int main(int argc, char *argv[])
115 {
116     if (argc < 2)
117     {
118         printf(1, "There are no input!\n");
119         exit();
120     }
121     if (argc > 8)
122     {
123         printf(1, "There are more than 7 inputs!\n");
124         exit();
125     }
126
127     char *AVStr;
128     char *MVStr;
129     char *LSDStr;
130     int avSize;
131     int mvSize;
132     int lsdSize;
133     AVStr = malloc(16);
134     MVStr = malloc(16);
135     LSDStr = malloc(16);
136     int numOfNums = argc - 1;
137     int numArray[numOfNums];
138
139     for (int i = 1; i < argc; i++)
140         numArray[i - 1] = atoi(argv[i]);
141
142     int average = calcAverage(numArray, numOfNums);
```

```
142     int average = calcAverage(numArray, numOfNums);
143
144     int k = 0, j = 0;
145     for (int i = 0; i < numOfNums; i++)
146     {
147         if (numArray[i] > average)
148             j++;
149         else
150             k++;
151     }
152     int lessThan[k];
153     int moreThan[j];
154     int x = 0, y = 0;
155     for (int i = 0; i <= numOfNums; i++)
156     {
157         if (numArray[i] > average)
158         {
159             moreThan[x] = numArray[i];
160             x++;
161         }
162         else
163         {
164             lessThan[y] = numArray[i];
165             y++;
166         }
167     }
```

```
167     AVStr = intToStr(average, &avSize);
168     MVStr = intToStr(MVariance, &mvSize);
169     LSDStr = intToStr(LDerivation, &lsdSize);
170
171     unlink("sdvar_result.txt");
172     int fd = open("sdvar_result.txt", O_CREATE | O_WRONLY);
173
174     if (fd < 0)
175     {
176         printf(1, "result_sdvar.:cannot create sdvar_result.txt\n");
177         exit();
178     }
179
180     write(fd, AVStr, avSize);
181     write(fd, " ", 1);
182     write(fd, LSDStr, lsdSize);
183     write(fd, " ", 1);
184     write(fd, MVStr, mvSize);
185     write(fd, "\n", 1);
186     close(fd);
187
188     exit();
189 }
```

8. متغیرهای Makefile و ULIB در UPROGS

متغیر UPROGS : این متغیر لیستی از برنامه‌های کاربر را دارد که در هنگام ساخت و کامپایل xv6، این برنامه‌ها نیز کامپایل و تبدیل به فایل‌های قابل اجرا توسط سیستم عامل می‌شوند. نام هر یک از این برنامه‌ها به صورت `file_name` در این لیست قرار گرفته است. تمام اسامی به صورت `file_name` (اسامی که یک _ ابتدایشان دارند)، یک هدف با پیشنازهای فایل آبجکت هدف (`file_name.o`) و متغیر ULIB دارد. بنابراین هدف‌های موجود در UPROGS منجر به ساخت فایل آبجکت برنامه‌های کاربر، اجرا شدن هدف‌های مربوط به ULIB می‌شود و در نهایت اجرای دستور `ld` می‌شود. دستور `ld` برای پیوند فایل‌های مورد نیاز و تولید یک فایل قابل اجرا مورد استفاده قرار می‌گیرد. علاوه بر آن فایل‌های آبجکت مربوط به هر برنامه (`file_name.o`) توسط یک قانون درونی Makefile ساخته می‌شوند و به صورت صریح در Makefile نوشته نشده‌اند.

متغیر ULIB : این متغیر 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 به عنوان پیشناز در قوانین قرار گرفته‌اند و در نهایت توسط دستور `ld` به فایل‌های اجرایی پیوند

منشوند. فایل‌های ULIB شامل توابعی مانند `printf`, `strcmp`, `strcpy`, `malloc` و ... هستند.

در نهایت، همانطور که از اسم این متغیرها نیز پیداست، User UPROGS معادل User Libraries است که به ترتیب برنامه‌های کاربر و ULIB Programs کتابخانه‌های کاربر محسوب می‌شوند.

11. مقایسه فایل باینری بوت با بقیه فایل‌های باینری xv6 و تبدیل آن به اسembly

در سیستم عامل xv6 ، فایلهای باینری آبجکت (object files)، از فرمت ELF (Executable Linkable Format) به عبارت دیگر (پیروی می‌کنند).

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

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

```

bootblock.o:      file format elf32-i386

Sections:
Idx Name      Size    VMA     LMA     File off  Align
0 .text       000001c3 00007c00 00007c00 00000074 2**2
               CONTENTS, ALLOC, LOAD, CODE
1 .eh_frame   000000b0 00007dc4 00007dc4 00000238 2**2
               CONTENTS, ALLOC, LOAD, READONLY, DATA
2 .comment    0000002b 00000000 00000000 000002e8 2**0
               CONTENTS, READONLY
3 .debug_aranges 00000040 00000000 00000000 00000318 2**3
               CONTENTS, READONLY, DEBUGGING, OCTETS
4 .debug_info   00000585 00000000 00000000 00000358 2**0
               CONTENTS, READONLY, DEBUGGING, OCTETS
5 .debug_abbrev 0000023c 00000000 00000000 000008dd 2**0
               CONTENTS, READONLY, DEBUGGING, OCTETS
6 .debug_line   00000283 00000000 00000000 00000b19 2**0
               CONTENTS, READONLY, DEBUGGING, OCTETS
7 .debug_str    00000221 00000000 00000000 00000d9c 2**0
               CONTENTS, READONLY, DEBUGGING, OCTETS
8 .debug_line_str 0000005c 00000000 00000000 00000fb0 2**0
               CONTENTS, READONLY, DEBUGGING, OCTETS
9 .debug_loclists 0000018d 00000000 00000000 000001019 2**0
               CONTENTS, READONLY, DEBUGGING, OCTETS
10 .debug_rnglists 00000033 00000000 00000000 0000011a6 2**0
                CONTENTS, READONLY, DEBUGGING, OCTETS

```

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

حال دستور زیر را اجرا می کنیم :

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

زمانی که این دستور را اجرا می کنیم، کد ماشینی خام از بخش "text" فایل آبجکت ورودی "bootblock.o" استخراج شده و در یک فایل باینری جدید با نام "bootblock" ذخیره

میشود. این فایل باینری میتواند به طور مستقیم توسط سخت افزار کامپیوتر بارگذاری و اجرا شود و بنابراین برای استفاده به عنوان یک بوت سکتور یا برنامه قابل بوت مورد استفاده قرار میگیرد. این جمله به این معناست که فایل "bootblock" با فرمت ELF (برخلاف بقیه فایل های باینری سیستم عامل xv6) مطابقتی ندارد و هیچ هدری در خود نمی گیرد. این فایل شامل کد اجرایی خام بدون هیچ اطلاعات اضافیای میباشد. در واقع، این فایل تنها حاوی کد ماشینی خام (raw executable code) برای اجرا میباشد و هیچ ساختار یا اطلاعات اضافی ندارد. بنابراین نوع فایل دو دویی مربوط به بوت raw binary می باشد.

دلیل اصلی این که چرا فایل "bootblock" به عنوان بوت سکتور در سیستم عامل xv6 از فرمت ELF استفاده نمی کند این است که وقتی که بوت سکتور اجرا میشود، هسته سیستم عامل هنوز اجرا نشده است و تنها پردازنده مرکزی (CPU) دارای کنترل است . CPU نمیتواند فرمت ELF را تشخیص دهد و قادر به خواندن آن نیست.

بنابراین، برای بوت سکتور، تنها کدهای ماشینی خام به CPU داده میشود. همچنین، یک دلیل دیگر برای استفاده از کدهای ماشینی خام این است که اندازه فایل باینری کاهش من یابد. با استخراج تنها بخش ".text" از فایل "bootblock" ، حجم آن کمتر میشود و در 510 بايت جا می گیرد. این امر دارای اهمیت ویژه برای بوت سکتورها است چرا که باید در 512 بايت جا شوند تا توسط BIOS به درستی بارگذاری شوند.

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

برای تبدیل bootblock به اس梅بلی، دستور

کنیم "objdump -D -b binary -m i386 -M addr16b,data16 bootblock" را اجرا می

```
bootblock:      file format binary

Disassembly of section .data:

00000000 <.data>:
0:   fa          cli
1:   31 c0       xor    %ax,%ax
3:   8e d8       mov    %ax,%ds
5:   8e c0       mov    %ax,%es
7:   8e d0       mov    %ax,%ss
9:   e4 64       in     $0x64,%al
b:   a8 02       test   $0x2,%al
d:   75 fa       jne    0x9
f:   b6 d1       mov    $0xd1,%al
11:  e6 64       out    %al,$0x64
13:  e4 64       in     $0x64,%al
15:  a8 02       test   $0x2,%al
17:  75 fa       jne    0x13
19:  b6 df       mov    $0xdf,%al
1b:  e6 60       out    %al,$0x60
1d:  0f 01 16 78 7c  lgdtw 0x7c78
22:  0f 20 c0       mov    %cr0,%eax
25:  66 83 c8 01  or    $0x1,%eax
29:  0f 22 c0       mov    %eax,%cr0
2c:  ea 31 7c 08 00  ljmp  $0x8,$0x7c31
31:  66 b8 10 00 8e d8  mov    $0xd88e0010,%eax
37:  8e c0       mov    %ax,%es
39:  8e d0       mov    %ax,%ss
3b:  66 b8 00 00 8e e0  mov    $0xe08e0000,%eax
```

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

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

"objcopy" در فرآیند "make" در xv6 برای اطمینان از اینکه مؤلفه‌های ضروری مانند بوت‌لودر و هسته در یک فرمتی باشند که به طور مستقیم توسط سخت‌افزار کامپیوتر قابل اجرا باشند، استفاده می‌شود. این فرآیند شامل حذف هدرهای ELF و تبدیل فایل‌های

باینری به فرمت باینری خام برای اجرای مستقیم توسط سخت‌افزار و همچنین کاهش اندازه فایلها (به دلیل محدودیت اندازه بوت لودر) و سادگی ساختار آنها است. این اقدام ضروری است تا بوت‌لودر و هسته بتوانند به درستی بارگذاری و اجرا شوند.

13. چرا برای بوت کردن فقط فایل C استفاده نشده و اسembly هم هست؟

چون که برخی از کارها نیازمند دسترسی سطح پایین به سیستم می‌باشند و با کد C نمی‌توان آنها را انجام داد.

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 شویم که این کار فقط در اسembly (با 1 کردن بیت اول Control Register 0 ممکن است).

14. یک ثبات عام منظوره، یک ثبات قطعه، یک ثبات وضعیت و یک ثبات کنترلی در معماری x86 را نام برد و وظیفه هر یک را به طور مختصر توضیح دهید.

ثبات عام منظوره: ثبات عام منظوره برای ذخیره داده‌های موقت داخل میکرопرոسسور استفاده می‌شوند. میکرопروپرسور 8086، 8 عدد رجیستر عام منظوره دارد. از این رجیسترها می‌توان به رجیستر شمارنده اشاره کرد. این به عنوان رجیستر شمارنده count register شناخته می‌شود. ۱۶ بیت آن به دو رجیستر ۸ بیتی تقسیم می‌شود، CH و CL، که اجازه اجرای دستورات ۸ بیتی را نیز می‌دهد. این به عنوان یک شمارنده برای حلقه‌ها عمل می‌کند و توسعه حلقه‌های برنامه را تسهیل می‌دهد. دستورات شیفت/چرخش و مدیریت رشته هر دو اجازه استفاده از count register به عنوان یک شمارنده را می‌دهند.

ثبات قطعه: در مورد 8086، چهار رجیستر قطعه وجود دارد: CS، DS، ES و SS. این‌ها به ترتیب نمایانگر Code Segment (رجیستر برش کد)، Data Segment (رجیستر برش)، Stack Segment (رجیستر برش اضافی) و Extra Segment (رجیستر برش پشتی) داده‌های اشاره می‌کنند. این رجیسترها همگی ۱۶ بیتی هستند و وظیفه انتخاب بلوك‌های (برش‌های) حافظه اصلی را دارند. به عبارت دیگر، یک رجیستر برش (مانند CS) به ابتدای یک برش در حافظه اشاره می‌کند. همان طور که گفته شده یکی از اینها CS است که به برشی از حافظه 64 اشاره می‌کند که شامل دستورات ماشینی در حال اجرا می‌باشد. با وجود محدودیت 64 کیلوبایتی برش در 8086، برنامه‌هایی که با این محدودیت در تداخل هستند می‌توانند بیشتر از 64 کیلوبایت باشند. می‌توان برش‌های مختلفی از کد را در حافظه قرار داده شود. از آنجا که می‌توان مقدار رجیستر CS را تغییر دهید، می‌توانید به برش جدیدی از کد منتقل شد و دستورات موجود در آنجا را اجرا کرد.

ثبات وضعیت: در معماری میکروپروسسور 8086، ویژگی‌های "وضعیتی" خاص وجود ندارد، مشابه ویژگی‌های معمولاً در برخی میکروپروسسورها. به جای آن، پردازنده 8086 از مجموعه‌ای از پرچم‌ها در رجیستر FLAGS (یا همان رجیستر وضعیت یا رجیستر پرچم) برای نمایش نتایج عملیات‌های مختلف و کنترل جریان برنامه استفاده می‌کند. یک نمونه از این پرچم‌ها پرچم DF است:

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

این پرچم‌ها برای انجام پرسش‌های شرطی و تصمیم‌گیری در داخل برنامه‌ها استفاده می‌شوند. برنامه‌نویسان می‌توانند این پرچم‌ها را با استفاده از دستورات پرسش شرطی برای ایجاد منطق بر اساس نتایج مختلف عملیات‌ها تست و کنترل کنند. رجیستر FLAGS که این پرچم‌ها را نگهداری می‌کند، یک رجیستر 16 بیتی است که هر پرچم یک بیت آن است.

ثبات کنترلی: پردازنده‌های مبتنی بر معماری اینتل دارای مجموعه‌ای از ثبت‌های کنترلی هستند که برای پیکربندی پردازنده در زمان اجرا (مانند تعویض بین حالت‌های اجرا) استفاده می‌شوند. این ثبت‌ها در معماری x86 به عرض 32 بیت و در معماری AMD64 (حالت بلند) به عرض 64 بیت هستند.

شش رجیستر کنترلی و یک رجیستر توانایی توسعه (EFER) وجود دارند CR0: این رجیستر شامل انواع پرچم‌های کنترلی است که عملکرد اصلی پردازنده را تغییر می‌دهند.

CR1: این رجیستر برای استفاده در آینده احتفاظ شده است.

CR2: این رجیستر شامل آدرس خطای صفحه (Page Fault Linear Address) در هنگام رخدادن خطای صفحه است.

15. کد معادل entry.s در هسته لینوکس

کد معادل entry.S برای معماری x86 در هسته لینوکس:

[قسمت بیسان هر دو](#)

[32 بیت](#)

[64 بیت.](#)

19. چرا این آدرس فیزیکی است؟

استفاده از آدرس فیزیکی از ترجمه آدرس مجازی به آدرس فیزیکی برای دسترسی به جدول صفحه ضروری است؛ زیرا استفاده از آدرس مجازی برای آن باعث ایجاد حلقه‌های بی‌پایان می‌شود و امکان دسترسی به جدول را ناممکن می‌کند. از دلایل دیگر، ایجاد جدایی و

امنیت در سیستم کامپیوتر است، که اجازه منع دهد هر پردازه مجموعه جدگانه‌ای از جداول صفحه داشته باشد و از دسترسی مستقیم به حافظه فیزیکی پردازه‌های دیگر جلوگیری کند. به علاوه، استفاده از آدرس فیزیکی از ترجمه آدرس مجازی به آدرس فیزیکی برای نرم‌افزار یک انتزاع فراهم می‌کند و مدیریت انعطاف‌پذیر حافظه و کنترل‌های امنیتی قوی را فراهم می‌کند.

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

در XV6، ترجمه آدرس‌ها برای کد و داده‌های سطح کاربر از ترجمه آدرس‌های هسته متفاوت است. تنظیم پرچم "USER_SEG" به این معناست که آدرس‌های کد و داده‌های سطح کاربر با سطح دسترسی محدودتری ترجمه می‌شوند تا از دسترسی غیرمجاز به مناطق حیاتی هسته جلوگیری شود.

23. اجزای آن در لینوکس و معادل struct proc

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
    uint sz;                                // Size of process memory (bytes)
    pde_t* pgdir;                            // Page table
    char *kstack;                            // Bottom of kernel stack for this process
    enum procstate state;                  // Process state
    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[NFILE];            // Open files
    struct inode *cwd;                     // Current directory
    char name[16];                          // Process name (debugging)
};
```

متغیر sz : حجم یا همان تعداد بایت های مموری پراسس را ذخیره می کند.

متغیر pgdir : یک پوینتر به page table مورد نیاز پراسس برای ترجمه آدرس مجازی است.

متغیر kstack : یک پوینتر از تایپ *char به kernel stack است که از آن برای ران کردن system call استفاده می شود.

متغیر state: استیت کنونی پراسس را ذخیره می کند که تایپ آن enum procstate است که در تصویر مشخص است.

متغیر pid: برای ذخیره سازی process ID ای که کرنل به این پراسس اختصاص داده است.

متغیر parent: پوینتر از همین تایپ استراکت به پراسس پدر ذخیره می کند.

متغیر tf: پوینتر از تایپ *struct trapframe که برای استیت برنامه را در حین ران شدن system call ذخیره می کند.

متغیر context: پوینتر از تایپ *struct context که دیتاها رجیسترهای برنامه را بخارط عملیات context switching ذخیره می کند.

متغیر chan: در صورتی که مقدار آن صفر نباشد، نشان می دهد که پراسس برای انجام کاری در حال wait کردن است. chan مخفف channel است.

متغیر killed: در صورتی که مقدار آن صفر نباشد، نشان می دهد که پراسس توسط سیستم عامل kill شده است.

متغیر ofile: آرایه ای با سایز 16 (NOFILE) که پوینترهایی که به فایل های open شده توسط پراسس اشاره می کنند را ذخیره می کند.

متغیر cwd: مخفف Current Working Directory است و آن را ذخیره می کند.

متغیر name: آرایه ای با سایز 16 از کاراکترها برای ذخیره اسم پراسس که صرفا برای debugging است. استراکت مشابه آن در لینوکس task_struct است که در این [لينك](#) قابل مشاهده است و همانطور که مشخص است، فیلد های بسیار بیشتری نسبت به proc در xv6 دارد.

27. بخش های از آماده سازی سیستم که در تمامی هسته های پردازنده مشترک هستند مشتمل از است؛ و بخش های اختصاصی که تنها در هسته اول اجرا می شوند شامل `Switchkvm`, `seginit`, `lalicinet`, `mpmain` `kinit1`, `kinit2`, `mpinit`, `fileinit`, `ioapicinit`, `uartinit`, `picinit`, `userinit`, `kvmalloc` (`setupkvm`), `consoleinit`, `pinit`, `tvinit`, `binit`, `ideinit`, `startothers`

توابع `mpmain` و `switchkvm` از بخش های مشترک هسته های پردازنده هستند؛ همه پردازنده ها به کمک تابع `switchkvm` آدرس `page table` را که توسط آدرس پردازنده اول ایجاد شده است را در رجیستر خود ذخیره می کنند. و دیگر اینکه تمام پردازنده ها توسط تابع `mpmain` آماده اجرای برنامه ها می شوند؛ به همین دلیل است که این توابع باید بین تمام پردازنده ها مشترک باشند.

از بخش های اختصاصی هسته اول هم می توان به تابع `ideinit` اشاره کرد که پردازنده به کمک این تابع دیسک را شناسایی می کند. همچنین، این پردازنده به کمک تابع `startothers` سایر پردازنده ها را `start` می کند؛ در هر دو مورد، دلیل این امر این است که نیازی نیست هر پردازنده این کار را انجام دهد. زمان بند توسط تابع `scheduler` انجام می پذیرد و بین تمامی هسته ها مشترک است؛ چرا که هر پردازنده باید `scheduler` مربوط به خود را داشته باشد.

روند اجرای GDB

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

```
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=
raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S -gdb tcp
::26000
[...]
QEMU [Paused]
Machine View
Guest has not initialized the display (yet).
```

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

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel...
+ target remote localhost:25000
S.gdbinit:24: Error in sourced command file:
localhost:25000: Connection timed out.
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
The target architecture is set to "i8086".
[f000:ffff] 0xfffff0: ljmp $0x3630,$0xf000e05b
0x0000ffff in ?? ()
(gdb) █
```

در این حالت دیباگر تنها روی کد سطح هسته کنترل دارد. برای مثال می‌توان روی تابع exec برک پوینت قرارداد تا وقتی اولین پردازه بخواهد بعد پردازه‌ی init شروع به کار کند، برنامه متوقف شود:

```
Remote debugging using tcp::26000
The target architecture is set to "i8086".
[f000:ffff] 0xfffff0: ljmp $0x3630,$0xf000e05b
0x0000ffff in ?? ()
(gdb) break exec
Breakpoint 1 at 0x801013c0: file exec.c, line 20.
(gdb) continue
Continuing.
The target architecture is set to "i386".
=> 0x801013c0 <exec>: push %ebp

Thread 1 hit Breakpoint 1, exec (path=0x1c "/init", argv=0x8dffffed0) at exec.c:2
0
20      struct proc *curproc = myproc();
(gdb) █
```

امباری دیاگ برنامه‌ی سطح کاربر باید به دیباگر گفته شودکه روی کد سطح کاربر کنترل انجام دهد. قبل از اینکار برک پوینت قرار داده شده روی exec را حذف می‌کنیم.

۱) برای مشاهده Breakpoint ها از چه دستوری استفاده می‌شود؟

با استفاده از دستور "breakinfo" مانند شکل زیر می‌توان برک پوینت‌ها را مشاهده کرد:

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

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

با استفاده از دستور "breakpoint_numdelete" می‌توان یک برک پوینت با شماره‌ی مشخص را حذف کرد.

برای مثال برای حذف برک پوینت شکل قبل اینگونه عمل می‌کنیم:

همانطورکه می‌بینید پس از این دستور دیگر برک پوینتی نداریم

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

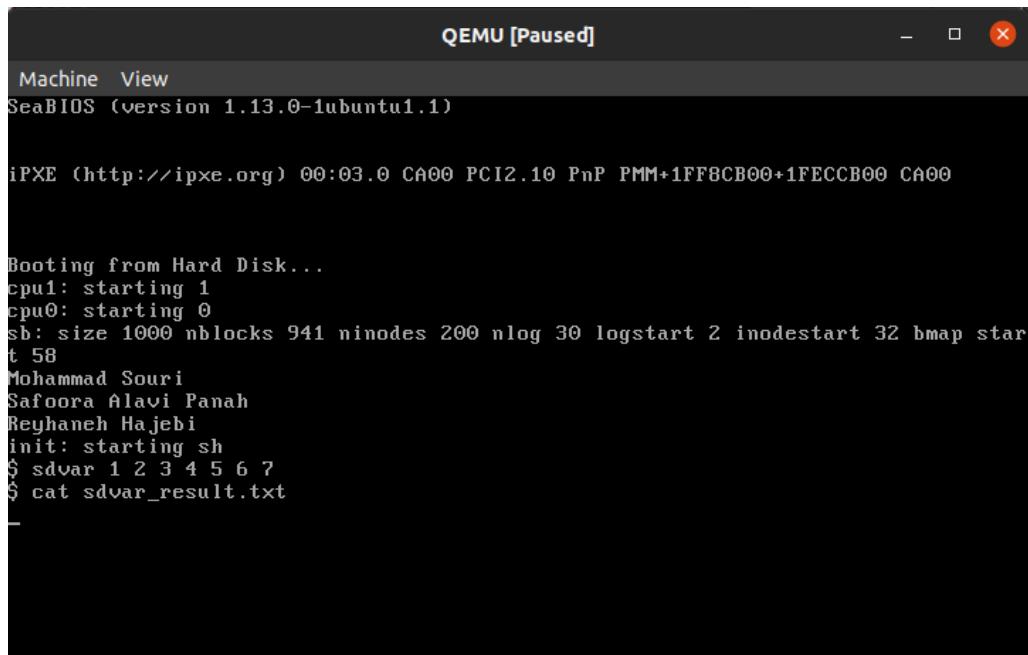
حال دیباگر را به کد سطح کاربر متصل می‌کنیم. برای این کار از دستور file استفاده می‌کنیم:

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

حال دیباگر به برنامه‌ی سطح کاربر دسترسی دارد و می‌توان در کد آن برک پوینت قرار داد:

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

اجرا را ادامه می‌دهیم و در سیستم عامل از دستور cat استفاده می‌کنیم تا بریک پوینت فعال شود:



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

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

دراین حالت می توان با اجرای دستورهای `next`, `step` و `finish` برنامه را گام به گام اجرا کرد:

• دستور `next`: به خط بعدی اجرا شده می رود اما وارد کد توابع دیگر نمی شود.

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

• دستور `step`: به خط بعدی اجرا شده می رود و در صورت نیاز وارد کد توابع دیگر هم می شود.

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

• دستور `finish`: تابع فعلی را تا زمانی که به اتمام برسد اجرا می کند.

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

۳) دستور زیر را اجرا کنید. خروجی آن چه چیزی را نشان می دهد؟

```
$bt
```

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

```
Thread 1 hit Breakpoint 2, cat (fd=3) at cat.c:12  
12          while((n = read(fd, buf, sizeof(buf))) > 0) {  
(gdb) bt  
#0  cat (fd=3) at cat.c:12  
#1  0x00000054 in main (argc=2, argv=0x2fdc) at cat.c:39  
(gdb) █
```

۴) دو تفاوت دستورهای `x` و `print` را توضیح دهید. چگونه می‌توان محتوای یک ثبات خاص را

چاپ

کرد؟ (راهنمایی: می‌توانید از دستور `help` استفاده نمایید: `x` و `print help`)

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

درحالیکه از `x` برای بررسی مقدار متغیرها و عبارات در لحظه‌ی فعلی استفاده می‌شود درحالیکه از `print` برای بررسی محتوای خام حافظه و بررسی آن در فرمت‌های مختلف استفاده می‌شود.

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

برای چاپ کردن محتوای یک رجیستر خاص هم از دستور "register_numregisters info" می‌توان استفاده

کرد:

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

۵) برای نمایش وضعیت ثبات‌ها از چه دستوری استفاده می‌شود؟ متغیرها محل چطور؟ در معماری x86 رجیسترهای `edi` و `esi` نشانگر چه چیزی هستند؟

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

```
(gdb) info registers
eax          0x3          3
ecx          0x2fd4        12244
edx          0xbfac        49668
ebx          0x2fe4        12260
esp          0x2f88        0x2f88
ebp          0x2f88        0x2f88
esi          0x2          2
edi          0x3          3
eip          0x93          0x93 <cat+3>
eflags        0x206        [ IOPL=0 IF PF ]
cs           0x1b         27
ss           0x23         35
ds           0x23         35
es           0x23         35
fs           0x0          0
gs           0x0          0
fs_base       0x0          0
gs_base       0x0          0
k_gs_base    0x0          0
cr0          0x00010011    [ PG WP ET PE ]
cr2          0x0          0
cr3          0xd0f13000    [ PDR=57107 PCID=0 ]
cr4          0x10         [ PSE ]
cr8          0x0          0
efer         0x0          [ ]
xmm0          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repe
ats 16 times>, 0x0, 0x0
, 0x0}, v2_int64 = {0x0, 0x0, uint128 = 0x0}}
xmm1          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repe
ats 16 times>, 0x0, 0x0
, 0x0}, v2_int64 = {0x0, 0x0, uint128 = 0x0}}
xmm2          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repe
ats 16 times>, 0x0, 0x0
, 0x0}, v2_int64 = {0x0, 0x0, uint128 = 0x0}}
xmm3          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repe
ats 16 times>, 0x0, 0x0
, 0x0}, v2_int64 = {0x0, 0x0, uint128 = 0x0}}
xmm4          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repe
ats 16 times>, 0x0, 0x0
, 0x0}, v2_int64 = {0x0, 0x0, uint128 = 0x0}}
xmm5          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repe
ats 16 times>, 0x0, 0x0
, 0x0}, v2_int64 = {0x0, 0x0, uint128 = 0x0}}
xmm6          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repe
ats 16 times>, 0x0, 0x0
, 0x0}, v2_int64 = {0x0, 0x0, uint128 = 0x0}}
xmm7          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x1f80000000000000}, v16_int
8 = {0x0 <repeats 12 times>, 0x0, 0x1f, 0x0, 0x0}, v2_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0
, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x1f80}, v2_int64 = {0x0, 0x1f8000000000}, uint128 = 0x1f80
, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x1f80}, v2_int64 = {0x0, 0x1f8000000000}, uint128 =
```

با استفاده از دستورهای info و registers می توان وضعیت رجیسترها و متغیرهای محلی را

مشاهده کرد:

- رجیستر edi مخفف index destination extended بوده و به عنوان اندیس مقصد در عملیات های

مربوط به رشته استفاده می شود.

- رجیستر esi هم مخفف index source extended بوده و به عنوان اندیس مبدأ در عملیات های

مربوط به رشته استفاده می شود.

۶) به کمک استفاده از GDB، درباره ساختار **input struct** موارد زیر را توضیح دهید:

- توضیح کلی این **struct** و متغیرهای درونی آن و نقش آنها

- نحوه و زمان تغییر مقدار متغیرهای درونی (برای **input.e** مثال، در چه حالتی تغییر می‌کند و چه مقداری می‌گیرد)

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

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

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

این ساختار شامل متغیرهای زیر است:

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

حالا در خطی از فایل **console.c** که کاراکتر "\n" بررسی می‌شود بروک پوینت میگذاریم (سیستم عامل و دیباگر را قبل از این مرحله روی استارت کردیم).

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

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

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

همانطور که می‌بینید در ابتدا بافر خالی است و مقادیر دیگر هم صفر هستند. برنامه را ادامه می‌دهیم و دستور `ls` را وارد کنسول می‌کنیم و اینتر را می‌زنیم. برنامه متوقف می‌شود و باز `input` را چاپ می‌کنیم:

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

با فرباریت `ls` را در خود دارد و مقادیر `e` و `end` به 2 تغییر کرده‌اند (که همان طول دستور است). با استفاده از `next` چند خط در کد جلو می‌رویم و باز `input` را چاپ می‌کنیم:

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

در این مرحله "\n" هم به بافر اضافه شده و مقادیر `w`, `e` و `end` به 3 رسیدند اما `r` همچنان صفر است. دلیل آن این است که در مرحله‌ی خواندن و اجرای دستور `w` را تا `w` جلو می‌بریم و دستور را می‌خوانیم.

باز `continue` را می‌زنیم و این بار دستور `zombie` را وارد می‌کنیم:

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

مقدار `r` برابر 3 شده. این به این معنی است که در این بین دستور `ls` با استفاده از `r` از روی بافر خوانده شده. باز دستور `zombie` را وارد می کنیم ولی این بار قبل از اینتر کرسر را به عقب می برمی:

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

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

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

(۷) خروجی دستورهای asm layout و src layout در UI چیست؟

دستور layout src سورس کد در حالت دیبیگ را به ما نشان می‌دهد و دستور layout asm سورس

asmبلی همان را به ما نشان می‌دهد:

```

remote Thread 1.1 In: mycpu
(gdb) layout src
(gdb) layout asm
(gdb) 

```

```

remote Thread 1.1 In: mycpu
(gdb) layout src
(gdb) 

```

(۸) برای جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستوراتی استفاده

می‌شود؟

```

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

```

از دستور bt برای دیدن این زنجیره و از دستورهای up و down برای جابجایی در این زنجیره استفاده

می‌شود.

Github link of project : [link](#)

Last commit message : final clean up