«به نام خدا»

گزارش کار پروژه اول آزمایشگاه سیستم عامل

گروه ۱۳:

ارشیا ابوالقاسمی - ۸۱۰۱۹۹۳۱۹

سیده زینب پیش بین - ۸۱۰۱۹۹۵۹۷

محمد جواد بشارتی - ۸۱۰۱۹۹۳۸۶


در پاسخ به: پروژه اول آزمایشگاه سیستم عامل
از محمد سعادتی در دوشنبه، 1 اسفند 1401، 2:01 عصر

با سلام

پاسخ به سوالات زیر از دو بخش اول الزامی است:
1,2,4,8,11,12,14,18,19,22,23,27
همچنین تمام سوالات بخش اشکال‌زدایی باید حل بشوند.

موفق باشید.

آشنایی با سیستم عامل xv6

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

سیستم عامل xv6 بر اساس Unix v6 پیاده سازی شده است. سیستم عامل Unix توسط Ken Thompson و Dennis Ritchie نوشته شده است. xv6 از ساختار Unix v6 استفاده می‌کند ولی با ANSI C برای پردازنده های مبتنی بر x86 پیاده سازی شده است.
از جمله دلایلی که برای دفاع از نظر خود داریم این است که در فایل x86.h می‌توانیم دستورات assembly مختص پردازنده های مبتنی بر x86 را مشاهده نماییم. در فایل asm.h نیز استفاده از معماری x86 ذکر شده است. هم چنین در فایل mmu.h از x86 memory management unit استفاده شده است. در فایل traps.h هم می‌توان trap هایی که مخصوص معماری x86 پیاده سازی شده اند را دید.

۲- یک پردازه در سیستم عامل xv6 از چه بخش هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازه های مختلف اختصاص می‌دهد؟

مطابق آنچه در صفحه ۸ کتاب آمده، یک پردازه xv6 از حافظه فضای کاربر (شامل دستورات، داده ها و پشته) و وضعیت پردازه که فقط برای هسته قابل مشاهده است، تشکیل شده است. xv6 میتواند زمان را بین پردازه ها تقسیم کند و به شکل نامحسوس از دید کاربر، به هر پردازه CPU ای از CPU های موجود را اختصاص دهد. وقتی پردازه ای در حال اجرا نباشد، xv6 مقادیر

موجود در ثبات های CPU اش را ذخیره می‌کند تا زمانی که قرار شد دوباره آن پردازه را اجرا کند، از مقادیر ثبات هایی که قبلا ذخیره کرده، استفاده نماید.
هسته xv6 به هر پردازه یک شناسه یکتا PID (Process Identifier) اختصاص می‌دهد که با استفاده از getpid() که یک فراخوانی سیستمی است، می‌توان PID پردازه در حال اجرا را یافت.

۴- فراخوانی های سیستمی exec و fork چه عملی انجام می‌دهند؟ از نظر طراحی، ادغام نکردن این دو چه مزیتی دارد؟

پردازه ها توسط فراخوانی های سیستمی متفاوتی ساخته می‌شوند که محبوب ترین آن ها exec و fork اند. در ادامه به توضیح هر یک از آن ها خواهیم پرداخت.
exec: مطابق آنچه در صفحه ۹ کتاب آمده است، فراخوانی سیستمی exec حافظه پردازه فراخواننده را با یک حافظه تصویری جدید که از یک فایل که در file system ذخیره شده است، بارگذاری می‌کند. این فایل بایستی فرمت مخصوصی داشته باشد که مشخص کند کدام قسمت فایل دستورالعمل ها را نگه داری می‌کند، کدام قسمت آن داده است، با چه دستورالعملی شروع کنیم و ... که xv6 از فرمت ELF استفاده می‌کند. زمانی که exec موفق شود (در زمان اجرای آن خطایی رخ ندهد)، به برنامه فراخواننده باز نمیگردد بلکه دستوراتی که از فایل بارگذاری شده اند، شروع به اجرای برنامه از نقطه ورودی می‌کنند که نقطه ورودی در header فایل ELF مشخص شده است. این تابع دو پارامتر ورودی دارد که پارامتر اول نام فایل برنامه و پارامتر دوم آرایه ای از آرگومان های ورودی برنامه است. می‌توان گفت exec راهی برای اجرای یک برنامه در پردازه فعلیست. قطعه کد زیر مثالی از اجرای این تابع است:

```
1 char *argv[3];
2 argv[0] = "echo";
3 argv[1] = "hello";
4 argv[2] = 0;
5 exec("/bin/echo", argv);
6 printf("exec error\n");
```

fork: مطابق آن چه در صفحات ۸ و ۹ کتاب آمده، یک پردازه ممکن است پردازه جدیدی را با استفاده از fork بسازد. پردازه جدیدی که fork می‌سازد، پردازه فرزند نامیده می‌شود که محتویات حافظه اش دقیقا با پردازه فراخواننده که همان پردازه پدر است، یکسان است. علیرغم این که محتویات حافظه پردازه فرزند و پدر یکسان است، پدر و فرزند با حافظه و ثبات های مخصوص به خودشان اجرا می‌شوند و تغییر یک متغیر در یکی از آن ها بر دیگری تأثیری ندارد. در صورت اجرای موفقیت آمیز دستور fork، در پردازه پدر، PID فرزند بازگردانده می‌شود و در پردازه فرزند، صفر. در صورت رخ دادن خطا در حین اجرای fork، در پردازه پدر ۱- برگردانده خواهد شد و پردازه فرزندی ساخته نخواهد شد.

اگر بعد از فراخوانی fork از تابع wait استفاده شود، پردازه پدر تا پایان یافتن پردازه فرزند صبر می‌کند و سپس کار خود را ادامه می‌دهد. خروجی این تابع، PID پردازه خاتمه یافته است و اگر پردازه پدر، فرزندی نداشته باشد، خروجی این تابع ۱- خواهد بود.

فراخوانی سیستمی exit باعث می‌شود که پردازه فراخواننده متوقف شده و منابعی مانند حافظه و فایل های باز آزاد شوند.

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

```
int pid = fork();
if(pid > 0){
printf("parent: child=%d\n", pid);
pid = wait();
printf("child %d is done\n", pid);
} else if(pid == 0){
printf("child: exiting\n");
exit();
} else {
printf("fork error\n");
}
```

شاید علت اصلی ادغام نکردن این دو دستور این باشد که جداسازی fork و exec این اجازه را می‌دهد که تنظیمات دلخواه محیط فرزند با استفاده از دیگر فراخوانی های سیستمی انجام شود. به عنوان مثال می‌توانیم:

- یک مجموعه دلخواه از file descriptor های باز تنظیم نماییم.
- signal mask را تغییر دهیم.
- دایرکتوری فعلی را تنظیم کنیم.
- process group ها یا session ها را تنظیم نماییم.
- کابر، گروه و گروه های تکمیلی را تنظیم کنیم.
- محدودیت های منابع سخت افزاری و نرم افزاری را تنظیم نماییم.

در صورت ادغام این دو فراخوانی، برای این که بتوانیم تمام این تغییرات احتمالی را به محیط فرزند encode کنیم یک رابط بسیار پیچیده خواهیم داشت و حتی اگر بخواهیم تنظیمات جدید اضافه کنیم، رابطمان باید تغییر کند. از سوی دیگر جدا بودن fork و exec این اجازه را می‌دهد که قبل از exec کردن از فراخوانی های سیستمی رایج مانند open، close، dup و ... برای تغییر دادن محیط فرزند استفاده کنیم.

اضافه کردن یک متن به Boot Message

برای این کار کافیست در فایل init.c خط زیر را تغییر دهیم:

```
23    printf(1, "init: starting sh\n");
```

که پس از تغییر به شکل زیر خواهد شد:

```
23    printf(1, "Group #13:\n1. Arshia\n2. Zeinab\n3. Javad\n");
```

و همانطور که در شکل زیر مشاهده می‌شود نام اعضای گروه اضافه شده است:

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
Group #13:
1. Arshia
2. Zeinab
3. Javad
$
```

اضافه کردن چند قابلیت به کنسول xv6

برای اضافه شدن قابلیت های خواسته شده بایستی در فایل console.c تغییراتی اعمال کنیم. ابتدا متغیر back_counter را برای نگه داری اندیس آخرین کاراکتر وارده شده تعریف می‌کنیم. در ادامه با توجه به این تعریف، برخی توابع xv6 را تغییر داده و تعدادی تابع جدید در این فایل اضافه می‌کنیم:

```
144   int back_counter = 0;
```

تغییرات اعمال شده در تابع cgaputc:

```
157   if(c == '\n')
158     pos += 80 - pos%80;
159   else if(c == BACKSPACE){
160     for(int i = pos - 1; i < pos + back_counter; i++){
161       crt[i] = crt[i + 1];
162     }
163
164     if(pos > 0) --pos;
165   }
166   else{
167       for(int i = pos + back_counter; i > pos; i--){
168         crt[i] = crt[i - 1];
169       }
170       crt[pos++] = (c&0xff) | 0x0700;  // black on white
171   }
```

```
186   crt[pos + back_counter] = ' ' | 0x0700;
187   //crt[pos] = ' ' | 0x0700;
188 }
```

توابع اضافه شده به فایل console.c:

از دو تابع روبرو برای دسترسی راحت تر به موقعیت cursor و تغییر راحت تر مکان آن استفاده می‌شود. که البته در خود فایل console.c هم موجود بود و فقط شکل تابع نداشت.

```
214   int
215   get_cursor_pos(){
216     int pos;
217     outb(CRTPORT, 14);
218     pos = inb(CRTPORT+1) << 8;
219     outb(CRTPORT, 15);
220     pos |= inb(CRTPORT+1);
221     return pos;
222   }
```

```
224   void
225   change_cursor_pos(int pos)
226   {
227     outb(CRTPORT, 14);
228     outb(CRTPORT+1, pos>>8);
229     outb(CRTPORT, 15);
230     outb(CRTPORT+1, pos);
231   }
232
```

```
230  static void
231  mv_cursor_backward()
232  {
233      int pos = get_cursor_pos();
234      pos--;
235      change_cursor_pos(pos);
236  }
```

```
238  static bool
239  is_at_beggening_of_line(int pos)
240  {
241      if(
242          crt[pos - DEFAULT_DIST_CURSOR_LF_WD] == (BEGIN_OF_LINE_SIGN | 0x0700)
243      ){
244          return true;
245      }else{
246          return false;
247      }
248  }
```

```
250  static bool
251  is_cursor_at_beggening_of_line()
252  {
253      return is_at_beggening_of_line(get_cursor_pos());
254  }
```

```
256  static void
257  mv_cursor_to_beginnign_of_line()
258  {
259      while(!is_cursor_at_beggening_of_line()){
260          mv_cursor_backward();
261          back_counter += 1;
262      }
263  }
```

```
266  static void
267  mv_cursor_to_end_of_line()
268  {
269    int pos = get_cursor_pos();
270    int next_pos = pos;
271
272    int change = pos % 80 - 2;
273
274    if(change == input.e){
275      return;
276    }
277
278    next_pos = input.e + pos - change;
279
280    back_counter += (next_pos - pos);
281    pos = next_pos;
282
283    change_cursor_pos(pos);
284  }
```

```
286  static bool
287  is_at_beggening_of_line_v2()
288  {
289    if(
290      input.e == input.w ||
291      input.buf[(input.e-1) % INPUT_BUF] == '\n' ||
292      is_cursor_at_beggening_of_line()
293    ){
294      return true;
295    }else{
296      return false;
297    }
298  }
```

```
299
300  static bool
301  is_at_beggening_of_word()
302  {
303    if(
304      input.buf[(input.e-1) % INPUT_BUF] == ' '
305    ){
306      return true;
307    }else{
308      return false;
309    }
310  }
```

```
312  static bool
313  can_del()
314  {
315    if(
316      is_at_beggening_of_line_v2() ||
317      is_at_beggening_of_word()
318    ){
319      return false;
320    }else{
321      return true;
322    }
323  }
```

```
325  static void
326  back_space()
327  {
328    input.e--;
329    consputc(BACKSPACE);
330
331    return;
332  }
```

```
334  static void
335  del_word()
336  {
337    while(
338      input.buf[(input.e-1) % INPUT_BUF] == ' ' &&
339      !is_at_beggening_of_line_v2()
340    ){
341      back_space();
342    }
343
344    while(can_del()){
345      back_space();
346    }
347  }
```

حال باید توابع اضافه شده را به ازای دستورات گفته شده در تابع consoleintr فراخوانی کنیم:

```
349  void
350  consoleintr(int (*getc)(void))
351  {
352    int c, doprocdump = 0;
353
354    acquire(&cons.lock);
355    while((c = getc()) >= 0){
356      switch(c){
357      case C('P'):  // Process listing.
358        // procdump() locks cons.lock indirectly; invoke later
359        doprocdump = 1;
360        break;
361      case C('U'):  // Kill line.
362        while(!is_at_beggening_of_line_v2()){
363          back_space();
364        }
365        break;
366      case C('H'): case '\x7f':  // Backspace
367        if(!is_at_beggening_of_line_v2()){
368          back_space();
369        }
370        break;
371      case SHIFT_PLUS_OPEN_BRUCKET:
372        mv_cursor_to_beginnign_of_line();
373        break;
374      case SHIFT_PLUS_CLOSE_BRUCKET:
375        mv_cursor_to_end_of_line();
376        break;
377      case C('W'):
378        del_word();
379        break;
```

اجرا و پیاده سازی یک برنامه سطح کاربر

کد برنامه خواسته شده در فایل mmm.c موجود است. فقط برای آن که برنامه کامپایل شود بایستی در قسمت EXTRA از Makefile نام برنامه را به شکل source_name.c و در قسمت UROGS از Makefile نام برنامه را به فرمت _source_name اضافه نماییم:

```
251  EXTRA=\
252      mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
253      ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
254      printf.c umalloc.c\ mmm.c\
255      README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
256      .gdbinit.tmpl gdbutil\
```

```
168  UPROGS=\
169      _cat\
170      _echo\
171      _forktest\
172      _grep\
173      _init\
174      _kill\
175      _ln\
176      _ls\
177      _mkdir\
178      _rm\
179      _sh\
180      _stressfs\
181      _usertests\
182      _wc\
183      _zombie\
184      _mmm\
```

قسمت EXTRA سورس را به دایرکتوری dist کپی می‌کند و مقدمات آماده سازی و افزودن برنامه را فراهم می‌کند و قسمت UROGS برنامه را در فایل سیستم که بعدا به qemu داده می‌شود، قرار می‌دهد تا برنامه بارگذاری شده و در دسترس قرار گیرد و کاربر بتواند دستورات را اجرا نماید.

از آنجایی که برای سیستم عامل برنامه مینوسیم، ممکن است برخی توابع استاندارد زبان C را در اختیار نداشته باشیم. تابع atoi به صورت پیشفرض در اختیارمان هست ولی برای آن که بتوانیم اعداد منفی را handle کرده و همچنین از کاراکتر های غیر عددی را تشخیص دهیم، به ترتیب از توابع atoi_neg و int_validation استفاده کردیم:

```
19  int atoi_neg(const char* str) {
20      int sign = 1;
21
22      if (*str == '-') {
23          sign = -1;
24          ++str;
25      }
26      else if (*str == '+') {
27          ++str;
28      }
29
30      int_validation(str);
31
32      return sign * atoi(str);
33  }
```

```
5   static void
6   int_validation(const char *str)
7   {
8       while(*str){
9           if((*str < '0') || (*str > '9')){
10              printf(1, "Invalid Input! Fuck you idiot\n");
11              exit();
12          }
13          ++str;
14      }
15
16      return;
17  }
```

نکته دیگر این است که تابع printf کمی با آنچه در C استاندارد بود، تفاوت دارد و بایستی شناسه محلی را نیز به آن pass داد. منظور از شناسه محلی، اندیس پردازه است که در کدام آدرس (خروجی استاندارد (stdout) یا ارور استاندارد (stderr)) بنویسد. و می‌دانیم که اندیس خروجی استاندارد 1 و اندیس ارور استاندارد 2 است.
نکته قابل توجه این است که شناسه محلی لزومی ندارد که به خروجی های ترمینال منتهی شود و می‌تواند مربوط به یک فایل هم باشد. همچنین می‌توان در کد تابع printf مشاهده کرد که در مراحل پایین تر فراخوانی سیستمی write را انجام می‌دهد و فقط رابط بهتری در اختیارمان می‌گذارد.

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
Group #13:
1. Arshia
2. Zeinab
3. Javad
$ mmm 8 2 8 4 2 3
$ cat mmm_result.txt
4 3 2
$ _
```

کامپایل سیستم عامل xv6

۸- در Makefile متغیر هایی به نام UPROGS و ULIB تعریف شده است. کاربرد آن ها چیست؟

UPROGS: این متغیر لیستی از برنامه‌های کاربر دارد که در حین ساخت و کامپایل xv6، این برنامه‌ها نیز کامپایل و تبدیل به فایل‌های قابل اجرا توسط سیستم عامل می‌شوند. هر برنامه‌ای که در این لیست باشد، نامش به فرمت _file_name است. تمامی اسامی با این فرمت یک هدف با پیشنیاز های فایل شیء هدف (file_name.o) و متغیر ULIB دارد. در نتیجه اهداف موجود در UPROGS باعث ساخته شدن فایل شیء برنامه‌های کاربر، اجرای اهداف مربوط به ULIB و در نهایت اجرای دستور ld می‌شود. دستور ld برای پیوند زدن فایل‌های مورد نیاز و تولید یک فایل قابل اجرا مورد استفاده قرار می‌گیرد. هم چنین فایل‌های شیء مربوط به هر برنامه (file_name.o) توسط یک قانون درونی Makefile ساخته می‌شوند و به شکل صریح در Makefile

نوشته نشده‌اند. هم چنین اگر بخواهیم برنامه سطح کاربری بنویسیم و به سیستم عامل اضافه کنیم، بایستی نامش را در این قسمت قرار دهیم.

ULIB: شامل برخی از کتابخانه‌های زبان C است. در بسیاری از کد های xv6 از توابع این کتابخانه‌ها استفاده شده و برای اجرای آن‌ها به کامپایل این فایل‌ها نیاز است. به عوان مثال برنامه‌های سطح کاربر به کامپایل فایل‌های ULIB نیاز دارند. فایل‌های ULIB شامل توابعی مانند strcmp، strcpy، printf، malloc و ... هستند.

مراحل بوت سیستم عامل xv6

۱۱- برنامه‌های کامپایل شده در قالب فایل‌های دودویی نگه داری می‌شوند. فایل مربوط به بوت نیز دودویی است. نوع این فایل دودویی چیست؟ تفاوت این نوع فایل دودویی با دیگر فایل‌های دودویی کد xv6 چیست؟ چرا از این نوع فایل دودویی استفاده شده است؟ این فایل را به زبان قابل فهم انسان (اسمبلی) تبدیل نمایید. (راهنمایی: از ابزار objdump استفاده کنید. باید بخشی از آن مشابه فایل bootasm.S باشد.)

همه فایل‌های دودویی شیء xv6 به فرمت ELF اند. در ابتدای این فرمت دودویی هدر هایی شامل اطلاعات بارگذاری فایل نوشته شده است و سپس چندین بخش دارد که هر یک حجمی از کد یا داده‌اند در آدرس مشخصی از حافظه بارگذاری می‌شوند. از بخش‌های یک فایل به فرمت ELF می‌توان به .data، .rodata، .text و .bss اشاره کرد.

- .text. شامل دستورات قابل اجرای برنامه است.
- .rodata. شامل داده‌های read-only از جمله string literal ها در زبان C است.
- .data. شامل داده‌های مقدار دهی شده مانند برخی متغیر های سراسری است.
- .bss. شامل داده‌های مقدار دهی نشده است که چون داده‌ای وجود ندارد، فقط آدرس و اندازه اش در فایل ذخیره می‌شود.

به وسیله دستور objdump -h bootblock.o می‌توان نوع فایل دودویی را مشاهده کرد. هم چنین در ادامه خروجی دستور بخش‌های ELF قابل مشاهده اند. boot loader پس از بارگزاری در آدرس ثابت 0x7C00، توسط پردازنده اجرا می‌شود. تا هسته را اجرا کند. تنها اطلاعات مهم در این قسمت کدیست که قرار است اجرا شود. با مقایسه bootblock.o با دیگر object file ها مشاهده می‌شود که بخش‌های .data. و غیره را ندارد و بخش اصلی‌اش فقط .text. است. از آنجا که بخش .txt. فایل bootblock دودویی خالص است، پس از فرمت ELF پیروی نمی‌کند و هیچ هدری هم ندارد. این فایل با دیگر فایل‌های دودویی xv6 متفاوت است و درواقع یک کد قابل اجرای خالص بدون هیچ اطلاعات اضافه ایست.

پس نوع فایل دودویی مربوط به بوت raw binary است و از آنجا که به فرمت ELF نیست، با دیگر فایل‌های دودویی متفاوت است.

علت استفاده نکردن از ELF برای bootblock این است که فرمت ELF را هسته سیستم عامل می‌داند و نه CPU. در نتیجه وقتی هنوز هسته اجرا نشده است، نمی‌توان فرمت ELF را خواند. اگر BIOS فایل bootblock.o را برای بوت شدن به CPU می‌داد، از آنجا که CPU هدر های ELF را نمی‌شناسد، همه محتوای فایل را به دید instruction ها نگاه می‌کند و برداشت اشتباهی از آن خواهد کرد. پس باید فقط دستورات خالص را به CPU داد. یک دلیل دیگر هم کم کردن حجم فایل است. با استخراج بخش text. فایل bootblock.o، حجم آن کم می‌شود و در 510 بایت جا می‌گیرد.

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

objdump -D -b binary -m i386 -Maddr16,data16 bootblock

```
javad@Javad-Bshrt:~/My Folders/University/6th Term/OS/Labs/lab1/xv6-public$ objdump -D -b binary -mi386 -Maddr16,data16 bootblock

bootblock:     file format binary


Disassembly of section .data:

00000000 <.data>:
   0:   fa                      cli
   1:   31 c0                   xor    %ax,%ax
   3:   8e d8                   mov    %ax,%ds
   5:   8e c0                   mov    %ax,%es
   7:   8e d0                   mov    %ax,%ss
   9:   e4 64                   in     $0x64,%al
   b:   a8 02                   test   $0x2,%al
   d:   75 fa                   jne    0x9
   f:   b0 d1                   mov    $0xd1,%al
  11:   e6 64                   out    %al,$0x64
  13:   e4 64                   in     $0x64,%al
  15:   a8 02                   test   $0x2,%al
  17:   75 fa                   jne    0x13
  19:   b0 df                   mov    $0xdf,%al
  1b:   e6 60                   out    %al,$0x60
  1d:   0f 01 16 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
  41:   8e e8                   mov    %ax,%gs
  43:   bc 00 7c                mov    $0x7c00,%sp
  46:   00 00                   add    %al,(%bx,%si)
  48:   e8 f0 00                call   0x13b
  4b:   00 00                   add    %al,(%bx,%si)
  4d:   66 b8 00 8a 66 89       mov    $0x89668a00,%eax
  53:   c2 66 ef                ret    $0xef66
  56:   66 b8 e0 8a 66 ef       mov    $0xef668ae0,%eax
  5c:   eb fe                   jmp    0x5c
```

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

با استفاده از این دستور می‌توان محتویات یک شیء فایل را در یک شیء فایل دیگر کپی کرد. این دستور برای ترجمه فایل‌های object، از فایل‌های موقت (temp) استفاده می‌کند و در ادامه آن‌ها را پاک می‌کند. در ادامه آپشن هایی از این دستور که در Makefile مربوط به xv6 استفاده شده اند، را توضیح می‌دهیم:

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

در این Makefile در چندین بخش از دستور objcopy استفاده شده است:

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

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

۱۴- یک ثبات عام منظوره، یک ثبات قطعه، یک ثبات وضعیت و یک ثبات کنترلی در معماری x86 را نام برده و وظیفه هر یک را به شکل مختصر توضیح دهید.

- ثبات عام منظوره: همانطور که در کتاب هم اشاره شده است، xv6 دارای ۸ ثبات عام منظوره است. این ثبات ها عبارتند از:
- %eax, %ebx, %ecx, %edx, %edi, %esi, %ebp, %esp و یک program counter به نام eip. حرف e در این ثبات ها نشان دهنده extended است چرا که ۳۲ بیتی اند. این ثبات ها برای نگهداری برخی اشاره‌گر ها، نگهداری داده‌ها و نگهداری عملیات های ریاضی استفاده می‌شوند.
- ثبات قطعه: آدرس استک، کد و داده در این ثبات ها نگه داری می‌شود. برای مثال SS اشاره‌گر به استک، CS اشاره‌گر به کد و DS اشاره‌گر به داده را نگه می‌دارد.
- ثبات وضعیت: شامل اطلاعات راجع به وضعیت پردازنده است. EFLAGS در این بخش محسوب می‌شود و اطلاعات پرچم‌هایی نظیر carry, zero, sign و غیره را مشخص می‌کند.
- ثبات کنترلی: کنترل CPU یا دستگاه‌های دیجیتال دیگر را در دست دارد.

  %cr0, %cr2, %cr3, %cr4

  از این ثبات هایند. این ثبات ها وظیفه تغییر مدل آدرس دهی، کنترل interrupt، کنترل paging و هم‌پردازنده ها را دارند.

۱۸- کد معادل entry.S در هسته لینوکس را بیابید.

https://github.com/torvalds/linux/blob/master/arch/arm64/kernel/entry.S

اجرای هسته xv6

۱۹- چرا این آدرس فیزیکی است؟

اگر این بخش را به صورت مجازی در نظر می‌گرفتیم، باز هم بایستی یک بخش فیزیکی در نظر می‌گرفتیم تا این بخش مجازی را مشخص کند، یعنی در نهایت نیاز به بخش فیزیکی بود.

۲۲- علاوه بر صفحه بندی در حد ابتدایی از قطعه بندی به منظور حفاظت هسته استفاده خواهد شد. این عملیات توسط seginit() انجام می‌شود. همان‌طور که ذکر شد، ترجمه قطعه تأثیری بر ترجمه آدرس منطقی نمی‌گذارد. چرا که تمامی قطعه‌ها اعم از کد و داده روی یکدیگر می‌افتند. با این حال برای کد و داده‌های سطح کاربر، پرچم SEG_USER تنظیم شده است. چرا؟ (راهنمایی: علت مربوط به ماهیت دستورالعمل ها و نه آدرس است.)

تا بتوان تفاوتی بین پردازه های سطح کاربر و پردازه های سطح هسته ایجاد کرد. از آنجایی که محتوای هر دوی این پردازه ها در یک فضای فیزیکی قرار گرفته اند، با این کار می‌شود تشخیص داد که آن داده‌ها یا کدها، مربوط به سطح کاربرند و اجازه دسترسی به هسته را ندارند.

اجرای نخستین برنامه سطح کاربر

۲۳- جهت نگه داری اطلاعات مدیریتی برنامه‌های سطح کاربر ساختاری تحت عنوان struct proc (خط ۲۳۳۶) ارائه شده است. اجزای آن را توضیح داده و ساختار معادل آن در سیستم عامل لینوکس را بیابید.

- SZ: سایز حافظه متعلق به پردازه به بایت
- pgdir: اشاره‌گر به page table است.
- kstack: پایین استک هسته برای این پردازه را مشخص می‌کند.
- state: وضعیت این پردازه را مشخص می‌کند.
- pid: عدد اختصاص داده شده به این پردازه است
- parent: پدر این پردازه یا به عبارت دیگر سازنده این پردازه را مشخص می‌کند.
- tf: چارچوب trap برای فراخوانی سیستمی حال حاضر است.
- context: برای context switching نگه داری شده است.
- chan: اگر صفر نباشد، به معنای خوابیدن پردازه است.
- killed: اگر غیر صفر باشد، یعنی پردازه kill شده است.
- ofile: فایل‌های باز شده توسط این پردازه است.
- cwd: پوشه کنونی را مشخص می‌کند.

- name: نام این پردازه است.

معادل این struct در هسته لینوکس در لینک زیر آمده است: (task_struct)

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

۲۷- کدام بخش از آماده سازی سیستم، بین تمامی هسته های پردازنده مشترک و کدام بخش اختصاصی است؟ (از هر کدام یک مورد را با ذکر دلیل توضیح دهید.) زمان بند روی کدام هسته اجرا می‌شود؟

هسته اول که فرایند بوت را انجام می‌دهد توسط کد entry.S وارد تابع main در فایل main.c می‌شود. تمامی توابع آماده سازی سیستم که در این تابع فراخوانده شده‌اند، توسط این هسته اجرا می‌شوند. از طرفی، هسته های دیگر از طریق کد entryother.S وارد تابع mpenter می‌شوند. در این تابع نیز ۴ تابع برای آماده سازی فراخوانده می‌شوند. در نتیجه می‌توان گفت این ۴ تابع بین تمامی هسته ها مشترک خواهند بود. یکی از این توابع به نام switchkvm به صورت مستقیم با هسته اول مشترک نیست. این تابع در mpemter صدا زده می‌شود در صورتی که در تابع main وجود ندارد.

بخش‌هایی از آماده سازی سیستم که در تمام هسته ها مشترک است، به شرح زیر است:

- switchkvm
- seginit
- lapicinit
- mpmain

همچنین بخش‌هایی که تنها در هسته اول (به شکل اختصاصی) اجرا می‌شوند، به شرح زیرند:

- kinit1
- setupkvm kvmalloc
- mpinit
- picinit
- ioapicinit
- consoleinit
- uartinit
- pinit
- tvinit

- binit
- fileinit
- ideinit
- startothers
- kinit2
- userinit

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

از طرفی، تمام پردازنده ها باید آدرس page table که توسط پردازنده اول ایجاد شده را در ثبات خود ذخیره کنند در نتیجه تابع switchkvm بین تمامی آن‌ها مشترک است. همچنین، همه پردازنده ها باید کار خود را شروع کنند و آماده اجرای برنامه‌ها شوند که این کار توسط تابع mpmain انجام می‌شود. پس این تابع هم بین همه پردازنده ها مشترک است.

زمان بند که توسط تابع scheduler انجام می‌شود در تابع mpmain صدا زده می‌شود که این تابع بین همه هسته ها مشترک است. درواقع هر پردازنده scheduler مربوط به خودش را دارد. در نتیجه این تابع بین تمامی پردازنده ها مشترک است.

اشکال زدایی

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

از دستور info breakpoints استفاده می‌شود.

```
(gdb) b cat.c:10
Breakpoint 1 at 0x93: file cat.c, line 10.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00000093 in cat at cat.c:10
(gdb)
```

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

دو روش برای حذف Breakpoint ها وجود دارد:

- روش اول: clear file_name:line_number که file_name اسم فایل و line_number هم شماره خط است.
- روش دوم: del breakpoint_number که breakpoint_number شماره breakpoint است و همان ستون Num پس از زدن دستور info breakpoints است.

در ادامه ۳ تا breakpoint به فایل cat.c اضافه می‌کنیم و دوتای آن‌ها را با دو روش گفته شده حذف می‌کنیم. برای بررسی صحت حذف شدن‌شان هم در انتها info breakpoints می‌زنیم تا ببینیم حذف شده‌اند یا خیر.

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) b cat.c:8
Breakpoint 1 at 0xf0: file cat.c, line 8.
(gdb) b cat.c:10
Breakpoint 2 at 0x93: file cat.c, line 10.
(gdb) b cat.c:12
Note: breakpoint 2 also set at pc 0x93.
Breakpoint 3 at 0x93: file cat.c, line 12.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000000f0 in cat at cat.c:8
2       breakpoint     keep y   0x00000093 in cat at cat.c:10
3       breakpoint     keep y   0x00000093 in cat at cat.c:12
(gdb) clear cat.c:8
Deleted breakpoint 1
(gdb) del 3
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x00000093 in cat at cat.c:10
(gdb)
```

۳- دستور bt را اجرا کنید. خروجی آن چه چیزی را نشان می‌دهد؟

این دستور مخفف backtrace است و سلسله توابعی که فراخوانی شده‌اند و به استک اضافه شده‌اند را نشان می‌دهد. به طور خلاصه می‌توان گفت این دستور نشان می‌دهد که برنامه چطور به جایی که اکنون در آن قرار دارد، رسیده است. برای نشان دادن خروجی دستور bt از دیباگ سطح هسته استفاده کردیم یعنی دستور gdb kernel را زدیم. در ادامه خروجی این دستور را در یک عکس آورده‌ایم:

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) b sleep
Breakpoint 1 at 0x801043d1: sleep. (2 locations)
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, sleep (chan=0x8010b554 <bcache+52>, lk=0x80112600 <idelock>) at proc.c:61
61        pushcli();
(gdb) bt
#0  sleep (chan=0x8010b554 <bcache+52>, lk=0x80112600 <idelock>) at proc.c:61
#1  0x8010280e in iderw (b=0x8010b554 <bcache+52>) at ide.c:163
#2  0x80100191 in bread (dev=1, blockno=1) at bio.c:103
#3  0x80101a4c in readsb (sb=0x801125b4 <sb>, dev=1) at fs.c:36
#4  iinit (dev=1) at fs.c:181
#5  0x80103d54 in forkret () at proc.c:408
#6  forkret () at proc.c:397
#7  0x80105cf2 in alltraps () at trapasm.S:21
(gdb)
```

#0 مربوط به فریم فعلی و محل breakpoint است و در #1، #0 صدا زده شده است و ...

۴- دو تفاوت دستور های x و print را توضیح دهید. چگونه می‌توان محتوای یک ثبات خاص را چاپ کرد؟ (راهنمایی: می‌توانید از دستور help استفاده نمایید: help x و help print با دستور list می‌توان کد نقطه توقف را مشاهده کرد.)

دستور print می‌تواند یک expression دریافت کند و مقدار آن را نمایش دهد اما دستور x بر اساس آدرس کار می‌کند و مقدار را نمایش می‌دهد. همچنین در دستور x استفاده از بخش FMT بر خلاف print الزامیست و به شکل تعداد تکرار همراه با حرف فرمت مانند o, x, d, u, t, a, f, i, c, s, z و حرف اندازه مانند b, h, w, g است. این دو دستور از جهت نحوه نمایش اطلاعات نیز با یکدیگر تفاوت دارند.

با استفاده از دستور info register register_name یا info registers register_name می‌توان یک ثبات خاص را نمایش داد، که در آن register_name نام ثبات مورد نظر است.

در ادامه یک نمونه استفاده از دستور print آورده شده است. در این مثال ابتدا با دستور gdb kernel اشکال زدایی را از سطح هسته شروع کردیم. سپس یک breakpoint در خط ۳۲۰ فایل cosole.c قرار دادیم. در ادامه با زدن c برنامه را ادامه دادیم و با زدن shift+’[’ به ابتدای خط آمدیم تا برنامه بتواند ادامه یابد. سپس مقدار input.e را بررسی کردیم که چون چیزی ننوشتیم، 0 بود. سپس دوباره c را زدیم تا برنامه ادامه یابد و این بار با نوشتن کلمه JAVAD و زدن دوباره shift+’[’ ، مقدار input.e را بررسی کردیم که ۵ شد.

تصاویر مربوط به این کار در ادامه آمده‌اند:

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) b console.c:320
Breakpoint 1 at 0x80100cf7: file console.c, line 320.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, consoleintr (getc=0x80102b90 <kbdgetc>) at console.c:320
320           go_to_beginning_of_line();
(gdb) print input.e
$1 = 0
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, consoleintr (getc=0x80102b90 <kbdgetc>) at console.c:320
320           go_to_beginning_of_line();
(gdb) print input.e
$2 = 5
(gdb)
```

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
Group #13:
1. Arshia
2. Zeinab
3. Javad
$ JAVAD_
```

در اینجا هم یک مثال از نحوه استفاده از دستور info register برای نمایش اطلاعات ثبات‌های esi و edi می‌زنیم:

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) info register edi
edi            0x0                 0
(gdb) info register esi
esi            0x0                 0
```

۵- برای نمایش وضعیت ثبات ها از چه دستوری استفاده می‌شود؟ متغیر های محلی چطور؟ نتیجه این دستور را در گزارش کار خود بیاورید. همچنین در گزارش کار خود توضیح دهید که در معماری x86 رجیستر های edi و esi نشانگر چه چیزی هستند؟

برای نمایش وضعیت ثبات‌ها می‌توان از دستور info registers استفاده کرد:

```
(gdb) info registers
eax            0x0                 0
ecx            0x0                 0
edx            0x663               1635
ebx            0x0                 0
esp            0x0                 0x0
ebp            0x0                 0x0
esi            0x0                 0
edi            0x0                 0
eip            0xfff0              0xfff0
eflags         0x2                 [ IOPL=0 ]
cs             0xf000              61440
ss             0x0                 0
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
fs_base        0x0                 0
gs_base        0x0                 0
k_gs_base      0x0                 0
cr0            0x60000010          [ CD NW ET ]
cr2            0x0                 0
cr3            0x0                 [ PDBR=0 PCID=0 ]
cr4            0x0                 [ ]
cr8            0x0                 0
efer           0x0                 [ ]
xmm0           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0
, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm1           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0
, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm2           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0
, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm3           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0
, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm4           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0
, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm5           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0
, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm6           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0
, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm7           {v4_float = {0x0, 0x0, 0x0, 0x1f80}, v2_double = {0x0, 0x1f8000000000}, v16_int8 = {0x0 <repeats 12 times>, 0x80, 0x1f, 0x0, 0x0}, v8_int16 = {0x0, 0x0--Type <RET> for
--Type <RET> for more, q to quit, c to continue without paging--
```

برای نمایش وضعیت متغیرهای محلی نیز کافیست از دستور info locals استفاده کنیم. خروجی زیر برای فایل console.c و ورودی خالی می‌باشد:

```
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) b console.c:320
Breakpoint 1 at 0x80100cf7: file console.c, line 320.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, consoleintr (getc=0x80102b90 <kbdgetc>) at console.c:320
320           go_to_beginning_of_line();
(gdb) info locals
c = 123
doprocdump = 0
(gdb)
```

و اگر بخواهیم تمامی متغیرها را به صورت سراسری مشاهده نماییم، می‌توانیم از دستور info variables استفاده نماییم. خروجی صفحه بعدذ برای فایل console.c و به ازای ورودی hello می‌باشد:

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) b console.c:320
Breakpoint 1 at 0x80100cf7: file console.c, line 320.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, consoleintr (getc=0x80102b90 <kbdgetc>) at console.c:320
320           go_to_beginning_of_line();
(gdb) info variables
All defined variables:

File /home/javad/My Folders/University/6th Term/OS/Lab Projects/Project1/xv6-public-master/kbd.h:
95:     static uchar ctlmap[256];
51:     static uchar normalmap[256];
34:     static uchar shiftcode[256];
73:     static uchar shiftmap[256];
44:     static uchar togglecode[256];

File bio.c:
36:     struct {
    struct spinlock lock;
    struct buf buf[30];
    struct buf head;
} bcache;

File console.c:
142:    int back_counter;
210:    struct {
    char buf[128];
    uint r;
    uint w;
    uint e;
} input;
36:     static struct {
    struct spinlock lock;
    int locking;
} cons;
140:    static ushort *crt;
31:     static int panicked;

File file.c:
13:     struct devsw devsw[10];
17:     struct {
    struct spinlock lock;
    struct file file[100];
} ftable;

File fs.c:
169:    struct {
    struct spinlock lock;
    struct inode inode[50];
--Type <RET> for more, q to quit, c to continue without paging--
```

DI مخفف عبارت Destination Index و SI مخفف عبارت Source Index است. با این ثبات ها تعداد محدودی عملیات قابل انجام است: REP STOSB | MOVSB | SCASB که عملیات ذخیره سازی، بارگذاری، و اسکن مکرر را انجام می‌دهند. به عنوان مثال MOVSB می‌تواند برای انتقال داده‌ها از یک بافر به بافر دیگر استفاده شود.

همچنین برخی عملیات‌ها مانند عملیات مربوط به رشته‌ها یا عملیات مربوط به کپی یا مقدار دهی یک آرایه در حافظه برای انجام شدن به ثبات های edi و esi نیازمندند. در اینگونه عملیات ها esi ثبات مکان مبدأ (نگه دارنده source) و edi ثبات مکان مقصد (نگه دارنده destination) است.

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

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

این استراکت در فایل console.c تعریف شده است و برای خط ورودی کنسول سیستم عامل مورد استفاده قرار می‌گیرد:

```
204  #define INPUT_BUF 128
205  struct {
206    char buf[INPUT_BUF];
207    uint r;  // Read index
208    uint w;  // Write index
209    uint e;  // Edit index
210  } input;
```

- آرایه buf خط ورودی را ذخیره می‌کند و اندازه آن حداکثر 128 کاراکتر است.
- متغیر r برای خواندن buf استفاده می‌شود و از اندیس نوشتن قبلی شروع به خواندن می‌کند.
- متغیر w اندیس شروع نوشتن خط ورودی در buf است.
- متغیر e محل فعلی cursor در خط ورودیست.

برای دیدن نحوه تغییر این متغیر ها از GDB استفاده می‌کنیم. برای آنکه مقادیر اولیه این متغیر ها را ببینیم یک breakpoint در خط ۳۳۲ فایل console.c قرار می‌دهیم. سپس برنامه را continue کرده، ctrl+c را میزنیم و input را چاپ می‌کنیم:

```
330         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
331           input.w = input.e;
332           wakeup(&input.r);
```

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
mycpu () at proc.c:48
48          for (i = 0; i < ncpu; ++i) {
(gdb) print input
$1 = {buf = '\000' <repeats 127 times>, r = 0, w = 0, e = 0}
```

همانطور که مشاهده می‌شود مقادیر ابتدایی متغیرهای w ،e و r برابر 0 است. در ادامه دوباره برنامه را continue کرده و در کنسول عبارت areshia و ctrl+d را مینویسیم. (برای ادامه اجرای برنامه فشردن کلید های ctrl+d الزامیست.) دوباره input را چاپ می‌کنیم:

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
Group #13:
1. Arshia
2. Zeinab
3. Javad
$ arshia
```

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, consoleintr (getc=0x80102b90 <kbdgetc>) at console.c:332
332               wakeup(&input.r);
(gdb) print input
$2 = {buf = "arshia\004", '\000' <repeats 120 times>, r = 0, w = 7, e = 7}
```

همان‌طور که مشاهده می‌شود متغیرهای e و w از 0 به 7 رسیدند. که 7 اندیس یکی بعد از آخرین کاراکتر وارد شده در buf است.

در ادامه برنامه را continue کرده و روند اجرا را با فشردن ctrl+c متوقف می‌کنیم و دوباره input را چاپ می‌کنیم:

```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
mycpu () at proc.c:48
48        for (i = 0; i < ncpu; ++i) {
(gdb) print input
$3 = {buf = "arshia\004", '\000' <repeats 120 times>, r = 7, w = 7, e = 7}
(gdb)
```

همان‌طور که مشاهده می‌شود مقدار r با w برابر شد. یعنی از ابتدای خط (w=0) شروع کرده و تا انتهای خط (w=7) را خوانده است و اندیس r را هم یک واحد یک واحد زیاده کرده است.

در ادامه با زدن c برنامه را ادامه می‌دهیم و سپس عبارت zeinab را می‌نویسیم ولی این بار ctrl+d نمی‌زنیم و خودمان با ctrl+c در gdb روند اجرا متوقف می‌کنیم. سپس input را چاپ می‌کنیم:

```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
mycpu () at proc.c:48
48        for (i = 0; i < ncpu; ++i) {
(gdb) print input
$5 = {buf = "arshia\nzeinab", '\000' <repeats 114 times>, r = 7, w = 7, e = 13}
(gdb)
```

همان‌طور که مشاهده می‌شود این بار فقط متغیر e زیاد شده است. حال اگر دوباره برنامه را continue کنیم، سپس کاراکتر آخر را پاک کنیم و دوباره با ctrl+c روند اجرا متوقف کنیم، با چاپ input خواهیم داشت:

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
Group #13:
1. Arshia
2. Zeinab
3. Javad
$ arshia
exec: fail
exec arshia failed
$ zeina
```

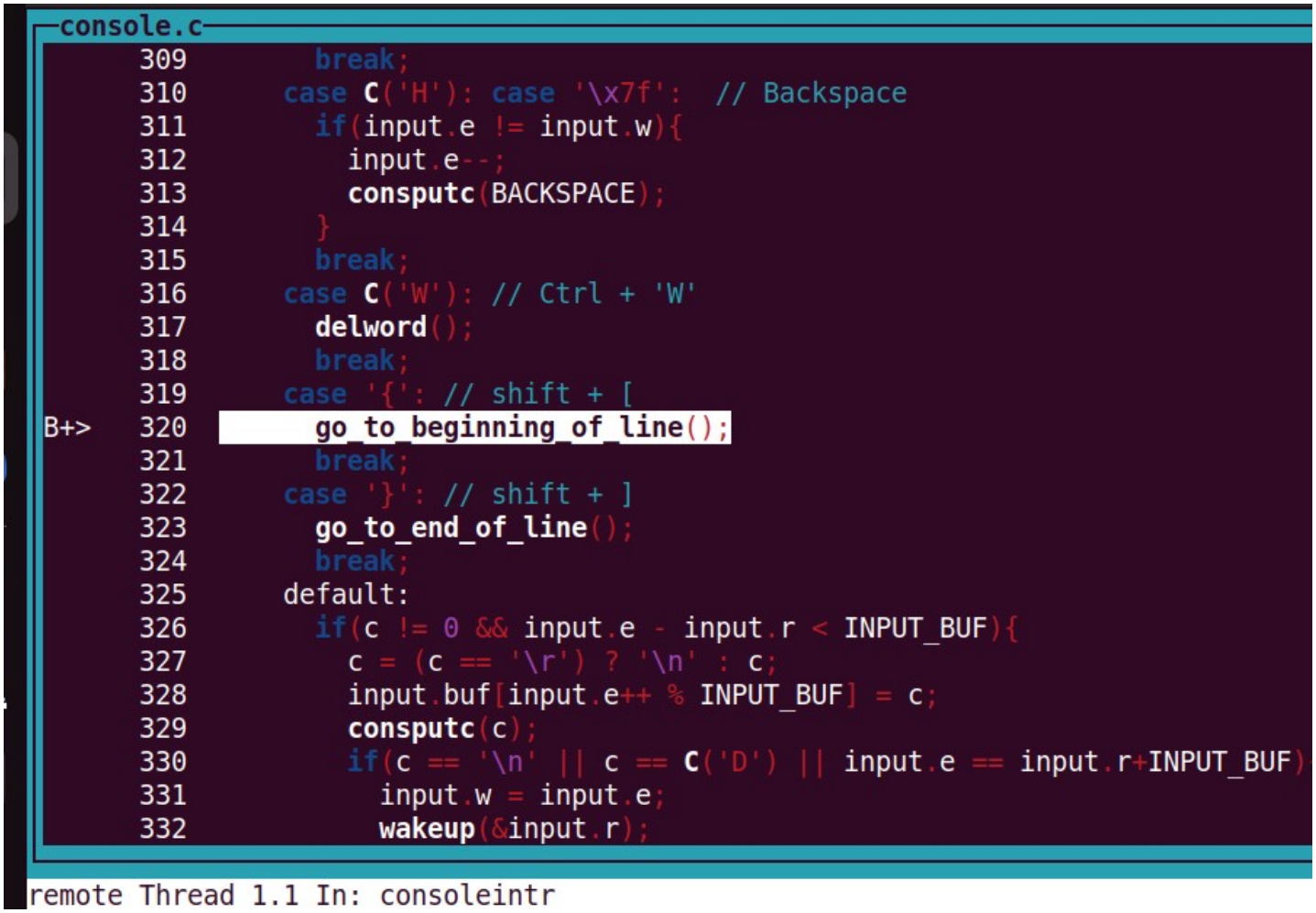
```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
mycpu () at proc.c:48
48          for (i = 0; i < ncpu; ++i) {
(gdb) print input
$6 = {buf = "arshia\nzeinab", '\000' <repeats 114 times>, r = 7, w = 7, e = 12}
(gdb)
```

همان‌طور که انتظار می‌رفت مقدار متغیر e یک واحد کم شده است.

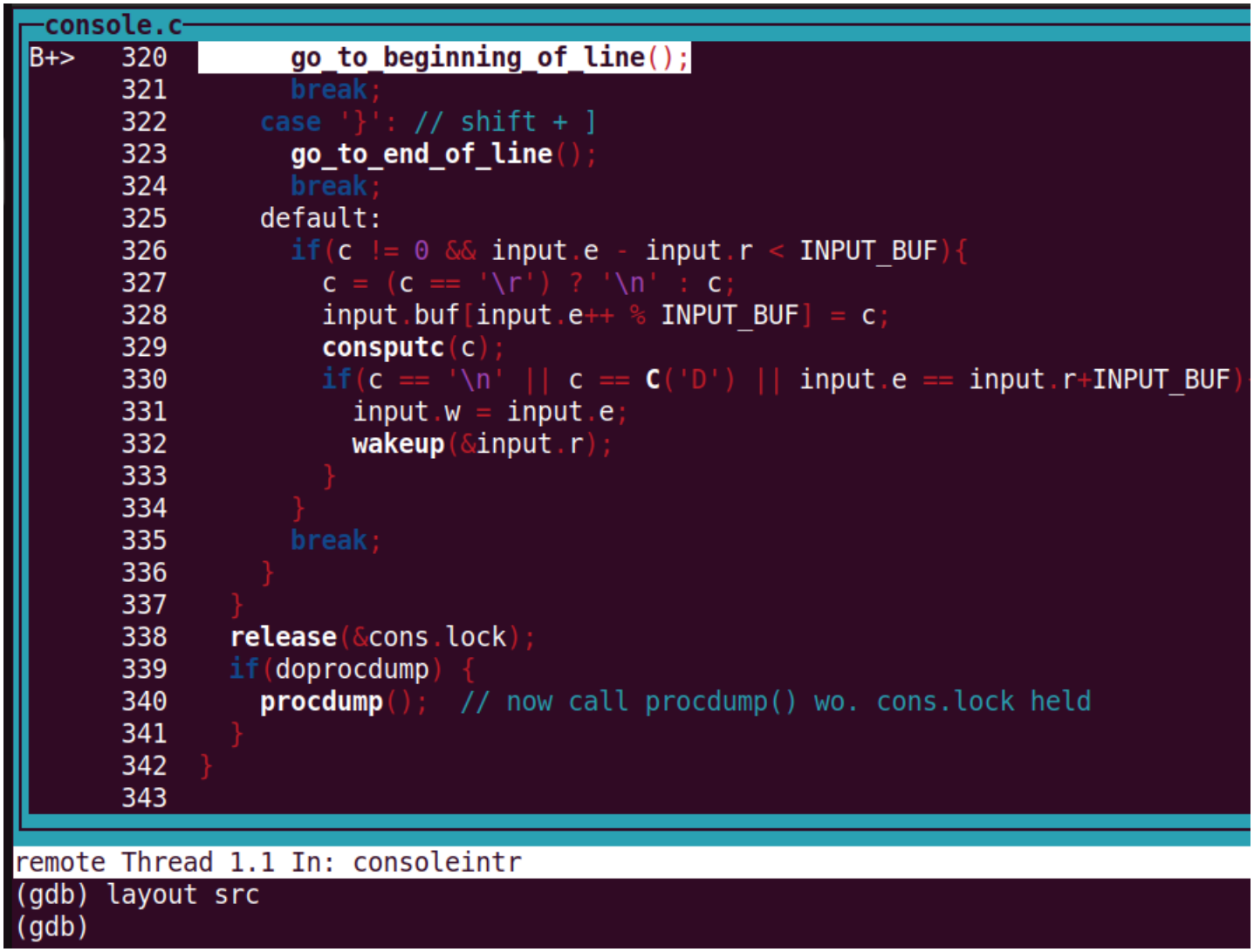
اشکال زدایی در سطح کد اسمبلی

۷- خروجی دستور های layout src و layout asm در TUI چیست؟

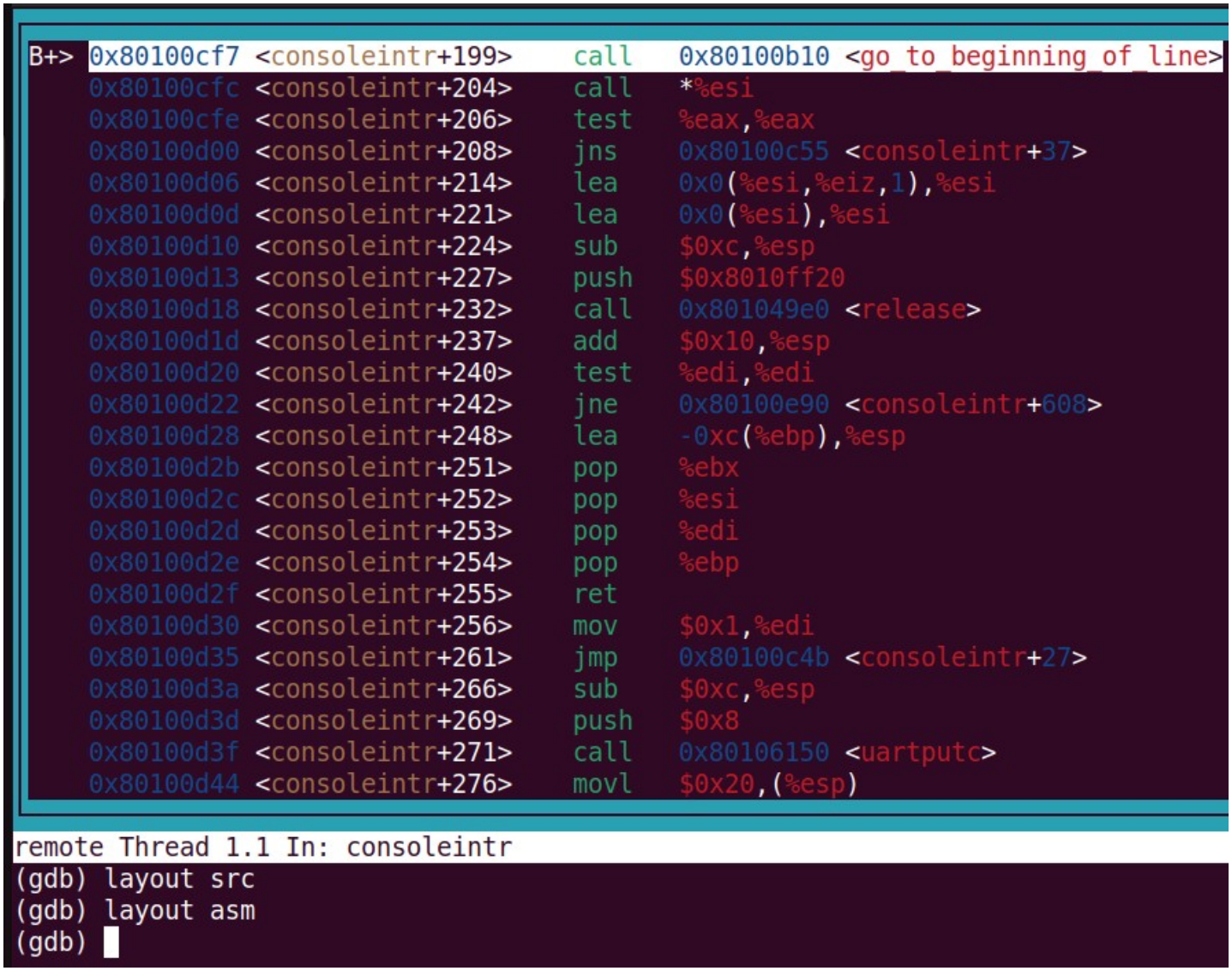
یک breakpoint برای خط ۳۲۰ فایل cosole.c می‌گذاریم. در صورت touch شدن آن، با باز کردن TUI، به همان خط ۳۲۰ خواهیم رفت. برای touch شدن آن کافیست پس از گذاشتن breakpoint مانند قسمت قبل برنامه را continue کرده و در ادامه با زدن shift+’[’ به ابتدای خط بیاییم حال اگر TUI را باز کنیم، خط ۳۲۰ نشان داده خواهد شد:



با استفاده از دستور layout src برنامه در حالت کد منبعش (در اینجا زبان C) قرار خواهد گرفت:



با استفاده از دستور layout asm می‌توان کد اسمبلی برنامه در حال دیباگ را مشاهده کرد:



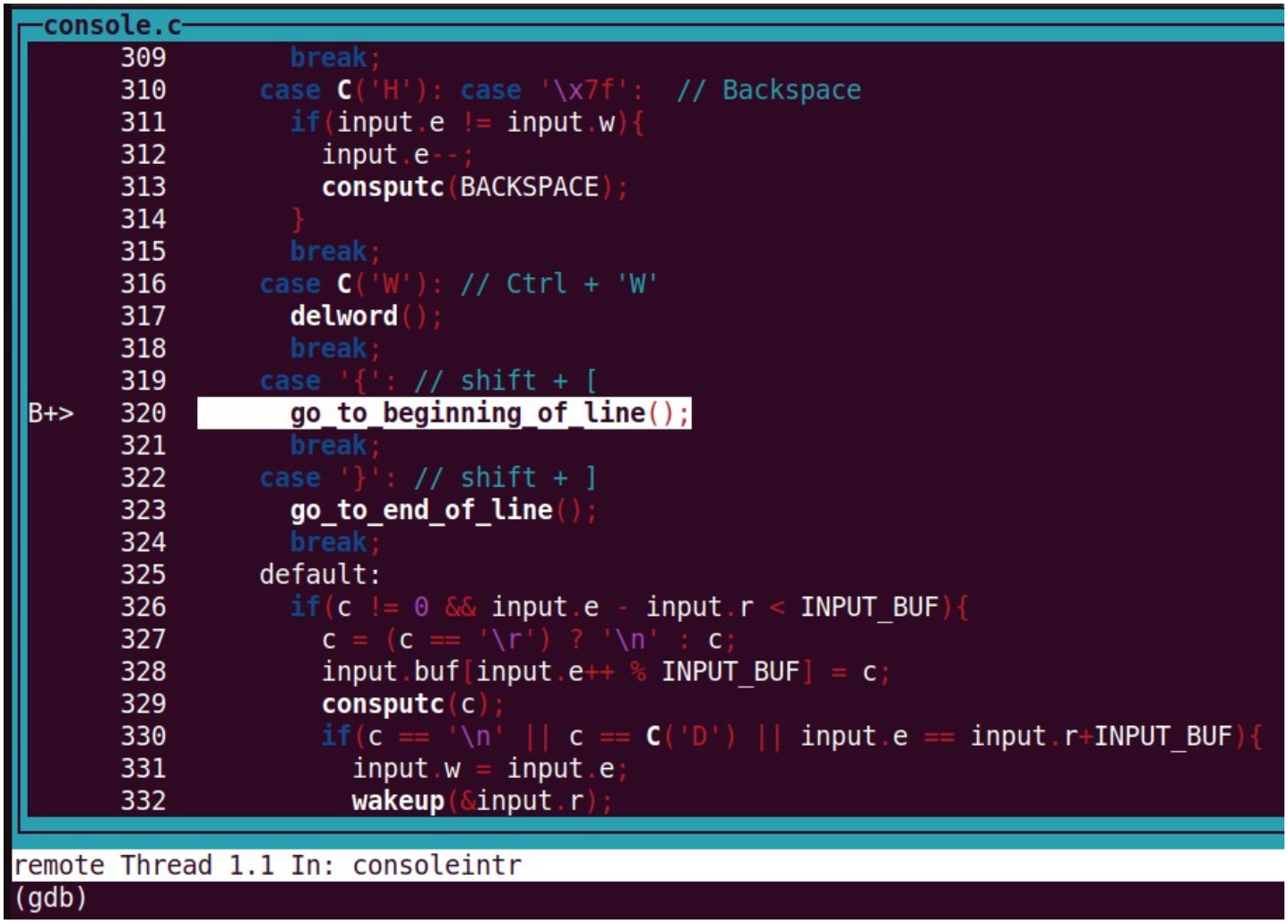
۸- برای جابجایی بین توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستوراتی استفاده می‌شود؟

به وسیله دستور up n، می‌توان به فریم قبلی یا بیرونی رفت. با دستور down n نیز امکان جابجایی به فریم بعدی یا داخلی را خواهیم داشت. n می‌گوید چند فریم جابجا شویم و به طور پیش‌فرض ۱ است. در فریم هایی که دستور bt نشان می‌دهد، به وسیله up می‌توان به فریم با شماره بیشتر و با دستور down می‌توان به فریم با شماره کمتر رفت. به عنوان مثال در صورتی که در خط ۳۲۰ فایل console.c یک breakpoint بگذاریم، breakpoint را touch کنیم و دستور bt را بزنیم، خواهیم داشت:

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) b console.c:320
Breakpoint 1 at 0x80100cf7: file console.c, line 320.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, consoleintr (getc=0x80102b90 <kbdgetc>) at console.c:320
320           go_to_beginning_of_line();
(gdb) bt
#0  consoleintr (getc=0x80102b90 <kbdgetc>) at console.c:320
#1  0x80102c80 in kbdintr () at kbd.c:49
#2  0x80105f95 in trap (tf=0x80116418 <stack+3912>) at trap.c:67
#3  0x80105cef in alltraps () at trapasm.S:20
#4  0x80116418 in stack ()
#5  0x801127a4 in cpus ()
#6  0x801127a0 in ?? ()
#7  0x801034df in mpmain () at main.c:57
#8  0x8010362c in main () at main.c:37
(gdb)
```

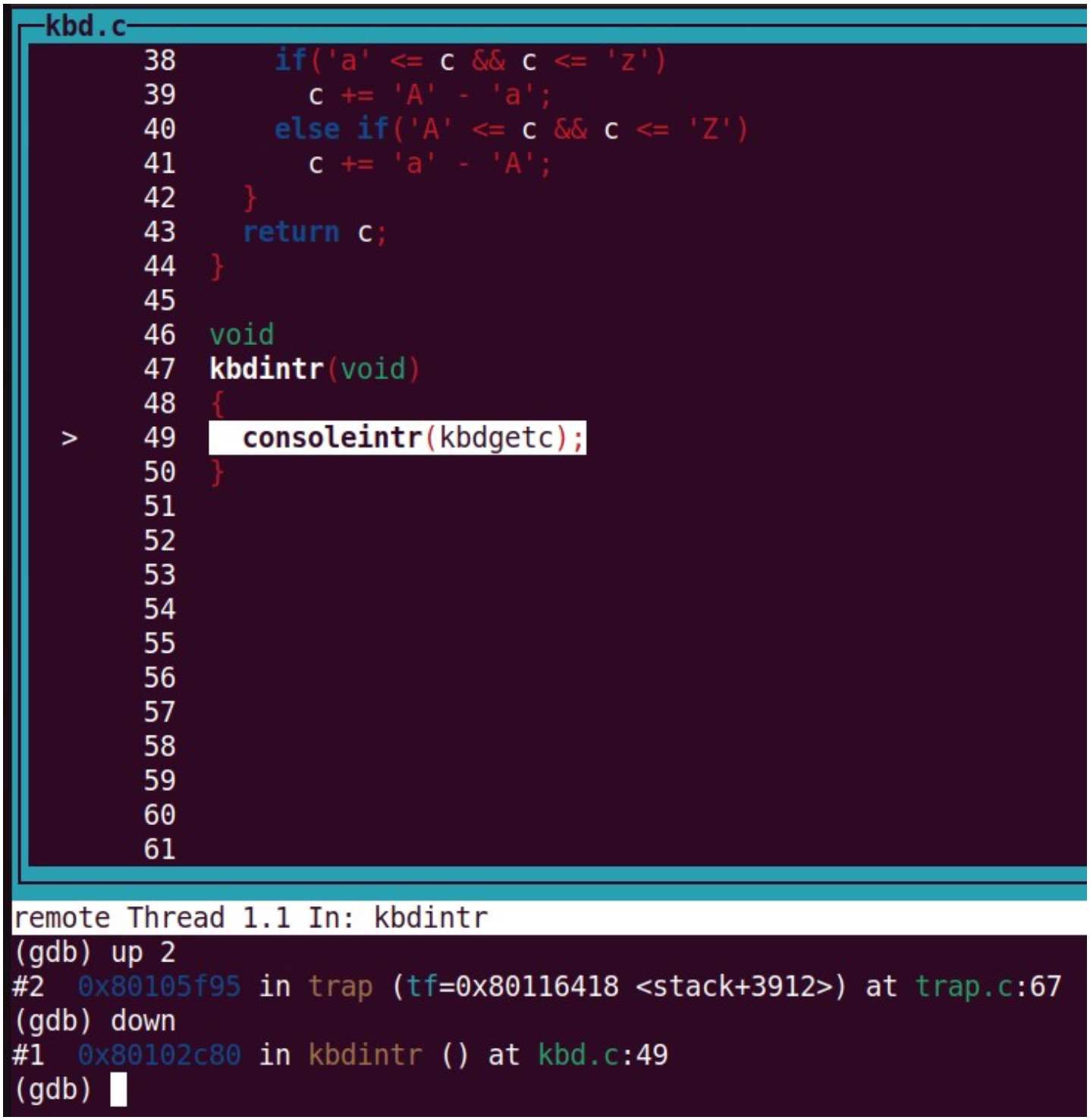
حال اگر به TUI برویم، در console.c خواهیم بود:



و در صورت استفاده از دستور up 2 به فایل trap.c خواهیم رفت:

```
trap.c
      56      }
      57      lapiceoi();
      58      break;
      59    case T_IRQ0 + IRQ_IDE:
      60      ideintr();
      61      lapiceoi();
      62      break;
      63    case T_IRQ0 + IRQ_IDE+1:
      64      // Bochs generates spurious IDE1 interrupts.
      65      break;
      66    case T_IRQ0 + IRQ_KBD:
  >   67      kbdintr();
      68      lapiceoi();
      69      break;
      70    case T_IRQ0 + IRQ_COM1:
      71      uartintr();
      72      lapiceoi();
      73      break;
      74    case T_IRQ0 + 7:
      75    case T_IRQ0 + IRQ_SPURIOUS:
      76      cprintf("cpu%d: spurious interrupt at %x:%x\n",
      77              cpuid(), tf->cs, tf->eip);
      78      lapiceoi();
      79      break;
remote Thread 1.1 In: trap
(gdb) up 2
#2  0x80105f95 in trap (tf=0x80116418 <stack+3912>) at trap.c:67
(gdb)
```

و به وسیله دستور down به فایل kbd.c منتقل خواهیم شد:



پیکربندی و ساختن هسته لینوکس (امتیازی)

برای نشان داده شدن نام اعضای گروه پس از زدن دستور sudo dmesg ابتدا بایستی یک ماژول بنویسیم و سپس ماژول را به هسته لینوکس اضافه کنیم. به همین منظور فایل members.c را نوشتیم و Makefile مناسب آن را نیز نوشتیم که در ادامه کد های مربوط به این دو فایل را می‌آوریم:

- فایل members.c:

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

MODULE_LICENSE("GPL");

int init_module(void){
    printk(KERN_INFO "Group #13:\n1. Arshia\n2. Zeinab\n3. Javad\n");
    return 0;
}

void cleanup_module(void) {}
```

- Makefile:

```
obj-m += members.o

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

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

حال فایل members.c را make می‌کنیم:

```
javad@Javad-Bshrt:~/new_module$ make
make -C /lib/modules/5.19.0-32-generic/build M="/home/javad/new_module" modules
make[1]: Entering directory '/usr/src/linux-headers-5.19.0-32-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
  You are using:           gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
  CC [M]  /home/javad/new_module/members.o
  MODPOST /home/javad/new_module/Module.symvers
  CC [M]  /home/javad/new_module/members.mod.o
  LD [M]  /home/javad/new_module/members.ko
  BTF [M] /home/javad/new_module/members.ko
Skipping BTF generation for /home/javad/new_module/members.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.19.0-32-generic'
```

حال اگر نگاهی به فایل‌های ایجاد شده بیندازیم، فایل members.ko را خواهیم دید که مهم‌ترین فایل است و ماژول هسته کامپایل شده است:

```
make[1]: Leaving directory '/usr/src/linux-headers-5.19.0-32-generic'
javad@Javad-Bshrt:~/new_module$ ls
Makefile  members.c  members.ko  members.mod  members.mod.c  members.mod.o  members.o  modules.order  Module.symvers
javad@Javad-Bshrt:~/new_module$
```

برای افزودن این ماژول کافیست دستور sudo insmod members.ko را بزنیم و همچنین برای حذف آن کافیست از دستور sudo rmmod members استفاده نماییم:

```
javad@Javad-Bshrt:~/new_module$ sudo insmod members.ko
[sudo] password for javad:
javad@Javad-Bshrt:~/new_module$
```

برای آنکه بررسی کنیم که یک ماژول به هسته اضافه شده است یا خیر کافیست از دستور lsmod استفاده کنیم. این دستور تمام ماژول های هسته را برای ما لیست خواهد کرد:

```
javad@Javad-Bshrt:~/new_module$ lsmod
Module                  Size  Used by
members                16384  0
ccm                    20480  0
udp_diag               16384  0
tcp_diag               16384  0
inet_diag              24576  2 tcp_diag,udp_diag
nf_tables             278528  16
libcrc32c              16384  1 nf_tables
nfnetlink              20480  1 nf_tables
```

همانطور که میبینید ماژول با موفقیت به هسته اضافه شده است. حال اگر دستور sudo dmesg را بزنیم، نام اعضای گروه قابل مشاهده خواهد بود:

```
[73288.799491] members: loading out-of-tree module taints kernel.
[73288.799582] members: module verification failed: signature and/or required key missing - tainting kernel
[75892.427419] audit: type=1107 audit(1677524042.819:9129): pid=1015 uid=103 auid=4294967295 ses=4294967295 subj=unconfined msg='apparmor="DENIED" operation="dbus_met
hod_call"  bus="system" path="/org/freedesktop/hostname1" interface="org.freedesktop.DBus.Properties" member="GetAll" mask="send" name=":1.1251" pid=2868 label="snap.
firefox.firefox" peer_pid=96419 peer_label="unconfined"
             exe="/usr/bin/dbus-daemon" sauid=103 hostname=? addr=? terminal=?'
[75892.427482] audit: type=1107 audit(1677524042.819:9130): pid=1015 uid=103 auid=4294967295 ses=4294967295 subj=unconfined msg='apparmor="DENIED" operation="dbus_met
hod_call"  bus="system" path="/org/freedesktop/hostname1" interface="org.freedesktop.DBus.Properties" member="GetAll" mask="send" name=":1.1251" pid=2868 label="snap.
firefox.firefox" peer_pid=96419 peer_label="unconfined"
             exe="/usr/bin/dbus-daemon" sauid=103 hostname=? addr=? terminal=?'
[76066.251166] Group #13:
               1. Arshia
               2. Zeinab
               3. Javad
javad@Javad-Bshrt:~/new_module$
```

github repository link:

https://github.com/ArshiAbolghasemi/ut-os-lab-projects

hash last commit:

507dab76355f0456e5149e5338a335b4f879d143