آزمایشگاه سیستم عامل

پروژه یک

اعضای گروه:

پارسا علیزاده - 810101572

نیلوفر مرتضوی - 220701096

محمدرضا خالصی - 810101580

Repository: https://github.com/pars1383/OS-xv6-public

Latest Commit: 7591081dc3d2dd22f19ee13c7cd1984a395547fd

آشنایی با سیستم عامل **xv**6

- 1 معماری سیستم عامل **xv**6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

سیستم عامل **xv6** یک سیستم عامل آموزشی **Unix-Based** است که بر اساس مستندات آن، یک **re-implementation** از **Unix V6** است و این سیستم عامل مبتنی بر پردازنده های **x86** نوشته شده است. در فایل **x86.h** نیز دستورات پردازنده **x86** استفاده شده.

سیستم عامل unix از سه بخش اصلی kernel, shell و user applications تشکیل شده است که 6xv هم همین روال را دارد و به طور کلی در اجرای پردازش ها از روش unix تبعیت می کند.

That operating system, xv6, provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie’s Unix operating system, as well as mimicking Unix’s internal design

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

معماری هسته 6xv به صورت یکپارچه (monolithic) است یعنی کل سیستم عامل در حالت سوپروایزر اجرا می شود.

One possibility is that the entire operating system resides in the kernel, so that the implementations of all system calls run in kernel mode. This organization is called a monolithic kernel. In this organization the entire operating system runs with full hardware privilege

- 2 یک پردازه در سیستم عامل **xv**6 از چه بخش هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازه های مختلف اختصاص می‌دهد؟

هر پردازه از دو بخش User-space memory و Per-process state private to the kernel تشکیل شده است. بخش User-space memory شامل دستورات(instructions)، اطلاعات(data) و پشته(stack) است. دستورات، کد برنامه ای است که پردازه در حال اجرا است و اطلاعات شامل متغیر ها، ثابت ها و بقیه اطلاعاتی است که توسط برنامه استفاده می‌شود و پشته قسمتی از حافظه است که برای فراخوانی ها و کنترل متغیر های محلی استفاده می‌شود. این بخش از پردازه مختص پردازه بوده و توسط بقیه پردازه ها قابل دسترسی نیست. Per-process state private to the kernel شامل اطلاعات و ساختارهای داده‌ای است که توسط هسته نگهداری می‌شوند تا اجرای یک فرایند را مدیریت و کنترل کنند. این شامل داده‌هایی مانند شناسه فرآیند (PID) است که هر فرآیند را به صورت یکتا شناسایی می‌کند، همچنین شامل سایر اطلاعات مربوط به فرآیند مانند وضعیت اجرای فعلی (اجرا، انتظار و غیره)، اولویت فرآیند، شماره‌گذاری فایل و سایر ساختارهای داده‌ای مرتبط با هسته است. این وضعیت فرآیند

به صورت خاص توسط هسته مدیریت و دسترسی دارد و به صورت مستقیم توسط خود فرآیند قابل دسترسی یا تغییر نیست.

این سیستم عامل به طور کلی پردازنده را به طریق time-share به پردازه های مختلف اختصاص می‌دهد. پردازنده بین پردازه های قابل اجرا(که منتظر اجرا شدن هستند) جا به جا می‌شود.

- 3 مفهوم **descriptor file** در سیستم عامل های مبنی بر **UNIX** چیست؟ عملکرد **pipe** در سیستم عامل **xv**6 چگونه است و به طور معمول برای چه هدفی استفاده می شود؟

هر پردازه یک آرایه خصوصی به نام ofile دارد که در آن اشاره گر ها به فایل هایی که باز کرده است وجود دارند. هنگامی که یک پردازه فایلی را باز می‌کند index ای از ofile که پوینتر به آن فایل در آن ذخیره شده باز گردانده می شود و پردازه با استفاده از این عدد می تواند در فایل بنویسید یا از آن بخواند.

هنگامی که یک پردازش دستور fopen را می دهد، kernal کوچک‌ترین fd که UNUSED است را به فایل مورد نظر اختصاص داده و آن را برمیگرداند. هنگامی که یوزر fclose را اجرا می کند سیستم fd مورد نظر را برای استفاده مجدد آزاد می کند.

Fd ها می توانند به فایل های عادی، پایپ یا device file اشاره کنند. در این حالت یوزر فارغ از اینکه تایپ فایلش چیست صرفا با یک fd کار می کند و به نوعی مانند یک interface عمل می کند.

در هر پروسه، file descriptor های اول تا سوم به ترتیب مربوط به فایل های زیر هستند:

Standard input

Standard output

Standard error

پایپ ها یک مکانیزم برای ارتباط بین پردازه های مختلف (inter-process communication) است که اجازه می دهد دو پردازه با تبادل داده ها با هم ارتباط بگیرند. با استفاده از پایپ ها می توان output یک پردازه را به input دیگری وصل کنیم.

پایپ ها توسط sysCall ()pipe ایجاد می شوند و دو fd یکی برای read end و دیگری برای write end بر می گرداند.

عملکرد پایپ ها باعث Synchronization اجرای پردازه ها نیز می شود.

برای مثال وقتی پردازه پدر تابع fork را صدا می زند، یک child process ایجاد می شود که این دو میتوانند از طریق پایپ با هم ارتباط بگیرند. هر پردازه سری که نمیخواد استفاده کند را می بندد تا در هر زمان فقط read end یا write end برای آن پردازه باز باشد. سپس پدر می تواند چیزی در پایپ بنویسد و فرزند از read end آن را بخواند.

لازم به ذکر است که عملکرد های پایپ blocking هستند یعنی اگر پردازه ای بخواهد از پایپ خالی چیزی بخواند تا زمانی که دیتایی وجود نداشته باشد پردازه بلاک خواهد شد. برای نوشتن در پایپ پر هم همینطور است.

- 4 فراخوانیهای سیستی **exec** و **fork** چه عملی انجام میدهند؟ از نظر طراحی، ادغام نکردن ان دو چه مزیتی دارد؟

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

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

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

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

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

در حقیقت اتفاقی که در سیستم‌عامل می‌افتد تا حدی در تکه کد بالا توضیح داده شده است.
تابع exec حافظه پردازه فعلی را با یک حافظه جدید که در آن یک برنامه با فایل ELF لود شده است، جایگزین می‌کند، اما file table اولیه را هم حفظ می‌کند. درواقع ()exec راهی برای اجرای یک برنامه در پردازه فعلی است. برخلاف تابع ()fork، برنامه به caller تابع ()exec بازنمی‌گردد و برنامه جدید اجرا می‌شود، مگر اینکه در زمان اجرای این تابع خطایی رخ دهد. برنامه جدید اجرا شده در یک نقطه‌ای با استفاده از تابع exit اجرای پردازه را خاتمه می‌دهد. تابع exec دو پارامتر ورودی دارد که پارامتر اول نام فایل برنامه و پارامتر دوم آرایه آرگومان‌های ورودی برنامه است.
قطعه کد زیر مثالی از اجرای این تابع را نشان می‌دهد:

```
char* args[] = { "ls", "-l", "/home", NULL }; // NULL is required

exec("/bin/ls", args);

printf("Exec failed!\n");
```

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

1. ابتدا دستور تایپ شده توسط کاربر در ترمینال را می‌خواند.
2. با استفاده از تابع fork یک پردازه جدید ایجاد می‌کند.
3. در پردازه فرزند با استفاده از تابع exec برنامه درخواست شده توسط کاربر را جایگزین پردازه فعلی (فرزند) می‌کند.
4. در پردازه پدر برای اتمام کار پردازه فرزند wait می‌کند.
5. پس از اتمام پردازه فرزند به main بازمی‌گردد و منتظر دستور جدید می‌شود.

زمانی که کاربر برای یک دستور از redirection استفاده می‌کند، تغییرات لازم در file descriptorها پس از fork و پیش از exec و در پردازه فرزند انجام می‌شود.
قطعه کد زیر این مورد را به شکل ساده نشان می‌دهد (فرض کنید دستور اجرا شده cat < in.txt است):

```
char* args = { "cat", NULL };

int pid = fork();

if (pid == 0) {

close(0); // close stdin

open("in.txt", O_RDONLY); // open in.txt for reading (fd: 0)

exec("/bin/cat", args);

printf("Exec failed!\n");

}
```

```
else if (pid > 0) {

wait();

printf("Child process has exited.\n");

}

else {

printf("The fork failed!\n");

}
```

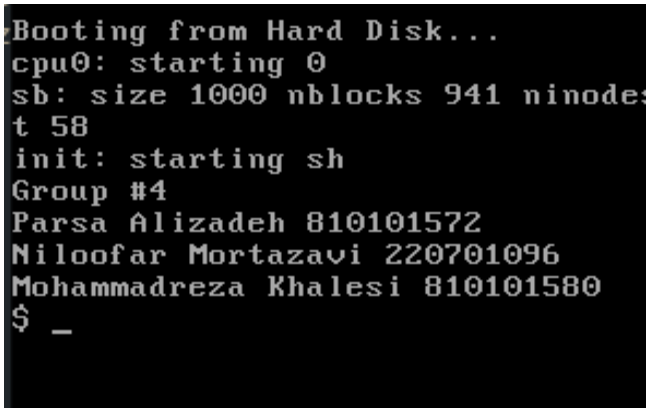
در صورتی که این دو تابع ادغام شوند، یا باید حالت‌های redirection به‌عنوان پارامتر به تابع forkexec پاس داده شوند که هندل کردن این حالت دردسرهای خودش را دارد و یا اینکه shell پیش از اجرای این تابع، file descriptorهای خود را تغییر دهد و بعد از اتمام کار این تابع نیز به حالت قبل برگرداند و یا در بدترین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

اضافه کردن چند قابلیت به **xv6**

اضافه کردن یک متن به **Boot Message**

برای نشان دادن نام اعضای گروه پس از بوت شدن کافیست در فایل **init.c** با دستور **printf** اسم ها را اضافه کنیم.

```
for(;;){
  printf(1, "init: starting sh\n");
  printf(1, "Group #4\n");
  printf(1, "Parsa Alizadeh 810101572\n");
  printf(1, "Niloofar Mortazavi 220701096\n");
  printf(1, "Mohammadreza Khalesi 810101580\n");
  pid = fork();
  if(pid < 0){
    printf(1, "init: fork failed\n");
    exit();
  }
```



با فشار دادن **ctrl+H** باید پنج دستور قبلی نمایش داده شوند. برای اینکار یک **struct** به نام **CommandHistory** تعریف می کنیم که وظیفه نگهداری دستورات قبلی را دارد.

```
struct CommandHistory {
  struct Input commands[MAX_COMMANDS]; // Buffer to store commands
  int count; // Number of commands stored
  int index; // Index for circular buffer
};
```

یک تابع تعریف کرده ایم به نام **print_history** که آن را در **consoleintr** کال کرده و در صورتی که **ctrl+H** فشرده

```
static void print_history(){

    release(&cons.lock);
    if(command_history.count < 5)
      for (int i = 0; i < command_history.count; i++)
      {
        cprintf(&command_history.commands[i].buf[command_history.commands[i].r]);
        //cprintf("if");
      }
    else
      for (int i = command_history.count - 5; i < command_history.count; i++)
      {
        cprintf(&command_history.commands[i].buf[command_history.commands[i].r]);
        //cprintf("else");
      }
    acquire(&cons.lock);
}
```

شود پنج دستور قبلی چاپ میشوند.

```
    break;
case C('H'):    // ctr+H
    print_history();
break;
```

تابع **shift_left_previous_histories** هر بار که تعداد دستورات از 10 تا بیشتر می گردد همه را شیفت داده و دستور جدید را به انتهای هیستوری ما اضافه می کند.

```
static void shift_left_previous_histories(){
  for (int i = 0; i < 9; i++) {
    command_history.commands[i] =  command_history.commands[i+1];
  }
}
```

در بخش **consolintr** از این تابع استفاده می کنیم. اگر تعداد هیستوری ها کمتر از ده تا بود صرفا دستور جدید را اضافه می کنیم.

```
default:
  if(c != 0 && input.e-input.r < INPUT_BUF){
    c = (c == '\r') ? '\n' : c;
    input_buf[input_pos++] = c;
    input.buf[input.e++ % INPUT_BUF] = c;
    consputc(c);
    if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
      input_buf[input_pos] = '\0'; // Null-terminate the input line

      // Store the command in history
      if (command_history.count == 10)
        shift_left_previous_histories();

      if(command_history.count < 10){
        command_history.count ++;
        command_history.index ++ ;
       }


      command_history.commands[command_history.index] = input;
      input.w = input.e;
      wakeup(&input.r);
    }
```

برای دستور **tab** نیز مشابه قبل از همان **struct** استفاده می کنیم و ده دستور قبل را نگه می داریم و دستور وارد شده را با آنها مطابقت می دهیم.

تابع **try_match_history** وظیفه این مقایسه را دارد و در صورتی که امکانش باشد آن را کامل میکند.

```
static void try_match_history(){
  if (input.buf[input.r] == '!')
  {
    return;
  }
  else
  {
    for (int i = command_history.count -1 ; i >= 0; i--)
  {
    int flag = 1;
    int k = command_history.commands[i].r;
    int j;
    for(j = input.r ; j < input.e; j++, k++){

      if(input.buf[j] != command_history.commands[i].buf[k]){
        flag = 0;
      }
    }
    if(flag == 1){
      for(; k < command_history.commands[i].e - 1; j++, k++){
        input.buf[input.e++ % INPUT_BUF] = command_history.commands[i].buf[k];
        consputc(command_history.commands[i].buf[k]);
      }
      return;
    }
  }
```

```
case ('\t'):
  try_match_history();
break;
```

در بخش سطح کاربر با دستور **app_name** ورودی کاراکتر به کاراکتر خوانده می شود و به ازای هر } متغیر **cnt** یکی اضافه و به ازای هر { یکی کم می شود. در انتهای حلقه اگر این متغیر غیر صفر باشد به معنای **match** نشدن براکت ها است و اگر متغیر **cnt** برابر صفر شود به معنای درستی **match** شدن است.

بعد از دادن ورودی ها با دستور **cat result.txt** فایل خروجی باز شده و خروجی مورد نظر **Wrong** یا **Right** در آن نوشته شده است.

مقدمه ای درباره سیستم عامل و **xv6**

1. سه وظیفه اصلی سیستم عامل را نام ببرید.
    - مدیریت منابع
    - مدیریت فرآیند ها
    - مدیریت فایل ها
    -
2. فایل های اصلی سیستم عامل 6xvدر صفحه یک کتاب 6xv لیست شده اند. به طور مختصر هر گروه را توضیح دهید. نام پوشه اصلی فایل های هسته سیستم عامل، فایل های سرایند و فایل سیستم در سیستم عامل لینوکس چیست؟ در مورد محتویات آن مختصرا توضیح دهید.
3.

- Basic Headers

این فایل‌ها شامل تعاریف و ثابت‌هایی هستند که در سراسر 6xv استفاده می‌شوند، مانند انواع داده، ماکروها و توابع عمومی.

- system calls

این گروه شامل کدهایی است که از طریق آنها برنامه‌های کاربری می‌توانند با هسته سیستم‌عامل تعامل کنند، مانند خواندن و نوشتن فایل‌ها، ایجاد پردازش‌ها و مدیریت حافظه.

- string operations

این فایل‌ها شامل توابعی برای کار با رشته‌ها مانند مقایسه، کپی کردن و جستجو در رشته‌ها هستند.

- low-level hardware

این بخش شامل کدهایی است که به مدیریت سخت‌افزار در سطح پایین می‌پردازند، مانند کنترل‌کننده وقفه، تایمر، و مدیریت ورودی/خروجی.

- file system

این فایل‌ها پیاده‌سازی سیستم فایل 6xv را شامل می‌شوند، از جمله مدیریت بلاک‌های دیسک، فهرستها، inode ها و عملیات فایل.

- user-level

این گروه شامل برنامه‌های کاربری پایه‌ای در 6xv مانند شل (shell) و ابزارهای کمکی دیگر است که با هسته تعامل دارند.

- entering xv6

این بخش مربوط به راه‌اندازی و ورود به 6xv است، شامل تنظیمات اولیه حافظه و راه‌اندازی پردازنده.

- Locks

این فایل‌ها شامل مکانیزم‌های همگام‌سازی مانند لاک‌های چرخشی و قفل‌های خواندن/نوشتن هستند که برای مدیریت همزمانی در 6xv استفاده می‌شوند.

- Processes

این گروه شامل کدهای مدیریت پردازش مانند زمان‌بندی، تغییر زمینه (context switch) و پیاده‌سازی پردازش‌های چندگانه است.

- Bootloader

این فایل‌ها مربوط به فرآیند راه‌اندازی اولیه 6xv هستند که پردازنده را تنظیم کرده و هسته سیستم‌عامل را از دیسک بارگذاری می‌کنند.

- Pipes

این بخش شامل پیاده‌سازی ارتباط بین پردازشی (IPC) از طریق مکانیزم لوله‌ها است که به پردازش‌ها امکان ارسال و دریافت داده را می‌دهد.

- Link

این گروه به فرآیند لینک کردن و مدیریت ماژول‌های مختلف 6xv مربوط است.

فایل‌های هسته سیستم‌عامل (Kernel)

- پوشه:/usr/src/linux/ یا /usr/src/linux-<version>/
- محتوا: شامل سورس‌کد هسته لینوکس، شامل زیرپوشه‌هایی مانند:
- kernel/ مدیریت فرآیندها، زمان‌بندی و سیستم فراخوانی‌ها
- mm/ مدیریت حافظه (paging, swapping)
- drivers/ درایورهای سخت‌افزاری (USB، شبکه، گرافیک، فایل سیستم و غیره)
- arch/ کدهای مرتبط با معماری‌های مختلف پردازنده (x86, ARM, RISC-V و غیره)

فایل‌های سرایند (Header Files)

- پوشه:/usr/include/
- محتوا: شامل فایل‌های سرایند عمومی که توابع و ساختارهای موردنیاز برنامه‌نویسان و کرنل را تعریف می‌کنند:
- stdio.h توابع ورودی/خروجی استاندارد
- stdlib.h توابع مدیریت حافظه و تبدیل داده‌ها
- unistd.h توابع سطح پایین سیستم‌عامل مانند fork, exec, read, write
- sys/ شامل هدرهای سیستمی برای مدیریت فرآیندها، حافظه، IPC و فایل سیستم

فایل سیستم (File System)

- پوشه/fs/ :(در کد منبع کرنل لینوکس)
- محتوا : شامل پیاده‌سازی سیستم فایل‌های مختلف:
- ext4/ کدهای مدیریت فایل سیستم **ext4**
- fat/ پشتیبانی از سیستم فایل FAT
- nfs/ پیاده‌سازی سیستم فایل شبکه (NFS)
- xfs/, btrfs/, tmpfs/ سیستم‌های فایل پیشرفته
- buffer.c, inode.c, super.c مدیریت کش دیسک، inodeها، و اطلاعات سوپر بلاک

کامپایل سیستم عامل **xv**6

4. دستور **make -n**را اجرا نماید. کدام دستور، فایل نهای هسته را می سازد؟

$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother

این خط کد برنامه های نهایی را لینک میکند و فایل نهایی را میسازد

5. در **Makefile** متغیرهایی به نامهای **UPROGS** و **ULIB** تعریف شده است. کاربرد آنها چیست؟

فایل های اجرایی هستند که به برنامه های سطح کاربر اجازه میدهند بدون نوشتن دوباره آنها در کد ULIB:شان

بتوانند از آنها استفاده کنند

UPROG: برنامه های سطح کاربر هستند.

6. دستور**make qemu -n** را اجرا نماید. دو دیسک به عنوان ورودی به شبیه ساز داده شده است. محتوای آنها چیست؟

هسته‌ی :**xv6**

- این دیسک شامل کرنل کامپایل‌شده‌ی **xv6** است که هنگام بوت سیستم عامل درون شبیه‌ساز بارگذاری می‌شود.
- این خروجی **xv6.img** نام دارد

سیستم فایل :**xv6**

- این دیسک حاوی تصویر سیستم فایل است که شامل دستورات اجرایی (user programs)، کتابخانه‌های موردنیاز، و داده‌های سیستم است.
- این خروجی **fs.img** نام دارد

8. علت استفاده از دستور **objcopy** در حن اجرای عملیات **make** چیست؟

در makefile سیستم عامل 6xv از دستور objcopy برای کپی کردن یک فایل آبجکت در آبجکت دیگر یا تبدیل فایل های باینری کامپایل شده به فایل باینری خام استفاده کرد. در طول فرایند make سورس کد 6xv کامپایل می شود و نتیجه آن یک فایل آبجکت برای هر سورس است. سپس این فایل ها با هم لینک می شوند و یک فایل دودویی قابل اجرا با فرمت ELF ایجاد می شود.

بعد از این مرحله از دستور objcopy برای تبدیل فایل ELF به فایل دودویی خام استفاده می شود که این فایل یک binary image برای سیستم عامل ایجاد می کند که این تصویر در طی فرآیند بوت بر روی حافظه لود می شود و توسط سخت افزار اجرا می شود.

با استفاده از این دستور فایل Makefile اطمینان حاصل می‌کند که کد کامپایل شده 6xv به یک تصویر دودویی تبدیل شده است که می‌توان آن را مستقیماً بارگذاری و اجرا کرد. این امر فرآیند بوت را ساده می‌کند و به سیستم عامل امکان اجرا بهینه روی سخت‌افزار هدف را می‌دهد.

13. کد **bootmain.c** هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در آدرس 100000**x**0 قرار می دهد. علت انتخاب ان آدرس چیست؟

در 6xv، آدرس 100000x0 (که برابر با 1 مگابایت است) برای بارگذاری هسته (kernel) انتخاب شده است. علت این انتخاب به معماری و ساختار حافظه در پردازنده‌های 86x بازمی‌گردد. در ادامه به دلایل اصلی این انتخاب اشاره می‌کنیم:

1. محدودیت‌های حافظه در حالت Real Mode

پردازنده‌های 86x هنگام راه‌اندازی در Real Mode اجرا می‌شوند که فقط به 1 مگابایت از حافظه دسترسی دارد. بخش بالایی این محدوده (معمولاً از 0000xA0 تا xFFFFF0) برای BIOS و حافظه ویدئویی رزرو شده است. بنابراین، کد بوت باید ابتدا در پایین این محدوده اجرا شود (معمولاً در 00x7C0، محل بارگذاری بوت‌لودر).

2. حفظ حافظه پایین برای BIOS و Bootloader

بوت‌لودر 6xv از bootmain.c استفاده می‌کند تا هسته را از دیسک خوانده و در آدرس 100000x0 بارگذاری کند. این کار چند دلیل دارد:

- حفظ فضای پایین حافظه (زیر 1 مگابایت) برای BIOS و بوت‌لودر.
- عدم تداخل با فضای 00x7C0 تا x9FFFF0 که ممکن است برای بوت‌لودر یا داده‌های دیگر مورد استفاده باشد.

3. پشتیبانی از Protected Mode و Paging

با ورود به Protected Mode، هسته 6xv حافظه را با Paging مدیریت می‌کند. آدرس 100000x0 یک مرز تمیز برای بارگذاری هسته است، و در آینده می‌توان به سادگی آن را در فضای آدرس‌دهی مجازی Remap کرد.

4. سازگاری با دیگر سیستم‌عامل‌های قدیمی (مانند Linux و BSD)

بسیاری از سیستم‌عامل‌های دیگر نیز آدرس 100000x0 را برای بارگذاری هسته انتخاب کرده‌اند.

18. علاوه بر صفحه بندی در حد ابتدای از قطعه بندی به منظور حفاظت هسته استفاده خواهد شد. ان عملیات توسط ()seginit انجام می گردد. همانطور که ذکر شد، ترجمه قطعه أثری بر ترجمه آدرس منطقی نمی گذارد. زیرا تمای قطعه ها اعم از کد و داده روی یکدیگر می افند. با این حال برای کد و داده های سطح کاربر پرچم USER_SEG تنظیم شده است. چرا؟

کنترل سطح دسترسی و جلوگیری از اجرای کد کاربر در سطح هسته

پردازنده‌های 86x از 4 سطح دسترسی (Privilege Levels) استفاده می‌کنند، که معمولاً:

- Ring 0: مخصوص هسته (kernel) است.
- Ring 3: مخصوص برنامه‌های کاربری (user mode programs) است.

با تنظیم بیت USER_SEG:

- کد و داده‌های سطح کاربر فقط در Ring 3 قابل دسترسی هستند.
- CPU هنگام اجرای دستورات در این بخش، سطح دسترسی را به Ring 3 کاهش می‌دهد.
- برنامه‌های کاربر نمی‌توانند به داده‌ها و کدهای هسته (Ring 0) دسترسی مستقیم داشته باشند.

بدون تنظیم این بیت، برنامه‌های کاربری می‌توانند دستورات ویژه سطح هسته را اجرا کنند که یک نقص امنیتی بزرگ خواهد بود.

جلوگیری از اجرای دستورات خاص در سطح کاربر

برخی از دستورات خاص (Privileged Instructions) فقط در Ring 0 مجاز هستند

جداسازی فضای آدرس‌دهی هسته و کاربر

در 6xv، تمامی قطعه‌های کد و داده روی یکدیگر منطبق هستند (flat segmentation)، یعنی:

- آدرس‌های منطقی و آدرس‌های خطی یکسان هستند.
- قطعه‌بندی روی ترجمه آدرس تأثیری ندارد (چراکه از paging برای مدیریت حافظه استفاده می‌شود).

اما قطعه‌بندی هنوز برای کنترل سطح دسترسی ضروری است. تنظیم USER_SEG برای کد و داده‌های کاربر، از اجرای مستقیم کدهای هسته در سطح کاربر جلوگیری می‌کند.

19. جهت نگهداری اطالعات مدیریتی برنامه های سطح کاربر ساخاری تحت عنوان **proc struct** ارائه شده است. اجزای آن را توضح داده و ساختار معادل آن در سیستم عامل لینوکس را بیابید.

این ساختار که در فایل proc.h تعریف شده از اجزای زیر تشکیل شده است:

uint sz: حجم حافظه اشغال شده توسط پردازش به بایت

pde_t* pgdir: پوینتر به page table directory entry

Page table در هر پردازش یک نگاشت بین حافظه مجازی و فیزیکی ایجاد می کند.

char *kstack: هر پردازش در kernel نیاز به یک stack جداگانه برای ذخیره حالتش دارد. این پوینتر به پایین ترین خانه kernel stack که به این پردازش اختصاص دارد اشاره می کند.

enum procstate state : وضعیت پردازش را مشخص می کند.

حالات ممکن: UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE

int pid : یک عدد یکتا برای هر پردازش با عنوان process identifier

struct proc *parent : اشاره گر به پردازش parent. وقتی پردازش پدر تابع fork را call کرده است این پردازش با همان حالت ها و مموری و ... ایجاد شده است.

struct trapframe *tf : اشاره گر به trap frame.

Trap frame آرگومان های لازم برای هندل کردن trap ها و اجرای sysCall ها را فراهم می کند و وضعیت پردازش را قبل از اجرای sysCall ذخیره می کند تا از همان حالت ادامه دهد.

struct context *context : اشاره گر به ساختار context

این struct هنگام suspend شدن پردازش و سوییچ کردن به دیگری با تابع swtch محتوای رجیستر ها را ذخیره می‌کند تا دوباره بتواند از همان جای قبلی ادامه یابد. این ساختار شامل اجزای زیر می باشد:

edi: Destination index, for string operations

esi: Source index, for string operations

ebx: Base index, for use with arrays

ebp: Stack Base Pointer, for holding the address of the current stack frame

eip: Instruction Pointer, points to instruction to be executed

void *chan: اگر مقداری غیر صفر داشته باشد یعنی پردازش در حالت sleeping است و برای انجام کاری wait می کند.

void *killed: اگر مقدار غیر صفر داشته باشد یعنی پردازش kill شده است.

struct file *ofile[NOFILE]e : آرایه ای پوینتر به فایل های باز شده توسط پردازش. هنگامی که فایلی باز می شود یک پوینتر در اولین خانه خالی این آرایه به آن فایل ذخیره می شود و index آن خانه به عنوان file descriptor باز گردانده می شود.

struct inode *cwd : current working directory

char name [16]e : نام پردازش برای debug

معادل `struct proc` در لینوکس

در لینوکس، ساختاری که اطلاعات مدیریتی پردازش‌ها را نگه می‌دارد، ساختار `task_struct` نام دارد که در فایل `include/linux/sched.h` تعریف شده است.

23. کدام بخش از آماده‌سازی سیستم،بین تمامی هسته‌های پردازنده مشترک و کدام بخش اختصاصی است؟(از هر کدام یک مورد را با ذکر دلیل توضیح دهید.) زمان‌بند روی کدام هسته اجرا می‌شود؟

در سیستم‌های چند هسته‌ای مانند 6xv، برخی بخش‌های آماده‌سازی سیستم بین تمامی هسته‌های پردازنده مشترک هستند و برخی دیگر به هر هسته اختصاص دارند. در اینجا به هر کدام یک مثال همراه با دلیل و همچنین توضیحی درباره زمان‌بند ارائه می‌شود:

بخش مشترک بین تمامی هسته‌ها: switchkvm,seginit,lapicinit,mpmain

مثال: جدول صفحه (Page Table)

جدول صفحه برای مدیریت حافظه مجازی استفاده می‌شود و معمولاً بین تمامی هسته‌ها مشترک است. زیرا تمامی هسته‌ها نیاز دارند به یک فضای آدرس مجازی یکسان دسترسی داشته باشند تا بتوانند به درستی حافظه را مدیریت کنند. در 6xv، جدول صفحه اصلی توسط هسته‌ی اصلی (Bootstrap CPU) تنظیم می‌شود و سپس توسط سایر هسته‌ها استفاده می‌شود.

بخش اختصاصی برای هر هسته:
kinitl,kvmalloc(setupkvm),mpinit,picinit,ioapicinit,consoleinit,
uartinit,pinit,tvinit,binit,fileinit,ideinit,startothers,kinit2,userinit

مثال: پشته هسته (Kernel Stack)

هر هسته‌ی پردازنده دارای یک پشته‌ی هسته‌ی اختصاصی است. این پشته برای مدیریت فراخوانی‌های سیستمی و وقفه‌هایی که توسط آن هسته پردازش می‌شوند استفاده می‌شود. از آنجا که هر هسته به طور مستقل کار می‌کند و ممکن است همزمان در حال اجرای کد هسته باشد، نیاز به یک پشته‌ی جداگانه دارد تا از تداخل و خرابی داده‌ها جلوگیری شود.

زمان‌بند (Scheduler):

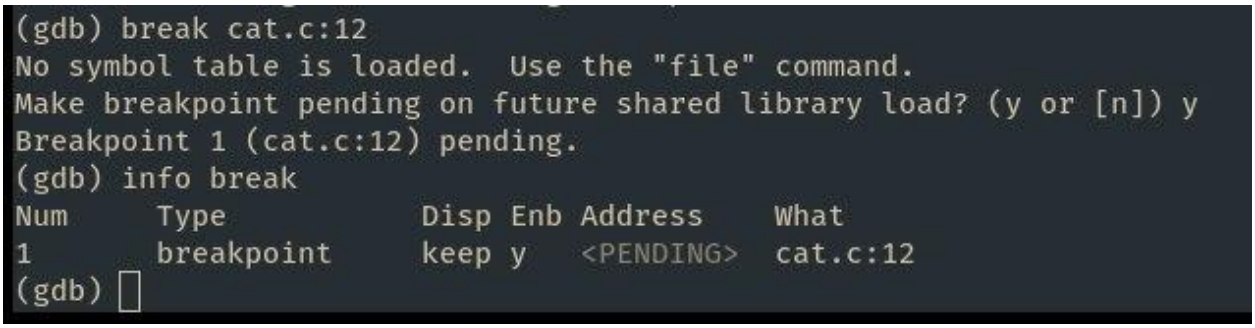
زمان‌بند روی کدام هسته اجرا می‌شود؟

زمان‌بند (Scheduler) در 6xv روی هر هسته به طور مستقل اجرا می‌شود. هر هسته‌ی پردازنده دارای یک زمان‌بند اختصاصی است که تصمیم می‌گیرد کدام فرآیند را در ادامه اجرا کند. این به این معناست که هر هسته می‌تواند به طور همزمان فرآیندهای مختلفی را اجرا کند و زمان‌بند هر هسته به طور مستقل عمل می‌کند.

اشکال زدایی

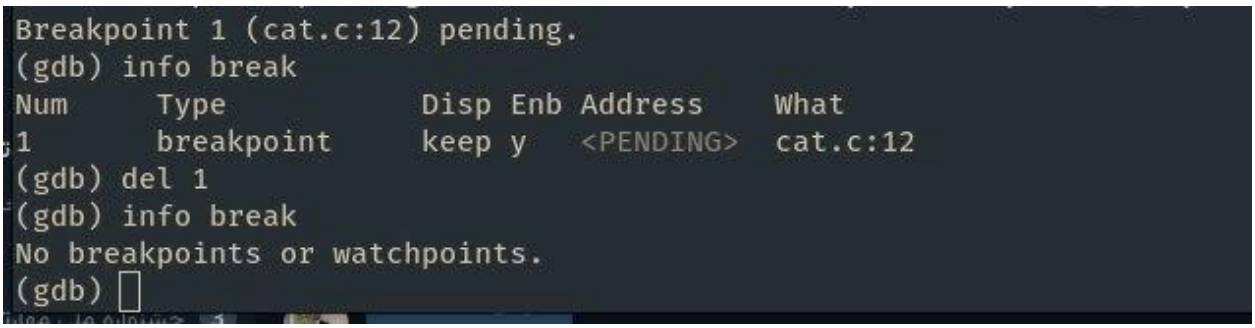
1. برای مشاهده **breakpoint** ها از چه دستوری استفاده می‌شود؟

برای این کار کافی است که از دستور info break، info breakpoint استفاده کنیم.



2. برای حذف یک **breakpoint** ازچه دستوری و چگونه استفاده می‌شود؟

برای حذف یک breakpoint میتوان ابتدا با دستور گفته شده در قسمت قبل شماره breakpoint را بدست آورد و سپس با استفاده از دستور del به صورت del <breakpoint_number>، breakpoint دلخواه را حذف کرد.



کنترل روند اجرا و دسترسی به حالت سیستم

3. خروجی دستور **bt** چه چیزی را نشان می‌دهد؟

با فراخوانی هر تابع، آن یک stack frame برای خود در نظر میگیرد و مکان فراخوانی ها (یا همان بازگشت ها) و متغیر های محلی را ذخیره می‌کند.

bt دستوری است که به وسیله آن در لحظه توقف برنامه می‌توان call stack برنامه را در لحظه توقف دید. در واقع آن خلاصه ای از مسیری است که برنامه برای رسیدن به آن نقطه طی کرده است.

خط اول مربوط frame مربوط به اخرین جایی است که برنامه در آن متوقف شده و در هر خط بعدی هر کدام صدازننده frame بالایی اش است.

با وارد کردن دستور bt به تنهایی، تمام call stack چاپ می‌شود. برای کنترل تعداد frame های چاپ شده می توان از bt n و bt -n استفاده کرد.

bt n : n frame درونی تر را چاپ می‌کند.

bt -n : n frame بیرونی‌تر را چاپ می‌کند.

```
(gdb) b swtch
Breakpoint 3 at 0x80104aab: file swtch.S, line 11.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 3, swtch () at swtch.S:11
11        movl 4(%esp), %eax
(gdb) bt
#0  swtch () at swtch.S:11
#1  0x80103ec7 in scheduler () at proc.c:346
#2  0x8010326f in mpmain () at main.c:57
#3  0x801033bc in main () at main.c:37
(gdb)
```

4. دو تفاوت دستور های **x** و **print** را توضیح دهید. چگونه می‌توان محتوای یک ثبات خاص را چاپ کرد؟

دستور **x** : همانطور که گفته شده این دستور محتویات یک خانه حافظه را نمایش می‌دهد و به اشکال زیر قابل استفاده است:

x [*Address expression*]
x /[*Format*] [*Address expression*]
x /[*Length*][*Format*] [*Address expression*

برای مشخص کردن format میتوانیم از موارد زیر استفاده کنیم:

o - octal

x - hexadecimal

d - decimal

u - unsigned decimal

t - binary

f - floating point

a - address

c - char

s - string

i - instruction

دستور **print**: مقدار متغیر داده شده را چاپ خواهد کرد و برای مثال میتوان به شکل زیر از آن استفاده کرد:

print [*Expression*]

print [*First element*]@[*Element count*]

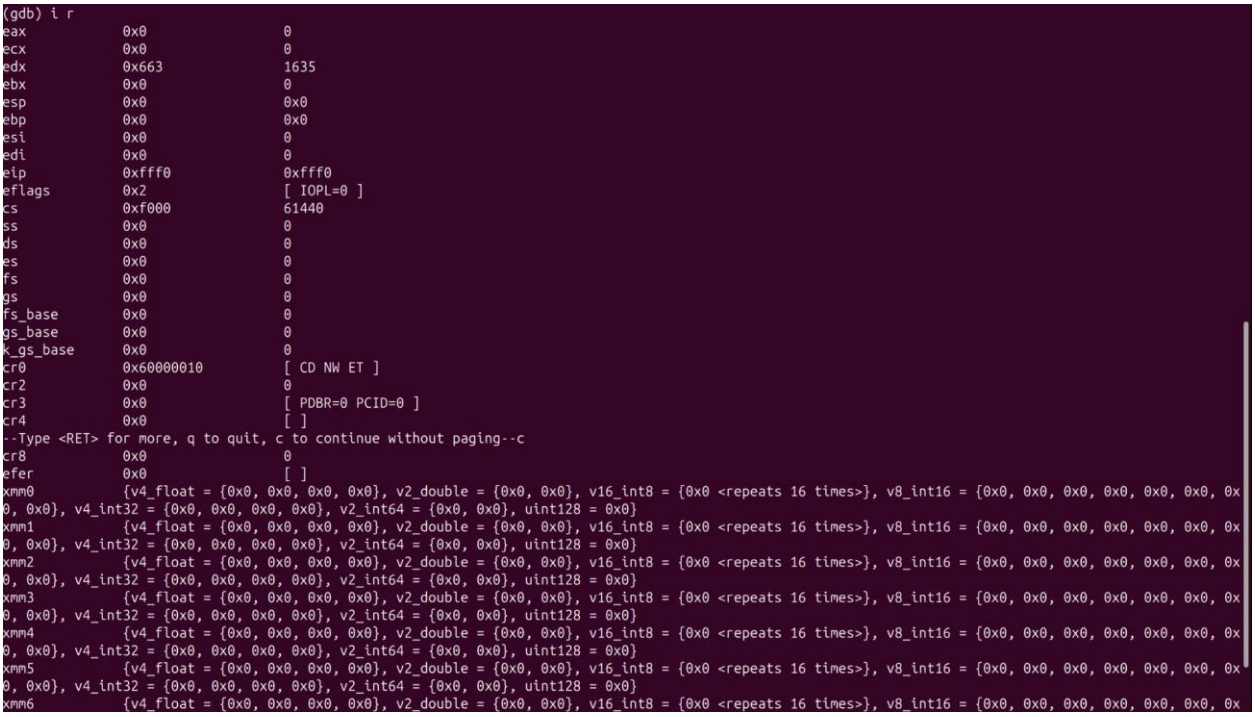
print /[*Format*] [*Expression*

```
Thread 1 hit Breakpoint 3, exec (path=0x7b6 "sh", argv=0x8dffeed0) at exec.c:20
20        struct proc *curproc = myproc();
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 3, exec (path=0x19a0 "ls", argv=0x8dfbeed0)
    at exec.c:20
20        struct proc *curproc = myproc();
(gdb) print argv[0]
$1 = 0x19a0 "ls"
(gdb) print argv[1]
$2 = 0x19a3 "-l"
(gdb) print argv[2]
$3 = 0x0
```

5. برای نمایش وضعیت ثبات ها از چه دستوری استفاده می‌شود؟ متغیرهای محلی چطور؟ در معماری **x**86 رجیستر های **edi** و **esi** نشانگر چه چیزی هستند؟

میتوان برای دیدن محتویات همه رجیستر ها از دستور info registers یا i r استفاده کرد.



برای دیدن متغیر های محلی نیز می‌توان از دستور info local استفاده کرد.

در معماری **x86**، رجیسترهای **ESI** و **EDI** معمولاً به عنوان **Source Index** (مبدأ) و **Destination Index** (مقصد) استفاده می‌شوند. این رجیسترها به‌ویژه در عملیات پردازش رشته‌ها (**String Operations)** و کپی داده‌ها نقش مهمی دارند.

رجیستر ESI: Extended Source Index

- معمولاً به‌عنوان مبدأ داده‌ها در عملیات پردازش رشته‌ها استفاده می‌شود.
- در دستورات مربوط به کپی حافظه (مانند MOVS و LODS)، این رجیستر به داده‌ای که باید خوانده شود اشاره می‌کند.
- در عملیات تکراری (مثلاً REP MOVS)، مقدار آن بعد از هر عملیات افزایش یا کاهش می‌یابد (بسته به مقدار فلگ .(DF

رجیستر EDI: Extended Destination Index

- به‌عنوان مقصد داده‌ها در عملیات پردازش رشته‌ها استفاده می‌شود.
- در دستوراتی مانند MOVS، داده‌ای که از آدرسی که ESI مشخص کرده خوانده شده، در آدرسی که EDI مشخص کرده نوشته می‌شود.
- مانند ESI، مقدار آن در عملیات تکراری افزایش یا کاهش می‌یابد.

در زمینه‌ی فراخوانی توابع، رجیسترهای EDI و ESI می‌توانند مقادیری را نگه‌داری کنند که طبق قرارداد فراخوانی (Calling Convention) در هنگام فراخوانی توابع حفظ می‌شوند، که این موضوع آن‌ها را برای ردیابی داده‌ها در بخش‌های مختلف یک برنامه مفید می‌سازد.

6. به کمک استفاده از **GDB**، درباره ساخار **input struct** موارد زر را توضح دهید:
    - توضح کلی ان **struct** و متغرهای درونی آن و نقش آنها
    - نحوه و زمان تغر مقدار متغرهای درونی.

این struct در فایل console.c قرار داده شده و از یک instance ان به نام input استفاده می‌شود.

این struct دارای 4 متغیر است:

- buf: آرایه ای به سایز 128 است که محل ذخیره خط ورودی است.
- e : نشان دهنده محل کنونی کرسر در خط است.
- w : محل شروع خط را نشان می‌دهد.
- r : برای خواندن buf بعد از زدن enter استفاده می‌شود.

در ابتدا که چیزی در ورودی نوشته نشده است مقدار متغیر های input به شکل زیر است:

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

با واردکردن عبارت elahe و سپس زدن enter ابتدا میبینیم که مقدار w برابر e است.

```
(gdb) print input
$3 = {buf = "elahe\n", '\000' <repeats 121 times>, r = 0, w = 6, e = 6}
```

سپس پس از خوانده شدن خط (در اخر switch case در اخر قسمت default) خواهیم دید که مقدار r نیز برابر مقدار wمی‌شود. در واقع r یکی یکی زیاد می‌شود و ورودی جدید رو میخواند تا به انتها(تا جایی که در آن خط ورودی نوشته شده بود) برسد.

```
(gdb) p input
$7 = {buf = "elahe\n", '\000' <repeats 121 times>, r = 6, w = 6, e = 6}
```

سپس عبارت op را در ورودی می‌نویسیم و خواهیم دید که به مقدار e دو واحد اضافه می‌شود(به اندازه طول ورودی جدید)

```
(gdb) p input
$2 = {buf = "elahe\nop", '\000' <repeats 119 times>, r = 6, w = 6, e = 8}
```

همچنین پس از زدن هر backspace نیز مقدار e یکی کم می‌شود.

در واقع input.e نشان دهنده جایی است که ما مینویسیم و با اضافه کردن هر حرف جدید یکی به آن اضافه می‌شود و با زدن هر backspace نیز یکی از مقدار آن کم می‌شود.

Inpput.w نیز در هر لحظه نشان دهنده ابتدای خطی است که در حال نوشتن هستیم. پس از وارد کردن هر enter مقدار آن برابر input.e می‌شود(می‌توان گفت به سر خط می‌رود).

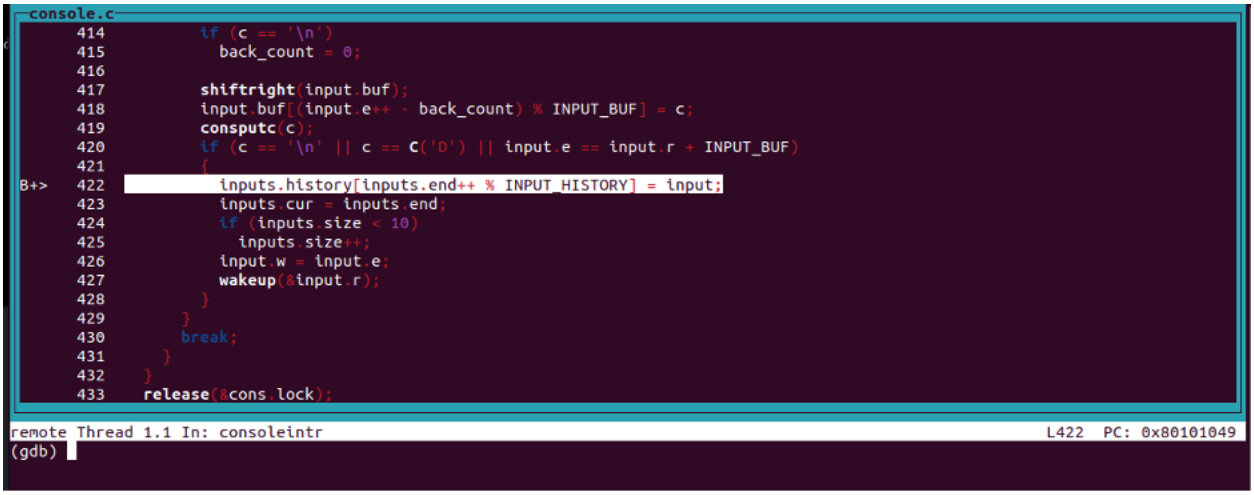
Inpput.r نیز برای خواندن بافر استفاده می‌شود. در هر مرحله پس زدن enter خط قبلی نوشته شده را میخواند.

باید عنوان کرد که در کد مربوط به پروژه آزمایشگاه به دلیل نیاز به پیاده‌سازی قابلیت های عنوان شده ما در کد خود input.e را جایی که مینویسیم در نظر نگرفته و بلکه آن را انتهای خطی که مینویسیم در نظر گرفته‌ایم.

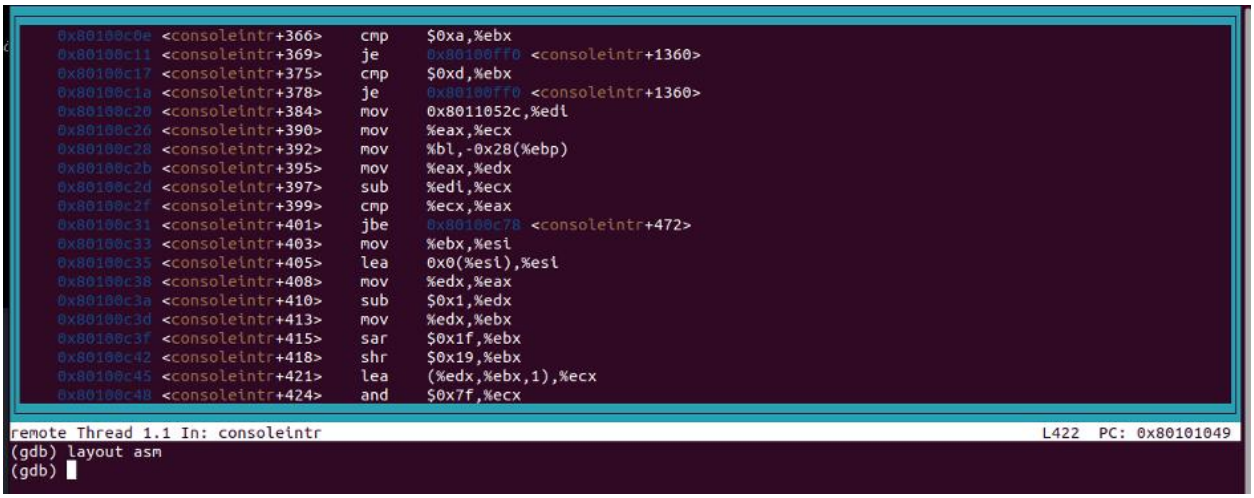
اشکال زدایی در سطح کد اسمبلی

7. خروجی دستور های **layout src** و **layout asm** در **TUI** چیست؟

با وارد کردن دستور layout src میتوانیم کد c مربوطه را هنگام توقف و دیباگ در محیط TUI ببینیم.



با وارد کردن دستور layout asm می‌توانیم کد اسمبلی را هنگام توقف و دیباگ در محیط TUI ببینیم.



میتوان با وارد کردن layout split می‌توان هر دو کد سورس و کد اسمبلی را همزمان دید.

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

میتوانیم پشته فراخوانی را با استفاده از دستور bt یا backtrace ببینیم. حال پس از آن برای جابه جایی میان توابع زنجیره فراخوانی می‌توان از دستور up و down استفاده کرد که به ترتیب به چند تابع بالاتر و چند تابع پایین تر می‌روند. میتوان تعداد توابعی که به بالا یا پایین می‌رویم را به صورت up> n> و down> n> مشخص کنیم. در صورت مشخص نکردن تعداد مقدار default برای آن 1 در نظر گرفته می‌شود.

برای مثال اگر پشته فراخوانی مانند تصویر زیر باشد:

```
(gdb) break console.c:136
Breakpoint 3 at 0x80100436: file console.c, line 146.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 3, cgaputc (c=c@entry=99) at console.c:146
146         if (c == '\n')
(gdb) bt
#0  cgaputc (c=c@entry=99) at console.c:146
#1  0x80100817 in consputc (c=99) at console.c:198
#2  consputc (c=99) at console.c:181
#3  cprintf (fmt=<optimized out>) at console.c:75
#4  0x801037c2 in mpmain () at main.c:54
#5  0x8010392c in main () at main.c:37
```

با وارد کردن دستور up به تابع consputc در خط 198 فایل console.c می‌رویم.