به نام خدا

گزارش آزمایشگاه آزمایش اول درس سیستم عامل

آیدین کاظمی: 810101561 علی زیلوچی: 810101560 بابک حسینی محتشم: 810101408

بخش اضافه کردن یک متن به Message Boot :

هربار که xv6 را اجرا می‌کنیم تصویر زیر را مشاهده می‌کنیم:

```
SeaBIOS (version 1.16.3-debian-1.16.3-2)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 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
init: starting sh
```

پس برای اضافه کردن متن به انتهای این پیام، متن آخرین خط را جستجو کردیم و متوجه شدیم که فایل init.c این پیام را چاپ می‌کند. پس از چاپ این پیام، نام اعضای گروه را چاپ می‌کنیم:

```c
#include "types.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

void
_put_name_in_console()
{
  char _name []="Welcome to xv6 modified by Babak-Aidin-Ali\n";
  printf(1,_name);
}

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    _put_name_in_console(); // Print our name after booting up
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

حال اگر سیستم‌عامل را اجرا کنیم تصویر زیر را مشاهده می‌کنیم:

```
SeaBIOS (version 1.16.3-debian-1.16.3-2)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 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
init: starting sh
Welcome to xv6 modified by Babak-Aidin-Ali
$ _
```

سوالات بخش آشنایی با سیستم عامل xv6:

1. معماری سیستم‌عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

معماری این سیستم‌عامل monolithic است. در دفاع از نظر خود می‌دانیم که xv6 پیاده‌سازی مدرنی از سیستم‌عامل Unix است که خود معماری monolithic دارد. همچنین پس از کار با این سیستم‌عامل متوجه شدیم که تمام قسمت‌های سیستم‌عامل دسترسی کامل به سخت‌افزار دارند. برای مثال می‌توانیم از هر فایل سیستم‌عامل بنویسیم یا بخوانیم که دسترسی به سخت افزار برای همه قسمت‌های یک سیستم‌عامل از ویژگی‌های معماری monolithic است.

2. یک پردازه در سیستم‌معامل xv6 از چه بخش‌هایی تشکیل شده است؟ این سیستم‌عامل به طور کلی چگونه پردازنده را به پردازه‌های مختلف اختصاص میدهد؟

در فایل proc.h ساختار داده زیر وجود دارد که همان ساختار داده یک پردازه در xv6 است:

```
35  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
36
37  // Per-process state
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52  };
```

هز پردازه در xv6 شامل حافظه‌ای در فضای کاربر که تشکیل شده از دستورات، داده‌ها و پشته است. همچنین kernel به هر پردازه یک pid می‌دهد که نشان دهنده آن پردازه خاص است. همچنین هر پردازه، استیتی که در آن قرار دارد، نام خود، پوینتری به پردازه پدر را هم نگه می‌دارد.

در فایل proc.c تابع scheduler قرار دارد:

```
315  // Per-CPU process scheduler.
316  // Each CPU calls scheduler() after setting itself up.
317  // Scheduler never returns.  It loops, doing:
318  //  - choose a process to run
319  //  - swtch to start running that process
320  //  - eventually that process transfers control
321  //      via swtch back to the scheduler.
322  void
323  scheduler(void)
324  {
325    struct proc *p;
326    struct cpu *c = mycpu();
327    c->proc = 0;
328
329    for(;;){
330      // Enable interrupts on this processor.
331      sti();
332
333      // Loop over process table looking for process to run.
334      acquire(&ptable.lock);
335      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
336        if(p->state != RUNNABLE)
337          continue;
338
339        // Switch to chosen process.  It is the process's job
340        // to release ptable.lock and then reacquire it
341        // before jumping back to us.
342        c->proc = p;
343        switchuvm(p);
344        p->state = RUNNING;
345
346        swtch(&(c->scheduler), p->context);
347        switchkvm();
348
349        // Process is done running for now.
350        // It should have changed its p->state before coming back.
351        c->proc = 0;
352      }
353      release(&ptable.lock);
354
355    }
```

می‌توان دید که در این سیستم‌عامل برای انتخاب پردازه‌ها از الگوریتم round-robin استفاده می‌شود و هر بار بین پردازه‌ها، پردازنده به پردازه‌ای که در استیت RUNNABLE است که یعنی قابل اجرا است داده می‌شود و استیت آن پردازه به RUNNING تغییر می‌کند و سپس تابع swtch صدا زده می‌شود تا محتوای پردازه فعلی را ذخیره و محتوای پردازه جدید را جدید را جایگزین کند. در این سیستم‌عامل در دو حالت بین پردازه‌ها جابه‌جا می‌شویم: یا پردازه‌ای از به دلایل مختلف مانند منتظر I/O بودن، از مکانیزم sleep استفاده کند و یا تغییر در پردازه‌ها به صورت تناوبی توسط xv6 وقتی که پردازه‌ای در حال اجرا کردن دستورات کاربر باشد.

3. مفهوم file decriptor در سیستم‌عامل‌های مبتنی بر UNIX چیست؟ عملکرد pipe در سیستم‌عامل xv6 چگونه است و به طور معمول برای چه هدفی استفاده میشود؟

File descriptor عدد صحیح کوچکی است که نشان دهنده شیئی است که یک پردازنده از آن می‌خواند یا در آن می‌نویسد. مفهوم File descriptor با انتزاعی کردن مفاهیم file، pipe، و device تفاوت بین آن‌ها را از بین می‌برد و ui یکسانی ارائه می‌دهد.

Pipe بافری کوچک در kernel است که بین چند پردازه قرار می‌گیرد و از آن هم برای خواندن و هم نوشتن استفاده می‌شود. نوشتن اطلاعات در یک سمت pipe، آن اطلاعات را در سمت دیگر در اختیار پردازه دیگر قرار می‌دهد. پردازه‌ها می‌توانند به کمک pipeها با هم ارتباط برقرار کنند. Pipeها مشابه فایل‌های موقتی عمل می‌کنند ولی 4 مزیت نسبت به یک فایل موقتی دارند: اول اینکه یک pipe خودش پس از پایان ارتباط، خودش را پاک می‌کند. دوم اینکه می‌شود با pipe مقدار زیادی داده فرستاد. سوم اینکه pipe اجازه اجرای موازی می‌دهد. چهارم اینکه در pipeها اگر پردازه‌ای سعی کند از pipe خالی بخواند، متوقف می‌شود و صبر می‌کند تا داده‌ای در pipe قرار گیرد.

4. فراخوانی‌های سیستمی exec و fork چه عملی انجام میدهند؟ از نظر طراحی، ادغام نکردن این دو چه مزیتی دارد؟

پردازه‌ها به کمک فراخوانی سیستمی fork می‌توانند پردازه‌های جدیدی را بسازند. این فراخوانی پردازه فرزند را با محتوای یکسان با پردازه پدر تشکیل می‌دهد. این فراخوانی در پردازه پدر، pid پردازه فرزند را برمی‌گرداند و در پردازه فرزند، صفر برمی‌گرداند.

فراخوانی سیستمی exec محتوای حافظه پردازه را با محتوای جدیدی که در فایلی ذخیره شده جایگزین می‌کند و پردازه را با محتوای جدید اجرا می‌کند. فایلی که در آن دستورات ذخیره شده باید فرمت خاصی به نام ELF باشد.

مزیت یکسان نکردن این دو فراخوانی سیستمی در این است که اگر این دو یکسان نباشند، shell می‌تواند پردازه‌ای را با fork بسازد، سپس از دستورات close،open و dup در پردازه فرزند استفاده کند تا تغییراتی در file descriptorها ایجاد کند و سپس از دستور exec استفاده کند.

بخش شرح پروژه:

بخش 1:

در ابتدا استراکت بافر را برای نگه داری محتویات یک بافر، شامل پوینتر های خواندن، نوشتن و ادیت و خود بافر که آرایه ای ازکاراکتر ها به طول 128 است تعریف میکنیم. از این استراکت هم در نگه داری history (که آرایه اس از استراکت ها به طول 10 خواهد بود) و هم در بافر ورودی استفاده خواهد شد:

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

#define _MOD(a,b) (a%b+b)%b
#define _N_HISTORY 10
struct _buffer _history[_N_HISTORY];
```

در اینجا باید دو تابع مهم برای فهمیدن و آپدیت مقدار اشاره گر در ترمینال را معرفی کنیم. تابع _get_cursor_pos با فرستادن چهار سیگنال به رجسیتر کنترلر vga، ابتدا دسترسی خواندن این رجیستر را گرفته و سپس یک بایت از آن را میخواند، که این کار را یک بار برای بایت بالایی و یک بار برای بایت پایینی انجام میدهد و پس از اتمام خواندن به همان شکل در متغیر _pos ذخیره کرده، به عنوان محل قرار گیری اشاره گر باز میگرداند:

```
#define ___HIGH_BYTE_CUR 14
#define ___LOW_BYTE_CUR 15

int
_get_cursor_pos()
{
  int _pos;
  // Cursor position: col + 80*row.
  outb(CRTPORT, ___HIGH_BYTE_CUR);
  _pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, ___LOW_BYTE_CUR);
  _pos |= inb(CRTPORT+1);
  return _pos;
}
```

تابع _update_cursor وظیفه آپدیت کردن مکان اشاره گر با مقداری دلخواه را دارد. این تابع نیز ابتدا دسترسی نوشتن در رجیستر مورد نظر را پیدا کرده، سپس بایت بالایی و پایینی آن را با مقدار دلخواه ورودی

تغییر میدهد. همچنین میتوان سیمبل قرار گیرنده بر روی ترمینال را نیز تغییر داد که در این پروژه استفاده ای از آن نمیشود:

```
void
_update_cursor(int _pos, char symbol)
{
  outb(CRTPORT, ___HIGH_BYTE_CUR);
  outb(CRTPORT+1, _pos>>8);
  outb(CRTPORT, ___LOW_BYTE_CUR);
  outb(CRTPORT+1, _pos);
  if(symbol!=0)
    crt[_pos] = symbol | 0x0700;
}
```

یک متغیر مهم _arrow میباشد که به صورت گلوبال تعریف شده و وظیفه مشخص کردن مقدار عقب بودن اشاره گر از پوینتر e بافر را دارد. مقدار این متغیر در صورتی که وسط جمله باشد، منفی فاصله آن تا سر جمله و در صورتی که سر جمله باشد صفر است:

```
int _arrow = 0;
```

تابع بعدی _arrow_key_console_handler میباشد که در این قسمت فقط دو بخش ابتدایی آن توضیح داده میشود. این تابع با ورودی گرفتن یک کاراکتر ابتدا مقدار فعلی اشاره گر در صفحه را گرفته، سپس در صورتی که کلید سمت چپ باشد، بررسی میکند که به انتهای سمت چپ نرسیده باشیم و در صورت برقرار بودن شرط پوزیشین اشاره گر را یکی کم کرده، متغیر _arrow را نیز یکی کم میکند و اشاره گر را روی صفحه آپدیت میکند. در صورتی که کلید راست فشرده شود و _arrow صفر نباشد، همین کار را به بعلاوه یک کردن تکرار میکند:

```
#define _UP_ARROW 0xe2
#define _DOWN_ARROW 0xe3
#define _LEFT_ARROW 0xe4
#define _RIGHT_ARROW 0xe5
```

```
void
_arrow_key_console_handler(int c)
{
  int _pos=_get_cursor_pos();
  if(c == _LEFT_ARROW)
  {
    if(-_arrow < input.e)
    {
      --_pos;
      _arrow--;
      _update_cursor(_pos,0);
    }
  }
```

```
  else if(c == _RIGHT_ARROW)
  {
    if(_arrow)
    {
      ++_pos;
      _arrow++;
      _update_cursor(_pos,0);
    }
  }
```

در اینجا باید چند تابع مهم و پر کاربرد معرفی شوند. ابتدا تابع ___shift_buf که محل شیفت پیدا کردن را گرفته و جهت شیفت را نیز میگیرد و به یکی از دو سمت چپ یا راست بافر را از محل مشخص شده شیفت میدهد و مقدار پوینتر e را یکی کم یا زیاد میکند (در شیفت راست خانه مشخص شده شیفت تغییر نمیکند و در شیفت چپ پاک میشود):

```
void
___shift_buf(int right_flag, int change_idx){
    if (right_flag)
    {
        for (int i = input.e; i > change_idx; i--)
        {
            input.buf[i % INPUT_BUF] = input.buf[(i-1) % INPUT_BUF];
        }
        ++input.e;
    }
    else
    {
        for (int i = change_idx-1; i < input.e; i++)
        {
            input.buf[i % INPUT_BUF] = input.buf[(i+1) % INPUT_BUF];
        }
        --input.e;
    }
}
```

تابع بعدی ___update_buffer میباشد که وظیفه قرار دادن یک کاراکتر جدید داخل بافر را دارد. در صورتی که کاراکتر بک اسپیس باشد، از محل تعیین شده بافر را به چپ شیفت میدهد، و در غیر این صورت از محل تعیین شده بافر را یکی به راست شیفت داده و کاراکتر جدید را جایگذاری میکند:

```
#define ___BACKSPACE 8

void ___update_buffer(int c, int change_idx)
{
    if (c <= 0)
        return;
    if (c == BACKSPACE || c == ___BACKSPACE)
    {
        ___shift_buf(0,change_idx);
    }
    else
    {
        ___shift_buf(1,change_idx);
        input.buf[(change_idx)%INPUT_BUF]=c;
    }
}
```

تابع بعدی ___clear_cmd میباشد که وظیفه خالی کردن ترمینال را دارد. این تابع ابتدا اشاره گر را به انتهای راست جمله روی ترمینال برده، سپس به تعداد کاراکتر های جمله بک اسپیس وارد ترمینال میکند تا ترمینال خالی شود:

```
void
___clear_cmd(int end)
{
  int _cursor_pos = _get_cursor_pos();
  _update_cursor(_cursor_pos-_arrow,0);
  for (int i = 0; i < end - input.w; i++) {
    consputc(BACKSPACE);
  }
}
```

تابع بعدی _write_from_buffer میباشد که مسئول نوشتن تمام کاراکتر های بافر به روی ترمینال است. این تابع تا رسیدن به کاراکتر نال یا نیو لاین کاراکتر های بافر را روی ترمینال قرار میدهد، و نهایتا پوینتر های خواندن و نوشتن از بافر را ریست میکند:

```
void
_write_from_buffer()
{
  for (int i = 0; i < INPUT_BUF; i++)
  {
    if(input.buf[i]=='\n' || input.buf[i]=='\0')
      break;
    consputc((int)input.buf[i]);
  }
  input.w=0;
  input.r=0;
}
```

حال تابع بسیار مهم _input_in_mid را میتوان از کنار هم قرار دادن این توابع بالا ساخت، که وظیفه آن مدیریت کاراکتر جدید وارد شده در وسط جمله میباشد. این تابع با گرفتن کاراکتر ورودی، مکان وارد کردن را با جمع کردن متغیر های e و _arrow پیدا کرده و با توجه به بک اسپیس بودن یا نبودن آن، بافر را آپدیت کرده، ترمینال را پاک و بافر جدید را نوشته و موقعیت اشاره گر جدید را نیز آپدیت میکند:

```
void
_input_in_mid(int c)
{
  int _cursor_pos=_get_cursor_pos();
  int change_index = input.e + _arrow;
  if(c==BACKSPACE || c == ___BACKSPACE)
  {
    if (_arrow <= -input.e) // no extra backspace allowed
      return;
    ___update_buffer(c,change_index);
    ___clear_cmd(input.e + 1);
    _write_from_buffer();
    _update_cursor(_cursor_pos-1,0);
  }
  else
  {
    ___update_buffer(c,change_index);
    ___clear_cmd(input.e - 1);
    _write_from_buffer();
    _update_cursor(_cursor_pos+1,0);
  }
}
```

حال با کمک این تابع و تابع _arrow_key_console_handler ، تابع consoleintr که وظیفه مدیریت کلید های ورودی را دارد را در سه جا آپدیت میکنیم:

1- هنگامی که ورودی بک اسپیس است، در صورتی که در وسط جمله باشیم (_arrow صفر نباشد) باید از _input_in_mid استفاده شود:

```
case C('H'): case '\x7f': // Backspace
    if(input.e != input.w){
      if (_arrow==0)
      {
        input.buf[--input.e] = '\0';
        consputc(BACKSPACE);
      }
      else
      {
        _input_in_mid(BACKSPACE);
      }
    }
    break;
```

2- در صورتی که کلید های چپ و راست (یا بالا و پایین که در قسمت های بعد توضیح داده خواهد شد) فشرده شوند باید از _arrow_key_console_handler استفاده شود:

```
// Handle arrow keys
case _LEFT_ARROW:
case _RIGHT_ARROW:
case _UP_ARROW:
case _DOWN_ARROW:
    _arrow_key_console_handler(c);
    break;
```

3- در صورتی که ورودی عادی باشد و در وسط جمله باشیم نیز باید از _input_in_mid استفاده شود:

```
default:
    if(c != 0 && input.e-input.r < INPUT_BUF){
        c = (c == '\r') ? '\n' : c;

        if(c=='\n')
            _arrow=0;

        if (_arrow==0)
        {
            input.buf[input.e++ % INPUT_BUF] = c;
            consputc(c);
        }
        else
        {
            _input_in_mid(c);
        }
```

بدین ترتیب فانکشنالیتی های چپ و راست رفتن روی ترمینال و وارد کردن متن در وسط جمله کاملا مدیریت میشوند.

```
$ operating-system
```

بخش 2:

```
#define _MOD(a,b) (a%b+b)%b
#define _N_HISTORY 11
struct _buffer _history[_N_HISTORY];
int _current_history=0;
int _last_history=0;
int _arrow = 0;
```

برای این بخش از متغیرهای بالا استفاده کردیم.

در زبان c اوپراتور % به صورت باقی‌مانده تعریف شده است اگر سمت چپ این اوپراتور عددی منفی باشد حاصل نیز منفی خواهد بود پس تابع MOD_ را به صورت بالا تعریف کردیم که مقدار a را در پیمانه همنهشتی b به ما می‌دهد.

ثابت N_HISTORY_ تعداد حداکثر دستوراتی است که با دستور history می‌توان به دست آورد که طبق صورت سوال برابر 11 گذاشتیم که شامل دستور فعلی و 10 دستور پیشین می‌شود.

متغیر history_ آرایه‌ای از جنس buffer_ به طول 10 است که 10 input اخیر را در آن ذخیره می‌کنیم. با این متغیر به صورت حلقوی رفتار می‌کنیم یعنی بعد از 10 دوباره به خانه اول برمی2گردیم

متغیر current_history_ جایگاه خانه فعلی در history_ را نشان می‌دهد.

متغیر last_history_ جایگاه آخرین دستور را نشان می‌دهد که اگر بخواهیم به دستورات گذشته جابه‌جا شویم، محتوای فعلی input را در این خانه از history_ ذخیره می‌کنیم.

برای این بخش چند تابع مختلف تعریف کردیم. اولی تابع buffer_with_str_cmp_ است. این تابع رشته‌ای را به عنوان ورودی می‌گیرد و محتوای بافر ورودی را با آن مقایسه می‌کند و در صورت یکسان بودن 0 وگرنه 1 برمی‌گرداند. از این تابع برای تشخیص اینکه محتوای بافر history است یا نه استفاده می‌کنیم.

```
365  int
366  _buffer_with_str_cmp(char* target_str)
367  {
368    int i=0;
369    while(target_str[i++]!='\0')
370      if (input.buf[i]!=target_str[i])
371        return 1;
372    return 0;
373  }
```

تابع بعدی handle_custom_commands_ است که در این تابع با کمک تابع قبل محتوای بافر ورودی با دستوراتی مشخص که در این پروژه فقط یک دستور history بود مقایسه می‌کند و در صورت یکسان بودن، توابع لازم را صدا می‌کند.

```
375   void
376   _handle_custom_commands()
377   {
378     char _history_str[]="history\n";
379     if (!_buffer_with_str_cmp(_history_str)) {
380       _history_command();
381       input.e = input.w = input.r = 0;
382     }
383   }
```

تابع بالا اگر بافر ورودی کلمه history بود، تابع _history_command را صدا می‌کند. این تابع محتوای تمام ورودی‌های ذخیره شده در _history را چاپ می‌کند. البته در ابتدای کار کمتر از 10 ورودی در آرایه ذخیره شده است. پس این تابع هر دفعه بررسی می‌کند که اگر آن خانه آرایه خالی بود، متوقف شود.

```
void
_history_command()
{
  release(&cons.lock);
  cprintf("Command history:\n");
  cprintf("-------------------------------------------------------------------------------\n");
  for (int i = 0; i < _N_HISTORY-1; i++) {
    if (_history[_MOD(_current_history-i-1,_N_HISTORY)].buf[0]=='\0')
      break;
    cprintf("*%d: %s", i + 1, _history[_MOD(_current_history-i-1,_N_HISTORY)].buf);
  }
  cprintf("\n$ ");
  acquire(&cons.lock);
}
```

در تابع consoleread این شرط را تغییر دادیم. پس از این‌که حرف نیولاین وارد شد، این تابع ورودی فعلی را در خانه _last_history آرایه _history ذخیره می‌کند و _last_history را یکی افزایش می‌دهد. همچنین با اضافه شدن input به _history، _current_history آپدیت می‌شود تا نشان دهنده جایگاه ورودی جدید باشد و ورودی جدیدی با مقادیر خالی تشکیل می‌دهیم و input را برابر آن قرار می‌دهیم.

```
606     if(c == '\n')
607     {
608       _history[_MOD(_last_history++,_N_HISTORY)]=input;
609       _current_history=_last_history;
610       struct _buffer new_input={"",0,0,0};
611       input=new_input;
612       break;
613     }
614   }
```

و بدین‌ترتیب دستور history به درستی کار می‌کند:

```
$ history
Command history:
--------------------------------------------------------------------------------
*1: Tehran
*2: University
*3: CEUT403
*4: 810101560
*5: 810101561
*6: 810101408
*7: ali
*8: aidin
*9: babak
*10: operating-system
```

در بخش قبل قسمت مربوط به کلید فلش چپ و راست تابع arrow_key_handler_ را توضیح دادیم. حال قسمت مربوط به فلش بالا و پایین این تابع را توضیح می‌دهیم.

```
else
{
  if (c == _UP_ARROW)
    if(_history[_MOD(_current_history-1,_N_HISTORY)].buf[0]=='\0' ||
    _MOD(_current_history-1,_N_HISTORY)==_MOD(_last_history,_N_HISTORY))
      return;
  if (c == _DOWN_ARROW)
    if(_MOD(_current_history+1,_N_HISTORY)==_MOD(_last_history+1,_N_HISTORY))
      return;
  input.buf[input.e]='\n';
  ___clear_cmd(input.e);
  _arrow=0;
  input.w=++input.e;
  if (_MOD(_current_history,_N_HISTORY)==_MOD(_last_history,_N_HISTORY))
    _history[_MOD(_current_history,_N_HISTORY)]=input;
  if(c == _UP_ARROW)
    _current_history--;
  else
    _current_history++;
  input=_history[_MOD(_current_history,_N_HISTORY)];
  input.e--;
  _write_from_buffer();
}
```

ابتدا بررسی می‌کنیم که خانه‌ای از history_ که سعی داریم به آن برویم خالی نباشد و همچنین به خانه انتهایی history_ نرسیده باشیم وگرنه نباید تغییری در ورودی فعلی بدهیم و همین‌جا return می‌کنیم. چون می‌خواهیم رشته فعلی کلا برود و تمام رشته‌های قبلی با نیولاین می‌رفتند و انتهایشان نیولاین قرار می‌گرفت، انتهای رشته نیز نیولاین می‌گذاریم. سپس صفحه را پاک می‌کنیم. بعد تنها در صورتی که مکان قبلی همان last_history_ بود، تغییرات اعمال شده را در history_ ذخیره می‌کنیم. و در نهایت input را برابر input مورد نظر می‌گذاریم و محتوای آن را بر صفحه چاپ می‌کنیم.

بخش 3:

در این بخش باید ctrl + s و ctrl + f مدیریت شوند، که این کار توسط تابع ___handle_ctrl_s انجام میشود. این تابع از دو بخش تشکیل شده است:

1- بخش اول تا زمانی که ctrl + f نیامده وظیفه دارد کارایی عادی ترمینال را انجام داده و بافر را آپدیت کند. ابتدا آرایه ای برای ذخیره اندیس حروف ورودی تخصیص داده و سپس کاراکتر ورودی را تا رسیدن ctrl + f بررسی میکند. در صورتی که کاراکتر ورودی نامعتبر یا کلید های بالا یا پایین یا ctrl + s باشد کاری انجام نمیشود (چرا که در حین این عملیات نمیتوان به تاریخچه دسترسی پیدا کرد). در صورتی که کاراکتر وارد شده بک اسپیس باشد و اشاره گر در انتهای سمت راست قرار داشته باشد، یک کاراکتر بک اسپیس روی ترمینال نوشته، بافر را از انتها یکی کم میکند و در صورتی که کاراکتر آخر جزو کاراکتر های جدید الورود بود آن را از آرایه اندیس ها حذف میکند. در صورتی که کلید چپ و راست وارد شوند، مکان اشاره گر را آپدیت میکند و در غیر این صورت، کاراکتر وارد شده جدید را به انتهای راست یا در وسط جمله در حال نوشتن اضافه میکند و مقادیر اندیس های ورودی را یکی به چپ یا راست شیفت میدهد:

```
void
___handle_ctrl_s(int (*getc)(void))
{
  int ___inputs_idx[INPUT_BUF] ;
  for (int i = 0; i < INPUT_BUF; i++)
    ___inputs_idx[i] = 0;

  int c;
  while((c = getc()) != C('F'))
  {
    if (c <= 0 || c == _UP_ARROW || c == _DOWN_ARROW || c == C('S'))
      continue;

    else if (c == ___BACKSPACE && _arrow == 0)
    {
      input.buf[--input.e] = '\0';
      consputc(BACKSPACE);
      if(___inputs_idx[input.e] == 1)
        ___inputs_idx[input.e] = 0;
    }

    else if (c == _LEFT_ARROW || c == _RIGHT_ARROW)
    {
      _arrow_key_console_handler(c);
    }
```

```
    else if(input.e-input.r < INPUT_BUF)
    {
      c = (c == '\r') ? '\n' : c;
      if (_arrow == 0)
      {
        ___inputs_idx[input.e] = 1;
        input.buf[input.e++ % INPUT_BUF] = c;
        consputc(c);
      }
      else
      {
        _input_in_mid(c);
        if (c == BACKSPACE || c == ___BACKSPACE)
        {
          for (int i = input.e + _arrow - 1; i < INPUT_BUF - 1; i++)
            ___inputs_idx[i] = ___inputs_idx[i+1];
        }
        else
        {
          for (int i = INPUT_BUF - 1; i > input.e + _arrow - 1 ; i--)
            ___inputs_idx[i] = ___inputs_idx[i-1];
          ___inputs_idx[input.e + _arrow - 1] = 1;
        }
      }
    }
```

همچنین توابع سوال 4 نیز در انتهای این بلاک آورده شده تا ویژگی های آن قسمت نیز در اینجا قابل انجام باشد:

```
    struct __exp_found exp = __find_expression();
    if (exp.success_flag)
    {
      int prev_e = input.e;
      int init_cursor_pos = _get_cursor_pos(), init_arrow = _arrow;
      int line_start = init_cursor_pos - init_arrow -prev_e;
      ___clear_buf_with_range(exp.start_exp_idx,exp.exp_size);
      ___put_num_in_buf(exp.num_str,exp.num_size,exp.start_exp_idx);
      int shift_count = exp.exp_size - exp.num_size;
      int change_idx = exp.start_exp_idx + exp.exp_size ;
      ___shift_buf_many_times(shift_count,change_idx);
      _arrow = exp.start_exp_idx + exp.num_size - input.e;
      _update_cursor( line_start + exp.start_exp_idx + exp.exp_size,0);// moving curser so clear cmd works
      ___clear_cmd(prev_e);
      _write_from_buffer();
      _update_cursor( line_start + exp.start_exp_idx + exp.num_size,0);
    }
  }
```

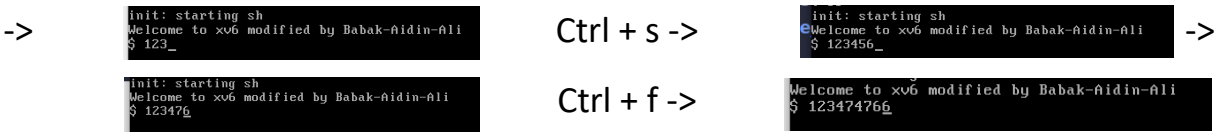
2- پس از دریافت ctrl + f، مقدار جدید e را در current_e ذخیره میکند تا در آینده بعدا استفاده شود. سپس بافر را دیپ کپی کرده و در یک حلقه for مقادیر وارد شده توسط کاربر (که اندیس آنها در آرایه ___inputs_idx ذخیره شده است) را از جایی که اشاره گر قرار دارد (با کمک _arrow) در بافر وارد میکند. دست آخر ترمینال ورودی را پاک کرده (که سایز آن هنوز تغییر نکرده و برابر سایز ابتدای عملیات است) و مقدار بافر را دوباره روی ترمینال مینویسد:

```
release(&cons.lock);

int current_e = input.e;
char buf_copy[INPUT_BUF];
for (int i = 0; i < INPUT_BUF; i++) // hard copy
  buf_copy[i] = input.buf[i];

int change_idx = current_e + _arrow;

for(int i = 0; i < INPUT_BUF; i++)
  if (___inputs_idx[i] == 1)
    ___update_buffer(buf_copy[i],change_idx++);

___clear_cmd(current_e);
_write_from_buffer();
acquire(&cons.lock);
}
```

نهایتا این تابع هنگام دریافت ctrl + s توسط consoleintr صدا زده شده و پس از انجام عملیات، مقدار اشاره گر روی صفحه به اندازه _arrow عقب میرود تا از ناهمگامی اشاره گر و بافر جلوگیری شود. بدین ترتیب کارایی های ctrl + s و ctrl + f به طور کامل مدیریت میشوند:

```
513     //handle ctrl + s
514     case C('S'):{
515       ___handle_ctrl_s(getc);
516       int pos = _get_cursor_pos();
517       _update_cursor(pos + _arrow,0);
518     }
519       break;
```

چند مثال از کارکرد این دستور:



بخش 4:

در این بخش باید در صورت مشاهده یک الگوی خاص در ورودی محاسبات ریاضی را انجام دهیم که برای این منظور از موارد زیر کمک گرفته‌ایم:

1. ساختار __exp_found__ که اگر الگوی مورد نظر توسط توابعی که در ادامه معرفی خواهند شد یافت شد، پاسخ آنرا بعد از محاسبه به صورت کاراکتر به همراه طول ورودی و خروجی و اندیس شروع الگو به توابع مربوط به انتقال به بافر تحویل میدهد تا آنها تغییرات لازم را ایجاد کنند.

```
struct __exp_found {
    char num_str[INPUT_BUF];
    int num_float;
    int num_size;
    int start_exp_idx;
    int exp_size;
    int success_flag;
} dum;
```

2. تابع __is_in_arr__ وجود یک کاراکتر در یک آرایه از کاراکترها را تعیین میکند که در ادامه برای تشخیص اعداد و عملگرها به کار خواهد آمد

```
int __is_in_arr(char inc, char* arr) {
  char c = 'a';
  int i = 0;
  while (c != '\0')
  {
    c = arr[i];
    if (c == inc)
      return 1;
    i += 1;
  }
  return 0;
}
```

3. از آنجایی که محاسبات ما عددی هستند ولی ورودی و خروجی باید به صورت کاراکتر باشند؛ توابع __char_to_int__، __int_to_char__ و __float1p_to_char__ برای رسیدگی به این مورد تعریف شده‌اند.

```
struct __exp_found
__int_to_char(int n, struct __exp_found exp) {
  int t = n;
  int sign =0;
  int l = 0;
  if (n == 0)
  {
    exp.num_size = 1;
    exp.num_str[0] = '0';
    return exp;
  }
  if (n < 0)
  {
    exp.num_str[l] = '-';
    sign++;
    l++;
    t = -t;
    n = -n;
  }

  while (t > 0.9)
  {
    t /= 10;
    l++;
  }
  for (int i = sign; i < l; i++)
  {
    exp.num_str[l-i-(sign?0:1)] = (n % 10) + '0'; // if negative start from one place in right
    n /= 10;
  }
  exp.num_size = l;
  return exp;
}
```

```
struct __exp_found
__float1p_to_char(float n, struct __exp_found exp) {
  float temp = n * 10;
  int t = temp;
  int l = 0;
  int sign = 0;
  if (n < 0)
  {
    exp.num_str[l] = '-';
    sign++;
    l++;
    t = -t;
    temp = -temp;
  }
  while (t > 0.9)
  {
    t /= 10;
    l++;
  }
  t = temp;
  for (int i = sign; i < l; i++)
  {
    exp.num_str[l - i- (sign?0:1)] = (t % 10) + '0';
    t /= 10;
  }

  // handling floating point
  l += 1;
  exp.num_str[l - 1] =  exp.num_str[l-2];
  exp.num_str[l-2] = '.';
  exp.num_size = l;
  return exp;
}
```

```
int __char_to_int(char* arr, int n) {
  int result = 0;
  for (int i = 0; i <= n; i++)
    result = 10 * result + arr[i] - '0';
  return result;
}
```

4. در ادامه تابع solve_exp___ را برای انجام محاسبات بعد از کشف الگو داریم. این تابع کاراکتر اعداد را از الگو استخراج کرده و به کمک توابع بخش قبل آنها را به فرم int در می آورد؛ سپس با توجه به اینکه چه اپراتوری باید اجرا شود، محاسبه را انجام داده و حاصل را به کمک توابع بخش قبل به کاراکتر در فرم یک exp_found___ خروجی میدهد.

```
struct __exp_found
__solve_exp(char* txt, int break_index, int end_index)
{
  struct __exp_found exp;
  exp.success_flag = 1;
  exp.exp_size = end_index + 2 + 1; // 2 for =? and 1 for index

  int num1 = __char_to_int(&txt[0], break_index);
  int num2 = __char_to_int(&txt[break_index + 2], end_index - 2 - break_index);
  int result = -1;
  switch (txt[break_index + 1])
  {
  case '+':{
    result = num1 + num2;
    exp.num_float = result;
    exp = __int_to_char(result, exp);
    }break;

  case '-':{
    result = num1 - num2;
    exp.num_float = result;
    exp = __int_to_char(result, exp);
    }break;

  case '*':{
    result = num1 * num2;
    exp.num_float = result;
    exp = __int_to_char(result, exp);
    }break;

  case '/':{
    float num1_float = num1;
    float r = num1_float / num2;
    exp.num_float = r;
    exp = __float1p_to_char(r, exp);
    }break;

  default:
    break;
  }

  for (int i = exp.num_size; i < INPUT_BUF; i++)
    exp.num_str[i] = '\0';

  return exp;
}
```

5. در نهایت تابع find_expression___ را داریم این تابع روی input buffer به صورت یک state machine به دنبال الگوی مورد نظر سوال می گردد، هر زمان که به state آخر رسید(الگو را کامل مشاهده کرد)، اطلاعات الگو را به تابع solve_exp___ که در بخش قبل گفته شد، میدهد و نتیجه این تابع را بعد تنظیم بعضی متغیر های ساختار exp_found___ خروجی میدهد.

```
struct __exp_found
__find_expression()
{
  struct __exp_found no_exp;
  no_exp.success_flag = 0;
  char c = 'a';
  char nums[11] = "0123456789";
  char ops[5] = "+-*/";
  int i = 0;
  int s = 0;
  int num1_start, num1_end, num2_end;
  while (c != '\0')
  {
    c = input.buf[i];
    switch (s)
    {
    case 0:{
      if (__is_in_arr(c,nums))
      {
        num1_start = i;
        num1_end = i;
        s = 1;
      }
      }break;

    case 1:{
      if (__is_in_arr(c,nums))
        num1_end = i;
      else if (__is_in_arr(c,ops))
        s = 2;
      else
        s = 0;
      }break;

    case 2:{
      if (__is_in_arr(c,nums))
      {
        num2_end = i;
        s = 3;
      }
      else
        s = 0;
      }break;

    case 3:{
      if (__is_in_arr(c,nums))
        num2_end = i;
      else if (c == '=')
        s = 4;
      else
        s = 0;
      }break;

    case 4:{
      if (c == '?')
        s = 5;
      else
        s = 0;
      }break;

    case 5:{
      struct __exp_found found_exp = __solve_exp(&input.buf[num1_start], num1_end - num1_start, num2_end - num1_start);
      found_exp.start_exp_idx = num1_start;
      return found_exp;
      }break;

    default:
      s = 0;
      break;
    }
    i += 1;
  }
  return no_exp;
}
```

نهایتا با وارد کردن کد های استفاده شده به تابع consoelintr، پس از صدا کردن فانکشن ها میتوان نتایج دلخواه این بخش را گرفت. برای آنکه تغییرات اعمال شده بر روی اکسپرشن وابسته به توالی ورود کاراکتر ها نباشد، لازم است هر بار که تغییری روی بافر اعمال شد یک بار این توابع بافر را برای پیدا کردن اکسپرشن چک کنند.

ابتدا لازم است توابع مورد نیاز این عملیات توضیح داده شوند:

```
void
___clear_buf_with_range(int start, int exp_size){
  for (int i = start; i < start + exp_size; i++)
    input.buf[i] = '\0';
}

void
___put_num_in_buf(char* num, int num_size, int start){
  for (int i = start; i < num_size + start; i++)
    input.buf[i] = num[i - start];
}

void
___shift_buf_many_times(int shift_count, int change_idx){
  for (int i = 0; i < shift_count; i++)
    ___shift_buf(0,change_idx - i);
};
```

تابع اول وظیفه دارد بخشی از بافر را پاک کرده و با کاراکتر نال جایگذاری کند. تابع دوم وظیفه دارد یک عدد به فرم رشته را از یک مکان مشخص وارد بافر کند. تابع سوم نیز وظیفه دارد بافر را از یک نقطه مشخص شده به تعداد دلخواه به چپ شیفت دهد.

حال با کنار هم قرار دادن این توابع، عملیات مورد نیاز بر روی بافر را انجام میدهیم:

```
struct __exp_found exp = __find_expression();
if (exp.success_flag)
{
  int prev_e = input.e;
  int init_cursor_pos = _get_cursor_pos(), init_arrow = _arrow;
  int line_start = init_cursor_pos - init_arrow -prev_e;
  ___clear_buf_with_range(exp.start_exp_idx,exp.exp_size);
  ___put_num_in_buf(exp.num_str,exp.num_size,exp.start_exp_idx);
  int shift_count = exp.exp_size - exp.num_size;
  int change_idx = exp.start_exp_idx + exp.exp_size ;
  ___shift_buf_many_times(shift_count,change_idx);
  _arrow = exp.start_exp_idx + exp.num_size - input.e;
  _update_cursor( line_start + exp.start_exp_idx +  exp.exp_size,0);// moving curser so clear cmd works
  ___clear_cmd(prev_e);
  _write_from_buffer();
  _update_cursor( line_start + exp.start_exp_idx +  exp.num_size,0);
}
```

در ابتدا در صورتی که اکسپرشن با موفقیت پیدا شده بود، پوینتر e و مقادیر اولیه _arrow و اشاره گر ذخیره شده، سپس مقدار ابتدای خط را حساب میکند تا بعد تر در جانمایی اشاره گر استفاده شود. در مرحله بعد بافر را از ابتدای شروع اکسپرشن به اندازه طول اکسپرشن خالی کرده و سپس عدد به دست آمده جواب را در ابتدای مکان خالی شده جایگذاری میکند. سپس از مکان انتهای اکسپرشن قبلی شروع کرده و فضای خالی بین انتهای عدد و مقدار خالی شده بافر را با کمک شیفت چپ پر میکند. سپس برای خالی کردن بافر، از آنجایی که در اثر

عملیات های انجام شده ممکن است مکان اشاره گر به هم ریخته باشد، ابتدا مکان اشاره گر روی ترمینال را به انتهای قبلی اکسپرشن انتقال داده، سپس تابع clear_cmd____ را صدا زده تا ترمینال را از مکان قبلی پوینتر e پاک کند (که هنوز روی صفحه دست نخورده اند). در مرحله بعد بافر آپدیت شده با تابع write_from_buffer_ بر روی ترمینال نوشته شده و نهایتا مکان اشاره گر به انتهای عدد جواب اکسپرشن منتقل میشود.

برنامه سطح کاربر:

ابتدا فایل encode.c و decode.c را تشکلی دادیم. چون این دو فایل تقریبا یکسانند، تنها encode.c را توضیح می‌دهیم.

```c
int
main(int argc, char *argv[])
{
  char output_file[]="result.txt";
  int fd;
  char text[SIZE];
  merge_argv(argc,text,argv);
  if((fd = open(output_file, O_CREATE)) < 0){
    printf(1, "encode: cannot create %s\n", output_file);
    exit();
  }
  close(fd);
  if((fd = open(output_file, O_WRONLY)) < 0){
    printf(1, "encode: cannot open %s\n", output_file);
    exit();
  }
  encode(fd,text);
  close(fd);
  exit();
}
```

ابتدا تابع merge_argv را صدا می‌زنیم. این تابع argv را می‌گیرد و آن را تبدیل به یک رشته می‌کند و برمی‌گرداند.

```c
void merge_argv(int count_strs,char merged_text[],char* argv[])
{
  int index=0;
  for (int i = 1; i < count_strs; i++)
  {
      for (int j = 0; argv[i][j]!='\0'; j++)
      {
        merged_text[index++]=argv[i][j];
      }
      if (i<count_strs-1)
      {
        merged_text[index++]=' ';
      }
  }
  merged_text[index++]='\n';
  merged_text[index]='\0';
  return;
}
```

سپس فایل result.txt را تشکیل می‌دهیم و این فایل را باز می‌کنیم و تابع encode را فرا می‌خوانیم.

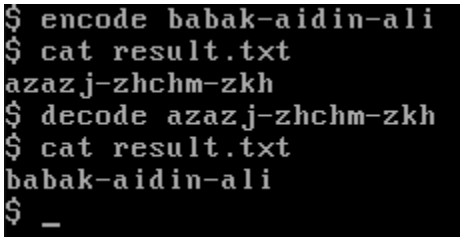
```
int SIZE=512;

void encode(int fd, char text[])
{
  int _ids[3]={810101408,810101561,810101560};
  int key=0;
  for (int i = 0; i < sizeof(_ids)/sizeof(int); i++)
  {
    key+=_ids[i]%100;
  }
  key=(key%26+26)%26;
  int i = 0;
  while(text[i]!='\0')
  {
    char base;
    if(text[i]>='a' && text[i]<='z')
    {
      base='a';
      text[i]=(text[i]+key-base)%26+base;
    }
    else if (text[i]>='A' && text[i]<='Z')
    {
      base='A';
      text[i]=(text[i]+key-base)%26+base;
    }
    i++;
  }
  if (write(fd, text, i) != i) {
    printf(1, "encode: write error\n");
    exit();
  }
}
```

این تابع ابتدا کلید را مطابق الگوریتم خواسته شده حساب می‌کند. سپس به ازای هر حرف رشته، با توجه به بزرگ یا کوچک بودن حرف، آن را مطابق الگوریتم سزار تغییر می‌دهد و در پایان رشته حاصل را در فایل result.txt می‌نویسد.

فایل decode نیز دقیقا همین کارها را انجام می‌دهد فقط key آن کمی متفاوت است.

```
void decode(int fd, char text[])
{
  int _ids[3]={810101408,810101561,810101560};
  int key=0;
  for (int i = 0; i < sizeof(_ids)/sizeof(int); i++)
  {
    key+=_ids[i]%100;
  }
  key=((-key)%26+26)%26;
```

و بدین ترتیب این دو برنامه سطح کاربر به درستی کار می‌کنند:



سوالات بخش مقدمه ای درباره سیستم عامل و xv6:

1- سه وظیفه اصلی سیستم عامل را نام ببرید؟

مدیریت منابع، قرار گرفتن میان کاربر و سخت افزار و مدیریت دسترسی ها و کاربران.

2- فایل های اصلی سیستم عامل xv6 در صفحه یک کتاب مربوطه 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.

1. Basic Headers:

این بخش شامل تعاریف و ماکروهای مرتبط با سیستم‌عامل مورد نظر است. ماکروها برای ایجاد بخش‌های مختلف مانند بخش‌های قابل اجرا، خواندنی و نوشتنی استفاده می‌شوند، همچنین ساختارهای مختلفی مانند `rtcdate`، `buf`، `proc`و `segdesc` برای مدیریت زمان، بافرها و پردازش‌ها در این بخش تعریف شده‌اند.

2. Entering xv6:

شامل فایل های اسمبلی و فایل main.c میباشد که وظیفه آماده سازی سیستم و ورود به کرنل بعد از عملیات بوت شدن را داراست.

3. Locks:

شامل فایل spinlock میباشد که موجب همگام سازی میان پروسس ها و اینتراپت هندلر ها میشود، به طوری که با مدیریت دسترسی ها از خراب شدن یک پروسس توسط دیگری جلوگیری میکند.

4. Processes:

این بخش نقش مهمی در زمینه مدیریت پروسس ها ایفا میکند، به نحوی که عملیات های مهمی نظیر ایجاد و خاتمه پروسس ها، کانتکست سوییچ و حافظه مجازی در این بخش پیاده سازی شده اند که برای مولتی تسکینگ ضروری محسوب میشوند.

5. System Calls:

اینترفیس های سیستم کال، نحوه پیاده سازی آنها به همراه ثابت های مهم و نحوه پیاده سازی ترپ در این بخش مشخص شده اند.

6. File System:

این گروه وظیفه مدیریت عملیات های فایل (خواندن، نوشتن و...)، فایل سیستم و مدیریت دیسک را بر عهده دارد.

7. Pipes:

این بخش برای ارتباط میان پروسس ها طراحی شده و مسئول تبادل داده میان پروسس ها است.

8. String Operations:

این گروه توابعی برای عملیات بر روی رشته ها در سطوح پایین (از جمله کپی و مقایسه) ارائه میدهند.

9. Low-level Hardware:

این گروه مسئول رسیدگی به سخت افزار از جمله دستگاه های ورودی خروجی، رسیدگی به اینتراپت های سخت افزاری و رسیدگی به ملزومات مولتی پروسسور ها میباشد.

10. User-level:

شامل برنامه های سطح کاربر (از جمله shell) بوده و مسئول تعامل برنامه های کاربر با کرنل میباشد.

11. Bootloader:

مسئول تعریف و پیاده سازی بوت لودر است که وظیفه لود کردن کرنل در مموری هنگام روشن شدن سیستم را داراست.

12. Link:

این بخش مسئول تعریف لینکر برای لینک کردن بخش های مختلف کرنل و جایگذاری هرکدام در فایل باینری نهایی کرنل میباشد.

لینوکس:

فایل های سرایند، کرنل و فایل سیستم ها همگی داخل دیرکتوری /usr/src/linux قرار دارند. این دیرکتوری شامل بخش های مهمی از جمله documentation حاوی فایل های راهنما، source code حاوی فایل های اجرایی مهم از جمله درایور ها، کرنل و فایل سیستم، makefile، config حاوی فایل های کانفیگ کرنل و ... میباشد.

سوالات بخش کامپایل سیستم عامل:

3- دستور make -n را اجرا کنید. کدام دستور فایل نهایی هسته را میسازد؟

دستور زیر مسئول ساخت فایل نهایی است:

```
ld -m    elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.
o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.o string.o swtch.o syscall.o sysfile.o sysproc.o trapasm
.o trap.o uart.o vectors.o vm.o  -b binary initcode entryother
```

این دستور کلید اصلی در فرآیند تولید فایل نهایی کرنل است. این دستور لینکر میباشد که با استفاده از فایل kernel.ld که قوانین لینک کردن را تعریف می‌کند، تمام آبجکت فایل های تولید شده از مراحل قبلی را به فایل اجرایی کرنل سیستم‌عامل لینک میکند.

در makefile متغیر هایی به نام های UPROGS و ULIB تعریف شده است. کاربرد آنها چیست؟

UPROGS:

لیستی از برنامه های کاربر است که کامپایل و در تصویر نهایی فایل سیستم ها (fs.img) لینک میشوند. این برنامه ها در واقع همان برنامه هایی هستند که کاربر میتواند اجرا کند (مانند cat، ls، echo) و همانطور که توضیح داده شد، پس از تولید فایل سیستم این برنامه ها داخل تصویر فایل سیستم قرار گرفته و قابلیت اجرا شدن پیدا خواهند کرد.

ULIB

لیستی از کتابخانه های سطح کاربر است که توابع مورد نیاز برنامه های سطح کاربر (مخصوصا توابع نوشته شده در c)را داخل خود دارند. فایل هایی از جمله ulib، usys، printf و umalloc خدماتی از جمله سیستم کال

ها، الوکیشن حافظه و ورودی خروجی سطح کاربر را ارائه میدهند که موجب میشود از نوشتن دوباره این توابع جلوگیری و نوشتن برنامه سطح کاربر بسیار آسان تر شود.

سوالات اجرا بر روی شبیه ساز QEMU:

5- دستور make qemu -n را اجرا نمایید. دو دیسک به عنوان ورودی به شبیه ساز داده شده است. محتوای آن ها چیست؟

```
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=di
sk,format=raw -smp 2 -m 512
```

دیسک های ورودی عبارتند از: xv6.img و fs.img. تصویر xv6.img شامل کد های اصلی کرنل سیستم عامل و بوت لودر برای بارگذاری سیستم عامل داخل حافظه میباشد. تصویر fs.img شامل فایل سیستم های مهم افزون بر ULIB و UPROGS است که عملیات های برنامه های سطح کاربر وابسته به آنها میباشد که در قسمت قبل توضیح داده شد.

سوالات مراحل بوت سیستم عامل:

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

این دستور برای تبدیل فایل های کامپایل شده به فرمت مناسب برای boot loader استفاده میشود. از آنجایی که boot loader انتظار مشاهده این فایل به صورت خام باینری دارد، این دستور تبدیل لازم را انجام میدهد.

13- کد bootmain.c هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در آدرس 0x100000 قرار می‌دهد .علت انتخاب این آدرس چیست؟

علت اینکه آنرا در آدرس های بالاتر قرار نمیدهیم این است که اگر ماشین کوچک بود و در حافظه محدودیت داشت نیز بتواند به آن دسترسی داشته باشد؛ همچنین آنرا در آدرس پایینتر نیز نمیتوان قرار داد چرا که آدرس های 0xa0000:0x100000 در اختیار دستگاه های ورودی و خروجی هستند.

سوالات اجرای هسته xv6:

18- همانطور که ذکر شد، ترجمه قطعه تأثیری بر ترجمه آدرس منطقی نمیگذارد. زیرا تمامی قطعه ها اعم از کد و داده روی یکدیگر میافتند. با این حال برای کد و داده های سطح کاربر پرچم USER_SEG تنظیم شده است. چرا؟

درست است که ترجمه قطعه تأثیری بر ترجمه آدرس ها نمیگذارد؛ اما همچنان پردازنده برای محافظت از آدرس های در انحصار هسته باید از سطح دسترسی دستورات مطلع باشد، به همین دلیل از این پرچم برای تعیین سطح دسترسی استفاده میشود.

سوالات اجرای نخستین برنامه سطح کاربر:

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

ما در این ساختار نام فرآیند، kernel stack، page table، وضعیت فرآیند ، id فرآیند، اندازه حافظه گرفته شده، فرآیند پدر، پرچمی برای تعیین خاتمه فرآیند، اشاره گری که اگر فرآیند معلق بود به آن اشاره کند، فایل های باز، پوشه فعلی، trap frame برای system call های فعلی و context برای تعویض را برای مدیریت برنامه ها داریم.

در linux، ساختار task_struct را برای این منظور استفاده میکنیم.

23- 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؟

موارد مشترک بین هسته های پردازنده:

مدیریت حافظه، ساختار مدیریت پردازش، زمان بندی، ارتباطات بین پردازنده ها

در این سیستم زمان بند روی تمامی پردازنده ها مستقل اجرا میشود که منجر به موازی سازی و همزمانی پردازش میشود.

موارد اختصاصی:

Interrupt های محلی، context و structure های اختصاصی

هر پردازنده به interrupt های سخت افزار خودش جداگانه رسیدگی میکند.

سوالات بخش اشکال زدایی:

1. برای مشاهده Breakpointها از چه دستوری استفاده میشود؟

info breakpoint

2. برای حذف یک Breakpoint از چه دستوری و چگونه استفاده میشود؟

برای حذف breakpoint با شماره i تا شماره j می‌توانیم از دستور زیر استفاده کنیم:

del i-j

برای حذف breakpoint از خط i فایل f از دستور زیر استفاده می‌کنیم:

clear f:i

3. دستور زیر را اجرا کنید. خروجی آن چه چیزی را نشان میدهد؟

$ bt

برای بررسی دستور، یک breakpoint در تابع _arrow_key_console_handler فایل console.c گذاشتیم و وقتی به این breakpoint رسیدیم، دستور فوق را اجرا کردیم.

```
(gdb) break console.c:230
Breakpoint 4 at 0x80100b46: file console.c, line 230.
(gdb) continue
Continuing.
[Switching to Thread 1.1]

Thread 1 hit Breakpoint 4, _arrow_key_console_handler (c=226) at console.c:230
230         if (c == _UP_ARROW)
(gdb) bt
#0  _arrow_key_console_handler (c=226) at console.c:230
#1  0x8010162d in consoleintr (getc=0x80103490 <kbdgetc>) at console.c:514
#2  0x80103580 in kbdintr () at kbd.c:49
#3  0x80106885 in trap (tf=0x80116ad8 <stack+3912>) at trap.c:67
#4  0x801065df in alltraps () at trapasm.S:20
#5  0x80116ad8 in stack ()
#6  0x80112e64 in cpus ()
#7  0x80112e60 in ?? ()
#8  0x80103ddf in mpmain () at main.c:57
#9  0x80103f2c in main () at main.c:37
```

همان‌طور که می‌توان دید، این دستور، call stack برنامه را نشان می‌دهد. یعنی فراخوانی‌هایی که انجام شده به مکان فعلی برسیم.

4. 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: help x و help print)

هدف این دو دستور متفاوت است. دستور x به ما اجازه می‌دهد که محتوای آدرس خاصی از حافظه را با فرمتی خاص بررسی کنیم. دستور print برای بررسی مقدار متغیرها و عبارات استفاده می‌شود.

از لحاظ کاربرد نیز این دو دستور متفاوتند. از دستور x برای دیباگ بررسی سطح پایین خانه‌های حافظه استفاده می‌شود ولی از print برای دیباگ سطح بالا و مشاهده مقدار متغیرها استفاده می‌شود.

همچنین برای مشاهده محتوای یک ثبات خاص از دستور print با نام آن ثبات می‌توان استفاده کرد:

```
(gdb) print $eax
$1 = 82
```

```
(gdb) print _current_history
$3 = 2
```

5. برای نمایش وضعیت ثبات‌ها از چه دستوری استفاده می‌شود؟ متغیرهای محلی چطور؟ نتیجه این دستور را در گزارش‌کار خود بیاورید. همچنین در گزارش خود توضیح دهید که در معماری x 86 رجیسترهای edi و esi نشانگر چه چیزی هستند؟

می‌توان از دستور info registers برای مشاهده وضعیت تمام ثبات‌ها استفاده کرد:

```
(gdb) info registers
eax            0x52               82
ecx            0x552              1362
edx            0x3d5              981
ebx            0x3d5              981
esp            0x80116a24         0x80116a24 <stack+3732>
ebp            0x80116a4c         0x80116a4c <stack+3772>
esi            0x3d4              980
edi            0xe                14
eip            0x80100b46         0x80100b46 <_arrow_key_console_handler+86>
eflags         0x93               [ IOPL=0 SF AF CF ]
cs             0x8                8
ss             0x10               16
ds             0x10               16
es             0x10               16
fs             0x0                0
gs             0x0                0
fs_base        0x0                0
gs_base        0x0                0
k_gs_base      0x0                0
cr0            0x80010011         [ PG WP ET PE ]
cr2            0x0                0
cr3            0x3ff000           [ PDBR=1023 PCID=0 ]
```

و از دستور info locals برای مشاهده متغیرهای محلی استفاده می‌شود:

```
(gdb) info locals
_pos = 1362
```

در معماری x86 این دو رجیستر به عنوان index registers شناخته می‌شوند. معمولا از این دو در انتقال داده و ایجاد تغییرات در رشته‌ها استفاده می‌شود.

معمولا از ثبات esi به عنوان پوینتر مبدا استفاده می‌شود. مثلا در کپی محتوا به خانه‌ای که از آن می‌خوانیم اشاره می‌کند.

edi به عنوان پوینتر مقصد استفاده می‌شود و به جایی که در آن می‌نویسیم اشاره می‌کند.

6. به کمک استفاده ازGDB ، 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 مثال، e.input در چه حالتی تغییر میکند و چه مقداری میگیرد)

با gdb، watchpoint روی input قرار دادیم هر وقت تغییر کرد مقادیر آن را با کمک print بررسی کنیم. در ابتدا buf مقدار نال دارد و بقیه متغیرهای input هم صفر هستند. با هر حرفی که کاربر وارد می‌کند، آن حرف جدید به buf اضافه می‌شود و همچنینی یکی به مقدار e اضافه می‌شود. با هر backspace که کاربر می‌زند، حرف آخر از buf پاک می‌شود و یکی از e کم می‌شود و در تمام این مدت w و r صفر هستند. در نهایت وقتی کاربر enter بزند، نیولاین هم به buf اضافه می‌شود و e هم یکی اضافه می‌شود. سپس w برابر e می‌شود و بعد r هم یکی یکی زیاد می‌شود تا برابر e و w شود و در نهایت هر سه دوباره مثل ابتدا صفر می‌شوند و buf هم خالی می‌شود. با بررسی انجام شده این نتایج احتمالی را می‌شود گرفت:

متغیر buf آرایه‌ای است که حروف تایپ شده توسط کاربر در آن ذخیره می‌شوند. با هر تغییری که کاربر ایجاد می‌کند، محتوای buf تغییر می‌کند.

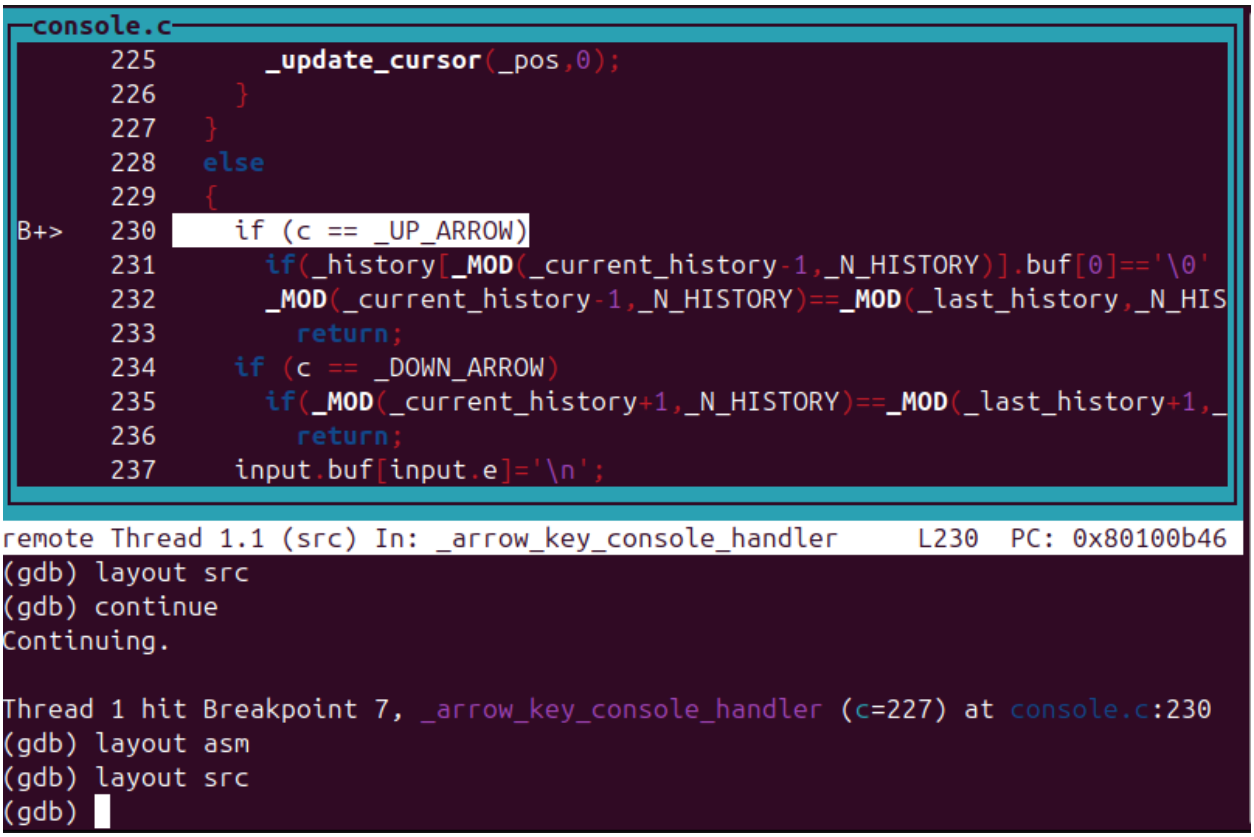
متغیر r اشاره به خانه بعدی از buf می‌کند که از آن می‌خواند. تنها موقع خواندن یعنی وقتی کاربر enter بزند افزایش میابد و بقیه اوقات صفر است.

متغیر w اشاره به اولین خانه از buf می‌کند. این متغیر هم همواره صفر است و فقط وقتی کاربر enter بزند، مقدارش برابر e می‌شود.

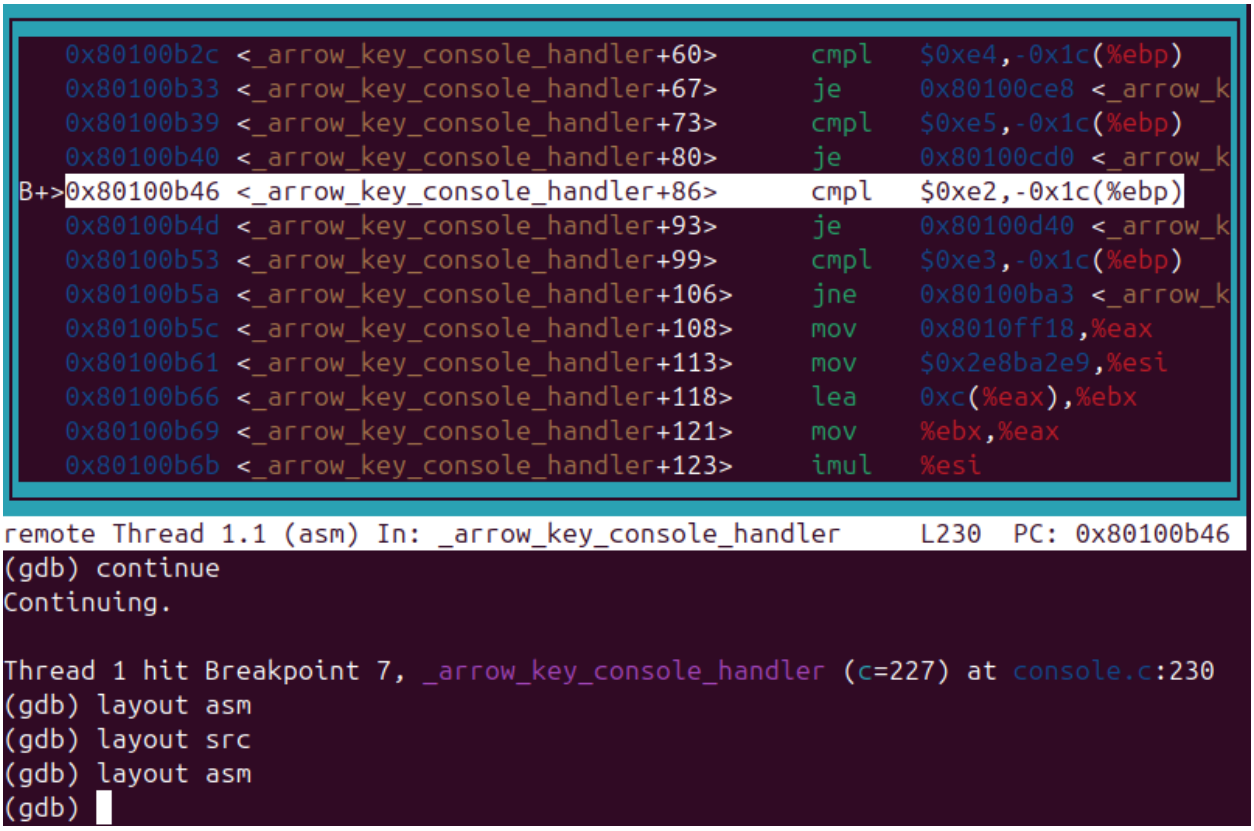
متغیر e اشاره به خانه‌ای از buf می‌کند که کاربر در حال ایجاد تغییراتی در آن است و با هر تغییر کاربر کم یا زیاد می‌شود و تغییر می‌کند.

7. خروجی دستورهای layout src و layout asm در TUI چیست؟

با استفاده از دستور layout src محیط terminal تغییر می‌کند به طوری‌که نیمه بالای ترمینال سورس کد برنامه و مکان فعلی را نشان دهد و نیمه پایین، مانند قبل بتوان دستورات gdb را وارد کرد:



دستور layout asm هم مانند دستور layout src است با این تفاوت که به جای سورس کد، کد اسمبلی نمایش داده می‌شود:

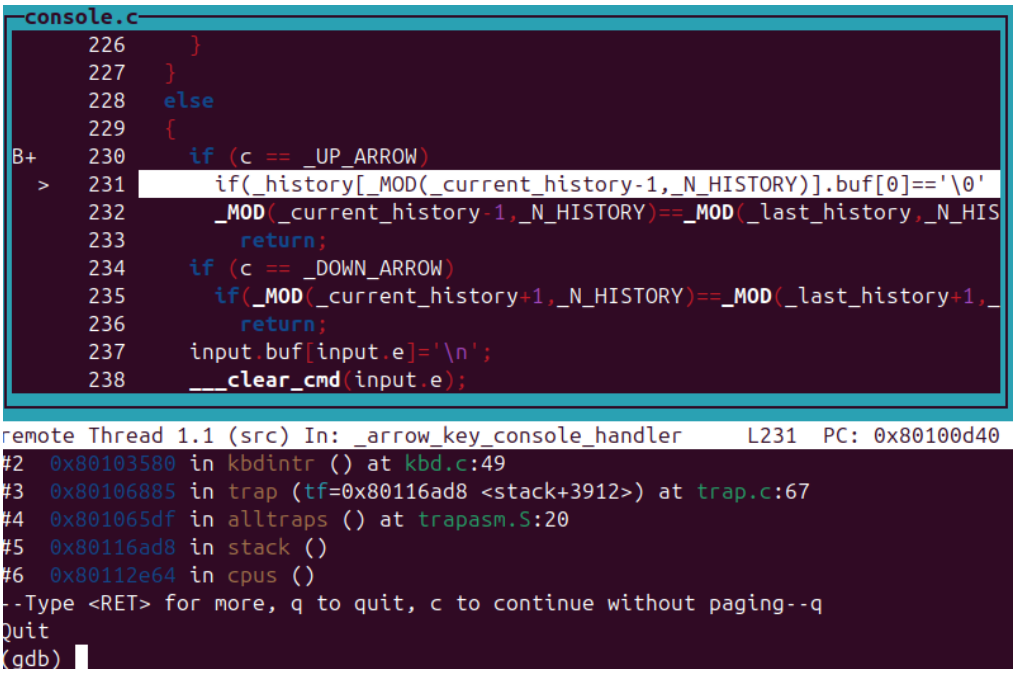


مشخص است که نمایش دستورات در این حالات به ردگیری برنامه کمک می‌کند چون می‌توانیم هم زمان کد را هم مشاهده کنیم.

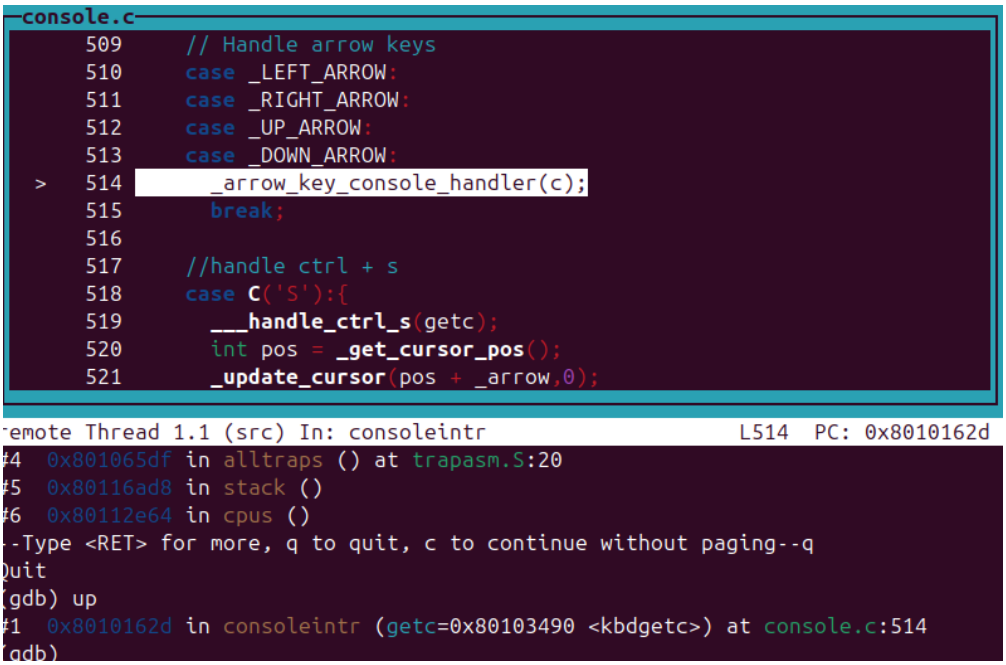
8. برای جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستوراتی استفاده میشود؟

در سوال 3 با دستور bt آشنا شدیم که زنجیره فراخوانی را به ما نمایش می‌داد. برای رفتن از تابع کنونی به تابع صدازننده از دستور up و برای برگشتن به تابع صدا زده شده از دستور down می‌شود استفاده کرد.

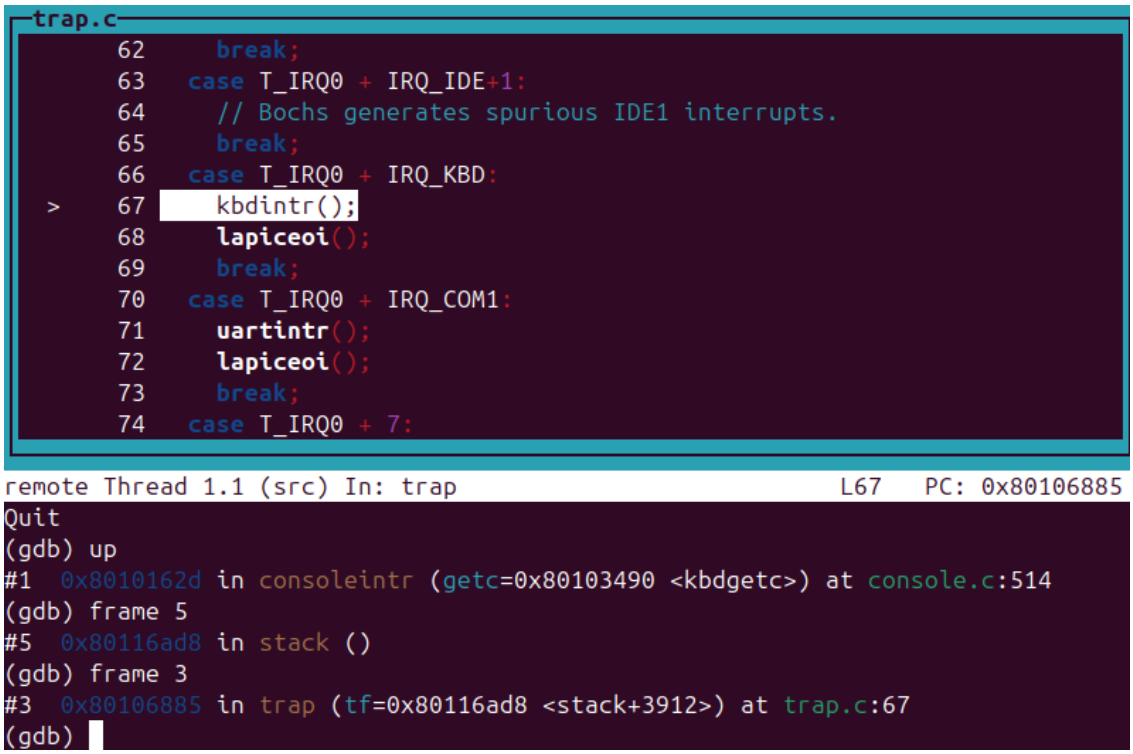
مکان فعلی:



برگشتن به تابع صدا زننده:



همچنین می‌توان با frame به تابع ام در زنجیره فراخوانی برویم:


trap.c
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:
remote Thread 1.1 (src) In: trap L67 PC: 0x80106885
Quit
(gdb) up
#1 0x8010162d in consoleintr (getc=0x80103490 <kbdgetc>) at console.c:514
(gdb) frame 5
#5 0x80116ad8 in stack ()
(gdb) frame 3
#3 0x80106885 in trap (tf=0x80116ad8 <stack+3912>) at trap.c:67
(gdb)