به نام خدا

گزارشکار آزمایشگاه یک آزمایشگاه سیستم عامل

اعضای گروه:

علی زمانی 810101436

برنا فروهری 810101480

نیما خنجری 810101416

آشنایی با سیستم عامل xv6

1. معماری سیستم عامل xv6 چگونه است.

xv6 یک پیاده سازی جدید از نسخه ششم یونیکس برای سیستمهای چند پردازنده x86 و RISC-V است. معماری این سیستم عامل ساختاری مشابه معماری کلی یونیکس دارد. xv6 دارای نوعی خاص از معماری است که به آن اجازه می دهد همه اجزای هسته در یک فضا عمل کنند. با استفاده از دستورات رایجی مانند exec , fork میتوان فرآیند ها را مدیریت کرد و دسته بندی فایلهای این سیستم عامل نیز مثل یونیکس System calls و user level program و مانند آن است.

این سیستم عامل در واقع شبه یونیکس (Unix (Like) و مشابه Unix v6 نوشته شده است. این سیستم عامل با ASIC و برای x86 multiprocessor و سیستمهای RISC-V طراحی شده است . برخی از اجزای اصلی معماری این سیستم عامل به شرح زیر هستند:

1. هسته(Kernel)
2. مدیریت فرآیند.
3. مدیریت حافظه
4. سیستم فراخوانی‌ها(System Calls)
5. سیستم فایل(File System)
6. درایورهای دستگاه
7. زمان‌بند
8. مدیریت وقفه‌ها(Interrupt Handling)
9. برنامه‌های کاربری(User Programs)
10. کنسول

2.یک پردازنده در xv6 از چه بخش هایی تشکیل شده؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازه های مختلف اختصاص می دهد؟

در سیستم‌عامل xv6، زمان بند(scheduler) هایی وجود دارند که به کمک آن ها میتوان به صورت بهینه پردازنده ها را به process هایی که در جریان اند اختصاص داد.(در زیر این فرآیند توضیح داده شده) . بخش های اصلی یک پردازنده xv6

شامل وضعیت و نام پردازنده ها و فضای حافظه یوزر با داده ها و دستورات میباشد که در اختیار هسته قرار میگیرد .همچنین پردازنده از جدول صفحه‌ها برای نگاشت حافظه مجازی به حافظه فیزیکی استفاده می‌کند.

سیستم‌عامل xv6از یک زمان‌بند ساده استفاده می‌کند تا پردازنده را بین پردازه‌های مختلف به صورت عادلانه اختصاص دهد. در این سیستم‌عامل، از الگوریتم زمان‌بندی خاصی استفاده می‌شود. نحوه اختصاص پردازنده به پردازه‌ها به طور کلی به این شکل است:

1. حالت زمان‌بندی:

هر پردازه یک زمان مشخص از پردازنده دریافت می‌کند. بعد از اینکه زمان پردازه به پایان رسید، پردازنده به پردازه‌ی بعدی اختصاص داده می‌شود.

2. لیست آماده:

پردازه‌هایی که آماده اجرا هستند در یک صف آماده (ready queue)نگهداری می‌شوند. زمان‌بند به ترتیب از ابتدای این صف پردازه‌ها را انتخاب می‌کند و پردازنده را به آن‌ها اختصاص می‌دهد.

3. وقفه‌های تایمر:

یک تایمر سخت‌افزاری پس از هر دوره زمانی مشخص، وقفه ایجاد می‌کند تا زمان‌بند سیستم فعال شود و فرآیند جاری را متوقف کند. در این لحظه، زمان‌بند بررسی می‌کند که آیا باید به پردازه‌ی جدیدی پردازنده اختصاص دهد یا پردازه فعلی همچنان ادامه یابد.

3. مفهوم file descriptor در سیستم عامل های مبتنی بر UNIX چیست؟عملکرد pipe در xv6 چگونه است؟

در سیستم‌عامل‌های مبتنی بر یونیکس (مانند xv6)، مفهوم File Descriptor (FD)یا توصیف‌گر فایل، یک شناسه‌ی عددی است که توسط سیستم‌عامل به هر فایل یا منبع باز شده (مثل فایل، سوکت، یا دستگاه‌های ورودی/خروجی) اختصاص داده می‌شود. این شناسه به فرآیندها اجازه می‌دهد تا بدون نیاز به دانستن جزئیات فیزیکی فایل یا منبع، به آن دسترسی داشته باشند.

درxv6 ، pipeبه عنوان یک فایل خاص در نظر گرفته می‌شود و همان‌گونه که فرآیندها با فایل‌ها تعامل دارند، می‌توانند از طریق توابع readو writeداده‌ها را به pipe ارسال یا دریافت کنند. داده‌ها در بافر pipe ذخیره می‌شوند تا زمانی که فرآیند خواننده آماده باشد آنها را دریافت کند. اگر بافر پر شود، فرآیند نویسنده باید منتظر بماند تا فضا خالی شود و اگر بافر خالی باشد، فرآیند خواننده منتظر می‌ماند تا داده‌ها دریافت شوند در سیستم‌عامل‌های مبتنی بر یونیکس مانند xv6، Pipeمکانیزمی است که برای ارتباط بین فرآیندها (Inter-Process Communication- IPC) استفاده می‌شود. این مکانیزم به فرآیندها اجازه می‌دهد که داده‌ها را از یک فرآیند به فرآیند دیگر به‌طور مستقیم از طریق یک "لوله (pipe) "ارسال کنند..

عملکرد Pipe :

یک فرآیند می‌تواند با استفاده از سیستم‌فراخوانی pipe یک "لوله" ایجاد کند. این فراخوانی دو File Descriptor بازگشتی به فرآیند می‌دهد: یکی برای خواندن از لوله و یکی برای نوشتن به لوله

Pipe به عنوان یک بافر در حافظه عمل می‌کند که در آن داده‌ها به ترتیب نوشته و خوانده می‌شوند.

4. فراخوانی های سیستمی exec و fork چه عملی انجام میدهند؟

از نظر طراحی ادغام نکردن این دو چه مزیتی دارد؟

فراخوانی سیستمی fork می‌گویند، کپی دقیقی از (child)یک فرآیند جدید ایجاد می‌کند. این فرآیند جدید، که به آن فرزند است (parent)فرآیند والد.

fork، هر دو فرآیند (والد و فرزند) از نقطه‌ای که fork فراخوانی شده ادامه می‌دهند.

fork معمولاً از برای ایجاد یک فرآیند جدید استفاده می‌شود تا از آن در کارهای همزمان و موازی بهره‌برداری شود.

fork، معمولاً نیاز است که این فرآیند جدید وظیفه‌ای متفاوت را اجرا کند. فراخوانی سیستمی exec به همین منظور طراحی شده است. این فراخوانی یک فرآیند را با یک برنامه جدید جایگزین می‌کند.

fork به طور معمول پس از برای تغییر وظیفه فرزند به یک برنامه جدید استفاده می‌شود. به عنوان مثال، پس از ایجاد یک exec فرآیند فرزند، والد می‌تواند با استفاده از به فرزند دستور دهد که یک برنامه خاص را اجرا کند.

دلایل ادغام نکردن این دوسیستم کال:

fork و exec می‌توان فرآیندهای فرزند را ایجاد کرد و سپس در زمان‌های مختلف برنامه‌های مختلفی را اجرا کرد. این امر انعطاف‌پذیری بیشتری را فراهم می‌کند. 1. با جدا نگه‌داشتن

exec والد می‌تواند شامل متغیرهای محیطی یا اطلاعات یا داده‌ها را به فرزند منتقل کند. این اطلاعات می‌تواند قبل از اجرای پارامترهای ورودی باشد که ممکن است نیاز باشد قبل از اجرای برنامه جدید به فرزند ارسال شود.

fork و exec طراحی سیستم‌عامل‌ها پایبند است که بر اساس جداسازی وظایف و طراحی جداگانه برای مسئولیت‌ها عمل می‌کند. این به بهبود کارایی و خوانایی کد کمک می‌کند.

شرح پروژه:

1. برای افزودن قابلیت چپ و راست رفتن در متن متغیر Backs را تعریف میکنیم که نشان دهنده دفعات فشرده شدن کلید → و ←می باشد. در واقع با فشرده شدن → یک واحد از این متغیر کم کرده و با فشردن ← یک واحد به آن اضافه می کنیم.

```
case LEFT_ARROW: //(jadid)left arrow ascii code
  if ((input.e - backs) > input.w) //ensure cursor posi
  {
    handle_cursor(BACK);
    backs++;
  }
  break;

case RIGHT_ARROW: // (jadid)right arrow ascii code
  if (backs > 0) // ensure back value stays positive
  {
    handle_cursor(FORWARD);
    backs--;
  }
  break;
```

```
static void handle_cursor(enum Arrow action)
{
  int position;

  // get the current position of cursuer using ports and registers
  outb(CRTPORT, 14);
  position = inb(CRTPORT + 1) << 8;
  outb(CRTPORT, 15);
  position |= inb(CRTPORT + 1);

  switch (action)
  {
  case BACK:
    position--;
    break;
  case FORWARD:
    position++;
    break;
  default:
    break;
  }
```

در واقع مقدار (input.e – back) نشان دهنده نقطه فعلی cursor می باشد. در ادامه ساختار input را توضیح می دهیم:

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

همانطور که مشاهده می شود ساختار Input برای ذخیره سازی ورودی در نظر گرفته شده. متغیر buf ورودی در کنسول را ذخیره سازی میکند.متغیر e پوینتر به آخرین حرف به عنوان ورودی می باشد.همچنین w پوینتر به ابتدای آخرین سطر ورودی است.(در سوال GDB 6 به طور مفصل این ساختار را تغریف کرده ایم.)

2.برای ذخیره سازی 10 دستور آخر از ساختار زیر استفاده میکنیم:

```
struct //jadid
{
  struct Input instructions[10];
  int index;
  int last;
  int count;
}history;
```

در این ساختار 10 متغیر input در instructions ذخیره شده همچنین index نشان دهنده دستور فعلی count نشان دهنده کل دستور ها و last نشان دهنده آخرین دستور می باشد. این متغیر ها در پیمایش میان 10 دستور آخر به ما کمک میکنند.

```
if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF)
{
  if(check_if_history(buf_value))
  {
    //WRITE LAST TEN
    release(&cons.lock);
    for(int i=0 ; i<history.count+1 ; i++)
      cprintf(&(history.instructions[i].buf[history.instructions[i].w]));
    acquire(&cons.lock);

  }
  else
  {
    if (history.count < 9){
    history.instructions[history.last + 1] = input;
    history.last ++ ;
    history.index = history.last;
    history.count ++ ;
    }
    else{
      for (int i = 0; i < 9; i++) {
        history.instructions[i] = history.instructions[i+1];
      }
      history.instructions[9] = input;
      history.index = 9;
      history.last = 9;
      history.count = 10;
    }
  }
```

هنگامی که کلید enter فشرده شود به داخل بلاک کد نشان داده شده در بالا میرویم.در صورتی که واژه تایپ شده history باشد 10 دستور آخر که در متغیر historyذخیره شده بود را نمایش میدهیم. در غیر این صورت به ذخیره سازی این دستور می پردازیم که این امر به دو حالت تقسیم می شود:

اگر تعداد دستور های وارد شده تا به کنون کمتر از 10 باشد آن را در انتهای آرایه instructions ذخیره میکنیم. در غیر این صورت اولین instruction در آرایه instructions را حذف کرده و instruction جدید را به آرایه اضافه می کنیم.توجه شود که در حالت دوم نیاز است تمام instruction های قبلی یک ایندکس شیفت بخورند.

```
static int check_if_history(char* word)
{
  int i=0;
  int flag = 0;
  char *h = "history";
  while (word[i]!='\0' && word[i]!='\n')
  {
    if(word[i]!=h[i])
    {
      flag = 1;
      break;
    }
    i++;
  }
  if(i!=7)
    flag = 1;
  return !flag;
}
```

حال برای بالا و پایین رفتن در دستور ها به نحو زیر عمل میکنیم:

```
static void handle_up_down_arrow(enum Arrow arrow){ //jadid
  remove_cur_line();
  if ((arrow == DOWN)&&(history.index < 9)&&(history.index + 1 < history.last )){
    handle_down_arrow();
  }
  else {
    handle_up_arrow();
  }

  for (int i = input.w ; i < input.e; i++)
  {
    consputc(input.buf[i]);
  }
}
```

ابتدا لاین فعلی را پاک کرده و سپس با توجه به کلید فشرده شده با استفاده از تابع های مخصوص به event مربوطه رسیدگی می کنیم:

```
static void handle_down_arrow()
{
    history.index ++ ;
    input = history.instructions[history.index + 1 ];
    input.buf[--input.e] = '\0'; //remove \n char from last buffered so we can continue typing

}

static void handle_up_arrow()
{
    input = history.instructions[history.index--];
    input.buf[--input.e] = '\0';
}
```

```
static void remove_cur_line()
{
  for (int i= 0 ; i < backs ; i ++){ //move cursor into later of current line
    handle_cursor(FORWARD);
  }
  backs = 0;
  for ( int i = input.e ; i > input.w ; i-- ){ //remove all letters of current line
    if (input.buf[i - 1] != '\n'){
      consputc(BACKSPACE);
    }
  }
}
```

3. پس از فشرده شدن ctrl S سیو شدن در بافر مخصوص (saveInp) آغاز می شود. همچنین ما جایگاه cursor را در (saveInp.start) ذخیره می کنیم. همچنین بافر مربوطه را در این لحظه خالی میکنیم.پس از فشرده شدن ctrl F چک می کنیم آیا saveInp فعال هست یا خیر. در متغیر saveInp.end جایگاه فعلی cursor را ذخیره میکنیم. سپس تعداد حروفی که تا این لحظه در بافر ذخیره شده است را محاسبه می کنیم.حال تمام حروف در input.buf که در جایگاه بعد از saveInp.end را به اندازه count شیفت میدهیم. سپس حروف ذخیره شده در بافر saveInp را در جایگاه مناسب آرایه Inp.buf کپی می کنیم.

```
struct SaveInput // jadid
{
  char copybuf[128];
  int start;
  int end;
  int active;
} saveInp;
```

```
case C('S'):
  saveInp.start = input.e - backs;
  saveInp.active = 1;
  for (int i = 0; i < 128; i++)
    saveInp.copybuf[i] = '\0';
  break;

case C('F'):
  if (saveInp.active == 1)
  {
    saveInp.end = input.e - backs;
    int count = 0;
    for (int i = saveInp.start; i < 128; i++) // saveInp.end-backs
    {
      if (saveInp.copybuf[i] != '\0')
        count++;
    }
    for (int j = input.e; j > saveInp.end; j--) // jadid
      input.buf[j + count] = input.buf[j];

    int j = 0;
    for (int i = saveInp.start; i < INPUT_BUF; i++) // saveInp.end-backs
    {
      if (saveInp.copybuf[i] != '\0')
      {
        input.buf[saveInp.end + j] = saveInp.copybuf[i];
        input.e++;
        consputc(saveInp.copybuf[i]);
        j++;
      }
    }
  }
```

4. برای محاسبه ?=NON بلافاصله پس از فضرده شدن کلید ? داخل تابع زیر می رویم:

```
static void check_previous_letters() //jadid
{
  if(input.buf[input.e-1] == 61)
  {
    int first_num_index=0;
    int second_num_index=0;
    if(input.buf[input.e-2]<48 || input.buf[input.e-2]>57)
      return;
    int first_num=read_num(input.e-2, &first_num_index);
    char result[128];
    int second_num = read_num(first_num_index-2, &second_num_index);

    delete_letters(second_num_index);

    consputc(BACKSPACE);

    switch (input.buf[first_num_index-1])
    {
    case 42:
      int_to_str(first_num * second_num, result);
      break;

    case 43:
      int_to_str(first_num + second_num, result);
      break;

    case 45:
      int_to_str(second_num - first_num, result);
      break;

    case 47:
      float_to_str((float)second_num / (float)first_num, result);
      break;

    default:
      break;
    }
    release(&cons.lock);
    cprintf(result);
    update_input(second_num_index, result);
    acquire(&cons.lock);
  }
}
```

ابتدا چک می کنیم حرف ماقبل علامت سوال = باشد سپس با استفاده از تابع read_num عدد های اول و دوم را میخوانیم.سپس تمام ورودی ?=NON را پاک نموده و با توجه به اپراتور مربوطه عملیات محاسباتی مربوطه را انجام می دهیم.سپس حاصل مربوطه را در کنسول نمایش داده و در input.buf ذخیره می کنیم.

```
static int read_num(int start_index,int *end_index)
{
  int num = 0;
  int count = 0;
  for(int i=start_index ; i>((int)input.w)-1 ; i--)
  {
    if(input.buf[i]>=48 && input.buf[i]<=57)
    {
      num += (power(10,count)*((int)input.buf[i]- ASCII_0));
      count++;
    }
    else if ((input.buf[i]==45) && ((input.buf[i-1]==42) || (input.buf[i-1]==43) ||(input.buf[i-1]==45) ||(input.buf[i-1]==47)))
    {
      num = -num;
      *end_index = i;
      break;
    }
    else if ((input.buf[i]==45) && (i==(int)input.r))
    {
      num = -num;
      *end_index = i;
      break;
    }
    else
    {
      *end_index = i+1;
      break;
    }
  }
  return num;
}
```

در تابع read_num عدد را از سمت راست میخوانیم به طوری که آرگومان start_index نشان دهنده آخرین حرف است.آرگومان end_index نشان دهنده اولین حرف است و به علت نیازی که در ادامه به آن خواهیم داشت pass by reference شده است.

```
void delete_letters(int end_index)
{
  for(int i=end_index; i<input.e ; i++)
  {
    consputc(BACKSPACE);
  }
}
```

```
void static update_input(int update_index, char new_string[])
{
  int i = update_index;
  int j =0;

  while (new_string[j]!='\0')
  {
    input.buf[i] = new_string[j];
    i++;
    j++;
  }
  while (input.buf[i]!='\0')
  {
    input.buf[i] = '\0';
    i++;
  }
  input.e = (uint)(update_index+j);

}
```

در این تابع حاصل محاسبه ?=NON در new_string به تابع پاس داده شده.update_index نیز نشان دهنده جایگاهی است که این رشته اضافه خواهد شد.

برنامه سطح کاربر:

این با توجه به کلید گروه که برابر 2 میباشد باید تمامی حروف انگلیسی پس از عبارت encode را با عدد 2 جمع کنیم و کرکتر جدید را در فایل result.txt ذخیره کنیم.دقت شود که حرف y ← a و z ← b مپ میشوند.برای decode هم مانند بالا تنها نیاز است حروف را از 2 کم کنیم و معکوس مپ بالا را انجام دهیم.برای پیاده سازی دو فایل encode.c و decode.c در فولدرxv6 قرار می دهیم و در makefile در بخش UPROGS این دو دستور encode و decode را تعریف می کنیم.

Encoder:

```c
int main(int argc, char *argv[])
{
    int i=0;
    int flag = 0;
    char *h = "encode";
    while (argv[0][i]!='\0' && argv[0][i]!='\n')
    {
        if(argv[0][i]!=h[i])
        {
            flag = 1;
            break;
        }
        i++;
    }
    if(i!=6)
        flag = 1;
    unlink("result.txt");
    int fd = open("result.txt", O_CREATE | O_RDWR);

    if (flag == 0) //key 2
    {
        for (int i = 1; i < argc; i++)
        {
            int j = 0;
            char arr[sizeof(argv[i])];
            int last = 0;
            while (argv[i][j] != '\0')
            {
                if ((argv[i][j] > 64)&&(argv[i][j] < 89) || (argv[i][j] > 96)&&(argv[i][j] < 121))
                {
                    arr[j] = argv[i][j] + 2;
                }
                else if (argv[i][j] == 89)
                {
                    arr[j] = 'A';
                }
                else if (argv[i][j] == 90)
```

```
            while (argv[i][j] != '\0')
                else if (argv[i][j] == 90)
                {
                    arr[j] = 'B';
                }
                else if (argv[i][j] == 121)
                {
                    arr[j] = 'a';
                }
                else if (argv[i][j] == 122)
                {
                    arr[j] = 'b';
                }
                else
                {
                    arr[j] = argv[i][j];
                }
                j++;
                last = j;
            }

            arr[last+1] = '\0';
            write(fd, arr, last);
            char space[2];
            space[0] = ' ';
            space[1] = '\0';
            write(fd, space, 1);
        }
    }
    char arr[2];
    arr[0] = '\n';
    arr[1] = '\0';
    write(fd, arr, 1);
    close(fd);
    exit();
}
```

:Decoder

```c
int main(int argc, char *argv[])
{
    int i=0;
    int flag = 0;
    char *h = "decode";
    while (argv[0][i]!='\0' && argv[0][i]!='\n')
    {
        if(argv[0][i]!=h[i])
        {
            flag = 1;
            break;
        }
        i++;
    }
    if(i!=6)
        flag = 1;
    unlink("result.txt");
    int fd = open("result.txt", O_CREATE | O_RDWR);
    if (flag == 0) //key 2
    {
        for (int i = 1; i < argc; i++)
        {
            int j = 0;
            char arr[sizeof(argv[i])];
            int last = 0;
            while (argv[i][j] != '\0')
            {
                if ((argv[i][j] > 66)&&(argv[i][j] < 91) || (argv[i][j] > 98)&&(argv[i][j] < 123))
                {
                    arr[j] = argv[i][j] - 2;
                }
                else if (argv[i][j] == 65)
                {
                    arr[j] = 'Y';
                }
                else if (argv[i][j] == 66)
                {
                    arr[j] = 'Z';
```

```
41                  else if (argv[i][j] == 66)
42                  {
43                      arr[j] = 'Z';
44                  }
45                  else if (argv[i][j] == 97)
46                  {
47                      arr[j] = 'y';
48                  }
49                  else if (argv[i][j] == 98)
50                  {
51                      arr[j] = 'z';
52                  }
53                  else
54                  {
55                      arr[j] = argv[i][j];
56                  }
57                  j++;
58                  last = j;
59              }
60              arr[last+1] = '\0';
61              write(fd, arr, last);
62              char space[2];
63              space[0] = ' ';
64              space[1] = '\0';
65              write(fd, space, 1);
66          }
67      }
68      char arr[2];
69      arr[0] = '\n';
70      arr[1] = '\0';
71      write(fd, arr, 1);
72      close(fd);
73      exit();
74  }
```

:Makefile

```
_forktest: forktest.o $(ULIB)
    # forktest has less library code linked in - needs to be small
    # in order to be able to max out the proc table.
    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o
    $(OBJDUMP) -S _forktest > forktest.asm

mkfs: mkfs.c fs.h
    gcc Wall -o mkfs mkfs.c

# Prevent deletion of intermediate files, e.g. cat.o, after first build, so
# that disk image changes after first build are persistent until clean.  More
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

UPROGS=\
    _cat\
    _decode\
    _encode\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\

```

مقدمه ای درباره سیستم عامل xv6

1.سه وضیفه اصلی سیستم عامل:

مدیریت منابع :سیستم‌عامل منابع سخت‌افزاری (مانند پردازنده، حافظه، و دیسک) را مدیریت می‌کند و از تخصیص بهینه و عادلانه آن‌ها به فرآیندها اطمینان حاصل می‌کند.

مدیریت فرآیندها :سیستم‌عامل ایجاد، اجرا، و خاتمه فرآیندها را مدیریت می‌کند. همچنین وظایف مربوط به همزمانی و هماهنگی بین فرآیندها را انجام می‌دهد.

مدیریت ورودی/خروجی :سیستم‌عامل ارتباطات بین کاربر و سخت‌افزار را مدیریت می‌کند، از جمله ورودی و خروجی داده‌ها از دستگاه‌های مختلف مانند کیبورد، ماوس، و چاپگرها. این شامل کنترل و هماهنگی با درایورهای دستگاه نیز می‌شود.

2. فایلهای اصلی سیستم عامل xv6 در صفحه یک کتاب xv6 لیست شده اند. به طور مختصر هر گروه را توضیح دهید. نام پوشه اصلی فایلهای هسته سیستم عامل فایل های سرایند و فایل سیستم در سیستم عامل لینوکس چیست؟ در مورد محتویات آن مختصراً توضیح دهید.

1. basic headers : این بخش شامل فایل هایی است که شامل ساختار های موردنیاز برای کار با xv6 و ارتباط با کرنل آن است. به عبارتی داده ساختار ها و توابعی که کمک میکنند تا کد های سیستم عامل را بهتر بفهمیم.

2. system calls : این گروه رابطی بین یوزر و هسته سیستم عامل است و شامل توابعی است که برای i/o و مدیریت فرآیند ها به کار میرود.

3. string operations : این گروه شامل بخشی از کتابخانه های استاندارد هستند که برای مدیریت کردن string ها به کار می رود.

4. entering xv6 : این گروه شامل entry.S , entryother.S , main.c میباشد. فایل entry.S که در هنگام بوت شدن سیستم اجرا میشود نقش بسیار مهمی در راه اندازی سیستم دارد. همچنین بعد از بوت شدن کنترل به کد هسته منتقل میشود وکد هسته هم در main.c قرار دارد.

5. file system : مسئول مدیریت ذخیره‌سازی داده‌ها و فایل‌ها می‌باشد.

6. low level hardwawre : این بخش شامل تعامل با کنترلرهای سخت‌افزاری مختلف مانند صفحه‌کلید، نمایشگر و دیسک‌هاو مدیریت منابع است.

7. locks : این گروه به این منظور طراحی شده که از رقابت کردن کامپوننت های مختلف برای دسترسی به منابع مشترک جلوگیری کند. Lock ها برای هماهنگ بودن داده ها در بخش های مختلف کد های این سیستم عامل به کار میروند.

8. user level: بخشی از سیستم است که فرآینهای یوزر ها در آن هندل میشود و فرآیند هایی که در این سطح در جریان هستند کاملا از سطح هسته جدا میباشند.در نتیجه در این سطح به طور مستقیم دسترسی به سخت افزار نداریم و از طریق سیستم کال ها با سخت افزار ارتباط برقرار میکنیم.

9. processes : یک فرایند در xv6 به عنوان یک نمونه در حال اجرا از یک برنامه تعریف می‌شود که شامل فضای آدرس اختصاصی خود و مجموعه‌ای از منابع است. مدیریت فرایندها در xv6 از طریق چندین فایل و ساختار پیاده‌سازی شده است.

10. pipes : (در بخش های دیگر این پروژه تعریف کردیم.)

11. link: در سیستم‌عامل xv6، link برای ایجاد یک نام جدید (hard link) برای یک فایل موجود استفاده می‌شود. این فراخوان سیستمی به شما اجازه می‌دهد تا یک فایل را با چندین نام مختلف در دایرکتوری‌های مختلف دسترسی داشته باشید.

نام پوشه اصلی فایل‌های هسته سیستم عامل/boot :

پوشه /boot شامل فایل‌های مهمی برای راه‌اندازی (Boot) سیستم است. این پوشه شامل:

کرنل لینوکس :فایل‌هایی که هسته سیستم عامل را تشکیل می‌دهند. برای مثال، در لینوکس یک فایل با نامی مانند vmlinuz یا vmlinux وجود دارد که کرنل فشرده شده است.

GRUB (Grand Unified Bootloader):فایل‌های بوت‌لودر برای مدیریت و بارگذاری سیستم عامل.

نام پوشه فایل‌های سرایند/usr/include: (Header Files)

/usr/include پوشه‌ای است که فایل‌های header قرار دارند. این فایل‌ها شامل اعلان‌ها و توابع کتابخانه‌های استاندارد و همچنین رابط‌های سیستم عامل با کاربران و برنامه‌ها هستند.

فایل‌های سرایند شامل مواردی مانند stdio.h, stdlib.h, و unistd.h می‌شوند که اعلان توابع استاندارد C را ارائه می‌دهند..

نام پوشه فایل سیستم/ :(File System)

پوشه / که به عنوان root directory شناخته می‌شود، ریشه سیستم فایل در لینوکس است. تمامی فایل‌ها و دایرکتوری‌های دیگر از این دایرکتوری منشعب می‌شوند. زیرمجموعه‌های مهمی در این دایرکتوری مانند /bin , /lib , غیره هستند .

کامپایل سیستم عامل xv6

3.

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

در فایل Makefile سیستم عامل xv6، متغیرهای UPROGS و ULIB نقش مهمی در کامپایل و لینک کردن برنامه‌های کاربری و کتابخانه‌های آن‌ها دارند.. با استفاده از این دو متغیر، سیستم بیلد xv6 می‌تواند به صورت خودکار برنامه‌ها و کتابخانه‌ها را به درستی کامپایل و لینک کند.

:UPROGS

این متغیر حاوی لیستی از برنامه‌های کاربری است که باید برای سیستم عامل کامپایل شوند. هر برنامه‌ای که در این لیست تعریف شده باشد، در فرآیند بیلد به عنوان بخشی از برنامه‌های کاربری کامپایل شده و برای اجرا در محیط xv6آماده می‌شود. به طور کلی، این متغیر شامل نام‌های فایل‌های اجرایی مانند sh (shell)، cat و غیره است. این فایل‌های اجرایی بعد از کامپایل به عنوان برنامه‌های کاربری در دسترس قرار می‌گیرند.

:ULIB

این متغیر اشاره به کتابخانه‌های کاربری دارد که برنامه‌های کاربری در هنگام کامپایل به آن‌ها لینک می‌شوند. این کتابخانه‌ها شامل توابع و کدهایی هستند که برنامه‌های کاربری به آن‌ها وابسته هستند. ULIB شامل توابع سیستم و سایر توابع مشترک برای برنامه‌های کاربری هستند.

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

دیسک اول:(Kernel Disk)

این دیسک حاوی فایل باینری کرنل (هسته) سیستم عامل است که نتیجه اصلی فرایند بیلد است. این فایل شامل کد هسته سیستم عامل xv6 است که مسئول مدیریت منابع سیستم، مدیریت پردازه‌ها، حافظه و سایر بخش‌های اصلی سیستم عامل است.

دیسک دوم:(File System Disk)

این دیسک حاوی سیستم فایل است که شامل برنامه‌های کاربری و فایل‌های مورد نیاز برای اجرای سیستم عامل xv6 می‌باشد. این دیسک شبیه یک دیسک مجازی است که به عنوان سیستم فایل برای xv6 عمل می‌کند. محتوای آن شامل برنامه‌های کاربری کامپایل شده مانند sh (Shell)، ls، echoو غیره می‌باشد. این دیسک که شامل فایل‌های مورد نیاز برای استفاده توسط سیستم و کاربران است معمولاً با نام fs.imgشناخته می‌شود.

مراحل بوت سیستم عامل xv6

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

دستور objcopy برای تبدیل فایل‌های اجرایی، (object files)، و کتابخانه‌های لینک استفاده می‌شود. در فایل Makefile سیستم عامل xv6، این دستور معمولاً در مراحل پایانی بیلد به عنوان فرآیند نهایی ساخت نرم افزار به کار می رود. این دستور فایل های اجرایی را به فرمتی باینری تبدیل کرده و در نتیجه بوت لودر میتواند از آن استفاده کند. برای مثال شبیه ساز QEMU بیشتر مواقع فایل های باینری را به عنوان input دریافت میکند پس میتوان با استفاده از objcopy فایل های خروجی را به فرمت مورد نیاز تبدیل کرد. از طرفی در بسیاری از مواقع فایل های خروجی دریافتی شامل دیتاهایی هستند که صرفا فایل را سنگین کرده اند. با استفاده از دستور objcopy میتوان اطلاعات غیرضروری را حذف کرد تا بتوان فایل خروجی بهینه تری با دیتای مورد نیاز داشته باشیم.

13. کد bootmain.c هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در آدرس x1000000 قرار میدهد علت انتخاب این آدرس چیست؟

انتخاب آدرس 0x100000(یا 1 مگابایت) برای بارگذاری هسته سیستم‌عامل در سیستم‌هایی مانند xv6به دلایل تاریخی و فنی مرتبط با معماری سیستم‌های x86است. در اینجا چند دلیل کلیدی برای انتخاب این آدرس وجود دارد:

محفوظ بودن آدرس‌های پایین‌تر از 1 مگابایت برای سیستم BIOS و دستگاه‌ها:

در سیستم‌های x86، بخش‌های پایین‌تر حافظه، به خصوص محدوده آدرس‌های زیر 1مگابایت (از 0 x000000تا xFFFFF0) برای استفاده توسط BIOS و سایر سخت‌افزارهای سیستم رزرو شده است. این بخش شامل ناحیه‌هایی برای Interrupt Vector Table، BIOS data area، و دستگاه‌هایی مانند کارت‌های گرافیک است. بنابراین، بارگذاری هسته سیستم‌عامل در این ناحیه می‌تواند باعث ایجاد تداخل با عملکرد سیستم یا دستگاه‌های جانبی شود.

آدرس 0x100000 معمولاً به عنوان نقطه شروعی برای بارگذاری هسته در نظر گرفته می‌شود، چرا که بالاتر از محدودیت حافظه 1 مگابایتی حالت واقعی است و از هرگونه تداخل با BIOS و دستگاه‌های جانبی جلوگیری می‌کند

(Protected Mode) :

معماری x86در ابتدا برای پشتیبانی از سیستم‌های 16 بیتی طراحی شده بود، اما بعدها قابلیت کار با حافظه‌های بزرگ‌تر با معرفی حالت Protected Mode اضافه شد. در حالت واقعی (Real Mode) که سیستم با روشن شدن در آن آغاز به کار می‌کند، دسترسی به حافظه به 1 مگابایت محدود است. با ورود به حالت Protected Mode، CPUمی‌تواند به آدرس‌های بالاتر از 1 مگابایت دسترسی پیدا کند.

استانداردهای قدیمی:

انتخاب آدرس 0x100000 به یک استاندارد در بسیاری از سیستم‌عامل‌های مبتنی بر معماری x86 تبدیل شده است، چرا که با بسیاری از سیستم‌های قدیمی سازگار بوده و سیستم‌عامل‌ها و بوت لودرهایی مانند GRUB نیز این آدرس را برای بارگذاری هسته استفاده می‌کنند.

18. علاوه بر صفحه بندی در حد ابتدایی از قطعه بندی به منظور حفاظت هسته استفاده خواهد شد. این عملیات توسط seginite انجام میگردد همان طور که ذکر شد ترجمه قطعه تأثیری بر ترجمه آدرس منطقی نمیگذارد. زیرا 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 دستورالعملها و نه آدرس است.

فلگ SEG USER به این منظور استفاده می شود که پردازنده های سطح هسته و پردازنده های سطح کاربر

قابل تفکیک و تمایز باشند.

اجرای نخستین برنامه سطح کاربر

19. تا به این لحظه از اجرا فضای آدرس حافظه هسته آماده شده است. بخش زیادی از مابقی تابع ()main زیرسیستمهای مختلف هسته را فعال مینماید مدیریت برنامه های سطح کاربر مستلزم ارائه انتزاعاتی برای ایجاد تمایز میان این برنامهها و برنامه مدیریت آنها .است کدی که تاکنون اجرا میشد را میتوان برنامه مدیریت کننده سیستم و برنامههای سطح کاربر دانست. 19. جهت نگهداری 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.

اجزای اصلی struct proc درxv6 :

1. State : این وضعیت نشان می‌دهد که فرآیند در چه مرحله‌ای از اجرا قرار دارد.
2. pid : شناسه فرآیند یا .Process ID هر فرآیند یک شناسه منحصربه‌فرد دارد که توسط سیستم‌عامل برای تمایز فرآیندها استفاده می‌شود.
3. parent : اشاره‌گری به فرآیند والد. این بخش به سیستم‌عامل کمک می‌کند تا فرآیندها را در قالب سلسله‌مراتبی سازمان‌دهی کند.
4. sz : اندازه حافظه فرآیند. این متغیر نشان می‌دهد که چه مقدار حافظه به فرآیند اختصاص داده شده است.
5. kstack : اشاره‌گری به پشته کرنل فرآیند. پشته کرنل فرآیند برای ذخیره وضعیت اجرای کرنل در طول زمان‌بندی و دستورات سیستمی استفاده می‌شود.
6. pgdir : اشاره‌گری به دایرکتوری صفحات حافظه (Page Directory) فرآیند. این بخش از ساختار فرآیند برای مدیریت حافظه مجازی و تبدیل آدرس‌های مجازی به فیزیکی استفاده می‌شود.
7. context : اطلاعاتی درباره وضعیت رجیسترهای CPU زمانی که فرآیند توسط زمان‌بند (scheduler) از اجرا باز می‌ماند.
8. killed : نشان‌دهنده این است که آیا فرآیند به دستور دیگری "کشته" شده است یا نه.
9. ofile : آرایه‌ای از اشاره‌گرها به فایل‌های باز شده توسط فرآیند.
10. cwd : دایرکتوری کاری فعلی فرآیند.
11. name : نام فرآیند برای اهداف تشخیصی.

معادل struct proc در سیستم عامل لینوکس:

در سیستم‌عامل لینوکس، معادل ساختار struct proc درxv6 ، ساختار task_struct است. این ساختار شامل اطلاعات مدیریتی فرآیند در سیستم‌عامل لینوکس است.

اجزای اصلی task_struct :

1. pid : شناسه فرآیند.
2. comm :نام فرآیند.

3. state : وضعیت فرآیند.
4. parent : اشاره‌گر به فرآیند والد.
5. stack : اشاره‌گر به پشته کرنل فرآیند.
6. mm : اشاره‌گر به فضای آدرس حافظه مجازی فرآیند.
7. files : اشاره‌گر به ساختار فایل‌های باز شده توسط فرآیند.

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 زمانبند روی کدام هسته اجرا میشود؟

در سیستم عامل‌های چند هسته‌ای مانند xv6، برخی بخش‌های آماده‌سازی و منابع سیستم به صورت مشترک بین تمامی هسته‌های پردازنده استفاده می‌شوند، در حالی که برخی بخش‌ها به صورت اختصاصی برای هر هسته پردازنده تنظیم می‌شوند. در اینجا به بررسی نمونه‌هایی از این موارد می‌پردازیم:

بخش مشترک:

جدول صفحات : (Page Table) جدول صفحات معمولاً برای فضای آدرس هسته مشترک بین تمامی هسته‌های پردازنده است. دلیل این اشتراک این است که هسته سیستم عامل باید به منابع سیستمی و حافظه‌های مشترک دسترسی داشته باشد و تمامی پردازنده‌ها باید بتوانند به یک فضای آدرس یکسان برای هسته دسترسی پیدا کنند. این اشتراک گذاری باعث می‌شود که تمامی هسته‌ها بتوانند به صورت کارآمد به حافظه هسته دسترسی داشته باشند.

بخش اختصاصی:

TSS (Task State Segment) : هر هسته پردازنده دارای یک TSS جداگانه است که وظیفه ذخیره وضعیت وظیفه (task) فعلی را بر عهده دارد. به همین دلیل، هر هسته به صورت جداگانه باید TSS خود را داشته باشد تا بتواند کانتکست سوئیچینگ و مدیریت پروسه‌های اختصاصی را انجام دهد.

زمانبند:

در xv6، زمانبند بر روی هر هسته پردازشی به صورت مستقل اجرا می‌شود. به عبارت دیگر، هر هسته پردازنده مسئول زمانبندی و اجرای فرآیندهای خودش است. به این روش، هر هسته به طور مستقل وظایف را از لیست فرآیندها برداشته و به اجرا می‌گذارد. سیستم عامل تلاش می‌کند که بار پردازشی را بین هسته‌ها تقسیم کند تا از منابع بهینه‌تر استفاده شود.

GDB :

1.برای مشاهده Breakpointها از چه دستوری استفاده میشود؟

کافی است از info breakpoints استفاده کنیم.

```
line to your configuration file "/home/alizm/.config/gdb/gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
--Type <RET> for more, q to quit, c to continue without paging--c
line to your configuration file "/home/alizm/.config/gdb/gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) b handle_down_arrow
Breakpoint 1 at 0x801009aa: file console.c, line 499.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x801009aa in handle_down_arrow
                                           at console.c:499
(gdb)
```

```
496
497   static void handle_down_arrow()
498   {
499       input = history.instructions[history.index--];
500       input.buf[--input.e] = '\0';
501   }
502
```

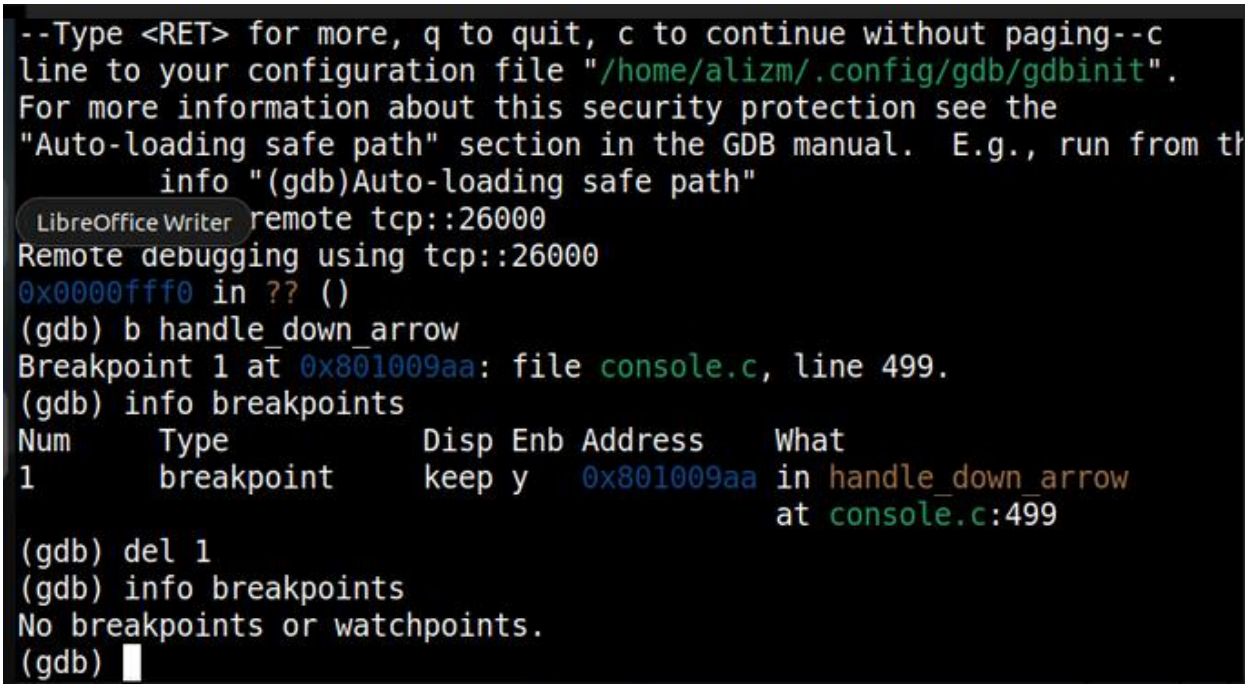
2.برای حذف یک Breakpoint از چه دستوری و چگونه استفاده میشود؟

کافی است ابتدا مثل سوال بالا از دستور info breakpoints استفاده کنیم تا بتوانیم شماره هر کدام از breakpoint ها را به دست بیاریم. پس از آن با داشتن شماره های breakpoint ها میتوانیم با استفاده از دستور del آن ها را حذف کنیم.

برای استفاده از del در کنار شماره های به دست آمده دو انتخاب داریم:

1_ del را بدون شماره وارد کنیم(دستور Del به تنهایی بیاید) ← تمامی breakpoint ها حذف میشوند.

2_اگر دستور del را به همراه شماره های به دست آمده بزنیم ← breakpoint با شماره داده شده حذف میشود



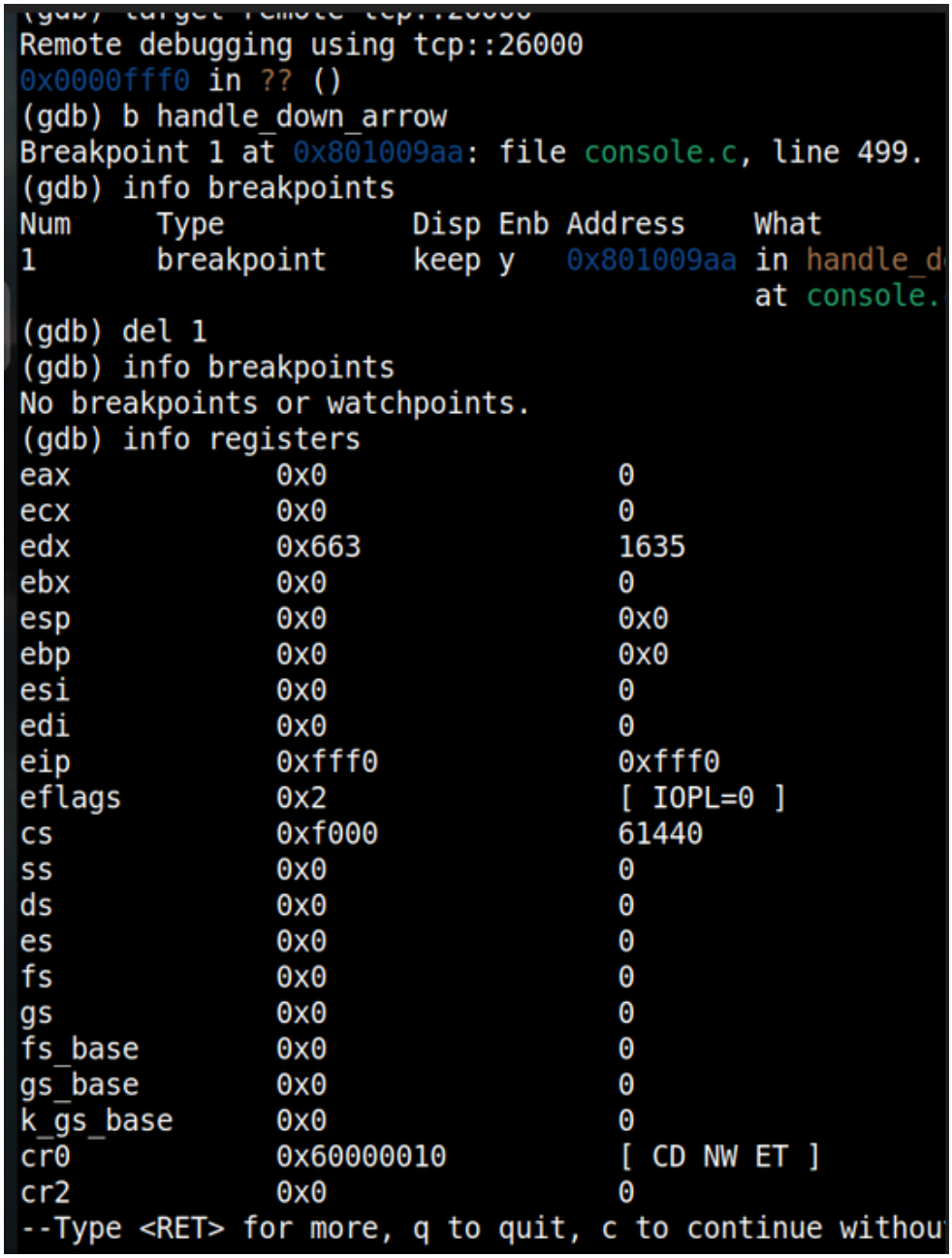
3. درGDB ، دستور bt مخفف Backtrace است و زمانی که آن را وارد می‌کنید، لیستی از پشته‌ی فراخوانی‌ها (Call Stack)نمایش داده می‌شود. این پشته مجموعه‌ای از فراخوانی‌های توابع است که منجر به وضعیت فعلی اجرای برنامه شده است.

4. در gdb دستور print به منظور نمایش value یک عبارت به کار میرود. این دستور می‌تواند یک متغیر، اشاره‌گر یا هر عبارت معتبر C را به همراه مقدار آن چاپ کند. اما دستور xبرای بررسی محتویات حافظه در آدرس مشخصی استفاده می‌شود. این دستور به شما اجازه می‌دهد تا حافظه را به صورت خام در قالب‌های مختلف مشاهده کنید. به همین دلیل بیشتر برای مشاهده مستقیم محتوای حافظه استفاده می‌شود. پس می توان گفت که دستور print رای مشاهده مقادیر متغیرها در سطح بالا مناسب است اما دستور x برای بازرسی دقیق آدرس‌های حافظه و داده‌های سطح پایین مناسب است. از طرف دیگر در دستور x چون با محتوای حافظه خام سر و کار داریم کنترل بیشتری روی فرمت داده های مورد بررسی داریم در مقایسه با دستور print.

محتوای یک ثبات خاص را میتوان با استفاده از دستور print و گذاشتن علامت $ قبل از نام ثبات محتوای موردنظر را چاپ کرد.

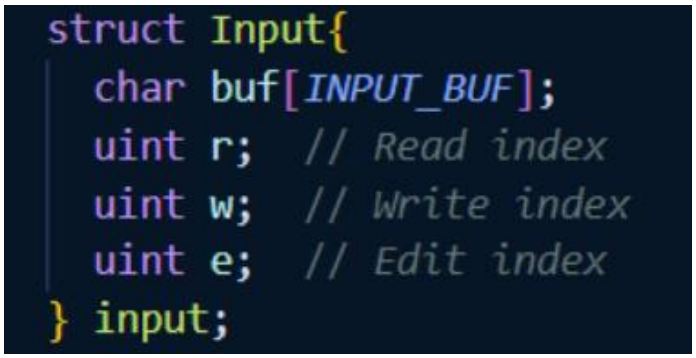
5. با استفاده از دستور info registers میتوان محتوای registerها و با استفاده از دستور info locals میتوان محتوای local variableها را نمایش داد.

در معماری x86، ثبات‌های EDIو ESIبه ترتیب به عنوان "Destination Index" و "Source Index" شناخته می‌شوند. این ثبات‌ها معمولاً برای عملیات مربوط به آرایه‌ها و رشته‌ها استفاده می‌شوند. EDI معمولاً به عنوان نشانگر مقصد برای عملیات کپی و مقایسه استفاده می‌شود و از طرف دیگرESI به عنوان نشانگر منبع برای عملیات مربوط به کپی و مقایسه استفاده می‌شود.




```
xmm7          {v4_float = {0x0, 0x0, 0x0, 0x1f80}, v2_double = {0x0, 0x1f8000000000}, v16_int8 = {0x0 <repeats 12 times>, 0x80, 0x1f, 0x0, 0x0},
--Type <RET> for more, q to quit, c to continue without paging--info locals
, 0x0, 0x0, 0x0, 0x0, 0x1f80, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x1f80}, v2_int64 = {0x0, 0x1f8000000000}, uint128 = 0x1f80000000000000000000000000
mxcsr         0x1f80             [ IM DM ZM OM UM PM ]
(gdb) info locals
No symbol table info available.
(gdb)
```

.6



همانطور که مشاهده می شود ساختار Input برای ذخیره سازی ورودی در نظر گرفته شده. متغیر buf ورودی در کنسول را ذخیره سازی میکند.متغیر e پوینتر به آخرین حرف به عنوان ورودی می باشد.e برای ادیت به کار می رود یعنی با backspace کردن e یک واحد کم میشود و با وارد شدن هر کارکتر به ورودی یک واحد به آن اضافه میشود. همچنین w پوینتر به ابتدای آخرین سطر ورودی استپس طبیعتا با زدن enter مقدارش یک واحد افزایش می یابد. r هم متغیری است که نشان میدهد چه تعداد بایتی از بافر خوانده شده است.

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

دستورهای layout src و layout asm برای تغییر نحوه نمایش محتوای برنامه در TUI استفاده می‌شوند .

در صورت اجرای layout src، خطوط کد منبع برنامه به همراه اطلاعات مربوط به متوقف شدن (breakpoints) و موقعیت فعلی برنامه نمایش داده می‌شوند. از طرف دیگر layout asm خطوط کد اسمبلی به همراه آدرس‌ها و دستورالعمل‌ها را نمایش میدهد.

8. با استفاده از دستورات up و down میتوان به فریم های بالاتر و پایینتر در زنجیره فراخوانی (call stack) منتقل شد. با استفاده از frame هم میتوان به یک فریم خاص در زنجیره فراخوانی رفت.به صورت روبرو باید شماره فریم موردنظر را به عنوان آرگومان داد: frame<n>;