صدرا مداینی اول 810102564
امیرحسین علیخانی 810102479
هستی ابوالحسنی 810102378

برای اینکه اول از همه اسم اعضای گروه نمایش داده شود باید در فایل init.c تغییراتی بعد از حلقه for اعمال کنیم که با یه printf ساده میتوان آن را پیاده سازی کرد و آن را نمایش داد.

```
for(;;){
  printf(1, "init: starting sh\n");
  printf(1, "Sadra Madayeni , AmirHossein Alikhani, Hasti Abolhassani \n");
  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");
  }
}
```

- قابلیت جابجایی پوینتر با استفاده از کلید های روی کیبرد.

برای پیاده سازی این قابلیت اول از هم باید ببینیم هر کلید داخل این سیستم به چه آدرسی مپ شده که این آدرس هارو از فایل kbd.h میتونیم پیدا کنیم الان فقط ما به کلید چپ و راست نیاز داریم.

که در فایل kbd.h به این آدرس ها مپ شده اند.

```
// Special keycodes
#define KEY_HOME        0xE0
#define KEY_END         0xE1
#define KEY_UP          0xE2
#define KEY_DN          0xE3
#define KEY_LF          0xE4
#define KEY_RT          0xE5
#define KEY_PGUP        0xE6
#define KEY_PGDN        0xE7
#define KEY_INS         0xE8
#define KEY_DEL         0xE9
```

برای سمت راست به آدرس 0xE5 و سمت چپ 0xE4 استفاده میکنیم

باید یه سری تغییراتی در فایل های kbd.c و console.c انجام دهیم که با زدن دکمه ها پوینتر ما هم جابجا شود

در فایل kbd.c دوتا عملیات خیلی ساده تعریف میکنیم

```
if(c == KEY_LF) {
  return KEY_LF;
}

if(c == KEY_RT) {
  return KEY_RT;
}
```

و برای انجام کار اصلی به فایل console.c میرویم

باید توی فایل cgaputc تغییراتی انجام بدیم که با زدن KEY_LF و KEY_RT پوینتر ما پوزیشن جدیدی به خودش بگیره علاوه بر اینکه صفحه آپدیت میشه. (و البته از صفحه بیرون نزنه)

```
// Cursor position: col + 80*row.
outb(CRTPORT, 14);
pos = inb(CRTPORT+1) << 8;
outb(CRTPORT, 15);
pos |= inb(CRTPORT+1);

if(c == '\n')
  pos += 80 - pos%80;
else if(c == BACKSPACE){
  if(pos > 0) --pos;
  // Erase character from screen memory
  crt[pos] = (' ' & 0xff) | 0x0700;
} else if(c == KEY_LF) {  // Left arrow key
  if(pos > 0) --pos;
} else if(c == KEY_RT) {  // Right arrow key
  if(pos < 24*80 - 1) ++pos;
} else {
  crt[pos++] = (c&0xff) | 0x0700;
}
```

داخل استراکتی که برامون تعریف شده و با استفاده از اون میتونیم اینپوت رو هندل کنیم یه متغیر

Mouse_pos

تعریف میکنیم (که با استفاده از input.e و یا غیره موقعیت پوینتر رو تغییر ندیم چون باعث به هم خوردن شیفت سیستم و باگ میشه)

اول از همه سمت چپ رفتن رو هندل کردیم چون بدون چپ رفتن نمیتونستیم راست بریم

```
case KEY_LF:  // Left arrow
  {
    uint effective_end = input.e;
    while (effective_end > input.w && input.buf[(effective_end - 1) % INPUT_BUF] == ' ') {
      effective_end--;
    }

    if (input.mouse_pos > effective_end) {
      int moves = input.mouse_pos - effective_end;
      input.mouse_pos = effective_end;
      for (int i = 0; i < moves; i++) {
        consputc(KEY_LF);
      }
    }
    else if (input.mouse_pos > input.w) {
      input.mouse_pos--;
      consputc(KEY_LF);
    }
  }
  break;
```

و بعدش هم سمت راست رفتن پوینتر با استفاده از شرط هایی مثل اینه اگه به ته رسید متوقف بشه و ... پیاده سازی کردیم

```
case KEY_RT:  // Right arrow
  {
    uint effective_end = input.e;
    while (effective_end > input.w && input.buf[(effective_end - 1) % INPUT_BUF] == ' ') {
      effective_end--;
    }

    if (input.mouse_pos < effective_end) {
      consputc(KEY_RT);
      input.mouse_pos++;
    }
  }
  break;
```

- Ctrl +
  D قابلیت

برای این قابلیت یه کیس به تابع console interface اضافه میکنیم که در صورتی که یوزر دکمه کنترل همراه با d رو بزنه این اتفاق بیوفته

```
case C('D'): // Move cursor to the BEGINNING of the next word.
    if (input.mouse_pos < input.e) {
      uint pos = input.mouse_pos;
      while (pos < input.e && input.buf[pos % INPUT_BUF] != ' ') {
        pos++;
      }
      while (pos < input.e && input.buf[pos % INPUT_BUF] == ' ') {
        pos++;
      }
      int moves = pos - input.mouse_pos;
      input.mouse_pos = pos;
      for (int i = 0; i < moves; i++) {
        consputc(KEY_RT);
      }
    }
    break;
```

دستور A

برای استفاده از این دستور نیز یه کیس باید ایجاد کنیم به شکل زیر (توضیحات بیشتر داخل کامنت ها داده شده که عملکرد کلی چجوریه)

Ctrl + ●

```
case C('A'): // Move cursor to the beginning of the current/previous word.
  if (input.mouse_pos > input.w) {
    uint new_pos = input.mouse_pos;

    // Check if the cursor is at the start of a word or on a space.
    // We do this by checking the character immediately to the left.
    if (input.buf[(new_pos - 1) % INPUT_BUF] == ' ') {
      // BEHAVIOR 2: Already at start or on space -> go to PREVIOUS word start.
      // 1. Scan backwards past any spaces.
      while (new_pos > input.w && input.buf[(new_pos - 1) % INPUT_BUF] == ' ') {
        new_pos--;
      }
      // 2. Scan backwards past the previous word's characters.
      while (new_pos > input.w && input.buf[(new_pos - 1) % INPUT_BUF] != ' ') {
        new_pos--;
      }
    } else {
      // BEHAVIOR 1: In the middle of a word -> go to CURRENT word start.
      // 1. Scan backwards past the current word's characters.
      while (new_pos > input.w && input.buf[(new_pos - 1) % INPUT_BUF] != ' ') {
        new_pos--;
      }
    }

    // Move the cursor to the new position.
    int moves = input.mouse_pos - new_pos;
    input.mouse_pos = new_pos;
    for (int i = 0; i < moves; i++) {
      consputc(KEY_LF);
    }
  }
  break;
```

قابلیت Z

برای پیاده سازی این قابلیت ما نیاز داریم به یه استراکت که بتونیم اون هیستوری که داشتیم از کلمات ورودی رو ذخیره کنیم و بر اساس تایم اونارو بتونیم حذف کنیم!

(مثلا اگر کاربر یه کلمه ای نوشت و وسط اون کلمه یه سری اعداد/حروف/اسپیس و یا هرچی اضافه کرد رو بتونه با استفاده از کنترل زد پاک کنه)

Ctrl + ●

```c
#define HISTORY_BUF 256
// Defines the type of operation: an insertion or a deletion
typedef enum { OP_INSERT, OP_DELETE } op_type;

// single edit operation (what was done, what character, and where)
typedef struct {
  op_type type;
  char c;
  uint pos;
} edit_op;

// A buffer to store the history of operations for the current line
static struct {
  edit_op ops[HISTORY_BUF];
  uint h_index; // Points to the next available slot in the history buffer
} history;
```

پیاده سازی کیس Ctrl + Z

```
case C('Z'):  // Undo last operation
  if (history.h_index > 0) {
    history.h_index--;
    edit_op* op = &history.ops[history.h_index];

    if (op->type == OP_INSERT) {
      // UNDO INSERTION by performing a deletion
      for (uint i = op->pos; i < input.e; i++) {
        input.buf[i % INPUT_BUF] = input.buf[(i + 1) % INPUT_BUF];
      }
      input.e--;

      // Move visual cursor to where the change happened
      if(input.mouse_pos > op->pos) for(uint i = 0; i < input.mouse_pos - op->pos; i++) consputc(KEY_LF);
      else for(uint i = 0; i < op->pos - input.mouse_pos; i++) consputc(KEY_RT);

      // Redraw line
      for (uint i = op->pos; i < input.e; i++) consputc(input.buf[i % INPUT_BUF]);
      consputc(' ');
      for (uint i = op->pos; i <= input.e; i++) consputc(KEY_LF);
      input.mouse_pos = op->pos;
     // UNDO DELETION by performing an insertion
    } else {
      if (input.e - input.r >= INPUT_BUF) {
        history.h_index++; // Cannot undo, revert history index
        break;
      }
      for (uint i = input.e; i > op->pos; i--) {
        input.buf[i % INPUT_BUF] = input.buf[(i - 1) % INPUT_BUF];
      }
      input.buf[op->pos % INPUT_BUF] = op->c;
      input.e++;

      // Move visual cursor to where the change happened
      if(input.mouse_pos > op->pos) for(uint i = 0; i < input.mouse_pos - op->pos; i++) consputc(KEY_LF);
      else for(uint i = 0; i < op->pos - input.mouse_pos; i++) consputc(KEY_RT);

      // Redraw line
      for (uint i = op->pos; i < input.e; i++) consputc(input.buf[i % INPUT_BUF]);
      for (uint i = op->pos + 1; i < input.e; i++) consputc(KEY_LF);
      input.mouse_pos = op->pos + 1;
    }
  }
  break;
```

- پیاده سازی Copy Paste

```
static int selection_mark = -1;
static int selection_start = -1;
static int selection_end = -1;
static char clipboard_buf[INPUT_BUF];
static int clipboard_len = 0;
```

```
static void
clear_selection_highlight(void)
{
  if (selection_start == -1)
    return;

  int screen_start = input.start_pos + (selection_start - input.w);
  int screen_end = input.start_pos + (selection_end - input.w);

  for (int i = screen_start; i < screen_end; i++) {
    crt[i] = (crt[i] & 0xff) | 0x0700;
  }

  selection_start = -1;
  selection_end = -1;
  selection_mark = -1;
}
```

برای ایجاد قابلیت کپی پیست این متغیر ها و این تابع کمکی رو تعریف کردیم . برای پیاده سازی کنترل اس به پوزیشن کورسور نیاز داریم و به این ترتیب پیاده سازیش میکنیک

```
    break;

case C('S'):
    if (selection_mark == -1) {
        clear_selection_highlight();
        selection_mark = input.mouse_pos;
    } else {
        uint start = selection_mark;
        uint end = input.mouse_pos;

        if (start > end) {
            uint tmp = start;
            start = end;
            end = tmp;
        }

        if (start == end) {
            selection_mark = -1;
            break;
        }

        selection_start = start;
        selection_end = end;

        int screen_start = input.start_pos + (selection_start - input.w);
        int screen_end = input.start_pos + (selection_end - input.w);

        for (int i = screen_start; i < screen_end; i++) {
            crt[i] = (crt[i] & 0xff) | 0x7000;
        }
        selection_mark = -1;
    }
    break;
```

برای کپی کردن رشته صرفا آن را در یک بافر کلیپبورد نگه میداریم

```
case C('C'):
    if (selection_start != -1 && selection_end != -1) {
        int len = selection_end - selection_start;
        if (len > 0 && len < INPUT_BUF) {
            for (int i = 0; i < len; i++) {
                clipboard_buf[i] = input.buf[(selection_start + i) % INPUT_BUF];
            }
            clipboard_len = len;
            clipboard_buf[clipboard_len] = '\0';
        }
        clear_selection_highlight();
    }
    break;
```

و برای پیست هم صرفا باید محتویات داخل بافر رو دوباره وارد کنسول کنیم
(محتویاتی که از قبل وارد بافر کلیپبورد ما شده بود از قبل)

```
case C('V'):
  if (clipboard_len > 0 && (input.e - input.r + clipboard_len < INPUT_BUF)) {
    clear_selection_highlight();

    for (int i = input.e - 1; i >= (int)input.mouse_pos; i--) {
      input.buf[(i + clipboard_len) % INPUT_BUF] = input.buf[i % INPUT_BUF];
    }

    for (int i = 0; i < clipboard_len; i++) {
      input.buf[(input.mouse_pos + i) % INPUT_BUF] = clipboard_buf[i];
    }

    input.e += clipboard_len;

    for (uint i = input.mouse_pos; i < input.e; i++) {
      consputc(input.buf[i % INPUT_BUF]);
    }

    input.mouse_pos += clipboard_len;

    for (uint i = input.mouse_pos; i < input.e; i++) {
      consputc(KEY_LF);
    }
  }
  break;
```

این clear بعد هرکدوم از این کارا ما موظفیم که اون سفید شدن سلکت رو از بین ببریم که با اون تابع کارو انجام میدیم

● برنامه سطح کاربر

برای انجام این بخش یک فایل find_sum.c درست میکنیم و داخل آن شروع به پیاده سازی منطق چیزی که داخل پروژه آورده شده می کنیم

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


int main(int argc, char *argv[]) {
    int total_sum = 0;


    for (int i = 1; i < argc; i++) {
        char *p = argv[i];

        while (*p) {

            if (*p < '0' || *p > '9') {
                p++;
                continue;
            }


            int current_num = 0;
            while (*p >= '0' && *p <= '9') {
                current_num = current_num * 10 + (*p - '0');
                p++;
            }
            total_sum += current_num;
        }
    }


    int fd;
    fd = open("result.txt", O_CREATE | O_WRONLY);

    if (fd < 0) {
        printf(2, "Error: cannot open result.txt for writing\n");
        exit();
    }


    printf(fd, "%d\n", total_sum);
    close(fd);

    exit();
}
```

در قسمت MAKEFILE هم باید اون تابعی که زدیم رو به همون فرمت استفاده کنیم چون ما میخوایم به برنامه های اون سیستم عامل QEMU ما اضافه شه و به جای کامپایل کردن داخل خود ترمینال یا هر محیط دیگه ای توی سیستم عامل خود QEMU این رو اجرا میکنیم و داخل یه فایل

ذخیره میکنیم.

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

- پیاده سازی تب

```
void
update_completions(void)
{
  char buf[4 + MAX_COMMANDS * (MAX_CMD_LEN + 1)];
  char *p = buf;
  int fd;
  struct dirent de;

  if((fd = open(".", 0)) < 0){

    return;
  }

  *p++ = 1;
  *p++ = 1;

  while(read(fd, &de, sizeof(de)) == sizeof(de)){
    if(de.inum == 0 || strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)
      continue;

    int name_len = strlen(de.name);
    if ((p - buf) + name_len + 3 > sizeof(buf)) {
      break;
    }

    memmove(p, de.name, name_len);
    p += name_len;
    *p++ = '\n';
  }
  close(fd);

  *p++ = 2;
  *p++ = 2;

  write(1, buf, p - buf);
}
```

برای پیاده سازی تب به صورت داینامیک نیاز به تغییر دو چیز داریم
با تغییییر دادن شل و ذخیره کردن هر کامندی که در هر اجرای دستور وارد شل میشود آن را به صورت داینامیک پیاده سازی میکنیم و یه تابع زدیم به نام کامل کردن کلمات که به صورت بالا است
همچنین در تابع مین شل نیز این تابع را فراخانی کرده تا همیشه اپدیت باشیم

```
int
main(void)
{
  static char buf[100];
  int fd;

  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }

  while(1){

    update_completions();

    if(getcmd(buf, sizeof(buf)) < 0)
      break; // Handle EOF

    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      buf[strlen(buf)-1] = 0;
      if(chdir(buf+3) < 0)
        printf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));
    wait();
  }
  exit();
}
```

حالا به سراغ کنسول میرویم و تعامل با شل رو باید برقرار کنیم با استفاده از توابع زیر میتونیم این کارو انجام بدیم

```
static int
find_completions(const char* prefix, const char** completions)
{
  int count = 0;
  int prefix_len = kstrlen(prefix);

  if (prefix_len == 0) return 0;

  for (int i = 0; dynamic_command_ptrs[i] != 0 && count < MAX_COMPLETIONS; i++) {
    if (kstrncmp(prefix, dynamic_command_ptrs[i], prefix_len) == 0) {
      completions[count++] = dynamic_command_ptrs[i];
    }
  }
  return count;
}
```

```
static int
find_longest_common_prefix(const char** completions, int count, char* lcp_buf)
{
    if (count <= 0) {
        lcp_buf[0] = '\0';
        return 0;
    }

    const char* first_str = completions[0];
    int lcp_len = kstrlen(first_str);

    for (int i = 1; i < count; i++) {
        int j = 0;
        while (j < lcp_len && j < kstrlen(completions[i]) && first_str[j] == completions[i][j]) {
            j++;
        }
        if (j < lcp_len) {
            lcp_len = j;
        }
    }
    for (int i = 0; i < lcp_len; i++) {
        lcp_buf[i] = first_str[i];
    }
    lcp_buf[lcp_len] = '\0';
    return lcp_len;
}
```

و تابع هندل کامل کردن اینا کار اصلی رو انجام میده که به دلیل طولانی بودن اون فقط بخشی رو داخل گزارش اووردیم

```
static void
handle_tab_completion(void)
{
    char prefix[INPUT_BUF];
    int len = 0;
    for(uint i = input.w; i < input.e; i++) {
        if (input.buf[i % INPUT_BUF] == ' ') {
            return;
        }
        prefix[len++] = input.buf[i % INPUT_BUF];
    }
    prefix[len] = '\0';

    if (kstrncmp(prefix, last_prefix, INPUT_BUF) != 0) {
        tab_press_count = 0;
        memmove(last_prefix, prefix, len + 1);
    }
```

و یه کیس در نهایت به کنسول اضافه میکنیم

```
case TAB_KEY:
    clear_selection_highlight();
    handle_tab_completion();
    break;
```

1) سه وظیفه اصلی سیستم عامل را نام ببرید

1. مدیریت و اشتراک‌گذاری منابع :(Multiplexing) یکی از مهم‌ترین کارهاش اینه که منابع محدود سیستم، مثل پردازنده(CPU) ، رو بین چندین برنامه به اشتراک بذاره. درست مثل یه شعبده‌باز که چندتا توپ رو همزمان توی هوا نگه می‌داره، سیستم‌عامل هم حواسش هست که همه‌ی برنامه‌ها به نوبت از منابع استفاده کنن.
2. ساده‌سازی :(Abstraction) سیستم‌عامل پیچیدگی‌های سخت‌افزار رو از چشم برنامه‌ها پنهان می‌کنه. شما برای رانندگی لازم نیست از جزئیات موتور ماشین سر در بیارید، فقط با فرمون و پدال‌ها کار می‌کنید. سیستم‌عامل هم دقیقاً همین کار رو می‌کنه؛ به جای درگیر کردن برنامه‌ها با جزئیات پیچیده‌ی سخت‌افزاری، یه سری دستور ساده و یکپارچه در اختیارشون قرار میده.
3. ایزوله کردن و مدیریت تعامل :(Isolation and Interaction) وظیفه‌ی سومش اینه که یه محیط امن و کنترل‌شده بسازه. سیستم‌عامل برنامه‌ها رو از هم جدا (ایزوله) می‌کنه تا اگه یکی از اون‌ها به خاطر باگ یا مشکلی هنگ کرد، به بقیه‌ی برنامه‌ها آسیبی نزنه. در عین حال، یه راه امن هم برای برنامه‌ها فراهم می‌کنه تا بتونن از داده‌های مشترک استفاده کنن.

2) آیا وجود سیستم عامل در تمام دستگاه ها الزامی است؟ چرا؟ در چه شرایطی استفاده از سیستم عامل لازم است؟

خیر، وجود سیستم عامل به شکل هسته (Kernel) در تمام دستگاه‌ها الزامی نیست.
زیرا می‌توان خدمات سیستم عامل را به سادگی به عنوان یک کتابخانه (library) پیاده‌سازی کرد که برنامه‌های کاربردی مستقیماً با آن لینک می‌شوند. در این سازماندهی، برنامه‌ها می‌توانند مستقیماً با منابع سخت‌افزاری تعامل داشته باشند و منابع را به شیوه‌ای بهینه برای عملکرد خود استفاده کنند.
سیستم‌هایی که این رویکرد را در پیش می‌گیرند شامل برخی سیستم‌های عامل برای دستگاه‌های تعبیه‌شده (embedded devices) یا سیستم‌های بلادرنگ (real-time systems) هستند.
با این حال، این رویکرد (استفاده از کتابخانه) یک نقص عمده دارد: اگر بیش از یک برنامه در حال اجرا باشد، همه برنامه‌ها باید خوش‌رفتار (well-behaved) باشند و به طور مشارکتی (cooperative time-sharing)منابع را تقسیم کنند.
استفاده از یک سیستم عامل با هسته (kernel) زمانی حیاتی می‌شود که نیاز به جداسازی قوی (strong isolation)و چندگانه سازی (multiplexing) منابع بین برنامه‌های مختلف وجود داشته باشد.

3) معماری سیستم عامل xv6چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

سیستم‌عامل xv6 یک نسخه آموزشی و ساده‌شده از یونیکس است که برای پردازنده‌های x86 طراحی شده. این سیستم‌عامل با الهام از نسخه ششم یونیکس (Unix V6) ساخته شده و هدف اصلی آن آموزش مفاهیم پایه‌ای سیستم‌عامل‌هاست.
ساختار xv6 از سه بخش اصلی تشکیل شده: هسته(kernel) ، پوسته (shell) و برنامه‌های کاربردی. معماری هسته آن به صورت یکپارچه (monolithic) است؛ یعنی تمام اجزای اصلی سیستم‌عامل در حالت سوپروایزر (supervisor mode) اجرا می‌شوند. این طراحی ساده به ما کمک می‌کند تا راحت‌تر با ساختار داخلی یک سیستم‌عامل واقعی آشنا شوند.

4) سیستم عامل xv6یک سیستم تک وظیفه ای است یا چندوظیفه ای؟

سیستم‌عامل xv6 یک سیستم چندوظیفه‌ای (Multi-tasking) به حساب می‌آید.
اساساً، xv6طوری طراحی شده تا منابع سخت‌افزاری، به‌خصوص پردازنده(CPU) ، را بین چندین برنامه به اشتراک بگذارد. این کار را با استفاده از تکنیک زمان‌بندی (Scheduling) و تقسیم زمان (Time-sharing)انجام می‌دهد. به این ترتیب، سیستم‌عامل به سرعت بین فرآیندهای مختلف جابجا می‌شود و به هر کدام از آن‌ها این حس را القا می‌کند که یک پردازنده اختصاصی در اختیار دارند، حتی اگر تعداد فرآیندها از تعداد هسته‌های پردازنده بیشتر باشد.
علاوه بر این، xv6برای اجرا روی پردازنده‌های چند هسته‌ای مدرن (RISC-V) ساخته شده که ذاتاً نیازمند مدیریت همزمان چندین فرآیند است. وجود فراخوانی‌های سیستمی (system calls) معروفی مثل ()forkبرای ساخت یک فرآیند جدید و ()execبرای اجرای یک برنامه جدید در دل یک فرآیند، به وضوح قابلیت‌های چندوظیفه‌ای آن را نشان می‌دهد.
در نتیجه، xv6با مدیریت هوشمندانه فرآیندها و ایجاد بستری برای اجرای همزمان برنامه‌ها، یک نمونه کلاسیک و آموزشی از یک سیستم‌عامل چندوظیفه‌ای است.

5) همانطور که میدانید به طور کلی چندوظیفگی تعمیمی است از حالت چندبرنامگی. چه تفاوتی میان یک برنامه و یک پردازه وجود دارد؟

برنامه (Program) مثل یک دستور پخت بی‌جان روی کاغذه. اما فرآیند (Process) اون موجودیت زنده‌ایه که وقتی سیستم‌عامل دستور پخت رو برمیداره، مواد اولیه (داده‌ها) رو توی حافظه می‌ریزه و شروع به پختن می‌کنه. به عبارت دیگه، فرآیند همون نسخه در حال اجرای یک برنامه است که سیستم‌عامل بهش جون داده.

6) ساختار یک پردازه در سیستم عامل xv6از چه بخشهایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازه های مختلف اختصاص می دهد؟

در سیستم‌عامل xv6 ، هر فرآیند (process) را می‌توان به دو بخش اصلی تقسیم کرد:

حافظه کاربر :(User Memory) این فضای شخصی و اختصاصیِ خودِ فرآیند که کدها، داده‌ها، پشته (stack) و هیپ (heap) برنامه توش قرار می‌گیره.
پشته هسته :(Kernel Stack) یک پشته‌ی خصوصی که فقط وقتی فرآیند برای انجام کارهای حساس (مثل یک فراخوانی سیستمی یا یک وقفه) وارد حالت هسته (Kernel) میشه، ازش استفاده می‌کنه.
جدول صفحه :(Page Table) این مثل یه نقشه‌ی مترجم عمل می‌کنه. آدرس‌های مجازی که فرآیند باهاشون کار می‌کنه رو به آدرس‌های واقعی در حافظه فیزیکی کامپیوتر ترجمه می‌کنه. این کار باعث میشه حافظه‌ی هر فرآیند از بقیه جدا و ایزوله بمونه .وضعیت فرآیند :(Process State) یک برچسب که نشون میده فرآیند در حال حاضر در چه وضعیتیه؛ مثلاً در حال اجرا(running) ، آماده (ready)یا در حالت انتظار.(waiting)
شناسه فرآیند :(PID) یک شماره منحصر به فرد که مثل کد ملی برای هر فرآیند عمل می‌کنه و اون رو از بقیه متمایز می‌کنه.
بافت ذخیره‌شده :(Saved Context) یک عکس فوری از وضعیت رجیسترهای پردازنده .(CPU) این عکس به سیستم‌عامل اجازه میده که یک فرآیند رو متوقف کنه و بعداً دقیقاً از همون نقطه‌ای که مونده بود، دوباره به کار بندازتش

نحوه تخصیص پردازنده:
xv6از یک روش ساده و عادلانه به اسم »زمان‌بندی نوبت‌گردشی (Round-Robin) «استفاده می‌کنه. این روش مثل یه چرخ‌وفلک کار می‌کنه:
توقف :(Pause) سیستم‌عامل فرآیند فعلی رو متوقف می‌کنه و وضعیت پردازنده (CPU) رو در «بافت ذخیره‌شده» اون کپی می‌کنه.
انتخاب :(Choose) زمان‌بند (scheduler) از بین فرآیندهایی که در صف انتظار و آماده‌ی اجرا هستن، نفر بعدی رو انتخاب می‌کنه.
ادامه :(Resume) سیستم‌عامل «بافت ذخیره‌شده» فرآیند جدید رو روی پردازنده بارگذاری می‌کنه و اون فرآیند کارش رو دقیقاً از همونجایی که قبلاً متوقف شده بود، ادامه میده.
و این چرخه مدام تکرار میشه تا همه حس کنن که همزمان در حال اجرا هستن.

7) مفهوم file descriptorدر سیستم عاملهای مبتنی بر UNIXچیست؟ عملکرد pipeدر سیستم عامل xv6چگونه است و به طور معمول برای چه هدفی استفاده می شود؟

توصیف‌گر فایل (File Descriptor) یک عدد صحیح کوچیکه که مثل یک دستگیره یا شماره رُند برای یک فایلِ باز، یک پایپ (لوله ارتباطی) یا یک دستگاه عمل می‌کنه. این عدد یک رابط کاربری یکپارچه برای عملیات ورودی/خروجی فراهم می‌کنه و به برنامه‌ها اجازه میده تا با استفاده از یک سری دستور یکسان مثل read، writeو close، با انواع مختلف منابع به راحتی کار کنن.
پایپ یک کانال ارتباطی موقت و یک‌طرفه است که فرآیندها رو به هم وصل می‌کنه. به زبان ساده‌تر، مثل یک لوله عمل می‌کنه که خروجی یک فرآیند رو مستقیماً به ورودی یک فرآیند دیگه می‌فرسته.
این قابلیت در سیستم‌عامل xv6 به این شکل کار می‌کنه:
ایجاد :(Creation) سیستم‌عامل یک پایپ ایجاد می‌کنه که به طور خودکار دو تا سر داره: یک سر برای نوشتن داده (ورودی لوله) و یک سر برای خوندن داده (خروجی لوله).
ارتباط :(Communication) وقتی یک فرآیند، یک فرآیند فرزند می‌سازه، هر دوی اون‌ها (هم والد و هم فرزند) به این پایپ دسترسی پیدا می‌کنن. حالا یکی از اون‌ها می‌تونه با نوشتن در سرِ ورودی، داده ارسال کنه و اون یکی با خوندن از سرِ خروجی، داده رو دریافت کنه.
هماهنگی :(Synchronization) سیستم‌عامل این لوله رو هوشمندانه مدیریت می‌کنه. اگه فرآیندی بخواد از یک لوله‌ی خالی چیزی بخونه، سیستم‌عامل اون رو به خواب موقت می‌بره تا وقتی که داده‌ای وارد لوله بشه. برعکس، اگه فرآیندی بخواد در یک لوله‌ی پُر بنویسه، منتظر می‌مونه تا فضا خالی بشه.

8) فراخوانیهای سیستمی execو forkدر سیستمعامل xv6چه عملیاتی را انجام میدهند؟ از نظر طراحی، ادغام نکردن این دو چه مزیتی دارد؟

دستور forkیک فرآیند فرزند جدید می‌سازه که یک کپی دقیق و کامل از فرآیند والد خودشه. یعنی همه چیزش، از حافظه گرفته تا توصیف‌گرهای فایلش، کپی میشه. انگار از روی فرآیند اصلی یک کلون (clone) ساخته بشه.
دستور execاما کارش کاملاً متفاوته. اون محتوای فعلی فرآیند (کدها و حافظه) رو پاک می‌کنه و یک برنامه‌ی کاملاً جدید رو از روی یک فایل، جایگزینش می‌کنه .نکته‌ی کلیدی اینه که execبه توصیف‌گرهای فایل فرآیند اصلی دست نمی‌زنه و اون‌ها رو حفظ می‌کنه.
این جدا بودن یک مزیت فوق‌العاده هوشمندانه داره. این جدایی به فرآیند والد این فرصت رو میده که قبل از اجرای برنامه‌ی جدید، محیط فرآیند فرزند رو دستکاری و آماده‌سازی کنه.
این »مرحله آماده‌سازی «خیلی حیاتیه. معروف‌ترین کاربردش در شِل (Shell) هست. شِل اول با forkیک کپی از خودش می‌سازه، بعد در اون کپی، توصیف‌گرهای فایل رو تغییر میده (مثلاً خروجی

استاندارد رو به جای صفحه نمایش به یک فایل هدایت می‌کنه) و*تازه بعد از این آماده‌سازی*، دستور execرو صدا می‌زنه تا برنامه‌ی جدید رو اجرا کنه.
در نتیجه، برنامه‌ی جدید بدون اینکه خودش خبر داشته باشه یا کد اضافه‌ای لازم داشته باشه، در یک محیط از پیش تنظیم شده اجرا میشه.

9) دستور make -n را اجرا نمایید. کدام دستور، فایل نهایی هسته را می سازد؟

در پاسخ به این سوال که کدام دستور، فایل نهایی هسته را می‌سازد، باید به دستور ldتوجه کنیم:

```
ld -m elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o ...
```

بیایید این دستور را به زبان ساده‌تر توضیح دهیم:
فکر کنید در حال ساختن یک مدل پیچیده از قطعات لگو هستید.
قطعات لگو (فایل‌های .o): هر کدام از فایل‌هایی که به .o ختم می‌شوند (مثل bio.o, console.o, main.o)، یک بخش کامپایل‌شده و آماده از سیستم‌عامل هستند. این‌ها مثل قطعات جداگانه لگوی شما هستند که هرکدام وظیفه خاصی دارند.
دستورالعمل مونتاژ (ابزار ld): دستور ldهمان linkerیا «اتصال‌دهنده» است. این ابزار وظیفه دارد تمام این قطعات .oرا بردارد و آن‌ها را به درستی به هم متصل کند تا یک ساختار واحد و کامل به وجود بیاید.
نقشه راه (kernel.ld): فایل kernel.ldیک اسکریپت است که به linker می‌گوید این قطعات را چطور و با چه ترتیبی کنار هم بچیند تا همه چیز سر جای درستش قرار بگیرد.
محصول نهایی (-o kernel): گزینه -o kernel به linker می‌گوید که محصول نهایی و مونتاژ شده را با نام kernelذخیره کند.
بنابراین، این دستور ldتمام اجزای مختلف و از قبل کامپایل‌شده‌ی سیستم‌عامل xv6 را به هم پیوند می‌دهد تا فایل نهایی و قابل اجرای هسته (kernel) را بسازد؛ همان فایلی که در زمان بوت شدن سیستم، توسط بوت‌لودر در حافظه بارگذاری می‌شود.

10) در Makefile متغیرهایی به نام های UPROGSو ULIBتعریف شده است. کاربرد آنها چیست؟

در Makefileسیستم‌عامل xv6 ، دو متغیر کلیدی به نام‌های UPROGSو ULIBوجود دارند که فرآیند ساخت برنامه‌های کاربردی را مدیریت می‌کنند. بیایید ببینیم هرکدام چه کاری انجام می‌دهند:

UPROGS (User Programs)
این متغیر، لیست تمام برنامه‌های نهایی و قابل اجرایی است که کاربر می‌تواند از آن‌ها استفاده کند؛ برنامه‌هایی مثل) lsبرای نمایش لیست فایل‌ها(، cat(برای نمایش محتوای فایل) یا خودِ پوسته .(shell)وقتی سیستم‌عامل کامپایل می‌شود، هرکدام از موارد موجود در این لیست به یک فایل اجرایی جداگانه تبدیل می‌شوند که در نهایت در پوشه userقرار می‌گیرند.

ULIB (User Libraries)
این متغیر، لیست کتابخانه‌ها یا "جعبه ابزارهای" برنامه‌نویسی است. این کتابخانه‌ها شامل مجموعه‌ای از کدها و توابع آماده و پرکاربرد (مثلاً توابع مربوط به کار با رشته‌ها یا ورودی و خروجی) هستند که برنامه‌های مختلف (UPROGS) برای انجام کارهایشان به آن‌ها نیاز دارند. این کتابخانه‌ها به تنهایی قابل اجرا نیستند، بلکه کدهایشان به برنامه‌های اصلی "متصل" یا لینک می‌شوند.
نحوه کار با یکدیگر
روند کار به این صورت است:
وقتی دستور makeرا اجرا می‌کنید، ابتدا تمام کتابخانه‌هایی که در لیست ULIBقرار دارند، کامپایل شده و به فایل‌های آبجکت (.o) تبدیل می‌شوند. این‌ها در واقع "ابزارهای" ما هستند.
سپس، Makefileبه سراغ لیست UPROGSمی‌رود. هر برنامه را به همراه "ابزارهای" مورد نیازش از ULIBبرداشته و به هم پیوند می‌دهد تا فایل اجرایی نهایی و کامل آن برنامه ساخته شود.

11) اگر به فایلهای موجود در xv6دقت کنید، میبینید که فایلی مربوط به دستور ،cdبرخلاف دستوراتی مانند lsو ،catوجود ندارد و این دستور در سطح کاربر اجرا نمیشود. توضیح دهید که این دستور cdدر کجا اجرا میشود. به نظر شما دلیل این تفاوت میان دستور cdو دستورات دیگر مثل lsو catچیست؟

دستور cdبه عنوان یک دستور داخلی شل (built-in shell command) پیاده‌سازی شده و مستقیماً توسط خود شل اجرا می‌شود.
دلیل تفاوت :دستور cdباید دایرکتوری کاری فعلی (current working directory) خود شل را تغییر دهد .اگر cdبه عنوان یک برنامه مجزا اجرا می‌شد، شل ابتدا forkمی‌کرد و یک فرآیند فرزند برای اجرای cdمی‌ساخت .در این صورت، دستور cdفقط دایرکتوری کاری فرآیند فرزند را تغییر می‌داد و پس از پایان اجرای آن، دایرکتوری کاری شل (فرآیند والد) بدون تغییر باقی می‌ماند .اما دستوراتی مانند lsو catنیازی به تغییر وضعیت داخلی شل ندارند و می‌توانند به عنوان فرآیندهای جداگانه اجرا شوند.

12) در xv6در سکتور نخست دیسک قابل بوت، محتوای چه فایلی قرار دارد؟)راهنمایی: خروجی دستور make -n را بررسی نمایید.

وقتی سیستم روشن می‌شود، اولین جایی که کامپیوتر برای پیدا کردن دستورالعمل‌ها به آن نگاه می‌کند، سکتور اول دیسک است. محتوای این سکتور (که فقط ۵۱۲ بایت حجم دارد) توسط Makefile طی چند مرحله ساخته می‌شود:
کامپایل و لینک :اولین کاری که Makefileانجام می‌دهد این است که فایل‌های bootmain.c و bootasm.Sرا کامپایل می‌کند. سپس این دو فایل آبجکت را به هم متصل (لینک) می‌کند تا یک فایل واحد به اسم bootblock.oبسازد.
استخراج کد اصلی :در مرحله بعد، با استفاده از ابزار objcopy، فقط بخش کد اجرایی (.text) را از فایل bootblock.oجدا کرده و آن را در فایلی به نام bootblockمی‌ریزد. این کار تمام اطلاعات اضافی را حذف می‌کند و فقط کد خالص باقی می‌ماند.
اضافه کردن امضا :در نهایت، این فایل bootblockبه یک اسکریپت (sign.pl) داده می‌شود تا یک امضای بوت ۲ بایتی مخصوص به انتهای آن اضافه کند. این امضا به سیستم می‌فهماند که این دیسک واقعاً قابل بوت است.
نتیجه‌ی نهایی این است که تمام این مراحل، همان محتویاتی است که دقیقاً در سکتور اول (۵۱۲ بایت اول) دیسک بوت قرار می‌گیرد و اولین کدی است که پس از روشن شدن کامپیوتر اجرا می‌شود.

13) برنامه های کامپایل شده در قالب فایلهای دودویی 20نگهداری میشوند. فایل مربوط به بوت نیز دودویی است. نوع این فایل دودویی چیست؟ چرا از این نوع فایل دودویی استفاده شده است؟ تفاوت این نوع فایل دودویی با دیگر فایل های دودویی کد xv6چیست؟ این فایل را به زبان قابل فهم انسان)اسمبلی21) تبدیل نمایید.)راهنمایی: از ابزار objdumpاستفاده کنید. باید بخشی از آن مشابه فایل bootasm.Sباشد.

نوع فایل :فایل بوت یک فایل دودویی خام (raw binary) است که مستقیماً از کد ماشین 16 بیتی و 32 بیتی تشکیل شده است. این فایل فرمت خاصی مانند ELF ندارد.
دلیل استفاده :بایوس (BIOS) انتظار دارد که یک قطعه کد 512 بایتی قابل اجرا را در آدرس حافظه 0x7c00بارگذاری کند .یک فایل دودویی خام فاقد هرگونه هدر یا فراداده است و می‌تواند مستقیماً توسط پردازنده در حالت واقعی (real mode) اجرا شود.
تفاوت با دیگر فایل‌ها :دیگر فایل‌های دودویی در xv6 (مانند هسته و برنامه‌های سطح کاربر) از فرمت ELF (Executable and Linkable Format)استفاده می‌کنند .فایل‌های ELF دارای هدرهایی هستند که اطلاعاتی در مورد بخش‌های مختلف برنامه (مانند کد، داده و نقطه ورود) ارائه می‌دهند و برای بارگذاری توسط یک بارگذار پیچیده‌تر (مانند bootmainیا فراخوانی سیستمی (execطراحی شده‌اند.

14) علت استفاده از دستور objcopyدر حین اجرای عملیات makeچیست؟

objcopyیک ابزار همه‌کاره در سیستم‌عامل‌های شبه یونیکس هست که کارش کپی کردن و «ترجمه «فایل‌های آبجکت (object files) هست. فایل آبجکت در واقع خروجی خام کامپایلره که هنوز به طور کامل برای اجرا آماده نشده.
این ابزار می‌تونه یک فایل آبجکت رو بگیره و با فرمتی کاملاً متفاوت تحویل بده. فکر کن مثل یک مبدل فایل عمل می‌کنه که می‌تونه یک فایل Wordرو بگیره، تمام عکس‌ها و اطلاعات اضافیش رو حذف کنه و اون رو به صورت یک فایل متنی ساده (.txt) ذخیره کنه.
رفتار دقیق objcopyبا فلگ‌ها (flags) یا آپشن‌هایی که بهش می‌دیم کنترل میشه. چندتا از پرکاربردترین‌هاش این‌ها هستن:

- O binary-این دستور به objcopyمیگه که خروجی رو به صورت باینری خام تحویل بده. وقتی این کار رو می‌کنه، در واقع یک کپی خالص از محتویات حافظه‌ی فایل ورودی رو می‌سازه. تمام اطلاعات اضافی مثل نمادها (symbols) و اطلاعات جابجایی حذف میشن و فقط کد خالص باقی می‌مونه.
- S-یا :strip-all--این دستور تمام نمادها (symbols) رو از فایل خروجی حذف می‌کنه. نمادها مثل اسم متغیرها و توابع هستن که بیشتر برای دیباگ کردن به کار میان و برای اجرای برنامه ضروری نیستن. حذف کردنشون باعث میشه حجم فایل نهایی کمتر بشه. دو بخش اصلی که حذف میشن جدول نمادها و رکوردهای جابجایی هستن.
- text. j-این آپشن میگه که فقط بخش text.فایل آبجکت رو نگه دار. بخش text.همون قسمتیه که کدهای اجرایی و دستورالعمل‌های اصلی برنامه توش قرار دارن.

در xv6 از این ابزار برای چندتا کار کلیدی استفاده میشه:
ساختن) bootblockبلوک بوت :(این اولین کدیه که موقع روشن شدن سیستم اجرا میشه. دستور زیر bootblock.o bootblock -j .text -S -O binary $ (OBJCOPY) :فایل bootblock.orو میگیره، تمام نمادهاش رو حذف می‌کنه(-S) ، فقط بخش کد اجرایی (.text) رو نگه می‌داره (-j) و در نهایت اون رو به فرمت باینری خام (-O binary) در فایلی به نام bootblockذخیره می‌کنه. بعداً یک اسکریپت دیگه چک می‌کنه که حجم این فایل بیشتر از ۵۱۰ بایت نباشه و در آخر دو بایت جادویی 0x55)و (0xaaکه به عنوان امضای بوت شناخته میشن، به انتهای اون اضافه می‌کنه.
ایجاد :entryother این هم یک قطعه کد دیگه‌ست که در مراحل اولیه‌ی بوت شدن سیستم در یک آدرس خاص از حافظه بارگذاری میشه و با objcopyاز بخش text.فایل bootblockother.o استخراج میشه.
ساختن :initcode این اولین کدیه که وقتی یک فرآیند جدید در سیستم ایجاد میشه، در محیط کاربری اجرا میشه. این فایل هم یک نسخه‌ی باینری خام از فایل initcode.outهست.

در نهایت، با لینک کردن و کنار هم قرار دادن فایل entry.o، بقیه‌ی فایل‌های o.*، و همچنین فایل‌های initcode و entryother که با objcopyآماده شدن، هسته‌ی (Kernel) سیستم‌عامل xv6ساخته میشه و آماده‌ی کار میشه.

15) در فایلهای موجود در ،xv6مشاهده میشود که بوت سیستم توسط فایل هایbootasm.S و bootmain.cصورت میگیرد. چرا تنها از کد Cاستفاده نشده است؟

جواب کوتاه اینه که هرکدوم برای کار خاصی ساخته شدن .زبان اسمبلی (Assembly) مثل آچار مخصوص مکانیکه که برای کارهای خیلی دقیق و حساس روی موتور (سخت‌افزار) لازمه، در حالی که زبان C مثل یک جعبه ابزار عمومیه که برای ساخت بقیه‌ی قسمت‌های ماشین (سیستم‌عامل) عالیه.
مراحل اولیه‌ی بوت شدن سیستم‌عامل، یک سری کارهای خیلی پایه‌ای و حساس داره که باید مستقیماً با سخت‌افزار کامپیوتر انجام بشه. کارهایی مثل:
آماده کردن پشته(stack)
تنظیم کردن رجیسترهای اصلی پردازنده
و مهم‌تر از همه، عوض کردن حالت کاری پردازنده
این کارها به قدری دقیق و سطح‌پایین هستن که زبان‌های سطح بالایی مثل C نمی‌تونن از پسشون بربیان و فقط از عهده‌ی زبان اسمبلی برمیاد.
مهم‌ترین وظیفه‌ی اسمبلی: تغییر حالت پردازنده
وقتی کامپیوتر روشن میشه و بایوس (BIOS) کد بوت رو اجرا می‌کنه، پردازنده‌ی x86 در یک حالت قدیمی و محدود به اسم »حالت واقعی (Real Mode) «قرار داره. در این حالت:
پردازنده مثل یک پردازنده‌ی ۱۶ بیتی قدیمی عمل می‌کنه.
فقط به ۱ مگابایت حافظه دسترسی داره.
آدرس‌دهی حافظه فقط به صورت فیزیکی و مستقیمه (خبری از آدرس‌دهی مجازی نیست).
برای اینکه بتونیم از تمام قدرت پردازنده استفاده کنیم (یعنی پردازنده‌ی ۳۲ بیتی و دسترسی به ۴ گیگابایت حافظه)، باید اون رو به »حالت محافظت‌شده (Protected Mode) «ببریم.
این تغییر حالت، یک کار فوق‌العاده حساسه که فقط و فقط با دستورات زبان اسمبلی ممکنه. این کار با تغییر دادن بیت اول یک رجیستر کنترلی مخصوص در پردازنده انجام میشه. به محض اینکه این کار انجام شد و سیستم وارد حالت محافظت‌شده شد، زبان اسمبلی کارش رو به زبان C تحویل میده تا بقیه‌ی قسمت‌های سیستم‌عامل رو بسازه.

16) یک ثبات عام منظوره ،22یک ثبات قطعه ،23یک ثبات وضعیت 24و یک ثبات کنترلی 25در معماری86xرا نام برده و وظیفه هر یک را به طور مختصر توضیح دهید

ثبات عام منظوره:(General Purpose Register)

- **EAX (Accumulator):برای عملیات محاسباتی، ورودی/خروجی و ذخیره مقدار بازگشتی توابع استفاده می‌شود.**

ثبات قطعه:(Segment Register)

- **CS (Code Segment):آدرس پایه قطعه کد فعلی را نگه می‌دارد .پردازنده دستورات را از این قطعه واکشی می‌کند.**

ثبات وضعیت:(Status Register)

- **EFLAGS:شامل بیت‌های وضعیتی است که نتیجه عملیات محاسباتی (مانند بیت سرریز یا صفر) و همچنین بیت‌های کنترلی) مانند بیت فعال/غیرفعال بودن وقفه‌ها یا (IF را نشان می‌دهد.**

ثبات کنترلی:(Control Register)

- **CR0:شامل بیت‌های کنترلی سیستم است .برای مثال، بیت PE (Protection Enable) برای فعال کردن مد محافظت شده و بیت PG (Paging Enable) برای فعال کردن صفحه‌بندی استفاده می‌شود**

17) پردازنده‌های x86دارای مدهای مختلفی هستند. هنگام بوت، این پردازنده‌ها در مد حقیقی27 قرار داده میشوند؛ مدی که سیستم عامل اماسداس28 (MS DOS) در آن اجرا می شد. چرا؟ یک نقص اصلی این مد را بیان نمایید. آیا در پردازنده‌های دیگر مانند ARMیا RISC-Vنیز مدها به همین شکل هستند یا خیر؟ توضیح دهید

پردازنده‌های x86 برای حفظ سازگاری با نسخه‌های قدیمی‌تر (backward compatibility) در مد حقیقی (real mode) شروع به کار می‌کنند .این مد، عملکرد پردازنده Intel 8088 را شبیه‌سازی می‌کند و به نرم‌افزارهای قدیمی) مانند (MS-DOS اجازه می‌دهد تا روی پردازنده‌های جدیدتر نیز اجرا شوند .بایوس (BIOS) نیز در همین مد اجرا می‌شود و وظایف اولیه را انجام می‌دهد.
نقص اصلی مد حقیقی :مهم‌ترین نقص مد حقیقی، عدم وجود حفاظت حافظه (memory protection)و محدودیت آدرس‌دهی به تنها ۱ مگابایت حافظه است .در این مد، هر برنامه‌ای می‌تواند به هر بخشی از حافظه دسترسی داشته باشد که این موضوع برای سیستم‌عامل‌های چندوظیفه‌ای مدرن کاملاً ناامن و ناکافی است.
مقایسه با ARM و :RISC-V پردازنده‌های ARM و RISC-V نیز دارای مدهای اجرایی متفاوتی هستند، اما ساختار آن‌ها با x86 متفاوت است. آن‌ها مد حقیقی به سبک x86 برای سازگاری با پردازنده‌های قدیمی ندارند.

- ARM: دارای چندین مد اجرایی (مانند User, Supervisor, IRQ) است که سطوح دسترسی متفاوتی را فراهم می‌کنند. پردازنده معمولاً در یک مد با امتیاز بالا) مانند (Supervisor شروع به کار می‌کند و سپس سیستم عامل می‌تواند آن را به مد کاربر منتقل کند.
- RISC-V: دارای مدهای امتیازی (Privilege Modes) مانند(User (U-mode ، Supervisor (S-mode)و Machine (M-mode) است. پردازنده در بالاترین سطح دسترسی (M-mode) بوت می‌شود و سپس کنترل را به یک ناظر) مانند سیستم عامل در-S mode)و در نهایت به برنامه‌های کاربردی) در (U-mode واگذار می‌کند.

در هر دو معماری، برخلاف x86 ، نیازی به گذر از یک مد 16 بیتی قدیمی و فعال کردن دستی ویژگی‌های مدرن نیست.

18) یکی دیگر در از مدهای مهم، مد حفاظتشده 29میباشد. وظیفهی اصلی این مود چیست؟ ها پردازنده در چه زمانی در این مود قرار میگیرند؟

وظیفه اصلی مد محافظت شده :(Protected Mode) وظیفه اصلی این مد، فراهم کردن حفاظت حافظه و پشتیبانی از چندوظیفگی است .این مد به سیستم عامل اجازه می‌دهد تا با استفاده از مکانیزم‌هایی مانند قطعه‌بندی (segmentation) و صفحه‌بندی(paging) ، فضای آدرس مجازی ایزوله برای هر فرآیند ایجاد کند و از دسترسی غیرمجاز یک فرآیند به حافظه فرآیند دیگر یا هسته جلوگیری نماید.
زمان قرارگیری در این مد :پردازنده پس از بوت شدن در مد حقیقی قرار دارد .بوت لودر سیستم عامل، پس از انجام تنظیمات اولیه) مانند بارگذاری(GDT ، با تنظیم بیت PE در ثبات CR0، پردازنده را به صورت نرم‌افزاری به مد محافظت شده منتقل می‌کند.

19) کد ،bootmain.cهسته را با شروع از یک سکتور پس از سکتور بوت، خوانده و در آدرس 0x100000قرار میدهد32 .علت انتخاب این آدرس چیست؟ چرا این آدرس از 0شروع نشده است؟

علت انتخاب آدرس 0x100000(۱ مگابایت) این است که ناحیه حافظه پایین‌تر از آن برای مقاصد خاصی رزرو شده است.ناحیه زیر 640 کیلوبایت :(Base Memory) توسط بایوس و بردارهای وقفه استفاده می‌شود.ناحیه بین 640 کیلوبایت تا ۱ مگابایت :(I/O Space) برای حافظه دستگاه‌های ورودی/خروجی (مانند حافظه کارت گرافیک) و حافظه فقط خواندنی بایوس (BIOS ROM) رزرو شده است .بنابراین، آدرس 0x100000اولین آدرس امن و در دسترس برای بارگذاری هسته پس از این نواحی رزرو شده است.

20) برای مشاهده Breakpointها از چه دستوری استفاده میشود؟

برای مشاهده breakponts ها از دستور info breakpoints یا info break استفاده می کنیم.

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x801043b0 in main at main.c:20
        breakpoint already hit 1 time
2       breakpoint     keep y   0x801043b0 in main at main.c:20
```

جداولی که نمایش داده می‌شوند شامل شماره، نوع، وضعیت فعال/غیرفعال، آدرس و مکان کد مربوط به هر breakpoint هستند.

21) برای حذف یک Breakpointاز چه دستوری و چگونه استفاده میشود؟

برای حذف یک breakpoint در GDB، از دستور delete استفاده می‌کنیم.
اگر شماره breakpoint را بدانیم:

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x801043b0 in main at main.c:20
        breakpoint already hit 1 time
2       breakpoint     keep y   0x801043b0 in main at main.c:20
(gdb) delete 2
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x801043b0 in main at main.c:20
        breakpoint already hit 1 time
```

حذف همه breakpoints:

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x801043b0 in main at main.c:20
        breakpoint already hit 1 time
3       breakpoint     keep y   0x801043b0 in main at main.c:20
4       breakpoint     keep y   0x801043b0 in main at main.c:20
5       breakpoint     keep y   0x801043b0 in main at main.c:20
(gdb) delete
Delete all breakpoints, watchpoints, tracepoints, and catchpoints? (y or n) y
(gdb) info break
No breakpoints, watchpoints, tracepoints, or catchpoints.
(gdb)
```

22

دستور btفهرستی از stack frames را نشان می‌دهد، یعنی مسیر فراخوانی توابع تا نقطه‌ای که برنامه متوقف شده است.

```
(gdb) bt
#0  consoleintr (getc=0x80103a10 <kbdgetc>) at console.c:598
#1  0x80103b50 in kbdintr () at kbd.c:63
#2  0x80106e65 in trap (tf=0x801180d8 <stack+3912>) at trap.c:67
#3  0x80106bcf in alltraps () at trapasm.S:20
#4  0x801180d8 in stack ()
#5  0x80114464 in cpus ()
#6  0x80114460 in ?? ()
#7  0x8010438f in mpmain () at main.c:58
#8  0x801044dc in main () at main.c:37
(gdb) CFLAGS
```

یعنی مسیر اجرا این‌طوری بوده:

main → mpmain → alltraps → trap → kbdintr → consoleintr

23) دو تفاوت دستورهای xو printرا توضیح دهید. چگونه میتوان محتوای یک ثبات خاص را چاپ کرد؟

دستور print در GDB برای نمایش مقدار متغیرها یا ساختارها است — نه فقط آدرس یا حافظه خام.

```
(gdb) print input
$1 = {buf = "aftab", '\000' <repeats 122 times>, r = 0, w = 0, e = 5, mouse_pos = 5}
```

یعنی:

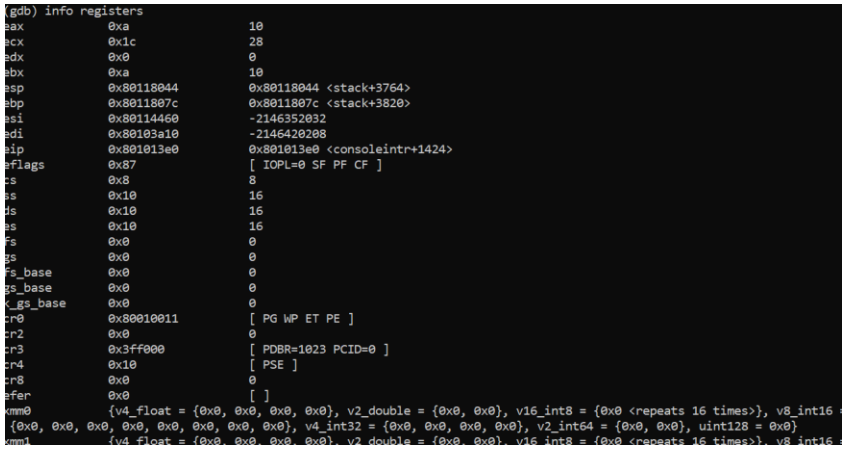
- **input یک struct است.**
- **فیلد buf یک آرایه ۱۲۸بایتی است که در آن "aftab" ذخیره شده و بقیه پر از '0\' است.**
- **r, w, e, mouse_pos اعضای دیگر ساختار هستند که مقدارشان به‌ترتیب 0, 0, 5, 5 است.**

print مقدار معنی‌دار متغیر را (مثل ساختار بالا) چاپ می‌کند، در حالی که x فقط محتوای خام حافظه را نشان می‌دهد.

برای نمایش محتوای یک ثبات CPU (مثل eax یا ebp یا rip در GDB، دو روش وجود دارد:

دستور info registers

نمایش تمام ثبات‌ها.



می‌توان یک ثبات خاص را چاپ کرد.

```
(gdb) print $eax
$2 = 10
```

24برای نمایش وضعیت ثبات ها از چه دستوری استفاده می شود؟ برای متغیرهای محلی چطور؟ نتیجه این دستور را در گزارش کار خود بیاورید. همچنین در گزارش خود توضیح دهید که در معماری x86رجیستر های edi وesiنشانگر چه چیزی هستند؟

برای دیدن محتوای همه‌ی ثبات‌های CPU در GDB، از دستور زیر استفاده می‌شود:
info registers
این دستور مقدار همه‌ی رجیسترهای اصلی (مثل eax, ebx, ecx, edx, esi, edi, ebp, esp, eip,...) را نشان می‌دهد.
برای دیدن تمام متغیرهای محلی ((local variables در تابع فعلی از دستور زیر استفاده می‌شود:
info locals
این دستور مقدار تمام متغیرهای محلی در فریم فعلی (تابع فعلی) را چاپ می‌کند.

```
(gdb) info locals
pos = <optimized out>
c = 10
doprocdump = <optimized out>
```

نقش ثبات‌های edi و esi در معماری 86x

ESI (Source Index):
اشاره‌گر به آدرس مبدأ در عملیات حافظه یا رشته‌ای مثلاً movs

EDI (Destination Index):
اشاره‌گر به آدرس مقصد در همان عملیات‌ها

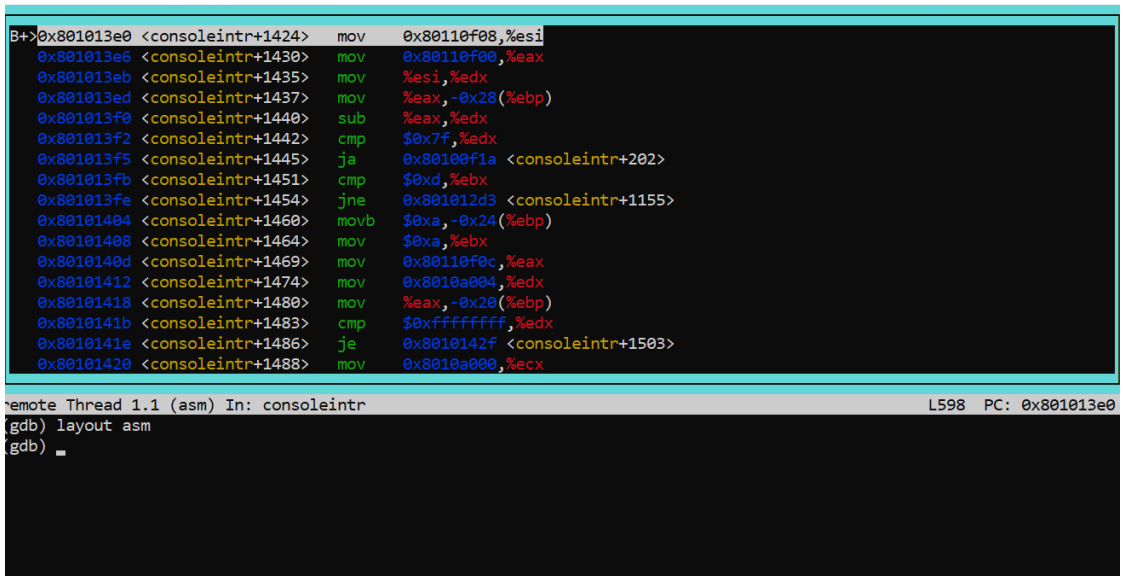
25) به کمک استفاده از GDB، درباره ساختار struct input موارد زیر را توضیح دهید:
● توضیح کلی این struct و متغیر های درونی آن و نقش آنها.
● نحوه و زمان تغییر مقدار متغیر های درونی (برای مثال، input.e در چه حالتی تغییر میکند و چه مقداری میگیرد

در سیستم‌عامل 6xv، ساختار input برای مدیریت ورودی‌های کیبورد و در برخی نسخه‌ها، موقعیت موس استفاده می‌شود. این ساختار شامل یک بافر کاراکتری به نام buf و چندین اندیس کنترلی است. بافر buf معمولاً ۱۲۸ بایت است و وظیفه ذخیره موقت کاراکترهای تایپ‌شده توسط کاربر را دارد تا زمانی که برنامه آن‌ها را خوانده یا پردازش کند. اندیس‌های داخلی شامل r، w، e و mouse_pos هستند که وضعیت بافر را مدیریت می‌کنند. اندیس r موقعیت خواندن بعدی از بافر را نشان می‌دهد، w موقعیت نوشتن داده‌ها به خروجی، e نشان‌دهنده انتهای داده‌های وارد شده و mouse_pos موقعیت موس در بافر است.
با استفاده از GDB و دستور print input می‌توان وضعیت فعلی این ساختار را مشاهده کرد. برای مثال، اجرای دستور:

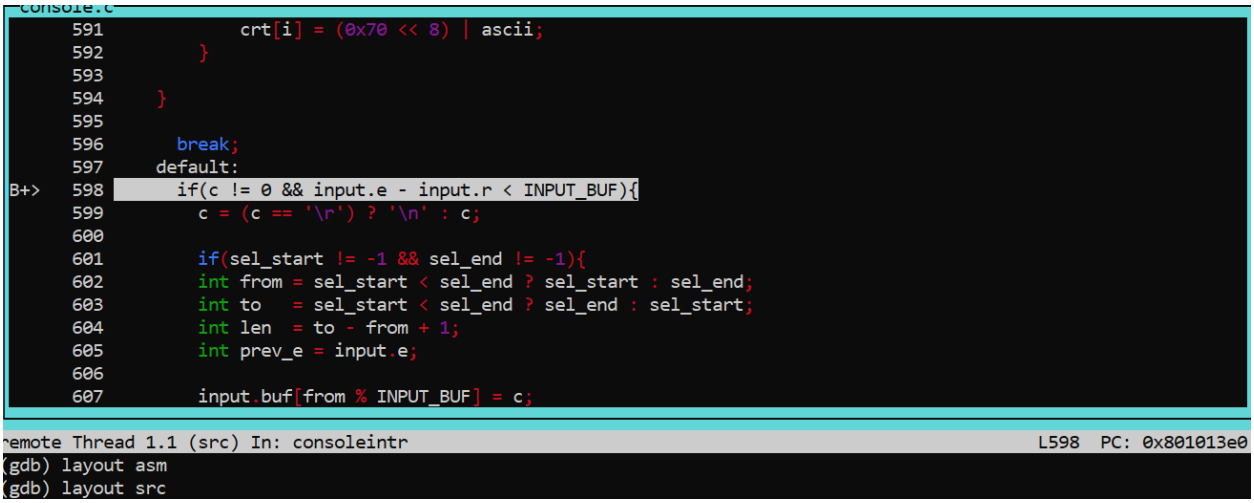
```
(gdb) print input
$1 = {buf = "aftab", '\000' <repeats 122 times>, r = 0, w = 0, e = 5, mouse_pos = 5}
```

نشان می‌دهد که کاربر پنج کاراکتر "aftab" وارد کرده است و بافر تا اندیس ۵ پر شده (e = 5). مقادیر r و w هنوز صفر هستند، به این معنی که داده‌ها هنوز توسط برنامه خوانده یا چاپ نشده‌اند. همچنین mouse_pos برابر ۵ است که مکان‌نمای ورودی را در انتهای رشته نشان می‌دهد.
مقدار متغیرهای داخلی این ساختار در زمان‌های مختلف تغییر می‌کند. هر بار که کاربر یک کاراکتر جدید تایپ می‌کند، این کاراکتر به buf[e % INPUT_BUF] نوشته شده و e افزایش می‌یابد. هنگامی که داده‌ها از بافر به خروجی فرستاده می‌شوند، مقدار w افزایش پیدا می‌کند و هنگام خواندن داده توسط برنامه، اندیس r افزایش می‌یابد.

(26



نمایش دستورها و آدرس‌های حافظه، بررسی دقیق اجرای برنامه



نمایش دستورها و آدرس‌های حافظه، بررسی دقیق اجرای برنامه

27) برای جابهجایی میان توابع زنجیره فراخوانی جاری)نقطه توقف(از چه دستوراتی استفاده می شود؟

برای جابه‌جایی میان توابع در زنجیره‌ی فراخوانی در GDB از Call Stack استفاده می‌کنیم. دستور bt یا backtrace کل زنجیره‌ی توابعی که برنامه را به نقطه توقف رسانده نشان می‌دهد. با دستور up می‌توان به تابع فراخواننده (فریم بالاتر) رفت و با دستور down به تابع فراخوانده شده (فریم پایین‌تر) برگشت. همچنین با frame n می‌توان مستقیماً به فریم شماره n پرش کرد. این دستورات امکان بررسی متغیرهای محلی و آرگومان‌های هر تابع را فراهم می‌کنند.