

# آزمایشگاه سیستم عامل

پروژه شماره 1

علی پادیاو – کسری حاجی حیدری – اولدوز نیساری

بهار 1402

Repository Link: <https://github.com/alumpish/OS-Lab-Projects>

Latest Commit Hash: 7443b8b570c3b7e97ddc94e3788ecec62186ed85

## آشنایی با سیستم عامل xv6

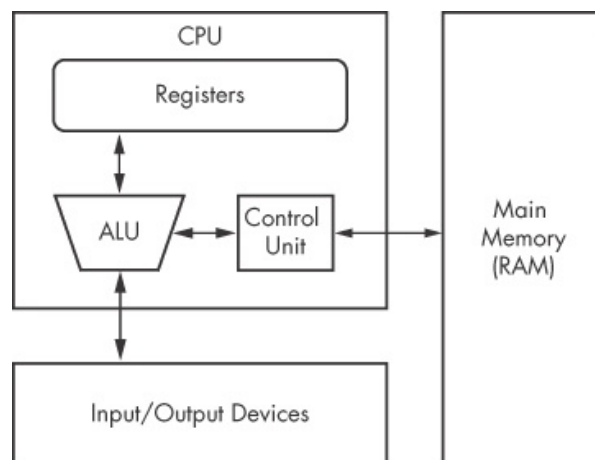
-1

xv6 در حقیقت یک نوع پیاده سازی مجدد از ورژن 6 ام unix است که با استفاده از زبان C و بر اساس پردازنده های مبتنی بر x86 نوشته شده است.

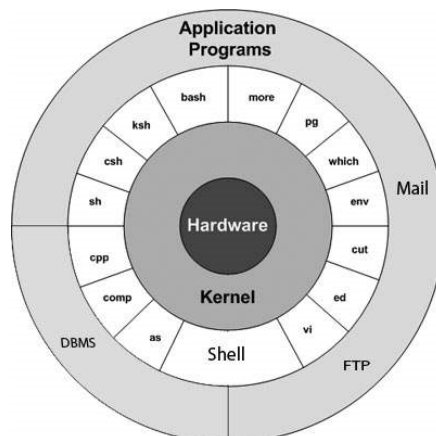
بر دفاع از نظر خودمان که گفتیم بر مبنای x86 است می توانیم به وجود فایل هایی با پسوند 86 اشاره کنیم مثل فایل های : x86.h در توضیحات آن هم نوشته شده است بر مبنای instruction های 86 است استناد کنیم.

```
// Routines to let C code use special x86 instructions.
```

نکات تکمیلی : خود x86 بر مبنای CISC نوشته شده است و یک مدل ساده از آن به صورت زیر است:



همچنین نمایی کلی از ساختار unix هم به صورت زیر است:



با دقت در فایل proc.h که در فایل های xv6 است می توانیم چند بخش مهم process را به صورت زیر بگوییم :

```
struct proc {
    uint sz;                // Size of process memory (bytes)
    pde_t* pgdir;           // Page table
    char *kstack;           // Bottom of kernel stack for this process
    enum procstate state;   // Process state
    int pid;                // Process ID
    struct proc *parent;    // Parent process
    struct trapframe *tf;   // Trap frame for current syscall
    struct context *context; // switch() here to run process
    void *chan;             // If non-zero, sleeping on chan
    int killed;             // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;       // Current directory
    char name[16];          // Process name (debugging)
};
```

#### (1) Kernel stack:

این بخش در استراکت با kstack مشخص شده است که از جنس پوینتر است. (پوینتری به kernel stack)

در زمان function call وضعیت رجیستر ها در استک کاربر ذخیره می شوند. به صورت مشابه وقتی پردازش به سیستم عامل می پرد تا کد کرنل را اجرا کند ، CPU context در kernel stack ذخیره می شود.

#### (2) List of open files:

نام این بخش در استراکت ofile[NOFILE] است که در حقیقت آرایه ای از پوینتر ها است به open file ها.

هر وقت کاربر یک فایل را باز می کند یک entry جدید به آرایه اضافه می شود که ایندکس آن entry به عنوان file describer کاربر برگردانده می شود. (زمانی که کاربر می خواهد فایلی را بخواند یا بنویسد از این file describer ها برای مراجعه استفاده می کنند)

هم چنین در این آرایه 3 خانه اول برای error , output , standard input کنار گذاشته شده اند.

### Page table (3):

نام این بخش در استراکت pgdir است که پوینتری به page table است. هر دستور یا داده در memory image (الگو برنامه نویسی که داده ها در آن ذخیره شده اند). یک آدرس واقعی و یک آدرس مجازی دارند ، در page table ، mapping بین این دو آدرس نگهداری می شود.

هم چنین بر اساس کتاب rev 11 میتوانیم بگوییم:

هر پردازش یک مموری دارد که شامل دستورات ، داده ها و یک استک است. دستورات محاسبات برنامه را انجام میدهد. داده ها بخش هایی هستند که محاسبات روی آن ها انجام می شوند و استک بخشی است فرآیند فراخوانی های برنامه در آن مدیریت می شود. به این حافظه user-space می گویند.

هر پردازش وضعیت پردازش ای هم دارد که برای kernel مخفی است.

برای مدیریت پردازش های مختلف روش مورد استفاده time-share است. در این حالت به صورت نامحسوس سیستم عامل ، پردازنده های مختلف را بین پردازش های مختلف که منتظر اجرا هستند به اشتراک می گذارد. زمانی که یک پردازش اجرا نمیشود تمام رجیستر های آن ذخیره میشوند تا در زمان بازگشت مجدد به آن پردازش بتوانند بازیابی شوند.

Kernel به هر کدام از پردازش ها یک id هم میدهد.

-3

همان طور که در سوال قبل اندکی توضیح داده شد ، جدولی در سیستم عامل وجود دارد که file describer مانند یک index برای آن عمل می کند. عملاً هر پردازش یک فضای خصوصی برای ذخیره file descriptor هایش دارد که از 0 شروع می شود و همان طور که گفته شده خانه اول (0) برای ورودی (stdin) ، خانه دوم (1) برای خروجی (stdout) و خانه سوم (2) برای ارور (stdout) اختصاص داده شده است.

File descriptor در حقیقت یک عدد صحیح است که به یک شی که kernel قابلیت مدیریت آن را دارد و پردازش ممکن است از آن بخواند یا بنویسد اشاره می کند.

هر پردازش زمانی که یک فایل ، یک directory یا یک device را باز می کند یک file descriptor به دست می آورد. کاری که file descriptor می کند این است که یک interface انتزاعی ایجاد می کند که تفاوت بین این ها دیده نشود و همگی مثل جریان هایی از بیت ها باشند.

عملکرد pipe به این صورت است که دو file descriptor را در نظر می گیرد و آن ها به هم وصل می کند. یکی را برای نوشتن و دیگری را برای خواندن. نوشتن روی یکی از file descriptor ها ، امکان خواندن را از روی دیگر file descriptor فراهم می کند. در حقیقت pipe ها امکان ارتباط و تعامل را برای file descriptor ها فراهم می کند.

-4

Fork برای هر پردازش یک پردازش دقیقا با همان محتوای حافظه به نام پردازش فرزند می سازد. به پردازش اولیه اصطلاحا پردازش پدر می گویند. Fork برای هر یک از پردازش های پدر و فرزند مقدار های متفاوتی را برمی گرداند برای پدر pid را برمی گرداند و برای فرزند 0.

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

در حقیقت اتفاقی که در سیستم عامل می افتد تا حدی در تکه کد بالا توضیح داده شده است. زمانی که pid صفر است ، exit system call یعنی فرآیند آزاد سازی منابع مثل حافظه و باز کردن فایل ها اتفاق می افتد. زمانی که pid از 0 بزرگتر باشد wait system call اتفاق می افتد. wait system call ، pid فرزند خارج شده پردازش را برمی گرداند. اگر هیچ یک از فرزندان پردازش خارج نشده بودند صبر می کند تا یکی از آن ها خارج شوند.

با وجود این که پردازش پدر و فرزند محتوای حافظه یکسانی دارند اما آن ها با مموری ها و رجیستر های متفاوتی اجرا می شوند. برای مثال تغییر دادن یک متغیر در حافظه یک پردازش روی دیگری تاثیر نمی گذارد.

Exec اما متفاوت است. exec پردازش در حال اجرا را با پردازش ای که آدرس و آرگومان آن داده می شود جایگزین می کند اما file table اولیه را هم حفظ میکند.

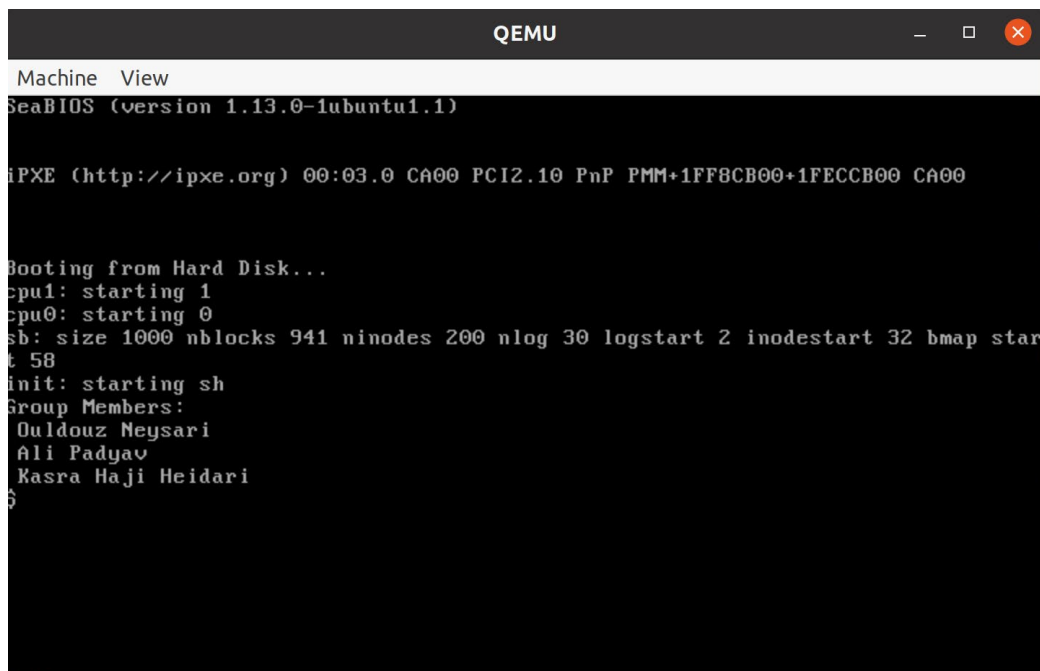
دلیل ادغام نکردن این دو در بخش پیاده سازی I/O است. در حقیقت ادغام نکردن از ساختن پردازش های بی مصرف و جایگزین شدن سریع آن ها توسط exec جلوگیری می کند.

در حالت عادی در زمانی که تابع fork و exec پشت سر هم اجرا می شوند. اگر این دو ادغام شوند علاوه بر پردازش های اضافه و میزان حافظه زیادی که اشغال می شود مدیریت آرگومان های توابع هم دشوار می شود.

## اضافه کردن یک متن به Boot message :

با توجه به فرمت printf در فایل user.h خط زیر را در فایل init.c اضافه می کنیم و در ادامه نتیجه:

```
printf(1, "init: starting sh\n");  
printf(1, "Group Members: \n Ouldouz Neysari \n Ali Padyav \n Kasra Haji Heidari \n");
```



```
QEMU  
Machine View  
SeaBIOS (version 1.13.0-1ubuntu1.1)  
  
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00  
  
Booting from Hard Disk...  
cpu1: starting 1  
cpu0: starting 0  
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star  
t 58  
init: starting sh  
Group Members:  
Ouldouz Neysari  
Ali Padyav  
Kasra Haji Heidari  
$
```

## اضافه کردن چند قابلیت به کنسول xv6 :

ابتدا دو قابلیت `move_backward_cursor` و `move_forward_cursor` را اضافه و از آن‌ها برای قابلیت‌های جدید استفاده می‌کنیم.

1- رفتن به ابتدای خط با دستور `shift +`

تابع زیر را به `console.c` اضافه کرده و در `switch case` حالت `( '[' )` این تابع را فراخوانی می‌کنیم.

```
1 void
2 move_to_start(){
3     while(input.e != input.w &&
4           input.buf[(input.e-1) % INPUT_BUF] != '\n'){
5         input.e--;
6         move_backward_cursor();
7     }
8 }
```

2- رفتن به انتهای خط با دستور `shift +`

مثل حالت قبل تابع مورد نظر را اضافه و در `case` مناسب آن را فراخوانی می‌کنیم.

```
1 void
2 move_to_end(){
3     if (input.e == input.w &&
4         input.buf[input.e % INPUT_BUF] != '\0') {
5         input.e++;
6         move_forward_cursor();
7     }
8     int temp = input.buf[(input.e-1) % INPUT_BUF] != '\0';
9     while(input.buf[(input.e-1) % INPUT_BUF] != '\0'){
10        input.e++;
11        move_forward_cursor();
12    }
13    if (temp) {
14        input.e--;
15        move_backward_cursor();
16    }
17 }
```

3- پاک کردن کلمه قبل از کرسر با دستور `ctrl + w`:

درست مثل حالت 2 قبل تابع `jump_left_cursor` را به `console.c` اضافه و در کیس `C('W')` تابع را فراخوانی میکنیم.

```
1  int isDelimiter(char c)
2  {
3      if (c == ' ' || c == '.' || c == ',' || c == ':' || c == ';')
4          return 1;
5      return 0;
6  }
7
8  void jump_left_cursor()
9  {
10     int count = input.e;
11
12     for (int i = 0; i < count; i++){
13         if (isDelimiter(input.buf[input.e]))
14             return;
15         input.l--;
16         input.e--;
17         consputc(BACKSPACE);
18     }
19 }
```

### اجرا و پیاده سازی یک برنامه سطح کاربر:

برنامه `mmm.c` نوشته و در فلدر اصلی `xv6` قرار داده شد. همینطور آن را به متغیر `UPGROS` در `makefile` اضافه کردیم تا به عنوان برنامه سطح کاربر شناخته شود.

```
$ mmm 8 2 8 4 2 3
$ cat mmm_result.txt
4 3 2
$
```



## کامپایل سیستم عامل xv6:

-8

متغیر UPROGS:

در حقیقت مخفف user programs است که یک لیست از برنامه های کاربر دارد که در هنگام کامپایل شدن xv6 این برنامه ها نیز کامپایل می شوند و هر یک قابل اجرا توسط سیستم عامل می شوند.

(در زمان اجرای دستور هایی مثل cat, mkdir مورد استفاده قرار می گیرند)

متغیر ULIB:

در حقیقت مخفف user libraries است که شامل لیست کتابخانه هایی است که در vx6 استفاده شده است. شامل موارد زیر می شود :

ulib.o

usys.o

printf.o

umalloc.o

## مراحل بوت سیستم عامل xv6 :

### اجرای بوت لودر:

-11

اگر در فایل های مربوط به boot در make file دقت کنیم ، متوجه می شویم که دو فایل Bootasm.s و bootmain.c در مرحله بوت کامپایل می شوند. (قابل مشاهده است که فایل هایی با پسوند.o برای آن ها تشکیل می شود)

Bootasm به زبان اسمبلی است و bootmain.c به زبان سی. تفاوت این فایل ها با بقیه فایل ها این است که در اولین سکتور هارد دیسک قرار می گیرند. علت استفاده از اسمبلی و C این است که این بخش سیستم عامل با سخت افزار رابطه نزدیکی دارد، و کنترل بهتری به سخت افزار برایمان فراهم می کنند.

-12

این دستور همان طور که از اسمش مشخص است کاری که انجام میدهد به این صورت است که محتویات یک فایل object را در یک فایل object دیگر کپی می کند. در حقیقت کاری که انجام میدهد به این صورت است که فرمت فایل را در هنگام کپی تغییر میدهد. این که objcpy دقیقاً چه کاری انجام دهد توسط کاربر و کامندی که در ترمینال می نویسد مشخص می شود اما به طور کلی objcpy فایل های موقتی تشکیل می دهد تا بتواند ترجمه هایش را انجام دهد.

در make file از objcpy استفاده می شود تا raw binary file هایی ساخته شود تا پردازنده بدون نیاز به سیستم عامل بتواند آن ها را اجرا کند.

-13

اجرای کد های زبان اسمبلی معمولاً سریع تر و کم حجم تر است. هم چنین گاهی نیاز است که یک دسترسی سطح سیستم داشته باشیم در آن صورت کد زبان C به تنهایی کافی نیست.

در حقیقت کاری که انجام می شود به این صورت است که bootasm.s پردازنده را به حالت حفاظت شده 32 بیتی می برد و در آنجا فایل bootmain.c صدا زده می شود. این انتقال نمونه ای از کار هایی است که برای اجرای آن ها به دسترسی سطح سیستم نیاز است.

-14

ثبات عام منظوره : در vx6 تعداد 8 عدد ثبات عام منظوره وجود دارد. نام همگی این ثبات ها به دلیل 32 بیتی بودن با حرف e که مخفف extended است شروع می شود.

تمام 8 عدد ثبات عام منظوره :

Eip, eax, ebx, ecx , edx , esi , ebp , esp

وظیفه ثبات عام منظوره نگهداری حاصل عملیات ریاضی ، پوینتر ها و برخی داده ها است.

ثبات قطعه : در xv6 تعداد 6 عدد ثبات قطعه وجود دارد. به طور کلی وظیفه این نوع ثبات نگهداری آدرس استک ، دیتا و کد است.

ثبات های مشهور این دسته CS, DS , SS هستند که به ترتیب وظایق گفته شده یعنی نگهداری پوینتر به استک ، پوینتر به دیتا و پوینتر به کد را نگهداری می کنند.

ثبات وضعیت : وظیفه این نوع ثبات ها نگهداری وضعیت پردازنده است. EFLAGS یک نمونه ثبات وضعیت است که در آن flag های مختلف نگهداری می شوند ( flag های zero, sign, carry ) ثبات کنترلی : وظیفه این ثبات ها کنترل cpu و دستگاه های دیجیتا دیگر است. cr0, cr2, cr3, cr4 نمونه هایی از این ثبات ها هستند که وظایفشان به ترتیب تغییر مدل آدرس دهی، کنترل interrupt ، paging و هم پردازنده ها است.

-15

در ابتدا توضیحاتی درباره real mode ارائه می دهیم:

این مود بر اساس پردازنده های 8086 و 8088 است. این مود در حقیقت مود دستور های 16 بیتی است به همین دلیل تمام برنامه ها در real mode باید شامل دستور های 16 بیتی باشند. در این مود هیچ protection وجود ندارد به همین دلیل این مود نمی تواند از overwrite شدن برنامه ها جلوگیری کند، به همین دلیل این مود multi-tasking نیست.

استفاده از real mode در پردازنده ها در حقیقت یک دلیل تاریخی دارد. تمام پردازنده های x86 در حالتی که وانمود می کنند پردازنده x8086 هستند شروع به کار می کنند. ( پردازنده ای 16 بیتی که توانایی دسترسی فقط به یک بیت از مموری را دارد.) بعد از آن مسئولیت BIOS است که مشخص کند که پردازنده واقعا از چه نوعی است.

این الگو در حقیقت یک الگو رایج است که کامپیوتر ها با یک تکه کد ساده شروع به کار می کند و سپس با سنجیدن شرایط و ارتباط برقرار کردن با بقیه سیستم ها ، به روی مود های پیشرفته تر شیفت میدهند. نقص اصلی real mode: از جمله نقص های این سیستم این است که هیچ مکانیزم امنیتی یا حفاظتی در این مود وجود ندارد از برنامه های پر اشتباه و مخرب جلوگیری کند.

-16

مموری در real mode یک توالی از خطی از بایت هاست که می تواند آزادانه آدرس دهی شوند با هر آدرس 20 بیتی که از 16 بیت آدرس segment و 4 بیت آدرس offset تشکیل شده باشد. برنامه می تواند به هر نقطه از مموری دسترسی پیدا کند و در آن بخواند و بنویسد فارغ از این که در چه بخشی از حافظه قرار دارد.

نهایتاً آدرس ما به صورت زیر محاسبه می شود :

$$\text{PhysicalAddress} = \text{Segment} * 16 + \text{Offset}$$

-18

کد ورود به هسته (معادل entry.s) در معماری x86 لینوکس برای 32 بیت و 64 بیت در لینک زیر قرار دارد.

<https://github.com/torvalds/linux/tree/master/arch/x86/entry>

## اجرای هسته xv6

-19

این آدرس نباید مجازی باشد زیرا برای دسترسی به آن باید آدرس مجازی اش را به آدرس فیزیکی تبدیل کنیم و اینکار با استفاده از این جدول امکان پذیر است اما از آنجایی که آدرس فیزیکی این جدول را نداریم وارد یک حلقه بی نهایت برای پیدا کردن آدرس فیزیکی میشویم و در نتیجه نمیتوانیم به این جدول دسترسی پیدا کنیم. در نهایت میتوان نتیجه گرفت که آدرس جدول نگاشت باید به صورت فیزیکی (در رجیستر cr3) ذخیره شود.

-22

segment، DS و SS های مختلفی هستند که لینوکس از descriptor های یکسانی برای آنها استفاده می کند. اما ما نیاز داریم که descriptor ها را برای user mode و kernel mode تفکیک کنیم. در اینجا از SEG\_USER استفاده می شود که دستورات user-side از kernel-side تمایز یابند.

این struct وضعیت هر پردازش را ذخیره میکند که در فایل h.proc تعریف شده و دارای متغیرهای زیر است:

```
1 struct proc{
2     uint sz;                // Size of process memory (bytes)
3     pde_t* pgdir;           // Page table (پردازش به page table پوینتر به)
4     char *kstack;           // Bottom of kernel stack for this process (از kernel استک بخشی از)
5     enum procstate state;    // Process state (allocated, ready to run, running, waiting for I/O, or exiting)
6     int pid;                // Process ID (یک عدد یکتا که به پردازش اختصاص داده میشود)
7     struct proc *parent;     // Parent process (پوینتر به پردازش سازنده پردازش کنونی)
8     struct trapframe *tf;    // Trap frame for current syscall (systemcall برای ذخیره وضعیت اجرای برنامه در هنگام اجرای یک)
9     struct context *context; // switch() here to run process (contextswitching مقادیر رجیسترهای مورد نیاز برای)
10    void *chan;              // If non-zero, sleeping on chan (میکنند wait اگر صفر نباشد یعنی پردازش خوابیده یا)
11    int killed;              // If non-zero, have been killed
12    struct file *ofile[NOFILE]; // Open files (فایل های باز شده توسط پردازش)
13    struct inode *cwd;        // Current directory
14    char name[16];           // Process name (used for debugging)
15 }
```

معادل این struct در لینوکس در لینک زیر و در استراکت task\_struct قرار دارد:

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

(Each element in the task list is a process descriptor of the type struct task\_struct, which is defined in <linux/sched. h>. The process descriptor contains all the information about a specific process.)

هسته اول فرآیند بوت را انجام می دهد و توسط کد entry.s وارد تابع main می شود. توابع مورد نیاز برای آماده سازی سیستم در این تابع فراخوانده شده اند و توسط این هسته اجرا می شوند.

```

1  main(void)
2    kinit1(end, P2V(4*1024*1024)); // phys page allocator
3    kvmalloc();           // kernel page table
4    mpinit();             // detect other processors
5    lapicinit();          // interrupt controller
6    seginit();            // segment descriptors
7    picinit();            // disable pic
8    ioapicinit();         // another interrupt controller
9    consoleinit();        // console hardware
10   uartinit();           // serial port
11   pinit();              // process table
12   tvinit();            // trap vectors
13   binit();             // buffer cache
14   fileinit();          // file table
15   ideinit();           // disk
16   startothers();       // start other processors
17   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
18   userinit();          // first user process
19   mpmain();            // finish this processor's setup
20

```

هسته های دیگر از طریق کد entryother.s وارد تابع mpenter می شوند. با توجه به اینکه این تابع در تابع main نیز فراخوانی میشود می توان گفت این 4 تابع بین تمامی هسته ها مشترک خواهند بود.

```

1  // Other CPUs jump here from entryother.S.
2  static void
3  mpenter(void)
4    switchkvm();
5    seginit();
6    lapicinit();
7    mpmain();
8

```

از موارد اختصاصی هسته اول می‌توان به تابع `kvmalloc` اشاره کرد که (با توجه به کد `kpgdir=setupkvm()`) یک `page table` ایجاد می‌کند که این مورد توسط هسته اول انجام می‌پذیرد یا تابع `startothers` که فقط پردازنده اول نیاز است بقیه پردازنده ها را `start` کند.

از توابع مشترک می‌توان به `mpmain` اشاره کرد زیرا همه پردازنده ها باید کار خود را شروع کنند و آماده اجرای برنامه ها شوند که این مورد توسط این تابع انجام می‌پذیرد یا تابع `switchkvm` زیرا همه پردازنده ها باید آدرس `page table` که توسط پردازنده اول ایجاد شده را در رجیستر خود ذخیره کنند در نتیجه این تابع بین همه آن ها مشترک است.

زمانبند توسط تابع `scheduler` انجام می‌پذیرد در تابع `mpmain` صدا زده می‌شود که این تابع بین تمامی هسته ها مشترک است.

## اشکال زدایی:

-1

برای مشاهده breakpoint ها از دستور `Info break points` استفاده می‌کنیم.

```
(gdb) break cat.c:12
Breakpoint 1 at 0x97: file cat.c, line 12.
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint      keep y   0x00000097 in cat at cat.c:12
(gdb) break cat.c:10
Note: breakpoint 1 also set at pc 0x97.
Breakpoint 2 at 0x97: file cat.c, line 10.
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint      keep y   0x00000097 in cat at cat.c:12
2        breakpoint      keep y   0x00000097 in cat at cat.c:10
(gdb) break cat.c:18
Breakpoint 3 at 0xd3: file cat.c, line 18.
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint      keep y   0x00000097 in cat at cat.c:12
2        breakpoint      keep y   0x00000097 in cat at cat.c:10
3        breakpoint      keep y   0x000000d3 in cat at cat.c:18
```

-2

برای حذف break point ها از دو روش می توان استفاده کرد:

روش اول: استفاده از Del breakpoint\_number

روش دوم: استفاده از Clear file\_name:line

```
(gdb) break cat.c:14
Breakpoint 4 at 0xdc: file cat.c, line 14.
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint      keep y 0x00000097 in cat at cat.c:12
2        breakpoint      keep y 0x00000097 in cat at cat.c:10
3        breakpoint      keep y 0x000000d3 in cat at cat.c:18
4        breakpoint      keep y 0x000000dc in cat at cat.c:14
(gdb) del 3
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint      keep y 0x00000097 in cat at cat.c:12
2        breakpoint      keep y 0x00000097 in cat at cat.c:10
4        breakpoint      keep y 0x000000dc in cat at cat.c:14
(gdb) clear cat.c:10

(gdb) info breakpoints
Deleted breakpoint 2 Num      Type             Disp Enb Address      What
1        breakpoint      keep y 0x00000097 in cat at cat.c:12
4        breakpoint      keep y 0x000000dc in cat at cat.c:14
```

-3

با اجرای bt سلسله از توابعی که تاکنون فراخوانی شده اند و در استک اضافه شده اند نمایش داده می شود. در حقیقت bt مخفف backtrace است.

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, wc (fd=0, name=0x96d "") at wc.c:15
15      while((n = read(fd, buf, sizeof(buf))) > 0){
(gdb) bt
#0  wc (fd=0, name=0x96d "") at wc.c:15
#1  0x0000008d in main (argc=1, argv=0x2ff4) at wc.c:41
(gdb)
```



همان طور که در تصاویر ها مشاهده می کنیم ، نحوه دریافت ورودی این دو دستور با هم فرق می کند. دستور x آدرس می گیرد اما دستور print، expression می گیرد. همچنین نحوه نمایش اطلاعات آن ها و موارد مختلفی که می توانند نمایش دهند با یکدیگر نیز متفاوت است.

```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
    t(binary), f(float), a(address), i(instruction), c(char), s(string)
    and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format. If a negative number is specified, memory is
examined backward from the address.

Defaults for format and size letters are those previously used.
Default count is 1. Default address is following last thing printed
with this command or "print".
(gdb) help print
Print value of expression EXP.
Usage: print [[OPTION]... --] [/FMT] [EXP]
```

```
(gdb) help print
Print value of expression EXP.
Usage: print [[OPTION]... --] [/FMT] [EXP]

Options:
  -address [on|off]
    Set printing of addresses.

  -array [on|off]
    Set pretty formatting of arrays.

  -array-indexes [on|off]
    Set printing of array indexes.

  -elements NUMBER|unlimited
    Set limit on string chars or array elements to print.
    "unlimited" causes there to be no limit.

  -max-depth NUMBER|unlimited
    Set maximum print depth for nested structures, unions and arrays.
    When structures, unions, or arrays are nested beyond this depth then they
    will be replaced with either '{...}' or '(...)' depending on the language.
    Use "unlimited" to print the complete structure.

  -null-stop [on|off]
    Set printing of char arrays to stop at first null char.

  -object [on|off]
    Set printing of C++ virtual function tables.

  -pretty [on|off]
    Set pretty formatting of structures.

  -raw-values [on|off]
    Set whether to print values in raw form.
```

هم چنین با استفاده از دستور register\_name registers info می‌توان یک ثابت خاص را نمایش داد:

```
(gdb) info registers eax
eax                0x1
(gdb) |
```

-5

برای نمایش وضعیت ثابت‌ها از دستور registers info استفاده می‌شود:

```
(gdb) info registers
eax                0x1                1
ecx                0x2fec             12268
edx                0xbfac             49068
ebx                0x2ff8             12280
esp                0x2f98             0x2f98
ebp                0x2f98             0x2f98
esi                0x1                1
edi                0x0                0
eip                0xa7               0xa7 <wc+7>
eflags             0x246              [ IOPL=0 IF ZF PF ]
cs                 0x1b               27
ss                 0x23               35
ds                 0x23               35
es                 0x23               35
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                0x4244c8d          69487757
cr3                0xdfc6000          [ PDBR=0 PCID=0 ]
cr4                0x10               [ PSE ]
cr8                0x0                0
efer               0x0                [ ]
```

برای وضعیت متغیرهای محلی از دستور locals info استفاده می‌کنند.

```
(gdb) info locals
i = <optimized out>
n = <optimized out>
l = 0
w = 0
c = 0
inword = 0
```

هم چنین اگر بخواهیم وضعیت تمام متغیرها را ببینیم یعنی حتی غیر محلی ها از دستور info variables استفاده میکنیم.

```
(gdb) info variables
All defined variables:

File umalloc.c:
21:     static Header base;
22:     static Header *freep

File wc.c:
5:     char buf[512];

Non-debugging symbols:
0x000008f8  digits
0x00000be0  __bss_start
0x00000be0  _edata
0x00000e00  _end
```

رجیستر های edi , esi هر دو رجیستر های عام منظوره هستند. اما معمولا esi برای نگهداری آدرس source و edi برای نگهداری آدرس destination در زمان عملیات کپی استفاده میشوند.

-6

با استفاده از دستور ptype input اطلاعات استراکت input را از gdb نمایش می‌دهیم.

```
(gdb) ptype input
type = struct {
    char buf[128];
    uint r;
    uint w;
    uint e;
}
```

متغیرهای استراکت :

- (1) Buf: آرایه ای از کاراکتر هاست که ظرفیت آن 128 کاراکتر است. ورودی که در ترمینال می‌نویسیم (به صورت رشته) در این آرایه ذخیره می‌شود.
- (2) R: ایندکسی از بافر را نشان میدهد. در حقیقت آخرین جایی است که خوانده ایم.
- (3) W: ایندکسی از بافر که محل شروع نوشتن در خط ورودی buf است.
- (4) E: ایندکسی از بافر که محل کنونی cursor را نگه می‌دارد.

تمام زمان هایی که مکان cursor تغییر می کند مثل زمان هایی که در ترمینال چیزی تایپ می کنیم input.e تغییر می کند.

همچنین زمانی که چیزی می نویسم input.w به input.r می رسد.

-7

خروجی دستور ها:

layout asm

در این حالت می توان برنامه را به صورت کد اسمبلی دید.

```
>0xffff0 add %al, (%eax)
0xffff2 add %al, (%eax)
0xffff4 add %al, (%eax)
0xffff6 add %al, (%eax)
0xffff8 add %al, (%eax)
0xffffa add %al, (%eax)
0xffffc add %al, (%eax)
0xffffe add %al, (%eax)
0x10000 add %al, (%eax)
0x10002 add %al, (%eax)
0x10004 add %al, (%eax)
0x10006 add %al, (%eax)
0x10008 add %al, (%eax)
0x1000a add %al, (%eax)
0x1000c add %al, (%eax)
0x1000e add %al, (%eax)
0x10010 add %al, (%eax)
0x10012 add %al, (%eax)
0x10014 add %al, (%eax)
0x10016 add %al, (%eax)
0x10018 add %al, (%eax)
0x1001a add %al, (%eax)
0x1001c add %al, (%eax)
0x1001e add %al, (%eax)
0x10020 add %al, (%eax)
0x10022 add %al, (%eax)
0x10024 add %al, (%eax)
0x10026 add %al, (%eax)
0x10028 add %al, (%eax)

remote Thread 1.1 In: L?? PC: 0xffff0
(gdb) layout asm
```

## :layout src

برنامه را در حالت کد سورسش نشان می دهد.

```
WC.C
1      #include "types.h"
2      #include "stat.h"
3      #include "user.h"
4
5      char buf[512];
6
7      void
8      wc(int fd, char *name)
9      {
10         int i, n;
11         int l, w, c, inword;
12
13         l = w = c = 0;
14         inword = 0;
15         while((n = read(fd, buf, sizeof(buf))) > 0){
16             for(i=0; i<n; i++){
17                 c++;
18                 if(buf[i] == '\n')
19                     l++;
20                 if(strchr(" \r\t\n\v", buf[i]))
21                     inword = 0;
22                 else if(!inword){
23                     w++;
24                     inword = 1;
25                 }
26             }
27         }
28         if(n < 0){
29             printf(1, "wc: read error\n");
30         }
31     }
```

remote Thread 1.1 In: main L37 PC: 0x0  
(gdb) layout src

-8

برای جا به جایی میان توابع از دستورات زیر استفاده می کنند:

- (1) دستور up n: با این دستور می توان به تعداد n، frame به سمت بالای استک رفت. (اگر n را مقدار دهی نکنیم با n=1 دستور را اجرا می کند)
- (2) دستور down n: با این دستور می توان به تعداد n، frame به سمت پایین استک رفت. (اگر n را مقدار دهی نکنیم با n=1 دستور را اجرا می کند)
- (3) دستور frame[frame\_selection\_spec]:

به جای frame\_selection\_spec مواردی مختلفی می توانیم قرار دهیم:

Function\_name: فریم استک متعلق به آن تابع را پیدا می کند و نمایش میدهد. اگر چند فریم متعلق به تابع باشد، داخلی ترین آن ها انتخاب می شود.

Level: منظور شماره فریمی از استک است که میخواهیم به آن برویم.

Stack\_address: آدرس فریمی از استک است که میخواهیم به آن برویم.

نمونه ای از استفاده از این دستورات در زیر نمایش داده شده است :

```
(gdb) where
#0  cat (fd=0) at cat.c:12
#1  0x00000007 in main (argc=1, argv=0x2ff4) at cat.c:30
(gdb) up
#1  0x00000007 in main (argc=1, argv=0x2ff4) at cat.c:30
30      cat(0);
(gdb) down
#0  cat (fd=0) at cat.c:12
12      while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) frame 0
#0  cat (fd=0) at cat.c:12
12      while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) frame 1
#1  0x00000007 in main (argc=1, argv=0x2ff4) at cat.c:30
30      cat(0);
```

## بخش امتیازی (لینوکس)

با دستور `uname -a` ورژن کرنل لینوکس نشان داده می‌شود.

```
root@syzkaller:~# uname -a
Linux syzkaller 6.2.2 #2 SMP PREEMPT_DYNAMIC Sat Mar  4 21:11:54 +0330 2023 x86_64 GNU/Linux
```

برای نمایش اسم اعضای گروه یک فایل c و یک makefile نوشتیم:

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3
4 MODULE_LICENSE("GPL");
5
6 int init_module(void)
7 {
8     printk(KERN_INFO "Group Members:\n1- Ouldouz Neysari\n2- Ali Padyav\n3- Kasra Haji Heydari\n");
9     return 0;
10 }
11
12 void cleanup_module(void) {}
13
```

```
1 obj-m += hello_world.o
2
3 all:
4     make -C /lib/modules/6.2.2/build M=$(PWD) modules
5
```

با دستور `make` فایل `hello_world.ko` ساخته می‌شود.

سپس با دستور `sudo insmod hello_world.ko` و `dmesg` خروجی را مشاهده می‌کنیم:

```
525 [ 6.971145] Group Members:
526 |         |         |         | 1- Ouldouz Neysari
527 |         |         |         | 2- Ali Padyav
528 |         |         |         | 3- Kasra Haji Heydari
529
```