

گزارش کار پروژه اول آزمایشگاه سیستم عامل  
گروه ۲۷

ارشیا عطایی نائینی | ۸۱۰۱۰۰۲۵۲

فاطمه کرمی محمدی | ۸۱۰۱۰۰۲۵۶

امیر پارسا موبد | ۸۱۰۱۰۰۲۷۱

آدرس مخزن گیت‌هاب: <https://github.com/phatemek/OS-lab-project-1>  
شناسه آخرین کامیت: b56aec749d574f4eb78ab89f1bcc9a1d80c14210

## آشنایی با سیستم عامل xv6

۱. معماری سیستم عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟  
در کد این سیستم عامل در فایل x86.h از دستورات پردازنده های x86 استفاده شده است و همچنین در فایل mmu.h از معماری x86 استفاده شده است. از این می‌توان نتیجه گرفت که این سیستم عامل بر اساس سیستم عامل v6 Unix (ورژن ششم سیستم عامل Unix) نوشته شده است و معماری نزدیک به آن دارد. سیستم عامل xv6 برای پردازنده x86 و سیستم های بر پایه RISC-V طراحی شده است.

۲. یک پردازنده در سیستم عامل xv6 از چه بخش هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازنده های مختلف اختصاص می‌دهد؟  
یک process در سیستم عامل xv6 از دو بخش زیر تشکیل شده است:  
- user-space memory (حافظه فضای کاربری) شامل دستورات، استک ها و داده ها  
- process state که به طور خصوصی برای هسته قابل خواندن است  
پخش زمان بین پردازنده ها در سیستم عامل xv6 به این صورت است که پردازنده ها را بین پردازنده های موجود پخش می‌کند و زمان هر پردازنده را به بخش های مختلف برای پردازش پردازنده ها تقسیم می‌کند. هنگامی که یک پردازنده در حال خارج شدن از یک پردازنده است xv6 رجیسترهای مربوط به آن پردازنده را ذخیره کرده و در اجرای بعدی آن پردازنده آن رجیستر ها را restore می‌کند. همچنین برای درستی انجام کار به هر پردازنده یک pid یکتا اختصاص می‌دهد.

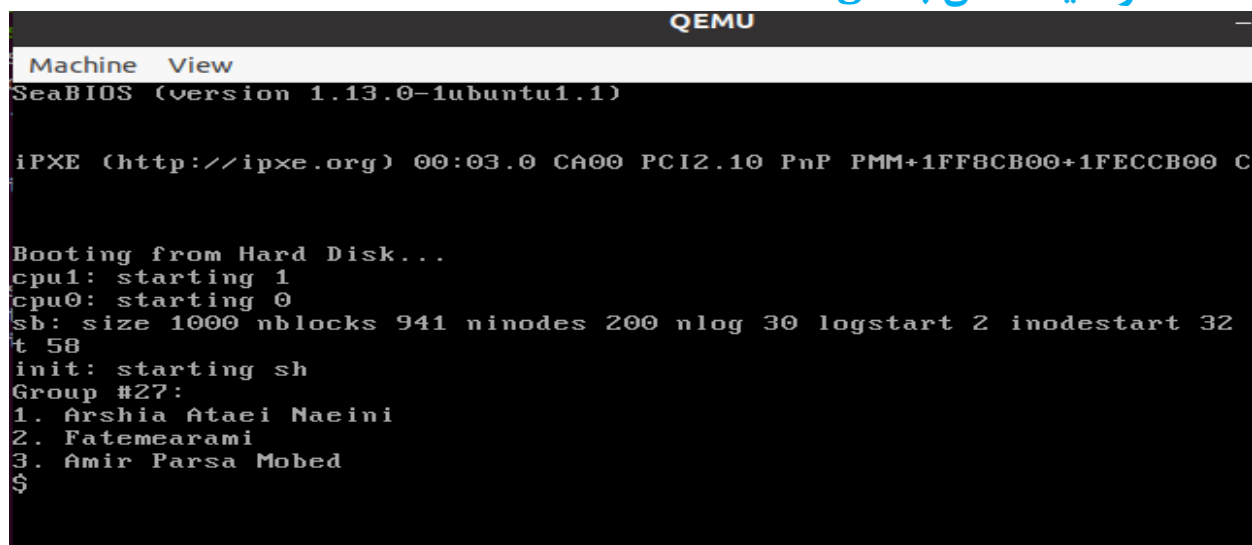
۴. فراخوانی های سیستمی exec و fork چه عملی انجام می‌دهند؟ از نظر طراحی ادغام نکردن این دو چه مزیتی دارد؟

تابع fork با ساختن یک پردازنده فرزند از پردازنده ای که این تابع را صدا زده است کار می‌کند. این تابع حافظه و دستورات پردازنده ای را که آن را صدا زده است را در حافظه پردازنده فرزند هم نگه می‌دارد. در واقع این یکسان سازی حافظه دو پردازنده با کپی کردن حافظه پردازنده پدر در حافظه فرزند کار می‌کند و پس از ساخت پردازنده فرزند این دو حافظه از هم مجزا هستند و با تغییر یکی از آنها، دیگری تغییر نخواهد کرد. تابع fork پس از اتمام کارش مقدار pid پردازنده فرزند را بازمی‌گرداند.

تابع exec حافظه پردازنده فراخوانی شده را با یک حافظه جدید حاوی فایل ELF جایگزین می‌کند اما جدول فایل ها را حفظ می‌کند. این عملکرد به شل اجازه می‌دهد که با استفاده از fork و باز کردن مجدد شماره های توصیف فایل مورد نظر، سپس exec کردن برنامه‌ی جدید، اجرای redirection ورودی و خروجی را پیاده‌سازی کند.

این کار هماهنگی کاری است که کد shell xv6 برای I/O redirection انجام می‌دهد. به طور کلی، در ابتدا shell یک fork از خود ایجاد می‌کند و سپس exec را صدا می‌زند تا برنامه‌ی جدید را لود کند. مزیت جدا کردن این دو فرآیند این است که shell می‌تواند پس از fork کردن فرزند، از باز کردن بستن و کپی کردن فایل‌ها در داخل فرزند برای تغییر دادن شماره‌های توصیف فایل‌های ورودی و خروجی استفاده کند و سپس با اجرای exec تغییرات اعمال شود. همچنین، این روش ایجاد هیچ تغییری در برنامه‌هایی که قرار است با exec اجرا شوند، نیاز ندارد. اگر فراخوانی‌های سیستمی exec و fork ادغام شده بودند، برای انجام I/O redirection نیاز به یک طرح پیچیده‌تر بود و شل یا خود برنامه باید شماره‌های توصیف فایل‌های ورودی و خروجی را تنظیم می‌کردند.

## اضافه کردن یک متن به Boot Message

A screenshot of a QEMU terminal window. The title bar says "QEMU". Below it, there's a "Machine View" tab. The terminal output shows the SeaBIOS boot process: "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 t 58", "init: starting sh", "Group #27:", "1. Arshia Ataei Naeini", "2. Fatemearami", "3. Amir Parsa Mobed", and "\$".

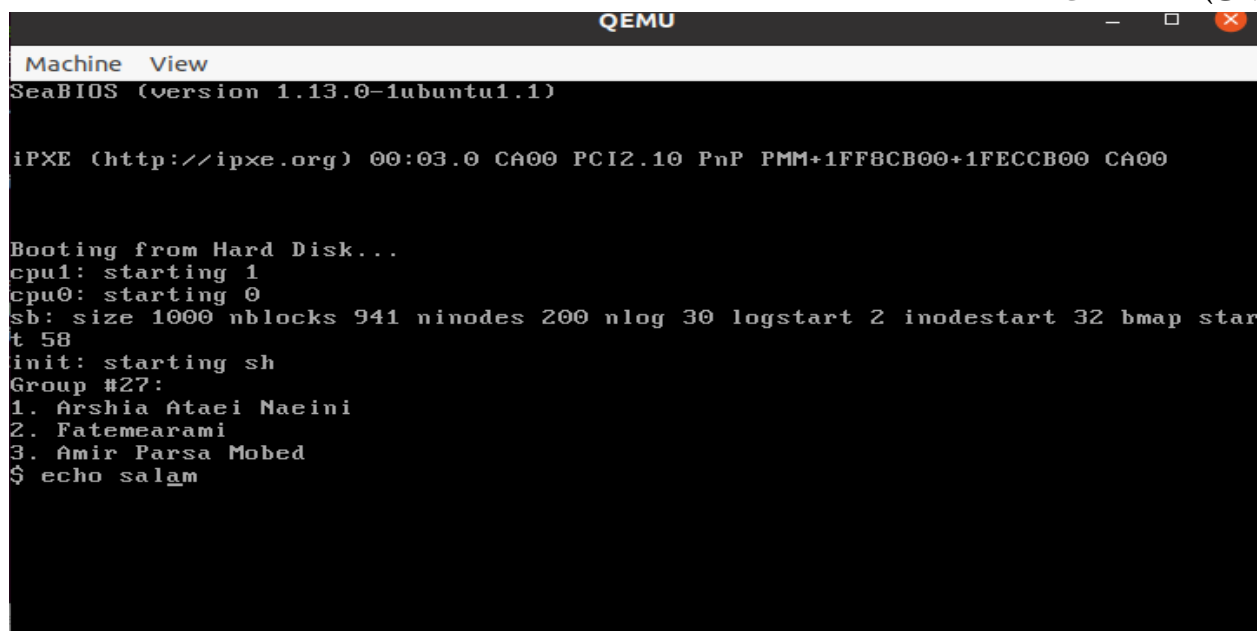
```
QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

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 t 58
init: starting sh
Group #27:
1. Arshia Ataei Naeini
2. Fatemearami
3. Amir Parsa Mobed
$
```

## اضافه کردن چند قابلیت به کنسول xv6

الف) CTRL+B :

A screenshot of a QEMU terminal window, similar to the previous one. The title bar says "QEMU". Below it, there's a "Machine View" tab. The terminal output is the same as the previous screenshot, but with an additional line at the bottom: "\$ echo salam".

```
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 #27:
1. Arshia Ataei Naeini
2. Fatemearami
3. Amir Parsa Mobed
$ echo salam
```

```
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 #27:
1. Arshia Ataei Naeini
2. Fatemearami
3. Amir Parsa Mobed
$ echo saiam
```

: CTRL+F (ب)

```
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 #27:
1. Arshia Ataei Naeini
2. Fatemearami
3. Amir Parsa Mobed
$ echo saitmsa
```

```
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 #27:
1. Arshia Ataei Naeini
2. Fatemearami
3. Amir Parsa Mobed
$ echo saitmsaw
```

: CTRL+L (ج)

```
QEMU
Machine View
$ _

$ echo salam
salam
$ echo malas
malas
$ echo sal_
```

: ↓ ↑ (د)

```
$ echo salam
salam
$ echo malas
malas
$ echo malas_

$ echo salam
salam
$ echo malas
malas
$ echo salam
```

```
$ echo salam
salam
$ echo malas
malas
$ echo malas
```

## اجرا و پیاده‌سازی یک برنامه سطح کاربر

```
$ strdiff apple banana
$ cat strdiff_result.txt
100011
$ _
```

۸. در **makefile** متغیرهایی به نام های **UPROGS** و **ULIB** تعریف شده‌است. کاربرد آنها چیست؟  
 متغیر **UPROGS** مخفف **user programs** است و در آلیستی از برنامه های کاربر وجود دارد که برای کامپایل کردن **xv6** نیاز است. اسم هر فایلی که در این لیست قرار دارد نشاندهنده یک **target** و یک متغیر **ULIB** است. **ULIB** که مخفف **user libraries** است شامل تعدادی کتابخانه زبان **C** است که در کدهای **xv6** از آنها استفاده شده است. این کتابخانه ها شامل توابعی مانند **umalloc**, **strcpy**, **strcmp** و... هستند.

۱۱. برنامه‌های کامپایل شده در قالب فایل‌های دودویی نگهداری می‌شوند. فایل‌های مربوط به بوت نیز دودویی است. نوع این فایل دودویی چیست؟ تفاوت این نوع فایل دودویی با دیگر فایل‌های دودویی کد **xv6** چیست؟ چرا از این نوع فایل دودویی استفاده شده است؟ این فایل را به زبان قابل فهم انسان (**Assembly**) تبدیل نمایید.

در پروژه **Makefile**، دو فایل **bootasm.S** (اسمبلی) و **bootmain.c** (به زبان **C**) به صورت جدا کامپایل می‌شوند و از آنها فایل‌های با پسوند **O** به وجود می‌آید. سپس، این دو فایل با یکدیگر توسط دستور **LD** لینک می‌شوند و فایل **bootblock.o** با فرمت **ELF (Executable Linkable Format)** تولید می‌شود. سپس بخش مربوط به متن این فایل به فرمت **raw binary** تبدیل شده و در فایل **bootblock** ذخیره می‌شود.

فایل‌های باینری (bin) تنها شامل داده‌های بایت به بایتی هستند که در یک آدرس خاص در حافظه قرار می‌گیرند اما فایل‌های ELF دارای اطلاعات اضافی نیز هستند که شامل symbol tables و debug information می‌شوند. فرمت ELF یک فرمت باینری استاندارد برای سیستم‌عامل‌هایی مانند لینوکس است. برخی از قابلیت‌های فرمت ELF عبارتند از توانایی اشتراک‌گذاری کتابخانه‌ها (shared library) با برنامه‌های دیگر و امکانات پیشرفته‌ای برای کنترل در زمان اجرا و dynamic loading. فرمت ELF به صورت یک نسخه فشرده از Bin است که CPU قادر به اجرای مستقیم آن نیست و نیاز به ترجمه آن توسط لینکر به یک شکل قابل اجرا برای CPU دارد. این ترجمه شامل تبدیل offsetها به موقعیت‌های صحیح نیز می‌شود. دلیل استفاده از فرمت دودویی خام raw binary در اینجا، از دو جهت مهم است. اولاً، حجم این نوع فایل بسیار کمتر از فرمت ELF است چرا که اطلاعات اضافی ندارد و فایل اولی که برای بوت اجرا می‌شود نباید حجم زیادی داشته باشد. دوماً، CPU قادر به اجرای مستقیم فایل ELF نیست و بنابراین سیستم‌عامل نیاز به یک فرمت قابل تشخیص برای بوت دارد که CPU آن را بفهمد، و این فرمت raw binary است.

```

Disassembly of section .data:
00000000 <.data>:
0: fa                cll
1: 31 c0             xor %ax,%ax
3: 8e d8             mov %ax,%ds
5: 8e c0             mov %ax,%es
7: 8e d0             mov %ax,%ss
9: e4 64            in $0x64,%al
b: a8 02            test $0x2,%al
d: 75 fa            jne 0x9
f: b0 d1            mov $0xd1,%al
11: e6 64            out %al,$0x64
13: e4 64            in $0x64,%al
15: a8 02            test $0x2,%al
17: 75 fa            jne 0x13
19: b0 df            mov $0xdf,%al
1b: e6 60            out %al,$0x60
1d: 0f 01 16 78 7c   lgdtw 0x7c78,%cr0,%eax
22: 0f 20 c0         mov %cr0,%eax
25: 66 83 c8 01     or $0x1,%eax
29: 0f 22 c0         mov %eax,%cr0
2c: ea 31 7c 08 00   jmp $0x8,$0x7c31
31: 66 b8 10 00 8e d8 mov $0xd88e0010,%eax
37: 8e c0             mov %ax,%es
39: 8e d0             mov %ax,%ss
3b: 66 b8 00 00 8e e0 mov $0xe08e0000,%eax
41: 8e e8             mov %ax,%gs
43: bc 00 7c         mov $0x7c00,%sp
46: 00 00             add %al,(%bx,%s1)
48: e8 fc 00         call 0x147
4b: 00 00             add %al,(%bx,%s1)
4d: 66 b0 00 8a 66 89 mov $0x89668a00,%eax
53: c2 66 ef         ret $0x66ef,%eax
56: 66 b8 e0 8a 66 ef mov $0xef668ae0,%eax
5c: eb fe            jmp 0x5c
5e: 66 90             xchg %eax,%eax
...
68: ff              (bad)
69: ff 00             incw (%bx,%s1)
6b: 00 00             add %al,(%bx,%s1)
6d: 9a cf 00 ff ff   lcall $0xffff,%eaxcf
72: 00 00             add %al,(%bx,%s1)
74: 00 92 cf 00     add %di,%ecx(%bp,%s1)
78: 17              pop %ax
79: 00 60 7c         add %ah,0x7c(%bx,%s1)
7c: 00 00             add %al,(%bx,%s1)
7e: f3 0f 1e fb     endbr32
82: ba f7 01         mov $0xf7f7,%dx
85: 00 00             add %al,(%bx,%s1)
87: ec             in (%dx),%al
88: e3 e0 c0         and $0xffc0,%ax
8b: 3c 40            cmp $0x40,%al
8d: 75 f8            jne 0x87
8f: c3              ret
  
```

کد اسمبلی خواسته شده

## ۱۲. علت استفاده از دستور objcopy در حین عملیات make چیست؟

این دستور با استفاده از کتابخانه GNU BFD محتوای یک object file را در یک فایل دیگر کپی می‌کند. این دستور با ساخت فایل‌های موقت و پاک کردن آنها پس از انجام عملیات ترجمه‌ها را انجام می‌دهد. دستور objcopy آپشن‌های مختلفی دارد از جمله -s که باعث می‌شود relocation و symbol information از فایل مبدا کپی نشوند. عملکرد دستور objcopy در makefile عامل xv6 این است که بخش متنی فایل bootblock.o را در فایل bootblock و در بخش entryother بخش متنی فایل bootblockother.o

را در فایل entryother کپی کند. همچنین فایل initcode.out را در initcode کپی می‌کند و سپس فایل‌های مربوط با هم لینک شده و سیستم عامل کامپایل می‌شود.

۱۴. یک ثبات عام‌منظوره یک ثبات قطعه، یک ثبات وضعیت و یک ثبات کنترلی در معماری x86 را نام برده و وظیفه هر یک را به طور مختصر توضیح دهید.

- ثبات عام منظوره: general purpose register ها در x86 ۳۲ بیتی هستند. در این سیستم ۸ عدد رجیستر عام منظوره با نام‌های eip, eax, ebx, ecx, edx, esi, ebp, esp وجود دارند. حرف e در ابتدای این رجیسترها به معنی extended است. بعضی پوینترها، عملیات ریاضی و دیتا در این رجیسترها نگه‌داشته می‌شوند.

- ثبات قطعه: چند نمونه از این نوع رجیستر، SS که در آذ پوینتر به استک، DS که در آذ پوینتر به دیتا و CS که پوینتر به کد نگه‌داشته می‌شود هستند.

- ثبات وضعیت: یک نمونه از این نوع رجیستر EFLAGS است که در آذ اطلاعاتی درباره وضعیت پردازنده از جمله zero flag, sign flag, carry flag نگهداری می‌شود.

- ثبات کنترلی: نمونه‌هایی از این نوع رجیستر cr0, cr2, cr3, cr4 هستند که با کنترلر interrupt و paging و coprocessor و تغییر مد آدرس دهی CPU را کنترل می‌کنند.

۱۸. کد معادل entry.S در هسته لینوکس را بیابید.

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

۱۹. چرا این آدرس فیزیکی است؟

اگر این حافظه مجازی باشد برای بدست آوردن حافظه فیزیکی‌اش باید به جدول نگاشت دسترسی داشته باشیم تا بفهمیم این آدرس به کدام آدرس فیزیکی می‌رود اما در ابتدا به حافظه فیزیکی جدول دسترسی نداریم (چون فقط آدرس مجازی‌اش را می‌دانیم) پس نمی‌توانیم نگاشت هیچ آدرسی را پیدا کنیم.

۲۲. علاوه بر صفحه‌بندی در حد ابتدایی از قطعه‌بندی به منظور حفاظت هسته استفاده خواهد شد. این

عملیات توسط seginit() انجام می‌گردد. همانطور 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؟

```

// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
    struct cpu *c;

    // Map "logical" addresses to virtual addresses using identity map.
    // Cannot share a CODE descriptor for both kernel and user
    // because it would have to have DPL_USR, but the CPU forbids
    // an interrupt from CPL=0 to DPL=3.
    c = &cpus[cpu_id()];
    c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
    c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
    c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
    c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
    lgdt(c->gdt, sizeof(c->gdt));
}

```

کد مربوط به تابع *seginit()* در فایل *vm.c*

```

// Normal segment
#define SEG(type, base, lim, dpl) (struct segdesc) \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
#define SEG16(type, base, lim, dpl) (struct segdesc) \
{ (lim) & 0xffff, (uint)(base) & 0xffff, \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
  (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
#endif

```

تعریف *SEG* در فایل *mmu.h*

هر قطعه تحت کنترل هسته یا کاربر به بخشی از حافظه دسترسی دارند. هر کدام از این قطعات توسط یک توصیفگر (descriptor) در جدول توصیفگر سراری (Global descriptor table یا gdt) توصیف شده اند. این توصیف اعم از آدرس شروع قطعه، اندازه قطعه و سطح دسترسی قطعه را به همراه دارد. در نتیجه وقتی یک دستور قرار است اجرا شود ابتدا آدرس قطعه آن از توصیفگرش بدست می آید و سپس آدرس فیزیکی دستور از صفحه موردنظر پیدا می شود و دستور از حافظه خوانده می شود اما سطح دسترسی این قطعه توانایی دسترسی دادن یا ندادن را برای اجرای این دستور برای سطح دسترسی موردنظر به ما می دهد.

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



```
// Per-process state
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)
};
```

کد مورد نظر در فایل *proch*

۱. `uint sz` : سایز حافظه این پردازش به بایت را نگهداری می‌کند.
۲. `pde_t* pgdir` : اشاره‌گری به `page table` این پردازش است.
۳. `char* kstack` : اشاره‌گری به پایین استک کرنل این پردازش است.
۴. `enum procstate state` : یک متغیر از جنس `enum` است که وضعیت پردازش را نشان می‌دهد. این وضعیت یکی از وضعیت‌های `UNUSED`, `EMBRYO`, `SLEEPING`, `RUNNABLE`, `RUNNING`, `ZOMBIE` می‌تواند باشد.
۵. `int pid` : شناسه یکتای یک پردازش را نشان می‌دهد.
۶. `struct proc *parent` : اشاره‌گری (از جنس پردازش) به پدر این پردازش، که پردازش‌ایست که سازنده پردازش کنونی است.
۷. `struct trapframe* tf` : اشاره‌گری به `trap frame` برای `syscall` فعلی است.
۸. `struct context* context` : اشاره‌گری به `struct context` که مقادیر رجیسترهای مورد نیاز را برای `context switching` در خودش نگهداری می‌کند. پ.ن: رجیسترهای `edi`, `esi`, `ebx`, `ebp`, `eip` در `struct context` ذخیره می‌شوند.
۹. `void* chan` : در صورتی که ناصفر باشد یعنی پردازش خوابیده است.
۱۰. `int killed` : اگر ناصفر باشد پردازش `kill` شده است.
۱۱. `struct file* ofile[NOFILE]` : آرایه‌ایست که فایل‌های باز شده را نگهداری می‌کند.
۱۲. `struct inode *cwd` : مشخص‌کننده `current working directory` است.
۱۳. `Char name[۱۶]` : اسم پردازش را نگهداری می‌کند (برای اشکالزدایی استفاده می‌شود).

۲۷. کدام بخش از آماده سازی سیستم، بین تمام هسته‌های پردازنده و کدام بخش اختصاصی است؟ (از هر کدام یک مورد را با ذکر دلیل توضیح دهید.) زمان‌بندی روی کدام هسته اجرا می‌شود؟

در انتهای فایل `entry.s`، فرصت اجرای کد هسته `C` فراهم می‌شود. در این نقطه، تابع `main` فراخوانی می‌شود، که به عنوان نقطه شروع اجرای هسته‌های سیستم عامل که در مرحله بوت کردن قرار دارند، عمل می‌کند. در این تابع `main`، چندین تابع دیگر نیز فراخوانی می‌شوند.

در مقابل، CPU‌های دیگر به کمک فایل `entryother.s` به تابع `mpenter` می‌روند. در این تابع ۴ تابع دیگر نیز فراخوانی می‌شود. این چهار تابع به عنوان پایه‌هایی عمل می‌کنند که تمامی هسته‌های پردازنده مشترک دارند. به عبارت دقیق‌تر، این توابع در داخل تابع `main` نیز فراخوانی می‌شوند. در واقع، وظایفی که در این چهار تابع انجام می‌شوند، عمدتاً متعلق به کارهایی هستند که باید توسط تمامی هسته‌های پردازنده مشترک انجام شوند. از طرف دیگر، وظایفی که در چهارده تابع دیگر در تابع `main` انجام می‌شوند، بیشتر اختصاصی و وابسته به هسته‌هایی هستند که سیستم عامل را بوت می‌کنند. به عنوان مثال تابع `consoleinit` در تابع `main` فراخوانی می‌شود و برای هسته‌هایی است که سیستم عامل را بوت می‌کنند. در این تابع، `initlock` فراخوانی می‌شود که برای قفل اولیه و مدیریت نوشته‌های روی کنسول اولیه استفاده می‌شود و نیازی نیست که این منابع بین تمامی هسته‌ها به اشتراک گذاشته شوند. تابع `mpmain` نیز یک تابع مشترک است که تنظیم و آماده‌سازی CPU‌ها را انجام می‌دهد و کارهای مرتبط با لود کردن ثبت‌های `IDT (Interrupt Descriptor Table)` را انجام می‌دهد. در نهایت، `scheduler` در تابع `mpmain` فراخوانی می‌شود و وظیفه‌ی توزیع زمان بین تمامی هسته‌ها را انجام می‌دهد. این کارها بین تمامی هسته‌ها به صورت مشترک انجام می‌شوند و هر CPU پس از تنظیم خودش، این توابع را فراخوانی می‌کند.

## اشکال زدایی

اجرای GDB

(۱) برای دیدن breakpoint ها از دستور `info breakpoints` استفاده می‌کنیم:

```
Reading symbols from _echo...
(gdb) break echo.c:10
Breakpoint 3 at 0x4: file echo.c, line 10.
(gdb) info breakpoints
Num      Type             Disp Enb Address            What
3        breakpoint       keep y   0x00000004 in main at echo.c:10
(gdb)
```

(۲) برای حذف یک breakpoint از دستور `del id` استفاده می‌کنیم به این صورت که `id` برابر شماره breakpoint مورد نظر است.

```
(gdb) break echo.c:10
Breakpoint 3 at 0x4: file echo.c, line 10.
(gdb) info breakpoints
Num      Type           Disp Enb Address      What
3        breakpoint     keep y   0x00000004   in main at echo.c:10
(gdb) del 3
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb) □
```

کنترل روند اجرا و دسترسی به حالت سیستم  
 (۳) دستور bt مخفف backtrace است و call stack را در لحظه کنونی برنامه نشان می دهد.

```
atalasion@ubuntu: ~/xv6-public  x  atalasion@ubuntu: ~/xv6-public  x  ▾
(gdb) p *input.buf@128
$2 = "ls\n", '\000' <repeats 124 times>
(gdb) p &input.e
$3 = (uint *) 0x80110fa8 <input+136>
(gdb) x 0x80110fa8
0x80110fa8 <input+136>:      add    (%eax),%eax
(gdb) bt
#0  consoleintr (getc=<optimized out>) at console.c:379
#1  0x80113d00 in ?? ()
#2  0x80102f04 in kbdintr () at kbd.c:49
#3  0x80102e10 in ?? ()
#4  0x8010621a in trap (tf=0x8010619d <trap+221>) at trap.c:67
#5  0x8010c504 in stack ()
#6  0x8010619d in trap (tf=0x801142d4 <ptable+52>) at trap.c:110
#7  0x00000030 in ?? ()
#8  0x801142d4 in ptable ()
#9  0x80113d00 in ?? ()
#10 0x80105ff3 in alltraps () at trapasm.S:20
#11 0x8010c508 in stack ()
#12 0x80113d04 in cpus ()
#13 0x80113d00 in ?? ()
#14 0x8010378f in mpmain () at main.c:57
#15 0x801038dc in main () at main.c:37
(gdb) □
```

(۴) با استفاده از دستور print می توان مقدار یک متغیر را چاپ کرد اما با استفاده از دستور X می توان محتویات یک خانه حافظه را چاپ کرد منتها هر دو به فرمت FMT/ در خروجی قرار می دهند.

```
atalasion@ubuntu: ~/xv6-public x atalasion@ubuntu: ~/xv6-public x
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
[f000:fff0] 0xfffff0: ljmp $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
(gdb) b console.c:379
Breakpoint 1 at 0x8010116c: file console.c, line 379.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x8010116c <consoleintr+1404>:      sub    $0xc,%esp
Thread 1 hit Breakpoint 1, consoleintr (getc=<optimized out>) at console.c:379
379      input.com = input.tot_com;
(gdb) p input.e
$1 = 3
(gdb) p *input.buf@128
$2 = "ls\n", '\000' <repeats 124 times>
(gdb) p &input.e
$3 = (uint *) 0x80110fa8 <input+136>
(gdb) x 0x80110fa8
0x80110fa8 <input+136>:      add    (%eax),%eax
(gdb) bt
#0  consoleintr (getc=<optimized out>) at console.c:379
#1  0x80113d00 in ?? ()
```

(۵) برای دیدن وضعیت ثبات‌ها، می‌توان از دستور info register استفاده کرد.

```
#15 0x801038dc in main () at main.c:37
(gdb) info register
eax             0x1             1
ecx             0xa            10
edx             0x3            3
ebx             0x0            0
esp             0x8010c484      0x8010c484 <stack+3780>
ebp             0x8010c4ac      0x8010c4ac <stack+3820>
esi             0x0            0
edi             0x0            0
eip             0x8010116c      0x8010116c <consoleintr+1404>
eflags          0x93          [ IOPL=0 SF AF CF ]
cs              0x8            8
ss              0x10           16
ds              0x10           16
es              0x10           16
fs              0x0            0
gs              0x0            0
fs_base         0x0            0
gs_base         0x0            0
k_gs_base       0x0            0
cr0             0x80010011      [ PG WP ET PE ]
cr2             0x0            0
cr3             0x3ff000      [ PDBR=0 PCID=0 ]
cr4             0x10          [ PSE ]
--Type <RET> for more, q to quit, c to continue without paging--
```

برای مشاهده متغیرهای محلی نیز می‌توان از دستور info locals استفاده کرد.

```
Quit
(gdb) info locals
c = <optimized out>
doprocDump = <optimized out>
(gdb) □
```

رجیستر esi یک رجیستر ۳۲ بیتی است که به عنوان ثابت مبدا، در عملیات‌های مربوط به رشته، مورد استفاده قرار می‌گیرد.

در مقابل رجیستر edi یک رجیستر ۳۲ بیتی است که به عنوان ثابت مقصد، در عملیات‌های مربوط به رشته، مورد استفاده قرار می‌گیرد.

۶) struct input به صورت زیر است:

```
c = <optimized out>
doprocdump = <optimized out>
(gdb) ptype input
type = struct {
  char buf[128];
  uint r;
  uint w;
  uint e;
  uint idx;
  uint com;
  uint tot_com;
}
(gdb) □
```

مقدار input.r جایگاهی است که پس از اینتر باید بافر خوانده شود.

مقدار input.w مقدار اول خط است.

مقدار input.e مقدار آخر خط است که با نوشتن اضافه و با بک اسپیس کم می‌شود.

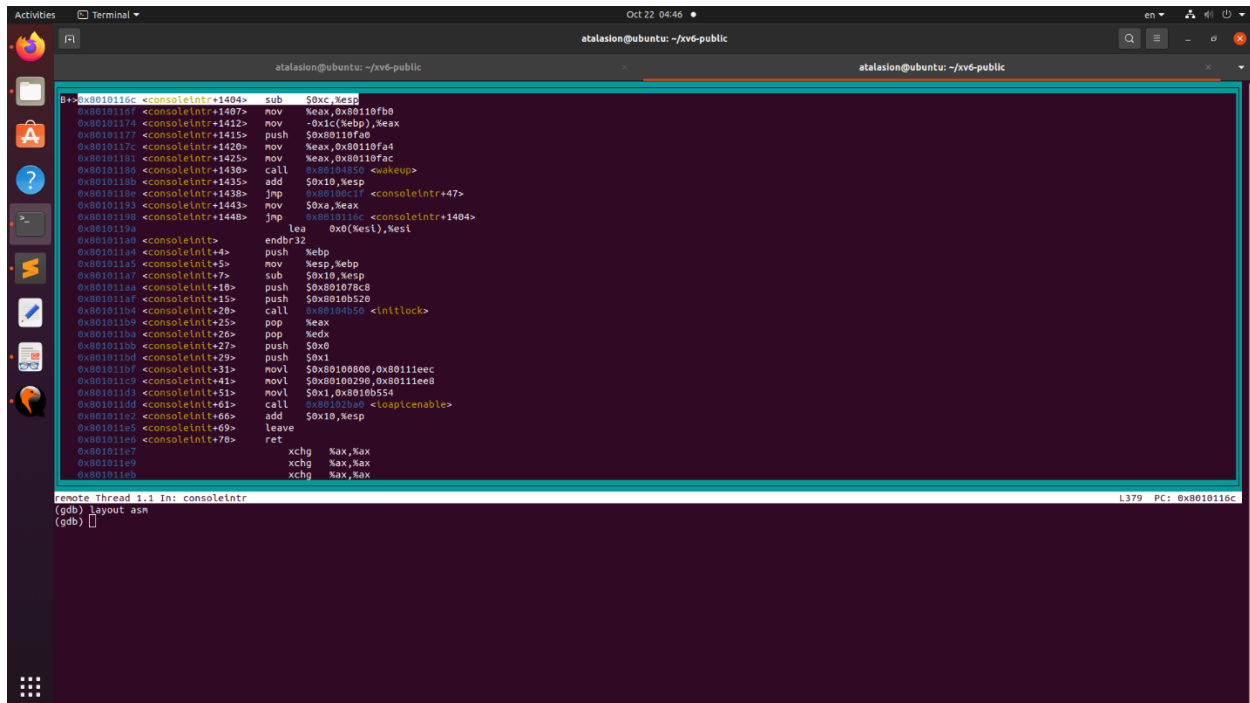
مقدار input.idx برابر خطی است که الاز کرسر بر روی آن قرار دارد و با تغییرات Ctrl + B و Ctrl + F تغییر می‌کند.

مقدار input.com برابر کامندی است که الاز بر روی آن قرار داریم (برای ArrowUp و ArrowDown)

مقدار input.tot\_com برابر کل تعداد کامندهای ذخیره شده برای عملیات‌های ArrowUp و ArrowDown است.

۷) دستور layout src سورس کد را نمایش می‌دهد:

دستور asm layout کد اسمبلی جای برک شده را نشان می‌دهد.



```
Oct 22 04:46
atalaslon@ubuntu: ~/xv6-public
atalaslon@ubuntu: ~/xv6-public
B+ 0x8010116c <consoleintr+1404> sub $0xc,%esp
0x8010116f <consoleintr+1407> mov %eax,0x80110fb0
0x80101174 <consoleintr+1412> mov -0xc(%ebp),%eax
0x80101177 <consoleintr+1415> push $0x80110fa0
0x8010117c <consoleintr+1420> mov %eax,0x80110fa4
0x80101181 <consoleintr+1425> mov %eax,0x80110fac
0x80101188 <consoleintr+1430> call 0x80104b50 <wakeupt>
0x8010118b <consoleintr+1435> add $0x10,%esp
0x8010118e <consoleintr+1438> jmp 0x80101177 <consoleintr+47>
0x80101193 <consoleintr+1443> mov $0xa,%eax
0x80101198 <consoleintr+1448> jmp 0x8010116c <consoleintr+1404>
0x8010119a <consoleintr+1450> lea 0x0(%esi),%esi
0x801011a0 <consoleintr+1458> endbr32
0x801011a4 <consoleintr+1462> push %ebp
0x801011a5 <consoleintr+1463> mov %esp,%ebp
0x801011a7 <consoleintr+1465> sub $0x10,%esp
0x801011aa <consoleintr+1468> push $0x801078c8
0x801011af <consoleintr+1473> push $0x8010b520
0x801011b4 <consoleintr+1480> call 0x80104b50 <initlock>
0x801011b9 <consoleintr+1485> pop %eax
0x801011ba <consoleintr+1486> pop %edx
0x801011bb <consoleintr+1487> push $0x0
0x801011bd <consoleintr+1489> push $0x1
0x801011bf <consoleintr+1491> movl $0x80100800,0x80111eec
0x801011c9 <consoleintr+1497> movl $0x80100290,0x80111ee8
0x801011d3 <consoleintr+1501> movl $0x1,0x8010b554
0x801011d8 <consoleintr+1506> call 0x80102b00 <loapicenable>
0x801011db <consoleintr+1509> add $0x10,%esp
0x801011e5 <consoleintr+1515> leave
0x801011e6 <consoleintr+1516> ret
0x801011e7 <consoleintr+1517> xchg %ax,%ax
0x801011e8 <consoleintr+1518> xchg %ax,%ax
0x801011e9 <consoleintr+1519> xchg %ax,%ax
remote Thread 1.1 In: consoleintr
(gdb) layout asm
(gdb)
```

۸) با استفاده از دستور where می‌توان وضعیت پشته فراخوانی را دید:

```
remote Thread 1.1 In:
#0  consoleintr (getc=<optimized out>) at console.c:379
#1  0x80113d00 in ?? ()
#2  0x80102f04 in kbdintr () at kbd.c:49
#3  0x80102e10 in ?? ()
#4  0x8010621a in trap (tf=0x8010619d <trap+221>) at trap.c:67
#5  0x8010c504 in stack ()
#6  0x8010619d in trap (tf=0x801142d4 <ptable+52>) at trap.c:110
#7  0x00000030 in ?? ()
#8  0x801142d4 in ptable ()
#9  0x80113d00 in ?? ()
#10 0x80105ff3 in alltraps () at trapasm.S:20
#11 0x8010c508 in stack ()
#12 0x80113d04 in cpus ()
#13 0x80113d00 in ?? ()
#14 0x8010378f in mpmain () at main.c:57
#15 0x801038dc in main () at main.c:37
(gdb)
```

با استفاده از دستور up و down می‌توان در پشته حرکت کرد:

```
emote inread 1.1 in:
gdb) #12 0x80113d84 in cpus ()
gdb) #14 0x8010378f in mpmain () at main.c:57
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) #15 0x801038dc in main () at main.c:37
gdb) up 2
#15 0x801038dc in main () at main.c:37
gdb) □
```