

سهیل حاجیان منش 810100119

امیرعلی رحیمی 810100146

مهدی نوری 810100231

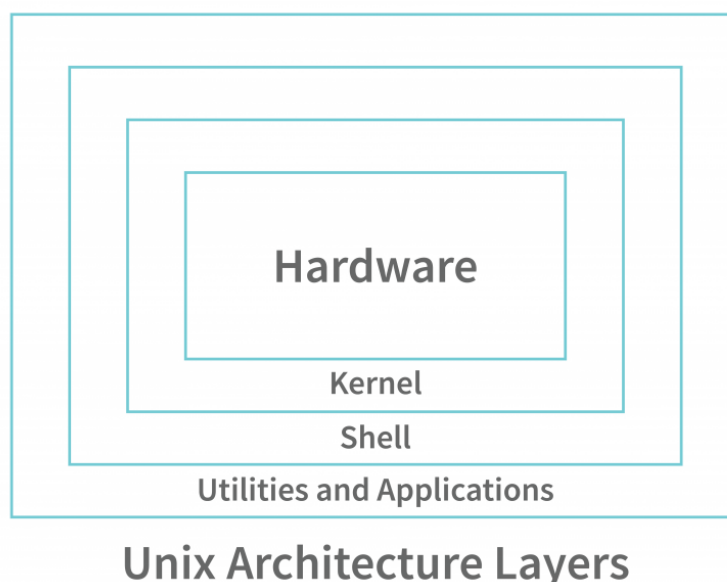
github repo : <https://github.com/MahdiNoori2003/OS-Lab-1>

last commit : 634fafa

آشنایی با سیستم عامل xv6

1. معماری سیستم عامل xv6

xv6، یک پیاده سازی مجدد از Unix Version 6 (v6) می باشد. این سیستم عامل، تقریباً از ساختار v6 پیروی می کند با این تفاوت که در ANSI C (که یکی از استانداردها برای زبان برنامه نویسی C می باشد) برای پردازنده های مبتنی بر x86 طراحی شده است. این نکته را می توان در فایل x86.h نیز مشاهده کرد که در آن از دستورات پردازنده های x86 استفاده شده است. همچنین در فایل های دیگری نظیر asm.h هم این نکته مشاهده می شود. معماری سیستم عامل Unix به شکل زیر می باشد:



2. اجزای مختلف پردازش و نحوه ی اختصاص پردازنده به پردازش های مختلف در

سیستم عامل xv6

هر برنامه ی در حال اجرا در xv6، که یک پردازش نامیده می شود، از دو بخش تشکیل شده است. اولی user-space-memory که شامل دستورات، داده ها و استک است. و دیگری per-process-state است که همان وضعیت پردازش می باشد که تنها هسته به آن دسترسی دارد. نحوه ی اختصاص پردازنده به پردازش های مختلف در xv6 به صورت time-share می باشد. به این صورت که به صورت متوالی و بدون متوجه شدن کاربر، پردازنده های در دسترس را به پردازش های در دست اجرا اختصاص می دهد و پس از مدتی کوتاه، این پردازنده ها را به پردازش های دیگری که منتظر اجرا شدن هستند، اختصاص می دهد. و همین روند ادامه پیدا می کند. به صورتی که پردازش ها به صورت تقریباً موازی با یکدیگر اجرا می شوند. زمانی که یک پردازش در حال اجرا نیست، سیستم عامل رجیستر های CPU که حاوی موارد مورد نیاز توسط آن پردازش بود را در حافظه ذخیره می کند و دوباره زمانی که نوبت به اجرای دوباره ی این پردازش رسید، آنها را بازیابی می کند. هسته به هر پردازش، یک pid اختصاص میدهد (process ID). که توسط آن، میتواند آن پردازش را با کمک system-call های مختلف مدیریت کند که لیستی از آنها را مشاهده میکنیم:

System call	Description
fork()	Create a process
exit()	Terminate the current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return the current process's pid
sleep(n)	Sleep for n clock ticks
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; the flags indicate read/write
read(fd, buf, n)	Read n bytes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

Figure 0-2. Xv6 system calls

4. کاربرد exec و fork و مزیت ادغام نکردن این دو

یک پردازش میتواند با استفاده از فراخوان سیستمی fork یک پردازش جدید ایجاد کند. fork یک پردازش جدید بعنوان پردازش فرزند میسازد که محتویات (دیتا و دستورات) حافظه یکسانی با پردازش پدر که fork را صدا زده دارد. با این وجود این دو پردازنده حافظه ای جداگانه خواهند داشت و تغییر یک متغیر در یکی از آنها موجب تغییر آن در دیگری نمی شود. نقطه شروع پردازنده فرزند caller تابع

fork است. از طرفی پس از ایجاد پردازش فرزند به همان caller تابع fork در پردازش پدر بازمیگردیم و اینگونه امکان اجرای هر دو پردازش ایجاد میشود. مقدار pid که خروجی تابع fork است سه حالت میتواند داشته باشد که هرکدام نشان دهنده یه وضعیت هستند:

1. pid=0 : در پردازش فرزند هستیم.
 2. pid>0 : پردازش فرزند ایجاد شده و در پردازش پدر هستیم و pid آی دی پردازش فرزند است.
 3. pid<0 : در هنگام اجرای پردازش فرزند به اروری خورده ایم و ایجاد نشده است.
- قطعه کد زیر را بعنوان مثال در نظر بگیرید:

```
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");
}
```

فراخوانی سیستمی wait باعث میشود پردازش پدر صبر کند تا پردازش فرزند پایان یابد سپس ادامه یابد. خروجی آن آی دی پردازش فرزند است اگر وجود نداشته باشد صفر خروجی میدهد. فراخوانی سیستمی exit منجر می شود پردازش صدا زده شده اجرایش قطع شود.

فراخوانی سیستمی exec حافظه پردازش فعلی را با یک حافظه که در یک فایل با فرمت ELF است جایگزین میکند. اگر exec با موفقیت اجرا شود به پردازش قبلی باز میگردد و برنامه موجود در حافظه جدید اجرا می شود. تابع exec دو ورودی دارد ورودی اول نام فایل برنامه و ورودی دوم آرگومان های برنامه است.

قطعه کد زیر را به عنوان مثال در نظر بگیرید:

```
v[3];

: "echo";
: "hello";
: 0;
n/echo", argv);
xec error\n");
```

زمانی که کاربر در یک برنامه را اجرا میکند کاری در shell انجام میشود اینگونه است که ابتدا بعد از خواندن دستور برنامه در ترمینال تابع fork صدا زده میشود و یک پردازش جدید ایجاد میکند در این پردازش جدید با فراخوانی exec برنامه کاربر را جایگزین پردازش جدید میکند. در پردازش پدر (پردازش اولیه) برای اتمام کار مربوط به پردازش حاوی برنامه کاربر wait میکند. بعد از اتمام کار این پردازش به main رفته و منتظر برنامه بعدی توسط کاربر میشود.

به قطعه کد زیر به عنوان مثال دقت کنید: (مربوط به دستور cat < input.txt)

```
char *argv[2];

argv[0] = "cat";
argv[1] = 0;
if(fork() == 0) {
    close(0);
    open("input.txt", O_RDONLY);
    exec("cat", argv);
}
```

اکنون روشن است چرا این دو فراخوانی سیستمی ادغام نشدند. چون در این حالت shell میتواند یک پردازش فرزند رو fork کند و با استفاده از open, close, dup در پردازش فرزند در ورودی ها و خروجی های استاندارد file descriptors تغییرات ایجاد کند و سپس exec را فراخوانی کند. هیچ تغییری در برنامه در حال اجرا مورد نیاز نیست. اگر این دو تابع ادغام شوند احتمالا طراحی پیچیده تری برای shell نیاز است تا redirection را هندل کند یا برنامه باید خودش بفهمد چگونه redirection I/O را هندل کند.

اجرا و اشکال زدایی

اضافه کردن یک متن به Boot Message

در فایل init.c بخش زیر را اضافه می کنیم:

```

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

for(;;){
    printf(1, "init: starting sh\n");
    printf(1, "Group #31:\n");
    printf(1, "1. Mahdi Noori\n");
    printf(1, "2. AmirAli Rahimi\n");
    printf(1, "3. Soheil Hajian\n");

    pid = fork();
    if(pid < 0){
        printf(1, "init: fork failed\n");
        exit();
    }
}

```

خروجی به صورت زیر است :

```

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 #31:
1. Mahdi Noori
2. AmirAli Rahimi
3. Soheil Hajian
$ _

```

اضافه کردن چند قابلیت به کنسول xv6

دستور CTRL+B

برای این بخش کمی تغییر در input دادیم که در زیر آمده است :

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

متغیر پوینتر محل کرسر و e انتهای خط را نشان میدهد . با استفاده از این دو بررسی میکنیم که آیا ctrl + b زده شده یا نه .

```
if (input.pointer != input.w)
{
```

مشخصا اگر ctrl+b زده نشده باشد پوینتر با e یکی است.

هنگامی که ctrl + b زده شود پوینتر یکی به عقب می آید . از طرفی pos که ایندکس آرایه نمایش crt است هم عقب می آید.

حال اگر کاراکتر جدیدی تایپ شود بافر و crt را یکبار به جلو شیفت میدهیم و کاراکتر جدید را وارد خانه ی خالی شده میکنیم.

```
void shiftRight()
{
    for (int i = input.e; i > input.pointer; i--)
    {
        input.buf[i % INPUT_BUF] = input.buf[(i - 1) % INPUT_BUF];
    }
}
```

```

else if (c == SHIFTRIGHT)
{
    for (int i = (pos + input.e - input.pointer); i > pos; i--)
    {
        crt[i] = crt[i - 1];
    }
}

```

هنگام حذف هم هر دو را یکی به عقب شیفت می‌دهیم.

```

else if (c == SHIFTLEFT)
{
    for (int i = pos - 1; i < (pos + input.e - input.pointer); i++)
    {
        crt[i] = crt[i + 1];
    }
    crt[pos + input.e - input.pointer] = ' ';
    pos--;
}

```

(در قسمت پایینی کد بالا عملیات پاک کردن کاراکتر رخ داده)

```

void shiftLeft()
{
    for (int i = input.pointer - 1; i < input.e; i++)
    {
        input.buf[i % INPUT_BUF] = input.buf[(i + 1) % INPUT_BUF];
    }
}

```

در ادامه هم پوینتر و e باید یکی عقب بیایند که در بخش BACKSPACE تابع ctrlintr انجام شده است.

دستور CTRL+F

در این بخش هنگامی که ctrl + f زده می‌شود اگر پوینتر به e نرسیده باشد آن را جلو می‌بریم.

همچنین pos هم یکی جلو می‌رود.

```
case C('F'):  
    if (input.pointer != input.e)  
    {  
        input.pointer++;  
        consputc(CURSORFORWARD);  
    }  
    break;
```

```
else if (c == CURSORFORWARD)  
{  
    pos++;  
}
```

دستور CTRL+L

در این بخش ابتدا crt را خالی میکنیم و pos را ریست میکنیم .

```
else if (c == CLEAR)  
{  
    for (int i = 0; i <= CRTLEN; i++)  
    {  
        crt[i] = ' ' | 0x0700;  
    }  
    pos = 0;  
}
```

سپس پوینتر و e را به ابتدای بافر می آوریم که همان w است. و در ادامه یک \$ و اسپیس میگذاریم .


```

case C('L'):
    consputc(CLEAR);
    input.e = input.w;
    input.pointer = input.w;
    consputc('$');
    consputc(' ');
    break;

```

↑ دستور

هر دستور را در یک struct با buffer حلقوی نگه میداریم که پوینتر نشان دهنده محل فعلی در استک است.

سایز بافر این این استک 10 در 128 است . 10 تعداد دستورات قابل نگهداری و 128 ماکس سایز هر دستور است. در این جا برای بررسی اجازه جابجایی از movement استفاده میکنیم به این صورت که به ازای هر بار arrow up یکی آن را زیاد میکنیم و با هر بار arrow down یکی از آن میکاهیم.

```

#define CMD_BUF_SIZE 10
struct
{
    char buf[CMD_BUF_SIZE][INPUT_BUF];
    uint r; // buffer head index
    uint w; // Write index
    uint pointer;
    uint s; // buffer size
    uint movement; // movement in relation to head
} cmd_buffer;

```

با هر بار arrow up پوینتر یکی کم شده و دستور قبلی لود میشود.

```

case ARROW_UP:
    if (CAN_ARROW_UP && cmd_buffer.s > 0)
    {
        input.e = input.w;
        input.pointer = input.w;
        if (cmd_buffer.pointer == 0)
        {
            cmd_buffer.pointer = 9;
        }
        else
        {
            cmd_buffer.pointer--;
        }
        cmd_buffer.movement++;
        cmd_buffer.pointer %= CMD_BUF_SIZE;
        loadCommand();
        consputc(ARROW_UP);
    }
    break;

```

```

void loadCommand()
{
    for (int i = 0;; i++)
    {
        if (cmd_buffer.buf[cmd_buffer.pointer][i] == END_OF_ARRAY)
        {
            input.pointer %= INPUT_BUF;
            input.e %= INPUT_BUF;
            break;
        }
        input.buf[(input.w + i) % INPUT_BUF] = cmd_buffer.buf[cmd_buffer.pointer][i];
        input.e++;
        input.pointer++;
    }
}

```

(لود کردن دستور در ارایه بافر)

```

else if (c == ARROW_UP || c == ARROW_DOWN)
{
    for (int i = (pos % 80); i > 2; i--)
    {
        crt[pos] = ' ' | 0x0700;
        pos--;
    }
    for (int i = 0; i < (input.e - input.w) % INPUT_BUF; i++)
    {
        crt[pos] = (input.buf[(i + input.w) % INPUT_BUF] & 0xff) | 0x0700;
        pos++;
    }
}
}

```

(لود کردن دستور در ارایه crt)

هر بار هم که دستوری وارد و ثبت شود وارد بافر میشود.

```

if (c == END_OF_LINE || c == C('D')) || input.e == input.r + INPUT_BUF)
{
    input.pointer = input.e;
    recordCommand();
}

```

```

void recordCommand()
{
    int i;
    if (cmd_buffer.s != CMD_BUF_SIZE)
    {
        for (i = 0; i < (input.e - input.w) % INPUT_BUF; i++)
        {
            if (input.buf[(i + input.w) % INPUT_BUF] != END_OF_LINE || input.buf[(i + input.w) % INPUT_BUF] != C('D'))
                cmd_buffer.buf[cmd_buffer.w][i] = input.buf[i + input.w];
        }
        cmd_buffer.buf[cmd_buffer.w][i] = END_OF_ARRAY;
        cmd_buffer.s++;
    }
    else
    {
        for (i = 0; i < (input.e - input.w) % INPUT_BUF; i++)
        {
            if (input.buf[(i + input.w) % INPUT_BUF] != END_OF_LINE || input.buf[(i + input.w) % INPUT_BUF] != C('D'))
                cmd_buffer.buf[cmd_buffer.r][i] = input.buf[i + input.w];
        }
        cmd_buffer.buf[cmd_buffer.r][i] = END_OF_ARRAY;
        cmd_buffer.r++;
        cmd_buffer.r %= 10;
    }
    cmd_buffer.w++;
    cmd_buffer.w %= 10;
    cmd_buffer.pointer = cmd_buffer.w;
}

```

(در این جا مکانیزم حلقوی بودن بافر لحاظ شده)

↓ دستور

در arrow down همان مکانیزم بالا پیاده میشود و تنها فرق این است که پوینتر به جای عقب جلو میرود. در بخش else در صورتی که کاربر arrow down بزند اما امکان آن وجود نداشته باشد خط وارد شده پاک میشود.

```
case ARROW_DOWN:
    input.e = input.w;
    input.pointer = input.w;
    if (CAN_ARROW_DOWN && cmd_buffer.s > 0)
    {
        cmd_buffer.pointer++;
        cmd_buffer.movement--;
        cmd_buffer.pointer %= CMD_BUF_SIZE;
        loadCommand();
        consputc(ARROW_DOWN);
    }
    else
    {
        consputc(CURSORRESET);
    }
    break;
```

توجه

هر دو دستور arrow up و arrow down برای لود دستور محدودیت دارند.

دستور arrow up بعد از اینکه به اولین دستور ورودی بافر میرسد نباید بالاتر برود.

دستور arrow down بعد از اینکه به آخرین دستور ورودی بافر میرسد نباید پایین تر برود.

دو شرط زیر برای بررسی این حالات در نظر گرفته شده است:

```
#define CAN_ARROW_UP (cmd_buffer.movement >= cmd_buffer.s) ? 0 : 1
#define CAN_ARROW_DOWN (cmd_buffer.movement <= 1) ? 0 : 1
```

اجرا و پیاده سازی یک برنامه سطح کاربر

برای این کار یک برنامه به زبان C به نام `strdiff.c` میسازیم و کد خود را در آن جا مینویسیم. سپس این برنامه را باید به برنامه های سطح کاربر اضافه کنیم که برای اینکار بازم است تغییراتی در `MakeFile` اعمال کنیم:

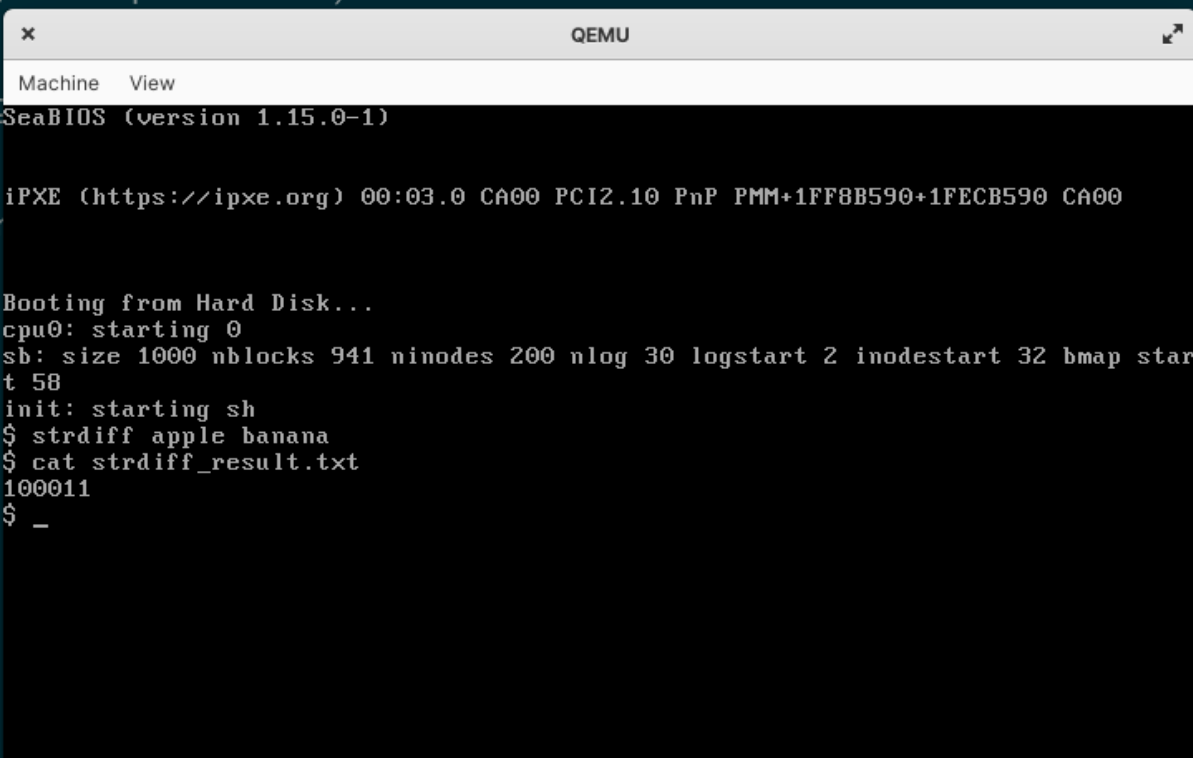
```
EXTRA=\
mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
strdiff.c\
printf.c umalloc.c\
README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
.gdbinit.tmpl gdbutil\
```

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

برنامه نوشته شده را به متغیر های `EXTRA` و `PROGS` در `MakeFile` اضافه میکنیم .

الگوریتم نوشته شده در `strdiff.c` هم اینگونه است که ابتدا یک ارایه به طول ماکزیمم دو رشته از کاراکتر های صفر و یک میسازیم که نشان دهنده تفاوت بین دو رشته است . سپس فایل را با آدرس گفته شده در سوال باز میکنیم و اگر از قبل وجود داشت متن موجود در آن را پاک میکنیم. سپس در مرحله اخر ارایه ساخته شده را در فایل مینویسم.

عکس زیر نمونه استفاده از `strdiff` که خروجی همان است که در صورت پروژه گفته شده است:



```
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ strdiff apple banana
$ cat strdiff_result.txt
100011
$ _
```

کامپایل سیستم عامل xv6

8. کاربرد UPROGS و ULIB در Makefile سیستم عامل

UPROGS: این متغیر، که نام آن مختصر شده `user programs` می باشد، همانطور که از اسمش پیداست لیستی از برنامه های سطح کاربر را دارد. نام هرکدام از اعضای آن به صورت `_fileName` می باشد که هنگام ساخت و کامپایل سیستم عامل، این برنامه های کاربر نیز به همراه آن کامپایل می شوند. هرکدام از این فایل ها، یک هدف دارند که نیاز به `fileName.o` و متغیر `ULIB` دارند.

ULIB: این متغیر، که نام آن مختصر شده user libraries می باشد. این متغیر شامل لیستی از کتابخانه های زبان C است که در بسیاری از فایل ها برای اجرا توابع مربوط به آن کتابخانه ها باید کامپایل شوند. برای مثال همانطور که در بالا گفته شد، پیش نیاز ساخت فایل هدف برنامه های کاربر، کامپایل ULIB می باشد.

مراحل بوت سیستم عامل xv6

11. تفاوت فایل دودویی بوت و دیگر فایل های دودویی xv6 و تبدیل آن به کد

اسمبلی

فرمت فایل های باینری آجکت xv6، ELF است. این نوع فایل چند هدر در ابتدا دارد که هرکدام بخشی از کد یا حافظه را در بر دارند. در فایل ELF.h، دو هدر ELF header و program header تعریف شده اند. e_entry در ELF header نقطه ی شروع برنامه را ذخیره می کند. برخی از section های ELF به صورت زیر است:

1. text. شامل دستورات قابل اجرای برنامه است.
2. rodata. شامل داده های read-only می باشد.
3. data. شامل داده های مقداردهی شده است.
4. bss. شامل داده های مقداردهی نشده است که چون داده ای نیست، فقط آدرس و اندازه ی آن ذخیره می شود.

با دستور `objdump -h bootblock.o`، می توان نوع فایل باینری و section های ELF را مشاهده کرد. بوت لودر در آدرسی ثابت لود می شود. در ادامه پردازنده به این آدرس مراجعه کرده و کرنل ساخته شده را اجرا می کند.

`bootblock.o` برخلاف باقی باینری فایل ها فقط بخش `text` را دارد.

`bootblock` `text` `objcopy -S -O binary -j`

با استفاده از دستور بالا فایل `text` را به صورت raw binary به `bootblock` می ریزد. مشخصا چون این دستور فرمت ELF ندارد، با فایل های باینری دیگر xv6 تفاوت دارد و به غیر از کد اجرایی اطلاعات دیگری ندارد.

دلیل اینکه فایل خروجی در `bootblock` ELF نیست، این است که CPU توان پردازش فایل های ELF را ندارد و نیاز به هسته ی سیستم عامل داریم.

برای تبدیل `bootblock` به اسمبلی، از کامند زیر استفاده می کنیم:

objdump -D -b binary -m i386 -M addr16,data16 bootblock

1. -D: برای disassemble کردن باینری.
2. -b binary: نوع فایل را raw binary در نظر می گیریم.
3. -m i386: معماری اسمبلی فایل را مشخص می کنیم.
4. -M addr 16,data16: برای یکسان سازی میزان بیت حالت disassemble با assemble

```
bootblock:      file format binary

Disassembly of section .data:

00000000 <.data>:
 0:  fa          cli
 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
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    ljmp     $0x8,$0x7c31
31:  66 b8 10 00 8e d8 mov     $0xd8e0010,%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,%si)
48:  e8 f0 00        call    0x13b
4b:  00 00        add     %al,(%bx,%si)
4d:  66 b8 00 8a 66 89 mov     $0x89668a00,%eax
53:  c2 66 ef        ret     $0xef66
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,%si)
6b:  00 00    add     %al, (%bx,%si)
6d:  9a cf 00 ff ff    lcall  $0xffff,$0xcf
72:  00 00    add     %al, (%bx,%si)
74:  00 92 cf 00    add     %dl, 0xcf(%bp,%si)
78:  17      pop     %ss
79:  00 60 7c    add     %ah, 0x7c(%bx,%si)
7c:  00 00    add     %al, (%bx,%si)
7e:  ba f7 01    mov     $0x1f7,%dx
81:  00 00    add     %al, (%bx,%si)
83:  ec      in      (%dx),%al
84:  83 e0 c0    and     $0xffc0,%ax
87:  3c 40      cmp     $0x40,%al
89:  75 f8      jne     0x83
8b:  c3      ret
8c:  55      push    %bp
8d:  89 e5      mov     %sp,%bp
8f:  57      push    %di
90:  53      push    %bx
91:  8b 5d 0c    mov     0xc(%di),%bx
94:  e8 e5 ff    call    0x7c
97:  ff      (bad)
98:  ff      (bad)
99:  b8 01 00    mov     $0x1,%ax
9c:  00 00    add     %al, (%bx,%si)
9e:  ba f2 01    mov     $0x1f2,%dx
a1:  00 00    add     %al, (%bx,%si)
a3:  ee      out     %al, (%dx)
a4:  ba f3 01    mov     $0x1f3,%dx
a7:  00 00    add     %al, (%bx,%si)
a9:  89 d8      mov     %bx,%ax
ab:  ee      out     %al, (%dx)
ac:  89 d8      mov     %bx,%ax
ae:  c1 e8 08    shr     $0x8,%ax
b1:  ba f4 01    mov     $0x1f4,%dx
b4:  00 00    add     %al, (%bx,%si)
b6:  ee      out     %al, (%dx)
b7:  89 d8      mov     %bx,%ax
b9:  c1 e8 10    shr     $0x10,%ax
bc:  ba f5 01    mov     $0x1f5,%dx

```

bf:	00 00	add	%al, (%bx, %si)
c1:	ee	out	%al, (%dx)
c2:	89 d8	mov	%bx, %ax
c4:	c1 e8 18	shr	\$0x18, %ax
c7:	83 c8 e0	or	\$0xffe0, %ax
ca:	ba f6 01	mov	\$0x1f6, %dx
cd:	00 00	add	%al, (%bx, %si)
cf:	ee	out	%al, (%dx)
d0:	b8 20 00	mov	\$0x20, %ax
d3:	00 00	add	%al, (%bx, %si)
d5:	ba f7 01	mov	\$0x1f7, %dx
d8:	00 00	add	%al, (%bx, %si)
da:	ee	out	%al, (%dx)
db:	e8 9e ff	call	0x7c
de:	ff	(bad)	
df:	ff 8b 7d 08	decw	0x87d(%bp, %di)
e3:	b9 80 00	mov	\$0x80, %cx
e6:	00 00	add	%al, (%bx, %si)
e8:	ba f0 01	mov	\$0x1f0, %dx
eb:	00 00	add	%al, (%bx, %si)
ed:	fc	cld	
ee:	f3 6d	rep insw	(%dx), %es:(%di)
f0:	5b	pop	%bx
f1:	5f	pop	%di
f2:	5d	pop	%bp
f3:	c3	ret	
f4:	55	push	%bp
f5:	89 e5	mov	%sp, %bp
f7:	57	push	%di
f8:	56	push	%si
f9:	53	push	%bx
fa:	83 ec 0c	sub	\$0xc, %sp
fd:	8b 5d 08	mov	0x8(%di), %bx
100:	8b 75 10	mov	0x10(%di), %si
103:	89 df	mov	%bx, %di
105:	03 7d 0c	add	0xc(%di), %di
108:	89 f0	mov	%si, %ax
10a:	25 ff 01	and	\$0x1ff, %ax
10d:	00 00	add	%al, (%bx, %si)
10f:	29 c3	sub	%ax, %bx
111:	c1 ee 09	shr	\$0x9, %si

```

114: 83 c6 01      add     $0x1,%si
117: 39 df         cmp     %bx,%di
119: 76 1a         jbe     0x135
11b: 83 ec 08      sub     $0x8,%sp
11e: 56           push    %si
11f: 53           push    %bx
120: e8 67 ff      call   0x8a
123: ff           (bad)
124: ff 81 c3 00   incw    0xc3(%bx,%di)
128: 02 00         add     (%bx,%si),%al
12a: 00 83 c6 01   add     %al,0x1c6(%bp,%di)
12e: 83 c4 10      add     $0x10,%sp
131: 39 df         cmp     %bx,%di
133: 77 e6         ja      0x11b
135: 8d 65 f4      lea     -0xc(%di),%sp
138: 5b           pop     %bx
139: 5e           pop     %si
13a: 5f           pop     %di
13b: 5d           pop     %bp
13c: c3           ret
13d: 55           push    %bp
13e: 89 e5         mov     %sp,%bp
140: 57           push    %di
141: 56           push    %si
142: 53           push    %bx
143: 83 ec 10      sub     $0x10,%sp
146: 6a 00         push    $0x0
148: 68 00 10     push    $0x1000
14b: 00 00         add     %al,(%bx,%si)
14d: 68 00 00     push    $0x0
150: 01 00         add     %ax,(%bx,%si)
152: e8 9d ff      call   0xf2
155: ff           (bad)
156: ff 83 c4 10   incw    0x10c4(%bp,%di)
15a: 81 3d 00 00   cmpw    $0x0,(%di)
15e: 01 00         add     %ax,(%bx,%si)
160: 7f 45         jg      0x1a7
162: 4c           dec     %sp
163: 46           inc     %si
164: 75 21         jne     0x187
166: a1 1c 00     mov     0x1c,%ax
169: 01 00         add     %ax,(%bx,%si)

```

```

16f: 01 00      add    %ax, (%bx,%si)
171: 0f b7 35   movzww (%di), %si
174: 2c 00      sub    $0x0, %al
176: 01 00      add    %ax, (%bx,%si)
178: c1 e6 05   shl    $0x5, %si
17b: 01 de      add    %bx, %si
17d: 39 f3      cmp    %si, %bx
17f: 72 15      jb     0x196
181: ff 15      call   *(%di)
183: 18 00      sbb    %al, (%bx,%si)
185: 01 00      add    %ax, (%bx,%si)
187: 8d 65 f4   lea    -0xc(%di), %sp
18a: 5b        pop    %bx
18b: 5e        pop    %si
18c: 5f        pop    %di
18d: 5d        pop    %bp
18e: c3        ret
18f: 83 c3 20   add    $0x20, %bx
192: 39 de      cmp    %bx, %si
194: 76 eb      jbe    0x181
196: 8b 7b 0c   mov    0xc(%bp,%di), %di
199: 83 ec 04   sub    $0x4, %sp
19c: ff 73 04   push   0x4(%bp,%di)
19f: ff 73 10   push   0x10(%bp,%di)
1a2: 57        push   %di
1a3: e8 4c ff   call   0xf2
1a6: ff        (bad)
1a7: ff 8b 4b 14 decw   0x144b(%bp,%di)
1ab: 8b 43 10   mov    0x10(%bp,%di), %ax
1ae: 83 c4 10   add    $0x10, %sp
1b1: 39 c1      cmp    %ax, %cx
1b3: 76 da      jbe    0x18f
1b5: 01 c7      add    %ax, %di
1b7: 29 c1      sub    %ax, %cx
1b9: b8 00 00   mov    $0x0, %ax
1bc: 00 00      add    %al, (%bx,%si)
1be: fc        cld
1bf: f3 aa      rep stos %al, %es:(%di)
1c1: eb cc      jmp    0x18f
...
1fb: 00 00      add    %al, (%bx,%si)
1fd: 00 55 aa   add    %dl, -0x56(%di)

```

12. دلیل استفاده از دستور objcopy در حین makefile

این ابزار محتویات یک فایل object را دی یک فایل object دیگر کپی میکند. objcopy از کتابخانه BFD برای خواندن و نوشتن فایل object استفاده میکند و میتواند فایل مقصد را در فرمتی متفاوت از فرمت فایل اولیه بنویسد.

آپشن های این دستور که در makefile مربوط به xv6 استفاده شده اند را در زیر شرح داده ام:

- S- : در این حالت اطلاعات relocation و symbol information را از فایل مبدا کپی نمی کند.

- O- : فایل خروجی را در فرمت گفته شده با این دستور درست میکند.
- J- : با این آپشن میتوان فقط قسمت های نشان داده شده را در فایل مقصد کپی کرد.

در makefile مربوط به xv6 قسمتی هایی که از objcopy استفاده کرده ایم را بطور خلاصه در پایین شرح میدهم:

```
bootblock: bootasm.S bootmain.c
$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
$(OBJDUMP) -S bootblock.o > bootblock.asm
$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
```

در این بخش پس از لینک کردن bootasm.S و bootmain.c در فایل bootblock.o محتویات text. فایل bootblock.o را در یک فایل با فرمت raw binary به نام bootblock کپی میکند.

```
entryother: entryother.S
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c entryother.S
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o
$(OBJCOPY) -S -O binary -j .text bootblockother.o entryother
$(OBJDUMP) -S bootblockother.o > entryother.asm
```

در این بخش محتویات text. فایل bootblockother.o در یک فایل با فرمت raw binary با نام entryother کپی میکند.

```
initcode: initcode.S
$(CC) $(CFLAGS) -nostdinc -I. -c initcode.S
$(LD) $(LDFLAGS) -N -e start -Ttext 0 -o initcode.out initcode.o
$(OBJCOPY) -S -O binary initcode.out initcode
$(OBJDUMP) -S initcode.o > initcode.asm
```

در این بخش هم محتویات فایل initcode.out در یک فایل با فرمت raw binary به نام initcode کپی میشود.

14. وظیفه ثبات های x86

- ثبات های عام منظوره: 8 ثبات عام منظوره در معماری x86 وجود دارد. نام این ثبات ها در پردازنده های 8 بیتی، 16 بیتی، 32 بیتی و 64 بیتی متفاوت است. برای مثال یکی از این ثبات های عام منظوره، ثبات اشاره گر به پشته می باشد. که به اختصار با SP نمایش داده می شود. حال اگر قبل از نام آنها، R قرار دهیم، یعنی RSP نشان دهنده یک اشاره گر 64 بیتی به پشته می باشد که مخصوص پردازنده های 64 بیتی می باشد. به طور مشابه اگر قبل از آنها E قرار دهیم، برای مثال ESP، این اشاره گر 32 بیتی و اگر بعد از آنها L قرار دهیم، برای مثال SPL، این اشاره گر 8 بیتی خواهد بود. وظیفه ی اشاره گر به پشته این است که به خانه ای در پشته اشاره کند که در زیر آن اطلاعات ذخیره شده ی قبلی قرار دارند و در صورتی که بخواهیم داده ی دیگری در پشته وارد کنیم، آن را در همین خانه ای که اشاره گر به آن اشاره می کند ذخیره کنیم و اشاره گر پس از ذخیره سازی، به خانه ی بالایی اشاره کند. انواع دیگر GDP ها را در شکل زیر مشاهده می کنیم:

Register	Accumulator		Counter		Data		Base		Stack Pointer		Stack Base Pointer		Source		Destination	
64-bit	RAX		RCX		RDX		RBX		RSP		RBP		RSI		RDI	
32-bit	EAX		ECX		EDX		EBX		ESP		EBP		ESI		EDI	
16-bit	AX		CX		DX		BX		SP		BP		SI		DI	
8-bit	AH	AL	CH	CL	DH	DL	BH	BL	SPL		BPL		SIL		DIL	

- ثبات های قطعه: 6 ثبات عام منظوره در x86 وجود دارد. یکی از آنها، Code Segment(CS) می باشد که وظیفه آن تعیین محل حافظه کد در حال اجرا می باشد.
- ثبات وضعیت: ثبات FLAGS یا همان ثبات وضعیت، نشان دهنده حالت فعلی پردازنده می باشد. هر بیت از این ثبات، نشان دهنده ی یک پرچم می باشد. این پرچم ها نشان دهنده وضعیت اعمال محاسباتی و منطقی یا محدودیت های اعمالی بر عملیات فعلی پردازنده می باشد. ثبات وضعیت برای پردازنده x86 به شکل زیر می باشد:

Intel x86 FLAGS register ^[1]						
Bit #	Mask	Abbreviation	Description	Category	=1	=0
FLAGS						
0	0x0001	CF	Carry flag	Status	CY(Carry)	NC(No Carry)
1	0x0002	—	Reserved, always 1 in EFLAGS ^{[2][3]}	—		
2	0x0004	PF	Parity flag	Status	PE(Parity Even)	PO(Parity Odd)
3	0x0008	—	Reserved ^[3]	—		
4	0x0010	AF	Auxiliary Carry flag ^[4]	Status	AC(Auxiliary Carry)	NA(No Auxiliary Carry)
5	0x0020	—	Reserved ^[3]	—		
6	0x0040	ZF	Zero flag	Status	ZR(Zero)	NZ(Not Zero)
7	0x0080	SF	Sign flag	Status	NG(Negative)	PL(Positive)
8	0x0100	TF	Trap flag (single step)	Control		
9	0x0200	IF	Interrupt enable flag	Control	EI(Enable Interrupt)	DI(Disable Interrupt)
10	0x0400	DF	Direction flag	Control	DN(Down)	UP(Up)
11	0x0800	OF	Overflow flag	Status	OV(Overflow)	NV(Not Overflow)
12-13	0x3000	IOPL	I/O privilege level (286+ only), always all-1s on 8086 and 186	System		
14	0x4000	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System		
15	0x8000	MD	Mode flag (NEC V-series only), ^[5] reserved on all Intel CPUs. Always 1 on 8086/186, 0 on 286 and later.	Control	(NEC only) Native Mode (186 compatible)	(NEC only) Emulation Mode (8080 compatible)

- **ثبات کنترلی:** ثبات کنترلی رفتار کلی CPU یا دیگر دستگاه های دیجیتال را تغییر می دهد. از وظایف مهم این ثبات های کنترلی، می توان به paging control و interrupt control اشاره کرد. یکی از این ثبات ها، CR0 می باشد که در پردازنده های 32 بیتی مانند i386 و بالاتر استفاده شده است بیت های این ثبات، نشان دهنده ی تغییرات و کنترل های مختلفی در رفتار کلی پردازنده هستند که به شکل زیر می باشد:

Bit	Label	Description
0	PE	Protected Mode Enable
1	MP	Monitor co-processor
2	EM	x87 FPU Emulation
3	TS	Task switched
4	ET	Extension type
5	NE	Numeric error
16	WP	Write protect
18	AM	Alignment mask
29	NW	Not-write through
30	CD	Cache disable
31	PG	Paging

18. کد معادل entry.S در هسته لینوکس

کد معادل entry.S در هسته لینوکس را در [این لینک](#) می توانید مشاهده کنید. که در گیت هاب لینوکس در فولدر x86 می باشد. کد نسخه ی ۳۲ بیتی و ۶۴ بیتی جدا می باشند.

19. دلیل استفاده قراردادن page table در آدرس فیزیکی

کاربرد این جدول این است که ما برای تبدیل آدرس مجازی به فیزیکی از این جدول استفاده میکنیم و به کمک آن آدرس مجازی را به آدرس فیزیکی اش نگاشت میکنیم. طبیعتا اگر این جدول حافظه مجازی داشته باشد برای دسترسی به حافظه فیزیکی آن باز به یک جدول دیگر برای نگاشت نیاز داریم و خب در یک حلقه بی پایان گیر میکنیم که بالاخره باید یک جدول آدرس فیزیکی داشته باشد تا مشکلمان حل شود. پس بهترین راه حل این است که page table آدرس فیزیکی داشته باشد.

22. دلیل تعریف پرچم SEG_USER برای کد و داده های سطح کاربر

تمام قطعه های کاربر و هسته، به حافظه دسترسی دارند. هر یک از این قطعه ها، با یک descriptor که در Global Descriptor Table که شامل اطلاعاتی مانند آدرس شروع قطعه، اندازه قطعه و سطح دسترسی قطعه می باشد. برای خواندن یک دستورالعمل، ابتدا قطعه ی آن از طریق descriptor آن پیدا می شود و سپس صفحه ی مربوط به آن پیدا می شود. سپس آدرس منطقی به آدرس فیزیکی تبدیل شده و دستور از حافظه خوانده و اجرا می شود. حال در حین اجرای دستورالعمل، یک سطح دسترسی فعلی داریم که از روی سطح دسترسی دیسکریپتور که در GDP بود مشخص می شود. حال با اینکه ممکن است دیسکریپتور این دستورالعمل ها، به قطعات یکسانی از حافظه اشاره کنند، سطح دسترسی متفاوتی داشته باشند. برای مثال دستور IN وظیفه خواندن یک بیت از پورت را دارد و این نیازمند است که سطح دسترسی ورودی/خروجی که برابر 0 است را داشته باشد. حال اگر کاربر بخواهد این دستور را اجرا کند، با اینکه به این قطعه از حافظه دسترسی دارد اما سطح دسترسی DPL_USER را دارد که پایین ترین سطح دسترسی است (مقدار 3 دارد) و نمیتواند از دستور IN استفاده کند. که این SEG_USER همان flag ای است که مشخص می کند سطح دسترسی در این دستورالعمل، سطح دسترسی کاربر است یا خیر. بنابراین کاربر نمیتواند هر دستوری را اجرا کند.

23. اجزای struct proc و معادل آن در لینوکس

این struct در فایل proc.h تعریف شده است، که برای ذخیره وضعیت هر پردازش استفاده می شود. این struct، متغیرهای زیر را دارد:

1. uint sz: اندازه حافظه گرفته شده توسط این پردازش به واحد بایت
2. pde_t* pgdir: یک پوینتر به page table پردازش
3. char* kstack: پوینتر به kernel stack
4. enum procstate state: وضعیت پردازش را مشخص میکند که میتواند به یکی از حالت های UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE باشد.
5. int pid: این عدد یک، process ID می باشد که بین همه ی پردازش ها یکتاست.
6. struct proc* parent: پوینتر به پردازش ی پدر
7. struct trapframe* tf: پوینتر به trapframe برای ذخیره وضعیت اجرای برنامه در حین فراخوانی syscall
8. struct context* context: پوینتر به context است که مقادیر رجیسترهای مورد نیاز برای context switching را نگه می دارد. کاربرد این متغیر، در هنگام استفاده از تابع swtch است که با استفاده از آن میتوان به یک پردازش دیگر switch کرد.
9. void* chan: در صورتی که مقدار غیر 0 داشته باشد، یعنی روی چنل chan در حال sleeping است.
10. int killed: اگر 0 نباشد، یعنی این پردازش kill شده است.
11. struct file* ofile[NOFILE]: آرایه ای از پوینترها به فایل های باز شده توسط پردازش است.
12. struct node* cwd: این متغیر current working directory را مشخص می کند.
13. char name[16]: نام پردازش را نگه می دارد که برای اشکال زدایی، کاربرد دارد.

معادل این struct در هسته لینوکس را در [این لینک](#) می توانید مشاهده کنید. (که نام آن task_struct می باشد)

27. بخش هایی از آماده سازی سیستم که بین تمامی هسته های پردازنده

مشترک هستند و بخش هایی که اختصاصی هستند؟ زمان بند روی کدام

هسته اجرا میشود؟

هسته اول که وظیفه بوت را انجام میدهد طبق توضیحات توسط کد entry.s وارد تابع main میشود پس تمامی توابعی که در main هستند بصورت اختصاصی بر روی هسته اول اجرا میشوند

اما تعدادی از آنها بر روی هسته های دیگر هم اجرا میشوند که در ادامه توضیح می دهیم. با توجه به توضیحات در خود کد پردازنده , هسته های دیگر از طریق کد S.other.entry وارد تابع mpenter میشوند و توابعی که در این تابع است بر روی آنها اجرا میشود. این توابع که چهار تا هستند با توابعی که در main بودند مشترک هستند به اینگونه که سه تا از آنها بطور مستقیم در main حضور دارند و تابع switchkvm بصورت غیر مستقیم در تابع kvmalloc که در main است بکار رفته است:

```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
    kpgdir = setupkvm();
    switchkvm();
}
```

طبق کامنت خود کد خط اول یک page table برای کرنل ایجاد میکند که مربوط به هسته اول است و در خط بعد به این page table سوییچ میکند که در تمام هسته ها نیاز است. بخش هایی از آماده سازی سیستم که اختصاصی هسته اول هستند:

```
17 int
18 main(void)
19 {
20     kinit1(end, P2V(4*1024*1024)); // phys page allocator
21     kvmalloc(); // kernel page table
22     mpinit(); // detect other processors
23     lapicinit(); // interrupt controller
24     seginit(); // segment descriptors
25     picinit(); // disable pic
26     ioapicinit(); // another interrupt controller
27     consoleinit(); // console hardware
28     uartinit(); // serial port
29     pinit(); // process table
30     tvinit(); // trap vectors
31     binit(); // buffer cache
32     fileinit(); // file table
33     ideinit(); // disk
34     startothers(); // start other processors
35     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
36     userinit(); // first user process
37     mpmain(); // finish this processor's setup
38 }
```

بغیر از خطوط ۲۳/۲۴/۳۷ و قسمت switchkvm در kvmalloc باقی قسمت ها اختصاصی هسته اول هستند.

بخش هایی از آماده سازی سیستم که بین تمام هسته ها مشترک هستند:

```
// Other CPUs jump here from entryother.S.
static void
mpenter(void)
{
    switchkvm();
    seginit();
    lapicinit();
    mpmain();
}
```

از بخش های اختصاصی بطور مثال تابع startothers طبق توضیحات خط ۳۴ بقیه پردازنده ها را استارت میکند که این کار فقط مختص پردازنده اول است و نیازی نیست هسته های دیگر آن را داشته باشند.

از قسمت های مشترک هم میتوان به mpmain اشاره کرد که همه پردازنده ها نیاز دارند setup شده و کار خود را شروع کنند. پس این تابع در همه آنها فراخوانی میشود. مطابق عکس زیر این تابع در mpmain صدا زده میشود که در تمام پردازنده ها مشترک است و هر تابع زمان بند مختص به خودش را خواهد داشت.

```
// Common CPU setup code.
static void
mpmain(void)
{
    cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
    idtinit();           // load idt register
    xchg(&(mycpu()->started), 1); // tell startothers() we're up
    scheduler();         // start running processes
}
```

اشکال زدایی

با استفاده از دستورات ارائه شده در صورت پروژه به kernel دیباگر gdb وصل می شویم :

```
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000ffff in ?? ()
```

حال برای دسترسی به cat.c از دستور _cat file استفاده می کنیم.

روند اجرای GDB

1. برای مشاهده Breakpoint ها از چه دستوری استفاده میشود؟

برای مشاهده breakpoint ها می توان از دستور breakpoints info استفاده کرد

```
(gdb) b 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) |
```

در مثال بالا ما در خط 12 فایل cat.c یک breakpoint داریم.

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

برای حذف یک breakpoint می توان از دستور <breakpoint number> del استفاده کرد.

مقدار breakpoint number را از دستور قبلی استخراج می کنیم.

```
(gdb) b 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) del 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb) |
```

کنترل روند اجرا و دسترسی به حالت سیستم

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

دستور bt که مخفف backtrace است stack call برنامه در لحظه کنونی را نشان می دهد. هر تابع که صدا زده می شود یک frame stack مخصوص به خودش را می گیرد که متغیرهای محلی و آدرس بازگشت و غیره در آن ذخیره می شود. خروجی این دستور در هر خط یک frame stack را نشان می دهد که به ترتیب از درونی ترین به بیرونی ترین است. می توان به این دستور مقدار n را داد که n بیرونی ترین لایه ها را نشان دهد. می توان به این دستور مقدار -n را داد که n درونی ترین لایه ها را نشان دهد.

```
(gdb) bt
#0 0x0000ffff in ?? ()
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, cat (fd=0) at cat.c:12
12 while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) bt
#0 cat (fd=0) at cat.c:12
#1 0xffffffff in ?? ()
```

در مثال بالا چون تنها تابع cat صدا شده و برنامه متوقف شده در stack تنها یک کال از cat داریم.

4. دو تفاوت دستورهای x و print را توضیح دهید. چگونه میتوان محتوای

یک ثبات خاص را چاپ کرد؟

در دستور print مقدار متغیری که به عنوان آرگومان به آن داده شده نمایش داده می شود.

```
Thread 1 hit Breakpoint 2, cat (fd=0) at cat.c:12
12 while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) bt
#0 cat (fd=0) at cat.c:12
#1 0xffffffff in ?? ()
(gdb) print fd
$1 = 0
```

در دستور x مقدار خانه ای از حافظه که به عنوان آرگومان به آن داده شده نمایش داده می شود.

```
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=0) at cat.c:12
12 while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) print &n
Can't take address of "n" which isn't an lvalue.
(gdb) bt
#0 cat (fd=0) at cat.c:12
#1 0xffffffff in ?? ()
(gdb) print &fd
$1 = (int *) 0x2fe0
(gdb) x 0x2fe0
0x2fe0: 0x00000000
```

همانطور که مشاهده می شود خانه ی متغیر fd در حافظه مقدار 0 را دارد.

در دستور info registers مقدار رجیستری که به عنوان آرگومان به آن داده شده نمایش داده می شود.

5. برای نمایش وضعیت ثباتها از چه دستوری استفاده میشود؟ متغیرها محلی چطور؟ نتیجه این دستور را در گزارش کار خود بیاورید. همچنین در گزارش خود توضیح دهید که در معماری x86 رجیسترهای edi و esi نشانگر چه چیزی هستند؟

با استفاده از دستور registers info می توان وضعیت ثباتها را مشاهده کرد.

```
(gdb) info registers
eax      0x1          1
ecx      0xa         10
edx      0x0          0
ebx      0x892        2194
esp      0x2fc0       0x2fc0
ebp      0x2fd8       0x2fd8
esi      0x0          0
edi      0x0          0
eip      0x97         0x97 <cat+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
fs_gs_base 0x0        0
cr0      0x80010011   [ PG WP ET PE ]
cr2      0x0          0
cr3      0xdfbb000    [ PDBR=57275 PCID=0 ]
cr4      0x10         [ PSE ]
cr8      0x0          0
xfer     0x0          [ ]
xmm0     {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm1     {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
--Type <RET> for more, q to quit, c to continue without paging--
```

با استفاده از دستور info locals می توان وضعیت متغیر های محلی را مشاهده کرد.

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=0) at cat.c:12
12      while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) info locals
n = <optimized out>
(gdb) |
```

رجیسترهای esi و edi دو مورد از ثباتهای همه منظوره موجود در معماری هستند. آنها رجیستر های 32 بیتی هستند.

esi: اساساً به عنوان source index برای عملیات رشته ها مانند کپی کردن یا مقایسه رشته ها استفاده می شود. همچنین معمولاً به عنوان نشانگر داده های منبع در عملیات data manipulation استفاده می شود.

edi: به عنوان destination index برای عملیات رشته ای عمل می کند که در آن داده ها نوشته یا اصلاح می شوند. مشابه esi , edi اغلب به عنوان یک اشاره گر به داده های مقصد یا آدرس مقصد برای دستورات data manipulation استفاده می شود.

هر دو esi و edi معمولاً همراه با سایر ثبات ها برای عملیات رشته ای مانند lods، cmps، movs و stos استفاده می شوند.

6. ساختار input struct

این استراکچر در فایل console.c تعریف شده است و برای command line کنسول سیستم عامل استفاده می شود.

با دستور ptype input به محتویات این متغیر دسترسی می یابیم

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

آرایه buf محل ذخیره خط ورودی است که اندازه آن حداکثر 128 کاراکتر است.

متغیر w محل شروع نوشتن خط ورودی کنونی در buf است.

متغیر e محل کنونی کرسر در خط ورودی است. (در طراحی جدید ما انتهای خط را نشان میدهد)

متغیر r برای خواندن buf استفاده می شود.

متغیر pointer که توسط ما افزوده شده موقعیت کرسر را نشان میدهد.

تغییر مقادیر

برای w زمانی که دستوری ثبت شود (اینتر بخورد) به انتهای بافر یا همان خطی که اجرا شده می رود.

برای e زمانی که حرفی پاک شود یکی به عقب میاید و وقتی حرفی نوشته شود به جلو میرود.

برای r وقتی دستوری اجرا می شود او از مکان خود به انتهای دستور میاید .

برای pointer همه چیز شبیه e است و همچنین وقتی کنترل بی بخورد به عقب میاید و با کنترل اف به جلو می رود.

مثال

```
(gdb) print input
$1 = {buf = '\000' <repeats 127 times>, r = 0, w = 0, e = 0, pointer = 0}
(gdb) |
```

استیت اولیه

```
(gdb) print input
$1 = {buf = "test", '\000' <repeats 123 times>, r = 0, w = 0, e = 4, pointer = 4}
(gdb) |
```

تایپ کلمه test

```
(gdb) print input
$2 = {buf = "test\n", '\000' <repeats 122 times>, r = 5, w = 5, e = 5, pointer = 5}
(gdb) |
```

اجرای کلمه test

```
(gdb) print input
$3 = {buf = "test\nls", '\000' <repeats 120 times>, r = 5, w = 5, e = 7, pointer = 7}
(gdb) |
```

تایپ دستور ls

```
(gdb) print input
$2 = {buf = "test\nls", '\000' <repeats 120 times>, r = 5, w = 5, e = 6, pointer = 6}
(gdb) |
```

حذف کاراکتر s (توجه شود که s از بافر حذف نشد و فقط e عقب آمد)

اشکال زدایی در سطح کد اسمبلی

7. خروجی دستورهای layout src و layout asm در TUI چیست؟
دستور layout src کد سورس برنامه در حال اجرا را نشان میدهد.


```

cat.c
7 void
8 cat(int fd)
9 {
10     int n;
11
12     while((n = read(fd, buf, sizeof(buf))) > 0) {
13         if (write(1, buf, n) != n) {
14             printf(1, "cat: write error\n");
15             exit();
16         }
17     }
18     if(n < 0){
19         printf(1, "cat: read error\n");
20         exit();
21     }
22 }
23
remote Thread 1.1 In: cat
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=0) at cat.c:12
(gdb) layout src
(gdb) |

```

دستور layout asm کد اسمبلی برنامه در حال اجرا را نشان میدهد.

```

0x82 <main+130> push    $0x1
0x84 <main+132> call    0x4f0 <printf+32>
0x89 <main+137> pop     %ecx
0x8a <main+138> pop     %ebx
0x8b <main+139> push    $0x8a6
0x90 <cat>      push    $0x1
0x92 <cat+2>    call    0x4f0 <printf+32>
B+> 0x97 <cat+7> call    0x38b <read>
0x9c <cat+12>  add     $0x10,%esp
0x9f <cat+15>  mov     %eax,%ebx
0xa1 <cat+17>  test    %eax,%eax
0xa3 <cat+19>  js      0xd1 <cat+65>
0xa5 <cat+21>  je      0xe4 <cat+84>
0xa7 <cat+23>  lea     0x0(%esi,%eiz,1),%esi
0xae <cat+30>  xchg    %ax,%ax
0xb0 <cat+32>  call    0x39b <close>
0xb5 <cat+37>  test    %eax,%eax

remote Thread 1.1 In: cat
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=0) at cat.c:12
(gdb) layout src
(gdb) layout asm
(gdb) |

```

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

برای وضعیت مشاهده پشته فراخوانی فعلی می توان از دستور where استفاده کرد.

```
console.c
349 }
350 }
351
352 void consoleintr(int (*getc)(void))
353 {
B+ 354     int c, doprocdump = 0;
355
> 356     acquire(&cons.lock);
357     while ((c = getc()) >= 0)
358     {
359         switch (c)
360         {
361             case C('P'): // Process listing.
362                 // procdump() locks cons.lock indirectly; invoke later
363                 doprocdump = 1;
364                 break;
365             case C('U'): // Kill line.
```

```
remote Thread 1.1 In: consoleintr
#0  consoleintr (getc=0x801064c0 <uartgetc>) at console.c:356
#1  0x80106644 in uartintr () at uart.c:76
#2  0x801063e6 in trap (tf=0x8010cac8 <stack+3912>) at trap.c:71
#3  0x801061a3 in alltraps () at trapasm.S:20
#4  0x8010cac8 in stack ()
#5  0x80113ca4 in cpus ()
#6  0x80113ca0 in ?? ()
#7  0x8010393f in mpmain () at main.c:57
#8  0x80103a8c in main () at main.c:37
(gdb) |
```

حال با استفاده از دو دستور up و down میتوان به تابع های بالاتر یا پایین تر رفت . آرگومان این دو تابع میزان جابجایی است که به صورت دیفالت 1 است.

در مثال پایین ما دو تابع بالا رفته ایم و وارد فایل trapasm شده ایم و در بخشی هستیم که تابع بعدی فراخوانی شده است.

```

-trapasm.S
13  # Set up data segments.
14  movw $(SEG_KDATA<<3), %ax
15  movw %ax, %ds
16  movw %ax, %es
17
18  # Call trap(tf), where tf=%esp
19  pushl %esp
> 20  call trap
21  addl $4, %esp
22
23  # Return falls through to trapret...
24  .globl trapret
25  trapret:
26  popal
27  popl %gs
28  popl %fs
29  popl %es

remote Thread 1.1 In: alltraps
#2  0x801063e6 in trap (tf=0x8010cac8 <stack+3912>) at trap.c:71
#3  0x801061a3 in alltraps () at trapasm.S:20
#4  0x8010cac8 in stack ()
#5  0x80113ca4 in cpus ()
#6  0x80113ca0 in ?? ()
#7  0x8010393f in mpmain () at main.c:57
#8  0x80103a8c in main () at main.c:37
(gdb) up 3
#3  0x801061a3 in alltraps () at trapasm.S:20
(gdb) |

```

بخش امتیازی (پیکربندی و ساخت هسته لینوکس):

بر روی vmware خود که ubuntu 22.04 روی آن نصب است ، دستورات گفته شده را اجرا میکنیم و هسته را نصب میکنیم .

با استفاده از دستور `uname -a` ورژن هسته لینوکس در این نسخه را متوجه میشویم که 15.5.0 است :

```

soheil@soheil:~$ uname -a
Linux soheil 5.15.0-87-generic #97~20.04.1-Ubuntu SMP Thu Oct 5 08:25:28 UTC 20
23 x86_64 x86_64 x86_64 GNU/Linux

```

برای پیاده سازی دستور گفته شده و نمایش نام اعضای گروه با دستور `dmesg` یک فایل به نام `group31.c` به همراه یک `Makefile` مربوط به آن میسازیم که کد آنها را در زیر قرار داده ام.

:group31.c

```
#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE("GPL");
int init_module(void)
{
    printk("Group 31:\n- Soheil Hajian Manesh :810100119\n- AmirAli Rahimi : 810100146\n-
        Mahdi Nouri: 810100231\n");
    return 0;
}
void cleanup_module(void) {}
|
```

:Makefile

```
obj-m += group31.o
all:
    make -C /lib/modules/$(uname -r)/build M=$(PWD) modules
```

در نهایت در همان آدرسی که Makefile قرار دارد دستور make را وارد میکنیم و فایل group31.ko ساخته میشود. سپس دستور sudo insmod group31.ko را وارد کرده و در نهایت با ورود دستور dmesg میتوانیم خروجی که نام اعضای گروه است را مشاهده کنیم :

```
[ 5269.816499] Group 31:
- Soheil Hajian Manesh :810100119
- AmirAli Rahimi : 810100146
- Mahdi Nouri: 810100231
soheil@soheil:~/Desktop/OS-entiazi$
```