

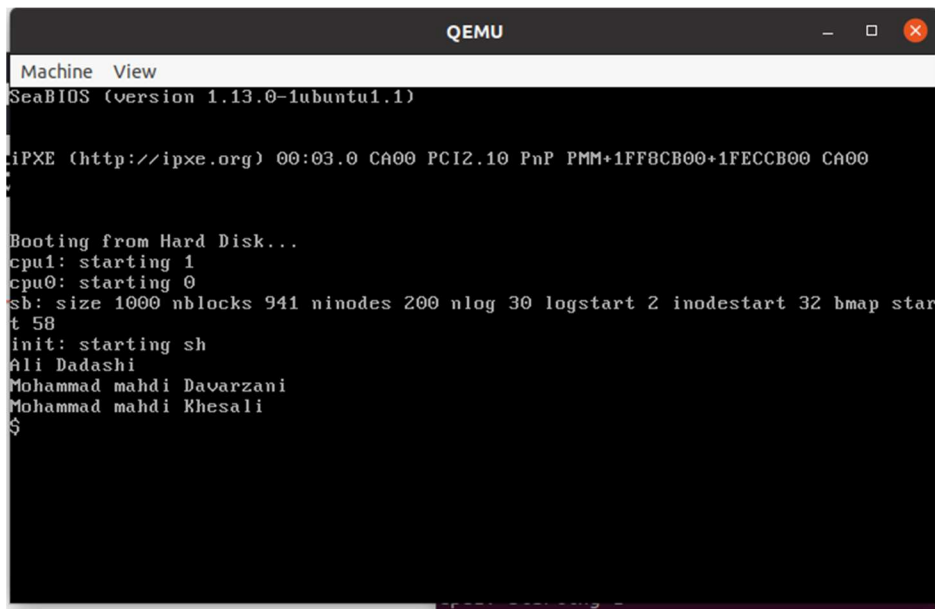
# OS\_Lab1

علی داداشی

محمد مهدی خصالی

محمد مهدی داورزنی

اسامی اعضا گروه:



```
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 start 58
init: starting sh
Ali Dadashi
Mohammad mahdi Davarzani
Mohammad mahdi Khesali
$
```

```
ma hdi@Linux: ~/OS/OS-Lab1/xv6-public
SeaBIOS (version 1.13.0-1ubuntu1.1)

ck ipXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1F0


Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodes8
init: starting sh
Ali Dadashi
Mohammad mahdi Davarzani
Mohammadd mahdi Khesali
$
```

نشان دادن دستورات قبلی:

```
rt
hi Machine View
macpu1: starting 1
macpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodes8
t 58
init: starting sh
Ali Dadashi
Mohammad mahdi Davarzani
Mohammad mahdi Khesali
$ commnad0
2 exec: fail
l exec commnad0 failed
an$ command1
3 exec: fail
l exec command1 failed
an$ command2
4 exec: fail
l exec command2 failed
l $ command3
an exec: fail
l exec command3 failed
$ command4
exec: fail
l exec command4 failed
$ command2_

m.d 27 sum += (STD_NUM3 % 100) / 10 + (STD_NUM3 % 100) % 10;
m.o 28
```

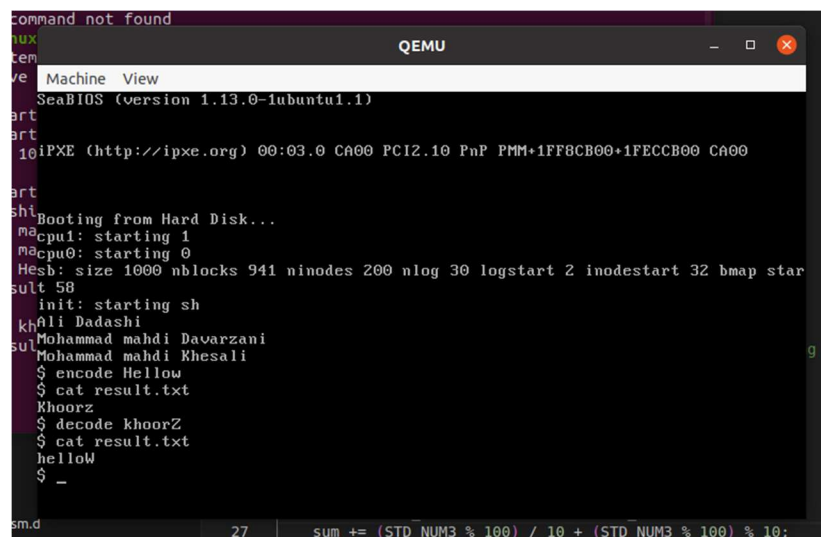
## اجرای رمزگذاری سزار:



```
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
Hesb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
ult 50
init: starting sh
Ali Dadashi
Mohammad mahdi Davarzani
Mohammad mahdi Khesali
$ encode Hellow
$ cat result.txt
Khoorz
$
```



```
Command not found
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
Hesb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
ult 50
init: starting sh
Ali Dadashi
Mohammad mahdi Davarzani
Mohammad mahdi Khesali
$ encode Hellow
$ cat result.txt
Khoorz
$ decode KhoorZ
$ cat result.txt
hellow
$ _
```

### 1- سه وظیفه اصلی سیستم عامل را نام ببرید.

- دیریت منابع سخت افزاری
- مدیریت و پاسخ به برنامه های کاربر و اپلیکیشن ها
- ارتباط سخت افزار با نرم افزار

## 2- فایل های اصلی سیستم عامل xv6:

- سیستم کال ها (System Calls): این مجموعه فایل ها شامل هندلر سیستم کال است و توابع کرنل مرتبط با هر کال را پیاده سازی می کند. سیستم کال ها به برنامه های کاربری اجازه می دهند تا عملیات های خاصی مانند مدیریت فایل و پردازش را انجام دهند
- هدرهای پایه (Basic Headers): این فایل ها انواع داده های پایه، ثابت ها، و پروتوتایپ های تابع را که به طور مداوم در سراسر کدهای xv6 استفاده می شوند، تعریف می کنند.
- پایپ ها (Pipes): این فایل ها مکانیزم پایپ را برای ارتباط بین پردازشی پیاده سازی می کنند و به پردازش ها امکان می دهند تا یک بافر مشترک را به اشتراک بگذارند تا یکی داده ای را بنویسد و دیگری آن را بخواند.
- بوت لودر (Bootloader): این فایل ها برای بوت کردن xv6 حیاتی هستند و شامل کدی هستند که کرنل را در حافظه بارگذاری کرده و اجرای آن را آغاز می کنند.
- پردازش ها (Processes): این فایل ها فعالیت های پردازش ها را در xv6 مدیریت کرده و توابعی را برای ایجاد، زمان بندی، و جابجایی بین پردازش ها، همچنین بارگذاری و اجرای برنامه ها پیاده سازی می کنند.
- سیستم فایل (File System): این گروه از فایل ها لایه ی سیستم فایل در xv6 را پیاده سازی می کنند و مدیریت فایل ها، دایرکتوری ها، و عملیات I/O دیسک را انجام می دهند. آن ها وظیفه خواندن و نوشتن فایل ها، ایجاد و حذف فایل ها، و پیمایش دایرکتوری ها را بر عهده دارند.
- چیدمان حافظه (Memory Layout): این فایل ساختار حافظه ی کرنل را تشریح کرده و مشخص می کند که فایل های آبجکت چگونه باید در طی فرآیند کامپایل به هم متصل شوند.

- سخت‌افزار سطح پایین (Low-Level Hardware): این فایل‌ها برای تعامل مستقیم با سخت‌افزار طراحی شده‌اند و مدیریت عملیات I/O در سطح بلوک دیسک، فراهم کردن درایورهای رابط دیسک، و نظارت بر ورودی و خروجی کنسول را انجام می‌دهند.
- عملیات رشته‌ای (String Operations): این فایل‌ها توابع کمکی برای مدیریت رشته‌ها فراهم می‌کنند و عملیات‌هایی مانند کپی کردن، به هم چسباندن، و مقایسه‌ی رشته‌ها را پوشش می‌دهند.
- قفل‌ها (Locks): این فایل‌ها مکانیزم‌های همزمان‌سازی مانند اسپین‌لاک (spinlocks) و اسلیپ‌لاک (sleeplocks) را ارائه می‌دهند که دسترسی ایمن به منابع مشترک بین چندین ترد یا پردازش را تضمین می‌کنند

### 3- دستور Make-n را اجرا کنید. کدام دستور، فایل نهایی هسته را می‌سازد؟

دستور `make -n` در زمینه ساخت نرم‌افزارهایی مانند QEMU یا هر پروژه‌ای که از `make` استفاده می‌کند، برای انجام شبیه‌سازی اجرای دستور به کار می‌رود. این دستور، فرمان‌هایی را که در طول فرآیند ساخت اجرا می‌شوند نمایش می‌دهد، بدون این که آنها را واقعاً اجرا کند. این قابلیت به خصوص برای اشکال‌زدایی یا فهمیدن این که `make` چه کارهایی انجام خواهد داد، بدون این که تغییری در فایل‌ها ایجاد شود یا ساخت واقعی انجام شود، مفید است.

به عنوان مثال، اجرای `make -n` در فرآیند ساخت QEMU، تمام دستورهای کامپایل، لینک کردن و سایر مراحل که در صورت اجرای `make` عادی اجرا می‌شدند، نمایش می‌دهد؛ اما در واقعیت هیچ کدام از فایل‌ها کامپایل یا لینک نمی‌شوند.

### 4- کاربرد UPROGS و USIB؟

متغیر **UPROGS**: این متغیر شامل لیستی از برنامه‌های کاربری است که در طی ساخت و کامپایل

xv6 توسط سیستم‌عامل، کامپایل و به فایل‌های اجرایی تبدیل می‌شوند. نام هر برنامه در این لیست به

صورت `file_name_`` نوشته می‌شود. تمام نام‌هایی که با ```_`` شروع می‌شوند (مثل `file_name_``) یک هدف در فایل `Makefile` ایجاد می‌کنند که شامل پیش‌نیازهای فایل‌های آبجکت (`file_name.o``) و متغیر `ULIB`` هستند. بنابراین، اهدافی که در `UPROGS`` قرار دارند منجر به ایجاد فایل‌های آبجکت برای برنامه‌های کاربری، اجرای اهداف مرتبط با `ULIB`` و در نهایت اجرای فرمان `ld`` می‌شوند. فرمان `ld`` برای لینک کردن فایل‌های مورد نیاز و تولید یک فایل اجرایی استفاده می‌شود. همچنین، فایل‌های آبجکت مربوط به هر برنامه (`file_name.o``) توسط یک قاعده داخلی در **Makefile** ایجاد می‌شوند و به صورت صریح در `Makefile` نوشته نشده‌اند.

### **UPROGS -> user programs**

متغیر `ULIB`: این متغیر شامل چندین کتابخانه زبان `C` است. بسیاری از کدهای `xv6` از توابع این کتابخانه‌ها استفاده می‌کنند و اجرای آن‌ها نیاز به کامپایل این فایل‌ها دارد. به عنوان مثال، برنامه‌های سطح کاربری نیاز به کامپایل فایل‌های `ULIB`` دارند. بنابراین، همانطور که در بخش قبل ذکر شد، فایل‌های `ULIB`` به عنوان پیش‌نیاز در قوانین نوشته شده و در نهایت توسط فرمان `ld`` به فایل‌های اجرایی لینک می‌شوند. فایل‌های `ULIB`` شامل توابعی مثل `printf``، `strcmp``، `strcpy``، `malloc`` و غیره هستند.

### **ULIB -> user libraries**

```

rt
nt Machine View
ma
cpu1: starting 1
macpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
na
init: starting sh
Ali Dadashi
Mohammad mahdi Davarzani
Mohammad mahdi Khesali
an$ commnad0
2 exec: fail
l exec commnad0 failed
an$ command1
3 exec: fail
l exec command1 failed
an$ command2
4 exec: fail
l exec command2 failed
l $ command3
an$ command3
exec: fail
exec command3 failed
$ command4
exec: fail
exec command4 failed
$ command2_
m.d 27 sum += (STD_NUM3 % 100) / 10 + (STD_NUM3 % 100) % 10;
m.o 28

```

ه، `UPROGS` مخفف \*User Programs\* و `ULIB` مخفف \*User Libraries\* است

که به ترتیب نمایانگر برنامه‌های کاربری و کتابخانه‌های کاربری هستند.

## 8- دلیل استفاده از `objcopy` ؟

در فایل `Makefile` سیستم‌عامل `xv6` ، از فرمان `objcopy` برای کپی کردن یک فایل آبجکت به فایل آبجکت دیگر یا تبدیل فایل‌های باینری کامپایل‌شده به فایل‌های باینری خام استفاده می‌شود. در طول فرایند ساخت (`make`) ، کد منبع `xv6` کامپایل می‌شود و نتیجه آن یک فایل آبجکت برای هر منبع است. سپس این فایل‌ها به هم لینک می‌شوند و یک فایل باینری اجرایی در فرمت `ELF` ایجاد می‌شود. استفاده از این فرمان `Makefile` تضمین می‌کند که کد کامپایل‌شده `xv6` به یک تصویر باینری تبدیل شود که می‌تواند مستقیماً بارگذاری و اجرا شود. این فرایند شروع (`boot`) را ساده می‌کند و به سیستم‌عامل اجازه می‌دهد تا به طور بهینه روی سخت‌افزار هدف اجرا شود. سپس، فرمان `objcopy` برای تبدیل فایل `ELF` به یک فایل باینری خام استفاده می‌شود که یک

تصویر باینری برای سیستم عامل ایجاد می کند و در طول فرآیند بوت، در حافظه بارگذاری شده و توسط سخت افزار اجرا می شود.

9-

زیرا برخی از وظایف نیاز به دسترسی سطح پایین به سیستم دارند و نمی توان آن ها را با کد C انجام داد. برای اینکه بتوانیم از پردازنده 32 بیتی استفاده کنیم و حداکثر 4 گیگابایت حافظه داشته باشیم، باید وارد حالت محافظت شده شویم که تنها با زبان اسمبلی ممکن است (با تنظیم اولین بیت از Register Control 0).

به عنوان مثال، ورود به حالت محافظت شده (protected mode) است. زمانی که BIOS کد بخش بوت را بارگذاری می کند، پردازنده x86 در حالت واقعی (real mode) اجرا می شود. در این حالت، آدرس دهی حافظه همیشه فیزیکی است، پردازنده 16 بیتی است و ما فقط 1 مگابایت حافظه داریم.

13-

دلیل انتخاب آدرس `x1000000` (۱ مگابایت) برای بارگذاری کرنل در xv6 به دلایل زیر است:

### تقسیم بندی حافظه در حالت واقعی و گذار به حالت محافظت شده:

در مراحل اولیه بوت، سیستم در حالت \*واقعی\* است که در آن آدرس دهی حافظه به ۱ مگابایت محدود می شود (به دلیل استفاده از آدرس دهی ۱۶ بیتی به شکل بخش آفست).

بوت لودر که معمولاً در سکتور بوت قرار دارد، کرنل را به آدرس امن `x1000000` بارگذاری می کند که بالاتر از منطقه پایین ۱ مگابایتی حافظه است. این منطقه معمولاً برای BIOS، بوت لودر، و سایر ساختارهای مهم حالت واقعی رزرو شده است.



گذار به حالت \*محافظت شده\* اجازه استفاده از آدرس دهی ۳۲ بیتی و دسترسی به حافظه بالای ۱ مگابایت را می دهد. با قرار دادن کرنل در `x1000000`، از تداخل حافظه جلوگیری می شود و فضای حافظه پایین برای مصارف دیگر رزرو می شود.

## فضا برای بوت لودر و ساختارهای داده اولیه

حافظه زیر `x1000000` اغلب برای موارد زیر استفاده می شود:

- خود بوت لودر (که معمولاً در `x7C000` بارگذاری می شود).
- جدول وکتورهای وقفه حالت واقعی (که از `x00000` تا `x04000` را اشغال می کند).
- مناطق داده BIOS و مقداری از حافظه ویدئویی.
- بارگذاری کرنل در `x1000000` تضمین می کند که کرنل، بوت لودر یا سایر داده های موجود در حافظه پایین را بازنویسی نمی کند.

## رویکرد تاریخی

- به صورت تاریخی، بسیاری از سیستم های عامل x86، از جمله نسخه های قدیمی تر لینوکس و دیگر سیستم های شبیه یونیکس، از نقطه ۱ مگابایت به عنوان نقطه انتقال بین حالت واقعی و حالت محافظت شده استفاده کرده اند. این رویکرد آن را به یک الگوی آشنا و آزمایش شده برای سیستم های عاملی مانند xv6 تبدیل کرده است.

## ساده‌سازی مدیریت حافظه:

- با بارگذاری کرنل در `x100000`، xv6 می‌تواند از یک چیدمان حافظه ساده‌تر در طول مراحل اولیه راه‌اندازی استفاده کند. فضای بالای این آدرس می‌تواند به‌صورت راحت‌تر مدیریت شود و از مکانیزم‌های نگاشت صفحات و محافظت حافظه استفاده شود.

- همچنین این مکان یک فضای پیوسته برای کد و داده‌های کرنل فراهم می‌کند و فرآیند نگاشت آدرس‌های مجازی به آدرس‌های فیزیکی را در حین گذار از مراحل اولیه بوت به حالت چندوظیفه‌ای کامل، ساده‌تر می‌کند.

به‌طور خلاصه، آدرس `x100000` (۱ مگابایت) به این دلیل انتخاب می‌شود که یک نقطه مناسب و رایج بالاتر از محدودیت‌های حافظه حالت واقعی است که گذار به حالت محافظت‌شده را آسان می‌کند و از حافظه پایین برای ساختارهای حیاتی سطح پایین و بوت‌لودر محافظت می‌کند.

-18

در xv6، هم بخش‌های هسته (kernel) و هم بخش‌های کاربر (user) دارای توصیفگرهایی در جدول توصیفگر سراسری (GDT) هستند. این توصیفگرها اطلاعاتی درباره بخش‌ها مانند آدرس شروع، اندازه و سطح دسترسی دارند. وقتی که یک دستور اجرا می‌شود، ابتدا بخش مربوطه (چه کد و چه داده) از طریق توصیفگرش در GDT شناسایی می‌شود. توصیفگرهای هسته و کاربر می‌توانند سطوح دسترسی متفاوتی داشته باشند، حتی اگر به حافظه فیزیکی یکسان اشاره کنند. سطح دسترسی مشخص می‌کند که چه امتیازی برای اجرای دستور لازم است. در این فرآیند، سطح امتیاز فعلی (CPL) بر اساس سطح دسترسی مشخص‌شده در توصیفگر تعیین می‌شود. اگر CPL کمتر از سطح دسترسی مورد نیاز باشد، دستور نمی‌تواند اجرا شود. به عنوان مثال، برخی از دستورات ویژه ممکن است نیاز به سطح دسترسی بالاتری داشته باشند که برای کد سطح کاربر مجاز نیست.

وقتی که پرچم `USER_SEG` تنظیم می‌شود، به این معنی است که کدها و داده‌های سطح کاربر مجوزهای محدودی دارند و نمی‌توانند برخی از عملیات ویژه را که می‌تواند به یکپارچگی سیستم آسیب بزنند، اجرا کنند. در مقابل، کدها و داده‌های سطح هسته معمولاً مجوزهای بیشتری دارند زیرا نیاز به تعامل با سخت‌افزار و مدیریت سیستم دارند.

تنظیم پرچم `USER_SEG` برای بخش‌های سطح کاربر کمک می‌کند تا اطمینان حاصل شود که فرایندهای کاربر نمی‌توانند دستورهای ویژه را اجرا کنند یا به بخش‌های محدود حافظه دسترسی پیدا کنند، و این امر امنیت و پایداری سیستم عامل را افزایش می‌دهد.

برای مثال دستورالعمل `IN`، وظیفه خواندن یک بایت از پورت را دارد و این عمل نیازمند این است که سطح دسترسی فعلی مقداری ممتازتر از سطح دسترسی ورودی/خروجی داشته باشد (سطح دسترسی ورودی/خروجی در رجیستر وضعیت `FLAG` مشخص شده است) که این مقدار در لینوکس برابر صفر است.

- 19

این `struct` که برای ذخیره وضعیت هر پردازش استفاده می‌شود، در فایل `proc.h` تعریف شده و شامل ۱۳ متغیر است:

- `uint sz`: اندازه حافظه تخصیص یافته توسط پردازش به واحد بایت.

- `pde_t* pgdir`: یک پوینتر به جدول صفحه پردازش (`pde`: ورودی دایرکتوری صفحه).

- `char* kstack`: یک پوینتر به استک کرنل. استک کرنل بخشی از فضای کرنل است، نه فضای کاربر، و برای اجرای `syscall` ها از برنامه استفاده می‌شود.

- `enum procstate state`: این `enum` وضعیت پردازش را مشخص می‌کند و می‌تواند مقادیر

`procstate`` را به خود بگیرد، یعنی `UNUSED`، `EMBRYO`، `SLEEPING`،

`RUNNABLE`، `RUNNING` و `ZOMBIE`.

- pid :int این عدد، PID (شناسای پردازش) است که یک عدد منحصر به فرد بین تمام پردازش‌ها می‌باشد.

- parent \*\*struct proc: یک پوینتر به پردازش والد (پردازشی که پردازش کنونی را با استفاده از تابع 'fork' ایجاد کرده است). نوع این پوینتر همانند نوع خود پردازش کنونی یعنی 'struct proc' است.

- tf struct trapframe: یک پوینتر به فریم تله که برای ذخیره وضعیت اجرای برنامه در هنگام اجرای یک syscall استفاده می‌شود.

- context struct context: یک پوینتر به 'struct context' که مقادیر رجیسترهای مورد نیاز برای تعویض زمینه را نگه می‌دارد. تابع 'switch' که در اسمبلی تعریف شده، می‌تواند برای سوئیچ به یک پردازش دیگر استفاده شود.

chan void\*: اگر مقدار آن ۰ نباشد، به این معنی است که پردازش خوابیده است (در انتظار چیزی است). در اینجا، 'chan' به یک کانال اشاره دارد و کانال‌های متعددی وجود دارد، از جمله کانال ورودی کنسول.

• killed int: اگر مقدار آن ۰ نباشد، به این معنی است که پردازش کشته شده است.

ofile[NOFILE] struct file: یک آرایه از پوینترها به فایل‌های باز شده توسط پردازش.

• cwd struct inode: این متغیر دایرکتوری کاری جاری را مشخص می‌کند.

• name [16] Char: نام پردازش برای اهداف اشکال‌زدایی.

هسته اول که فرآیند بوت را انجام می‌دهد، از طریق کد `entry.S` وارد تابع `main` در فایل `main.c` می‌شود. تمامی توابع آماده‌سازی سیستم که در این تابع فراخوانی شده‌اند، توسط این هسته اجرا می‌شوند. از طرفی، سایر هسته‌ها از طریق کد `entryother.S` وارد تابع `mpenter` می‌شوند. در این تابع، چهار تابع آماده‌سازی فراخوانی می‌شوند. در نتیجه می‌توان گفت این چهار تابع بین تمامی هسته‌ها مشترک خواهند بود.

به طور کلی در main 18 تابع فراخوانی شده اند که آن 4 تابع فراخوانی شده در mpenter بین تمامی هسته های پردازنده مشترک هستند و 14 تابع دیگر اختصاصی هستند یکی از این توابع، به نام `switchkvm`، به طور مستقیم با هسته اول مشترک نیست. این تابع در `mpenter` صدا زده می‌شود، در حالی که در تابع `main` وجود ندارد. در واقع، تابع `kvmalloc` که در `main` فراخوانی می‌شود به صورت زیر است.

```
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    picinit(); // disable pic
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
    pinit(); // process table
    tvinit(); // trap vectors
    binit(); // buffer cache
    fileinit(); // file table
    ideinit(); // disk
    startothers(); // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
    userinit(); // first user process
    mpmain(); // finish this processor's setup
}
```

اجرای GDB :

1- از دستور `info breakpoints` استفاده میکنیم.

```
Breakpoint 4 at 0x555555551e3: file debug.c, line 11.
(gdb) info breakpoints
Num      Type      Disp Enb Address      What
1        breakpoint keep y 0x0000555555551b8 in factorial at debug.c:4
          breakpoint already hit 2 times
2        breakpoint keep y 0x00005555555551b8 in factorial at debug.c:4
3        breakpoint keep y 0x00005555555551b8 in factorial at debug.c:4
```

2- از دستور `del <breakpoint_number>` استفاده میکنیم. و همچنین میتونن مقدار `breakpoint number` را از دستور `info break` به دست اوریم.

```
(gdb) info break
Num      Type      Disp Enb Address      What
1        breakpoint keep y 0x000000097 in cat at cat.c:12
2        breakpoint keep y 0x000000dc in cat at cat.c:14
```

3- دستور ``bt`` که همان `backtrace` است، لیست `call stack` برنامه را که در لحظه‌ای که اجرای برنامه متوقف شده است، نمایش می‌دهد. هر تابعی که فراخوانی می‌شود، یک `stack frame` مخصوص به خود دارد که شامل متغیرهای محلی، آدرس بازگشت و سایر اطلاعات است. خروجی این دستور، هر خط را به عنوان یک `*stack frame*` نمایش می‌دهد و از درونی‌ترین `frame` شروع می‌شود. می‌توان با دستور ``bt n`` که در آن ``n`` یک عدد است، تنها ``n`` فریم داخلی را نمایش داد و با ``bt -n`` تنها ``n`` فریم بیرونی را مشاهده کرد. برای استفاده از این دستور، می‌توان از کلمات کلیدی مختلفی مانند ``bt``، ``backtrace``، ``wher`` و ``info stack`` استفاده کرد.

#### 4- تفاوت `print` , `x`

متغیر را به عنوان آرگومان ورودی میدیم و مقدار آن متغیر با دستور `print (p)` قابل چاپ است. ولی با دستور `X` محتویات خانه در حافظه چاپ میشود. که آگومانش آدرس خانه حافظه است. بذای مشاهده اطلاعات یک رجیستر از `<reg_name> info registers` استفاده میکنیم.

5- دستوری به نام `lr` داریم که معادل همان `info register` هست. دستور `info local` نیز برای

دیدن متغیرهای محلی استفاده میشود. ثابت `SI` مخفف `Source Index` است و برای اشاره به یک

منبع در عملیات `Stream` استفاده می شود. `DI` مخفف `Destination Index` است و برای اشاره

به یک مقصد در عملیات جریان استفاده می شود. "E" در ابتدای این نام های ثابت به معنای

`Extended` است و در حالت 32 بیتی استفاده می شود. `SI` به عنوان یک اشاره گر داده و به عنوان

منبع در عملیات رشته خاصی عمل می کند، در حالی که `DI` به عنوان یک اشاره گر داده و مقصد در

برخی از عملیات رشته ای عمل می کند.

```
(gdb) info locals
fd = <optimized out>
i = <optimized out>
```

### **x86 Architecture: EDI and ESI Registers:**

- In x86 architecture, the EDI (Extended Destination Index) and ESI (Extended Source Index) registers are commonly used for string and memory operations

#### **EDI (Extended Destination Index):**

- Primarily used as a pointer for the destination in string operations.

When

performing operations like `MOVS`, `LODS`, and `STOS`, the EDI register points to

the destination buffer where data will be written.

#### **ESI (Extended Source Index):**

- Used as a pointer for the source in string operations. It typically points to the

source buffer from which data will be read during these operations.  
In the context of function calls, EDI and ESI can also hold values that are preserved across function calls according to the calling convention used, making them useful for keeping track of data across different parts of a program.

## : Struct input -6

این ساختار در فایل `console.c` تعریف شده و برای خط ورودی کنسول سیستم عامل استفاده می شود.  
ساختار در کد به صورت زیر تعریف شده 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;
```

آرایه `buf` یک بافر است که خط ورودی را با حداکثر اندازه 128 کاراکتر ذخیره می کند.

سایر متغیرها اعداد صحیح هستند که هر کدام نمایانگر یک شاخص برای `buf` هستند.

متغیر `w` موقعیت شروع را برای نوشتن خط ورودی فعلی در `buf` نشان می دهد.

متغیر `e` نشان دهنده موقعیت فعلی مکان نما در خط ورودی است.

متغیر `r` برای خواندن از `buf` استفاده می شود که از `w` قبلی شروع می شود.



می بینیم که مقدار «r» به همان مقدار «w» رسیده است. این بدان معنی است که از «w» قبلی (که 0 بود) شروع شد و به «w» فعلی رسید و کل خط را خواند. (با تنظیم یک نقطه نظارت، می توانیم مشاهده کنیم که r یکی یکی پیش می رود).

این بار عبارت `another` را وارد می کنیم:

```
(gdb) print input
$3 = {buf = "test\nanother\n", '\000' <repeats 114 times>, r = 5, w = 13, e = 13}
```

«w» دوباره به انتهای «buf» منتقل می شود و «e» در ابتدای خط ورودی جدید است، بنابراین با «w» 8- مطابقت دارد. اگر برنامه را ادامه دهیم و سپس آن را متوقف کنیم، می بینیم که «r» به «w» می رسد.

-7

در یک محیط TUI (واسط کاربری متنی) در اشکال زدایی مانند GDB، می توان از دستورات مختلفی برای مشاهده نمایش های مختلف برنامه در حال اشکال زدایی استفاده کرد. به طور مشخص: مشاهده کد منبع: با استفاده از دستور «`layout src`» می توانید کد منبع برنامه ای که در حال حاضر اشکال زدایی می شود را نمایش دهید. این دستور به شما این امکان را می دهد که به راحتی کد را مرور کنید و بفهمید در کجای برنامه قرار دارید.

مشاهده کد اسمبلی: با دستور «`layout asm`» می توانید کد اسمبلی مربوط به برنامه را مشاهده کنید. این به ویژه برای توسعه دهندگانی مفید است که نیاز به دیدن اطلاعات دقیق در مورد نحوه عملکرد برنامه در سطح پایین تر دارند.

مشاهده همزمان کد منبع و اسمبلی: دستور «تقسیم طرح» به شما امکان می دهد هم کد منبع و هم کد اسمبلی را همزمان در محیط TUI مشاهده کنید. این ویژگی دیدگاه جامع تری در اجرای برنامه ارائه می دهد و به شما کمک می کند تا نقاط ضعف یا مشکلات موجود در کد را عیب یابی و شناسایی

کنید. ین دستورات به شما در بهبود فرآیند اشکال زدایی و به دست آوردن بینش بهتر در مورد نحوه عملکرد برنامه کمک می کند.

```
0xf0 <cat+96>    push    %eax
0xf1 <cat+97>    push    %eax
0xf2 <cat+98>    push    $0x7fa
0xf7 <cat+103>   push    $0x1
0xf9 <cat+105>   call    0x4c0 <printf>
0xfe <cat+110>   call    0x363 <exit>
0x103             xchg    %ax,%ax
0x105             xchg    %ax,%ax
0x107             xchg    %ax,%ax
0x109             xchg    %ax,%ax
0x10b             xchg    %ax,%ax
0x10d             xchg    %ax,%ax
```

8- برای مشاهده پشته تماس فعلی، می توانید از دستور «where» یا «backtrace» در محیط کاربر TUI استفاده کنید. در اینجا، من یک نقطه شکست در خط 48 فایل «proc.c» تنظیم کرده ام و پس از توقف اجرا در این نقطه، از دستور «where» برای مشاهده پشته تماس استفاده می کنیم:

برای پیمایش در پشته تماس، می توانید از دستورات «بالا <n>» و «پایین <n>» یا اختصارات آنها «u» و «d» استفاده کنید. در اینجا، «n» مشخص می کند که چند تابع به بالا یا پایین پشته می خواهید جابجا شود. اگر «n» ارائه نشده باشد، به طور پیش فرض یک حرکت می کند. به عنوان مثال، با دستور "up 2" به تابع "cupid" در خط 32 فایل "proc.c" حرکت می کنیم: