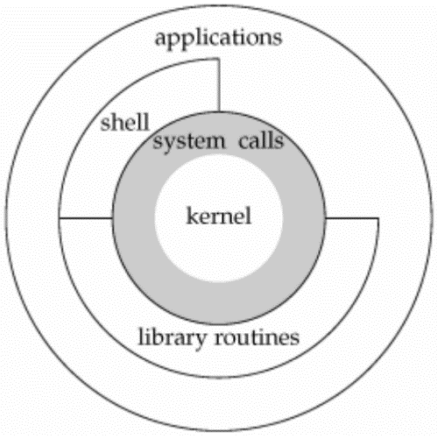
Repository Link: https://github.com/cylobi/xv6-public

آشنایی با سیستم‌عامل xv6

1. معماری سیستم‌عامل xv6

سیستم عامل xv6 مشابه Unix v6 نوشته شده و معماری و ساختاری شبیه به آن دارد. این سیستم عامل برای پردازنده‌های مبتنی بر x86 نوشته شده (مطابق با داکیومنت این سیستم عامل؛ xv6-rev11). علاوه بر آن در دفاع از این سخن می‌توان به فایل x86.h اشاره کرد که از دستورات پردازنده‌های x86 استفاده شده است. در دیگر فایل‌های "basic headers"، نظیر asm.h و mmu.h نیز می‌توان اشاراتی به معماری x86 مشاهده کرد.
معماری کلی سیستم عامل Unix بصورت زیر می‌باشد:



همانطور که گفته شد، معماری xv6 نیز از Unix پیروی می‌کند. این موضوع از دسته‌بندی فایل‌ها که شامل file systems، system calls، user-level و... می‌شود نیز قابل مشاهده است.

2. بخش‌های پردازه و چگونگی اختصاص پردازنده به پردازه‌های مختلف

یک پردازه در xv6 از حافظهٔ فضای کاربری (user-space) (شامل دستورات، داده‌ها و استک)، و وضعیت پردازه که فقط برای هسته قابل رؤیت است تشکیل شده است.
xv6 زمان را بین پردازه‌ها تقسیم می‌کند و به صورت نامحسوس پردازنده‌ها را برای اجرای دستورات به پردازه‌ها اختصاص می‌دهد. هروقت یک پردازه قرار است از اجرا توسط پردازنده خارج شود، سیستم‌عامل register های CPU که حاوی مقادیر مورد نیاز آن پردازه بوده را ذخیره می‌کند تا دفعه بعدی که آن پردازه قرار است اجرا شود، آنها را بازگرداند.
هسته xv6 به هر پردازه یک شناسه یکتا (PID (Process Identifier اختصاص می‌دهد. با استفاده از system call ء `getpid()` می‌توان PID پردازهٔ کنونی را دریافت کرد.

3. مفهوم file descriptor و عملکرد pipe در xv6

مفهوم file descriptor در این سیستم عامل‌ها در واقع به یک عدد اشاره می‌کند که به کمک آن می‌تواند از یک فایل بخواند یا در آن بنویسد. به ازای هر پردازه یک جدول برای نگهداری file descriptorها وجود دارد که باعث می‌شود این مقادیر برای پردازه‌ها به صورت خصوصی باشد و برای هر کدام از آن‌ها از مقدار 0 آغاز شود. طبق قرارداد، مقدار 0 برای stdin، مقدار 1 برای stdout و مقدار 2 برای stderr تعریف شده است. با توجه به

اینکه file descriptor می‌تواند مربوط به یک فایل، یک دستگاه یا pipe باشد، سیستم عامل با استفاده از پیاده‌سازی file descriptorها به این شکل، توانسته است یک interface انتزاعی برای هرکدام از این موارد ایجاد کند و همه آن‌ها را به یک شکل ببیند.

عملگر pipe برای ارتباط بین پردازه‌ها استفاده می‌شود. در واقع به کمک این عملگر می‌توانیم stdout یک پردازه را به stdin یک پردازه دیگر متصل کنیم.

عملکرد pipe در سیستم عامل xv6 به این صورت است که ابتدا به کمک تابع `pipe`، دو file descriptor که به هم متصل هستند ایجاد می‌کند. سپس برای پردازه سمت چپ ابتدا بخش قابل خواندن پایپ را می‌بندد و سپس بخش قابل نوشتن آن را به عنوان stdout برای این پردازه قرار می‌دهد و دستور را اجرا می‌کند. برای پردازه سمت راست ابتدا بخش قابل نوشتن پایپ را می‌بندد و سپس بخش قابل خواندن آن را به عنوان stdin در نظر می‌گیرد و در نهایت این دستور را هم اجرا می‌کند. سپس منتظر می‌ماند تا هر 2 دستور خاتمه یابند. ممکن است دستور سمت راست پایپ شامل دستوراتی باشد که در خود آن‌ها نیز از پایپ استفاده شده است. در این صورت، درختی از دستورات اجرا می‌شوند. لازم به ذکر است که پردازه سمت راست تا زمانی که stdin آن به end of file نرسیده باشد، منتظر داده جدید می‌ماند.

4. توابع exec و fork

تابع `fork` برای ایجاد یک process جدید استفاده می‌شود. در واقع این تابع یک نسخه کپی از پردازه‌ای می‌سازد که این تابع را صدا زده است. منظور از کپی این است که دیتا و دستورات پردازه فعلی در حافظه پردازه جدید (child) کپی می‌شوند. با وجود اینکه در لحظه ایجاد پردازه فرزند، داده‌های آن (متغیرها و رجیسترها) با پردازه پدر یکسان هستند، اما در واقع این دو پردازه حافظه جداگانه‌ای خواهند داشت و تغییر یک متغیر در پردازه پدر، آن متغیر در پردازه فرزند را تغییر نمی‌دهد. پردازه پدر پس از ایجاد پردازه فرزند، به caller تابع `fork` بازمی‌گردد که امکان اجرای همزمان دو پردازه را فراهم می‌سازد. مقدار return شده از تابع `fork` نیز pid پردازه فرزند خواهد بود. نقطه شروع پردازه فرزند نیز دقیقا همان caller تابع `fork` است با این تفاوت که مقدار خروجی این تابع عدد 0 خواهد بود. پس اگر با استفاده از کد `pid = fork();` یک پردازه جدید درست کنیم، یکی از حالت‌های زیر برای مقدار pid رخ می‌دهد:

- pid = 0: در پردازه فرزند هستیم.
- pid > 0: در پردازه پدر هستیم و مقدار pid در واقع آی‌دی پردازه فرزند است.
- pid < 0: در زمان اجرای تابع fork و پردازه جدید اروری وجود داشته و پردازه فرزند ایجاد نشده است.

اگر پس از fork کردن از تابع `wait()` استفاده شود، پردازه پدر منتظر پایان یافتن پردازه فرزند می‌شود و سپس کار خود را ادامه می‌دهد. خروجی این تابع، pid پردازه پایان یافته است. اگر پردازه فعلی هیچ پردازه فرزندی نداشته باشد، خروجی این تابع 1- خواهد بود.

قطعه کد زیر مثالی برای استفاده از تابع `fork` را نشان می‌دهد:

```c
int pid = fork();
if (pid == 0) {
    printf("This is child process\n");
    printf("Child process is exiting\n");
    exit(0);
}
else if (pid > 0) {
    printf("This is parent process\n");
    printf("Waiting for child process to exit\n");
    wait();
    printf("Child process exited\n");
}
else {
    printf("Fork failed\n");
}
```

تابع `exec` حافظه پردازه فعلی را با یک حافظه جدید که در آن یک برنامه با فایل ELF لود شده است، جایگزین می‌کند. در واقع `exec()` راهی برای اجرای یک برنامه در پردازه فعلی است. بر خلاف تابع `fork()`، برنامه به caller تابع `exec()` باز نمی‌گردد و برنامه جدید اجرا می‌شود مگر اینکه در زمان اجرای این تابع یک ارور رخ دهد. برنامه جدید اجرا شده در یک نقطه‌ای با استفاده از تابع `exit` اجرای پردازه را خاتمه می‌دهد. تابع `exec` دو پارامتر ورودی دارد که پارامتر اول نام فایل برنامه و پارامتر دوم آرایه آرگومان‌های ورودی برنامه است.
قطعه کد زیر مثالی از اجرای این تابع را نشان می‌دهد:

```c
char* args[] = {“ls”, “-l”, “/home”, NULL}; // Null is required
exec(“/bin/ls”, args);
printf(“Exec failed\n”);
```

مزیت ادغام نکردن این دو تابع در زمان I/O redirection خودش را نشان می‌دهد. زمانی که کاربر در shell یک برنامه را اجرا می‌کند، کاری که در پشت صحنه توسط shell انجام می‌شود به شرح زیر است:

1. ابتدا دستور تایپ شده توسط کاربر در ترمینال را می‌خواند.
2. با استفاده از تابع `fork` یک پردازه جدید ایجاد می‌کند.
3. در پردازه فرزند با استفاده از تابع `exec` برنامه درخواست شده توسط کاربر را جایگزین پردازه فعلی (فرزند) می‌کند.
4. در پردازه پدر برای اتمام کار پردازه فرزند `wait` می‌کند.
5. پس از اتمام پردازه فرزند به main باز می‌گردد و منتظر دستور جدید می‌شود.

زمانی که کاربر برای یک دستور از redirection استفاده می‌کند، تغییرات لازم در file descriptorها پس از `fork` و پیش از `exec` و در پردازه فرزند انجام می‌شود.
قطعه کد زیر این مورد را به شکل ساده شده نشان می‌دهد (فرض کنید دستور اجرا شده `cat < in.txt` است):

```
char* args = {“cat”, NULL};
int pid = fork();
if (pid == 0) {
    close(0); // close stdin
    open(“in.txt”, O_RDONLY); // open in.txt for reading (fd: 0)
    exec(“/bin/cat”, args):
    printf(“Exec failed\n”)
}
else if (pid > 0) {
    wait();
    printf(“Child process has exited\n”);
}
else {
    printf(“The fork failed\n”);
}
```

در صورتی که این دو تابع ادغام شوند، یا باید حالت‌های redirection به عنوان پارامتر به تابع forkexec پاس داده شوند که هندل کردن این حالت دردسرهای خودش را دارد و یا اینکه shell پیش از اجرای این تابع، file descriptorهای خود را تغییر دهد و بعد از اتمام کار این تابع نیز به حالت قبل برگرداند و یا در بدترین حالت، هندل کردن redirection را در هر برنامه مانند cat پیاده‌سازی کنیم.

اضافه کردن یک متن به Boot Message

پس از بوت شدن سیستم‌عامل نام اعضای گروه نمایش داده شده است.
این کار با افزودن یک printf در فایل init.c انجام شده است.

مقدمه‌ای درباره سیستم‌عامل و xv6

5. سه وظیفه اصلی سیستم‌عامل

1. انتزاع سخت افزارهای سطح پایین برای برنامه های کاربردی، بنابراین یک برنامه کاربردی لازم نیست نگران مدیریت حافظه، ورودی / خروجی و غیره است.
2. با استفاده از پروتکل های اشتراک زمانی، سخت افزار را بین برنامه ها به اشتراک می گذارد با توجه به بحرانی بودن، اولویت، و غیره. همچنین باید اطمینان حاصل شود که یک برنامه از منابع زیاد استفاده نمی کند و آنها را هدر نمی دهد.
3. راه های کنترل شده ای را برای تعامل برنامه ها فراهم می کند تا بتوانند به اشتراک بگذارند داده یا کار با هم

6. گروه‌های فایل های اصلی xv6

- Basic Headers: این فایل ها انواع داده های اساسی، ثابت ها و نمونه های اولیه تابعی که در سرتاسر پایگاه کد xv6 استفاده می شوند.
- قفل ها: این فایل ها مکانیزم های همگام سازی مانند spinlocks و sleeplocks. این قفل ها برای اطمینان از چندتایی استفاده می شوند رشته ها یا فرآیندها می توانند به طور ایمن به منابع مشترک بدون تداخل دسترسی پیدا کنند با همدیگر.
- Processes: این فایل ها مدیریت فرآیند را در xv6 انجام می دهند. اجرا می کنند توابع برای ایجاد، زمان بندی و جابجایی بین فرآیندها. آنها همچنین حاوی کدهای لازم برای بارگذاری و اجرای برنامه ها در فرآیندها
- تماس های سیستمی: این فایل ها کنترل کننده تماس های سیستمی را پیاده سازی کرده و ارائه می کنند توابع هسته مرتبط با هر فراخوانی سیستم. تماس های سیستمی به کاربر اجازه می دهد برنامه هایی برای درخواست عملیات ممتاز از هسته، مانند فایل عملیات یا مدیریت فرآیند
- File System: این فایل ها لایه سیستم فایل را در xv6 پیاده سازی می کنند. فراهم می کنند

توابعی برای مدیریت فایل ها، دایرکتوری ها و عملیات ورودی / خروجی دیسک. رسیدگی می کنند
عملیاتی مانند خواندن و نوشتن روی فایل ها، ایجاد و حذف
فایل‌ها و مسیریابی دایرکتوری‌ها.

- Pipes: این فایل ها مکانیزم لوله را پیاده سازی می کنند که اجازه می دهد
ارتباطات بین فرآیندی لوله ها راهی برای فرآیندها فراهم می کنند
با به اشتراک گذاشتن یک بافر مشترک ارتباط برقرار کنید و به یک فرآیند اجازه نوشتن را می دهد
داده هایی که فرآیند دیگری می تواند بخواند.
- عملیات رشته: این فایل ها توابع کاربردی را برای رشته ارائه می کنند
دستکاری - اعمال نفوذ. آنها عملیات رایج را روی رشته ها اجرا می کنند، مانند
کپی کردن، الحاق و مقایسه رشته ها.
- سخت افزار سطح پایین: این فایل ها در سطح پایینی با سخت افزار تعامل دارند.

آنها عملیات ورودی / خروجی دیسک در سطح بلوک را مدیریت می کنند و درایور دیسک را فراهم می کنند
رابط، و مدیریت ورودی و خروجی کنسول.

- User-Level: این فایل ها حاوی رابط سطح کاربری xv6 هستند. شامل می شوند
کد اسمبلی برای تماس های سیستمی در سطح کاربر، توابع ابزار برای کاربر
برنامه ها و کتابخانه ای از توابع که می توانند توسط برنامه های کاربر استفاده شوند.
- Bootloader: این فایل ها وظیفه bootstrapping xv6 را بر عهده دارند. آنها حاوی
کد بوت لودر که هسته xv6 را در حافظه بارگذاری می کند و
اجرای خود را آغاز می کند.
- Link: این فایل طرح بندی حافظه هسته و نحوه شی را مشخص می کند

فایل ها باید در طول فرآیند کامپایل به یکدیگر پیوند داده شوند.

نام پوشه ها در لینوکس:

- هسته: / کرنل
- فایل های سرصفحه: / include
- سیستم های فایل: / fs

کامپایل سیستم‌عامل xv6

7. دستور make -n و کدام دستور فایل نهایی را می‌سازد؟

پس از دستور make -n، می توانیم ببینیم:

```
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno
-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -o console.o console.c
ld -m    elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o io
apic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.
o string.o swtch.o syscall.o sysfile.o sysproc.o trapasm.o trap.o uart.o vectors.o vm.o  -b
inary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

در این دستور، گزینه o- kernel نام فایل خروجی را مشخص می کند که به طور معمول می باشد

هسته این نشان می دهد که فایل هسته نهایی با پیوند دادن فایل های شی با استفاده از آن تولید می شود دستور ld

8. متغیرهای UPROGS و ULIB در Makefile

متغیر UPROGS: این متغیر لیستی از برنامه‌های کاربر را دارد که در هنگام ساخت و کامپایل xv6، این برنامه‌ها نیز کامپایل و تبدیل به فایل‌های قابل اجرا توسط سیستم عامل می‌شوند. نام هر یک از این برنامه‌ها به صورت file_name_ در این لیست قرار گرفته است. تمام اسامی به صورت file_name_ (اسامی که یک _ ابتدایشان دارند)، یک هدف[1] با پیشنیاز[2]های فایل آبجکت هدف (file_name.o) و متغیر ULIB دارد. بنابراین هدف‌های موجود در UPROGS منجر به ساخت فایل آبجکت برنامه‌های کاربر، اجرا شدن هدف‌های مربوط به ULIB می‌شود و در نهایت اجرای دستور ld می‌شود. دستور ld برای پیوند[3] فایل‌های مورد نیاز و تولید یک فایل قابل اجرا مورد استفاده قرار می‌گیرد. علاوه بر آن فایل‌های آبجکت مربوط به هر برنامه (file_name.o) توسط یک قانون درونی[4] Makefile ساخته می‌شوند و به صورت صریح در Makefile نوشته نشده‌اند.
متغیر ULIB: این متغیر شامل تعدادی از کتاب‌خانه‌های زبان c می‌باشد. در بسیاری از کدهای xv6 توابع این کتاب‌خانه‌ها استفاده شده‌اند و برای اجرایشان به کامپایل این فایل‌ها نیاز داریم. برای مثال برنامه‌های سطح کاربر نیازمند کامپایل فایل‌های ULIB می‌باشند؛ بنابراین همانطور که در بخش قبل نیز گفته شد، فایل‌های ULIB به عنوان پیشنیاز در قوانین قرار گرفته‌اند و در نهایت توسط دستور ld به فایل‌های اجرایی پیوند می‌شوند. فایل‌های ULIB شامل توابعی مانند printf، strcmp، strcpy، malloc و... هستند.
در نهایت، همانطور که از اسم این متغیرها نیز پیداست، UPROGS معادل User Programs و ULIB معادل User Libraries است که به ترتیب برنامه‌های کاربر و کتاب‌خانه‌های کاربر محسوب می‌شوند.

اجرا بر روی شبیه‌ساز QEMU

9. محتوای دو دیسک ورودی QEMU

- xv6.img: این فایل تصویر دیسکی است که شامل سیستم عامل xv6 است. آی تی شامل هسته و تمام کدهای سطح سیستم لازم برای اجرای xv6 است.
- fs.img: این یک فایل تصویر دیسک اضافی است که یک سیستم فایل را نشان می دهد. برای ذخیره برنامه ها و داده های سطح کاربر استفاده می شود.

مراحل بوت سیستم‌عامل xv6

اجرای بوت‌لودر

10. محتوای سکتور نخست دیسک قابل بوت

[1] Target
[2] Prerequisite
[3] Link
[4] Built-in implicit rule

اولین کامندهای اجرا شونده توسط Makefile شامل کامپایل کردن object file های bootmain.c و bootasm.S، پیوند زدن این دو و تولید bootblock.o، objcopy کردن بخش text. فایل bootblock.o به فایل bootblock و در نهایت داده شدن به اسکریپت sign.pl برای اضافه کردن 2 بایت boot signature به bootblock است.

در سکتور نخست (512 بایت اول) دیسک قابل بوت، محتوای فایل bootblock قرار دارد.

11. مقایسه فایل باینری بوت با بقیه فایل‌های باینری xv6 و تبدیل آن به اسمبلی

همه فایل‌های باینری آبجکت xv6 در فرمت ELF[1] هستند. این فرمت باینری از بخش‌های مختلفی تشکیل شده است. در ابتدای آن هدرهایی شامل اطلاعات لود شدن فایل نوشته شده است و سپس چند section دارد که هر کدام حجمی از کد یا داده اند که در آدرس مشخصی از حافظه لود می‌شوند.

فرمت فایل ELF برای انواع object file ها یعنی relocatable (فایل‌های o. که توسط linker استفاده می‌شوند)، executable و shared object ها تعریف شده است.

دو هدر ELF Header و Program Header در فایل elf.h به زبان سی تعریف شده‌اند.

در ELF Header بخشی به نام e_entry وجود دارد که آدرس نقطه ورود برنامه را مشخص می‌کند.

از section های ELF می‌توان به text. و rodata. و data. و bss. اشاره کرد.

- `.text` شامل دستورات قابل اجرای برنامه است.
- `.rodata` حاوی داده‌های read-only از جمله string literal ها در زبان سی است.
- `.data` شامل داده‌های مقدار دهی شده مانند برخی متغیرهای گلوبال است.
- `.bss` شامل داده‌های مقدار دهی نشده است که چون داده‌ای وجود ندارد فقط آدرس و اندازه اش در فایل ذخیره می‌شود.

با استفاده از دستور `objdump -h bootblock.o` می‌توانیم نوع فایل باینری (که مانند بقیه فایل‌های باینری xv6 به فرمت elf32-i386 است)، و در ادامه خروجی دستور، section های ELF را مشاهده کنیم.

بوت لودر پس از لود شدن در آدرس ثابت 0x7C00، توسط پردازنده اجرا می‌شود تا کرنل را اجرا کند. در اینجا تنها اطلاعات مهم، کدی است که قرار است اجرا بشود. با مقایسه bootblock.o با بقیه object file ها می‌بینیم که بخش‌های data. و غیره را ندارد و بخش اصلی اش فقط text. است.

از آنجا که bootblock.o در آدرس خاصی شروع به اجرا شدن می‌کند، در هنگام ساخته شدنش از فلگ -Ttext 0x7C00 استفاده شده است که آدرس بخش text. فایل خروجی را مشخص می‌کند. فلگ -e start هم می‌گوید که نقطه شروع برنامه لیبل start در bootasm.S است.

خود فایلی که در سکتور بوت قرار دارد یعنی bootblock با استفاده از دستور:

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

(و اضافه کردن boot signature) تولید می‌شود. این فلگ‌های objcopy در بخش بعدی توضیح داده شده‌اند.

این دستور محتویات بخش text. را به صورت raw binary به فایل bootblock می‌ریزد. این یعنی فایل bootblock از فرمت ELF پیروی نمی‌کند و هیچ هدری هم ندارد. این فایل با دیگر فایل‌های باینری xv6 تفاوت دارد و کد قابل اجرای خالص بدون هیچ اطلاعات اضافه‌ای است.

پس نوع فایل دودویی مربوط به بوت raw binary است (که در حالت کلی چیز مشخصی نیست) و اینجا همان محتویات بخش text. (instruction های قابل اجرا بر روی معماری x86) می‌باشد.

[1] Executable and Linkable Format

این با بقیه فایل‌های باینری از آنجا که به فرمت ELF نیست تفاوت دارد.
دلیل استفاده نکردن از ELF برای bootblock این است که فرمت ELF را هسته سیستم‌عامل می‌داند و نه CPU. پس وقتی که هسته هنوز اجرا نشده نمی‌توان فرمت ELF را خواند. اگر BIOS فایل bootblock.o را برای بوت شدن به CPU می‌داد، از آنجا که CPU هدرهای ELF را نمی‌شناسد همه محتوای فایل را به دید instruction ها نگاه کرده و برداشت اشتباهی از آن می‌کند. پس باید فقط دستورات خالص را به CPU داد.
یک دلیل دیگر هم کم کردن حجم فایل است. با استخراج بخش text. فایل bootblock.o، حجم آن کاهش یافته و در 510بایت جا می‌گیرد.
برای تبدیل bootblock به اسمبلی، از کامند زیر استفاده می‌کنیم:

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

از آنجا که bootblock باینری خام است و هیچ هدری برای مشخص کردن معماری اش ندارد، آنها را باید دستی به objdump بدهیم. فلگ‌هایی که استفاده شده:

- `-D` : برای disassemble کردن باینری.
- `-b binary` : نوع فایل را raw binary در نظر می‌گیریم.
- `-m i386` : معماری اسمبلی فایل را مشخص می‌کنیم.
- `-M addr16,data16` : از آنجا که وقتی BIOS سکتور بوت را لود می‌کند در real mode هستیم و CPU در حالت 16بیت است، اسمبلی 16بیت نیز استفاده شده است پس هنگام disassemble کردن هم می‌گوییم که آدرس‌ها و داده‌ها را 16بیت در نظر بگیرد.

می‌توانیم با استفاده از فلگ `-adjust-vma=0x7C00` آدرس شروع قرار گرفتن اسمبلی خروجی در حافظه را تغییر بدهیم که مانند واقعیت از آدرس 0x7C00 شروع بشود.
با مشاهده خروجی کامند می‌بینیم که ابتدای آن بسیار مشابه با bootasm.S است:

```
bootblock:     file format binary


Disassembly of section .data:

00007c00 <.data>:
    7c00:       fa                      cli
    7c01:       31 c0                   xor    %ax,%ax
    7c03:       8e d8                   mov    %ax,%ds
    7c05:       8e c0                   mov    %ax,%es
    7c07:       8e d0                   mov    %ax,%ss
    7c09:       e4 64                   in     $0x64,%al
    7c0b:       a8 02                   test   $0x2,%al
    7c0d:       75 fa                   jne    0x7c09
    7c0f:       b0 d1                   mov    $0xd1,%al
    7c11:       e6 64                   out    %al,$0x64
    7c13:       e4 64                   in     $0x64,%al
    7c15:       a8 02                   test   $0x2,%al
    7c17:       75 fa                   jne    0x7c13
    7c19:       b0 df                   mov    $0xdf,%al
    7c1b:       e6 60                   out    %al,$0x60
    7c1d:       0f 01 16 78 7c          lgdtw  0x7c78
    7c22:       0f 20 c0                mov    %cr0,%eax
    7c25:       66 83 c8 01             or     $0x1,%eax
    7c29:       0f 22 c0                mov    %eax,%cr0
    7c2c:       ea 31 7c 08 00          ljmp   $0x8,$0x7c31
    7c31:       66 b8 10 00 8e d8       mov    $0xd88e0010,%eax
    7c37:       8e c0                   mov    %ax,%es
    7c39:       8e d0                   mov    %ax,%ss
    7c3b:       66 b8 00 00 8e e0       mov    $0xe08e0000,%eax
    7c41:       8e e8                   mov    %ax,%gs
    7c43:       bc 00 7c                mov    $0x7c00,%sp
    7c46:       00 00                   add    %al,(%bx,%si)
    7c48:       e8 f0 00                call   0x7d3b
```

12. علت استفاده از objcopy در هنگام make

با استفاده از این دستور می‌توان محتویات یک فایل object را در یک فایل object دیگر کپی کرد. برای این کار نیازی نیست فرمت فایل ورودی با فرمت فایل مقصد یکسان باشد. با توجه به اینکه این برنامه کار ترجمه فایل را با استفاده از کتاب‌خانه BFD انجام می‌دهد، تمامی فرمت‌های موجود در این کتاب‌خانه پشتیبانی می‌شوند و امکان تبدیل بین آن‌ها وجود دارد. این دستور برای ترجمه فایل‌های object از فایل‌های موقت (temp) استفاده می‌کند و سپس آن‌ها را پاک می‌کند. آپشن‌هایی از این دستور که در Makefile مربوط به xv6 استفاده شده‌اند به طور خلاصه در بخش زیر توضیح داده شده است:

- `-S` : در صورت استفاده از این آپشن، اطلاعات مروبط به symbol table و relocation records در فایل مقصد حذف می‌شوند. داده‌های symbol table نام و مکان متغیرها و فرآیندهایی را ذخیره می‌کنند که ممکن است در فایل‌های object دیگر از آن‌ها استفاده شده باشد. داده‌های relocation records نیز اطلاعاتی در مورد آدرس‌هایی از فایل object ذخیره می‌کند که در هنگام ساخت فایل مشخص نبوده و نیاز است در ادامه توسط linker مقداردهی شوند. این آدرس‌ها می‌توانند مربوط به متغیرها و توابعی باشند که در فایل‌های دیگر تعریف شده‌اند و در خود فایل وجود ندارند. در این حالت linker در زمان لینک کردن فایل‌ها، این آدرس‌ها را مقداردهی می‌کند.
- `-O` : این آپشن نوع فرمت فایل مقصد را نشان می‌دهد. برای مثال با استفاده از آپشن `-O binary` فایل تولید شده از نوع raw binary خواهد بود. این نوع فایل‌ها به فرمت خاصی نوشته نشده‌اند. از جمله این فایل‌ها می‌توان به فایل‌های memory dump اشاره کرد.
- `-j` : با استفاده از این آپشن می‌توانیم تنها بخشی از فایل object را به فایل جدید کپی کنیم.

در این Makefile در چند بخش زیر از دستور objcopy استفاده شده است:

1. در bootblock پس از لینک شدن bootmain.o و bootasm.o در فایلی به نام bootblock.o، محتویات بخش text. این فایل را در یک فایل raw binary به نام bootblock کپی می‌کند. سپس این فایل را به اسکریپت sign.pl می‌دهد که ابتدا سایز فایل را بررسی می‌کند که بیشتر از 510 بایت نباشد و سپس 2 بایت 0x55 و 0xAA که boot signature اند را به انتهای فایل اضافه می‌کند.
2. در entryother محتویات بخش text. فایل bootblockother.o را در یک فایل raw binary به نام entryother کپی می‌کند.
3. در initcode محتویات فایل initcode.out در یک فایل raw binary به نام initcode کپی می‌شود.

در نهایت با لینک شدن فایل‌های entry.o و فایل‌های object که در متغیر OBJS تعریف شده‌اند و فایل‌های باینری initcode و entryother که پیش‌تر با استفاده از دستور objcopy ساخته شدند، فایل kernel ساخته می‌شود.

13. چرا برای بوت کردن فقط از فایل C استفاده نشده و اسمبلی هم هست؟

چون که برخی از کارها نیازمند دسترسی سطح پایین به سیستم می‌باشند و با کد C نمی‌توان آنها را انجام داد. یک نمونه از این کارها وارد شدن به protected mode است.

وقتی که BIOS کد سکتور بوت را لود می‌کند، پردازنده x86 در real mode اجرا می‌شود. در این حالت آدرس دهی حافظه همیشه فیزیکی است، پردازنده 16بیت است و فقط 1 مگابایت حافظه داریم.

برای اینکه بتوانیم از پردازنده 32بیت استفاده کنیم و تا 4 گیگابایت حافظه داشته باشیم، باید وارد protected mode بشویم که این کار فقط در اسمبلی (با 1 کردن بیت اول Control Register 0) ممکن است.

14. وظیفه ثبات‌های x86

- ثبات عام منظوره: پردازنده‌های x86 دارای 8 ثبات عام منظوره هستند. از این ثبات‌ها می‌توان به ثبات انباشت‌کننده[1] اشاره کرد که یک ثبات میانی برای ذخیره خروجی بخش محاسباتی (ALU) است. نام این ثبات از این رو انباشت‌کننده نهاده شده که پس از هر بار انجام محاسبات، نتیجه در آن ذخیره شده و در محاسبات بعدی از مقدار ذخیره شده در آن به عنوان ورودی استفاده می‌شود و دوباره نتیجه آن در همین ثبات ذخیره می‌شود. به عبارتی دیگر بصورت نوبتی، نتایج محاسبات در آن انباشت می‌شوند.
- ثبات قطعه: پردازنده‌های x86 دارای 6 ثبات قطعه هستند. یک ثبات قطعه، ثبات پشته[2] می‌باشد. این ثبات اطلاعاتی مربوط به قطعه‌ای از حافظه را ذخیره می‌کند که از آن برای پشته فراخوانی[3] استفاده می‌شود. دقت شود که ثبات قطعه پشته (SS) با ثبات نشانگر پشته (SP) تفاوت دارد؛ برای اطلاعات بیشتر در این خصوص به این پیوست مراجعه کنید.
- ثبات وضعیت: ثبات FLAGS، ثبات وضعیتی است که نشان‌دهنده حالت فعلی پردازنده است. این ثبات مخصوص پردازنده‌های 16 بیتی است. EFLAGS و RFLAGS ثبات‌های مشابه برای پردازنده‌های 32بیتی و 64بیتی می‌باشند. هر بیت از این ثبات نشان‌دهنده یک پرچم[4] برای یک وضعیت می‌باشد که می‌تواند حالت درست یا غلط داشته باشد. این پرچم‌ها نشان‌دهنده وضعیت اعمال منطقی و محاسباتی یا محدودیت‌های اعمال شده بر عملیات فعلی پردازنده هستند. واضح است که عملکرد این پرچم‌ها به تعداد بیت‌های رجیستر و معماری پردازنده بستگی دارد. ثبات FLAGS برای پردازنده Intel x86 به شرح زیر می‌باشد:

Intel x86 FLAGS register			
بیت	مخفف	توضیح	دسته‌بندی
0	CF	Carry flag	وضعیت
1		رزرو شده	
2	PF	Parity flag	وضعیت
3		رزرو شده	
4	AF	Adjust Flag	وضعیت
5		رزرو شده	
6	ZF	Zero flag	وضعیت
7	SF	Sign flag	وضعیت
8	TF	Trap flag	کنترل
9	IF	Interrupt enable flag	کنترل
10	DF	Direction flag	کنترل
11	OF	Overflow flag	وضعیت

[1] Accumulator register (AX)
[2] Stack
[3] Call stack
[4] Flag

12-13	IOPL	سطح دسترسی ورودی خروجی	سیستم
14	NT	پرچم فعالیت تو در تو	سیستم
15		رزرو شده	

- ثبات کنترلی: این نوع از ثبات‌ها مسئول تغییر در رفتار کلی پردازنده و یا دیگر دستگاه‌های مرتبط اند. از این دسته ثبات‌ها می‌توان به ثبات CR0 اشاره کرد که در پردازنده‌های 32بیتی مانند i386 و بالاتر استفاده می‌شود. بیت‌های این ثبات نشان‌دهنده تغییرات و کنترل‌های مختلفی در رفتار کلی پردازنده هستند که به شرح زیر می‌باشد:

بیت	مخفف	نام
0	PE	Protected Mode Enable
1	MP	Monitor co-processor
2	EM	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 disabled
31	PG	Paging

15. نقص اصلی real mode پردازنده x86

در طول فرآیند بوت شدن یک سیستم مبتنی بر x86، پردازنده در یک حالت شروع می شود
به نام "حالت واقعی". این یک حالت 16 بیتی است که با قدیمی تر سازگاری دارد
پردازنده های x86 با این حال، حالت واقعی چندین محدودیت دارد، مانند حداکثر 1 مگابایت
حافظه آدرس پذیر
برای غلبه بر این محدودیت ها، سیستم عامل های مدرن به سرعت پردازنده را تغییر می دهند
به حالت محافظت شده، که از آدرس دهی 32 بیتی پشتیبانی می کند و دسترسی به ویژگی هایی مانند
حافظه مجازی. در مورد سیستم های 64 بیتی، پردازنده بیشتر به "long mode" تغییر می کند
برای پشتیبانی 64 بیتی

16. آدرس‌دهی حافظه در real mode

در حالت واقعی، که یک حالت 16 بیتی است که در تمام پردازنده های x86 وجود دارد، آدرس دهی به این صورت است
با استفاده از مدل حافظه تقسیم شده انجام می شود. این حالت با یک قطعه بندی 20 بیتی مشخص می شود
فضای آدرس حافظه، دقیقاً 1 مگابایت (مبی بایت) حافظه آدرس پذیر را فراهم می کند.

آدرس دهی حافظه در حالت واقعی از یک سیستم segment:offset استفاده می کند. شش 16 بیتی وجود دارد
ثبت بخش ها: CS، DS، ES، FS، GS و SS. بخش ها و افست ها به فیزیکی مربوط می شوند
آدرس ها با معادله: PhysicalAddress = Segment * 16

17. کد bootmain.c چرا هسته را در آدرس 0x100000 قرار می‌دهد؟

آدرس x1000000 (یا 1 مگابایت در اعشار) اغلب به عنوان آدرس شروع برای بارگیری استفاده می شود
هسته در بسیاری از سیستم عامل ها، از جمله xv6. این یک کنوانسیون است که قدمت آن برمی گردد
روزهای اولیه کامپیوترهای شخصی دلیل آن به شرح زیر است:
1. Memory Layout: در چیدمان حافظه یک سیستم مبتنی بر x86، قسمت پایینی
حافظه (زیر 1 مگابایت) معمولاً برای بایوس، حافظه ویدیویی و سایر ورودی / خروجی رزرو شده است
دستگاه ها با بارگذاری هسته بالای 1 مگابایت، از بازنویسی این مناطق جلوگیری می کند.
2. حالت محافظت شده: پردازنده x86 در حالت واقعی (حالت 16 بیتی) شروع می شود، جایی که می تواند
فقط آدرس 1 مگابایت حافظه برای استفاده از حافظه بیشتر و فعال کردن ویژگی هایی مانند مجازی
حافظه و چندوظیفگی، پردازنده باید به حالت محافظت شده سوئیچ کند.
بارگذاری کرنل بالای 1 مگابایت تضمین می کند که پس از دسترسی پردازنده، به آن دسترسی پیدا کنید
به حالت محافظت شده تبدیل شد.
3. Memory Segmentation: در حالت واقعی، حافظه قطعه بندی می شود و هر بخش می تواند
حداکثر 64 کیلوبایت باشد. آدرس x1000000 جایی است که اولین بخش آن است
با دیگری همپوشانی ندارد شروع می شود. بنابراین، مکان مناسبی برای بارگیری هسته است

18. کد معادل entry.s در هسته لینوکس

کد معادل entry.S برای معماری x86 در هسته لینوکس:
https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry.S
که برای 32بیت و 64بیت جدا است.

اجرای هسته xv6

19. دلیل فیزیکی بودن آدرس page table

برای تبدیل آدرس مجازی به آدرس فیزیکی نیازمند جدول ذکر شده هستیم و برای دسترسی به این جدول نیاز به آدرس آن داریم. در صورتی که آدرس این جدول به صورت مجازی ذخیره شود، برای پیدا کردن آدرس فیزیکی‌اش به خودش نیاز خواهیم داشت و حلقه بی‌نهایتی به وجود می‌آید که این حالت باعث ایجاد تناقض می‌شود و هیچ وقت نمی‌توانیم به این جدول دسترسی پیدا کنیم. در صورتی که بخواهیم از یک جدول دیگر برای پیدا کردن آدرس فیزیکی این جدول استفاده کنیم، در نهایت نیاز به یک آدرس فیزیکی برای پایان دادن به حلقه خواهیم داشت. در نتیجه آدرس دسترسی به این جدول به صورت فیزیکی ذخیره می‌شود.

20. توابع entry.s را توضیح دهید و تابع معادل در هسته لینوکس را بیابید

- Multiboot Header: هدر multiboot توسط بوت لودرهایی مانند GNU Grub استفاده می شود.

کرنل را در حافظه بارگذاری کنید. جادو و پرچم ها مقادیر خاصی هستند که توسط مشخصات Multiboot.

- _start: این نماد نقطه ورود ELF را مشخص می کند. از آنجایی که حافظه مجازی وجود نداشته است هنوز راه اندازی شده است، نقطه ورود آدرس فیزیکی ورودی است.
- ورودی: در اینجاست که xv6 در پردازنده بوت اجرا می‌شود و صفحه‌بندی خاموش است. آی تی پسوند اندازه صفحه را برای صفحات 4 مگابایت روشن می کند، دایرکتوری صفحه را تنظیم می کند، تبدیل می کند

در صفحه بندی، نشانگر پشته را تنظیم می کند و سپس به main() می پرد.

- پشته: این یک بلوک رایج حافظه است که برای پشته هسته رزرو شده است

همه آنها به جز ورودی در لینوکس نیز وجود دارد

21. مختصری راجع به محتوای فضای آدرس مجازی هسته

فضای آدرس مجازی هسته با استفاده از مکانیزم صفحه‌بندی دو سطحی تنظیم می‌شود پشتیبانی شده توسط واحد مدیریت حافظه (MMU). آدرس های مجازی هسته شروع می شود از (0x80000000) KERNBASE، که بسیار بالاتر از آدرس های برنامه کاربر است.

این طراحی به همان کد هسته اجازه می دهد تا در فضای آدرس مجازی نگاشت شود هر فرآیند

اولین قسمت از فضای آدرس مجازی هسته از KERNBASE تا KERNBASE 4MB است به همان آدرس های فیزیکی، یعنی از 0 تا 4 مگابایت نگاشت شده است. این با استفاده از enter.S انجام می شود

یک دایرکتوری صفحه دو ورودی ساده به نام enterpgdir. بقیه مجازی هسته فضای آدرس بعداً در main.c توسط تابع kvmalloc () تنظیم می شود

22. چرا برای کد و داده های سطح کاربر پرچم SEG_USER تنظیم شده است؟

قطعه‌بندی در xv6 در تابع `seginit` و در تکه کد زیر انجام می‌شود:

```
  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);
```

علاوه بر آن تعریف SEG به صورت زیر می‌باشد:

```
#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 }
```

بنابراین همانطور که مشخص است (و در توضیحات آزمایش نیز آمده است) تمام قطعه‌های هسته و کاربر یک بخش از حافظه را در اختیار دارند. هر یک از این قطعه‌ها با یک دسکریپتور در GDT[1] مشخص شده که این دسکتریپتور شامل اطلاعاتی مانند آدرس شروع قطعه، اندازه قطعه و سطح دسترسی قطعه می‌باشد.

[1] Global descriptor table

برای خواندن یک دستورالعمل، ابتدا قطعه آن از طریق دسکریپتورش یافت می‌شود (که در اینجا قطعه کدِ دسکریپتورِ هسته و کاربر یکسان اند) و سپس صفحه مربوط به آن پس از طی مراحل مربوطه پیدا می‌شود. پس از این مراحل و تبدیل آدرس منطقی به آدرس فیزیکی، دستورالعمل از حافظه خوانده شده و اجرا می‌شود. موضوعی که در این مرحله باید به آن دقت کرد، سطح دسترسی مورد نیاز یک دستور برای اجرای آن است. هنگامی که مکان قطعه از روی دسکریپتور قطعه مشخص می‌شود، سطح دسترسی فعلی یا همان CPL[1] نیز از روی سطح دسترسی دسکریپتور یا همان DPL[2] مشخص می‌شود. بدین گونه از طریق DPL متفاوت می‌توان سطح دسترسی فعلی دستورالعمل‌ها را نیز تعیین کرد؛ حتی اگر این دسکتریپتورها قطعات یکسانی از حافظه را تعریف کنند.

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؛ مقدار دسترسی فعلی (CPL) برابر مقدار سطح دسترسی دسکتریپتور (DPL) قطعه‌ای است که کد مربوط به این دستور العمل در آن قرار گرفته است و اگر این دستورالعمل در قطعه کاربر قرار گرفته باشد قابل اجرا نخواهد بود؛ چرا که قطعه مربوط به کد کاربر، سطح دسترسیش برابر DPL_USER یا همان 3 (کمترین میزان دسترسی) است.

بنابراین با وجود اینکه هر دو بخش کاربر و هسته به قطعات یکسانی دسترسی دارند، اما سطح دسترسی متفاوتی داشته و کاربر هر دستورالعملی را نمی‌تواند اجرا کند.

اجرای نخستین برنامه سطح کاربر

23. اجزای struct proc و معادل آن در لینوکس

این struct که برای ذخیره وضعیت هر پردازه به کار می‌رود، در فایل proc.h تعریف شده و 13 متغیر در آن قرار دارد:

- `uint sz` : حجم و اندازه حافظه گرفته شده توسط پردازه به واحد بایت.
- `pde_t* pgdir` : پوینتر به page table پردازه است. (pde: page directory entry)
- `char* kstack` : پوینتر به kernel stack است. استک کرنل قسمتی از kernel space است و نه user space و برای اجرای syscall ها از برنامه استفاده می‌شود.
- `enum procstate state` : این enum وضعیت پردازه را مشخص می‌کند و می‌تواند به حالت‌های procstate یعنی `UNUSED`, `EMBRYO`, `SLEEPING`, `RUNNABLE`, `RUNNING`, `ZOMBIE` باشد.
- `int pid` : این عدد (Processor Identifier) PID است که عدد یکتایی بین همه پردازه‌ها است.
- `struct proc* parent` : پوینتر به پردازه پدر (پردازه سازنده پردازه کنونی توسط تابع `fork`) است. تایپ این پوینتر مثل خود پردازه کنونی struct proc است.
- `struct trapframe* tf` : پوینتر به trap frame برای ذخیره وضعیت اجرای برنامه در هنگام اجرای یک syscall.

[1] Current privilege level

[2] Descriptor privilege level

- `struct context* context` : پوینتر به struct context است که مقادیر رجیسترهای مورد نیاز برای context switching را نگه می‌دارد. با استفاده از تابع `swtch` (که با اسمبلی تعریف شده) می‌توان به یک پردازه switch کرد.
- `void* chan` : در صورتی که مقدار آن 0 نباشد، یعنی پردازه خوابیده است (برای کاری wait می‌کند). اینجا chan به معنای channel است و چنل‌های متعددی از جمله چنل خط ورود کنسول داریم.
- `int killed` : در صورتی که مقدار آن 0 نباشد یعنی پردازه kill شده است.
- `struct file* ofile[NOFILE]` : آرایه‌ای از پوینترها به فایل‌های باز شده توسط پردازه است.
- `struct inode* cwd` : این متغیر current working directory را مشخص می‌کند.
- `char name[16]` : نام پردازه برای اشکال زدایی.

معادل این struct در هسته لینوکس:

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

استراکت `task_struct` در فایل <linux/sched.h>

24. چرا به خواب رفتن در کد مدیریت‌کننده سیستم مشکل‌ساز است؟

عملکرد خواب با آزاد کردن قفل قبل از رها کردن CPU کار می کند. این کار انجام می شود تا
از بن بست جلوگیری کنید و اطمینان حاصل کنید که سایر فرآیندها می توانند قفل را در حین جریان به دست آورند
فرآیند خواب است با این حال، این طراحی می تواند منجر به شرایط مسابقه در صورت فرآیند دیگری شود
قفل را بدست می آورد و وضعیت سیستم را قبل از فرآیند خواب تغییر می دهد
به طور کامل به حالت خواب منتقل شده است

25. تفاوت فضای آدرس هسته با فضای آدرس توسط kvmalloc

● kvmalloc(): این تابع با ایجاد a حافظه مجازی هسته را تنظیم می کند
مکانیسم صفحه بندی دو سطحی برای ایجاد دایرکتوری صفحه جدید، setupkvm() را فراخوانی می کند
و سپس محدوده ای از آدرس های فیزیکی را در جداول صفحه در مشخص شده نگاشت می کند
دایرکتوری صفحه
● setupkvm(): این تابع برای تنظیم بخشی از هسته مجازی استفاده می شود
حافظه هنگام کپی کردن کل حافظه مجازی (هسته کاربر) از یک صفحه
فهرست راهنما. در طول copyuvm() نامیده می شود که هنگام ایجاد یک کپی از a استفاده می شود
جدول صفحه فرآیند در مقابل، allocuvm() حافظه مجازی موجود را گسترش می دهد
(به طور خاص قسمت پشته)، و از آنجایی که قبلاً بخش هسته وجود دارد
نگاشت در allocuvm()، نیازی به فراخوانی setupkvm() ندارد

26. تفاوت فضای آدرس inituvm با فضای آدرس کاربر در کد مدیریت سیستم

● inituvm(): این تابع برای مقداردهی اولیه بخشی از فضای آدرس برای کاربر استفاده می شود
یک روند جدید یک صفحه از حافظه فیزیکی را اختصاص می دهد، فایل اجرایی init را کپی می کند
در آن حافظه، و یک PTE 14 را برای صفحه اول آدرس مجازی کاربر تنظیم می کند
فضا.

- آدرس کاربر: فضای آدرس کاربر در xv6 محدوده ای از آدرس های مجازی است که a فرآیند کاربر می تواند دسترسی داشته باشد. برای هر فرآیندی، آدرس مجازی حافظه کاربر (VA) محدوده از 0 تا KERNBASE است، که در آن 0x80000000 KERNBASE است (یعنی 2 گیگابایت حافظه) برای هر فرآیند در دسترس است. هنگامی که یک کاربر پردازش درخواست حافظه به بخش کاربری خود را از آدرس بسازد

27. کدام بخش از آماده‌سازی سیستم بین تمامی هسته‌های پردازنده مشترک و کدام بخش اختصاصی است؟

هسته اول که فرآیند بوت را انجام می‌دهد توسط کد entry.S وارد تابع `main` در فایل main.c می‌شود. تمامی توابع آماده سازی سیستم که در این تابع فراخوانده شده‌اند توسط این هسته اجرا می‌شوند. از طرفی، هسته‌های دیگر از طریق کد entryother.S وارد تابع `mpenter` می‌شوند. در این تابع نیز 4 تابع برای آماده‌سازی فراخوانده می‌شوند. در نتیجه می‌توان گفت این 4 تابع بین تمامی هسته‌ها مشترک خواهند بود. یکی از این توابع به نام `switchkvm` به صورت مستقیم با هسته اول مشترک نیست. این تابع در `mpenter` صدا زده می‌شود در صورتی که در تابع `main` وجود ندارد. در واقع تابع `kvmalloc` که در `main` صدا زده می‌شود به صورت زیر است:

```
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

خط اول تابع یک page table برای کرنل ایجاد می‌کند که این مورد توسط هسته اول انجام می‌پذیرد. پس از آن باید هسته به این page table سوییچ کند که این کار در تمامی هسته‌ها انجام می‌پذیرد. بخش‌هایی از آماده‌سازی سیستم که در تمام هسته‌ها مشترک هستند به شرح زیر است:

- `switchkvm`
- `seginit`
- `lapicinit`
- `mpmain`

همچنین بخش‌هایی که تنها در هسته اول (به صورت اختصاصی) اجرا می‌شوند به شرح زیر است:

- `kinit1`
- `(setupkvm) kvmalloc`
- `mpinit`
- `picinit`
- `ioapicinit`
- `consoleinit`
- `uartinit`
- `pinit`
- `tvinit`
- `binit`
- `fileinit`
- `ideinit`
- `startothers`

- kinit2
- userinit

از موارد اختصاصی هسته اول می‌توان به تابع startothers اشاره کرد که واضح است فقط پردازنده اول نیاز است بقیه پردازنده‌ها را start کند و نیازی نیست هر پردازنده در زمان بالا آمدن این کار را انجام دهد. یا برای مثال زمانی که پردازنده اول به کمک تابع ideinit دیسک را شناسایی می‌کند، نیازی نیست بقیه پردازنده‌ها این کار را انجام دهند.

از طرفی، همه پردازنده‌ها باید آدرس page table که توسط پردازنده اول ایجاد شده را در رجیستر خود ذخیره کنند در نتیجه تابع switchkvm بین همه آن‌ها مشترک است. همچنین، همه پردازنده‌ها باید کار خود را شروع کنند و آماده اجرای برنامه‌ها شوند که این مورد توسط تابع mpmain انجام می‌پذیرد. در نتیجه این تابع هم بین تمام پردازنده‌ها مشترک خواهد بود.

زمان‌بند که توسط تابع scheduler انجام می‌پذیرد در تابع mpmain صدا زده می‌شود که این تابع بین تمامی هسته‌ها مشترک است. این مورد از کامنت‌های داکیومنت تابع ذکر شده نیز قابل برداشت است:

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
```

هر پردازنده scheduler مربوط به خودش را خواهد داشت و در نتیجه این تابع بین تمامی پردازنده‌ها مشترک است.

28. برنامه معادل initcode.s در هسته لینوکس

https://github.com/torvalds/linux/blob/master/arch/arm/boot/bootp/init.S

اشکال زدایی

روند اجرای GDB

1. دستور مشاهده breakpointها

برای مشاهده breakpointهای فعلی می‌توان از دستور info breakpoints استفاده کرد:

```
(gdb) b cat.c:12
Breakpoint 1 at 0x93: file cat.c, line 12.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00000093 in cat at cat.c:12
```

2. دستور حذف یک breakpoint

برای حذف یک breakpoint می‌توان از دستور del <breakpoint_number> استفاده کرد. مقدار breakpoint number را از طریق دستور info break می‌توان مشاهده کرد. برای مثال در نمونه زیر، دستور info دو breakpoint در خطوط 12 و 14 فایل cat.c را نشان می‌دهد:

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

حال با استفاده از دستور `del 2`، breakpoint واقع شده در خط 14 را حذف کرده و سپس مجددا با دستور `info break` صحت این عمل را تایید می‌کنیم:

```
(gdb) del 2
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00000097 in cat at cat.c:12
```

بنابراین با استفاده از دستور `delete` یا همان `del` و با بهره‌گیری از شماره breakpoint می‌توان آن را حذف کرد. علاوه بر آن با استفاده از دستور `clear` و مکان مشخص breakpoint (ترکیبی از نام فایل و شماره خط آن به صورت `<file_name>:<line_number>`) می‌توان breakpoint موجود در آن مکان را پاک کرد.

کنترل روند اجرا و دسترسی به حالت سیستم

3. خروجی bt

دستور `bt` که مخفف backtrace است call stack برنامه در لحظه کنونی (در حین متوقف بودن روند اجرای برنامه) را نشان می‌دهد.

هر تابع که صدا زده می‌شود یک stack frame مخصوص به خودش را می‌گیرد که متغیرهای محلی و آدرس بازگشت و غیره در آن قرار دارند.

خروجی این دستور در هر خط یک stack frame را نشان می‌دهد که به ترتیب از درونی‌ترین frame که در آن قرار داریم شروع می‌شود.

می‌توان با دستور `bt n` که n یک عدد است فقط n فریم درونی‌تر را نشان داد و با دستور `bt -n` فقط n فریم بیرونی‌تر را نشان داد.

برای استفاده از این دستور می‌توان از کلیدواژه‌های مختلفی استفاده کرد از جمله:

```
bt, backtrace, where, info stack
```

در مثال زیر، در خط 15 فایل wc.c یک breakpoint گذاشته شده است. این خط کد، داخل تابعی به نام `wc` قرار دارد که از داخل تابع `main` ورودی برنامه wc صدا می‌شود.

پس از اجرای کامند `wc README` مشاهده می‌کنیم که روی خط 15 متوقف شده و دستور `bt` به طور صحیح call stack را نشان می‌دهد.

```
Reading symbols from _wc...
(gdb) break 15
Breakpoint 1 at 0xa0: file wc.c, line 15.
(gdb) continue
Continuing.
[  1b:  a0]    0x250 <strchr>:  push   %ebp

Thread 1 hit Breakpoint 1, wc (fd=3, name=0x2ff4 "README") at wc.c:15
15          while((n = read(fd, buf, sizeof(buf))) > 0){
(gdb) bt
#0  wc (fd=3, name=0x2ff4 "README") at wc.c:15
#1  0x00000056 in main (argc=2, argv=0x2fe8) at wc.c:50
(gdb)
```

4. تفاوت دستور x و print

همانطور که در help این دو دستور نوشته شده است، با استفاده از دستور `print` (به اختصار `p`) می‌توان مقدار یک متغیر را چاپ کرد. آرگومان ورودی این دستور، نام متغیر خواهد بود.
با استفاده از دستور `x` می‌توان محتویات یک خانه حافظه را چاپ کرد. بدیهی‌ست که آرگومان ورودی این دستور، آدرس خانه حافظه مذکور است.
لازم به ذکر است که هر دو دستور ذکر شده می‌توانند فرمت خروجی را به صورت `/FMT` در آرگومان‌های ورودی خود دریافت کنند.
در مثال زیر پس از دستور `cat prime_numbers.txt` متغیر fd چاپ می‌شود. برای پیدا کردن آدرس این متغیر نیز از دستور `print &fd` استفاده شده است:

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=3) at cat.c:12
12          while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) print fd
$1 = 3
(gdb) print &fd
$2 = (int *) 0x2f90
(gdb) x 0x2f90
0x2f90: 0x00000003
(gdb) x/d 0x2f90
0x2f90: 3
```

همچنین برای مشاهده مقدار یک ثبات خاص می‌توان از دستور `info registers <reg_name>` استفاده کرد:

```
(gdb) info registers eax
eax            0x3                 3
```

5. نمایش وضعیت ثبات‌ها و متغیرهای محلی؛ رجیسترهای edi و esi

با استفاده از دستور `info register` می‌توان وضعیت ثبات‌ها را مشاهده کرد. علاوه بر آن از مخفف این دستور یعنی `i r` نیز می‌توان استفاده کرد. خروجی این دستور به شرح زیر می‌باشد:

```
(gdb) info registers
eax            0x0                0
ecx            0x0                0
edx            0x0                0
ebx            0x82               130
esp            0x8010b500         0x8010b500 <stack+3904>
ebp            0x8010b508         0x8010b508 <stack+3912>
esi            0x80113540         -2146355904
edi            0x80112fa4         -2146357340
eip            0x80103bf5         0x80103bf5 <mycpu+21>
eflags         0x46               [ IOPL=0 ZF PF ]
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 ]
cr8            0x0                0
efer           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}
xmm2           {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}
xmm3           {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}
xmm4           {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}
xmm5           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 =
--Type <RET> for more, q to quit, c to continue without paging--c
```

برای مشاهده متغیرهای محلی نیز می‌توان از دستور `info locals` استفاده کرد. خروجی این دستور پس برای اشکال‌زدایی فایل cat.c به صورت زیر می‌باشد:

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

ثبات SI مخفف Source Index بوده و برای اشاره به یک مبدا در عملیات stream به کار می‌رود. DI نیز مخفف Destination Index بوده و برای اشاره به یک مقصد در عملیات stream به کار می‌رود. E در ابتدای اسامی این ثبات‌ها به معنی Extended بوده و در حالت 32 بیت به کار می‌رود. SI به عنوان نشانگر داده و به عنوان مبدا در برخی عملیات مربوط به رشته‌ها استفاده می‌شود. DI نیز به عنوان نشانگر داده و مقصد برخی عملیات مربوط به رشته‌ها استفاده می‌شود.

6. ساختار struct input

این struct در فایل console.c تعریف شده است و برای خط ورودی کنسول سیستم عامل استفاده می‌شود. این استراکت در کد چنین تعریف شده است:

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

یعنی از یک instance به نام input از یک unnamed struct استفاده می‌شود. این را در GDB هم می‌توان با کامند `ptype` برای پرینت کردن تایپ یک متغیر مشاهده کرد:

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

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

متغیرهای دیگر عدد هستند و هر کدام ایندکس‌ای را برای buf را مشخص می‌کنند.

- متغیر w محل شروع نوشتن خط ورودی کنونی در buf است.
- متغیر e محل کنونی کرسر در خط ورودی است.
- متغیر r برای خواندن buf استفاده می‌شود. (از w قبلی شروع می‌کند)

نحوه تغییر این متغیرها را با یک مثال می‌بینیم:

در ابتدای کار مقادیر اولیه متغیرها را پرینت می‌کنیم و یک breakpoint در تابع `consoleintr` در انتهای بخش default، (جایی که اینتر یا ctrl+d زده می‌شود یا کرسر از buf فراتر می‌رود) می‌گذاریم:

```
(gdb) print input
$1 = {buf = '\000' <repeats 127 times>, r = 0, w = 0, e = 0}
(gdb) break console.c:340
Breakpoint 1 at 0x80100dc6: file console.c, line 340.
```

حال continue می‌کنیم و عبارت test را وارد می‌کنیم:

```
(gdb) print input
$2 = {buf = "test\n", '\000' <repeats 122 times>, r = 0, w = 5, e = 5}
```

طبق breakpoint ای که گذاشتیم اجرای برنامه متوقف می‌شود.

می‌بینیم که ورودی در buf قرار گرفته و متغیر e به 5 تغییر یافته که مکان بعد از آخرین حرف buf است.

حال دوباره continue کرده و دستی (با ctrl+c) روند اجرا را متوقف می‌کنیم:

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

می‌بینیم که مقدار r به همان مقدار w رسیده است. یعنی از w قبلی (که 0 بود) شروع کرده و به w کنونی می‌رسد تا کل خط را بخواند. (با گذاشتن یک watchpoint می‌توان دقیق‌تر بررسی کرد که r یکی یکی جلو می‌رود)

این بار عبارت another را وارد می‌کنیم:

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

w باز هم به آخر buf رفته و e هم در ابتدای خط ورودی جدید است پس با w برابر است.

اگر برنامه را continue و سپس متوقف کنیم، می‌بینیم که r به w می‌رسد:

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

حال continue کرده و عبارت xyz را می‌نویسیم ولی اینتر نمی‌زنیم و دستی برنامه را متوقف می‌کنیم:

```
(gdb) print input
$4 = {buf = "test\nanother\nxyz", '\000' <repeats 111 times>, r = 13, w = 13, e = 16}
```

طبق انتظار متغیر e جلو رفته است. اگر کاراکتر آخر را پاک کنیم:

```
(gdb) print input
$5 = {buf = "test\nanother\nxyz", '\000' <repeats 111 times>, r = 13, w = 13, e = 15}
```

e یک واحد به عقب بر می‌گردد.

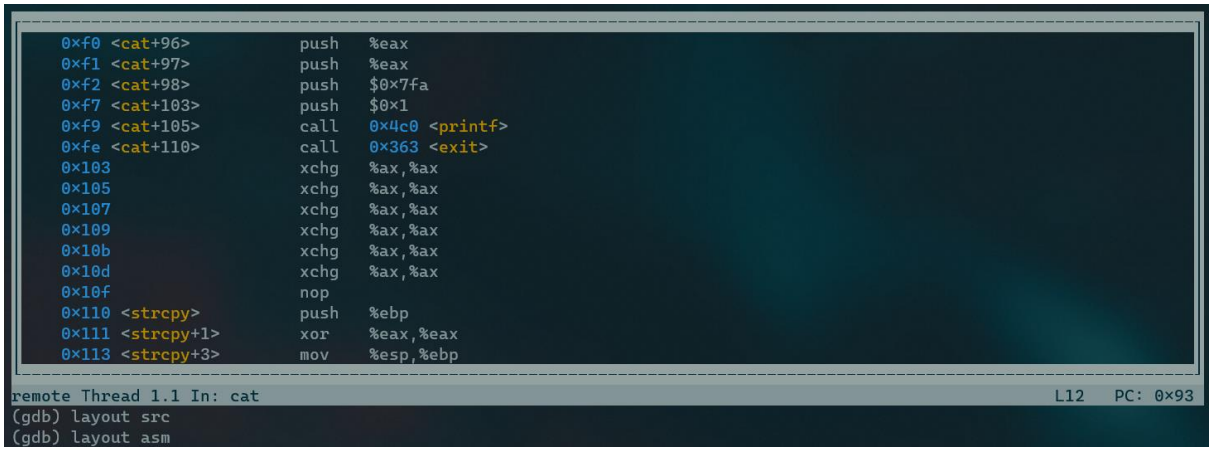
توجه که هر حرکت کرسر خود 3 کاراکتر در این بافر می‌ریزد و مقدار e را افزایش دهد حتی اگر رو به عقب باشد.

اشکال زدایی در سطح کد اسمبلی

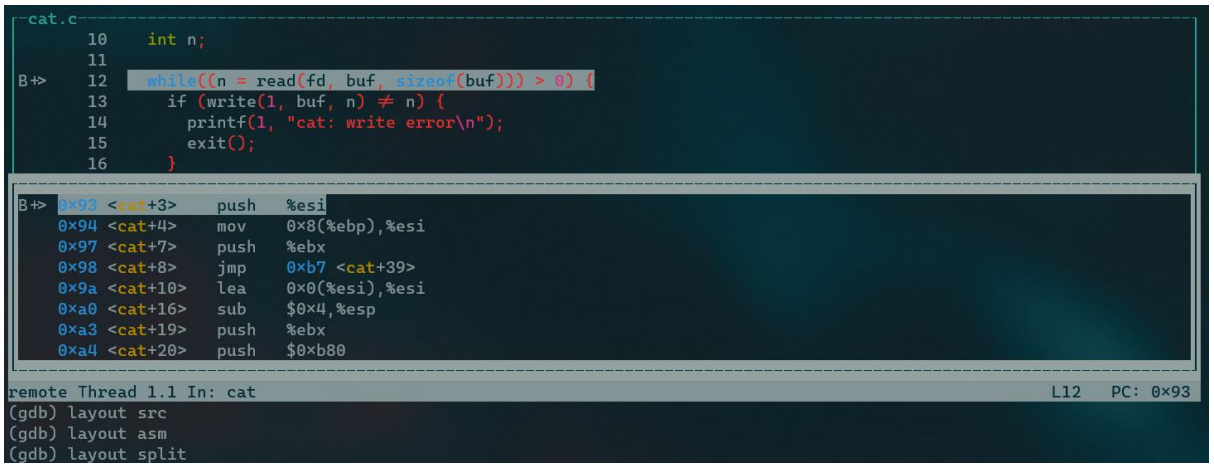
7. خروجی دستورهای layout src و layout asm در TUI

در محیط TUI با استفاده از دستور `layout src` می‌توان کد سورس برنامه در حال دیباگ را نمایش داد:

همچنین با استفاده از دستور `layout asm` می‌توانیم کد اسمبلی برنامه در حال دیباگ را مشاهده کنیم:

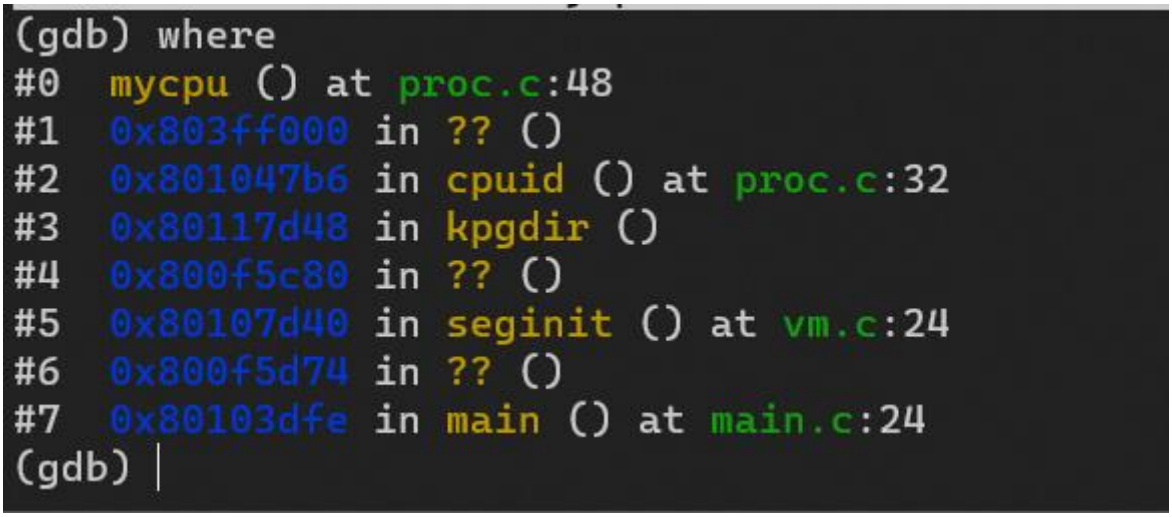


در نهایت با استفاده از دستور `layout split` می‌توانیم کد سورس برنامه و اسمبلی آن را به طور همزمان مشاهده کنیم:



8. دستورهای جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف)

برای وضعیت مشاهده پشته فراخوانی فعلی می‌توان از دستور `where` یا `backtrace` در محیط کاربری TUI استفاده کرد. در اینجا یک breakpoint در خط 48 فایل proc.c گذاشته‌ایم و پس از توقف اجرا در این نقطه، با استفاده از دستور `where` پشته فراخوانی را مشاهده می‌کنیم:



برای حرکت در پشته فراخوانی می‌توان از دستورات `up <n>` و `down <n>` یا مخفف‌های آن‌ها به ترتیب `u` و `d` استفاده کرد. `n` در اینجا مشخص می‌کند که چند تابع بالاتر یا پایین‌تر در پشته برویم. در صورتی که `n` مشخص نشود، به صورت پیش فرض یک فرض می‌شود. برای مثال با دستور `up 2` به تابع `cupid` در خط 32 فایل proc.c می‌رویم:

```
proc.c
   25          {
   26            initlock(&ptable.lock, "ptable");
   27          }
   28
   29          // Must be called with interrupts disabled
   30          int
   31          cpuid() {
  >32            return mycpu()-cpus;
   33          }
   34
   35          // Must be called with interrupts disabled to avoid the caller being
   36          // rescheduled between reading lapicid and running through the loop.
   37          struct cpu*
   38          mycpu(void)
   39          {
   40            int apicid, i;
   41
remote Thread 1.1 In: cpuid                                          L32   PC: 0x801047b6
#1  0x803ff000 in ?? ()
#2  0x801047b6 in cpuid () at proc.c:32
#3  0x80117d48 in kpgdir ()
#4  0x800f5c80 in ?? ()
#5  0x80107d40 in seginit () at vm.c:24
#6  0x800f5d74 in ?? ()
#7  0x80103dfe in main () at main.c:24
(gdb) up 2
#2  0x801047b6 in cpuid () at proc.c:32
(gdb)
```