گزارش کار تمرین کامپیوتری اول (xv6 os)

گروه 9

سجاد تقی زاده 810102425

دنیز مصطفی زادگان 810102603

محمد حسین شفیعی 810102470

- Q1: سه وظیفه اصلی سیستم عامل را نام ببرید:

1- یک واسط بین hardware و software هست. برنامه ها اگر بخوان از منابع سخت افزاری استفاده کنن باید این رو از سیستم عامل درخواست کنن.
2- مدیریت منابع , مدیریت لایه زیرین که از هر منبع چجوری استفاده کنیم . مانند حافظه به شکلی که منابع هدر نروند .
3-مدیریت کاربر, به نیاز کاربران توجه کنه و به نسبت درخواست هاشون ,اولویت هاشون توجه کند .

- Q2: آیا وجود سیستم عامل در تمام دستگاه ها الزامی است؟ چرا؟ در چه شرایطی استفاده از سیستم عامل لازم است؟

وجود سیستم‌عامل در تمام دستگاه‌ها الزامی **نیست**:

۱-مواردی که سیستم‌عامل لازم نیست:
در دستگاه‌هایی که فقط یک کار ثابت و ساده انجام می‌دهند و نیازی به اجرای چند برنامه هم‌زمان ندارند، معمولاً سیستم‌عامل وجود ندارد.
در این حالت، برنامه‌ی اصلی مستقیماً با سخت‌افزار در ارتباط است.
مثل : ماشین‌حساب ساده , Embedded Systems که تنها یک وظیفه دارند.
در این دستگاه‌ها، برنامه‌ی اصلی به‌طور مستقیم روی سخت‌افزار نوشته می‌شود و دیگر نیازی به مدیریت پیچیده‌ی منابع یا چندوظیفگی نیست.
چون فقط یک برنامه اجرا می‌شود و سخت‌افزار محدود است، وجود سیستم‌عامل فقط باعث افزایش پیچیدگی و مصرف حافظه می‌شود.
a

۲. مواردی که سیستم‌عامل **لازم است :**

وقتی دستگاه یا سیستم باید چند کار مختلف را به‌صورت هم‌زمان یا متوالی انجام دهد، یا کاربران و برنامه‌های مختلف از منابع مشترک استفاده کنند، وجود سیستم‌عامل ضروری می‌شود.

وقتی چند برنامه باید هم‌زمان اجرا شوند , مثل : مرورگر.

وقتی منابع سخت‌افزاری باید بین چند برنامه یا کاربر تقسیم شوند , مثل : CPU، حافظه، دیسک.

وقتی امنیت و ایزوله‌سازی برنامه‌ها اهمیت دارد تا خطا یا خرابی یک برنامه باعث از کار افتادن کل سیستم نشود.

وقتی برنامه‌ها باید با هم تعامل داشته باشند

- Q3: معماری سیستم عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

سیستم‌عامل xv6 با معماری Monolithic Kernel طراحی شده.در این معماری، تمام اجزای اصلی سیستم‌عامل در Kernel Space اجرا می‌شوند و با سطح دسترسی کامل به سخت‌افزار فعالیت می‌کنند. در این معماری، تمام بخش‌های سیستم‌عامل:

مدیریت حافظه (Memory Management)

زمان‌بندی و کنترل فرآیندها (Scheduler)

فایل‌سیستم (File System)

درایورهای سخت‌افزار (Device Drivers)

و مکانیزم فراخوانی سیستمی (System Call Handler)

در Kernel Mode و با دسترسی مستقیم به منابع سخت‌افزار اجرا می‌شوند

تمام فرآیندهای کاربر در xv6 فضای آدرس مخصوص خود را دارند

فراخوانی‌هایی مانند fork, exec, wait, read, write همه از طریق مکانیزم system call وارد کرنل میشن

در monolithic kernel ، بخش‌های مختلف سیستم می‌توانند به‌صورت مستقیم با هم در ارتباط باشند و از داده‌ها یا توابع یکدیگر استفاده کنند.Scheduler می‌تواند مستقیماً وضعیت حافظهٔ هر فرایند را بررسی کند.

چون تمام اجزای سیستم‌عامل درون یک فضای کرنل اجرا می‌شوند، هنگام انجام عملیات‌هایی مثل خواندن از دیسک یا تخصیص حافظه، نیازی به Message Passing نیست. بنابراین، کرنل یکپارچه معمولاً عملکرد سریع‌تری نسبت به میکروکرنل‌ها دارد.

- Q4: سیستم عامل xv6 یک سیستم تک وظیفه ای است یا چند وظیفه ای؟

سیستم‌عامل xv6 یک Multitasking Operating System است . یعنی xv6 می‌تواند چندین فرآیند را به‌صورت هم‌زمان اجرا کند با یک cpu .

این سیستم میتواند چند Process را به طور هم زمان در حافظه نگه دارد و و CPU را بین آن‌ها به صورت Time-Sharing تقسیم کند. در نتیجه کاربر حس می‌کند چند برنامه به‌طور هم‌زمان در حال اجرا هستند، در حالی که در واقع سیستم‌عامل با سرعت زیاد بین آن‌ها سوئیچ می‌کند.

در xv6 هر فرآیند می‌تواند با fork فرآیند جدیدی بسازد و سپس با wait منتظر پایان آن بماند

این یعنی در هر لحظه ممکن است چند فرآیند فعال و در حال اجرا باشند.

در این سیستم هر فرآیند دارای مجموعه‌ای ازregisters، پشته، و فضای آدرس مخصوص به خود است.

کرنل در هنگام سوئیچ، وضعیت cpu را ذخیره می‌کند و وضعیت فرآیند بعدی را بارگذاری می‌کند.

این فرآیند بارها تکرار می‌شود و باعث می‌شود چندین برنامه به نوبت اجرا شوند.

- Q5: همانطور که میدانید به طور کلی چند وظیفگی تعمیمی است از حالت چند برنامگی چه تفاوتی میان یک برنامه و یک پردازه وجود دارد؟

در multiprogramming، چند program روی دیسک وجود دارند که سیستم‌عامل آن‌ها را یکی‌یکی در حافظه بارگذاری می‌کند تا CPU همیشه مشغول باشد.

در multitasking همان چند program به‌صورت هم‌زمان در حال اجرا هستند و CPU میان آن‌ها سوئیچ می‌کند.

program :

مجموعه ای از کد ها توی دیسک که هنوز اجرا نشدن

روی دیسک است

ماهیت static

از این می توان چند نسخه تولید کرد

منبع به آن داده نشده

process :

اجرای یک برنامه در حافظه که منابعی به ان داده شده مانند پشته و داده ها

روی حافظه اصلی ram هست

ماهیت dynamic

هر نسخه ان منحصر به فرد در حال اجرا است

منابعی متل حافظه فایل ها cpu به ان داده شده است

- Q6: ساختار یک پردازه در سیستم عامل xv6 از چه بخش هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازه های مختلف اختصاص می دهد؟

در سیستم‌عامل xv6 , هر Process یک موجودیت مستقل است که نمایندهٔ اجرای یک برنامه در حافظه است.

کرنل xv6 برای هر process ساختار دادهٔ خاصی نگهداری می‌کند و با استفاده از Scheduler تصمیم می‌گیرد که در هر لحظه کدام روی CPU اجرا شود.

در xv6، اطلاعات مربوط به هر process در ساختاری به نام struct proc ذخیره می‌شود.

Pid : شناسه پردازه process id

State : وضعیت process مانند zombie , sleeping

Parent : پدر این process . اشاره گر به والد

Sz : حافظه اختصاص داده شده

Kstack : فضای اجرای توابع در کرنل

Context : رجیستر هایی برای بازگشت به حالت اولیه بعد از سوییچ

سیستم‌عامل xv6 از یک Scheduler ساده و غیراولویتی استفاده می‌کند .

در هر CPU یک حلقهٔ بی‌نهایت وجود دارد که در آن کرنل دائماً بین پردازه‌ها گردش می‌کند.

Scheduler در آرایه‌ای به دنبال پردازه‌هایی می‌گردد که وضعیتشان RUNNABLE باشد.

وقتی چنین process پیدا شد > کرنل CPU را به آن پردازه اختصاص می‌دهد . context را بارگذاری می‌کند حالت آن را به RUNNING تغییر می‌دهد . وقتی زمان process تمام شد یا پردازه خود وارد حالت SLEEPING شد، زمان‌بند CPU را به پردازهٔ بعدی می‌دهد.

اگر چند CPU داشتیم در این حالت، هر CPU نسخهٔ جداگانه‌ای از تابع scheduler() را اجرا می‌کند . و process به‌طور هم‌زمان اجرا می شوند . برای جلوگیری از تداخل از Spinlocks استفاده می‌کند تا دسترسی هم‌زمان به جدول پردازه‌ها ایمن باشد.

- Q7: مفهوم file descriptor در سیستم عامل های مبتنی برUNIX چیست؟ عملکرد pipe در سیستم عامل xv6 چگونه است و به طور معمول برای چه هدفی استفاده می شود؟

در سیستم‌عامل‌های یونیکسی مثل xv6، وقتی یک برنامه می‌خواهد با فایل، صفحه‌نمایش، یا هر دستگاه ورودی/خروجی کار کند، خودش مستقیم با سخت‌افزار حرف نمی‌زند.
در عوض، کرنل برایش یک «شماره» صادر می‌کند به نام File Descriptor

Pipe روشی است برای فرستادن داده از یک برنامه به برنامه‌ی دیگر بدون استفاده از فایل موقت.

وقتی shell دستور(ls | grep txt)رو اجرا می‌کند:

1. کرنل با دستور pipe(fd) یک Pipe درست می‌کند که دو عدد File Descriptor برمی‌گرداند:
   1 fd[0] برای خواندن
   1 fd[1] برای نوشتن
2. shell دو فرآیند می‌سازد (یکی برای ls و یکی برای grep)
   در ls : خروجی استاندارد (عدد 1) به fd1 وصل می‌شود.
   پس هر چیزی که ls می‌نویسد، وارد pipe می‌شود.
   در grep : ورودی استاندارد (عدد 0) به fd0 وصل می‌شود.
   پس grep داده را مستقیماً از pipe می‌خواند.
3. در پایان، وقتی ls تمام می‌شود و انتهای نوشتن بسته می‌شود، grep متوجه می‌شود pipe خالی شده و اجرای آن هم تمام می‌شود.

هدف از pipe :

Communication . برای اتصال خروجی یک برنامه به ورودی دیگری (Pipeline) و برای جلوگیری از ساخت فایل موقت روی دیسک

- Q8: فراخوانی های سیستمی exec و fork در سیستم عامل xv6 چه عملیاتی را انجام میدهند؟ از نظر طراحی، ادغام نکردن این دو چه مزیتی دارد؟

fork() : برای ایجاد یک فرآیند جدید (کپی از پدر) . تمام متغیرها، فایل‌های باز (File Descriptors)، و فضای حافظه دقیقاً مثل فرآیند پدر کپی می‌شوند . فقط شناسهٔ پردازه (PID) و مقدار بازگشتی فرق دارند

(در پدر مقدار بازگشتی = PID فرزند / در فرزند مقدار بازگشتی = 0)

exec(): برای جایگزین کردن برنامهٔ جدید در همان فرآیند . PID و parent حفظ می‌شوند. فایل اجرایی مشخص‌شده را از دیسک می‌خواند / محتوای حافظهٔ فعلی را پاک می‌کند/ کد و دادهٔ جدید را در حافظه بارگذاری می‌کند / Page Table جدید برای آن برنامه می‌سازد / شمارندهٔ برنامه (PC) را روی نقطهٔ شروع برنامهٔ جدید قرار می‌دهد

جدا بودن این دو باعث می‌شود قبل از اجرای برنامه، بتوان ورودی/خروجی را تنظیم کرد . هرکدام کار خودشان را ساده و مشخص انجام می‌دهند .

- Q10: کاربرد متغیرهای UPROGS و ULIB در Makefile چیست؟

این دو متغیر نقش مهمی در فرآیند ساخت برنامه‌های سطح کاربر و فایل سیستم xv6 ایفا می‌کنند.

UPROGS (User Programs – برنامه‌های سطح کاربر):

کاربرد: این متغیر لیستی از تمام برنامه‌های سطح کاربری را تعریف می‌کند که باید کامپایل شده و در نهایت در فایل سیستم مجازی xv6 قرار گیرند. به عبارت دیگر، UPROGS مشخص می‌کند که چه دستوراتی در شل xv6 برای کاربر قابل دسترس باشند.

ULIB (User Library – کتابخانه سطح کاربر):

کاربرد: این متغیر مجموعه‌ای از فایل‌های آبجکت (.o) کتابخانه‌ای مشترک را تعریف می‌کند که تمام برنامه‌های سطح کاربر برای اجرا به آن‌ها نیاز دارند. این کتابخانه شامل توابع پایه‌ای مانند فراخوانی‌های سیستمی، مدیریت حافظه و چاپ فرمت‌بندی شده است. با استفاده از ULIB، از نوشتن کدهای تکراری در هر برنامه جلوگیری می‌شود.

- Q11: تفاوت دستور cd با ls و cat و محل اجرای آن چیست؟

محل اجرای cd

دستور cd برخلاف ls و cat که برنامه‌هایی مجزا در فایل سیستم هستند، یک دستور داخلی شل است. این یعنی منطق دستور cd مستقیماً در داخل کد خود شل پیاده‌سازی شده و به عنوان یک پروسه جداگانه اجرا نمی‌شود.

دلیل این تفاوت

دلیل این طراحی به مفهوم Process Environment و به خصوص دایرکتوری کاری فعلی برمی‌گردد.

1. **پروسه‌های جدا از هم :** وقتی شما دستوری مانند ls را اجرا می‌کنید، شل یک پروسه **فرزند** با استفاده از fork() ایجاد می‌کند. سپس این پروسه فرزند، کد برنامه ls را با exec() اجرا می‌کند. پروسه ls کار خود را انجام می‌دهد و سپس خارج می‌شود. تمام تغییراتی که این پروسه در محیط خود ایجاد کرده، با خروجش از بین می‌رود و پروسه **والد** ، یعنی خود شل، هیچ تغییری نکرده است.
2. **مشکل cd به عنوان برنامه خارجی:** حال تصور کنید cd هم مانند ls یک برنامه خارجی بود. وقتی شما cd / x را اجرا می‌کردید، شل یک پروسه فرزند fork می‌کرد. این پروسه فرزند برنامه cd را exec می‌کرد. سپس برنامه cd با موفقیت دایرکتوری کاری خودش را به /x تغییر می‌داد و بلافاصله خارج می‌شد. با خروج این پروسه، تمام محیط آن از جمله دایرکتوری کاری جدیدش نابود می‌شد و شما به شل اصلی برمی‌گشتید که دایرکتوری کاری آن هیچ تغییری نکرده بود. در نتیجه این دستور کاملا بی‌فایده می‌شد.
3. **راه حل: دستور داخلی:** برای اینکه دایرکتوری کاری خود شل تغییر کند، خود پروسه شل باید فراخوانی سیستمی chdir() را اجرا کند. به همین دلیل cd باید به عنوان یک دستور داخلی در خود شل پیاده‌سازی شود تا بتواند محیط پروسه خودش را تغییر دهد و این تغییر برای دستورات بعدی که در همان شل اجرا می‌کنید، باقی بماند.

- Q9: دستور make -n را اجرا نمایید. کدام دستور، فایل نهایی هسته را می‌سازد؟

وقتی شما make -n را اجرا می‌شود، make هیچ دستوری را اجرا نمی‌کند، بلکه فقط دستوراتی را که قرار بود برای رسیدن به هدف مشخص شده اجرا کند، روی صفحه چاپ می‌کند. این کار به ما اجازه می‌دهد تا بدون کامپایل واقعی پروژه، ببینیم چه اتفاقی در پشت می‌افتد.

برای یافتن دستوری که فایل نهایی کرنل را می‌سازد، باید به دنبال تارگت کرنل در Makefile و دستورات مربوط به آن بگردیم.

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

1. **خط اول (وابستگی‌ها):**
   - این خط می‌گوید که برای ساختن kernel، ابتدا باید تمام فایل‌های لیست شده در متغیر OBJS ، به علاوه فایل‌های entry.o, entryother, initcode و kernel.ld آماده باشند.
2. **خط دوم (دستور ساخت هسته):**
   - این همان دستوری است که فایل نهایی هسته را می‌سازد.
   - $(LD): این متغیر به لینکر اشاره دارد معمولاً ld. وظیفه لینکر این است که فایل‌های آبجکت مختلف را که هر کدام بخشی از برنامه هستند، به یکدیگر متصل کرده و یک فایل اجرایی واحد بسازد.
   - -o kernel: این گزینه به لینکر می‌گوید که نام فایل خروجی نهایی، kernel باشد.
   - $(OBJS) entry.o ...: این بخش، تمام فایل‌های آبجکت کامپایل شده را به عنوان ورودی به لینکر می‌دهد.
   - -T kernel.ld: این گزینه به لینکر می‌گوید که از اسکریپت kernel.ld برای Memory Layout فایل اجرایی استفاده کند. این اسکریپت مشخص می‌کند که بخش‌های مختلف کد و داده در چه آدرس‌هایی از حافظه قرار بگیرند.

در نتیجه :

با اجرای make -n kernel (یا make -n اگر kernel هدف باشد)، دستوری که به عنوان سازنده اصلی فایل نهایی هسته چاپ می‌شود، دستور لینکر (ld) است.

- Q12:در xv6، در سکتور نخست دیسک قابل بوت (Boot Sector)، محتوای چه فایلی قرار دارد؟

در سیستم‌عامل آموزشی xv6، وقتی سیستم boot می‌شود، اولین چیزی که CPU اجرا می‌کند، کد بوت (boot loader) است که در سکتور صفر دیسک (Boot Sector) قرار دارد. وقتی رایانه روشن می‌شود سکتور صفر دیسک را می‌خواند. همان را در آدرس حافظه‌ی 0x7C00 بارگذاری و اجرا می‌کند.ین کد بوت، وظیفه دارد کرنل اصلی xv6 را از دیسک بخواند و در حافظه اجرا کند.محتوای سکتور نخست دیسک (Boot Sector) برابر است با فایل bootblock.

- Q13: نوع فایل دودویی بوت در xv6 چیست؟

سکتور اوّل دیسک بوت حاوی boot loader است که به‌صورت باینری خامِ ساخته می‌شود، نه ELF . دلیلش این است که BIOS دقیقاً همین ۵۱۲ بایت را در آدرس 0x7C00 بارگذاری و اجرای مستقیم می‌کند؛ پس نباید سربارِ هِدرِ قالب‌هایی مثل ELF داشته باشد. در xv6 بوت‌لودر از دو منبع ساخته می‌شود: bootasm.S (اسمبلی 16/32بیتی) و bootmain.c، و کل خروجی داخل همین ۵۱۲ بایت جا می‌گیرد.

چرا از «باینری خام» استفاده شده؟

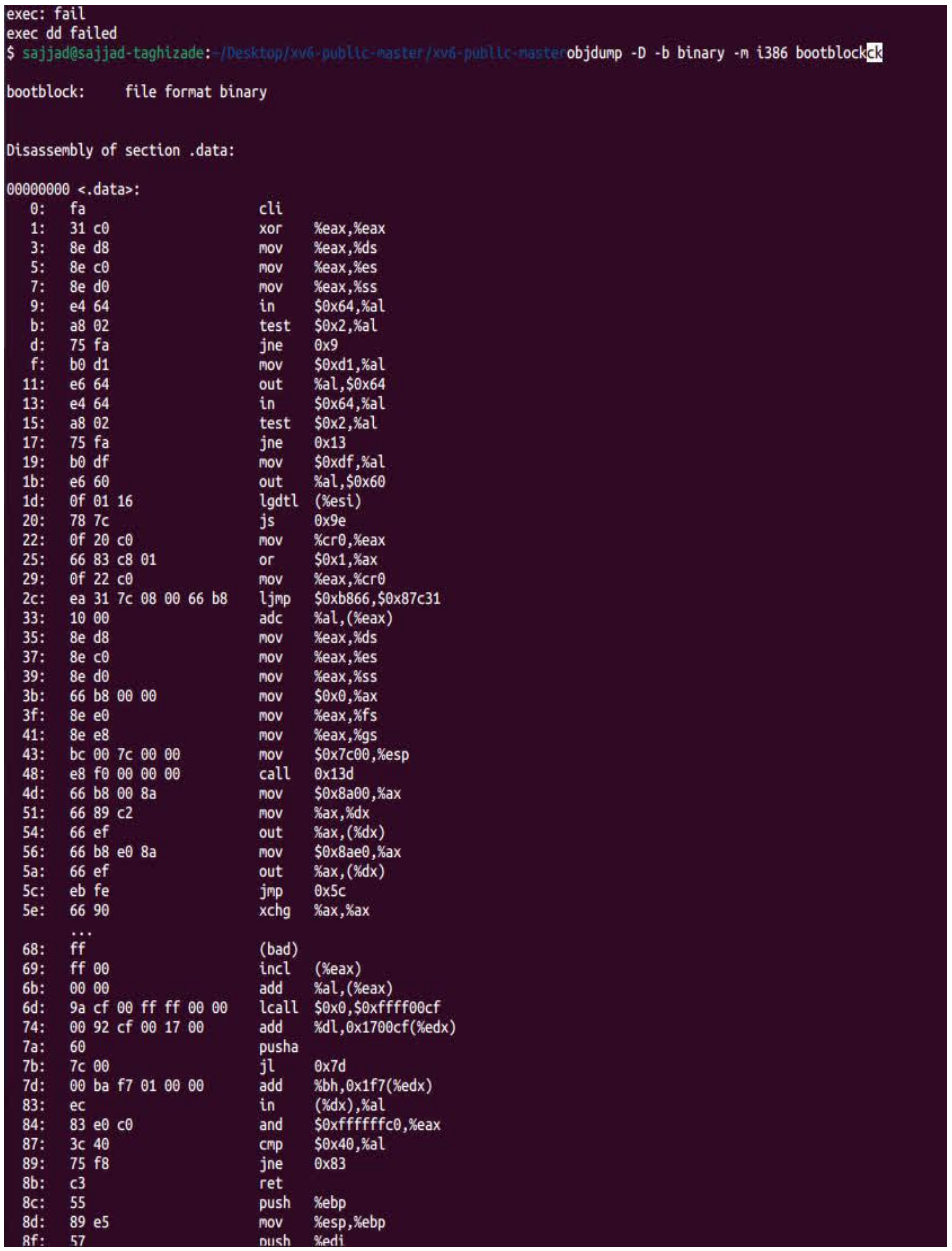
چون BIOS فقط ۵۱۲ بایت اوّل را می‌خوانَد و مستقیماً اجرا می‌کند؛ بوت لودر باید بدون هدر/جداول اضافی در همین فضا جا شود و آمادهٔ پرش به کُد 32 بیتی و بارگذاری کرنل باشد. بوت لودر xv6 عمداً بسیار کوچک نگه داشته شده (حدود چندصد بایت) تا در همین محدودیت جا شود. بوت لودر در این مرحله هنوز فایل سیستم یا قابلیت‌های پیچیده ندارد؛ فقط باید کرنل را از سکتورهای بعدی بخواند و اجرا کند.

تفاوت این نوع باینری با فایل‌های دودویی دیگر

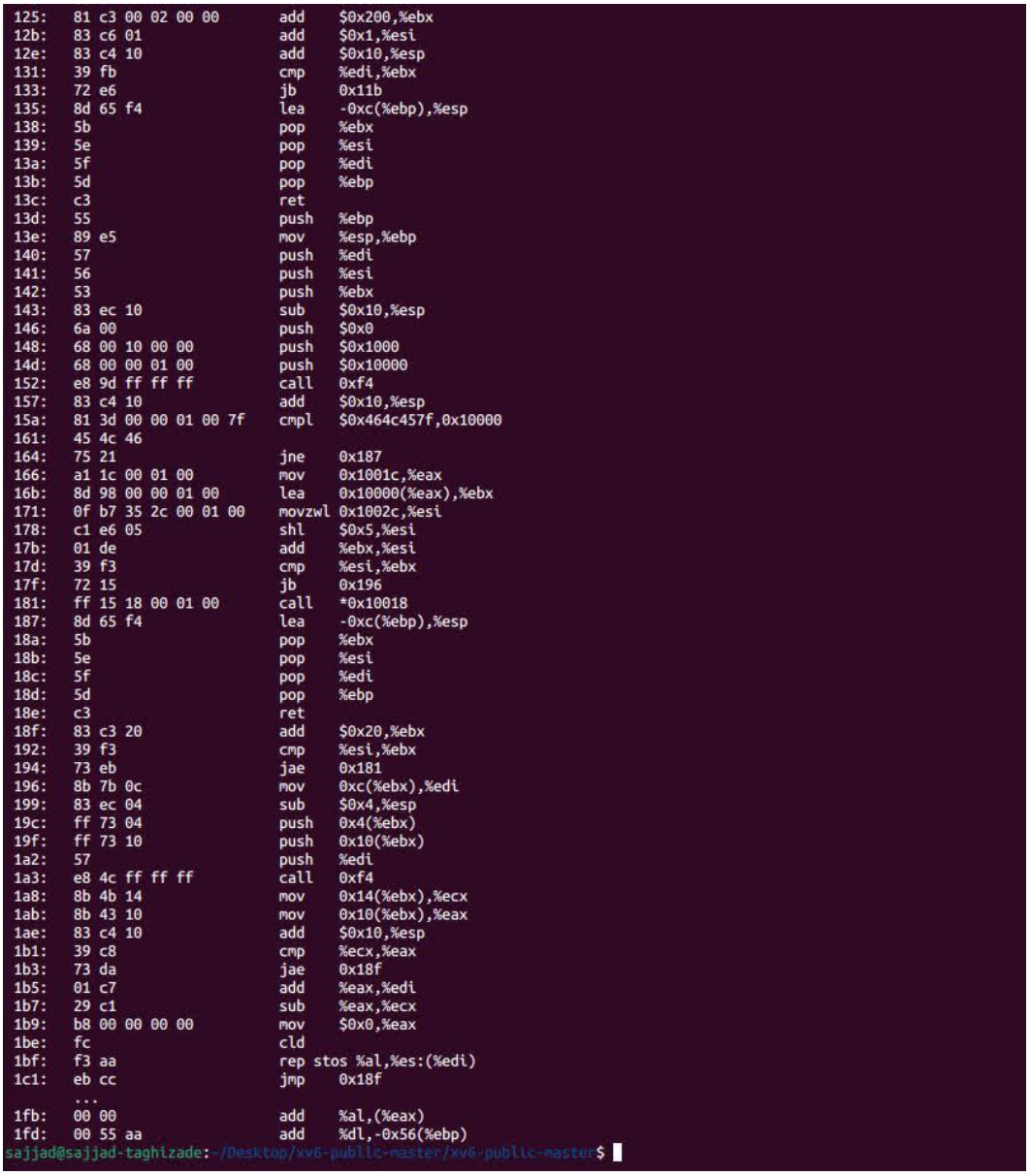
بوت لودر: باینری خام ۵۱۲بایتی بدون هدر، برای اجرای مستقیم توسط BIOS در 0x7C00؛ ابتدا در Real Mode شروع می‌کند، سپس Protected/32-bit را فعال می‌کند و bootmain را صدا می‌زند.

کرنل و برنامه‌های کاربر: همگی elf32-i386 (ELF) هستند؛ exec در کرنل هدر ELF را بررسی و سگمنت‌ها را طبق آدرس‌های ELF در حافظه بارگذاری می‌کند. حتی خودِ کرنل نیز یک ELF با «entry point» مشخص است.

چطور فایل بوت را «اسمبلی‌خوان» کنیم (disassemble) تا قابل فهم انسان شود؟



```
 7b:   7c 00                   jl     0x7d
 7d:   00 ba f7 01 00 00       add    %bh,0x1f7(%edx)
 83:   ec                      in     (%dx),%al
 84:   83 e0 c0                and    $0xffffffc0,%eax
 87:   3c 40                   cmp    $0x40,%al
 89:   75 f8                   jne    0x83
 8b:   c3                      ret
 8c:   55                      push   %ebp
 8d:   89 e5                   mov    %esp,%ebp
 8f:   57                      push   %edi
 90:   53                      push   %ebx
 91:   8b 5d 0c                mov    0xc(%ebp),%ebx
 94:   e8 e5 ff ff ff          call   0x7e
 99:   b8 01 00 00 00          mov    $0x1,%eax
 9e:   ba f2 01 00 00          mov    $0x1f2,%edx
 a3:   ee                      out    %al,(%dx)
 a4:   ba f3 01 00 00          mov    $0x1f3,%edx
 a9:   89 d8                   mov    %ebx,%eax
 ab:   ee                      out    %al,(%dx)
 ac:   89 d8                   mov    %ebx,%eax
 ae:   c1 e8 08                shr    $0x8,%eax
 b1:   ba f4 01 00 00          mov    $0x1f4,%edx
 b6:   ee                      out    %al,(%dx)
 b7:   89 d8                   mov    %ebx,%eax
 b9:   c1 e8 10                shr    $0x10,%eax
 bc:   ba f5 01 00 00          mov    $0x1f5,%edx
 c1:   ee                      out    %al,(%dx)
 c2:   89 d8                   mov    %ebx,%eax
 c4:   c1 e8 18                shr    $0x18,%eax
 c7:   83 c8 e0                or     $0xffffffe0,%eax
 ca:   ba f6 01 00 00          mov    $0x1f6,%edx
 cf:   ee                      out    %al,(%dx)
 d0:   b8 20 00 00 00          mov    $0x20,%eax
 d5:   ba f7 01 00 00          mov    $0x1f7,%edx
 da:   ee                      out    %al,(%dx)
 db:   e8 9e ff ff ff          call   0x7e
 e0:   8b 7d 08                mov    0x8(%ebp),%edi
 e3:   b9 80 00 00 00          mov    $0x80,%ecx
 e8:   ba f0 01 00 00          mov    $0x1f0,%edx
 ed:   fc                      cld
 ee:   f3 6d                   rep insl (%dx),%es:(%edi)
 f0:   5b                      pop    %ebx
 f1:   5f                      pop    %edi
 f2:   5d                      pop    %ebp
 f3:   c3                      ret
 f4:   55                      push   %ebp
 f5:   89 e5                   mov    %esp,%ebp
 f7:   57                      push   %edi
 f8:   56                      push   %esi
 f9:   53                      push   %ebx
 fa:   83 ec 0c                sub    $0xc,%esp
 fd:   8b 5d 08                mov    0x8(%ebp),%ebx
100:   8b 75 10                mov    0x10(%ebp),%esi
103:   89 df                   mov    %ebx,%edi
105:   03 7d 0c                add    0xc(%ebp),%edi
108:   89 f0                   mov    %esi,%eax
10a:   25 ff 01 00 00          and    $0x1ff,%eax
10f:   29 c3                   sub    %eax,%ebx
111:   c1 ee 09                shr    $0x9,%esi
114:   83 c6 01                add    $0x1,%esi
117:   39 fb                   cmp    %edi,%ebx
119:   73 1a                   jae    0x135
11b:   83 ec 08                sub    $0x8,%esp
11e:   56                      push   %esi
```



- Q14: علت استفاده از دستور objcopy در حین اجرای عملیات make چیست؟

یک ابزار از مجموعه‌ی binutils لینوکسه و کارش اینه که فایل قابل‌اجرای «پیچیده» (مثل ELF) رو تبدیل به «کد خام» می‌کنه که مستقیماً توسط سخت‌افزار قابل‌اجرا باشه.در طی فرآیند ساخت xv6، از دستور objcopy برای تبدیل فایل ELF خروجیِ مرحله‌ی لینک (bootblock.o) به یک فایل باینری خام (bootblock) استفاده می‌شود.

این کار لازم است چون BIOS فقط می‌تواند باینری خالص را از سکتور صفر دیسک اجرا کند، نه فایل‌های دارای هدر و اطلاعات اضافی مثل ELF.

در نتیجه، objcopy بخش .text را استخراج کرده و آن را به یک فایل ۵۱۲ بایتی آماده‌ی بوت تبدیل می‌کند.

- Q15 : در فایلهای موجود در xv6، مشاهده میشود که بوت سیستم توسط فایل های bootasm.S و bootmain.cصورت میگیرد. چرا تنها از کد Cاستفاده نشده است؟

وقتی کامپیوتر روشن می‌شود، CPU در حالت 16-بیتی (Real Mode) شروع می‌کند و BIOS فقط ۵۱۲ بایت اول دیسک (Boot Sector) را در آدرس 0x7C00 لود و همان را *مستقیماً* اجرا می‌کند. در این مرحله: هیچ محیط اجرایی C وجود ندارد , استکِ مطمئن تنظیم نشده؛

کارهایی لازم است که فقط با اسمبلی دقیق و کوتاه قابل انجام است : قرار دادن کُد در دقیقاً 0x7C00 و رعایت محدودیت ۵۱۲ بایتی , تنظیم ثبات‌ها ...

بعد از آنکه «سخت‌افزار» به حالت ۳۲بیتی پایدار رسید، نوشتن بقیهٔ منطق به C راحت‌تر است

bootasm.s -> راه‌اندازیِ اولیهٔ خیلی کم‌حجم و دقیقِ سخت‌افزار

bootasm.c -> منطقِ بارگذاری کرنل , خواندن دیسک، پارس ELF، کپی سگمنت‌ها، و پرش به کرنل

- Q16:یک ثبات عام منظوره ،یک ثبات قطعه ،یک ثبات وضعیت و یک ثبات کنترلی در معماریx86 را نام برده و وظیفه هر یک را به طور مختصر توضیح دهید .

General Purpose Registers : **EAX**

برای انجام عملیات محاسباتی، منطقی، و نگه‌داری داده‌ها در حین اجرای برنامه‌ها.

Segment Registers : CS

برای مشخص کردن بخش‌های مختلف حافظه (کد، داده، پشته و غیره).

Status Register : EFLAGS

برای نگهداری پرچم‌ها (Flags) مثل نتیجه‌ی عملیات، علامت، صفر بودن، یا وضعیت وقفه‌ها.

Control Registers : CR0

تنظیم حالت کاری CPU (مثل فعال‌کردن Protected Mode یا Paging).

- Q17:چرا پردازنده‌های x86 هنگام بوت در "مد حقیقی" (Real Mode) شروع به کار می‌کنند؟

ویژگی‌های Real Mode

پردازنده فقط از آدرس‌دهی ۲۰ بیتی (۱ مگابایت حافظه) استفاده می‌کند , هیچ حفاظت حافظه‌ای وجود ندارد , همه‌ی کدها و داده‌ها در حافظه‌ی فیزیکی ساده نگهداری می‌شوند.

CPU در بوت در Real Mode است چون :

BIOS که قبل از سیستم‌عامل اجرا می‌شود، خودش در حالت 16 بیتی نوشته شده است.کد بوت (Bootloader) که در سکتور صفر دیسک قرار دارد نیز باید با BIOS در تعامل باشد . این وقفه‌ها فقط در Real Mode قابل استفاده هستند. بنابراین، CPU باید در همین حالت ساده و سازگار بوت شود.

پردازنده‌های x86 در هنگام بوت در Real Mode شروع می‌شوند تا بتوانند BIOS و کدهای قدیمی را اجرا کنند و سازگاری با پردازنده‌های اولیه 8086 حفظ شود؛ سپس Bootloader سیستم‌عامل آن را به Protected Mode منتقل می‌کند تا از امکانات پیشرفته‌تر استفاده کند.

آیا در پردازنده های دیگر مانند ARM یا RISC-V نیز مدها به همین شکل هستند یا خیر؟ توضیح دهید

ARM و RISC-V مثل x86 نیستند؛ خبری از «Real Mode» نیست.از مدل‌های (privilege/exception levels) استفاده می‌کنند.

x86 از قدیم با Real Mode تکامل یافته و به‌خاطر سازگاری عقب‌رو، CPU پس از روشن شدن در real mode شروع می‌کند.

- Q18:مد حفاظت‌شده (Protected Mode) در پردازنده‌های x86 چیست؟

Protected Mode حالتی از کار پردازنده است که در آن حفاظت حافظه، چندوظیفگی، و سطوح دسترسی (Privilege Levels) فعال می‌شود.

پردازنده بعد از بوت (که در حالت Real Mode است) توسط Bootloader یا سیستم‌عامل به این حالت منتقل می‌شود تا بتواند سیستم‌عامل مدرن را اجرا کند.

وظیفه‌ی اصلی آن چیست و پردازنده چه زمانی وارد این حالت می‌شود؟

جلوگیری از این‌که برنامه‌ای به حافظه‌ی برنامه یا کرنل دیگر دسترسی پیدا کند و

هر برنامه (Process) فضای آدرس خودش را دارد؛ CPU می‌تواند بین آن‌ها سوئیچ کند.

اجرای دستورات حساس فقط در سطح کرنل (Ring 0) مجاز است؛ برنامه‌های کاربر در Ring 3 اجرا می‌شوند.

حافظه فیزیکی به صفحات تقسیم می‌شود تا مدیریت و ایزوله‌سازی راحت‌تر شود

- Q19 : کد bootmain.c هسته را با شروع از یک سکتور بس از سکتورِ بوت، خوانده و در آدرس 0x100000 قرار میدهد . علت انتخاب این آدرس چیست؟ چرا این آدرس از 0شروع نشده است؟

ممکن است در ماشین‌های کوچک در آدرس‌های خیلی بالا اصلاً رم فیزیکی نباشد؛ پس فعلاً نمی‌شود کرنل را همان‌جا ریخت. بازهٔ 0x000A0000 تا 0x00100000 «سوراخ I/O» است و برای کارت گرافیک/بایوس/دستگاه‌ها رزرو شده؛ نباید کرنل را در بازه‌های پایین‌تر که با این ناحیه تداخل پیدا می‌کند بارگذاری کرد .

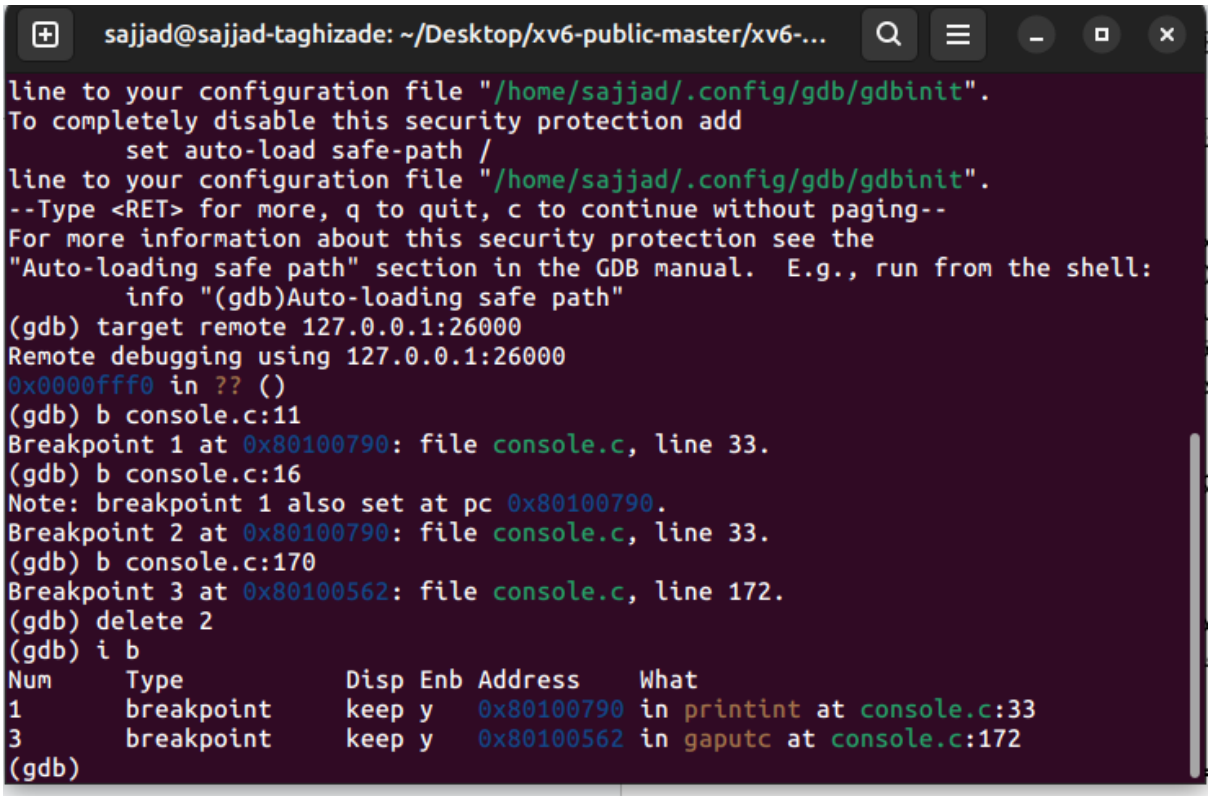
سپس در مرحلهٔ ورودِ کرنل، جدول صفحه‌ای اولیه طوری تنظیم می‌شود که آدرس‌های مجازیِ بالای KERNBASE (یعنی از 0x80100000) به آدرس‌های فیزیکیِ پایین (از 0x00100000) نگاشت شوند؛ این همان جایی است که کرنل توقع دارد کد/دادهٔ خودش را ببیند (لینک‌اسکریپت کرنل هم همین را تعیین کرده است). بعد از فعال شدن paging، کرنل در فضای مجازیِ بالا اجرا می‌شود، در حالی که فیزیکی‌اش همان 1MB است.

چون 0 تا حدود 640KB base memory است که برداشته می‌شود و داده‌های بایوس/جدول‌های قدیمی هم آن حوالی‌اند؛

و 0xA0000–0x100000 هم I/O hole است. بارگذاری کرنل در این محدوده‌ها خطرناک و ناپایدار است. انتخاب 0x00100000 هم ساده و هم ایمن است و با نگاشت‌های بعدیِ صفحه‌بندیِ xv6 سازگار می‌ماند.

- Q20: برای مشاهده Breakpoint ها از چه دستوری استفاده می‌شود؟

برای مشاهده لیست تمام Breakpoint هایی که در یک GDB تنظیم شده‌اند، از دستور info breakpoints یا i b استفاده می‌شود. این دستور اطلاعات کاملی از جمله شماره، نوع، وضعیت و محل دقیق هر Breakpoint را نمایش می‌دهد. مخفف این دستور i b

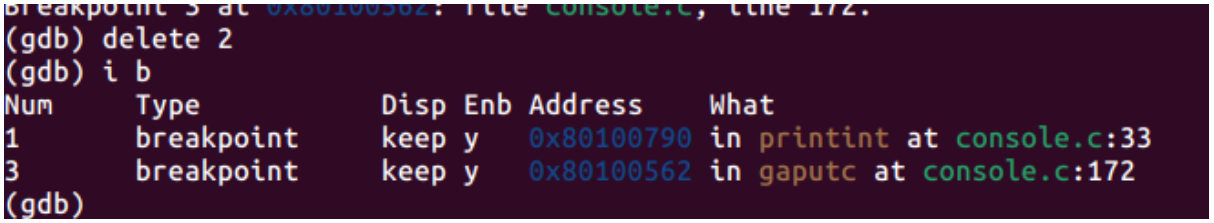


است که استفاده از آن سریع‌تر می‌باشد.

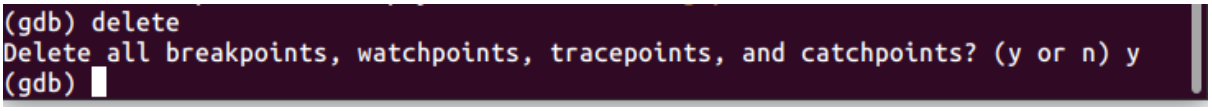
Q21: برای حذف یک Breakpoint از چه دستوری و چگونه استفاده می‌شود؟

برای حذف Breakpointها در GDB از دستور delete (یا مخفف آن d) استفاده می‌شود. این دستور به دو شکل قابل استفاده است:

۱: حذف یک Breakpoint خاص: برای این کار، دستور delete به همراه شماره Breakpoint مورد نظر (که از خروجی دستور info breakpoints به دست می‌آید) به کار می‌رود.
مثلا برای حذف Breakpoint شماره ۲ از دستور زیر را وارد می‌کنیم:

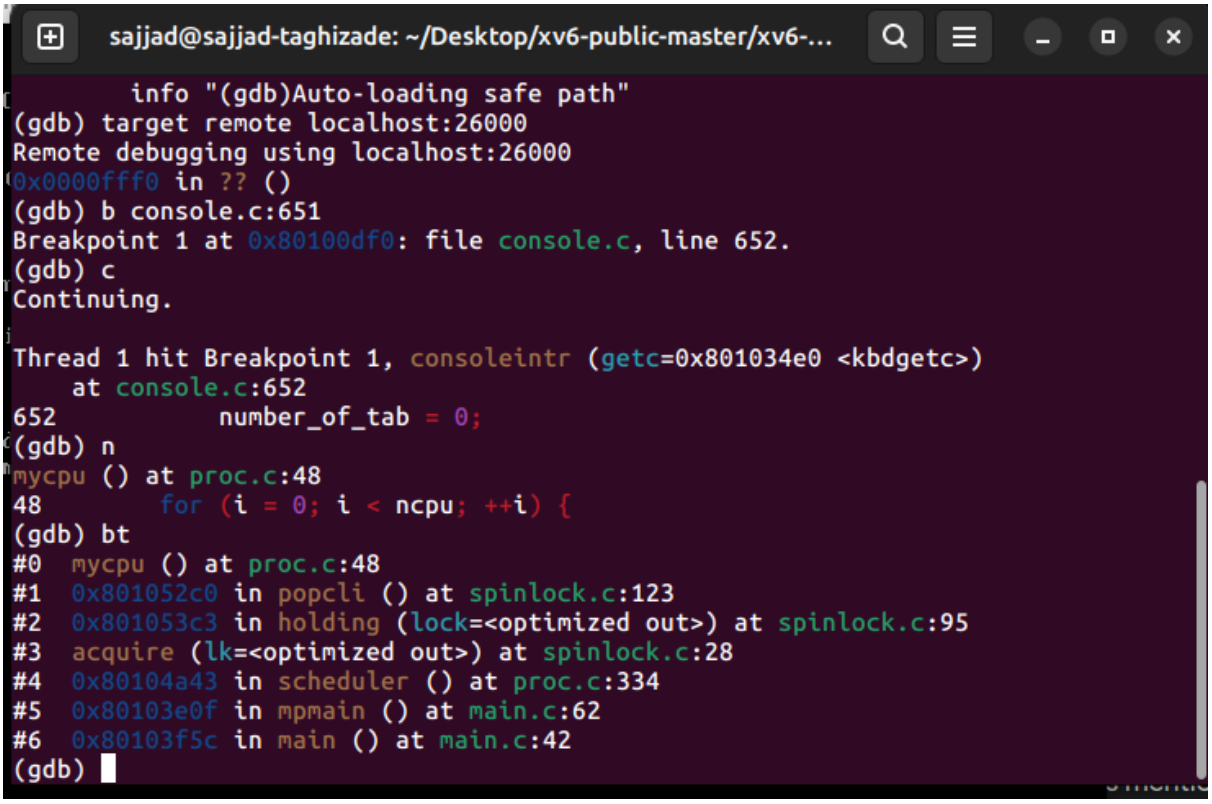


۲: حذف تمام Breakpointها: برای حذف تمام Breakpointها به صورت یک‌جا، می‌توان از دستور delete بدون هیچ آرگومانی استفاده کرد. در این حالت، GDB برای تایید، از کاربر سوال خواهد کرد.



- Q22: : دستور bt چه چیزی را نشان می‌دهد؟

دستور bt مخفف backtrace زنجیره فراخوانی توابع را نشان می‌دهد. این دستور می‌گوید که برای رسیدن به نقطه‌ای که در آن متوقف شده، چه توابعی به چه ترتیبی یکدیگر را فراخوانی کرده‌اند.نشان می‌دهد که "چگونه به اینجا رسیدیم



0 آن تابعی است که در آن متوقف شده ایم.

1 تابع فراخواننده 0 است و2 هم فراخواننده 1 و ///

- Q23: دو تفاوت دستورهای x و print را توضیح دهید. چگونه می‌توان محتوای یک ثبات خاص را چاپ کرد؟

تفاوت‌های x و print:

1. ورودی:

print (یا p): برای نمایش مقدار یک متغیر یا عبارت در ز C طراحی شده است. نام متغیر را به آن می‌دهیم. GDB نوع متغیر را می‌شناسد و آن را به شکل مناسب چاپ می‌کند.

مثال: p struct محتوای یک ساختار را به صورت جز به جز نمایش می‌دهد.

مثال: p string خود رشته را نمایش می‌دهد

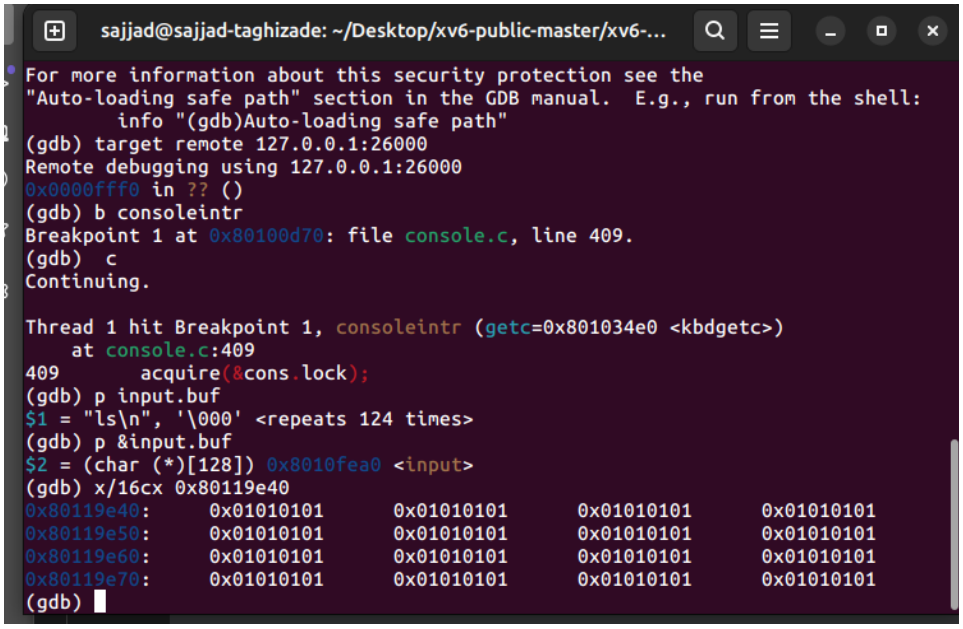
x مخفف examine: برای نمایش محتوای خام یک آدرس حافظه طراحی شده است. یک آدرس به آن می‌دهیم و به آن می‌گویید که داده‌های آن آدرس را با چه فرمتی (مثلاً عدد صحیح، کاراکتر، یا رشته) به شما نشان دهد.

مثال: x/4wx 0x1000 چهار کلمه را به صورت هگزادسیمال از آدرس حافظه 0x1000 نمایش می‌دهد.

2. قالب‌بندی خروجی:

print: خروجی را بر اساس نوع داده متغیر در کد C قالب‌بندی می‌کند. (مثلاً برای int عدد، برای char* رشته و برای struct اعضای آن را نمایش می‌دهد).

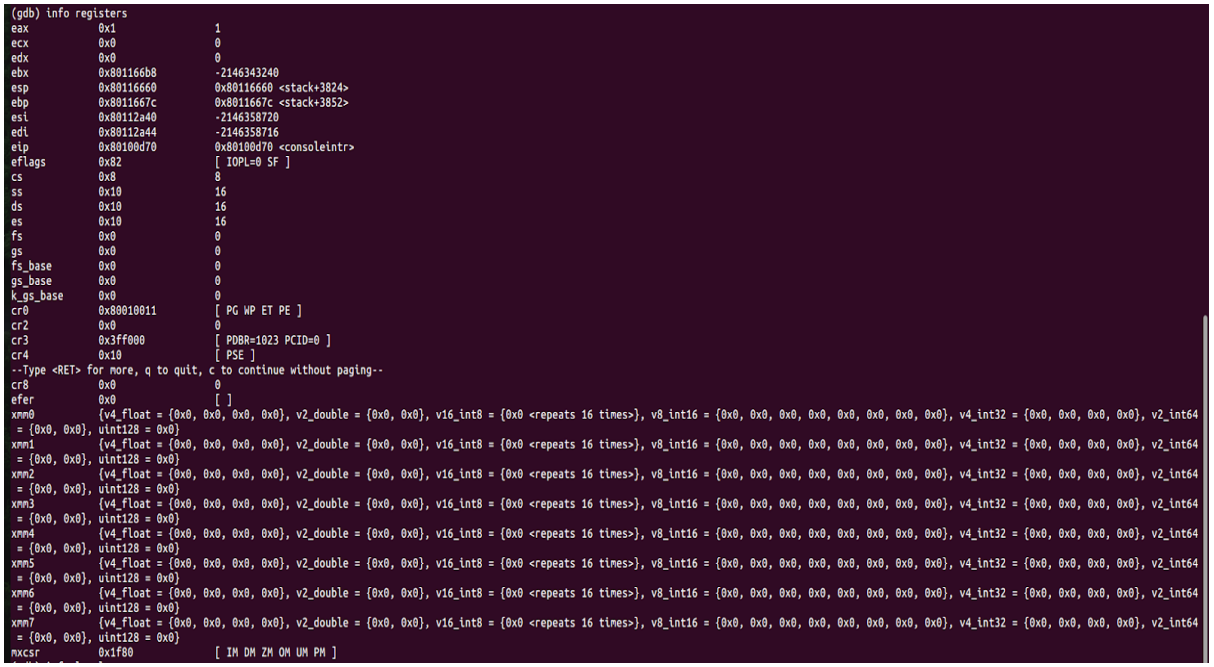
x: خروجی را بر اساس فرمت دلخواه شما نمایش می‌دهد. می‌توان با استفاده از / فرمت‌های مختلفی مانند d (decimal), x (hex), s (string), i (instruction) و... را مشخص کرد.



- Q24:برای نمایش وضعیت ثبات‌ها از چه دستوری استفاده می‌شود؟ برای متغیرهای محلی چطور؟ ... همچنین در گزارش خود توضیح دهید که در معماری x86 رجیسترهای edi و esi نشانگر چه چیزی هستند؟

نمایش وضعیت Registers:

- ☐ دستور اصلی برای نمایش تمام Registers عمومی، info registers است. این دستور مقادیر فعلی Registers مانند eax, ebx, eip, esp را نمایش می‌دهد.



نمایش Local Variables:

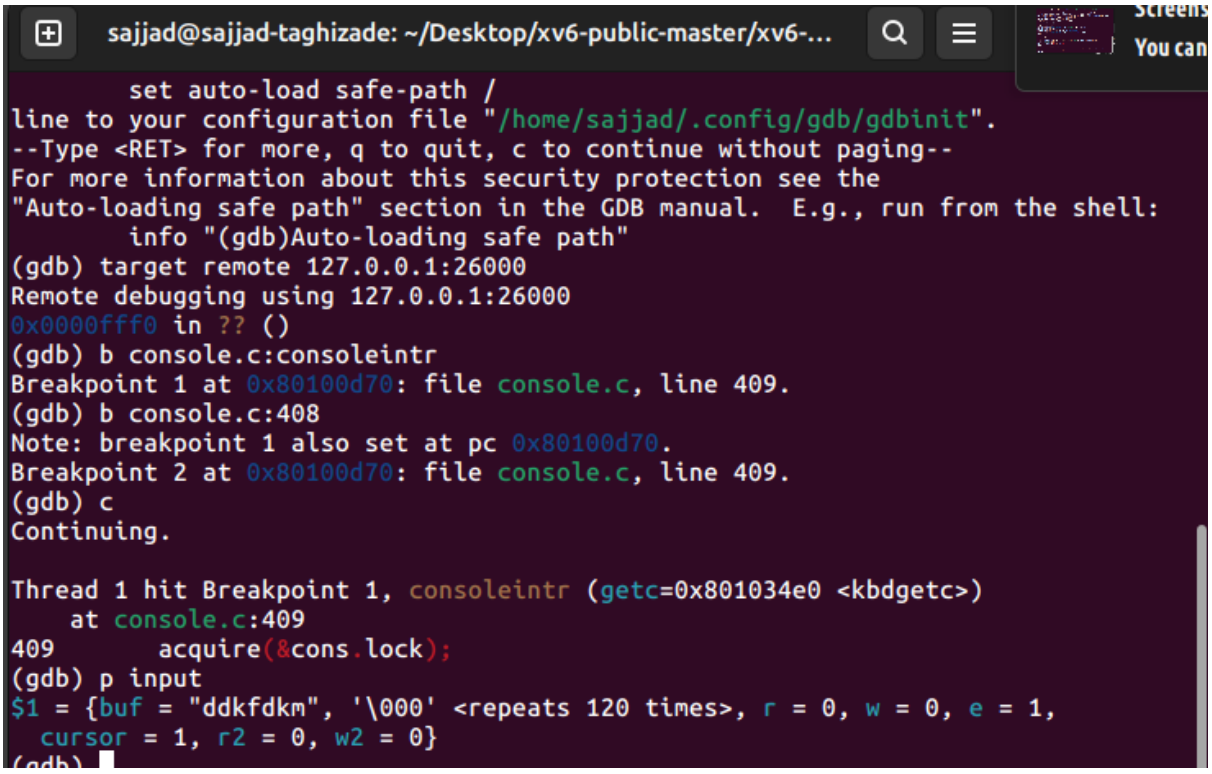
- [ ] دستور اصلی برای نمایش تمام متغیرهای محلی در تابع فعلی، info locals است. این دستور نام، نوع و مقدار فعلی تمام متغیرهایی که در آن تابع تعریف شده‌اند را لیست می‌کند.

کاربرد رجیسترهای edi و esi:

در معماری x86 ، esi و edi دو رجیستر عمومی هستند که علاوه بر کاربردهای محاسباتی عادی، به طور قراردادی برای عملیات‌های خاصی استفاده می‌شوند:

- [ ] esi : این رجیستر به عنوان "اشاره‌گر مبدا" در عملیات‌های مربوط به رشته‌ها و کار با آرایه‌ها استفاده می‌شود. دستوراتی مانند movs از esi به عنوان آدرس مبدا برای خواندن داده استفاده می‌کنند.
- [ ] edi: این رجیستر به عنوان "اشاره‌گر مقصد" در همان عملیات‌های رشته‌ای عمل می‌کند. دستوراتی مانند movs و stos از edi به عنوان آدرس مقصد برای نوشتن داده استفاده می‌کنند.

- Q25:به کمک استفاده از GDB درباره ساختار struct input موارد زیر را توضیح دهید.



توضیح کلی :

ساختار input که در فایل console.c تعریف شده، بافر اصلی ورودی خط فرمان در کرنل xv6 است. این ساختار یک بافر چرخشی را پیاده‌سازی می‌کند که ارتباط بین گرداننده وقفه کیبورد (که کاراکترها را می‌نویسد) و پروسه‌هایی مانند شل که منتظر خواندن ورودی هستند را مدیریت می‌کند.

- ☐ buf[INPUT_BUF] : آرایه‌ای که خود کاراکترهای تایپ شده را نگه می‌دارد.
- ☐ r : این اندیس به اولین کاراکتری اشاره دارد که هنوز توسط هیچ پروسه‌ای خوانده نشده است. تابع consoleread از این اندیس برای خواندن داده استفاده می‌کند.
- ☐ w : این اندیس نشان‌دهنده انتهای بخشی از بافر است که "نهایی" شده و برای خواندن آماده است. در console.c استاندارد، این اندیس فقط زمانی جلو می‌رود که کاربر کلید Enter را فشار دهد. تا زمانی که r == w باشد، پروسه‌ها در consoleread به خواب می‌روند.
- ☐ e : این اندیس به اولین خانه خالی در بافر اشاره دارد و نشان‌دهنده انتهای خطی است که در حال ویرایش است. consoleintr کاراکترهای جدید را در این اندیس می‌نویسد و آن را جلو می‌برد.

- [ ] cursor : این متغیر که شما اضافه کرده ایم، موقعیت منطقی مکان‌نما را در خط در حال ویرایش (بین w و e) دنبال می‌کند و برای قابلیت‌هایی مثل حرکت مکان‌نما با کلیدهای جهت‌نما استفاده می‌شود.
- [ ] r2 و w2: صرفا متغیری کمکی برای زمان اتصال شل به کنسول و خراب نشدن r , w های واقعی است.

نحوه و زمان تغییر مقدار input.e:

متغیر input.e تقریباً با هر بار فشردن کلید توسط کاربر تغییر می‌کند.

زمان تغییر: این متغیر در داخل تابع consoleintr و به طور خاص در توابع کمکی مانند write_character یا در بلوک default از switch تغییر می‌کند.

چگونه تغییر می‌کند:

- [ ] هنگام درج یک کاراکتر: وقتی کاربر یک کلید معمولی را فشار می‌دهد، تابع write_character فراخوانی می‌شود. این تابع:
  1. کاراکتر جدید را در موقعیت input.buf[input.cursor] قرار می‌دهد.
  2. مقدار input.e را یک واحد افزایش می‌دهد (input.e++) تا نشان دهد که طول خط در حال ویرایش یک واحد بیشتر شده است.
- [ ] هنگام حذف یک کاراکتر : وقتی کاربر Backspace می‌زند، تابع backspace فراخوانی می‌شود. این تابع:
  1. کاراکترها را در بافر جابجا می‌کند تا کاراکتر مورد نظر حذف شود.
  2. مقدار input.e را یک واحد کاهش می‌دهد (input.e--) تا نشان دهد طول خط یک واحد کمتر شده است.

مقدار: مقدار input.e همیشه بین input.w و input.w + INPUT_BUF در نوسان است و به صورت چرخشی با عملگر باقیمانده % مدیریت می‌شود. این متغیر به همراه input.w و input.r، وضعیت بافر چرخشی را به طور کامل مشخص می‌کند

- Q26:خروجی دستورات src layout و asm layout در TUI چیست؟
  - src layout

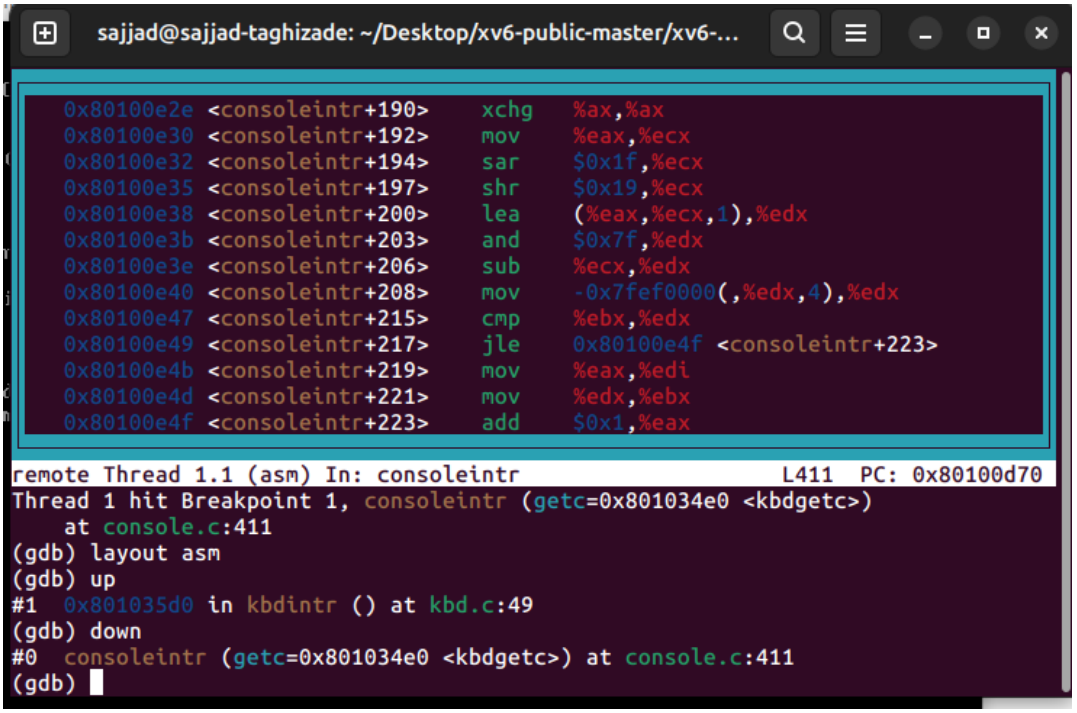
این دستور به سورس کد c ای که در آن بریک پوینت گذاشته ایم میرود.

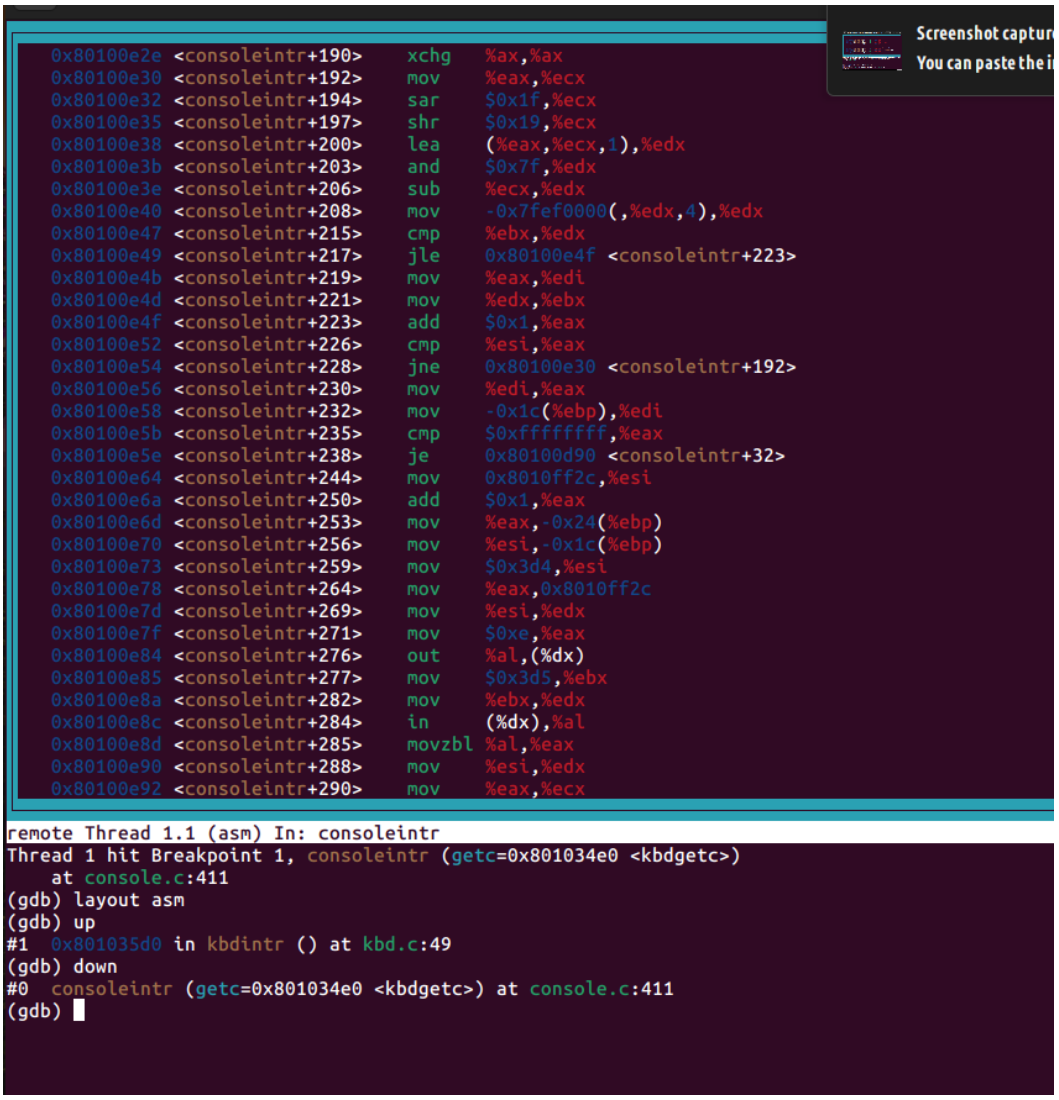
  - asm layout

این دستور به سورس کد اسمبلی ای که در آن بریک پوینت گذاشته ایم میرود.

- Q27:برای جابه‌جایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستوراتی استفاده میشود؟
  - با دستور up , down جابه جا میشود

عکس ها در پایین ضمیمه شده است.




0x80100e2e <consoleintr+190> xchg %ax,%ax
0x80100e30 <consoleintr+192> mov %eax,%ecx
0x80100e32 <consoleintr+194> sar $0x1f,%ecx
0x80100e35 <consoleintr+197> shr $0x19,%ecx
0x80100e38 <consoleintr+200> lea (%eax,%ecx,1),%edx
0x80100e3b <consoleintr+203> and $0x7f,%edx
0x80100e3e <consoleintr+206> sub %ecx,%edx
0x80100e40 <consoleintr+208> mov -0x7fef0000(,%edx,4),%edx
0x80100e47 <consoleintr+215> cmp %ebx,%edx
0x80100e49 <consoleintr+217> jle 0x80100e4f <consoleintr+223>
0x80100e4b <consoleintr+219> mov %eax,%edi
0x80100e4d <consoleintr+221> mov %edx,%ebx
0x80100e4f <consoleintr+223> add $0x1,%eax
0x80100e52 <consoleintr+226> cmp %esi,%eax
0x80100e54 <consoleintr+228> jne 0x80100e30 <consoleintr+192>
0x80100e56 <consoleintr+230> mov %edi,%eax
0x80100e58 <consoleintr+232> mov -0x1c(%ebp),%edi
0x80100e5b <consoleintr+235> cmp $0xffffffff,%eax
0x80100e5e <consoleintr+238> je 0x80100d90 <consoleintr+32>
0x80100e64 <consoleintr+244> mov 0x8010ff2c,%esi
0x80100e6a <consoleintr+250> add $0x1,%eax
0x80100e6d <consoleintr+253> mov %eax,-0x24(%ebp)
0x80100e70 <consoleintr+256> mov %esi,-0x1c(%ebp)
0x80100e73 <consoleintr+259> mov $0x3d4,%esi
0x80100e78 <consoleintr+264> mov %eax,0x8010ff2c
0x80100e7d <consoleintr+269> mov %esi,%edx
0x80100e7f <consoleintr+271> mov $0xe,%eax
0x80100e84 <consoleintr+276> out %al,(%dx)
0x80100e85 <consoleintr+277> mov $0x3d5,%ebx
0x80100e8a <consoleintr+282> mov %ebx,%edx
0x80100e8c <consoleintr+284> in (%dx),%al
0x80100e8d <consoleintr+285> movzbl %al,%eax
0x80100e90 <consoleintr+288> mov %esi,%edx
0x80100e92 <consoleintr+290> mov %eax,%ecx
Screenshot captur
You can paste the i
remote Thread 1.1 (asm) In: consoleintr
Thread 1 hit Breakpoint 1, consoleintr (getc=0x801034e0 <kbdgetc>)
at console.c:411
(gdb) layout asm
(gdb) up
#1 0x801035d0 in kbdintr () at kbd.c:49
(gdb) down
#0 consoleintr (getc=0x801034e0 <kbdgetc>) at console.c:411
(gdb)