

گزارش آزمایش اول درس آزمایشگاه سیستم عامل

(آشنایی با هسته سیستم عامل)

علی اخگری 810198341، پرنیان فاضل 810198516، پریا خوشتاب 810198387

اضافه کردن یک متن به Message Boot

در این قسمت برای اضافه کردن اسمی گروه به Message Boot، باید فایل init.c را تغییر دهیم. فایل init.c در سرایند user-level قرار گرفته و از فایل‌های اولیه‌ای است که در اجرای سیستم‌عامل xv6 اجرا می‌شود. با صدا زدن تابع main اسمی گروه را با استفاده از تابع printf که در خود سیستم‌عامل xv6 تعریف شده است، روی ترمینال چاپ می‌کنیم.

```
Booting from Hard Disk...
xv6...
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
Group #17:
1. Ali Akhgarri
2. Paria Khoshtab
3. Parnian Fazel
$
```

اضافه کردن چند قابلیت به کنسول xv6

- در این بخش کافی است که فقط دو حرف آخر از بافر ورودی (input.buf) را جابجا کنیم.
- در این بخش روی input.buf پیمایش می‌کنیم تا زمانی که یا به کاراکتر space یا به انتهای خط برسیم و در هر گام، حروفی که از قبل بزرگ هستند یا کاراکتر هایی که جزو حروف الفبا نیستند را تغییری نمی‌دهیم و باقی حروف را بزرگ می‌کنیم. همچنین در هر گام input.e را نیز یکی زیاد می‌کنیم تا بعداً بدانیم که تا کجا حروف روی input.buf را تغییر داده ایم.

● در این بخش ابتدا تمام ورودی را روی buf_copy (که یک ارایه خالی از جنس کاراکتر است) ذخیره می

کنیم، سپس input.buf را کاملا پاک می کنیم.

● در قسمت default، ابتدا buf_copy را روی input.buf می ریزیم و سپس input.buf را چاپ می کنیم.

آشنایی با سیستم عامل xv6

1. معناری سیستم عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارد؟

معناری سیستم عامل xv6 از نوع "x86" است.

در میان فایل های سیستم عامل xv6 فایل هایی نظیر x86.h وجود دارد و در Makefile هم به این معناری اشاره شده است. در واقع تعاریف لازم برای این معناری در این سیستم عامل نوشته شده است. همچنین ساختار این سیستم عامل حول یک هسته‌ی یکپارچه (Monolithic) طراحی شده است.

2. 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؟

یک پردازه در xv6 از حافظه user-space شامل دستورات، داده ها و پشتنه و همچنین بخش pre-process تشکیل شده است که مخصوص هسته می باشد. این سیستم عامل با استفاده از روش time-sharing به صورت transparent، پردازنده ها را مدیریت می کند. می دانیم روش time-sharing برای multiprogramming استفاده می شود. در این روش اگر برنامه ای در حال اجرا مدت زمان زیادی CPU را اشغال نگه داشت در صورتی که سهمیه زمانی اش تمام شد، CPU از این برنامه گرفته شده و نتایج را در رجیسترها مشخصی ذخیره می کند و در پردازه های بعدی آنها را بازیابی می کند، سپس CPU به برنامه ای بعدی اختصاص داده می شود. به طوری که همه برنامه ها با هم به طور همزون رو به پیشرفت باشند. هسته سیستم عامل به هر پردازه یک شناسه مشخص (pid) اختصاص می دهد.

مقدمه‌ای درباره سیستم‌عامل و xv6

4. فایلهای اصلی سیستم‌عامل xv6 در صفحه یک کتاب xv6 لیست شده‌اند. به طور مختصر هر گروه را توضیح دهید. نام پوشش اصلی فایلهای هسته سیستم‌عامل، فایلهای سرایند و فایل سیستم در سیستم‌عامل لینوکس چیست؟ در مورد محتويات آن مختصراً توضیح دهید.

: basic headers -

این بخش شامل `mmu.h`, `segdesc`, `define`, `typedef` و `struct` مثل که مربوط به هندل کردن حافظه‌ی مجازی می‌باشد.

: entering xv6 -

ابتدا در فایل `entry.s` هسته شروع به اجرا می‌کند و پس از آماده سازی شرایط جهت اجرای کد `C`, تابع `main` صدا زده می‌شود که عملیات آماده‌سازی اجرای هسته، اجرای تابع اولیه‌ی آن و اختصاص دادن حافظه را بر عهده دارد.

: locks -

وظیفه این بخش در حالت تعليق(spin) قرار دادن سیستم‌عامل و بخش‌هایی نظیر کامپایلر C و پردازنده است تا زمانی که اجزاهه اجرا شدن به آنها داده شود و سیستم از حالت قفل خارج شود. این کار توسط تابع مهمی مثل `release` و `acquire` می‌باشد که وظیفه‌ی آنها به ترتیب رها کردن قفل و نگه داشتن آن می‌باشد.

: processes -

در این بخش کارهای مربوط به حافظه مانند نگاشت حافظه‌ی منطقی به مجازی با استفاده از `identity map` برای استفاده کاربران (در فایل `vm.c`) به این موضوع اشاره شده است) و `memory allocation` انجام می‌شود.

: system calls -

سیستم کال‌ها با یک عدد مشخص می‌شوند که در این بخش عدد و تابع فراخوانی آن‌ها تعریف شده است. در فایل `traps.h` ثابت‌های معماری x86 برای `trap`‌ها و `interrupt`‌ها تعریف شده است.

:file system -

این گروه فایلهای مختلفی دارد که کارهای عمدۀ آن‌ها شامل تعریف ساختارهای stat sleeplock و buf، ...، عملیات مربوط به بازکردن، خواندن، نوشتند، بستن فایل‌ها، log کردن ساده سیستم که اجازه‌ی فراخوانی‌های سیستمی همزمان را می‌دهد، پیاده‌سازی سطح پایین سیستم فایل (شامل ۵ لایه blocks، files، directories، name)، بررسی سطح بالای فراخوانی‌های سیستمی سطح بالا و اجرا کردن و لود کردن برنامه‌ها در مموری می‌باشد. این بخش به کمک بخش processes کارهای مورد نیازه‌ر command را انجام میدهد و ما مدام به این دو بخش برمی‌گردیم.

:pipes -

این بخش شامل تعریف ساختار pipe و توابع مربوط به آن مثل pipealloc، piperead، pipewrite و pipeclose می‌باشد.

:string operations -

این بخش شامل توابعی برای کار با استرینگ‌ها مثل memcmp، strlen، memset و ... می‌باشد.

:low-level hardware -

این بخش شامل تعریف ساختارها و توابع مربوط به multiprocessing، کار با I/O و دریافت ورودی از کبیورد یا پورت سریالی و نشان دادن خروجی روی صفحه نمایش یا پورت سریالی، مدیریت وقفه‌های سخت افزاری برای یک سیستم SMP، مدیریت وقفه‌های داخلی غیر I/O و تعریف پورت سریالی Intel 8250 به نام UART می‌باشد.

:user-level -

در این بخش، ارتباط کاربر با xv6، اجرای اولین برنامه سطح کاربر، کارهای مربوط به ترمینال و تجزیه کردن آرگمنهای ورودی برای هر command در ترمینال انجام می‌شود. همچنین فراخوانی‌های سیستمی در سطح کاربر انجام می‌شود.

:bootloader -

این بخش به طور کلی عمل‌های لازم برای بوت شدن و بالا آمدن سیستم عامل xv6 را انجام می‌دهد. این بخش

شامل 2 فایل S bootmain.c و bootasm.S است که شامل کدهای اس梅بی و c است و وظیفه‌ی لود کردن

و قرار دادن BIOS به حالت حفاظت‌شده 32 بیت را بر عهده دارد.

:link -

این بخش شامل کد اتصال‌دهنده‌ی ساده به هسته‌ی JOS است.

در سیستم عامل لینوکس، فایل‌های هسته سیستم عامل در فولدری به نام /boot شامل فایل‌های پیکربندی برای بوت کردن

لینوکس، قرار دارد. سرایند ها در فولدر include شامل فایل‌های سرایند برای کامپایلرهای C نظیر stdio.h، قرار دارند و فایل سیستم‌ها نیز عمدها در فولدر fs شامل فایل‌های پشتیبانی شده توسط نسخه فعلی لینوکس قرار دارند.

اجرای بوتلودر

۸. در xv6 در سکتور نخست دیسک قابل بوت، محتواهی چه فایلی قرار دارد. (راهنمایی: خروجی

دستور make -n را بررسی نمایید.)

```
(base) ali_akhgari@Amins-MacBook-Pro:~/xv6-public % make -n
x86_64-elf-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 -fno-pic -O -nostdinc -I . -c bootmain.c
x86_64-elf-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 -fno-pic -nostdinc -I . -c bootasm.S
x86_64-elf-ld -m elf_i386 -N -e start -Ttext 0x7000 -o bootblock.o bootasm.o bootmain.o
x86_64-elf-objdump -S bootblock.o > bootblock.asm
x86_64-elf-objcopy -S -O binary -j .text bootblock.o bootblock
x86_64-elf-pl -l bootblock
x86_64-elf-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 bio.o bio.c
x86_64-elf-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
x86_64-elf-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 exec.o exec.c
x86_64-elf-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 file.o file.c
x86_64-elf-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 fs.o fs.c
x86_64-elf-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 i386.o i386.c
x86_64-elf-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 iopic.o iopic.c
x86_64-elf-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 kalloc.o kalloc.c
x86_64-elf-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 kbd.o kbd.c
x86_64-elf-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 lapi.o lapi.c
x86_64-elf-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 log.o log.c
x86_64-elf-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 main.o main.c
x86_64-elf-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 mp.o mp.c
x86_64-elf-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 picirq.o picirq.c
x86_64-elf-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 pipe.o pipe.c
x86_64-elf-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 proc.o proc.c
x86_64-elf-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 sleeplock.o sleeplock.c
x86_64-elf-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 spinlock.o spinlock.c
x86_64-elf-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 string.o string.c
x86_64-elf-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 syscall.o syscall.c
x86_64-elf-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 sysfile.o sysfile.c
x86_64-elf-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 sysproc.o sysproc.c
x86_64-elf-ld -m elf_i386 -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o
x86_64-elf-objcopy -S -O binary -j .text bootblockother.o entryother
x86_64-elf-objdump -S bootblockother.o > entryother.asm
x86_64-elf-ld -m elf_i386 -N -e start -Ttext 0 -O initcode.out initcode.o
x86_64-elf-objcopy -S initcode.o > initcode
x86_64-elf-objdump -S initcode.o > initcode.asm
x86_64-elf-ld -m elf_i386 -T kernel.ld -o kernel.entry.o bio.o console.o exec.o file.o fs.o ide.o iopic.o kalloc.o kbd.o lapi.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.o string.o
x86_64-elf-objcopy -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
(base) ali_akhgari@Amins-MacBook-Pro:~/xv6-public %
```

همانطور که در خط سوم تصویر بالا مشخص می‌باشد، ابتدا فایل‌های bootasm.o و bootmain.o و bootblock.o را

لینک می‌کنیم و سپس با استفاده از Ttext آغاز برنامه را با استفاده از e- در قسمت مشخص شده در خروجی می‌ریزیم.

در نتیجه محتوای فایل bootasm.s در سکتور نخست قابل بوت قرار می گیرد. در این حالت مقدار رجیستر CS(ثباتی که محتوای قطعه کدی که برنامه در آن اجرا می شود را نگه می دارد) برابر 0 و محتوای رجیستر IP(ثباتی که محتوای اشاره گر به دستور بعدی که فقط قابل خواندن می باشد را نگه می دارد) برابر 7C00 خواهد بود.

11. بوت سیستم توسط فایل های bootmain.c و bootasm.S صورت میگیرد. چرا تنها از کد C استفاده نشده است؟

در ابتدای بوت سیستم، فایل bootasm.S که یک کد به زبان اسembly است، اجرا می شود و پردازنده را در حالت 32 بیت protected-mode قرار داده که این حالت باعث می شود رجیسترها، آدرس های مجازی و اکثر محاسبات عددی به جای 16 بیت با 32 بیت هندل شوند. برای این که این تغییر ممکن شود نیاز به زبان اسembly داریم زیرا processor تنها زبان اسembly را می فهمد. (بیشتر کدهای XV6 به زبان C است و کدهای به زبان اسembly (شامل استفاده از پشته برای صدا کردن توابع) به صورت خودکار توسط کامپایلر تولید می شوند. بخش کوچکی از کدهای XV6 به زبان اسembly است اما چون در بعضی مواقع سیستم عامل نیاز دارد که کنترل و تسلط بیشتری روی انجام عملیات داشته باشد، این بخش کدها به زبان سطح پایین تر اسembly نوشته شده اند تا با سخت افزار ارتباط بهتری داشته باشد.) برای تعویض از حالت real به حالت bootstrap GDT از protected استفاده میکنیم که آدرس های مجازی را مستقیماً به آدرس های فیزیکی نگاشت یک به یک می کند. سپس بوت سکتور تابع bootmain که به زبان C است را call می کند که وظیفه آن لود و اجرا کردن هسته سیستم عامل می باشد.

12. یک ثبات عام منظوره، یک ثبات قطعه، یک ثبات وضعیت و یک ثبات کنترلی در معماری x86 را نام بده و وظیفه هر یک را به طور مختصر توضیح دهید.

ثبات عام منظوره (general-purpose register): اکثر دستورات روی این رجیسترها اجرا می شوند که به رجیسترها 16 و 8 بیت هم شکسته می شوند. مانند: EAX یک نمونه از ثبات عام منظوره 32 بیت، AX یک نمونه از ثبات عام منظوره 16 بیت و AH و AL دو نمونه از ثبات عام منظوره 8 بیت می باشند. توجه شود که پسوندهای H و L در ثبات های عام منظوره 8 بیت به ترتیب نشان دهنده Low-byte و High-byte هستند.

هر 4 رجیستر EAX، AX، AH و AL رجیسترها Accumulator نامیده می شوند و برای دسترسی به پورت I/O،

محاسبات و ایجاد وقفه‌ها استفاده می‌شوند.

ثبات قطعه (segment register): ثبات‌های قطعه‌ای بخشهایی از آدرس‌های آیتم‌های مختلف را نگهداری می‌کنند. این ثبات‌ها تنها از طریق ثبات‌های عام‌منظوره یا دستورات خاص، مقداردهی می‌شوند. برخی از این رجیسترها برای اجرای خوب و درست برنامه‌ها بسیار مهم هستند و در multi-segment programming از آن‌ها استفاده می‌شود. برای مثال رجیسترهای SS، CS، DS، ES و 20 بیتی از رجیسترهای 16 بیتی را تولید می‌کنند.

ثبات وضعیت (status register): رجیسترهای EFLAGS وضعیت کنونی پردازنده را نگهداری می‌کنند که برای مقایسه‌ی برخی پارامترها، حلقه‌های شرطی و جهش‌های شرطی استفاده می‌شوند. هر بیت وضعیت یک پارامتر خاص از آخرین دستور را نگهداری می‌کند. مثلاً بیت شماره 2، PF یعنی parity flag است که زوجیت را مشخص می‌کند.

ثبات کنترلی (control registers): رجیسترهای کنترلی همراه رجیسترهای قطعه‌ای در protected mode programming استفاده می‌شوند و اطلاعات کنترلی و فراداده‌های metadata را نگهداری می‌کنند. رجیسترهای کنترلی از CR0 تا CR4 هستند. برای مثال CR3 به page table اشاره می‌کند.

14. مدهای مختلف در پردازنده‌های 86x را مختصرًا توضیح دهید.

مد حقيقی (real mode): در هنگام بوت، پردازنده در مد حقيقی قرار دارد. این مد به محیط برنامه‌ی پردازنده اجازه می‌دهد تا با سایر مدها تعویض شود. در این مد سیستم عامل MS-DOS اجرا می‌شود که به دسترسی مستقیم به حافظه و سخت‌افزار سیستم نیاز دارد. اگر یک برنامه در این مد اجرا شود این امکان وجود دارد که سیستم عامل crash کند یا جواب دادن به دستورات متوقف شود.

مد حفاظت‌شده (protected mode): این مد، مد ذاتی پردازنده است که در آن تمام دستورات دستگاه در دسترس است. در این مد به هر برنامه، یک بخش حافظه اختصاص داده می‌شود که segments نامیده می‌شوند و پردازنده به برنامه‌ها اجازه نمی‌دهد که به حافظه خارج از بخش segment خود مراجعه کنند.

مد مدیریت سیستم (system-management mode): این مد روشی را برای پیاده سازی توابع مختلف مانند مدیریت قدرت یا امنیت سیستم در اختیار سیستم عامل قرار می‌دهد. که این‌ها معمولاً توسط سازندگانی اجرا می‌شوند که پردازنده را برای

تنظیمات خاصی تغییر می‌دهند. فقط توسط firmware یعنی BIOS یا UEFI استفاده می‌شود نه توسط سیستم‌عامل یا برنامه‌های نرم‌افزاری.

مد مجازی(virtual-8086 mode): در این مد که تحت مد حفاظت‌شده است، CPU قادر است که برنامه‌های real-address مثل برنامه‌های MS-DOS در محیطی امن اجرا کند یعنی اگر برنامه crash کند یا سعی کند داده‌ها را در حافظه‌ی سیستم بنویسد، بر سایر برنامه‌های درحال اجرا تاثیر نمی‌گذارد. این مد به اندازه‌ای منعطف هستند که برنامه‌های در مد حقیقی و برنامه‌های در مد مجازی به صورت همزمان اجرا شوند.

مد طولانی(long mode): در معماری کامپیوتر ۸۰۸۶x این مد مدعی است که در آن سیستم عامل ۶۴ بیتی میتواند به دستورات ۶۴ بیتی و رجیسترها دسترسی پیدا کند. برنامه‌های ۶۴ بیتی در یک مد فرعی(sub-mode) به نام bit-64 اجرا می‌شوند در حالی که برنامه‌های ۳۲ بیتی و برنامه‌های مد حفاظت‌شده ۱۶ بیتی در یک مد فرعی(sub-mode) به نام compatibility اجرا می‌شوند. برنامه‌های مد حقیقی و مد مجازی ۸۰۸۶ نمی‌توانند در مد طولانی اجرا شوند.

18. چرا از همان ابتدا هسته در حافظه توسط BIOS بارگذاری نمی‌شود؟ علت اصلی را بیان نمایید.

در صورتی که هسته در ابتدا توسط BIOS بارگذاری شود، باید بتواند فایل‌های سیستم، بوت‌لودر و نحوه ذخیره‌ی اجزای سیستم عامل بر روی دیسک را درک و تفسیر کند. BIOS یک قطعه‌ی سخت‌افزاری است اما هسته از آن سطح بالاتر است. بنابراین قبل از بالا آمدن سیستم، بایوس ابتدا CPU، سوکت حافظه را بررسی می‌کند و رجیسترها را مقداردهی می‌کند و سپس کرنل را برای اجرا بارگزاری می‌کند. متن و داده‌های هسته در نیمه‌ی بالایی حافظه مجازی قرار دارد و از آنجایی که آدرس مجازی را به آدرس فیزیکی مپ می‌کند و قبل از لود شدن کرنل و در مرحله‌ی بوت (entry.s) در حافظه فیزیکی لود می‌شود، بنابراین نمیتوان هسته از ابتدا توسط بایوس بارگذاری شود.

اجرای هسته xv6

19. چرا این آدرس فیزیکی است؟

این مقدار باید یک آدرس فیزیکی باشد زیرا قسمت hardware paging نمی‌داند چگونه آدرس مجازی را ترجمه کند و هنوز ای ندارد که بخواهد توسط آن آدرس مجازی را ترجمه کند.

22. علاوه بر صفحه بندی در حد ابتدایی از قطعه بندی به منظور حفاظت هسته استفاده خواهد شد. این عملیات توسط `seginit()` انجام می‌گردد. همانطور که ذکر شد، ترجمه قطعه تأثیری بر ترجمه آدرس منطقی نمی‌گذارد. زیرا تمامی قطعه‌ها اعم از کد و داده روی یکدیگر می‌افتد. با این حال برای کد و داده‌های سطح کاربر پرچم `SEG_USER` تنظیم شده است. چرا؟ (راهنمایی: علت مربوط به ماهیت دستورالعمل‌ها و نه آدرس است.)

تابع `seginit` در شروع هر پردازنده، یکبار اجرا می‌شود. همچنین این تابع با توجه به مقادیر `SEG_UCODE` و `mmu.h` که در مقادیر ۳ و ۴ دیفاین شده‌اند، هیچ code descriptor را بین هسته و کاربر به اشتراک نمی‌گذارد. با توجه به اولویت بیشتر برنامه‌های هسته، پردازنده جلوی وقفه را در حالتی که دسترسی از `CLP=0` را می‌گیرد بنابراین در سیستم عامل xv6 هنگامی که پردازنده در حالت انجام برنامه سطح کاربر است، آدرس‌های بخش هسته غیرقابل دسترسی هستند. به همین دلیل به یک پرچم نیاز داریم که سطوح دسترسی بین هسته و کاربر را مشخص کند.

23. مدیریت برنامه‌های سطح کاربر مستلزم ارائه انتزاعاتی برای ایجاد تمایز میان این برنامه‌ها و برنامه مدیریت آنها است. جهت نگهداری اطلاعات مدیریتی برنامه‌های سطح کاربر ساختاری تحت عنوان `struct proc` (خط 2336) ارائه شده است. اجزای آن را توضیح داده و ساختار معادل آن در سیستم عامل لینوکس را بیابید.

این `struct` جهت نگهداری ویژگی‌های مختلف یک process است:

Sz : سایز حافظه‌ی process به بایت •

- page table :pgdir •
- Kstack : پایین پشته‌ی هسته برای این پردازه •
- process state :state •
- (process pid : شناسه این پردازه •
- parent : والد این پردازه •
- trapframe :tf برای فراخوانی سیستمی کنونی •
- Context swtch() : با پردازه را اجرا می‌کنیم. •
- chan : در صورتی که chan مقداری غیر صفر داشته باشد، sleepinh on chan اتفاق می‌افتد. •
- killed : در صورتی که killed مقداری غیر صفر داشته باشد، kill شده است. •
- ofile : آرایه‌ای از فایل‌های باز است. •
- directory cwd : کنونی •
- name : نام پردازه •

ساختار معادل این struct در سیستم عامل لینوکس، task_struct در فایل sched.h در آدرس linux/include/linux/sched.h است.

۲۷. فراخوانی سیستمی exec() در لینوکس چه وظیفه ای دارد؟

فراخوانی سیستمی exec برای اجرای یک فایل که در یک پردازه فعال قرار دارد استفاده می‌شود. وقتی exec فراخوانی می‌شود، فایل اجرایی قبلی جایگزین می‌شود و فایل جدید اجرا می‌شود.

به طور دقیق تر می‌توانیم بگوییم که فراخوانی سیستمی exec، فایل یا برنامه قدیمی از پردازه را با یک فایل یا برنامه جدید جایگزین می‌کند و در واقع کل محتوای پردازه با یک برنامه جدید جایگزین می‌شود.

بخش داده کاربر که فراخوانی سیستمی exec را اجرا می‌کند، با فایل داده ای که نام آن در آرگومان exec (هنگام فراخوانی) آمده جایگزین می‌شود.

اشکال زدایی

GDB روند اجرای

. 1. برای مشاهده Breakpoint ها از چه دستوری استفاده می شود؟

برای مشاهده breakpoint ها از دستور info breakpoint(s) یا info b استفاده می شود.

```
(gdb) b console.c:199
Breakpoint 1 at 0x80100880: file console.c, line 201.
(gdb) b console.c:215
Breakpoint 2 at 0x801008f8: file console.c, line 215.
(gdb) info breakpoint
Num      Type            Disp Enb Address      What
1        breakpoint      keep y  0x80100880 in consoleintr at console.c:201
2        breakpoint      keep y  0x801008f8 in consputc at console.c:215
(gdb)
```

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

برای حذف یک breakpoint از دستور del استفاده می شود، به صورتی که این دستور را همراه با شماره آن

تایپ می کنیم. (شماره breakpoint ها را می توان از طریق دستور info به دست آورد.)

به طور مثال اگر بخواهیم breakpoint شماره 2 را حذف کنیم، تایپ می کنیم: del 2

```
(gdb) info b
Num      Type            Disp Enb Address      What
1        breakpoint      keep y  0x80100880 in consoleintr at console.c:201
2        breakpoint      keep y  0x801008f8 in consputc at console.c:215
(gdb) del 2
(gdb) info b
Num      Type            Disp Enb Address      What
1        breakpoint      keep y  0x80100880 in consoleintr at console.c:201
(gdb) []
```

کنترل روند اجرا و دسترسی به حالت سیستم

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

دستور bt یا backtrace command ، خلاصه ای از اینکه چگونه برنامه ما به نقطه توقف رسیده است را نمایش می دهد.

به عبارت دیگر خروجی این دستور به این صورت است که در هر خط، یک فریم نمایش داده می شود که به ترتیب از فریم اول که نقطه توقف است شروع می شود و در هر خط فریم های دیگر stack فراخوانی می شوند. یعنی هر فریم نشان دهنده جایی در آن برنامه است که قبلی را فراخوانی کرده است. (این روند نوشتمن را می شود با ctrl+c متوقف کرد).

```
(gdb) bt
#0  consoleintr (getc=0x801154ec <stack+3964>) at console.c:204
#1  0x801154ec00000000 in ?? ()
#2  0x00000000801154ec in stack ()
#3  0x000000108dff000 in ?? ()
#4  0x0000000000000000 in ?? ()
(gdb)
```

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

دستور print مقدار ذخیره شده در یک متغیر را نشان می دهد. به این صورت که p variable_name مقدار ذخیره شده در variable_name را نشان می دهد.

```
(gdb) print input
$2 = {buf = "efg", '\000' <repeats 124 times>, r = 0, w = 0, e = 3, size = 6}
(gdb)
```

دستور x محتوای یک آدرس از مموری را نشان می دهد. به این صورت که x variable_address مقدار ذخیره شده در آدرس variable_address را نشان می دهد.

همچنین با دستور print می توان مشخص کرد که آرایه ها، ساختار ها و ... چگونه نمایش داده شوند. به طور مثال اگر دستور off set print address در هنگام نمایش stack و breakpoint و غیره، آدرس را نمایش نمی دهد.

با استفاده از دستور x می توان تعدادی از خانه های حافظه را با فرمت خاص چاپ کرد. این دستور به فرمت x/nfu addr است که n تعداد خانه های ممکن است که می خواهیم نمایش داده شود را مقدار دهی می کند که به طور پیش فرض ۱ است. این عدد اگر مثبت باشد، خانه های حافظه ای که بعد از addr هستند را نمایش می دهد و اگر منفی باشد، ادرس های قبل از addr مورد بررسی قرار داده می شوند.

f برای مشخص کردن فرمت نمایش است.

u برای مشخص کردن unit size است.

با استفاده از دستور info registers register_name میتوان محتویات رजیستر register_name را مشاهده کرد. (به اختصار می توان از دستور i r register_name استفاده کرد.)

به طور مثال، در اینجا محتویات رجیستر pc را می بینیم:

```
(gdb) info registers pc
pc            0xffff0          0xffff0
(gdb)
```

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

برای نمایش نام و محتوای تمامی ثبات ها از دستور info registers استفاده می کنیم، به این صورت که این دستور نام و محتوای تمامی رجیستر ها را نمایش می دهد.

```
(gdb) info registers
rax      0x0          0
rbx      0x282        642
rcx      0x0          0
rdx      0x0          0
rsi      0x80111840   2148603968
rdi      0x80111844   2148603972
rbp      0x801154d8   0x801154d8 <stack+3944>
rsp      0x801154d0   0x801154d0 <stack+3936>
r8       0x0          0
r9       0x0          0
r10      0x0          0
r11      0x0          0
r12      0x0          0
r13      0x0          0
r14      0x0          0
r15      0x0          0
rip      0x80103c21   0x80103c21 <mycpu+17>
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=1023 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

برای نمایش نام و مقدار تمامی متغیر های محلی میتوان از دستور info locals استفاده کرد به طوری که این دستور تمام متغیر های محلی موجود در stack frame کنونی را نمایش می دهد.

```
(gdb) b console.c:215
Breakpoint 1 at 0x801008f8: file console.c, line 215.
(gdb) continue
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
warning: Selected architecture i386 is not compatible with reported target architecture i386:x86-64
Architecture `i386' not recognized.
The target architecture is set to "auto" (currently "i386:x86-64").
=> 0x80103c21 <mycpu+17>:     mov    -0x7feee7dc(%rip),%esi      # 0x21544b
mycpu () at proc.c:48
48          for (i = 0; i < ncpu; ++i) {
(gdb) info locals
apicid = 0
i = 0
(gdb)
```

به طور مثال در شکل بالا، ما یک breakpoint در خط 215 از فایل console.c و در تابع consoleintr قرار داده ایم که متغیر های محلی این تابع، i و apicid هستند که با دستور info locals توانستیم نام و مقدار آنها را مشاهده کنیم. رجیستر های esi و edi در معماری X86، به منظور استفاده از اندیس ها و اشاره گر ها استفاده می شوند، edi به عنوان اندیس مقصد و برای رشته ها، کپی کردن آرایه از مموری و آدرس دهی از طریق اشاره گر کاربرد دارد.

esi به عنوان رجیستر مبدأ و برای رشته ها و کپی کردن آرایه از مموری کاربرد دارد.

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

● توضیح کلی این struct و متغیرهای درونی آن و نقش آنها

استراکت input دارای چهار متغیر است که به کارهای مربوط به ورودی گرفته شده از کاربر رسیدگی میکند.

buf یک رشته از کاراکتر ها است که همان ورودی هایی است که کاربر وارد کرده است.

e یک عدد مثبت است برای تغییر دادن buf به عنوان یک اندیس عمل می کند.

w یک عدد مثبت است برای اینکه مشخص کند در کجا باید نوشته شود.

r یک عدد مثبت است برای اینکه مشخص کند از کجا بافر باید خوانده شود.

البته طبق کد اصلی، متغیر های w و r در ابتدا صفر هستند.

● نحوه و زمان تغییر مقدار متغیرهای درونی

به طور مثال e زمانی تغییر می کند که کاربر در ترمینال چیزی بنویسد.

به طور مثال اگر کاربر کلمه abc را وارد کند، input به شکل زیر تغییر می کند:

(متغیر size در مرحله اضافه کردن shortcut ها به struct input اضافه شده است.)

```
(gdb) p input
$1 = {buf = "abc", '\000' <repeats 124 times>, r = 0, w = 0, e = 3, size = 3}
```

اشکال زدایی در سطح کد اسمنلی

7. خروجی دستورهای layout asm و layout src در TUI چیست؟

به طور کلی دستورات layout name مشخص می کنند که کدام یک از پنجره ها نمایش داده شوند، name می تواند نام

یکی از پنجره هایی باشد که قبل از ساخته شده اند(built-in) یا یک layout باشد که توسط کاربر مشخص شده است.

اگر دستور layout src را وارد کنیم، می توانیم مکان توقف را در کد بینیم.

```

console.c
165     crt[pos] = ' ' | 0x0700;
166 }
167
168 void
169 consputc(int c)
170 {
> 171     if(panicked){
172         cli();
173         for(;;)
174             ;
175     }
176
177     if(c == BACKSPACE){
178         uartputc('\b'); uartputc(' '); uartputc('\b');
179     } else
180         uartputc(c);

```

remote Thread 1.1 In: consoleintr

L171 PC: 0x80100958

اگر دستور asm layout را وارد کنیم، می توانیم مکان توقف را می توانیم مکان توقف را در کد اس梅بلی بینیم.

<pre> 0x801004bc <consputc+188> mov \$0x720,%eax 0x801004c1 <consputc+193> mov %ax,(%esi) 0x801004c4 <consputc+196> lea -0xc(%ebp),%esp 0x801004c7 <consputc+199> pop %ebx 0x801004c8 <consputc+200> pop %esi 0x801004c9 <consputc+201> pop %edi 0x801004ca <consputc+202> pop %ebp 0x801004cb <consputc+203> ret 0x801004cc <consputc+204> lea 0x0(%esi,%eiz,1),%esi 0x801004d0 <consputc+208> lea -0x1(%eax),%esi 0x801004d3 <consputc+211> test %eax,%eax 0x801004d5 <consputc+213> jne 0x8010046f <consputc+111> 0x801004d7 <consputc+215> movb \$0x0,-0x19(%ebp) 0x801004db <consputc+219> mov \$0x800b8000,%esi 0x801004e0 <consputc+224> xor %edi,%edi 0x801004e2 <consputc+226> jmp 0x80100496 <consputc+150> </pre>	
---	--

remote Thread 1.1 In: consoleintr

L171 PC: 0x80100958

8. برای جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستورهایی استفاده می شود؟

از دستورات next و finish می توان استفاده کرد.

با استفاده از این دستور می شود از تابع کنونی خارج شود. این دستور به این صورت است که اجرا شدن را ادامه

می دهد تا دقیقا زمانی که به تابع بعدی در استک فریم برود.

```
(gdb) b console.c:199
Breakpoint 1 at 0x80100880: file console.c, line 199.
(gdb) b console.c:325
Breakpoint 2 at 0x80100280: file console.c, line 326.
(gdb) continue
Continuing.
warning: Selected architecture i386 is not compatible with reported target architecture i386:x86-64
Architecture `i386' not recognized.
The target architecture is set to "auto" (currently "i386:x86-64").
=> 0x80100280 <consoleread>:    push    %rbp

Thread 1 hit Breakpoint 2, consoleread (ip=0x0, dst=0x0, n=0) at console.c:326
326      iunlock(ip);
(gdb) finish
Run till exit from #0  consoleread (ip=0x0, dst=0x0, n=0) at console.c:326
=> 0x80100880 <consoleintr>:    push    %rbp

Thread 1 hit Breakpoint 1, consoleintr (getc=0x801154ec <stack+3964>) at console.c:202
202      acquire(&cons.lock);
(gdb) 
```

: با استفاده از این دستور می توان تمام تابع را اجرا کرد. به عبارت دیگر در حالت عادی gdb به آن تابع رفته و خط های آن تابع را یکی یکی اجرا می کند، اما با این دستور می توان تمام تابع را با یک کلید از کیبورد اجرا کرد.

```
+ symbol-file kernel
(gdb) b console.c:199
Breakpoint 1 at 0x80100880: file console.c, line 199.
(gdb) b console.c:325
Breakpoint 2 at 0x80100280: file console.c, line 326.
(gdb) continue
Continuing.
warning: Selected architecture i386 is not compatible with reported target architecture i386:x86-64
Architecture `i386' not recognized.
The target architecture is set to "auto" (currently "i386:x86-64").
=> 0x80100280 <consoleread>:    push    %rbp

Thread 1 hit Breakpoint 2, consoleread (ip=0x0, dst=0x0, n=0) at console.c:326
326      iunlock(ip);
(gdb) next
=> 0x80101b10 <iunlock>:    push    %rbp
iunlock (ip=0x0) at fs.c:318
318      if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
(gdb) next
=> 0x80104630 <holdingsleep>:    push    %rbp
holdingsleep (lk=0x0) at sleeplock.c:49
49      acquire(&lk->lk);
(gdb) next
=> 0x80104850 <acquire>:    push    %rbp
acquire (lk=0x80107b24) at spinlock.c:27
27      pushcli(); // disable interrupts to avoid deadlock.
(gdb) next
=> 0x80104700 <pushcli>:    push    %rbp
pushcli () at /Users/ali_akhgary/Desktop/XV6.tmp/xv6-public/x86.h:98
98      asm volatile("pushfl; popl %0" : "=r" (eflags));
(gdb) next
```

آدرس مخزن:

https://gitlab.com/ali_akhgari/os-lab

شناسه اخرين :commit

910f4e9c885c942781412cdbb0e166166f5409d3