پروژه ۱ آزمایشگاه سیستم‌عامل

گروه ۱۵ **اعضای گروه:** نسا عباسی مقدم-آوا میرمحمدمهدی-سپهر آزردار

1) معماری سیستم عامل xv6 چیست؟ چه دلایلی برای دفاع از نظر خود دارید؟

سیستم عامل xv6 با استفاده از ویرایش ششم سیستم عامل Unix (Unix v6) پیاده‌سازی شده است؛ این سیستم عامل با AMSI C و برای x86 multiprocessor و سیستم‌های RISC-V طراحی شده است؛ از دلایل این می‌توان به وجود فایل asm.h که در آن استفاده از معماری x86 ذکر شده اشاره کرد؛ همچنین در فایل mmu.h از واحد مدیریت حافظه x86 استفاده شده است و در فایل x86.h دستورات اسمبلی که مخصوص پردازنده های مبتنی بر x86 است، بکار گرفته شده است.

۲) یک پردازه در سیستم عامل xv6 از چه بخش‌هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازه‌های مختلف اختصاص می‌دهد؟

یک پردازه در سیستم عامل xv6 از بخش های زیر تشکیل شده است:

- user-space memory که شامل instructions و data و stack است.
- state هر پردازه که به طور خصوصی در اختیار kernel قرار می‌گیرد.

فرآیند time-sharing در xv6 این‌گونه انجام می‌شود که این سیستم عامل به ازای مجموعه پردازه‌هایی که در انتظار اجرا شدن هستند بین CPUهای موجود سوییچ می‌کند و زمانی که پردازه‌ای در حال اجرا شدن نیست، registerهای مربوط به آن را در CPU ذخیره می‌کند و بار دیگر که آن پردازه را اجرا می‌کند، آن ها را restore می‌کند. کرنل برای اینکار به هر پردازه یک process identifier یا pid اختصاص می‌دهد.

۳) مفهوم file descriptor در سیستم عامل‌های مبتنی بر UNIX چیست؟ عملکرد pip در سیستم عامل xv6چگونه است و به طور معمول برای چه هدفی استفاده می‌شود؟

در سیستم عامل‌های مبتنی بر UNIX، file descriptor یک شناسه‌ی یکتا برای فایل یا دیگر منابع IO مثل pipe و network socket است که پردازه از روی آن می‌خواند یا روی آن می‌نویسد. file descriptor ها معمولا دارای اعدادصحیح غیرمنفی هستند و مقادیر منفی آنها برای نشان دادن no value یا error بکار می‌رود. رابط file descriptor تفاوت بین جزییات fileها و pipeها و deviceها را نادیده می‌گیرد و کاری می‌کند که تمامی آنها streamای از بایت‌ها به نظر برسند. کرنل در xv6، file descriptor را به عنوان اندیسی در table هر پردازه استفاده می‌کند به طوری که هر پردازه یک فضای خصوصی از file

descriptorها دارد که از ۰ شروع می‌شوند. طبق قرارداد یک پردازه از file descriptor 0 (standard input) می‌خواند، خروجی را در file descriptor 1 (standard output) می‌نویسد و پیام‌های ارور را در file descriptor2 (standard error) می‌نویسد.shell از این قرارداد استفاده می‌کند تا I/O redirection و pipeline ها را پیاده‌سازی کند همچنین تضمین می‌کند که همیشه file descriptor 3 باز داشته باشد که به طور پیش‌فرض file descriptorهای کنسول هستند.

Pipe یک kernel buffer کوچک است که به صورت یک pair از file descriptor ها (یکی برای خواندن و یکی برای نوشتن) در معرض پردازه‌ها قرار می‌گیرد. نوشتن داده در یک انتهای pipe، آن داده را برای خواندن از سر دیگر pipe در دسترس قرار می‌دهد. هدف از استفاده از pipe ها، فراهم کردن راهی برای ارتباط برقرار کردن پردازه‌ها است.

۴) فراخوانی‌های سیستمی exec و fork چه عملی انجام می‌دهند؟ از نظر طراحی، ادغام نکردن این دو چه مزیتی دارد؟

Fork و file descriptor ها با تعامل با یکدیگر، پیاده‌سازی I/O redirection را راحت می‌کنند. Fork، file descriptor table والد را به همراه حافظه‌ی آن کپی می‌کند تا فرزند دقیقا با فایل‌های باز یکسان با والدش کار خود را آغاز کند. Exec، حافظه‌ی پردازه‌ی فراخوانی شده را جایگزین می‌کند ولی file table آن را حفظ می‌کند؛ این عملکرد به shell اجازه می‌دهد تا I/O redirection را توسط fork و دوباره باز کردن file descriptor های انتخاب شده و سپس exec کردن برنامه‌ی جدید، پیاده‌سازی کند؛ این همان کاری است که کد نوشته شده در xv6 shell برای I/O redirection انجام می‌دهد، در واقع در ابتدا shell، shell فرزند را fork می‌کند و run cmd، exec را فراخوانی می‌کند تا برنامه‌ی جدید را لود کند. مزیت ادغام نکردن این‌دو این است که در این صورت shell می‌تواند فرزند را fork کند ، از باز کردن، بستن و کپی کردن درون فرزند استفاده کند تا file descriptor های I/O استاندارد را تغییر دهد و سپس فراخوانی exec انجام می‌شود و همچنین هیچ تغییری در برنامه‌ای که درحال exec شدن است، نیاز نیست. اگر فراخوانی‌های سیستمی exec و fork ادغام بود، یک طرح پیچیده‌تر برای redirect کردن standard I/O توسط shell نیاز بود یا خود برنامه باید می‌فهمید که چگونه standard I/O را redirect کند.

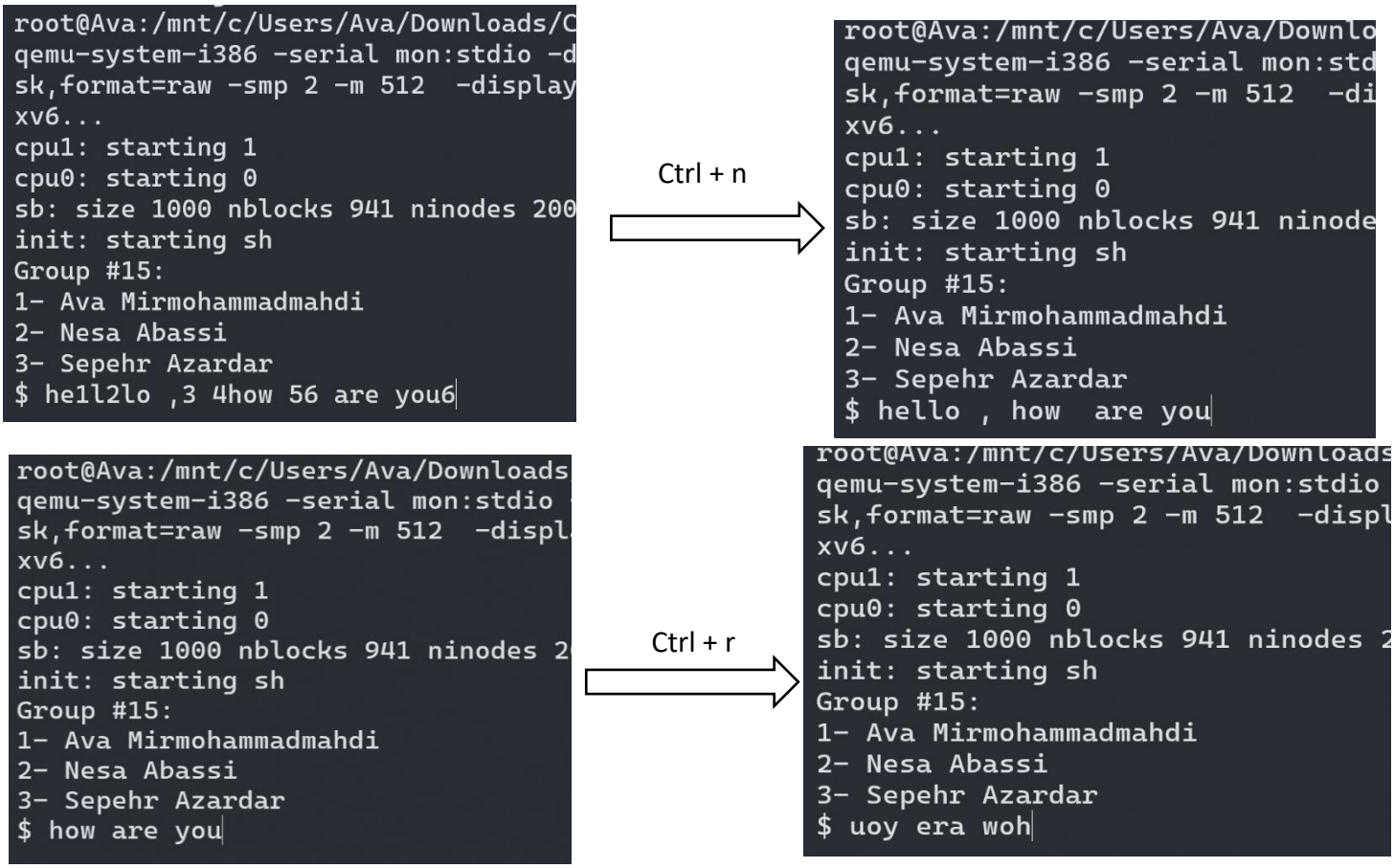
اضافه کردن یک متن به Boot Message:

همان طور که در تصویر زیر دیده می‌شود نام اعضای گروه پس از بوت شدن سیستم عامل در انتهای پیام‌های نمایش داده شده، اضافه شده است؛ برای اینکار در فایل init.c تغییر ایجاد شده است.

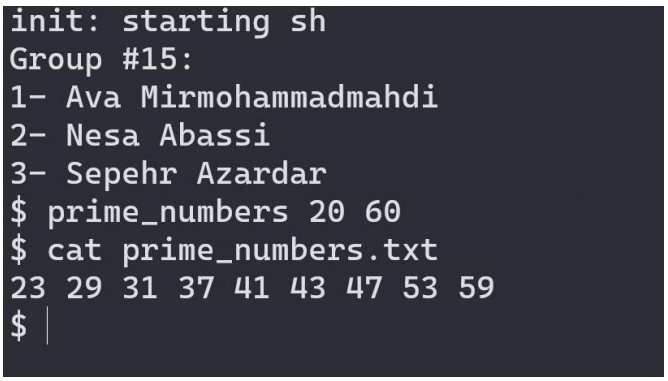
```
Microsoft Windows [Version 10.0.22000.978]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Ava\Downloads\Compressed\xv6-public-master\xv6-public-master>bash
root@Ava:/mnt/c/Users/Ava/Downloads/Compressed/xv6-public-master/xv6-public-master# make qemu
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-s
tack-protector -fno-pie -no-pie   -c -o init.o init.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _init init.o ulib.o usys.o printf.o umalloc.o
objdump -S _init > init.asm
objdump -t _init | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > init.sym
./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 941 total 1000
balloc: first 667 blocks have been allocated
balloc: write bitmap block at sector 58
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=
disk,format=raw -smp 2 -m 512  -display none
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 #15:
1- Ava Mirmohammadmahdi
2- Nesa Abassi
3- Sepehr Azardar
$
```

اضافه کردن چند قابلیت به کنسول xv6:



اجرا و پیاده‌سازی یک برنامه سطح کاربر:



برای این کار، prime_numbers را در Makefile در قسمت EXTRA و UPROGS اضافه کردیم.

5) سه وظیفه‌ی اصلی سیستم عامل را نام ببرید.

- مدیریت منابع کامپیوتر مانند واحد پردازش مرکزی، حافظه، درایوهای دیسک و چاپگرها(مدیریت منابع سخت افزار و مدیریت منابع نرم افزار)
- ایجاد یک رابط کاربری
- اجرا و ارائه خدمات برای نرم‌افزارهای کاربردی

۸) در Makefile متغیرهایی به نام های UPROGS و ULIB تعریف شده است. کاربرد آنها چیست؟

در UPROGS برنامه‌های کاربر قرار دارد (در واقع مخفف user programs است) و برای اجرای دستورات کاربر به کار گرفته می‌شوند مانند cat و echo و mkdir؛ پس از نوشتن کد prime_numbers نیز نیاز بود تا این برنامه را در بخش UPROGS در Makefile اضافه کنیم. در ULIB که مخفف user libraries است، کتابخانه‌هایی قرار دارند که در xv6 از آنها استفاده شده است مثل printf و umalloc .

۱۱) برنامه‌های کامپایل شده در قالب فایلهای دودویی نگهداری میشوند. فایل مربوط به بوت نیز دودویی است. نوع این فایل دودویی چیست؟ تفاوت این نوع فایل دودویی با دیگر فایل‌های دودویی کد xv6 چیست؟ چرا از این نوع فایل دودویی استفاده شده است؟

طبق Makefile دو فایل bootasm.S (به زبان اسمبلی) و bootmain.c (به زبان c) کامپایل شده و فایل‌هایی با پسوندo. برای آنها تشکیل می‌شود که توسط دستور LD این دو فایل باهم لینک شده و فایل bootblock.o تشکیل می‌شود که فرمت آن ELF (executable linkable format) است؛ حال دستور objcopy (که در سوال۱۲ توضیح داده شده است)، قسمت تکست این فایل را می‌گیرد و به فرمت raw binary تبدیل کرده و در فایل bootblock می‌ریزد. تفاوت فایل raw binary و ELF را در ادامه توضیح می‌دهیم؛ فایل های bin، فایل های تماما باینری هستند که شامل بیت و بایت هایی هستند که در آدرس مشخصی از حافظه قرار می‌گیرند در حالی که فایل های ELF، شامل اطلاعات ببیشتری مثل symbol tables و debig information هستند. . ELF یک فرمت باینری استاندارد برای سیستم‌عامل هایی مثل لینوکس است؛ برخی از قابلیت‌های ELF عبارتند از: dynamic loading و تحمیل کردن run time control به یک برنامه و متد پیشرفته برای ساختن shared libraryها. Bin آخرین راهی است که حافظه قبل از اینکه CPU آن را اجرا کند، به نظر می‌رسد. ELF یک ورژن فشرده شده‌ی آن است به طوری که CPU نمی‌تواند آن را مستقیما اجرا کند و linker باید آن را به حالتی که برای CPU قابل اجرا باشد تبدیل کند و offset ها را در موقعیت درست برگرداند.

دلیل استفاده از این نوع فایل دودویی این است که حجمش کم است چون این فایل شامل اطلاعات اضافه نیست و اولین چیزی که برای بوت شدن اجرا می‌شود نمی‌تواند حجم زیادی داشته باشد (حداکثر حجم هر سکتور ۵۱۲ بایت است). (حجم فایل bootblock به طور قابل توجهی از فایل bootblock.o کمتر است). دلیل دیگر استفاده از این نوع فایل این است CPU نمی‌تواند فایل ELF را مستقیما اجرا کند و سیستم‌عامل این نوع فایل را می‌شناسد پس نیاز است در بوت از فایلی استفاده کنیم که CPU مساقیما آن را بشناسد که این همان فرمت raw binary است.

در ادامه فایل bootblock را تبدیل به فایل اسمبلی کرده‌ایم و توضیحات آن داده شده است:

```
root@AVa:/mnt/c/Users/AVa/Desktop/xv6-public-master# objdump -D -b binary -m i386 -M addr16,data16 bootblock

bootblock:     file format binary


Disassembly of section .data:

00000000 <.data>:
   0:   fa                      cli
   1:   31 c0                   xor    %ax,%ax
   3:   8e d8                   mov    %ax,%ds
   5:   8e c0                   mov    %ax,%es
   7:   8e d0                   mov    %ax,%ss
   9:   e4 64                   in     $0x64,%al
   b:   a8 02                   test   $0x2,%al
   d:   75 fa                   jne    0x9
   f:   b0 d1                   mov    $0xd1,%al
  11:   e6 64                   out    %al,$0x64
  13:   e4 64                   in     $0x64,%al
  15:   a8 02                   test   $0x2,%al
  17:   75 fa                   jne    0x13
  19:   b0 df                   mov    $0xdf,%al
  1b:   e6 60                   out    %al,$0x60
  1d:   0f 01 16 78 7c          lgdtw  0x7c78
  22:   0f 20 c0                mov    %cr0,%eax
  25:   66 83 c8 01             or     $0x1,%eax
  29:   0f 22 c0                mov    %eax,%cr0
  2c:   ea 31 7c 08 00          ljmp   $0x8,$0x7c31
  31:   66 b8 10 00 8e d8       mov    $0xd88e0010,%eax
  37:   8e c0                   mov    %ax,%es
  39:   8e d0                   mov    %ax,%ss
  3b:   66 b8 00 00 8e e0       mov    $0xe08e0000,%eax
  41:   8e e8                   mov    %ax,%gs
  43:   bc 00 7c                mov    $0x7c00,%sp
  46:   00 00                   add    %al,(%bx,%si)
  48:   e8 fc 00                call   0x147
```

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

D- برای disassemble کردن کلی بکار می‌رود.

b- نوع فایل باینری را می‌گوید که ما آن را binary (raw binary) در نظر می‌گیریم.

m- معماری را مشخص می‌کند که ما آن را i386 گذاشتیم.

M- برای مشخص کردن آپشن‌های dissembler است که با تعیین addr16,data16 به آن می‌گوییم برای ذخیره آدرس و داده رجیسترهای ۱۶بیتی درنظر بگیر و به صورت پیش‌فرض رجیسترهای ۳۲بیتی درنظر گرفته می‌شود.(این قسمت برای شبیه شدن کد خروجی به bootasm.S اضافه شده است.)

۱۲) علت استفاده از دستور objcopy در حین اجرای عملیات make چیست؟

این دستور محتوای یک object file **را در یک فایل دیگر کپی می‌کند و برای خواندن و نوشتن** object file از کتابخانه GNU BFD استفاده می‌کند و می‌تواند object file مقصد را با فرمتی متفاوت از object file مبدا بنویسد. objcopy ، فایل‌های temporary درست می‌کند تا ترجمه‌هایش را انجام دهد و سپس آنها را حذف می‌کند؛ به تمامی فرمت‌های توصیف شده در BFD دسترسی دارد و به همین دلیل می‌تواند بیشتر فرمت‌ها را بدون اینکه صریحا به آن اعلام شود، تشخیص دهد. رفتارهای دقیق objcopy توسط آپشن‌های command-line مشخص می‌شود؛ مثلا با استفاده از s- می‌گوییم relocation و Symbol information را از فایل مبدا کپی نکن یا g- می‌گوید debugging symbols را از فایل مبدا کپی نکن.

Objcopy در Makefile سیستم عامل xv6 در بخش bootblock استفاده شده است؛ در اینجا objcopy بخش text فایل bootblock.o را در فایل bootblock کپی می‌کند همچنین objcopy در بخش entryother نیز، بخش تکست فایل bootblockother.o را در فایل entryother کپی می‌کند؛ این دستور در بخش initcode نیز فایل initcode.out را در فایل initcode کپی می‌کند و در آخر ، initcode و entryother و تعدادی فایل دیگر با هم لینک شده و کرنل را می‌سازند.

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

- ثبات عام منظوره: ثبات‌های عام منظوره در x86 با حرف e شروع می‌شوند که نشان‌دهنده‌ی کلمه extended است و ۳۲ بیتی بودن آنها بیان می‌کند. این ثبات‌ها ۸ عدد هستند با نام‌های: eip (این ثبات یک program counter است)، eax، ebx، ecx، edx، esi، ebp و esp . در این ثبات‌ها دیتا و عملیات ریاضی و برخی از پوینترها نگه داشته می‌شوند.

- ثبات قطعه: در این نوع ثبات‌ها، آدرس استک و دیتا و کد نگه داشته می‌شوند؛ SS و DS و CS مثال‌هایی از این نوع ثبات هستند که به ترتیب از راست به چپ در آنها اشاره‌گر به استک، اشاره‌گر به داده و اشاره‌گر به کد نگه داشته می‌شوند.

- ثبات وضعیت: در این نوع ثبات‌ها، اطلاعاتی راجه به وضعیت پردازنده نگهداری می‌شود؛ برای مثال در EFLAGS که یک ثبات وضعیت است، اطلاعات ... ,zero flag, sign flag, carry flag نگهداری می‌شود.

-ثبات کنترلی: و ظیفه‌ی این ثبات‌ها این است که CPU یا دیگر دستگاه‌های دیجیتال را کنترل کنند؛ از مثال‌های این نوع ثبات می‌توان به cr4 و cr3 و cr2 و cr0 اشاره کرد که کارشان تغییر مدل آدرس‌دهی، کنترل کردنinterrupt و paging و coprocessor ها است.

۱۸) کد معادل در entry.S هسته لینوکس را بیابید.

این کد در آدرس https://github.com/torvalds/linux/blob/master/arch/arm64/kernel/entry.S قرار دارد.

همان طور که دیده می‌شود چون آدرس مجازی کد هسته بزرگتر از 0X80100000 است نمی‌توان آن را اجرا کرد.

```
Group #15:
1- Ava Mirmohammadmahdi
2- Nesa Abassi
3- Sepehr Azardar
$ cat kernel.sym
cat: cannot open kernel.sym
$
```

۱۹) چرا این آدرس فیزیکی است؟

دلیل این که آدرس فیزیکی است این است که اگر اینطور نبود (یعنی از آدرس کجازی استفاده می‌شد)، نیاز به قسمتی فیزیکی داشتیم تا مشخص کند که این قسمت مجازی چه مکانی را نشان می‌دهد و باز هم به یک قسمت فیزیکی نیازمند بودیم.

۲۲) برای کد و دادههای سطح کاربر پرچم USER_SEG تنظیم شده است. چرا؟

اطلاعات هر کدام از قطعه‌هایی که تحت کنترل هسته و کاربر هستند نظیر سطح دسترسی آن(که از ۰ تا۳ می‌تواند باشد) و همچنین آدرس شروع آن قطعه را توسط یک descriptor در جدول توصیفگر سراسری مشخص شده است و همانطور که می‌دانیم یک بخش از حافظه در اختیار تمامی قطعه‌های سطح کاربر و هسته است؛ وقتی می‌خواهیم یک دستور را بخوانیم، آن را از طریق descriptor که در ابتدا و اینجا برای کاربر و هسته یکسان است پیدا می‌کنیم تا اجرا شود. (سطح دسترسی فعلی توسط سطح دسترسی descriptor مشخص می‌شود و با اینکه descriptor ها بخش‌های یکسان حافظه را نشان می‌دهند بازهم با وجود سطح دسترسی descriptor که تفاوت دارد می‌توان اینکار را انجام داد.) در هنگام اجرای یک دستور با وجود دسترسی بخش کاربر و هسته به یک بخش ممکن است به دلیل سطح دسترسی متفاوت، کاربر نتواند آن دستور را اجرا کند.

بخش بندی توضیح داده شده در فایل vm.c وجود دارد:

```
// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpuid()];
  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);
  lgdt(c->gdt, sizeof(c->gdt));
}
```

تعریف SEG نیز در فایل mmu.h آمده است که در زیر نشان داده شده است:

```
// Normal segment
#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 }
#define SEG16(type, base, lim, dpl) (struct segdesc)  \
{ (lim) & 0xffff, (uint)(base) & 0xffff,              \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
#endif
```

۲۳) جهت نگهداری اطلاعات مدیریتی برنامه‌های سطح کاربر ساختاری تحت عنوان struct proc ارائه شده است. اجزای آن را توضیح داده و ساختار معادل آن در سیستم عامل لینوکس را بیابید.

- unit sz : سایز حافظه (به واحد بایت) پردازه را نشان می‌دهد.
- pde_t* pgdir: اشاره‌گری به page table directory است.
- char* kstack: اشاره‌گری است که به پایین استک کرنلی که متعلق به این پردازه است، اشاره می‌کند.
- enum procstate state: وضعیت پردازه را مشخص می‌کند.
- int pid: شناسه‌ی پردازه را مشخص می‌کند.
- struct proc* parent: نشان‌دهنده‌ی والد پردازه (سازنده‌ی پردازه) است.
- struct trapframe* tf: اشاره‌گری است که به trapframe متعلق به system call فعلی، اشاره می‌کند.
- struct context* context: switch کردن برای اجرای پردازه در اینجا انجام می‌شود.
- void* chan: در صورت غیرصفر بودن یعنی پردازه در حالت sleeping است.(برای multi-process)
- voide* killed: درصورت غیرصفر بودن یعنی پردازه kill شده است.
- struct file* ofile[NOFILE]: فایل های بازشده توسط پردازه است.
- struct inode* cwd: directory کنونی است.
- char name[16]: نام پردازه است.

ساختار معادل struct proc در لینوکس، task_struct است که در خط ۷۲۷ کد موجود در لینک زیر آمده است.

https://github.comtorvalds/linux/blob/master/include/linux/sched.h

۲۷) کدام بخش از آماده‌سازی سیستم، بین تمامی هسته‌های پردازنده مشترک و کدام بخش اختصاصی است؟ (از هر کدام یک مورد را با ذکر دلیل توضیح دهید.) زمانبند روی کدام هسته اجرا می‌شود؟

در انتهای entry.s امکان اجرای کد C هسته فراهم میشود تا در انتها تابع main صدا زده شود و این تابع در هسته‌ای که سیستم‌عامل بوت کرده است فراخوانی می‌شود. (در این تابع ۱۸ تابع دیگر فراخوانی شده است.) بقیه‌ی CPUها از entryother.s به تابع mpenter می‌روند که در این تابع ۴ تابع دیگر فراخوانی می‌شود که این چهار تابع در تابع mainنیز فراخوانی شده‌اند.(تابع switvhkvm که در mpenter است در تابع kvmalloc که در تابع main است فراخوانی شده‌است)

در واقع کارهایی که در این چهار تابع انجام می‌شود بین تمامی هسته‌های پردازنده مشترک است و کارهایی که در ۴-۱۸ تابع دیگر در main انجام می‌شوند اختصاصی است. برای مثال تابع consoleinit در تابع main فراخوانی می‌شود و اختصاصی هسته‌ای است که سیستم‌عامل را بوت کرده‌است؛ در این تابع، initlock فراخوانی می‌شود که برای قفل اولیه است و نوشته‌های روی کنسول اولیه است و نیازی نیست بین تمامی هسته‌ها مشترک باشد. تابع mpmain بین تمامی هسته‌ها مشترک است که در آن کد مشترک setup کردن CPU ها است و در آن کار لود کردن idt register انجام می‌شود و به تابع startothers می‌گوید که آماده است و شروع به اجرای پردازه‌ها می‌کند. زمان‌بند یا همان scheduler در تابع mpmain فراخوانی می‌شود و بین تمامی هسته‌ها مشترک است و هر CPU بعد از set up کردن خودش، آن را فراخوانی می‌کند.

اشکال‌زدایی:

۱) برای مشاهده‌ی Breakpoint ها از چه دستوری استفاده می‌شود؟

برای این کار از دستور main info breakpoints استفاده می‌کنیم.

```
(gdb) break cat.c:10
Note: breakpoint 1 also set at pc 0x97.
Breakpoint 2 at 0x97: file cat.c, line 10.
(gdb) break cat.c:12
Note: breakpoints 1 and 2 also set at pc 0x97.
Breakpoint 3 at 0x97: file cat.c, line 12.
(gdb) main info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00000097 in cat at cat.c:12 inf 1
        breakpoint already hit 1 time
1.1                    y   0x00000097 in cat at cat.c:12 inf 1
2       breakpoint     keep y   0x00000097 in cat at cat.c:10 inf 1
2.1                    y   0x00000097 in cat at cat.c:10 inf 1
3       breakpoint     keep y   0x00000097 in cat at cat.c:12 inf 1
3.1                    y   0x00000097 in cat at cat.c:12 inf 1
(gdb)
```

۲) برای حذف یک Breakpoint از چه دستوری و چگونه استفاده می‌شود؟

برای این کار از دستور clear filename:line استفاده می‌کنیم و به جای filename، نام فایل مورد نظر و به جای line، نام خطی که می‌خواهیم Breakpoint از آن حذف شود را می‌نویسیم.

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

bt همان backtrace است؛ با استفاده از این دستور لیستی از فراخوانی‌های توابع که در حال حاضر در یک thread فعال هستند و در استک اضافه شده‌اند، نمایش داده می‌شود.

```
(gdb) b cat.c:12
Breakpoint 1 at 0x97: file cat.c, line 12.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00000097 in cat at cat.c:12
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=0) at cat.c:12
12        while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=0) at cat.c:12
12        while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) bt
#0  cat (fd=0) at cat.c:12
#1  0x00000087 in main (argc=1, argv=0x2ff4) at cat.c:30
(gdb)
```

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

دستور x آدرس expression را به عنوان ورودی می‌گیرد و محتوای آن آدرس را چاپ می‌کند ولی دستور print می‌تواند یک expression را به عنوان ورودی بگیرد و مقدارش را نمایش دهد؛ همچنین نحوه‌ی نمایش اطلاعات با استفاده از دستور x متفاوت از نمایش آن با استفاده از دستور print است.

برای چاپ محتوای یک ثبات خاص می‌توان از دستور info register name استفاده کرد که name در آن نشان‌دهنده‌ی نام رجیستر است.

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

برای نمایش وضعیت رجیسترها از دستور info registers استفاده می‌شود:

```
(gdb) info registers
eax            0x1                1
ecx            0xa                10
edx            0x0                0
ebx            0x898              2200
esp            0x2fc0             0x2fc0
ebp            0x2fd8             0x2fd8
esi            0x0                0
edi            0x0                0
eip            0x97               0x97 <cat+7>
eflags         0x246              [ IOPL=0 IF ZF PF ]
cs             0x1b               27
ss             0x23               35
ds             0x23               35
es             0x23               35
fs             0x0                0
gs             0x0                0
fs_base        0x0                0
gs_base        0x0                0
k_gs_base      0x0                0
cr0            0x80010011         [ PG WP ET PE ]
cr2            0x0                0
cr3            0xdfbb000          [ 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 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm6           {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}
xmm7           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 12 times>, 0x80, 0x1f, 0x0, 0x0}, v8_int16 = {0x0, 0x0, 0x0, 0x0,
 0x0, 0x0, 0x1f80, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x1f80}, v2_int64 = {0x0, 0x1f8000000000}, uint128 = 0x1f8000000000000000000000000000}
mxcsr          0x1f80             [ IM DM ZM OM UM PM ]
(gdb)
```

برای نمایش وضعیت متغیرهای محلی از دستور info locals استفاده می‌شود؛ همچنین برای مشاهده وضعیت تمامی متغیرهای global دستور info variable استفاده می‌شود.

```
(gdb) info locals
n = <optimized out>
(gdb) info variables
All defined variables:

File cat.c:
5:      char buf[512];

File umalloc.c:
21:     static Header base;
22:     static Header *freep;

Non-debugging symbols:
0x00000878  digits
0x00000b5c  __bss_start
0x00000b5c  _edata
0x00000d80  _end
(gdb)
```

ESI و EDI رجیسترهای عام منظوره هستند. بعضی از instruction ها زمانی که بلوکی از دیتا را کپی می‌کنند، از ESI و EDI به ترتیب برای نگهداری نشانگر به آدرس source و destination استفاده می‌کنند.

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

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

● نحوه و زمان تغییر مقدار متغیرهای درونی (برای مثال، e.input در چه حالتی تغییر میکند و چه مقداری میگیرد)

در این struct، چهار متغیر char buf[INPUT_BUF] و unit r و unit w و unit e وجود دارد:

input.buf: در buf که آرایه‌ای از کاراکترها است و به اندازه‌ی ۱۲۸ کاراکتر جا دارد، رشته ای که در ترمینال می‌نویسیم نوشته می‌شود.

input.e: انتهای خطی که در آن نوشته‌ایم و یا به عبارتی مکان cursor را نشان می‌دهد.

input.r: نشان‌دهنده‌ی آخرین جایی است که کنسول آن را خوانده ودستورش را اجرا کرده‌است.

input.w: نشان‌دهنده‌ی مکان ابتدای خط است.

(در واقع input.w و input.r یک چیز را نشان می‌دهند چون آخرین جایی که کنسول آن را خوانده برابر است با اولین جایی که می‌توانیم در آن بنویسیم.)

input.e زمانی تغییر می‌کند که جایcursor عوض شود، مثلا وقتی در ترمینال تایپ می‌کنیم؛ همان طور که در تصویر مشخص است در ابتدا مقدار این متغیر ۰ بوده و زمانی که با تایپ کردن hello hi به اندازه‌ی ۸ کاراکتر جلو آمدیم، مقدار آن به ۸ تغییر یافت؛ مقدار input.buf از مقدار تهی به hello hi تغییر می‌کند و مقادیر input.r و input.w همان ۰ باقی می‌ماند.

در تصویر زیر breakpoint را در خط ابتدای case C('U') گذاشته‌ایم و همانطور که می‌بینیم بعد از فشردن ctrl + u تغییرات زیر در مقادیر متغیرها رخ می‌دهد.

```
(gdb) break console.c:206
Breakpoint 1 at 0x80100953: file console.c, line 207.
(gdb) print input.e
$1 = 0
(gdb) print input.w
$2 = 0
(gdb) print input.r
$3 = 0
(gdb) print input.buf
$4 = '\000' <repeats 127 times>
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, consoleintr (getc=0x80105f90 <uartgetc>) at console.c:207
207           while(input.e != input.w &&
(gdb) print input.e
$5 = 8
(gdb) print input.w
$6 = 0
(gdb) print input.r
$7 = 0
(gdb) print input.buf
$8 = "hello hi", '\000' <repeats 119 times>
(gdb)
```

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 i
nodestart 32 bmap start 58
init: starting sh
Group #15:
1- Ava Mirmohammadmahdi
2- Nesa Abassi
3- Sepehr Azardar
$ hello hi
```

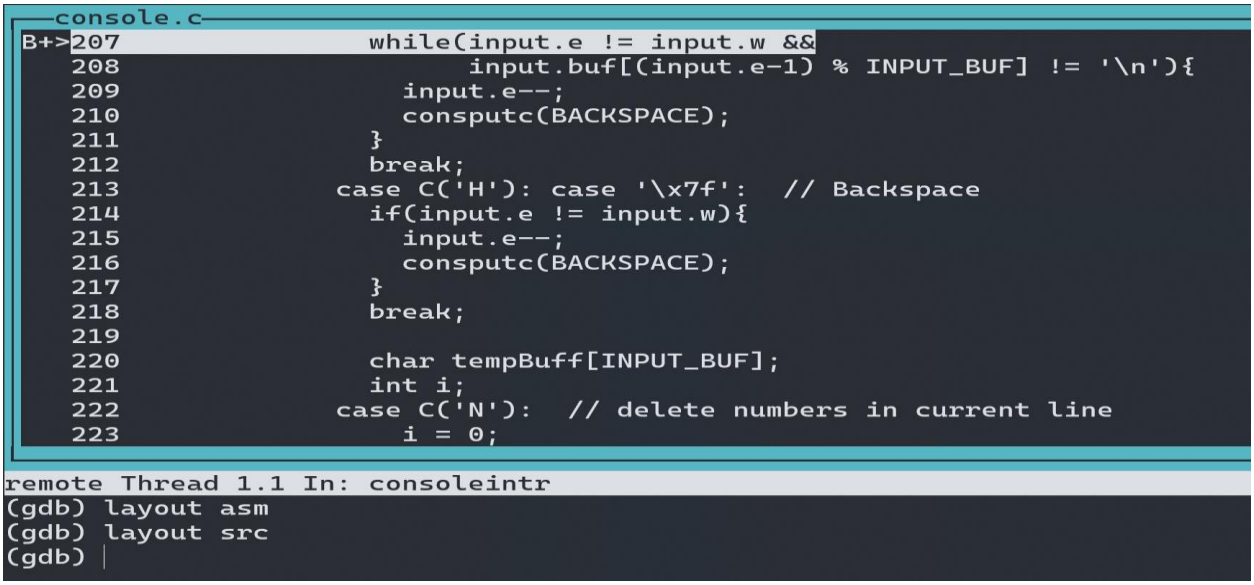
۷) خروجی دستورهای layout src و layout asm در TUI چیست؟

خروجی دستور layout src برنامه در حالت کد سورس‌اش است و خروجی دستور layout asm برنامه در حالت کد اسمبلی‌اش است.

با فشردن ctrl + x + a وارد محیط TUI شدیم و با دستور layout asm برنامه‌مان به حالت کد اسمبلی درمی‌آید.

```
B+>0x80100953 <consoleintr+243>     mov    0x8010ffa8,%eax
   0x80100958 <consoleintr+248>     cmp    %eax,0x8010ffa4
   0x8010095e <consoleintr+254>     je     0x80100892 <consoleintr+50>
   0x80100964 <consoleintr+260>     sub    $0x1,%eax
   0x80100967 <consoleintr+263>     mov    %eax,%edx
   0x80100969 <consoleintr+265>     and    $0x7f,%edx
   0x8010096c <consoleintr+268>     cmpb   $0xa,-0x7fef00e0(%edx)
   0x80100973 <consoleintr+275>     je     0x80100892 <consoleintr+50>
   0x80100979 <consoleintr+281>     mov    %eax,0x8010ffa8
   0x8010097e <consoleintr+286>     mov    0x8010a558,%eax
   0x80100983 <consoleintr+291>     test   %eax,%eax
   0x80100985 <consoleintr+293>     je     0x80100a30 <consoleintr+464>
   0x8010098b <consoleintr+299>     cli
   0x8010098c <consoleintr+300>     jmp    0x8010098c <consoleintr+300>
   0x8010098e <consoleintr+302>     mov    0x8010ffa8,%eax
   0x80100993 <consoleintr+307>     xor    %esi,%esi
   0x80100995 <consoleintr+309>     cmp    %eax,0x8010ffa4
remote Thread 1.1 In: consoleintr
(gdb) layout asm
(gdb)
```

با دستور layout src نیز کد ما به حالت سورس کد مه در اینجا زبان سی است در می‌آید.



۸) برای جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستورهایی استفاده می‌شود؟

دستور up n، به اندازه‌ی n فریم به سمت بالای استک می‌رود و برای اعداد مثبت n، به سمت داخلی‌ترین فریم پیش می‌رویم.(در حالت پیش‌فرض n برابر با ۱ است.) دستور down n به اندازه‌ی n فریم به سمت پایین استک می‌رود و برای اعداد مثبت n، به سمت خارجی‌ترین فریم پیش می‌رویم؛ دستور دیگر، frame [frame-selection-spec] است به طوری که frame-selection-spec می‌تواند موارد زیر باشد:

level یا level num: می‌توان شماره فریمی که می‌خواهیم به آن برویم را انتخاب کنیم؛ عدد ۰ به معنای داخلی‌ترین فریم(فریمی که در حال حاضر درحال اجرا باشد) است و فریم ۱، فریمی است که فریم ۰ را فراخوانی کرده است.

address stack-address: فریم با آدرس stack-address را انتخاب می‌کند؛ stack-address یک فریم را می‌توان در خروجی دستور info frame مشاهده کرد.

function function-name: فریم استک را برای فانکشن function-name انتخاب می‌کند و اگر چند فریم استک برای این فانکشن وجود داشت، داخلی ترین فریم استک انتخاب می‌شود.

view stack-address [pc-addr]: یک فریم که متعلق به backtrace اشکال‌زدای GDB نیست را نشان می‌دهد؛ فریم نشان داده شده دارای آدرس stack-address و program counter با آدرس pc-addr است.(اختیاری)؛ این قابلیت معمولا زمانی کاربردی است که زنجیره‌ی فریم‌های استک توسط یک باگ خراب شده باشد و در این صورت GDB نمی‌تواند به طور مناسب به همه‌ی فریم‌ها شماره بدهد و همچنین زمانی که برنامه‌ی ما چندین استک دارد و بین آنها سوییچ می‌کند نیز کاربردی است.

برخی از دستورهای بالا در ترمینال پایین نشان داده شده‌است:

```
Thread 1 hit Breakpoint 1, cat (fd=0) at cat.c:12
12          while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) up
#1  0x00000087 in main (argc=1, argv=0x2ff4) at cat.c:30
30              cat(0);
(gdb) down
#0  cat (fd=0) at cat.c:12
12          while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) frame level 0
#0  cat (fd=0) at cat.c:12
12          while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) frame 1
#1  0x00000087 in main (argc=1, argv=0x2ff4) at cat.c:30
30              cat(0);
(gdb) frame function cat
#0  cat (fd=0) at cat.c:12
12          while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb)
```