پروژه اول

گروه ۳: علی هدائی، پویا صادقی، علی عطاللهی

۱_ معماری سیستم عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

سیستم عامل xv6 بر اساس unix version 6(v6) ساخته دنیس ریچی و کن تامپسون ساخته شده است. پس از ساختارو استایل v6 بهره میبرد و اما با ANCI C برای پردازنده های چند هسته ای مبتنی بر x86 پیاده سازی شده است.

در فایل x86.h ذکر شده است که دستور های نوشته شده از ساختار و دستورهای مربوط به پردازنده های مبتنی بر x86 استفاده شده است.در فایل asm.h هم ذکر شده است که از ماکرو های اسمبلر برای ساختن بخش های x86 استفاده شده است. همچنین در traps.h هم ذکر شده که از trap های x86 استفاده شده است. همچنین در mmu.h هم نوشته شده که از mmu مربوط به x86 استفاده شده است.

۲_ یک پردازه در سیستم عامل xv6 از چه بخش هایی تشکیل شده است؟ این سیستم عامل به صورت کلی چگونه پردازنده را به پردازه های مختلف اختصاص میدهد؟

هر پردازه (process) در xv6 از user space memory که شامل stack و data و instruction و همچنین وضعیت چیش پردازش (pre-process state) که مربوط به هسته (kernel) است، تشکیل شده است.Xv6 میتونه پردازه های time-share انجام بده. آن بین cpu های موجود برای مجموعه ای از پردازش ها منتظر اجرا هستن سوییچ میکند. زمانی که پردازه ای اجرا نمیشود ، xv6 رجیستر های cpu آن را ذخیره میکند تا برای پردازش دفعه بعدی استفاده کند. کرنل به هر پردازه یک آیدی یا همان pid نسبت میدهد.

۴_ فراخوانی های سیستمی fork و exec چه عملی انجام میدهند؟ از نظر طراحی ادغام نکردن این دو چه مزیتی دارد؟

fork: پردازشی که ممکن است یه پردازش دیگه تولید کند از فراخوانی سیستمی fork استفاده میکند. Fork یه پردازش جدیدی به نام پردازش child میسازد ، دقیق با محتوای حافظه که آن پردازش را فراخوانی کرده است ، که پردازش parent نامیده میشود.Fork هم در child و هم در parent مقداری برمیگرداند. در parent ، pid ی child را برمیگرداند و در child ، ۰ را برمیگرداند.

Exec: فراخوانی سیستمی exec ، ممووری پردازشی که فراخوانی شده است را با یک ممووری ایمیجی که در یک فایل در سیستم ذخیره شده است جایگزین میکند. آن فایل باید فورمت مشخصی داشته باشد که مشخص شده باشد که در کدام قسمت فایل دستورات ذخیره شده ، کدام قسمت از آن داده است و از کدام دستور باید شروع کند.xv6 از فورمت ELF استفاده میکند. زمانی که exec با موفقیت به پایان میرسد ، به برنامه ای که آن را فراخوانده است باز نمیگردد بلکه دستورات ای که از فایل لود شده اند ، از نقطه مشخص شده شروع به اجرا میشوند.exec دو آرگومان میگیرد یکی اسم فایلی که قرار است اجرا شود و دیگری آرایه ای استرینگی از آرگومان ها.

جدا کردن فراخوانی ها برای ساختن یک پردازش و لود کردن پردازش استفاده های هوشمندانه ای در IO redirection دارد. برای جلوگیری از گردازش های تکراری و سریع جایگزین کردن آن ، کرنل ها با استفاده از تکنیک های ممووری مجازی ، پیاده سازی fork را برای این زمینه بهبود میبخشند .

۸_ در Makefie متغیر هایی به نام UPROGS و ULIB تعریف شده است. کاربرد آنها چیست؟

UPROGS : شامل لیستی از user program هایی که باید بیلد شوند و یوزر میتواند از آنها استفاده کند را نگه میدارد. به همین دلیل اگر بخواهیم user program ای بسازیم و اضافه کنیم باید در این قسمت نامش را ذکر کنیم.

برای مثال برنامه یا دستور هایی که کاربر میتوانند اجرا کنند شامل
_cat\ _echo\ _forktest\ _grep\ _init\ _kill\ _ln\ _ls\ _mkdir\ _rm\ _sh\ _stressfs\ _usertests\ _wc\ _zombie\

که میبینیم نامشان همگی در UPROG لیست شده اند

ULIB : برای اینکه کاربران بتوانند برنامه هایی توسعه دهند یکسری دستورات و لایبرری های برای آنها ساخته شده است که این متغیرمسئول نگه داشتن آنهاست مثل فایل آبجکت ulib.o که دستور های strcpy و memset و ... را دارد. فایل آبجکت usys.o که سیستم کال ها را شامل میشود، umaloc.o که شمال دستور های dynamic memory allocation هاست و printf.o .

۱۱_ برنامه های کامپایل شده در قالب فایل های دودویی نگه داری می شوند .فایل مربوط به بوت نیز دودویی است .نوع این فایل دودویی چیست؟ تفاوت این نوع فایل دودویی با دیگر فایل های دودویی کد xv6 چیست؟ چرا از این نوع فایل دودویی استفاده شده است؟ این فایل را به زبان قابل فهم انسان(اسمبلی) تبدیل نمایید.

همه آبجکت فایل ها ELF اند.(هدر ELF دارند.) ولی فایل bootblock این هدر را ندارد و در واقع یک فایل raw binary و هدری ندارد. و این فایل در سکتور اول مموری لود میشود(سکتور اول مربوط به بوت است و کرنل از سکتور دوم به بعد است.) از آنجا که وقتی سیستم بالا می آید cpu از سکتور اول که مربوط به بوت است شروع به اجرای دستورات میکند و قاعدتا چون هنوز سیستم عامل اجرا نشده است نمیتواند فایل ELF را متوجه شود(این سیستم عامل است که ELF را میشناسد) به همین دلیل فقط قسمت تکست bootblock.o که مربوط به instruction ها است را جدا میشود و در حافظه قرار میگیرد تا cpu بتواند آنرا اجرا کند. همچنین این کد به معماری cpu وابسته است. به همین دلیل باید برای human readable کردن آن باید معماری آنرا که i386 است و ۱۶ بیت است ذکر کنیم.

```
   amd64       Display instruction in AMD64 ISA
   intel64     Display instruction in Intel64 ISA
ali@ali-ASUS-TUF-Gaming-A17-FA706II-FX706II:~/Desktop/OS/project1/os-project1/codes/xv6$ objdump -D -b  binary -mi386 -Maddr16,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
  4b:   00 00                   add    %al,(%bx,%si)
  4d:   66 b8 00 8a 66 89       mov    $0x89668a00,%eax
  53:   c2 66 ef                ret    $0xef66
  56:   66 b8 e0 8a 66 ef       mov    $0xef668ae0,%eax
  5c:   eb fe                   jmp    0x5c
```

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

همانطور که در بخش قبل گفتیم برای اینکه قسمت تکست فایل bootblock.o را که همان instruction ها هستند را بتواند کپی کند و فایل raw binary عه bootblock را بسازد از objcopy استفاده شده است.

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

General purpose registers : EAX, EBX, ECX, EDX, ESI, and EDI رجیسترهایی هستند برای نگه داشتن متغیر ها حین محاسبه کردن. و همچنین رجیستر EIP همان PC است.(instruction pointer)

segment registers : اشاره گر به قطعه های مختلف مثل استک ، داده و کد در آنها نگه داری میشود مثل SS که پوینتر به استک یا CS که پوینتر به کد را نگه میدارند.

Status Registers : اطلاعات وضعیت کنونی پردازنده را نگه میدارند. مثل EFLAGS که اطلاعتی درباره فلگ های zero و overflow و... را نگه میدارد.

Control registers : مسئول تغییر و یا کنترل پردازنده را بر عهده دارند مثل CR0 که وظیفه های مثل فعال کردن protected mode و یا سوییچ کردن بین تسک ها را دارد.

۱۸_ کد معادل entry.s را درهسته لینوکس بیابید.

https://github.com/torvalds/linux/blob/master/arch/arm64/kernel/entry.S

۱۹_ چرا این آدرس فیزیکی است؟

چون اگر آن را به صورت مجازی در نظر میگرفتیم باز باید یک بخش فیزیکی در نظر میگرفتیم که آدرس این مکان مجازی را به فیزیکی تبدیل کند که عملا کار بیهوده ای است.

۲۲_ علاوه بر صفحه بندی در حد ابتدایی از قطعه بندی به منظور حفاظت هسته استفاده خواهد شد .این عملیات توسط seginit انجام می گردد .همان طور که ذکر شد، ترجمه قطعه تأثیری بر ترجمه آدرس منطقی نمی گذارد .زیرا تمامی قطعه ها اعم از کد و داده روی یکدیگر می افتند. با این حال برای کد و داده های سطح کاربر پرچم SEG_USER تنظیم شده است. چرا؟

زیرا بتوان تفاوتی بین پردازه های سطح کاربر و پردازه های سطح کرنل ایجاد نمود .از انجایی که محتوای هر دو این پردازنده ها در یک فضای فیزیکی قرار گرفته اند، با این کار می توان تشخیص داد که آن داده ها یا کد ها، داده ها و کد های سطح کاربرمی باشند و اجازه دسترسی به هسته را ندارند.

۲۳_ جهت نگه داری اطلاعات مدیریتی برنامه های سطح کاربر ساختاری تحت عنوان struct proc ارائه شده است. اجزای آن را توضیح داده و ساختار معدل آن در سیستم عامل لینوکس را بیابید.

Sz : سایز ممووری مربوط به پردازه (به بایت)

Pgdir : پوینتر به page table است.

Kstack : پایین استک کرنل برای این پردازه را مشخص می کند.

State : وضعیت پردازه را مشخص میکند.

Pid : آیدی اختصاص داده شده به این پردازه را مشخص میکند.

Parent : والد یا پدر با در واقع سازنده این پردازه را مشخص میکند.

Tf : فریم trap برای system call فعلی

Context : context switching را نگه میدارد.

Chan : اگر صفر نباشد به معنی خوابیدن پردازه است.

Killed : اگر صفر نباشد به معنی kill شدن پردازه است.

Ofile : فایل های باز شده توسط این پردازه را مشخص میکند.

Cwd : پوشه فعلی را مشخص میکند.

Name :نام این پردازه را مشخص میکند.

معدل ان در لینوکس در کد زیر است(task_struct):

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

۲۷_ کدام بخش از آماده سازی سیستم، بین تمامی هسته های پردازنده مشترک و کدام بخش اختصاصی است؟ (از هر کدام یک مورد را با ذکر دلیل توضیح دهید).زمان بند روی کدام هسته اجرا می شود؟

همانطور که در کد main.c مشخص است یکسری دستورات مثل allocate کردن phsycal page و ساختن trap vector ها ، ساختن لینک لیستی از بافر ها و ... به عهده Bootstrap processor است. ولی کارهایی مثل مپ کردن آدرس منطقی به آدرس مجازی که توسط تابع seginit انجام میشود بین آنها مشترک است.

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}

// Other CPUs jump here from entryother.S.
static void
mpenter(void)
{
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
```

اشکال زدایی

سوال۱- برای مشاهده Breakpointها از چه دستوری استفاده میشود؟

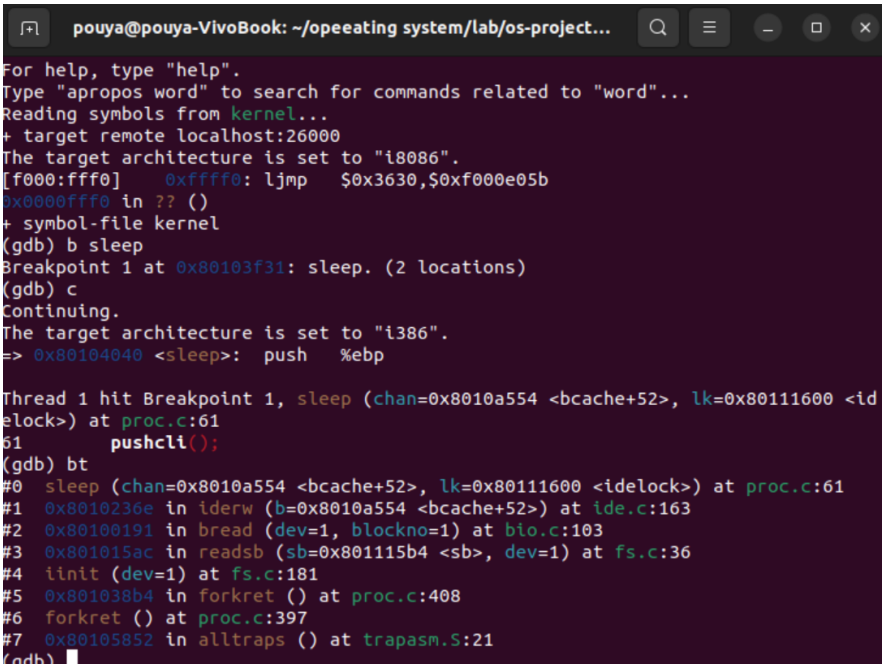
از دستور maintenance info breakpoints استفاده میشود.(همچنین دستور info break)

سوال۲- برای حذف یک BreakPoint از چه دستوری و چگونه استفاده میشود؟

میتوان با دستور clear <filename>:<linenumber>، بریک پوینت قرار داده شده در خط linenumber از فایل filename را حذف کرد. همچنین دستورات مشاهده BreakPointها، به هرکدام یک شماره نسبت میدهند که به فرمت del <number> میتوان BreakPoint مورد نظر را حذف کرد.

سوال۳- دستور bt را اجرا کنید. خروجی آن چه چیزی را نشان میدهد؟

این دستور، همان backtrace و یک لیست از زنجیره توابع فراخوانی شده تا رسیدن به آن Breakpoint را نشان میدهد. این کار را به کمک استک فراخوانی توابع انجام میدهد و باعث میشود بدانیم که تابع چگونه به جایگاه کنونی(محل Breakpoint که میتواند یک تابع خاص یا یک آدرس از حافظه ویا خطی در سورس کد باشد) رسیده است.

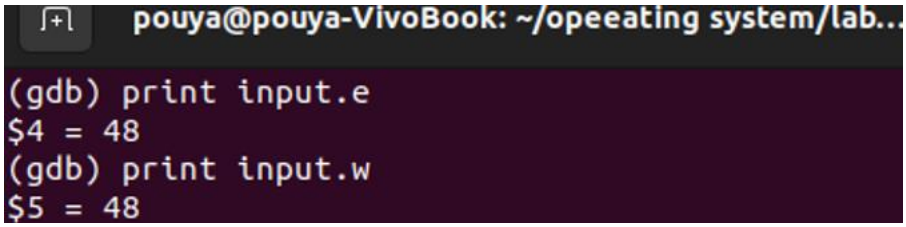


در این تصویر، #۰ مربوط فریم فعلی و محل bp میباشد. در #۱، #۰ صدا زده شده و الی آخر.

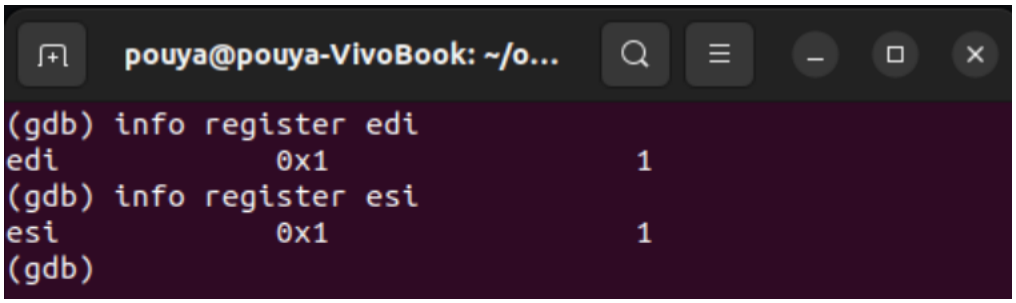
سوال ۴- دو تفاوت دستورهای x و print را توضیح دهید. چگونه میتوان یک محتوای خاص را چاپ کرد؟

دستور print، یک عبارت(EXP) میگیرد که مقدار(value) آنرا با درنظر گرفتن آپشن های مشخص شده و به فرمت مشخص شده نمایش میدهد. مبنای کارکرد دستور x، آدرس میباشد و محتوای(content) آنرا نمایش میدهد. همچنین بخش FMT در این دستور (برخلاف print) الزامی میباشد و بصورت تعداد تکرار همراه با حرف فرمت (مثل o, x, d, u, t, a, f, i, c, s, z) و حرف اندازه (مثل b, h, w, g)

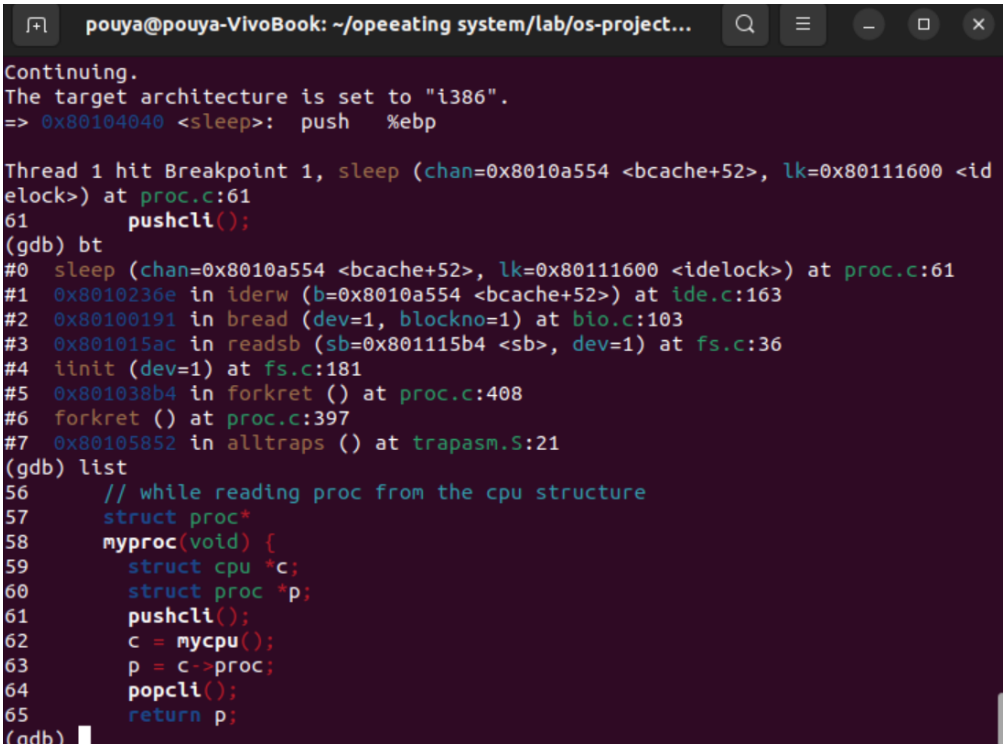
میباشد. یک نمونه از استفاده از دستور print:



با کمک دستور info register <reg_name> میتوان محتوای یک رجیستر خاص را چاپ کرد:



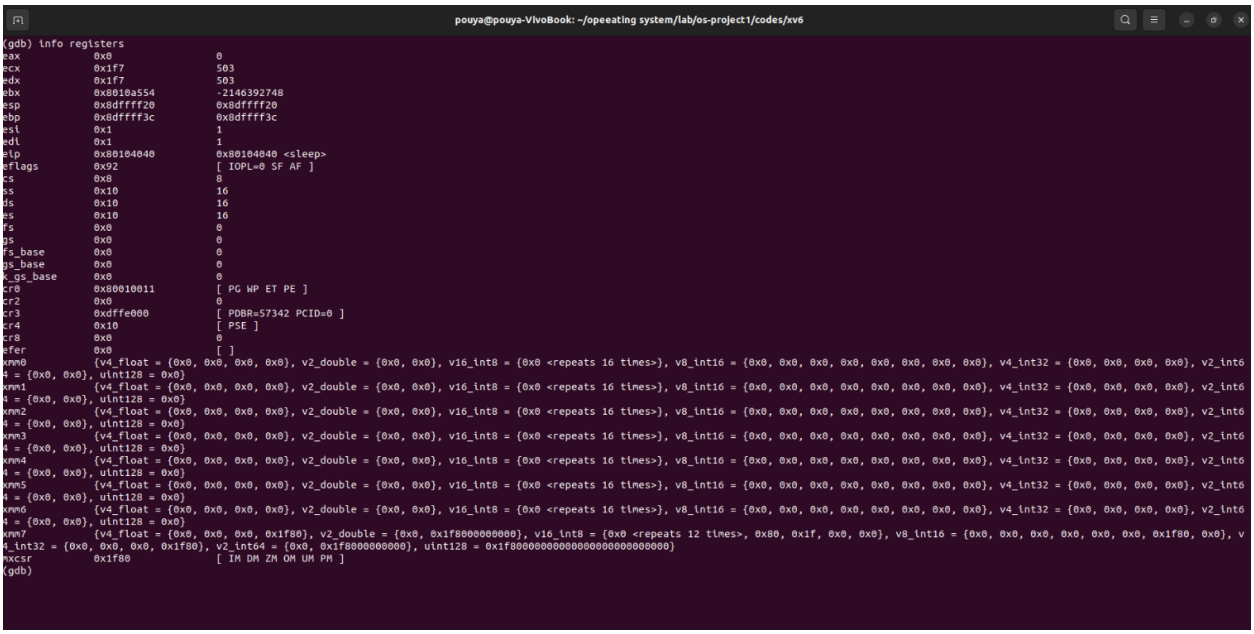
خروجی دستور list:



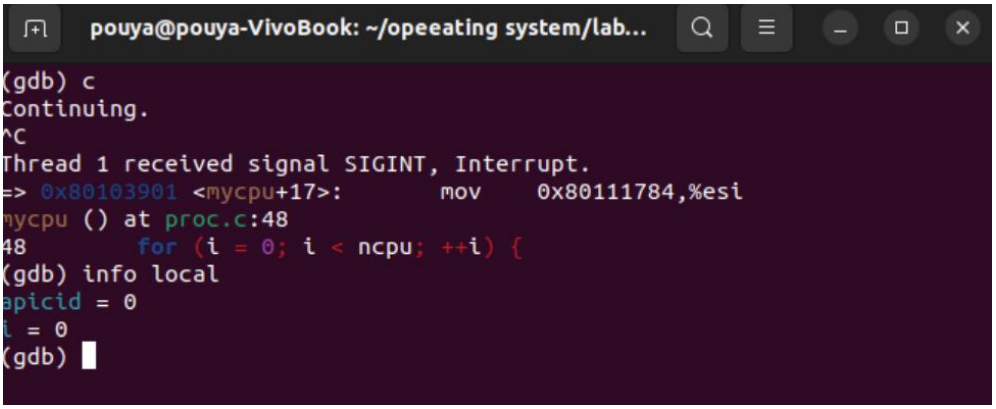
سوال ۵-برای نمایش وضعیت ثباتها از چه دستوری استفاده میشود؟ متغیرها محلی چطور؟ توضیح دهید که در معماری x86 رجیسترهای edi و esi نشانگر چه چیزی هستند؟

با دستور info register <reg_name> میتوان وضعیت یک ثبات خاص را مشاهده کرد.

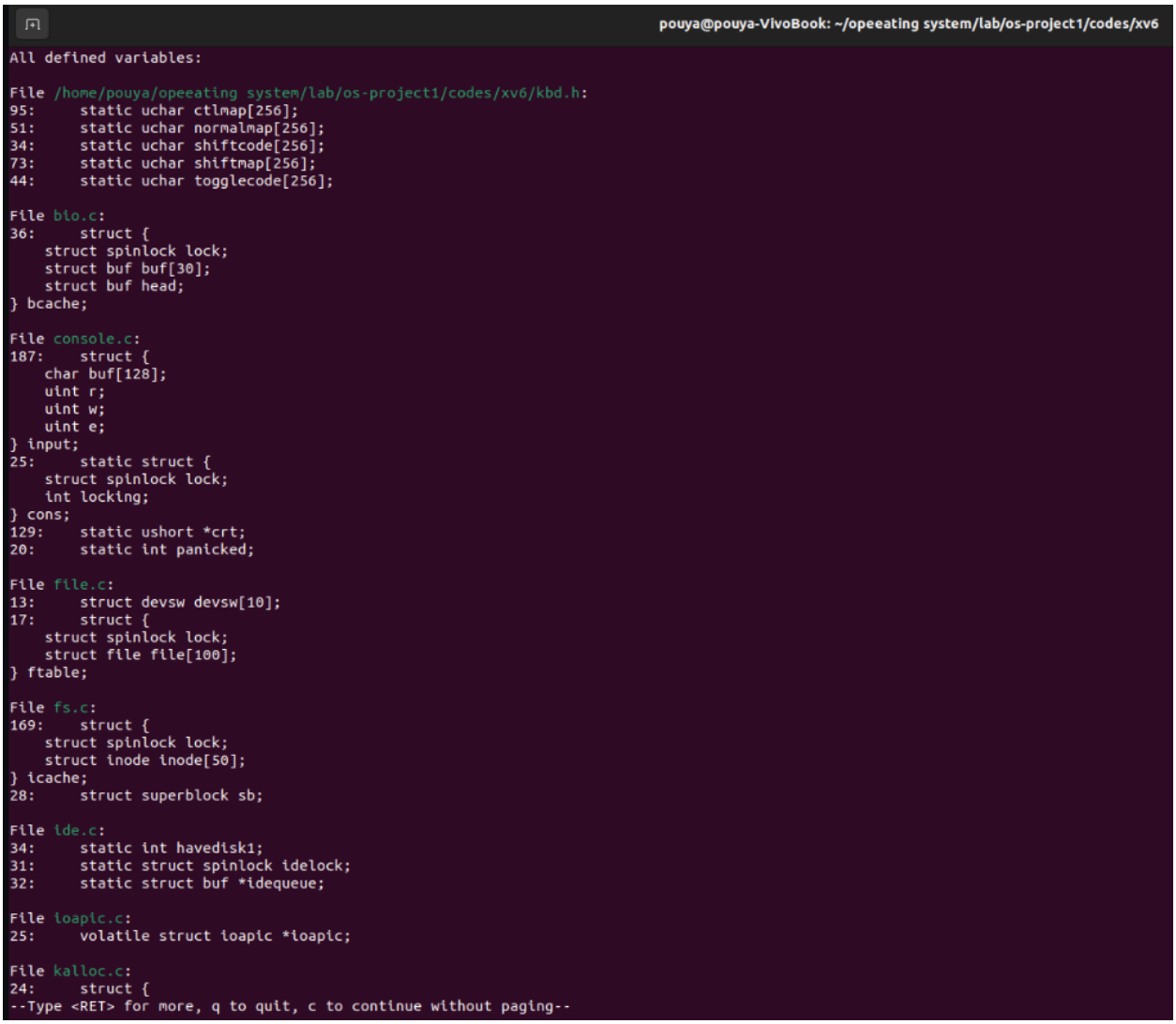
با دستور info registers میتوان وضعیت ثباتها را نشان داد:



با دستور info locals میتوان وضعیت متغیرهای محلی را نشان داد:



همچنین با دستور info variables نیز میتوان وضعیت متغیر های global و static را مشاهده کرد:



برخی از عملیاتها برای انجام، به رجیسترهای edi و esi احتیاج دارند. همانند عملیات مربوط به رشته ها یا عملیات مربوط به کپی یا مقدار دهی یک آرایه در حافظه. در اینگونه عملیاتها، esi رجیستر مکان مبدا (نگهدارنده source) و edi رجیستر مکان مقصد (نگهدارنده destination) میباشند.

برای بررسی تخصصی تر:

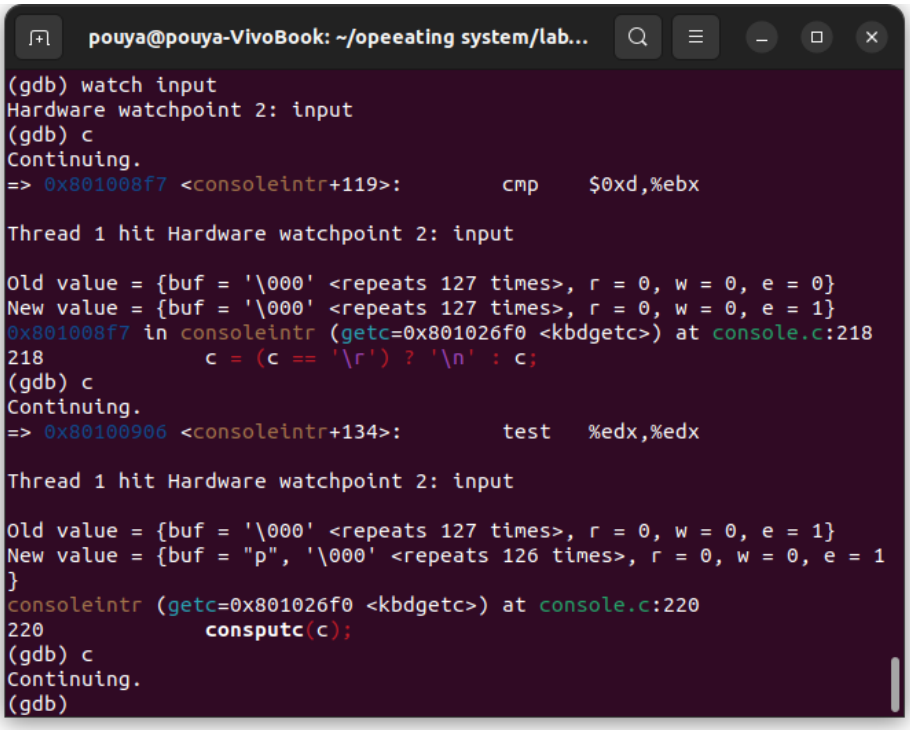
باید ذکر کرد که DI مخفف برای Destination Index و SI مخفف برای Source Index میباشند. فقط تعدادی عملیات وجود دارد که با این رجیسترها میتوان انجام داد: REP STOSB|MOVSB|SCASB که عملیات ذخیره سازی، بارگذاری و اسکن مکرر را انجام میدهند. برای خلاصه کردن، به این مثال اکتفا میکنیم که MOVSB میتواند برای انتقال داده ها از یک بافر به بافر دیگر بکار برود (کلا جابه جایی بایت ها)

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

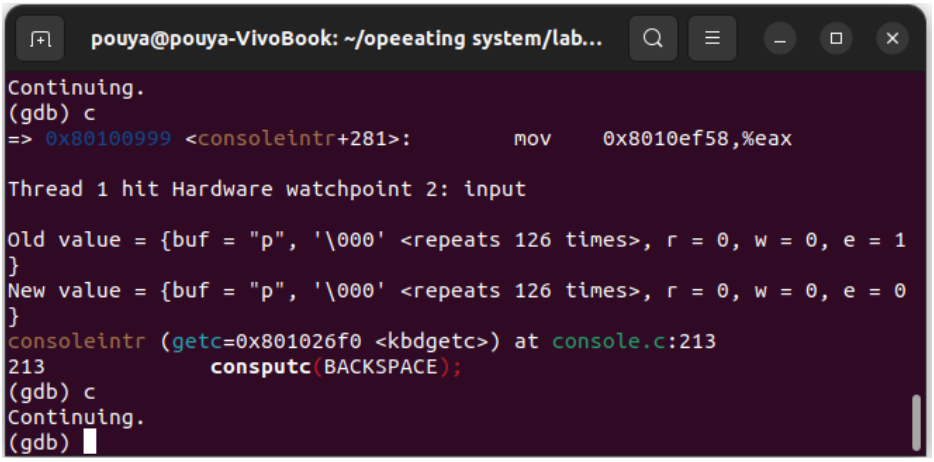
- توضیح کلی این struct و متغیرهای درونی آن و نقش آنها

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

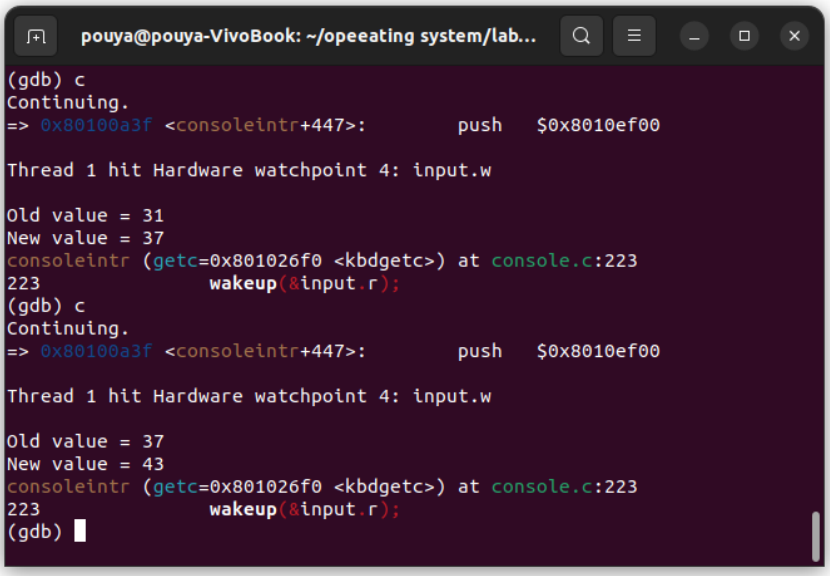
این استراکچر که در فایل console.c تعریف شده است، وظیفه دریافت و نگهداری ورودی و احتمالا وظایف مرتبط با آنرا برعهده دارد. این ساختار حاوی متغیر buf از نوع آرایه ای از کاراکترها را شامل میشود که وظیفه نگهداری از ورودی را دارد، همچنین سه متغیر r, w, e که قرار است کارکرد آنها را متوجه شویم. برای اینکار، واچ پوینتمان را برروی Input قرار میدهیم. حال منسول را باز کرده و حرف p را وارد میکنیم:

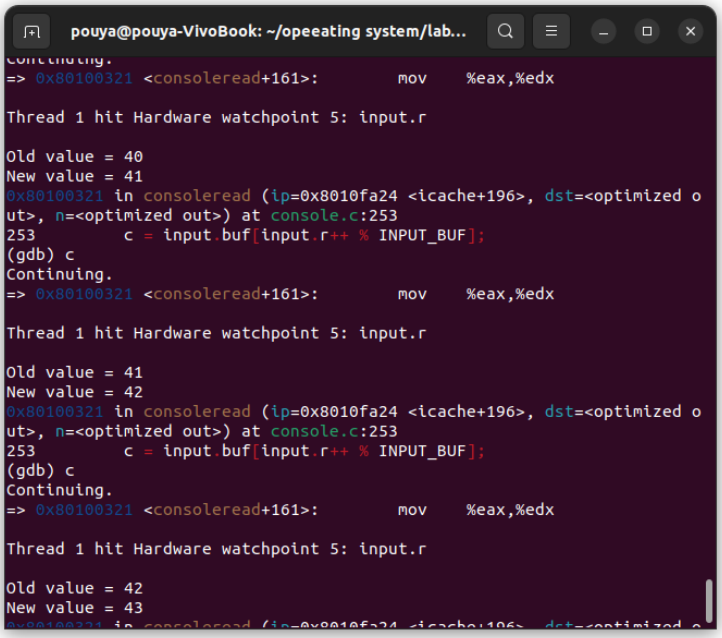


همانگونه که مشاهده میشود،متغیر e دوباره افزایش مقدار یافته، یکبار برای $ و بار دوم برای p. این متغیر مقدار نشان میدهد که حرف ورودی بعدی، در کدام اندیس از آرایه buf باید نوشته شود؛ یعنی با وارد کردن یک کاراکتر جدید، ابتدا مقدار input->e یک واحد افزایش میابد، سپس کاراکتر ورودی در input->buf[input->e - 1] نوشته میشود (در حقیقت، input.buf[input.e++] = c). با کمی امتحان بیشتر، این فرض را تایید میکنیم. حرف p را حذف میکنیم تا کرسر یک واحد به عقب بازگردد:

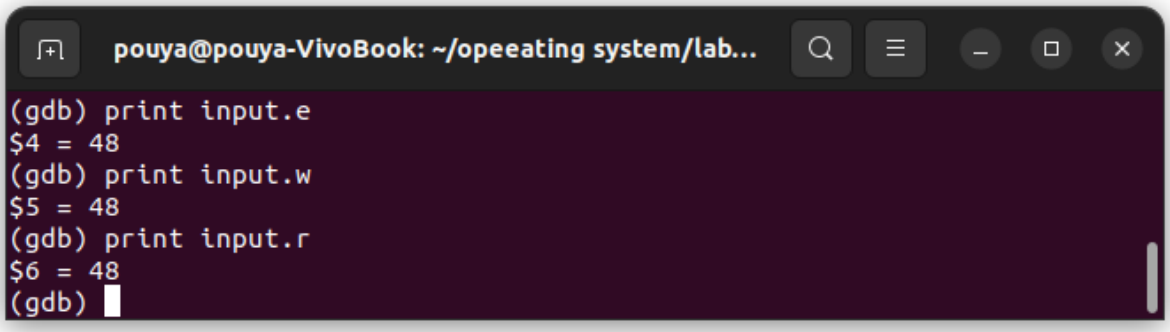


حال میخواهیم عملکرد w را بررسی کنیم. بنظر میآید ارتباطی با خواندن وجود دارد. با توجه به مشاهدات قبلی، میدانیم که به کاراکترهای نوشته شده در ترمینال در لحظه نوشتن مربوط نیس و فقط buf[], e از آن تاثیر میگیرند. متوجه میشویم که در دوحالت این واچ پوینت تاچ میشود؛ حال اول مربوط به زمان فشردن کلیدenter میباشد و حالت دوم هم زمانی که تعداد زیادی دکمه روی کیبورد زدیم(تعداد زیادی کاراکتر ورودی جدید) میباشد(حالت سومی هم به صورت فشردن ctrl + D وجود دارد). هم چنین از گزارش هایی که gdb به ما نشان میدهد(هم چنین افزودن واچ پوینت برای r)، متوجه میشویم که در این زمانها (که واچ پوینت w) تاچ شده، مقدار آن برابر با مقدار input.e قرار میگیرد و سپس متقیر input.r واحد به واحد از مقدار اولیه اش (که برابر با مقدار قبلی input.w میباشد) افزایش میبابد تا به مقدار جدید input.w برسد و هرسه متغیر مقدار برابر بگیرند:



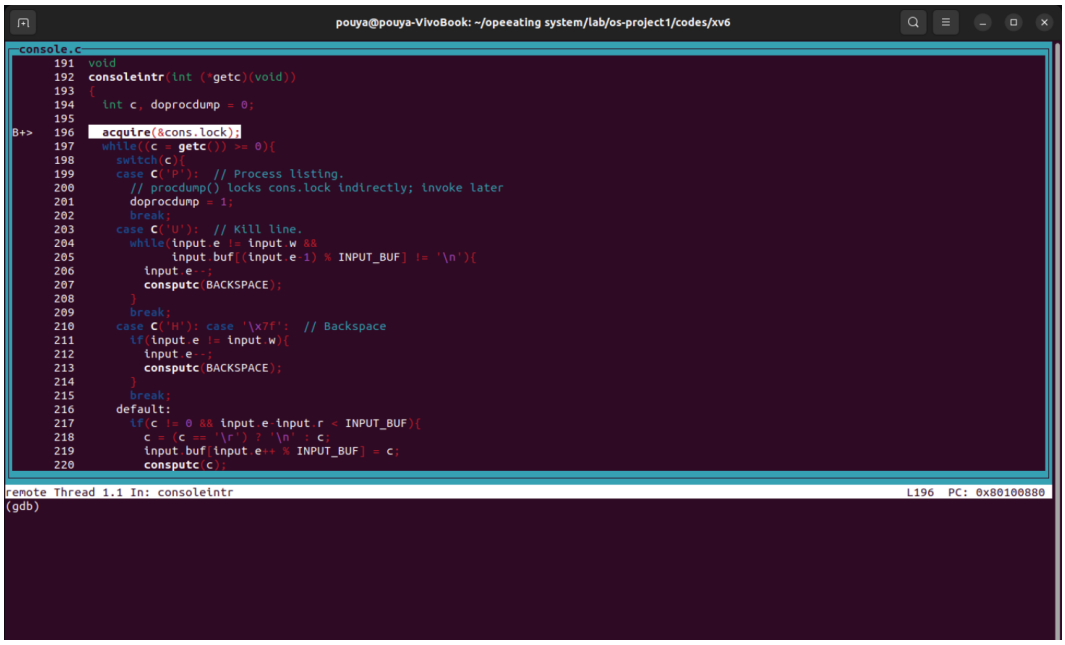


برای تایید اینکه درنهایت(در حالت پایا) هر سه متغیر یاد شده به مقدار یکسان میرسند، مقدار هر سه را مشاهده میکنیم:



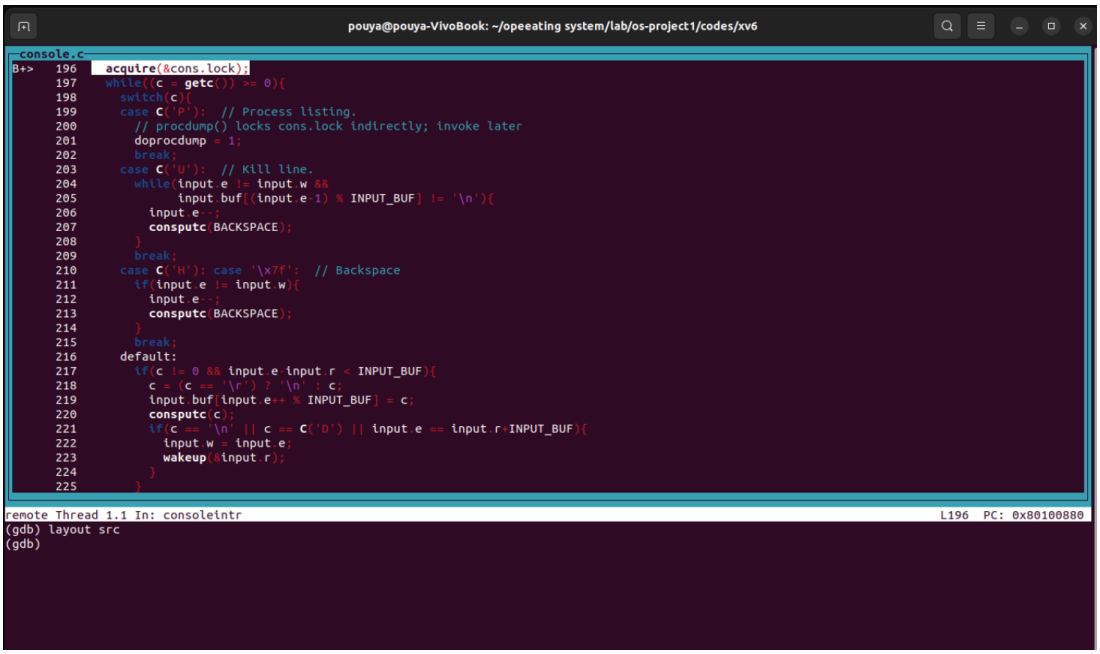
اشکال زدایی در سطح کد اسمبلی

در این بخش، یک breakpoint برای خط ۱۹۶ در فایل console.c قرار میدهیم. بعد از تاچ شدن بریک پوینت، با رفتن به محیط TUI، میتوان سورس کد محل تاچ را مشاهده کرد (با توجه به ضبط شدن فریم های طی شده تا محل تاچ، میتوان بین فریم ها نیز جابجا شد و سورس آنها را نیز مشاهده کرد:



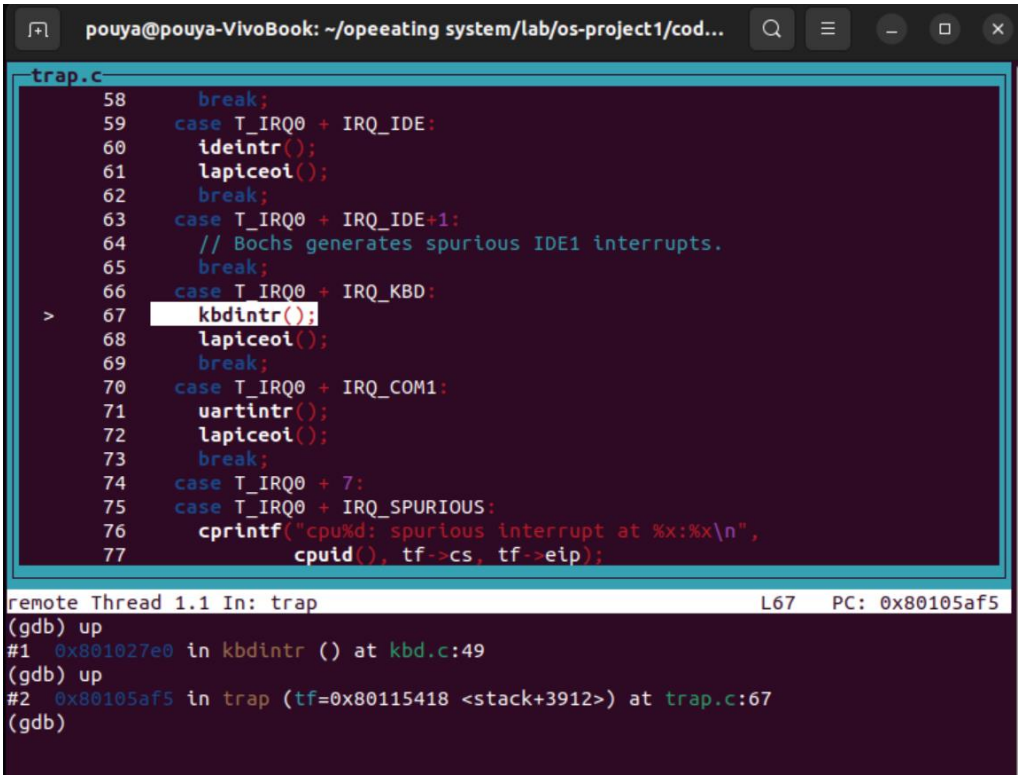
سوال ۷- خروجی دستورهای layout src, layout asm در TUI چیست؟

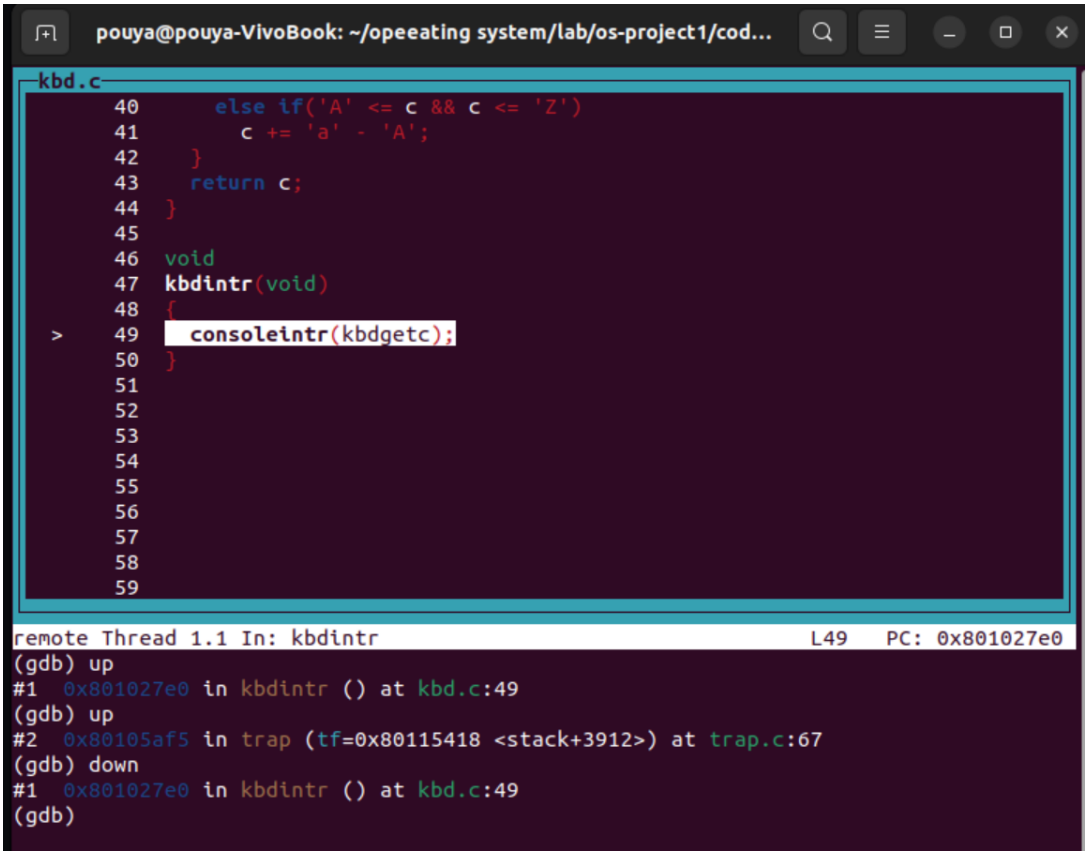
با وارد کردن فرمان layout src ، برنامه درحالت کد منبع آن (در اینجا، زبان C) مورد نمایش قرار میگیرد.



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

با دستور up، میتوانیم به فریم قبلی یا بیرونی برویم و با دستور down، امکان جابه جایی به فریم بعدی با داخلی را خواهیم داشت(در فریم هایی که دستور bt نشان میدهد، با up به سمت فریم با شماره بیشتر و down به فریم با شماره کمتر در صورت وجود، جابه جا میشویم).


pouya@pouya-VivoBook: ~/opeeating system/lab/os-project1/cod...
trap.c
58 break;
59 case T_IRQ0 + IRQ_IDE:
60 ideintr();
61 lapiceoi();
62 break;
63 case T_IRQ0 + IRQ_IDE+1:
64 // Bochs generates spurious IDE1 interrupts.
65 break;
66 case T_IRQ0 + IRQ_KBD:
> 67 kbdintr();
68 lapiceoi();
69 break;
70 case T_IRQ0 + IRQ_COM1:
71 uartintr();
72 lapiceoi();
73 break;
74 case T_IRQ0 + 7:
75 case T_IRQ0 + IRQ_SPURIOUS:
76 cprintf("cpu%d: spurious interrupt at %x:%x\n",
77 cpuid(), tf->cs, tf->eip);
remote Thread 1.1 In: trap L67 PC: 0x80105af5
(gdb) up
#1 0x801027e0 in kbdintr () at kbd.c:49
(gdb) up
#2 0x80105af5 in trap (tf=0x80115418 <stack+3912>) at trap.c:67
(gdb)


pouya@pouya-VivoBook: ~/opeeating system/lab/os-project1/cod...
kbd.c
40 else if('A' <= c && c <= 'Z')
41 c += 'a' - 'A';
42 }
43 return c;
44 }
45
46 void
47 kbdintr(void)
48 {
> 49 consoleintr(kbdgetc);
50 }
51
52
53
54
55
56
57
58
59
remote Thread 1.1 In: kbdintr L49 PC: 0x801027e0
(gdb) up
#1 0x801027e0 in kbdintr () at kbd.c:49
(gdb) up
#2 0x80105af5 in trap (tf=0x80115418 <stack+3912>) at trap.c:67
(gdb) down
#1 0x801027e0 in kbdintr () at kbd.c:49
(gdb)

اضافه کردن یک متن به boot message

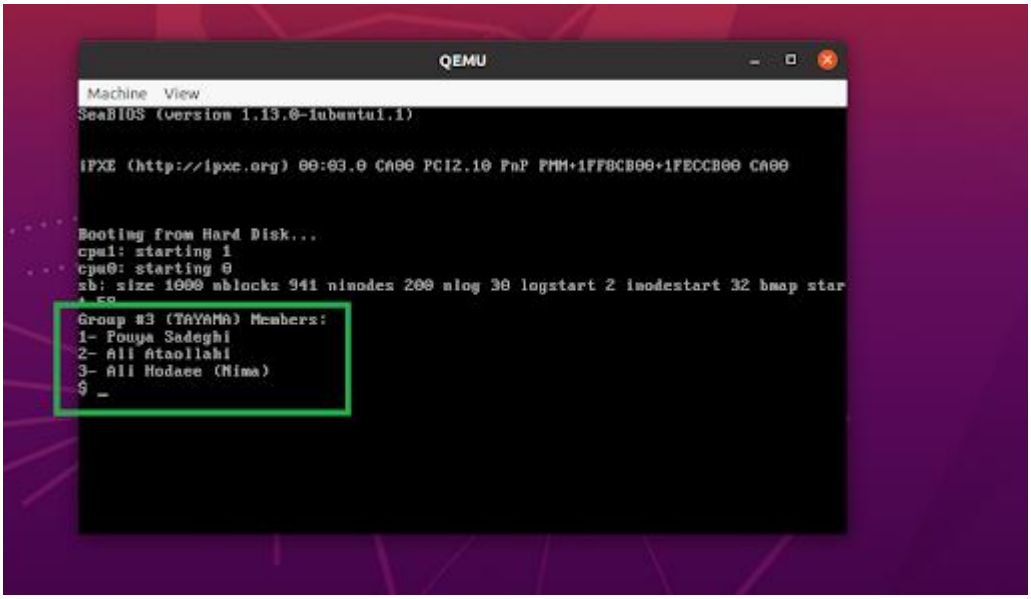
برای نوشتن نام اعضای گروه به فایل init.c مراجعه می کنیم و نام اعضای گروه را چاپ می کنیم:

```
int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "Group #3 (TAYAMA) Members:\n1- Pouya Sadeghi\n2- Ali Ataollahi\n3- Ali Hodaee (Nima)\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

در ادامه برای مشاهده نتیجه با make qemu دوباره عملیات شبیه سازی را انجام می دهیم:



اضافه کردن چند قابلیت به کنسول xv6

برای نوشتن دستورات به فایل console.c مراجعه می کنیم.
ابتدا برای سادگی کد نویسی مقداری ریفکتور و تغییر روی کد اصلی انجام می دهیم تا فرایند کد نویسی تسریع گردد.
ابتدا دو تابع زیر برای تغییر مکان cursor به کد اضافه شد.

```
int get_pos() {
  int pos;

  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  return pos;
}

static void change_pos(int pos) {
  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
}
```

انتهای تابع زیر کمی ریفکتور شد.

```
static void cgaputc(int c)
{
  int pos;

  // Cursor position: col + 80*row.
  pos = get_pos();

  if(c == '\n')
    pos += 80 - pos%80;
  else if(c == BACKSPACE){
    if(pos > 0) --pos;
  } else
    crt[pos++] = (c&0xff) | 0x0700;  // black on white

  if(pos < 0 || pos > 25*80)
    panic("pos under/overflow");

  if((pos/80) >= 24){  // Scroll up.
    memmove(crt, crt+80, sizeof(crt[0])*23*80);
    pos -= 80;
    memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
  }

  change_pos(pos);
  if(c == BACKSPACE)
    crt[pos] = ' ' | 0x0700;
}
```

برای آنکه خوانایی کد بیشتر شود و قابلیت های struct input بیشتر شود یک عدد end به آن اضافه شد.

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

تابع شیفت به سمت چپ و شیفت به سمت راست اضافه شد

```
void shift_input_right() {
  int index, next_char, pos;
  pos = get_pos();
  change_pos(pos + 1);
  index = input.e;
  next_char = input.buf[index % INPUT_BUF];
  while(index < input.end) {
    int temp = next_char;
    next_char = input.buf[(index + 1) % INPUT_BUF];
    input.buf[(index + 1) % INPUT_BUF] = temp;
    consputc(input.buf[(index + 1) % INPUT_BUF]);
    index++;
  }
  input.end++;
  change_pos(pos);
}
```

```
void shift_input_left() {
  int index, pos;
  pos = get_pos();
  index = input.e - 1;
  while(index < input.end) {
    input.buf[index % INPUT_BUF] = input.buf[(index + 1) % INPUT_BUF];
    consputc(input.buf[index % INPUT_BUF]);
    index++;
  }
  consputc(' ');
  input.end--;
  change_pos(pos);
}
```

همچنین کنترلر Ctrl+U به صورت زیر ریفکتور شد

```
case C('U'):
  kill_line();
  break;
```

```
void kill_line(){
  while(input.e != input.w && input.buf[(input.e-1) % INPUT_BUF] != '\n') {
    if(input.e != input.w){
      consputc(BACKSPACE);
      shift_input_left();
      input.e--;
    }
  }
}
```

و back_space به صورت زیر:

```
  case C('H'): case '\x7f':
    back_space();
    break;
```

```
void back_space() {
    if(input.e != input.w){
    consputc(BACKSPACE);
    shift_input_left();
    input.e--;
  }
}
```

با توجه به اضافه شدن end به input باید دیفالت سوئیچ نوشته شده در consoleintr دچار تغییر شود. برای اینکار این قسمت را به بدین شکل در می اوریم:

```
default:
  if(c != 0 && input.e-input.r < INPUT_BUF){
    c = (c == '\r') ? '\n' : c;
    if(c == '\n' || c == C('D'))
      input.buf[input.end++ % INPUT_BUF] = c;
    else {
      shift_input_right();
      input.buf[input.e++ % INPUT_BUF] = c;
    }

    consputc(c);

    if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
      char* key;
      key = input.buf + input.w;
      input.e = input.end;
      input.w = input.e;
      wakeup(&input.r);
    }
  }
  break;
}
```

۱) Cntrl+n :

مربوط به پاک کردن اعداد در خط:

```
case C('N'):{
  delete_num_of_line();
  break;
}
```

```
void delete_num_of_line() {
  go_to_first_of_line();
  while (input.e < input.end){
    int pos = get_pos();
    if(is_num(input.buf[(input.e) % INPUT_BUF])){
      int pos=get_pos();
      change_pos(++pos);
      input.e++;
      consputc(BACKSPACE);
      shift_input_left();
      input.e--;
    }
    else{
      input.buf[(input.e) % INPUT_BUF] = input.buf[(input.e) % INPUT_BUF];
      consputc(input.buf[(input.e) % INPUT_BUF]);
      change_pos(++pos);
      input.e++;
    }
  }
}
```

```
void go_to_first_of_line() {
  int pos;
  pos = get_pos();
  int change;
  change = pos%80 - 2;
  input.e -= change;
  change_pos(pos - change);
}
```

```
int is_num(int c) {
  return (c >= '0' && c<='9') ? 1 : 0;
}
```

۲) Cntrl+r:

مربوط به ریورس کردن:

```
case C('R'): {
  reverse_line();
  break;
}
```

```
void reverse_line() {
  char* key = input.buf + input.w;
  print_word_reverse(key);
}
```

```
void print_word_reverse(char* key) {
  char temp[INPUT_BUF];
  strncpy(temp,key,INPUT_BUF);
  kill_line();
  for (int i = strlen(temp)-1; i >= 0; i--) {
    input.buf[input.e % INPUT_BUF] = temp[i];
    consputc(input.buf[input.e % INPUT_BUF]);
    input.e++;
    input.end++;
  }
}
```

۳) Tab

برای آنکه تاریخچه دستورات زده شده ذخیره شود یک فایل به نام prefix_predict.c اضافه شد که دو تابع update_history و starts_withدر آن قرار دارد.

```
void update_history(char* inp,int size_command) {

    stringcpy(command[command_num % MAX_COMMAND_NUM ], inp , size_command);
    command_num++;
    if(size_command < MAX_COMMAND_NUM) size_command++;

}
int starts_with(char *pre, char *str)
{
  if(strncmp(str, pre, strlen(pre)) == 0) return 1;
  return 0;
}
```

همچنین با توجه به اینکه به فایل user.h در این فایل نیاز بود و strcpy احتیاج بود که strcpy در defs.h قرار داشت که با user.h همزمان نمی توانستیم آن را اینکلود کنیم پس تابع stringcpy به این فایل اضافه شد.

```
static char* stringcpy(char *s, const char *t,int size)
{
  char *os;

  os = s;
  while(((*s++ = *t++) != 0))
    ;
  return os;
}
```

همچنین باید هر دستور در هیستوری ذخیره شود که دیفالت در consoleintr را بدین صورت تغییر می دهیم

```
default:
  if(c != 0 && input.e-input.r < INPUT_BUF){
    c = (c == '\r') ? '\n' : c;
    if(c == '\n' || c == C('D'))
      input.buf[input.end++ % INPUT_BUF] = c;
    else {
      shift_input_right();
      input.buf[input.e++ % INPUT_BUF] = c;
    }

    consputc(c);

    if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
      char* key;
      key = input.buf + input.w;
      updatehistory(key, input.e - input.w);
      sizeCommand++;
      command_num++;
      input.e = input.end;
      input.w = input.e;
      wakeup(&input.r);
    }
  }
  break;
```

حال سراغ خود کد اصلی در کنسول میرویم:

```
    case '\t': {
      predict_process();
      break;
    }
```

```c
#define MAX_COMMAND_NUM 15
extern int command_num ,size_command ;
extern char command[MAX_COMMAND_NUM][128];
```

```c
void predict_process() {
    int index = predict_in_command_list();
    if(index!=-1)
        print_word(command[index]);
}
```

```c
int predict_in_command_list() {
    int index = -1;
    char* key = input.buf + input.w;
    if(size_command<MAX_COMMAND_NUM)
        for (int i = size_command; i >= 0; i--)
            if(starts_with(key,command[i]))
                index = i;
    else {
        int endIndex = ((command_num % MAX_COMMAND_NUM) + (MAX_COMMAND_NUM - 1))
        % MAX_COMMAND_NUM ;
        int i = ( command_num % MAX_COMMAND_NUM);
        while (i != endIndex)
        {
            if(starts_with(key,command[i])) {
                index = i;
            }
            i++;
            if(i == MAX_COMMAND_NUM) i = 0;
        }
        if(starts_with(key,command[i])) index = i;
    }
    return index;
}
```

```c
void print_word(char* key) {
    char temp[INPUT_BUF];
    strncpy(temp,key,INPUT_BUF);
    kill_line();
    for (int i = 0; i < strlen(temp)-1; i++){
        input.buf[input.e % INPUT_BUF] = temp[i];
        consputc(input.buf[input.e % INPUT_BUF]);
        input.e++;
        input.end++;
    }
}
```

اجرا و پیاده سازی یک برنامه سطح کاربر

در ابتدا، باید کد برنامه را بزنیم. الگوریتم که مشخص است و ارتباطی به سیستم عامل ندارد. از آنجا که در اینجا کد عادی C نمیزنیم و کد را برای سیستم عامل میزنیم، باید نکاتی را درنظر بگیریم. اولین نکته این است که ممکن است خیلی از توابع استاندار زبان در اینجا موجود نباشند و یا بعضاً header فایلی با نام مشابه وجود داشته باشد(هردو اتفاق اینجا افتاده)؛ پس باید برنامه را با توجه به این توابع توسعه بدیم. در ابتدا توابع استفاده شده را شرح میدهیم.

توجه شود که تابع atoi بظور پیشفرض وجود دارد و برای ما قابل استفاده است، اما یک تابع با قابلیت مشابه توسعه دادیم که بتوانیم ارور های برنامه را بهتر کنترل کنیم(یک عدد منفی در صورت ایجاد مشکل برگردانیم و مشکلات مورد نظرمان را همراه با تبدیل به عدد چک کنیم). این ارور ها، شامل ارور در صورت وارد کردن عدد منفی یا عدد با فرمت غلط میباشد که برنامه بدرستی کاربر را از ایراد با خبر میسازد.

تابع بعدی، printf میباشد که عملکرد آن مشابه با تابع شاخته شده در c میباشد، با این تفاوت که یک آرگومان دیگر به عنوان شناسه محلی که باید در آن بنویسید را در ابتدا دریافت میکند. از ابتدا آدرس ۱ به خروجی استاندار و 2 مربوط به ارور استاندارد میباشد(توجه شود که شناسه یاد شده، مربوط به اندیس پردازه میباشد که دستور متوجه شود در کدام آدرس باید بنویسد). هم چنین این شناسه، نیابد لزومه به خروجی های ترمینال منتهی شود و میتواند مربوط به یک فایل باشد و در فایل عملیات نوشتن را انجام دهد. موضوع قابل توجه، این است که این تابع، در مراحل پایین تر، دارد فراخوانی سیستمی write را صدا میزند و فقط رابط بهتری را در اختیارمان قرار میدهد.

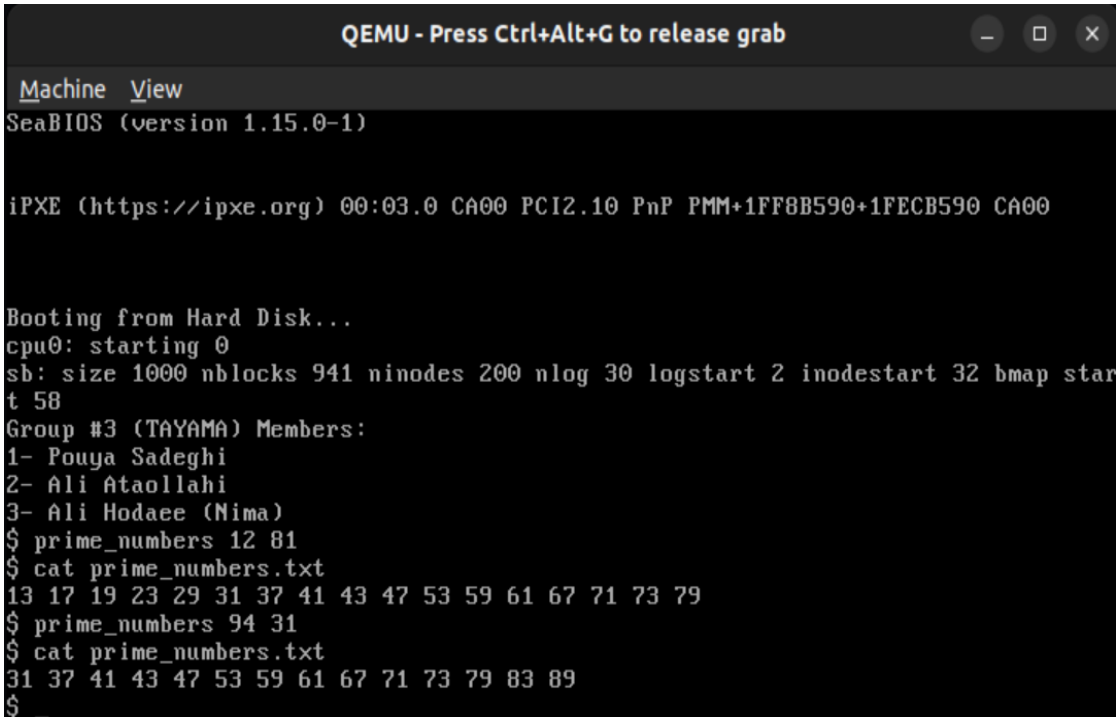
مابقی توابع نیز بصورت عادی میباشد و همانند چیزی که قبلا دیده ایم، عمل میکنند.

حال که برنامه مورد نظر را نوشتیم، برنامه را درکنار سایر کدمنبع های سیستم عامل قرار میدهیم. نکته قابل توجه نام فایل است که بعدا برنامه به همین نام پردازش شده و در آینده نیز کاربر باید برنامه را به همین نام صدا کند.

قبل از کامپایل و اجرای سیستم عامل، برای اینکه برنامه ما در سطح کاربر در سیستم عامل در دسترس قرار بگیرد، باید تغییراتی را در Makefile سیستم عامل ایجاد کنیم. باید به EXTRA به فرمت <source_name>.c و در UROGS به فرمت <source_name>_ اضافه کنیم که برنامه در دسترس کاربر قرار بگیرد.

UROGS برنامه را در فایل سیستم قرار میدهد (که بعدا به qemu داده میشود) تا برنامه لود شده و در دسترس قرار بگیرد و یوزر بتواند آن دستورها را اجرا کند. EXTRA نیز موجب کپی شدن سورس به دایرکتوری dist میشود و مقدمات اماده سازی و افزودن برنامه را فراهم میکند.

و تمام. فقط کافی است که به شمل عادی سیستم عامل را کامپایل کرده و در qemu اجرا کنیم. برنامه ما اماده اجرا شدن میباشد.



گزارش کار افزودن ماژول به هسته لینوکس

در اولین مرحله برای ساختن یک ماژول/درایو برای لینوکس، باید کد مورد نظر رو بنویسیم. باید توجه داشت که این کد با کدهای عادی که به زبان C نوشته میشوند، تفاوت دارد و این تفاوت از جهت کتابخانه ها و ساختاری میباشد. در وهله اول، باید کتابخانه های مربوط به لینوکس را اینکلود کنیم(خط های ۱-۳). همچنین شایان ذکر است که در این کد، تعدادی کال میبینیم که خارج از توابع انجام شده اند(خطوط ۵-۷ و ۱۸ و ۱۹) که مهم ترین انها، خط پنج است که لایسنس و استاندارد مارا مشخص میکند. همچنین باید مشخص کنیم که هنگام لود کردن درایو یا ریموو کردن آن، چه تابعی باید اجرا شود(خطوط ۱۸ و ۱۹).

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Group#3 members");
MODULE_DESCRIPTION("this module print group#3's members name");

static int __init print_members_name(void) {
    printk(KERN_INFO "Pouya Sadeghi, Ali Ataollahi, Ali Hodaee\n");
    return 0;
}

static void __exit outro(void) {
    printk(KERN_INFO "we are group#3\n");
}

module_init(print_members_name);
module_exit(outro);
```

حال، باید این کد را کامپایل کنیم. برای این منظور از Makefile استفاده میکنیم و آنرا با محتوای زیر ایجاد میکنیم. حال با اجرای دستور make در ترمینال، برنامه ما کامپایل میشود. از بین فایل های ایجاد شده، فایلی به نام driver.ko برای ما مهم ترین فایل میباشد که ماژول هسته کامپایل شده ما میباشد (از آنجا که نام فایل ما driver.c بود، این نام به فایل مذکور نسبت داده شد).

```
obj-m += driver.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

در ادامه برای اضافه کردن این ماژول به کرنل، دستور sudo insmod driver.ko را اجرا میکنیم. برای اینکه چک کنیم آیا ماژول به درستی در کرنل لود شده است، از دستور lsmod استفاده میکنیم:

```
pouya@pouya-VivoBook:~/linux_kernel_mod$ lsmod
Module                  Size  Used by
driver                 16384  0
ccm                    20480  6
rfcomm                 81920  4
cmac                   16384  2
algif_hash             16384  1
algif_skcipher         16384  1
af_alg                 32768  6 algif_hash,algif_skcipher
snd_hda_codec_hdmi     77824  1
bnep                   28672  2
snd_hda_codec_realtek   159744  1
snd_hda_codec_generic   102400  1 snd_hda_codec_realtek
ledtrig_audio          16384  1 snd_hda_codec_generic
joydev                 32768  0
intel_tcc_cooling      16384  0
x86_pkg_temp_thermal    20480  0
intel_powerclamp       20480  0
snd_soc_skl           172032  0
snd_soc_hdac_hda       24576  1 snd_soc_skl
snd_hda_ext_core       32768  2 snd_soc_hdac_hda,snd_soc_skl
snd_soc_sst_ipc        20480  1 snd_soc_skl
```

ماژول لود شده در کرنل

با مشاهده نام ماژول در لیست ماژول های لود شده در کرنل، متوجه میشویم که برنامه ب درستی رد کرنل لود شده است.

برای حذف ماژول از کرنل نیز، دستور sudo rmmod driver را اجرا میکنیم.

حال برای چک کردن خروجی، از دستور sudo dmesg استفاده میکنیم و میبینیم که خروجی ها بدرستی چاپ شده اند و ماژول عملکرد مورد انتظار را داشته و بدستی به کرنل اضافه و از آن حذف شده است:

```
[  764.336907] audit: type=1420 audit(1665405870.947:103): subj_apparmor=unconfined
[  764.635695] audit: type=1400 audit(1665405871.243:104): apparmor="DENIED" operation="open" profile="snap.s
sted_mask="r" denied_mask="r" fsuid=1000 ouid=0
[  839.250000] kauditd_printk_skb: 1 callbacks suppressed
[  839.250002] audit: type=1326 audit(1665405945.865:106): auid=1000 uid=1000 gid=1000 ses=3 subj=? pid=2901
ll=93 compat=0 ip=0x7fe01749239b code=0x50000
[ 1116.527460] nouveau 0000:01:00.0: Enabling HDA controller
[ 1116.674915] nouveau 0000:01:00.0: bus: MMIO read of 00000000 FAULT at 6013d4 [ PRIVRING ]
[ 1125.050926] nouveau 0000:01:00.0: Enabling HDA controller
[ 1192.972190] driver: loading out-of-tree module taints kernel.
[ 1192.972234] driver: module verification failed: signature and/or required key missing - tainting kernel
[ 1192.972493] Pouya Sadeghi, Ali Ataollahi, Ali Hodaee
[ 1204.082108] we are group#3
```

گزارش مربوط به لود و مقدار دهی اولیه شدن ماژول

گزارش ایجاد شده در زمان حذف ماژول از هسته