**آزمایشگاه سیستم عامل**

**پروژه یک**

**اعضای گروه:**

الهه خداوردی - 810100132

فرشته باقری - 810100089

عاطفه میرزاخانی - 810100220

**Repository:** <https://github.com/elahekhodaverdi/Operating-System-Lab-Projects>

**Latest Commit:** 7931e3db341b8c1a811bbe2b484e1fd515528520

**آشنایی با سیستم عامل xv6**

1. معماری سیستم عامل **xv**6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

سیستم عامل xv6 یک سیستم عامل آموزشی Unix-Based است که طبق داکیومنت xv6 ، یک re-implementation از Unix V6 است و این سیستم عامل مبتنی بر پردازنده های x86 نوشته شده است. در فایل x86.h نیز از دستورات پردازنده x86 استفاده شده است.

سیستم عامل unix از سه بخش اصلی kernel, shell و user applications تشکیل شده است که xv6 هم همین روال را دارد و به طور کلی در اجرای پردازش ها از روش unix تبعیت می کند.

That operating system, xv6, provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie’s Unix operating system, as well as mimicking Unix’s internal design.

Xv6 runs on Intel 80386 or later (‘‘x86’’) processors on a PC platform, and much of its low-level functionality (for example, its process implementation) is x86-specific.

معماری هسته xv6 به صورت یکپارچه (monolithic) است یعنی کل سیستم عامل در حالت سوپروایزر اجرا می شود.

One possibility is that the entire operating system resides in the kernel, so that the implementations of all system calls run in kernel mode. This organization is called a monolithic kernel. In this organization the entire operating system runs with full hardware privilege.

1. یک پردازه در سیستم عامل **xv**6 از چه بخش هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازه های مختلف اختصاص می‌دهد؟

هر پردازه از دو بخش User-space memory و Per-process state private to the kernel تشکیل شده است. بخش User-space memory شامل دستورات(instructions)، اطلاعات(data) و پشته(stack) است. دستورات، کد برنامه ای است که پردازه در حال اجرا است و اطلاعات شامل متغیر ها، ثابت ها و بقیه اطلاعاتی است که توسط برنامه استفاده می‌شود و پشته قسمتی از حافظه است که برای فراخوانی ها و کنترل متغیر های محلی استفاده می‌شود. این بخش از پردازه مختص پردازه بوده و توسط بقیه پردازه ها قابل دسترسی نیست. Per-process state private to the kernel شامل اطلاعات و ساختارهای داده‌ای است که توسط هسته نگهداری می‌شوند تا اجرای یک فرایند را مدیریت و کنترل کنند. این شامل داده‌هایی مانند شناسه فرآیند (PID) است که هر فرآیند را به صورت یکتا شناسایی می‌کند، همچنین شامل سایر اطلاعات مربوط به فرآیند مانند وضعیت اجرای فعلی (اجرا، انتظار و غیره)، اولویت فرآیند، شماره‌گذاری فایل و سایر ساختارهای داده‌ای مرتبط با هسته است. این وضعیت فرآیند به صورت خاص توسط هسته مدیریت و دسترسی دارد و به صورت مستقیم توسط خود فرآیند قابل دسترسی یا تغییر نیست.

این سیستم عامل به طور کلی پردازنده را به طریق time-share به پردازه های مختلف اختصاص می‌دهد. پردازنده بین پردازه های قابل اجرا(که منتظر اجرا شدن هستند) جا به جا می‌شود

1. مفهوم **file descriptor** در سیستم عامل مبتنی بر **UNIX** چیست؟ عملکرد **pipe** در سیستم عامل **xv**6 چگونه است و به طور معمول برای چه هدفی استفاده می‌شود؟

هر پردازه یک آرایه خصوصی به نام ofile دارد که در آن اشاره گر ها به فایل هایی که باز کرده است وجود دارند. هنگامی که یک پردازه فایلی را باز می‌کند index ای از ofile که پوینتر به آن فایل در آن ذخیره شده باز گردانده می شود و پردازه با استفاده از این عدد می تواند در فایل بنویسید یا از آن بخواند.

هنگامی که یک پردازش دستور fopen را می دهد، kernal کوچک‌ترین fd که UNUSED است را به فایل مورد نظر اختصاص داده و آن را برمیگرداند. هنگامی که یوزر fclose را اجرا می کند سیستم fd مورد نظر را برای استفاده مجدد آزاد می کند.

Fd ها می توانند به فایل های عادی، پایپ یا device file اشاره کنند. در این حالت یوزر فارغ از اینکه تایپ فایلش چیست صرفا با یک fd کار می کند و به نوعی مانند یک interface عمل می کند.

در هر پروسه، file descriptor های اول تا سوم به ترتیب مربوط به فایل های زیر هستند:

Standard input

Standard output

Standard error

پایپ ها یک مکانیزم برای ارتباط بین پردازه های مختلف (inter-process communication) است که اجازه می دهد دو پردازه با تبادل داده ها با هم ارتباط بگیرند. با استفاده از پایپ ها می توان output یک پردازه را به input دیگری وصل کنیم.

پایپ ها توسط pipe() sysCall ایجاد می شوند و دو fd یکی برای read end و دیگری برای write end بر می گرداند.

عملکرد پایپ ها باعث Synchronization اجرای پردازه ها نیز می شود.

برای مثال وقتی پردازه پدر تابع fork را صدا می زند، یک child process ایجاد می شود که این دو میتوانند از طریق پایپ با هم ارتباط بگیرند. هر پردازه سری که نمیخواد استفاده کند را می بندد تا در هر زمان فقط read end یا write end برای آن پردازه باز باشد. سپس پدر می تواند چیزی در پایپ بنویسد و فرزند از read end آن را بخواند.

لازم به ذکر است که عملکرد های پایپ blocking هستند یعنی اگر پردازه ای بخواهد از پایپ خالی چیزی بخواند تا زمانی که دیتایی وجود نداشته باشد پردازه بلاک خواهد شد. برای نوشتن در پایپ پر هم همینطور است.

1. فراخوان های سیستمی **exec** و **fork** چه عملی انجام می‌دهند؟ از نظر طراحی ادغام نکردن این دو چه مزیتی دارد؟

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

پس اگر با استفاده از قطعه کد int pid = fork(); یک پردازه جدید درست کنیم، یکی از حالت‌های زیر برای pid رخ می‌دهد:

* pid = 0: در پردازه فرزند هستیم.
* pid > 0: در پردازه پدر هستیم و مقدار pid درواقع شناسه پردازه فرزند است.
* pid < 0: در زمان اجرای تابع fork و پردازه جدید خطایی رخ داده و پردازه فرزند ایجاد نشده‌است.

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

int pid = fork();  
if (pid == 0) {  
 printf("This is child process.\n");  
 printf("Child process is exiting...\n");  
 exit(0);  
}  
else if (pid > 0) {  
 printf("This is parent process, child's PID = %d.\n", pid);  
 printf("Waiting for child process to exit...\n");  
 wait();  
 printf("Child process exited.\n");  
}  
else {  
 printf("Fork failed!\n");  
}

در حقیقت اتفاقی که در سیستم‌عامل می‌افتد تا حدی در تکه کد بالا توضیح داده شده است.

تابع exec حافظه پردازه فعلی را با یک حافظه جدید که در آن یک برنامه با فایل ELF لود شده است، جایگزین می‌کند، اما file table اولیه را هم حفظ می‌کند. درواقع exec() راهی برای اجرای یک برنامه در پردازه فعلی است. برخلاف تابع fork()، برنامه به caller تابع exec() بازنمی‌گردد و برنامه جدید اجرا می‌شود، مگر اینکه در زمان اجرای این تابع خطایی رخ دهد. برنامه جدید اجرا شده در یک نقطه‌ای با استفاده از تابع exit اجرای پردازه را خاتمه می‌دهد. تابع exec دو پارامتر ورودی دارد که پارامتر اول نام فایل برنامه و پارامتر دوم آرایه آرگومان‌های ورودی برنامه است.

قطعه کد زیر مثالی از اجرای این تابع را نشان می‌دهد:

char\* args[] = { "ls", "-l", "/home", NULL }; // NULL is required  
exec("/bin/ls", args);  
printf("Exec failed!\n");

در حقیقت ادغام نکردن این دو تابع از ساختن پردازه‌های بی‌مصرف و جایگزین شدن سریع آنها توسط exec جلوگیری می‌کند. در حالت عادی توابع fork و exec پشت سر هم اجرا می‌شوند. اگر این دو ادغام شوند، علاوه بر پردازه‌های اضافه و میزان حافظه زیادی که اشغال می‌شود، مدیریت آرگومان‌های توابع هم دشوار می‌شود. مزیت ادغام نکردن این دو تابع در زمان I/O redirection خودش را نشان می‌دهد. زمانی که کاربر در shell

یک برنامه را اجرا می‌کند، کاری که در پشت صحنه انجام می‌شود به شرح زیر است:

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

زمانی که کاربر برای یک دستور از redirection استفاده می‌کند، تغییرات لازم در file descriptorها پس از fork و پیش از exec و در پردازه فرزند انجام می‌شود.

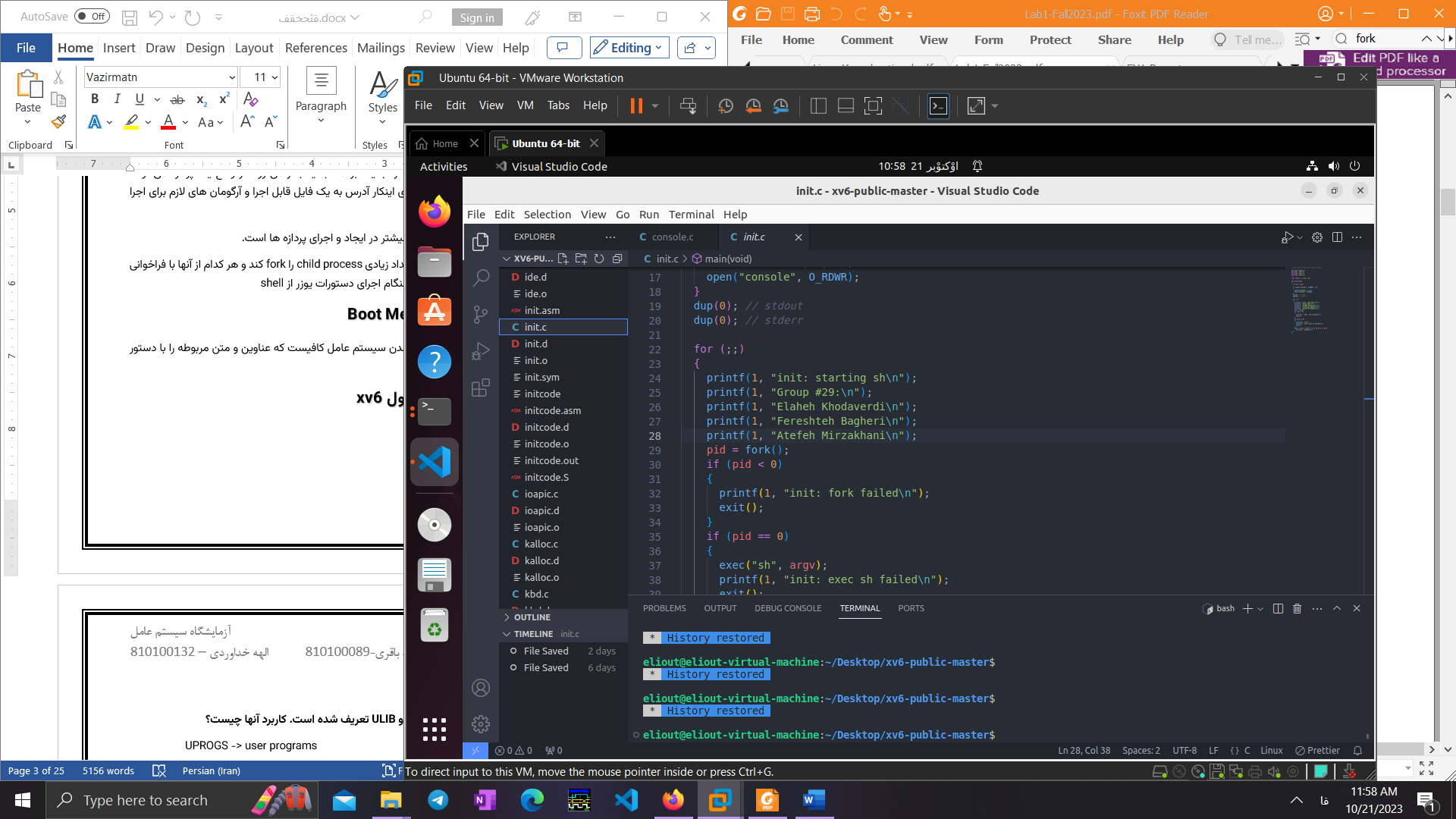
قطعه کد زیر این مورد را به شکل ساده نشان می‌دهد (فرض کنید دستور اجرا شده cat < in.txt است):

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

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

**اضافه کردن یک متن به Boot Message**

برای نشان دادن نام اعضای گروه پس از بوت شدن سیستم عامل کافیست که عناوین و متن مربوطه را با دستور printf به فایل init.c اضافه کنیم:



A screenshot of a computer

Description automatically generated

**اضافه کردن چند قابلیت به کنسول xv6**

**نکته: برای پیاده‌سازی دستور های داده شده در کد مربوط به سیستم عامل xv6 یک متغیر global به نام back\_count قرار داده شده است که نشان دهنده تعداد کاراکترهایی است که به عقب بازگشته ایم:**

1. **اضافه کردن دستور Ctrl+B**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. **اضافه کردن دستور Ctrl+F:**

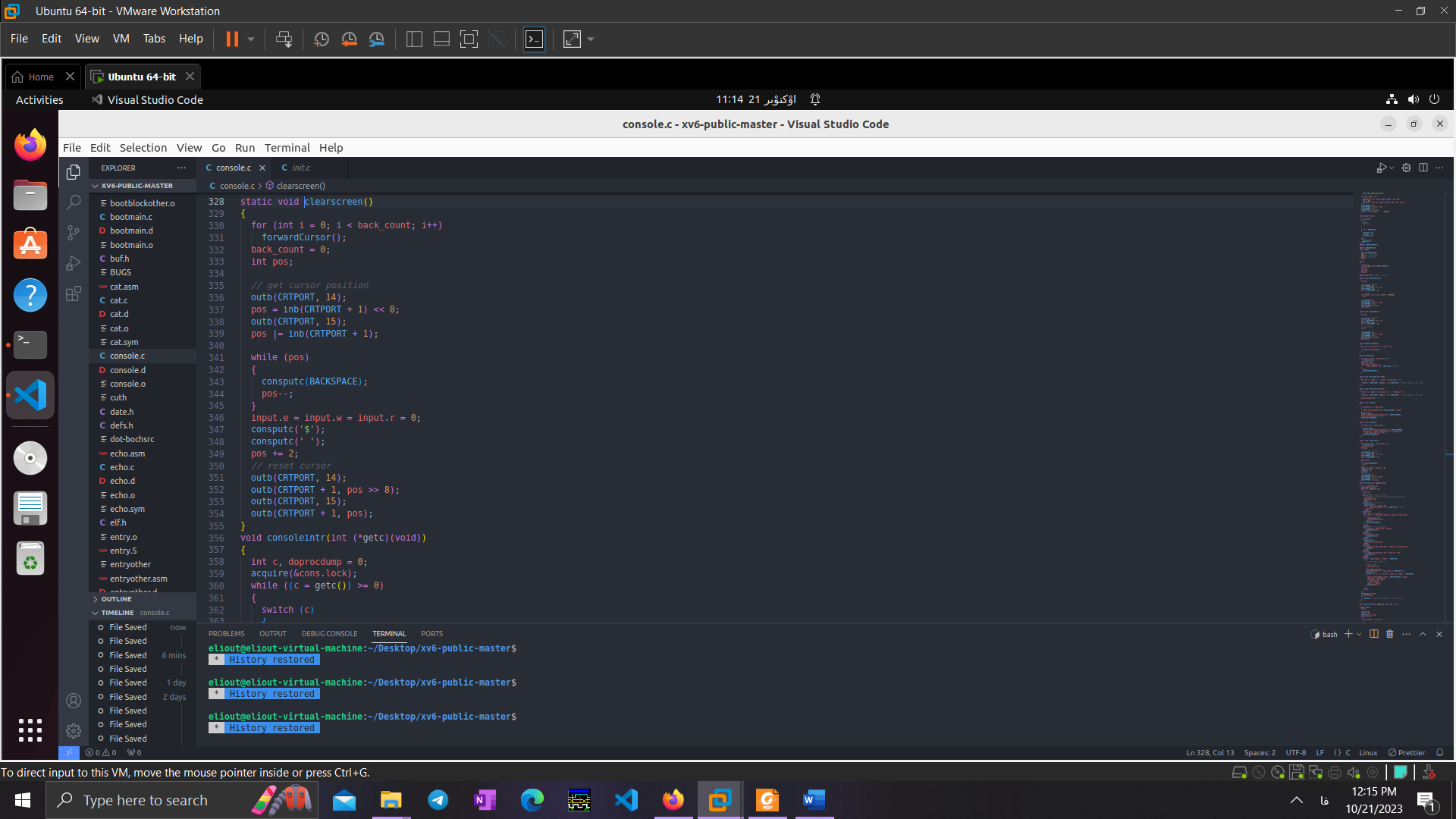
A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. **اضافه کردن دستور Ctrl+L :**



1. **اضافه کردن دستور arrow up:**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. **اضافه کردن دستور arrow down:**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

اجرا و پیاده سازی یک برنامه سطح کاربر

A screenshot of a computer

Description automatically generated

کامپایل سیستم عامل **xv6**

8. در **Makefile** متغیرهایی به نامهای **UPROGS** و **ULIB** تعریف شده است. کاربرد آنها چیست؟

UPROGS -> user programs

این متغیر شامل لیستی از برنامه های سطح کاربر است که در هنگام ساخت و کامپایل xv6 کامپایل شده و به برنامه قابل اجرا توسط OS تبدیل می شوند. این فایل ها عموما در user directory سیستم عامل xv6 قرار دارند

ULIB -> user libraries

این متغیر شامل تعدادی کتابخانه زبان c است که در اجرای برنامه ها (یا سیستم عامل) به آنها نیاز است. پس این کتابخانه ها باید ساخته و به برنامه کاربرد لینک شوند تا بتوان آنها را اجرا نمود. کتابخانه ها عموما از کد های قابل استفاده و توابعی که در اکثر برنامه ها استفاده می شوند تشکیل شده اند و در ulib directory سورس کد xv6 قرار دارند.

وقتی makefile اجرا می شود ابتدا کتابخانه های ulib کامپایل شده و به فایل آبجکت تبدیل می شوند. این آبجکت ها با برنامه های کاربر در uprogs لینک شده و تبدیل به فایل قابل اجرا می شوند که کاربرد میتواند در xv6 آن ها را اجرا کند.

**مراحل بوت سیستم عامل xv6**

11. برنامه های 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 6**xv** چیست؟ چرا از این نوع فایل دودویی استفاده شده است؟ این فایل را به زبان قابل فهم انسان اسمبلی تبدیل نمایید.(راهنمایی: از ابزار **objdump** استفاده کنید. باید بخشی از آن مشابه فایل **bootasm.S** باشد.)

در قسمتی از Makefile مشاهده می‌کنیم:A screen shot of a computer code

Description automatically generated  
دو فایل bootmain.c و bootasm.S که به ترتیب به زبان های c و assembly هستند کامپایل شده و فایل های bootmain.o و bootasm.o که دارای پسوند .o هستند ساخته می‌شوند و این فایل ها توسط دستور LD با هم لینک می‌شوند و فایل bootblock.o ساخته می‌شود.

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

بعد از ساخت bootblock.o، با استفاده از دستور OBJCOPY که در سوال بعد توضیح داده می‌شود فایل bootblock ساخته می‌شود که نوع آن باینری خام است. اما دیگر فایل های دودویی از فرمت ELF هستند.  
ELF برای تعریف فرمت فایل های اجرایی در xv6 استفاده می شود. هنگامی که یک برنامه C را در xv6 کامپایل کنیم، یک فایل اجرایی ELF تولید می‌کند که سیستم عامل می تواند بارگذاری و اجرا کند.  
فایل های ELF دارای یک هدر هستند که حاوی اطلاعات حیاتی در مورد فایل اجرایی است، مانند نقطه ورود (جایی که برنامه باید اجرا شود)، بخش های برنامه، و جزئیات مربوط به پیوند پویا.  
فایل های ELF داری section هایی نیز هستند.که می‌توان به text.(كه حاوی کد اصلی برنامه است)، .data(حاوی داده‌های اولیه و متغیرهای static و global) و .bss(حاوي متغیرهایی که مقداردهی اولیه ندارند) .rodata (حاوی داده‌هایread-only است که در طول اجرای برنامه نمی‌توان آنها را تغییر داد.) اشاره كرد.

بنابراین فايل bootblock از فرمت باینری خام است و ELF نيست و هدر و بخش هایی مانند .data و غیره را ندارد و بخش اصلی آن text. است.

از آنجا که فایل باینری خام فقط بخش text را دارد پس حجم آن کم می‌باشد (حداکثر 512 بایت) و به همین منظور از این نوع برای فایل بوت استفاده می‌شود زیرا اولین فایلی که برای بوت شدن اجرا می‌شود نباید حجم زیادی داشته باشد.

همچنین فایل ELF نمی‌تواند مستقیما توسط CPU اجرا شود زیرا هسته سیستم‌عامل آن را می‌شناسد پس تا زمانی که هسته اجرا نشده است فایل ELF نمی‌تواند اجرا شود و می‌بایست به فایل باینری خام تبدیل شود که توسط CPU شناخته شده است.

برای تبدیل bootblock به زبان اسمبلی از دستور زیر استفاده می‌کنیم:

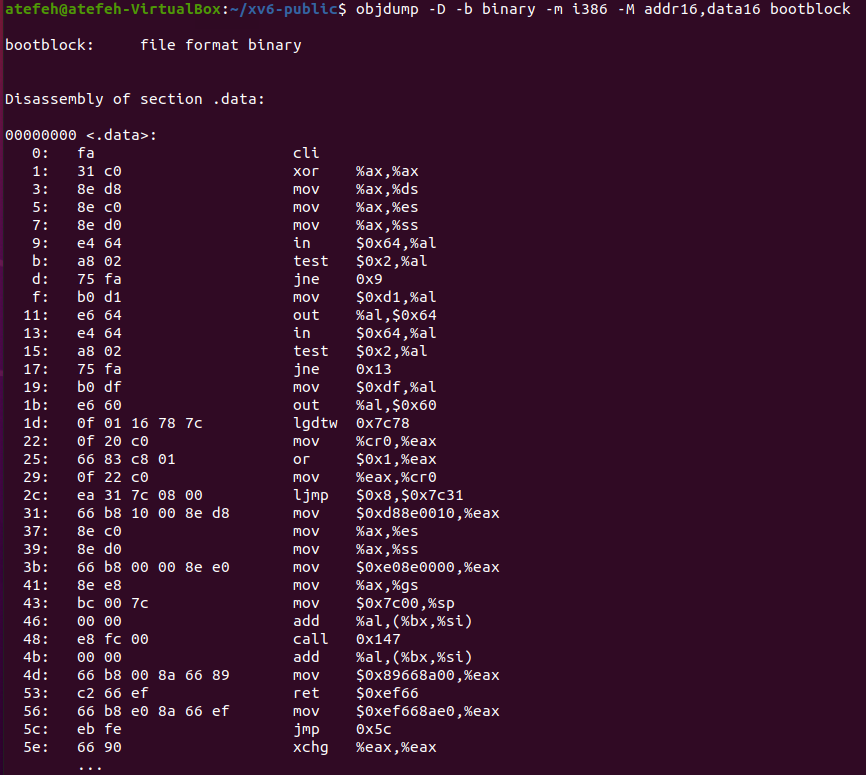
objdump -D -b binary -m i386 -M addr16,data16 bootblock

D-: برای disassemble کردن باینری بکار می‌رود.

b binary-: نوع فایل باینری را مشخص می‌کند که ما آن را باینری خام در نظر گرفتیم.

-m i386: معماری اسمبلی را مشخص می‌کند که آن را i386 گذاشتیم.

M addr16,data16-: برای تعیین option های disassembler استفاده می شود. در این مورد، disassembler باید کد را با استفاده از آدرس دهی 16 بیتی (addr16) و عملوندهای داده 16 بیتی (data16) در نظر بگیرد.(این قسمت برای شبیه شدن کد خروجی به bootasm.S اضافه شده است.)

11. علت استفاده از دستور **objcopy** در حین اجرای عملیات **make** چیست؟

در makefile سیستم عامل xv6 از دستور objcopy برای کپی کردن یک فایل آبجکت در آبجکت دیگر یا تبدیل فایل های باینری کامپایل شده به فایل باینری خام استفاده کرد. در طول فرایند make سورس کد xv6 کامپایل می شود و نتیجه آن یک فایل آبجکت برای هر سورس است. سپس این فایل ها با هم لینک می شوند و یک فایل دودویی قابل اجرا با فرمت ELF ایجاد می شود.

بعد از این مرحله از دستور objcopy برای تبدیل فایل ELF به فایل دودویی خام استفاده می شود که این فایل یک binary image برای سیستم عامل ایجاد می کند که این تصویر در طی فرآیند بوت بر روی حافظه لود می شود و توسط سخت افزار اجرا می شود.

با استفاده از این دستور فایل Makefile اطمینان حاصل می‌کند که کد کامپایل شده xv6 به یک تصویر دودویی تبدیل شده است که می‌توان آن را مستقیماً بارگذاری و اجرا کرد. این امر فرآیند بوت را ساده می‌کند و به سیستم عامل امکان اجرا بهینه روی سخت‌افزار هدف را می‌دهد.

13. بوت سیستم توسط فایلهای **bootasb.s** و **bootmain.c** صورت میگیرد. چرا تنها از کد C استفاده نشده است؟

در سیستم عامل xv6، فرآیند بوت سیستم شامل دو فایل اصلی است. دلیل استفاده از هر دو فایل C و فایل اسمبلی برای مدیریت جنبه‌های مختلف فرآیند بوت به صورت کارآمد است.

زبان اسمبلی سطح پایین است و کنترل مستقیم بر سخت‌افزار را فراهم می‌کند.

این فایل مسئول تنظیم محیط اولیه است، مانند تنظیم استک، فعال‌سازی حالت محافظت شده و انتقال از حالت واقعی به حالت محافظت شده است. کد اسمبلی برای انجام عملیات سطح پایین استفاده می‌شود که نیاز به کنترل دقیق بر سخت‌افزار دارد، مانند پیکربندی جدول بردارهای وقفه و مقداردهی واحد مدیریت حافظه (MMU).

یکی از این کار ها وارد شدن به protected mode است که فقط با کد اسمبلی قابل اجراست و با کد c نمیتواند از real mode به protected mode رفت.

زبان C، abstraction سطح بالاتری دارد و برای منطق پیچیده و مدیریت داده مناسب‌تر است و مسئول بارگذاری بقیه سیستم عامل به حافظه و شروع اجرای آن است. کد C برای مدیریت عملیات سیستم فایل، خواندن تصویر هسته از دیسک و انجام وظایف سطح بالاتر مورد نیاز برای بوت سیستم عامل استفاده می‌شود.

با استفاده از ترکیبی از کد اسمبلی و C، فرآیند بوت می‌تواند از کنترل سطح پایین ارائه شده توسط زبان اسمبلی بهره‌برداری کند و در عین حال از انتزاعات و قابلیت‌های سطح بالاتر ارائه شده توسط C بهره‌مند شود. این امر باعث می‌شود فرآیند بوت در xv6 به صورت کارآمد و قابل تنظیم باشد.

14. یک ثبات عام منظوره، یک ثبات قطعه ، یک ثبات وضعیت و یک ثبات کنترلی در معماری 86**x** را نام برده و وظیفه هر یک را به طور مختصر توضیح دهید.

ثبات عام منظوره: پردازنده x86 دارای 8 رجیستر عام منظوره است که یکی از آنها Accumulator register یا همان AX است که برای ذخیره خروجی محاسبات ALU استفاده می شود. در هر مرحله مقدار محاسبه شده در ALU در این رجیستر ذخیره شده و در جای دیگر به عنوان ورودی استفاده می شود.

سایر general-purpose register ها به شرح زیر هستند:

Counter register (CX). Used in shift/rotate instructions and loops.

Data register (DX). Used in arithmetic operations and I/O operations.

Base register (BX). Used as a pointer to data (located in segment register DS, when in segmented mode).

Stack Pointer register (SP). Pointer to the top of the stack.

Stack Base Pointer register (BP). Used to point to the base of the stack.

Source Index register (SI). Used as a pointer to a source in stream operations.

Destination Index register (DI). Used as a pointer to a destination in stream operations.

ثبات قطعه: این پردازنده 6 ثبات قطعه (segment register) دارد که به شرح زیر می باشند:

Stack Segment (SS). Pointer to the stack ('S' stands for 'Stack').

Code Segment (CS). Pointer to the code ('C' stands for 'Code').

Data Segment (DS). Pointer to the data ('D' stands for 'Data').

Extra Segment (ES). Pointer to extra data ('E' stands for 'Extra').

F Segment (FS). Pointer to more extra data ('F' comes after 'E').

G Segment (GS). Pointer to still more extra data ('G' comes after 'F').

برای مثال ثبات پشته برای ذخیره اطلاعات مربوط به قطعه ای است که از آن پشته برای فراخوانی استفاده می شود.

ثبات کنترلی: EFLAGS register یک رجیستر 32 بیتی است که یک رشته از 0 و 1 است که هر کدام یک flag برای یک وضعیت می باشند و میتوانند درست یا غلط باشند. این فلگ ها نشان دهنده وضعیت اعمال منطقی و محاسباتی یا محدودیت های اعمال شده بر عملیات فعلی پردازنده هستند.

FLAGS -> 16-bit

EFLAGS -> 32-bit

RFLAGS -> 64-bit

A screenshot of a document

Description automatically generated

ثبات های کنترلی: این ثبات ها در رفتار کلی پردازنده یا دستگاه های مرتبط به آن نقش دارند برای مثال رجیستر CR0 است که نشان دهنده تغییرات و کنترل های مختلف در رفتار کلی پردازنده است.

18. کد معادل **S.entry** در هسته لینوکس را بیابید**.**

https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry.S

19. چرا این آدرس فیزیکی است؟

در فرآیند بوت، جدول صفحه مسئول نگاشت آدرس‌های مجازی به آدرس‌های فیزیکی است. اما در مراحل ابتدایی بوت، سیستم حافظه مجازی هنوز تنظیم نشده است. بنابراین، استفاده از یک آدرس فیزیکی برای جدول صفحه به سیستم امکان می‌دهد تا نگاشت‌های اولیه را برقرار کرده و خود سیستم حافظه مجازی را تنظیم کند.

حافظه مجازی برای نگاشت به page table نیاز دارد و اگر آدرس آن نیز مجازی بود یک حلقه dependency بی نهایت به وجود می آمد پس حتما آدرس جدول صفحه باید فیزیکی باشد.

**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 تاثیری بر ترجمه آدرس منطقی نمیگذارد. زیرا تمامی قطعه ها اعم از کد و داده روی یکدیگر می‌افتند. با این حال برای کد و داده های سطح کاربر پرچم USER\_SEG تنظیم شده است. چرا؟ (راهنمایی: علت مربوط به ماهیت دستورالعمل‌ها و نه آدرس است.)**  
در xv6، هر دو بخش هسته و کاربر دارای توصیف گرهایی در جدول توصیفگر سراسری(GDT) هستند. این توصیفگرها حاوی اطلاعاتی در مورد بخش ها هستند، مانند آدرس شروع، اندازه و سطح دسترسی.

هنگام اجرای یک دستورالعمل، قطعه مربوطه (اعم از کد یا داده) ابتدا از طریق توصیفگر آن در GDT قرار می گیرد. توصیفگرهای هسته و کاربر می توانند سطوح دسترسی متفاوتی داشته باشند، حتی اگر به حافظه فیزیکی یکسانی اشاره کنند. سطح دسترسی، امتیاز مورد نیاز برای اجرای دستورالعمل را تعیین می کند.

در طی این فرآیند، سطح امتیاز فعلی (CPL) بر اساس سطح دسترسی مشخص شده در توصیفگر تعیین می شود. اگر CPL کمتر از سطح دسترسی مورد نیاز باشد، دستور نمی تواند اجرا شود. برای مثال، برخی دستورالعمل‌های ممتاز ممکن است به سطح دسترسی بالاتری نسبت به آنچه برای کد سطح کاربر مجاز است نیاز داشته باشند.

وقتی USER\_SEG تنظیم می‌شود، به این معنی است که کدهای سطح کاربر و بخش‌های داده مجوزهای محدودی دارند و از اجرای برخی عملیات ممتاز برنامه‌های سطح کاربر که می‌تواند به یکپارچگی سیستم آسیب برساند، جلوگیری كند. در مقابل، کدهای سطح kernel و بخش‌های داده معمولاً مجوزهای آزادتری دارند زیرا نیاز به تعامل با سخت‌افزار و مدیریت سیستم دارند.

تنظیم پرچم USER\_SEG برای بخش‌های سطح کاربر کمک می‌کند تا اطمینان حاصل شود که فرآیندهای کاربر نمی‌توانند دستورالعمل‌های ممتاز را اجرا کنند یا به بخش‌های محدود حافظه دسترسی پیدا کنند، و امنیت و پایداری سیستم عامل را افزایش می‌دهد.

قطعه‌بندی در xv6 در تابع seginit و در تکه کد زیر انجام می‌شود که در فایل vm.c وجود دارد :

تعریف SEG نیز به صورت زیر می‌باشد که در فایل mmu.h وجود دارد:A screenshot of a computer program

Description automatically generatedA screen shot of a computer code

Description automatically generated

اجرای نخستین برنامه سطح کاربر

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

این ساختار که در فایل proc.h تعریف شده از اجزای زیر تشکیل شده است:

**uint sz:** حجم حافظه اشغال شده توسط پردازش به بایت

**pde\_t\* pgdir:** پوینتر به page table directory entry

Page table در هر پردازش یک نگاشت بین حافظه مجازی و فیزیکی ایجاد می کند.

**char \*kstack:** هر پردازش در kernel نیاز به یک stack جداگانه برای ذخیره حالتش دارد. این پوینتر به پایین ترین خانه kernel stack که به این پردازش اختصاص دارد اشاره می کند.

**enum procstate state :** وضعیت پردازش را مشخص می کند.

حالات ممکن: UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE

**int pid :** یک عدد یکتا برای هر پردازش با عنوان process identifier

**struct proc \*parent :** اشاره گر به پردازش parent. وقتی پردازش پدر تابع fork را call کرده است این پردازش با همان حالت ها و مموری و … ایجاد شده است.

**struct trapframe \*tf :** اشاره گر به trap frame.

Trap frame آرگومان های لازم برای هندل کردن trap ها و اجرای sysCall ها را فراهم می کند و وضعیت پردازش را قبل از اجرای sysCall ذخیره می کند تا از همان حالت ادامه دهد.

**struct context \*context :** اشاره گر به ساختار context

این struct هنگام suspend شدن پردازش و سوییچ کردن به دیگری با تابع swtch محتوای رجیستر ها را ذخیره می‌کند تا دوباره بتواند از همان جای قبلی ادامه یابد. این ساختار شامل اجزای زیر می باشد:

edi: Destination index, for string operations

esi: Source index, for string operations

ebx: Base index, for use with arrays

ebp: Stack Base Pointer, for holding the address of the current stack frame

eip: Instruction Pointer, points to instruction to be executed

**void \*chan:** اگر مقداری غیر صفر داشته باشد یعنی پردازش در حالت sleeping است و برای انجام کاری wait می کند.

**void \*killed:** اگر مقدار غیر صفر داشته باشد یعنی پردازش kill شده است.

**struct file \*ofile :** آرایه ای پوینتر به فایل های باز شده توسط پردازش. هنگامی که فایلی باز می شود یک پوینتر در اولین خانه خالی این آرایه به آن فایل ذخیره می شود و index آن خانه به عنوان file descriptor باز گردانده می شود.

current working directory **: struct inode \*cwd**

**char name :** نام پردازش برای debug

معادل این struct در هسته لینوکس:

استراکت task\_struct

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

27. 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. هسته های دیگر از طریق

entryother.s به تابع mpenter می‌روند که در این تابع 4 تابع فراخوانی می‌شود که این چهار تابع در تابع main نیز فراخوانی شده‌اند. (تابع switchkvm در تابع kvmalloc که در main است فراخوانی شده است.)

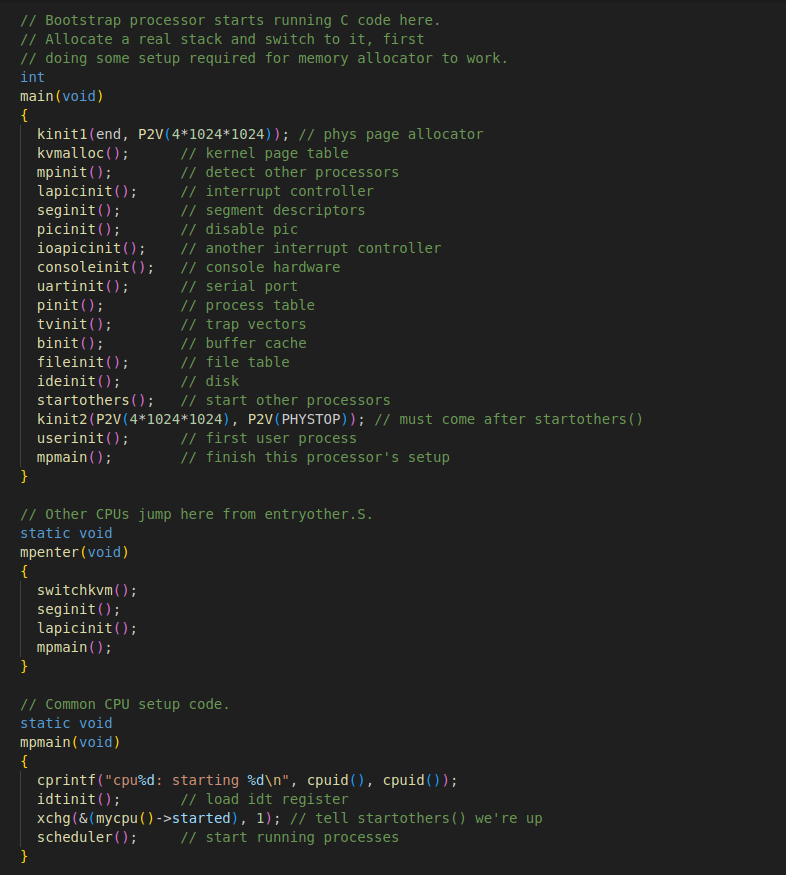
به طور کلی در main 18 تابع فراخوانی شده اند که آن 4 تابع فراخوانی شده در mpenter بین تمامی هسته های پردازنده مشترک هستند و 14 تابع دیگر اختصاصی هستند.

مشترک ها: switchkvm,seginit,lapicinit,mpmain

اختصاصی‌ها:

kinitl,kvmalloc(setupkvm),mpinit,picinit,ioapicinit,consoleinit,

uartinit,pinit,tvinit,binit,fileinit,ideinit,startothers,kinit2,userinit

زمانبند یا scheduler در تابع mpmain صدا زده می‌شود و بين تمامی هسته های پردازنده مشترک است و هر پردازنده scheduler خودش را دارد و بعد از setup کردن خودش آن را فراخوانی می‌کند.

که از توضیحات قبل از تعریف scheduler قابل برداشت است:

**اشکال زدایی**

**روند اجرای GDB**

**.1برای مشاهده breakpoint ها از چه دستوری استفاده می‌شود؟**

برای این کار کافی است که از دستور info breakpoint، info break استفاده کنیم.

A screenshot of a computer

Description automatically generated

**2. برای حذف یک breakpoint ازچه دستوری و چگونه استفاده می‌شود؟**

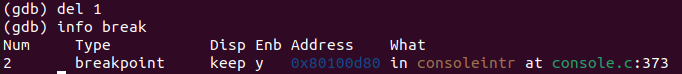
برای حذف یک breakpoint میتوان ابتدا با دستور گفته شده در قسمت قبل شماره breakpoint را بدست آورد و سپس با استفاده از دستور del به صورت del <breakpoint\_number>، breakpoint دلخواه را حذف کرد.

برای مثال اگر فرض کنیم breakpoint های ما در ابتدا به صورت زیر است:

A screenshot of a computer program

Description automatically generated

سپس با وارد کردن دستور del 1 خواهیم داشت:



همانطور که میبینیم breakpoint شماره یک حذف شد.

**کنترل روند اجرا و دسترسی به حالت سیستم**

**3. خروجی دستور bt چه چیزی را نشان می‌دهد؟**

با فراخوانی هر تابع، آن یک stack frame برای خود در نظر میگیرد و مکان فراخوانی ها ( یا همان بازگشت ها) و متغیر های محلی را ذخیره می‌کند.

bt (backtrace) دستوری است که به وسیله آن در لحظه توقف برنامه می‌توان call stack برنامه را در لحظه توقف دید. در واقع آن خلاصه ای از مسیری است که برنامه برای رسیدن به آن نقطه طی کرده است.

خط اول مربوط frame مربوط به اخرین جایی است که برنامه در آن متوقف شده و در هر خط بعدی هر کدام صدازننده frame بالایی اش است.

bt [option] … [qualifier]… [count]

backtrace [option] … [qualifier]… [count]

[count]:

با وارد کردن دستور bt به تنهایی، تمام call stack چاپ می‌شود. برای کنترل تعداد frame های چاپ شده می توان از bt n و bt -n استفاده کرد.

bt n : n frame درونی تر را چاپ می‌کند.

bt -n : n frame بیرونی‌تر را چاپ می‌کند.

مثال [option]:

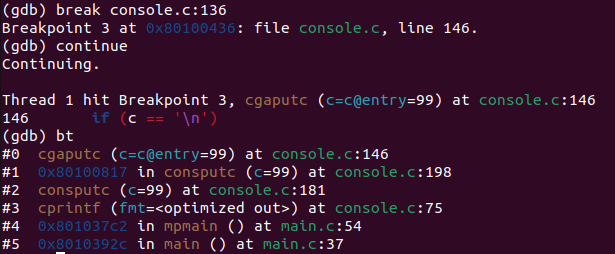
با وارد کردن bt -full در هر خط مقدار متغیر های محلی نیز چاپ می‌شود.

در مثال زیر با قرار دادن breakpoint روی خط 8 فایل cat.c پس از وارد کردن دستور bt میبینیم که ابتدا در main در خط 39 تابع cat فراخوانی می‌شود و سپس به تابع cat در خط 12 رفته و در آن تابع برنامه متوقف می‌شود.

A screenshot of a computer

Description automatically generated

مثالی دیگر:



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

نحوه دریافت ورودی این دو دستور و نیز نحوه نمایش اطلاعات آن‌ها با هم متفاوت است. با استفاده از دستور print (به اختصار p) می‌توان مقدار یک متغیر (variable) یا یک عبارت دلخواه (arbitrary expression) را چاپ کرد.

در هر دو دستور می‌توانیم فرمت خروجی را به صورت /FMT در آرگومان ها مشخص کنیم.

**دستور x :** همانطور که گفته شده این دستور محتویات یک خانه حافظه را نمایش می‌دهد و به اشکال زیر قابل استفاده است:

**x** [*Address expression*]  
**x** /[*Format*] [*Address expression*]  
**x** /[*Length*][*Format*] [*Address expression*]

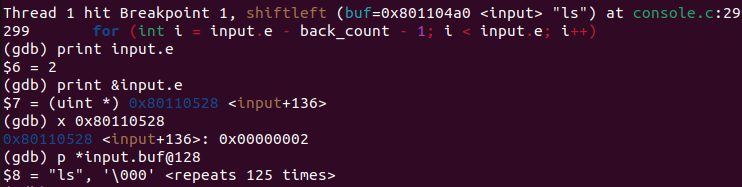
برای مشخص کردن format میتوانیم از موارد زیر استفاده کنیم:

* o - octal
* x - hexadecimal
* d - decimal
* u - unsigned decimal
* t - binary
* f - floating point
* a - address
* c - char
* s - string
* i - instruction

**دستور print:** مقدار متغیر داده شده را چاپ خواهد کرد و برای مثال میتوان به شکل زیر از آن استفاده کرد:

**print** [*Expression*]  
**print** [*First element*]@[*Element count*]  
**print** /[*Format*] [*Expression*]

برای مثال میتوان مقدار متغیر i را با وارد کردن دستور print i مشاهده کرد. همچنین می‌توانیم آرایه هارا ببا وارد کردن \*array@length چاپ کنیم که در آن array نام آرایه و length طولی از آرایه که مایل به چاپ آن هستیم را نشان می‌دهند.



در ادامه باید عنوان کرد که برای مشاهده محتوای همه ثبات ها میتوان از دستور info registers استفاده کرد که میتوان از کوتاه شده آن به صورت i r استفاده کرد.

اگر بخواهیم تنها محتویات یک ثبات خاص را ببینیم از دستور info registers <register\_name> استفاده می‌کنیم که محتویات ثبات <register\_name> را نشان خواهد داد.



**5. برای نمایش وضعیت ثبات ها از چه دستوری استفاده می‌شود؟ متغیرهای محلی چطور؟ در معماری x86 رجیستر های edi و esi نشانگر چه چیزی هستند؟**

همانطور که پیش تر گفته شد میتوان برای دیدن محتویات همه رجیستر ها از دستور info registers یا i r استفاده کرد.

A screenshot of a computer

Description automatically generated

برای دیدن متغیر های محلی نیز می‌توان از دستور info local استفاده کرد.

برای مثال متغیر های محلی مربوط به تابع consoleintr پس از وارد کردن Ctrl+U به شکل زیر است:

A black background with white text

Description automatically generated

در معماری x86، تعدادی عملیات وجود دارد که فقط با استفاده از رجیسترهای DI و SI قابل انجام هستند. این عملیات عبارتند از:

REP STOSB

REP MOVSB

REP SCASB

که به ترتیب عملیات‌های ذخیره‌سازی تکراری (= جمعی)، بارگیری و اسکن می‌باشند. شما SI و/یا DI را برای اشاره به یک یا هر دو عملوند تنظیم می‌کنید، شاید یک شمارنده را در CX قرار دهید و سپس عملیات را انجام می‌دهید. این عملیات‌ها بر روی یک دسته از بایت‌ها به صورت همزمان عمل می‌کنند و نوعی حالت خودکار به CPU اعمال می‌کنند. زیرا شما حلقه‌ها را به صورت صریح کد نمی‌کنید.

رجیستر EDI (Extended Destination Index) معمولاً برای نشان دادن آدرس مقصد در عملیات‌های حلقه (مانند دستورات LOOP و REP) استفاده می‌شود. این رجیستر در عملیات‌هایی که نیاز به تکرار عملیات بر روی داده‌ها دارند، مورد استفاده قرار می‌گیرد.

رجیستر ESI (Extended Source Index) نیز برای نشان دادن آدرس منبع در عملیات‌های جابجایی داده‌ها (مانند دستورات MOVSB و CMPSB) استفاده می‌شود. این رجیستر در عملیات‌هایی که نیاز به انتقال داده‌ها از یک مکان به مکان دیگر دارند، مورد استفاده قرار می‌گیرد. ، آنها کار خود را به طور معمول به طور کارآمدتری انجام می‌دهند تا یک حلقه به صورت دستی کدنویسی شده.

به طور خلاصه، رجیستر EDI برای نشان دادن آدرس مقصد و رجیستر ESI برای نشان دادن آدرس منبع در عملیات‌های حلقه و جابجایی داده‌ها استفاده می‌شوند.

**6. به کمک استفاده از gdb ساختار struct input را توضیح دهید.**

نکته: این توضیحات ابتدا با فرض کد داده شده( لینک گیت هاب قرار داده شده) توضیح داده می‌شود و سپس توضیحات تغییرات داده شده در کد خود گروه نیز داده می‌شود.

struct {

char buf[INPUT\_BUF];

uint r; // Read index

uint w; // Write index

uint e; // Edit index

} input;

این unnamed struct در فایل console.c قرار داده شده و از یک instance ان به نام input استفاده می‌شود.

این struct دارای 4 متغیر است:

* buf: آرایه ای به سایز 128 است که محل ذخیره خط ورودی است.
* e : نشان دهنده محل کنونی کرسر در خط است.
* w : محل شروع خط را نشان می‌دهد.
* r : برای خواندن buf بعد از زدن enter استفاده می‌شود.

برای نشان دادن تغییرات مربوط به اطلاعات ایندکس ها(e,w,r) دو breakpoint در consoleintr قرار داده شد.

در ابتدا که چیزی در ورودی نوشته نشده است مقدار متغیر های input به شکل زیر است:



با واردکردن عبارت elahe و سپس زدن enter ابتدا میبینیم که مقدار w برابر e است. 

سپس پس از خوانده شدن خط ( در اخر switch case در اخر قسمت default) خواهیم دید که مقدار r نیز برابر مقدار wمی‌شود. در واقع r یکی یکی زیاد می‌شود و ورودی جدید رو میخواند تا به انتها(تا جایی که در آن خط ورودی نوشته شده بود) برسد.

سپس عبارت op را در ورودی می‌نویسیم و خواهیم دید که به مقدار e دو واحد اضافه می‌شود( به اندازه طول ورودی جدید)



همچنین پس از زدن هر backspace نیز مقدار e یکی کم می‌شود.

در واقع input.e نشان دهنده جایی است که ما مینویسیم و با اضافه کردن هر حرف جدید یکی به آن اضافه می‌شود و با زدن هر backspace نیز یکی از مقدار آن کم می‌شود.

Input.w نیز در هر لحظه نشان دهنده ابتدای خطی است که در حال نوشتن هستیم. پس از وارد کردن هر enter مقدار آن برابر input.e می‌شود(می‌توان گفت به سر خط می‌رود(.

Input.r نیز برای خواندن بافر استفاده می‌شود. در هر مرحله پس زدن enter خط قبلی نوشته شده را میخواند.

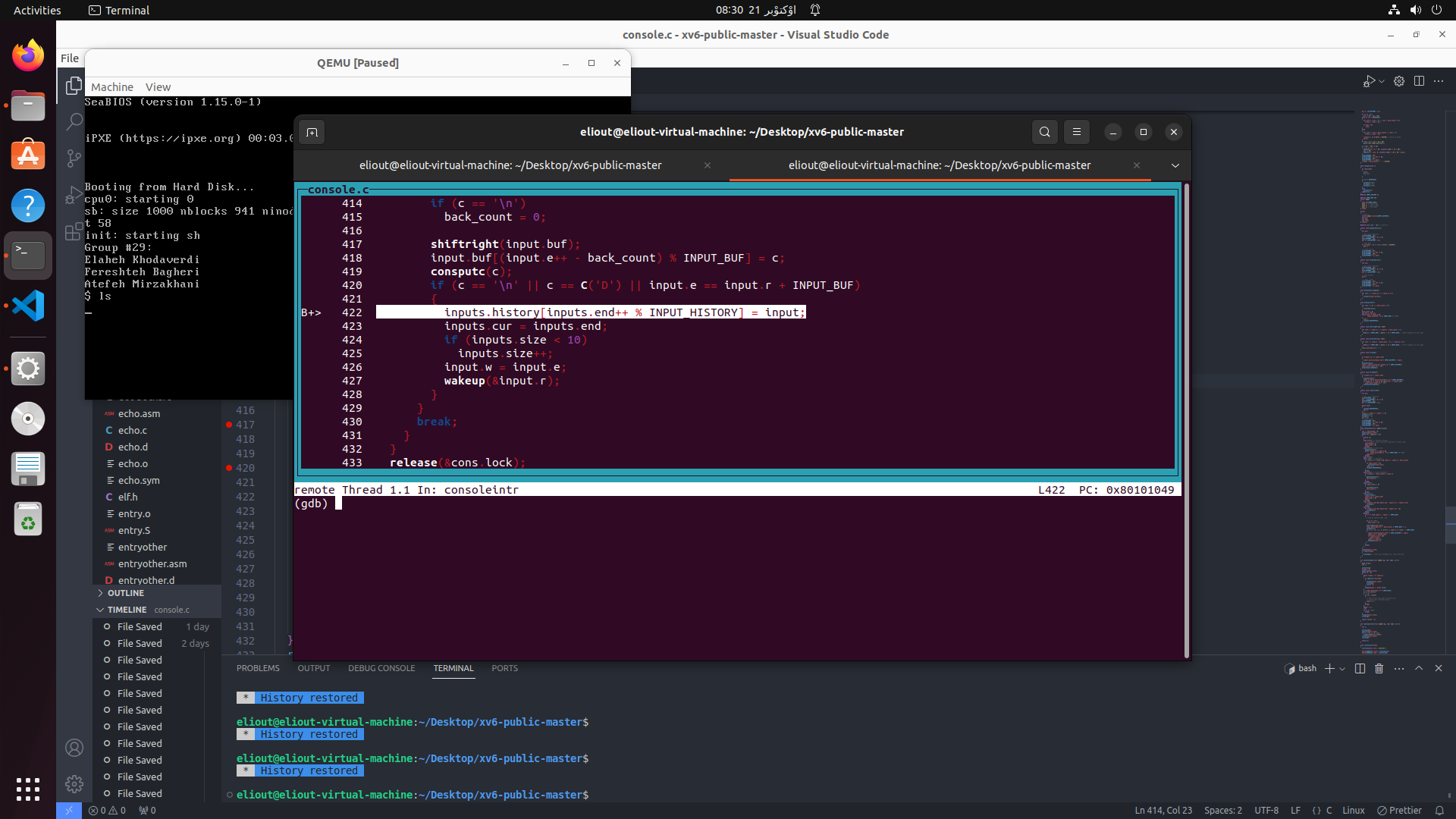
باید عنوان کرد که در کد مربوط به پروژه آزمایشگاه به دلیل نیاز به پیاده‌سازی قابلیت های عنوان شده ما در کد خود input.e را جایی که مینویسیم در نظر نگرفته و بلکه آن را انتهای خطی که مینویسیم در نظر گرفته‌ایم.

برای پیاده‌سازی قابلیت ها، input.e در کد ما نشان دهنده آخر خط و برای پیدا کردن جایی که قرار داریم و در حال ویرایش آن نقطه هستیم از متغیر back\_count استفاده کردیم که نشان دهنده تعداد خانه هایی است که از آخر خط به عقب بازگشته ایم. همچنین به علت استفاده از این struct در جای دیگر ما آن را Input نامگذاری کردیم.

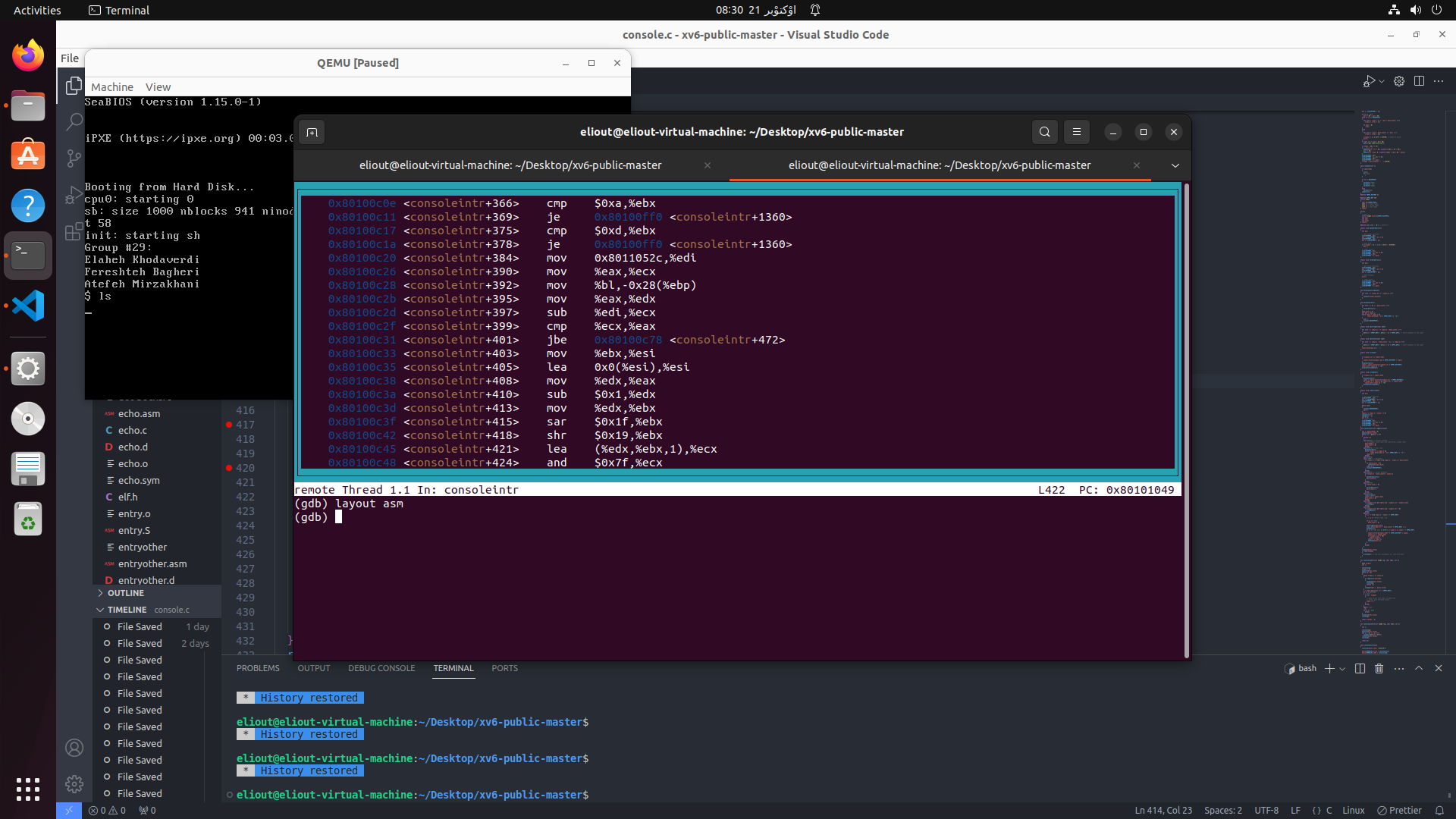
**اشکال زدایی در سطح کد اسمبلی**

**7. خروجی دستور های layout src و layout asm در TUI چیست؟**

با وارد کردن دستور layout src میتوانیم کد c مربوطه را هنگام توقف و دیباگ در محیط TUI ببینیم.



با وارد کردن دستور layout asm می‌توانیم کد اسمبلی را هنگام توقف و دیباگ در محیط TUI ببینیم.

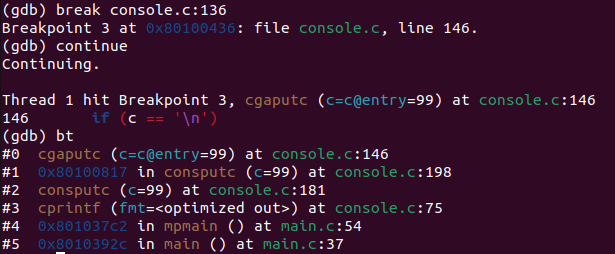


میتوان با وارد کردن layout split می‌توان هر دو کد سورس و کد اسمبلی را همزمان دید.

**8. برای جا به جایی میان توابع زنجیره ای فراخوانی جاری (نقطه توقف) از چه دستور هایی استفاده می‌شود؟**

همانطور که در سوال های بالاتر دیدیم میتوانیم پشته فراخوانی را با استفاده از دستور bt یا backtrace ببینیم. حال پس از آن برای جابه جایی میان توابع زنجیره فراخوانی می‌توان از دستور up و down استفاده کرد که به ترتیب به چند تابع بالاتر و چند تابع پایین تر می‌روند. میتوان تعداد توابعی که به بالا یا پایین می‌رویم را به صورت up <n> و down <n> مشخص کنیم. در صورت مشخص نکردن تعداد مقدار default برای آن 1 در نظر گرفته می‌شود.

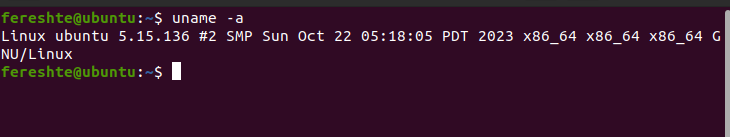
برای مثال اگر پشته فراخوانی مانند تصویر زیر باشد:



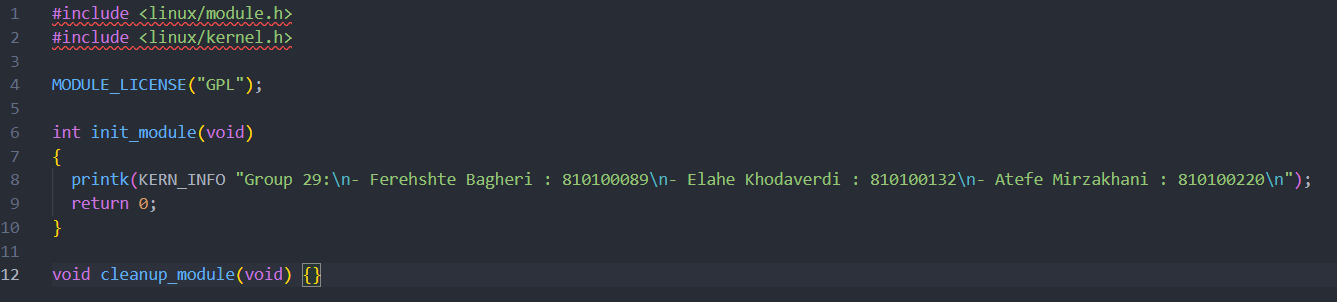
با وارد کردن دستور up به تابع consputc در خط 198 فایل console.c می‌رویم.

**پیکربندی هسته لینوکس (امتیازی)**

پس از نصب ابونتو روی VMware با استفاده از دستور عمل مربوط به این [لینک](https://davidaugustat.com/linux/how-to-compile-linux-kernel-on-ubuntu) یک نسخه نزدیک به هسته قبلی خود دانلود و نصب کردیم و پس از انجام مراحل با دستور uname -a نسخه جدید را دیدیم که نسخه 5.15.136 است.



سپس با ساختن یک فایل به نام group.c کد مربوط به چاپ اعضای گروه را در آن گذاشته و برای آن یک makefile در همان جا ایجاد می‌کنیم. کد مربوط به هر فایل به صورت زیر است:



A screen shot of a computer

Description automatically generated

پس از وارد کردن دستور make در ترمینال از دستور sudo insmod group1.ko استفاده میکنیم. در نهایت اگر دستور dmesgرا اجرا کنیم، میتوانیم اسم  
اعضای گروه را در انتهای خروجی این دستور مشاهده کنیم:

A computer screen shot of a program

Description automatically generated

A screenshot of a computer code

Description automatically generated