**آزمایشگاه سیستم عامل**

پروژه شماره 1

علی پادیاو – کسری حاجی حیدری – اولدوز نیساری

بهار 1402

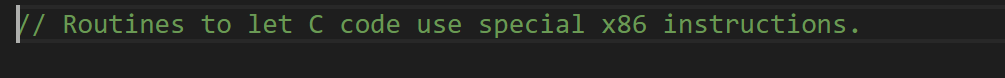
Repository Link: <https://github.com/alumpish/OS-Lab-Projects>

Latest Commit Hash: 7443b8b570c3b7e97ddc94e3788ecec62186ed85

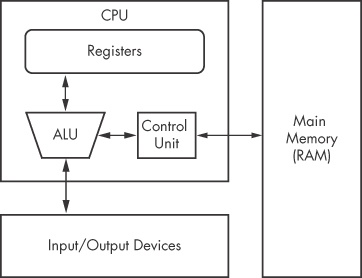
**آشنایی با سیستم عامل xv6**

xv6 در حقیقت یک نوع پیاده سازی مجدد از ورژن 6 ام unix است که با استفاده از زبان C و بر اساس پردازنده های مبتنی بر x86 نوشته شده است.

بر دفاع از نظر خودمان که گفتیم بر مبنای x 86 است می توانیم به وجود فایل هایی با پسوند 86 اشاره کنیم مثل فایل های : x86.h در توضیحات آن هم نوشته شده است بر مبنای instruction های 86 است استناد کنیم.



نکات تکمیلی : خود x86 برمبنای CISC نوشته شده است و یک مدل ساده از آن به صورت زیر است:

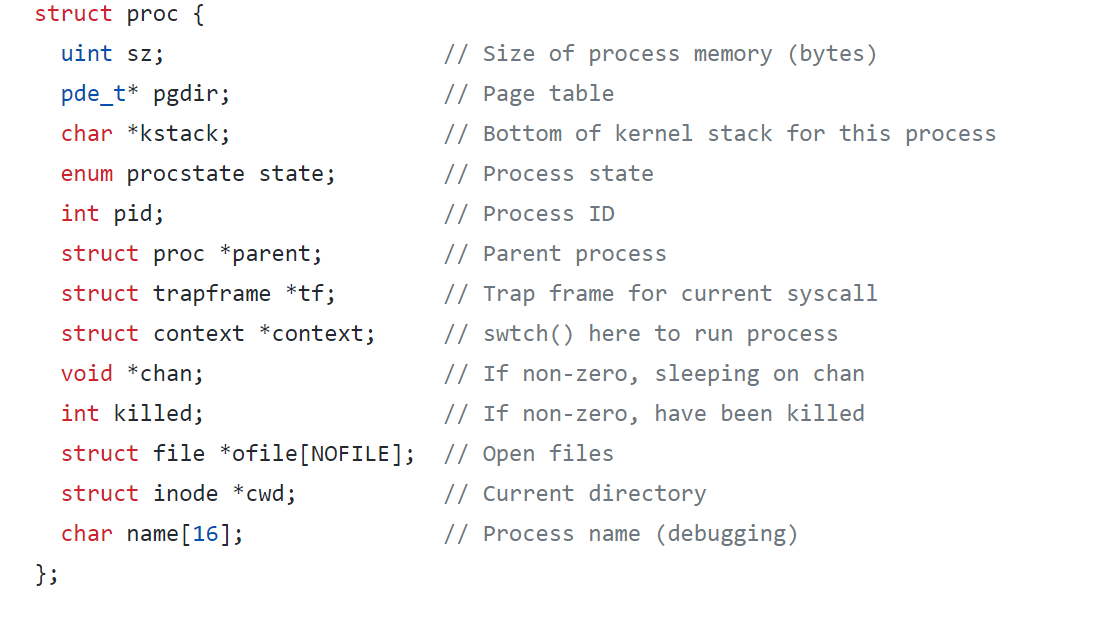


همچنین نمایی کلی از ساختار unix هم به صورت زیر است:





با دقت در فایل proc.h که در فایل های xv6 است می توانیم چند بخش مهم process را به صورت زیر بگوییم :



1. Kernel stack:

این بخش در استراکت با kstack مشخص شده است که از جنس پوینتر است. (پوینتری بهkernel stack)

در زمان function call وضعیت رجیستر ها در استک کاربر ذخیره می شوند. به صورت مشابه وقتی پردازه به سیستم عامل می پرد تا کد کرنل را اجرا کند ، CPU context در kernel stack ذخیره می‌شود.

1. List of open files:

نام این بخش در استراکت ofile[NOFILE] است که در حقیقت آرایه ای از پوینتر ها است به open file ها.

هر وقت کاربر یک فایل را باز می کند یک entry جدید به آرایه اضافه می شود که ایندکس آن entry به عنوان file describer کاربر برگردانده می شود. (زمانی که کاربر می خواهد فایلی را بخواند یا بنویسد از این file describer ها برای مراجعه استفاده می کنند)



هم چنین در این آرایه 3 خانه اول برای standard input, output , error کنار گذاشته شده‌اند.

1. Page table:

نام این بخش در استراکت pgdir است که پوینتری به page table است.

هر دستور یا داده در memory image ( الگو برنامه نویسی که داده ها در آن ذخیره شده اند).

یک آدرس واقعی و یک آدرس مجازی دارند ، درpage table ، mappingبین این دو آدرس نگهداری می‌شود.

هم چنین بر اساس کتاب rev 11 میتوانیم بگوییم:

هر پردازه یک مموری دارد که شامل دستورات ، داده ها و یک استک است.دستورات محاسبات برنامه را انجام میدهد. داده ها بخش هایی هستند که محاسبات روی آن ها انجام می شوند و استک بخشی است فرآیند فراخوانی های برنامه در آن مدیریت می شود.به این حافظه user-space می‌گویند.

هر پردازه وضعیت پردازه ای هم دارد که برای kernel مخفی است.

برای مدیریت پردازه های مختلف روش مورد استفاده time-share است. در این حالت به صورت نامحسوس سیستم عامل ، پردازنده های مختلف را بین پردازه های مختلف که منتظر اجرا هستند به اشتراک می گذارد. زمانی که یک پردازه اجرا نمیشود تمام رجیستر های آن ذخیره میشوند تا در زمان بازگشت مجدد به آن پردازه بتوانند بازیابی شوند.

Kernel به هر کدام از پردازه ها یک id هم میدهد.

3-

همان طور که در سوال قبل اندکی توضیح داده شد ، جدولی در سیستم عامل وجود دارد که file describer مانند یک index برای آن عمل می کند. عملا هر پردازه یک فضای خصوصی برای ذخیره file descriptor هایش دارد که از 0 شروع می شود و همان طور که گفته شده خانه اول (0) برای ورودی (stdin) ، خانه دوم (1) برای خروجی (stdout) و خانه سوم (2) برای ارور (stdout) اختصاص داده شده است.

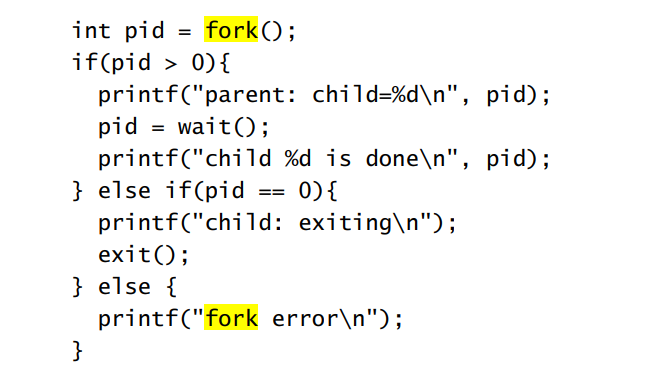
File descriptor در حقیقت یک عدد صحیح است که به یک شی که kernel قابلیت مدیریت آن را دارد و پردازه ممکن است از آن بخواند یا بنویسد اشاره می‌کند.

هر پردازه زمانی که یک فایل ، یک directory یا یک device را باز می کند یک file descriptor به دست می آورد. کاری که file descriptor می‌کند این است که یک interface انتزاعی ایجاد می کند که تفاوت بین این ها دیده نشود و همگی مثل جریان هایی از بیت ها باشند.

عملکرد pipe به این صورت است که دو file descriptor را در نظر می گیرد و آن ها به هم وصل می کند. یکی را برای نوشتن و دیگری را برای خواندن. نوشتن روی یکی از file descriptor ها ، امکان خواندن را از روی دیگر file descriptor فراهم می کند. در حقیقت pipe ها امکان ارتباط و تعامل را برایfile descriptor ها فراهم می کند.

4-

Fork برای هر پردازه یک پردازه دقیقا با همان محتواهای حافظه به نام پردازه فرزند می سازد. به پردازه اولیه اصطلاحا پردازه پدر می گویند. Fork برای هر یک از پردازه های پدر و فرزند مقدار های متفاوتی را برمی گرداند برای پدر pid را برمی گرداند و برای فرزند 0.



در حقیقت اتفاقی که در سیستم عامل می افتد تا حدی در تکه کد بالا توضیح داده شده است. زمانی که pid صفر است ، exit system call یعنی فرآیند آزاد سازی منابع مثل حافظه و باز کردن فایل ها اتفاق می افتد. زمانی که pid از 0 بزرگتر باشد wait system call اتفاق می افتد.wait system call ، pid فرزند خارج شده پردازه را برمیگرداند.اگر هیچ یک از فرزندان پردازه خارج نشده بودند صبر می کند تا یکی از آن ها خارج شوند.

با وجود این که پردازه پدر و فرزند محتوای حافظه یکسانی دارند اما آن ها با مموری ها و رجیستر های متفاوتی اجرا می شوند.برای مثال تغییر دادن یک متغیر در حافظه یک پردازه روی دیگری تاثیر نمی گذارد.

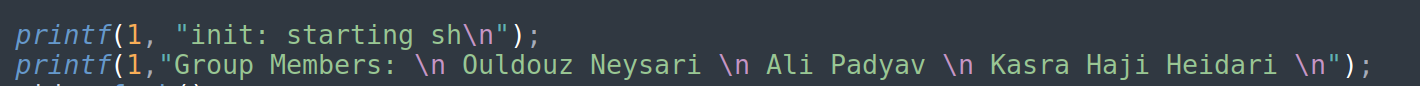
Exec اما متفاوت است. exec پردازه در حال اجرا را با پردازه ای که آدرس و آرگومان آن داده می شود جایگزین می کند اما file table اولیه را هم حفظ میکند.

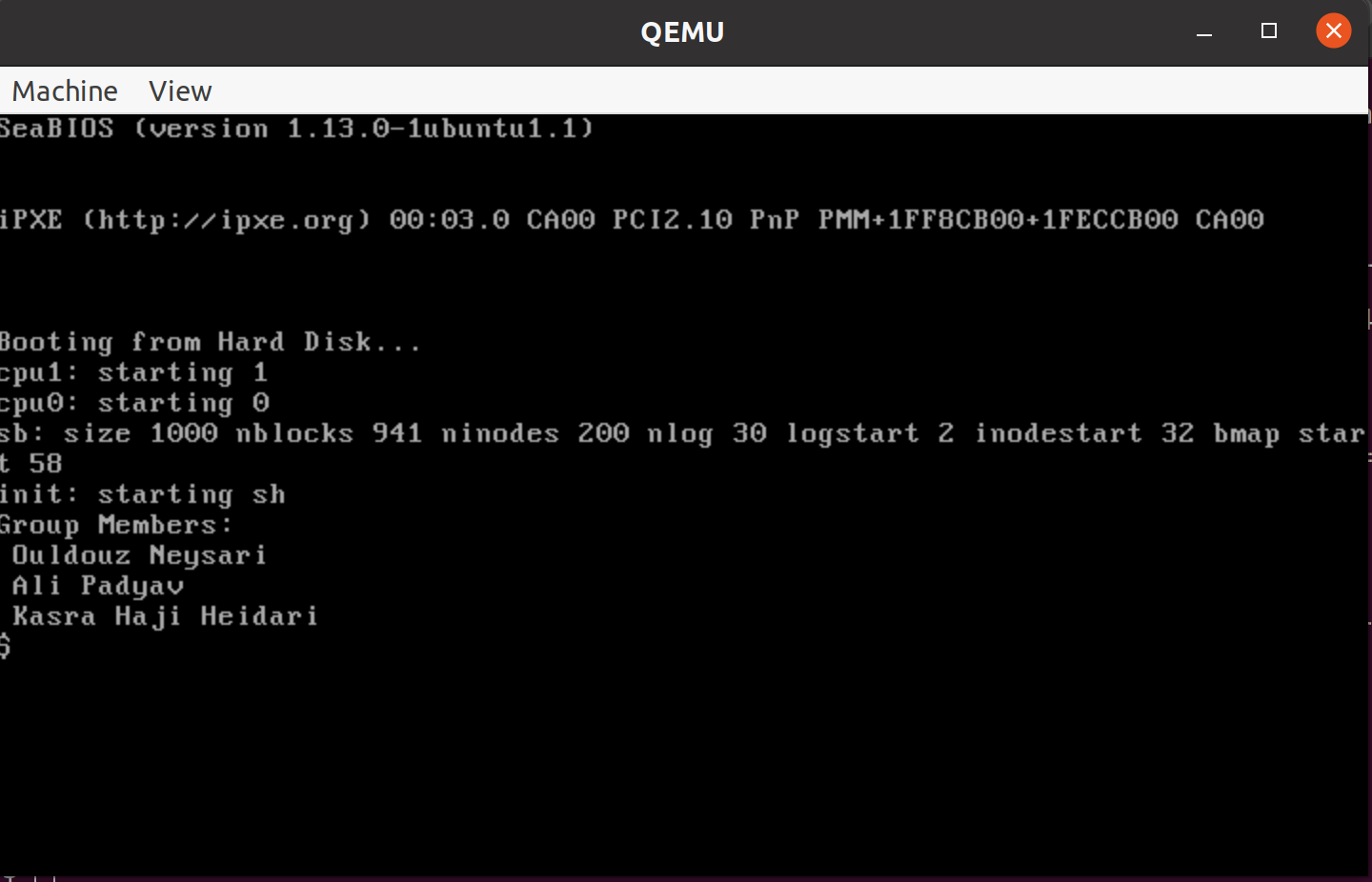
دلیل ادغام نکردن این دو در بخش پیاده سازی I/O است.در حقیقت ادغام نکردن از ساختن پردازه های بی مصرف و جایگزین شدن سریع آن ها توسط exec جلوگیری می کند.

در حالت عادی در زمانی که تابع fork و exec پشت سر هم اجرا می شوند. اگر این دو ادغام شوند علاوه بر پردازه های اضافه و میزان حافظه زیادی که اشغال می شود مدیریت آرگومان های توابع هم دشوار می شود.

**اضافه کردن یک متن به Boot massage :**

با توجه به فرمت printf در فایل user.h خط زیر را در فایل init.c اضافه می کنیم و در ادامه نتیجه:



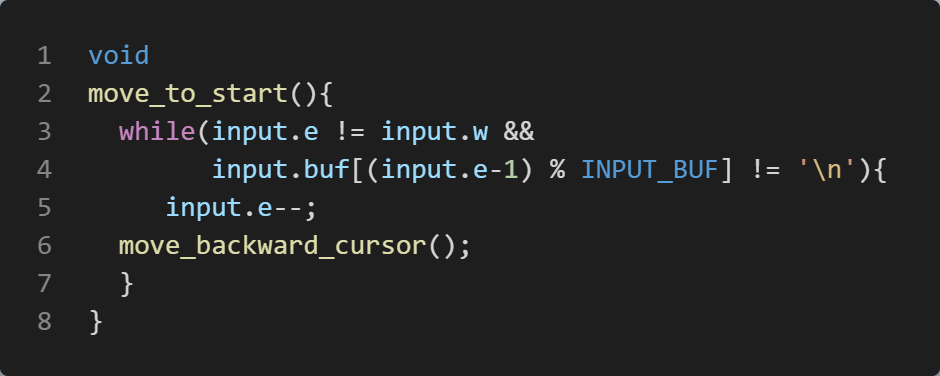


**اضافه کردن چند قابلیت به کنسول xv6 :**

ابتدا دو قابلیت move\_backward\_cursor و move\_forward\_cursor را اضافه و از آن‌ها برای قابلیت‌های جدید استفاده می‌کنیم.

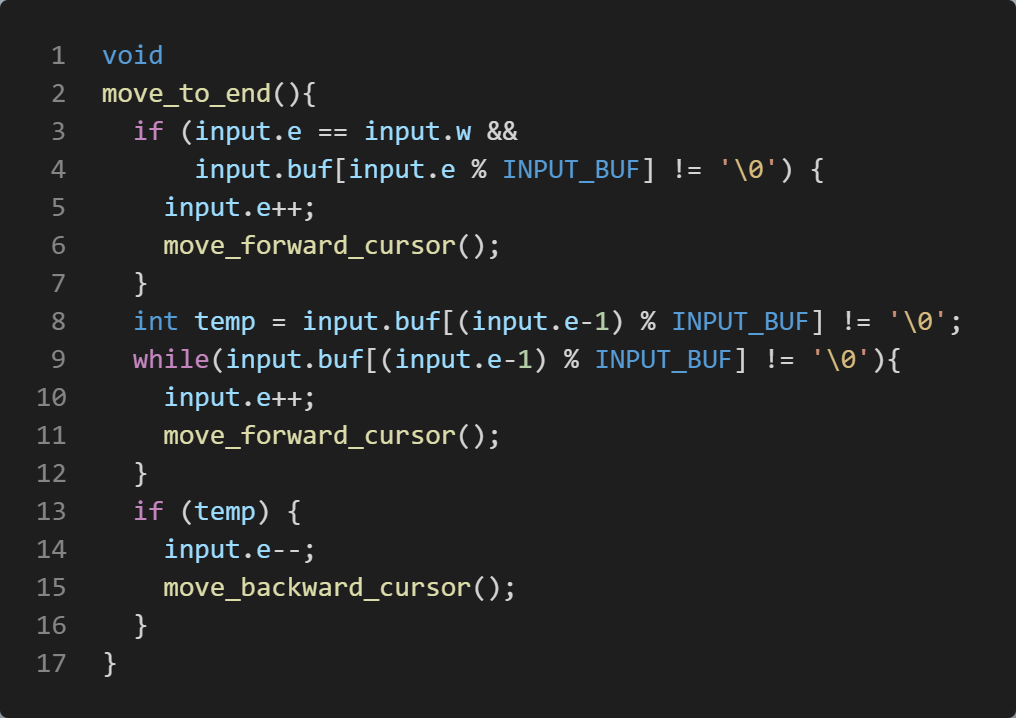
1. رفتن به ابتدای خط با دستور shift + [:

تابع زیر را به console.c اضافه کرده و در switch case حالت S(‘[‘) این تابع را فراخوانی می‌کنیم.



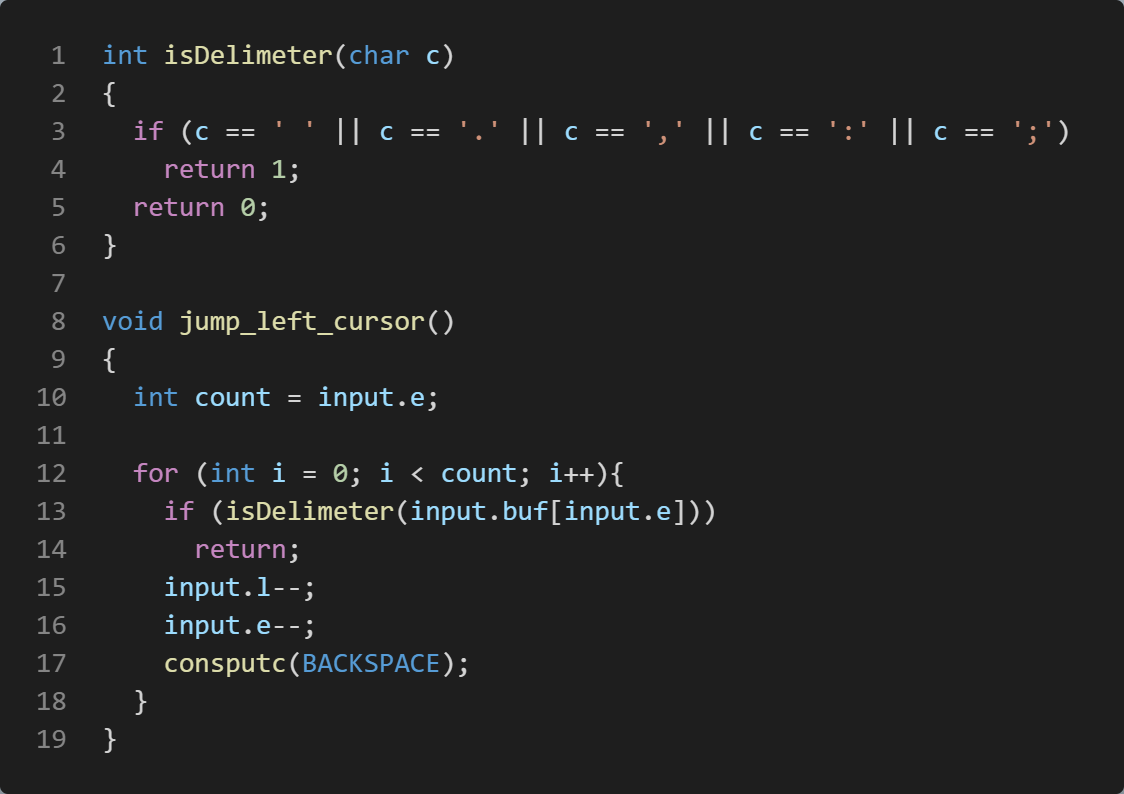
1. رفتن به انتهای خط با دستور shift + ]:

مثل حالت قبل تابع مورد نظر را اضافه و در case مناسب آن را فراخوانی می‌کنیم.



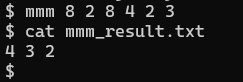
1. پاک کردن کلمه قبل از کرسر با دستور ctrl + w:

درست مثل 2 حالت قبل تابع junp\_left\_cursor را به console.c اضافه و در کیس C(‘W’) تابع را فراخوانی میکنیم.



**اجرا و پیاده سازی یک برنامه سطح کاربر:**

برنامه mmm.c نوشته و در فلدر اصلی xv6 قرارداده شد. همینطور آن را به متغیر UPGROS در makefile اضافه کردیم تا به عنوان برنامه سطح کاربر شناخته شود.



**کامپایل سیستم عامل xv6 :**

8-

متغیر UPROGS:

در حقیقت مخفف user programs است که یک لیست از برنامه های کاربر دارد که در هنگام کامپایل شدن xv6 این برنامه ها نیز کامپایل می شوند و هر یک قابل اجرا توسط سیستم عامل می شوند.

(در زمان اجرای دستور هایی مثل cat, mkdir مورد استفاده قرار می گیرند)

متعیر ULIB:

در حقیقت مخفف user libraries است که شامل لیست کتابخانه هایی است که در vx6 استفاده شده است. شامل موارد زیر می شود :

ulib.o

usys.o

printf.o

umalloc.o

**مراحل بوت سیستم عامل xv6 :**

**اجرای بوت لودر:**

11-

اگر در فایل های مربوط به boot در make file دقت کنیم ، متوجه می شویم که دو فایل Bootasm.s و bootmain.c در مرحله بوت کامپایل می شوند. (قابل مشاهده است که فایل هایی با پسوند.o برای آن ها تشکیل می شود)

Bootasm به زبان اسمبلی است و bootmain.c به زبان سی. تفاوت این فایل ها با بقیه فایل ها این است که در اولین سکتور هارد دیسک قرار می گیرند. علت استفاده از اسمبلی و c این است که این بخش سیستم عامل با سخت افزار رابطه نزدیکی دارد، و کنترل بهتری به سخت افزار برایمان فراهم می کنند.

12-

این دستور همان طور که از اسمش مشخص است کاری که انجام میدهد به این صورت است که محتویات یک فایل object را در یک فایل object دیگر کپی می کند.در حقیقت کاری که انجام میدهد به این صورت است که فرمت فایل را در هنگام کپی تغییر میدهد. این که objcpy دقیقا چه کاری انجام دهد توسط کاربر و کامندی که در ترمینال می نویسد مشخص می شود اما به طور کلی objcpy فایل‌های موقتی تشکیل می دهد تا بتواند ترجمه هایش را انجام دهد.

در make file از objcpy استفاده می شود تا raw binary file هایی ساخته شود تا پردازنده بدون نیاز به سیستم عامل بتواند آن ها را اجرا کند.

13-

اجرای کد های زبان اسمبلی معمولا سریع تر و کم حجم تر است. هم چنین گاهی نیاز است که یک دسترسی سطح سیستم داشته باشیم در آن صورت کد زبان c به تنهایی کافی نیست.

در حقیقت کاری که انجام می شود به این صورت است که bootasm.s پردازنده را به حالت حفاظت شده 32 بیتی می برد و در آنجا فایل bootmain.c صدا زده می شود. این انتقال نمونه ای از کار هایی است که برای اجرای آن ها به دسترسی سطح سیستم نیاز است.

14-

ثبات عام منظوره : در vx6 تعداد 8 عدد ثبات عام منظوره وجود دارد. نام همگی این ثبات ها به دلیل 32 بیته بودن با حرف e که مخفف extended است شروع می شود.

تمام 8 عدد ثبات عام منظوره :

Eip, eax, ebx, ecx , edx , esi , ebp , esp

وظیفه ثبات عام منظوره نگهداری حاصل عملیات ریاضی ، پوینتر ها و برخی داده ها است.

ثبات قطعه : در xv6 تعداد 6 عدد ثبات قطعه وجود دارد. به طور ی کلی وظیفه این نوع ثبات نگهداری آدرس استک ، دیتا و کد است.

ثبات های مشهور این دسته SS, DS , CS هستند که به ترتیب وظایق گفته شده یعنی نگهداری پوینتر به استک ، پوینتر به دیتا و پوینتر به کد را نگهداری می کننند.

ثبات وضعیت : وظیفه این نوع ثبات ها نگهداری وضعیت پردازنده است. EFLAGS یک نمونه ثبات وضعیت است که در آن flag های مختلف نگهداری می شوند( فلگ هایی نظیر zero, sign , carry )

ثبات کنترلی : وظیفه این ثبات ها کنترل cpu و دستگاه های دیجیتا دیگر است.

cr0, cr2, cr3, cr4 نمونه هایی از این ثبات ها هستند که وظایفشان به ترتیب تغییر مدل آدرس دهی، کنترل interrupt ، کنترل paging و هم پردازنده‌ها است.

15-

در ابتدا توضیحاتی درباره real mode ارائه می دهیم:

این مود بر اساس پردازنده های 8086 و 8088 است.این مود در حقیقت مود دستور های 16 بیتی است به همین دلیل تمام برنامه ها در real mode باید شامل دستور های 16 بیتی باشند. در این مود هیچ protection وجود ندارد به همین دلیل این مود نمی تواند از overwtrite شدن برنامه ها جلوگیری کنند، به همین دلیل این مود multi-tsaking نیست.

استفاده از real mode در پردازنده ها در حقیقت یک دلیل تاریخی دارد. تمام پردازنده های x86 در حالتی که وانمود می کنند پردازنده x8086 هستند شروع به کار می کنند.( پردازنده ای 16 بیتی که توانایی دسترسی فقط به یک بیت از مموری را دارد.) بعد از آن مسئولیت BIOS است که مشخص کند که پردازنده واقعا از چه نوعی است.

این الگو در حقیقت یک الگو رایج است که کامپیوتر ها با یک تکه کد ساده شروع به کار می کند و سپس با سنجیدن شرایط و ارتباط برقرار کردن با بقیه سیستم ها ، به روی مود های پیشرفته تر شیفت میدهند.

نقص اصلی real mode: از جمله نقص های این سیستم این است که هیچ مکانیزم امنیتی یا حفاظتی در این مود وجود ندارد از برنامه های پر اشتباه و مخرب جلوگیری کند.

16–

مموری در real mode یک توالی از خطی از بایت هاست که می تواند آزادانه آدرس دهی شوند با هر آدرس 20 بیتی که از 16 بیت آدرس segment و 4 بیت آدرس offset تشکیل شده باشد. برنامه می تواند به هر نقطه از مموری دسترسی پیدا کند و در آن بخواند و بنویسید فارغ از این که در چه بخشی از حافظه قرار دارد.

نهایتا آدرس ما به صورت زیر محسابه می شود :

PhysicalAddress = Segment \* 16 + Offset

18-

کد ورود به هسته (معادل (entry.sدر معماری x86 لینوکس برای 32بیت و 64بیت در لینک زیر قرار دارد.

<https://github.com/torvalds/linux/tree/master/arch/x86/entry>

**اجرای هسته xv6**

19-

این آدرس نباید مجازی باشد زیرا برای دسترسی به آن باید آدرس مجازی اش را به آدرس فیزیکی تبدیل کنیم و اینکار با استفاده از این جدول امکان پذیر است اما از آنجایی که آدرس فیزیکی این جدول را نداریم وارد یک حلقه بی نهایت برای پیدا کردن آدرس فیزیکی میشویم و در نتیجه نمیتوانیم به این جدول دسترسی پیدا کنیم.در نهایت میتوان نتیجه گرفت که آدرس جدول نگاشت باید به صورت فیزیکی (در رجیستر cr3) ذخیره شود.

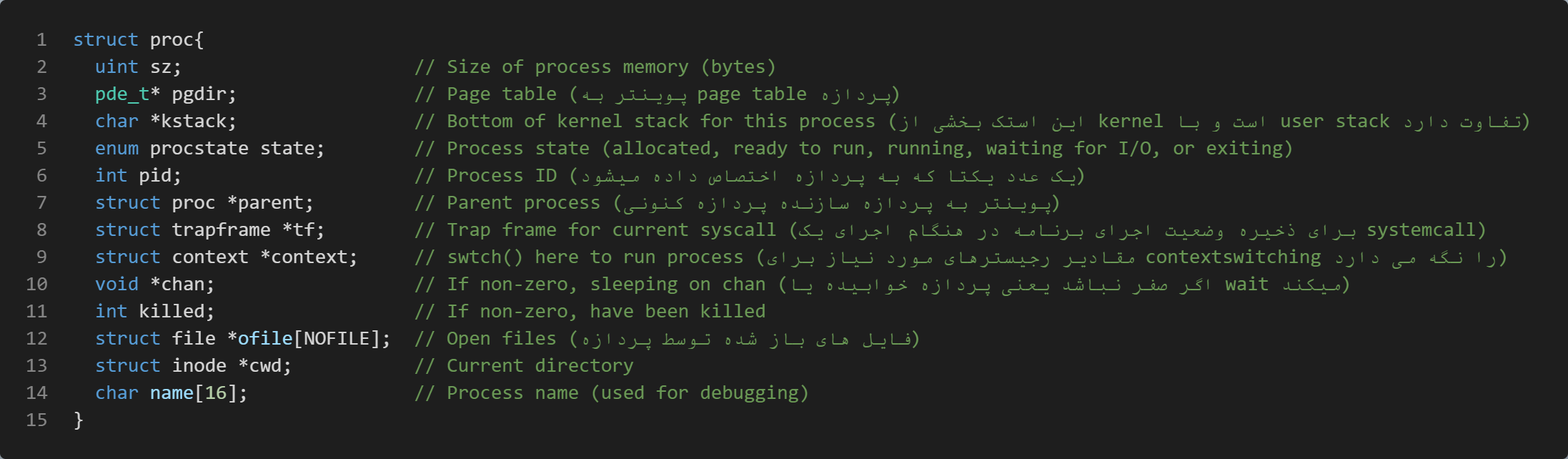
22-

SS وDS ، segment های مختلفی هستند که لینوکس از descriptor های یکسانی برای آن‌ها استفاده می‌کند. اما ما نیاز داریم که descriptor ها را برای user mode و kernel mode تفکیک کنیم. در اینجا از SEG\_USER استفاده می شود که دستورات user-side از kernel-side تمایز یابند.

**اجرای نخستین برنامه سطح کاربر**

23-

این struct وضعیت هر پردازه را ذخیره میکند که در فایل h.proc تعریف شده و دارای متغیر های زیر است:



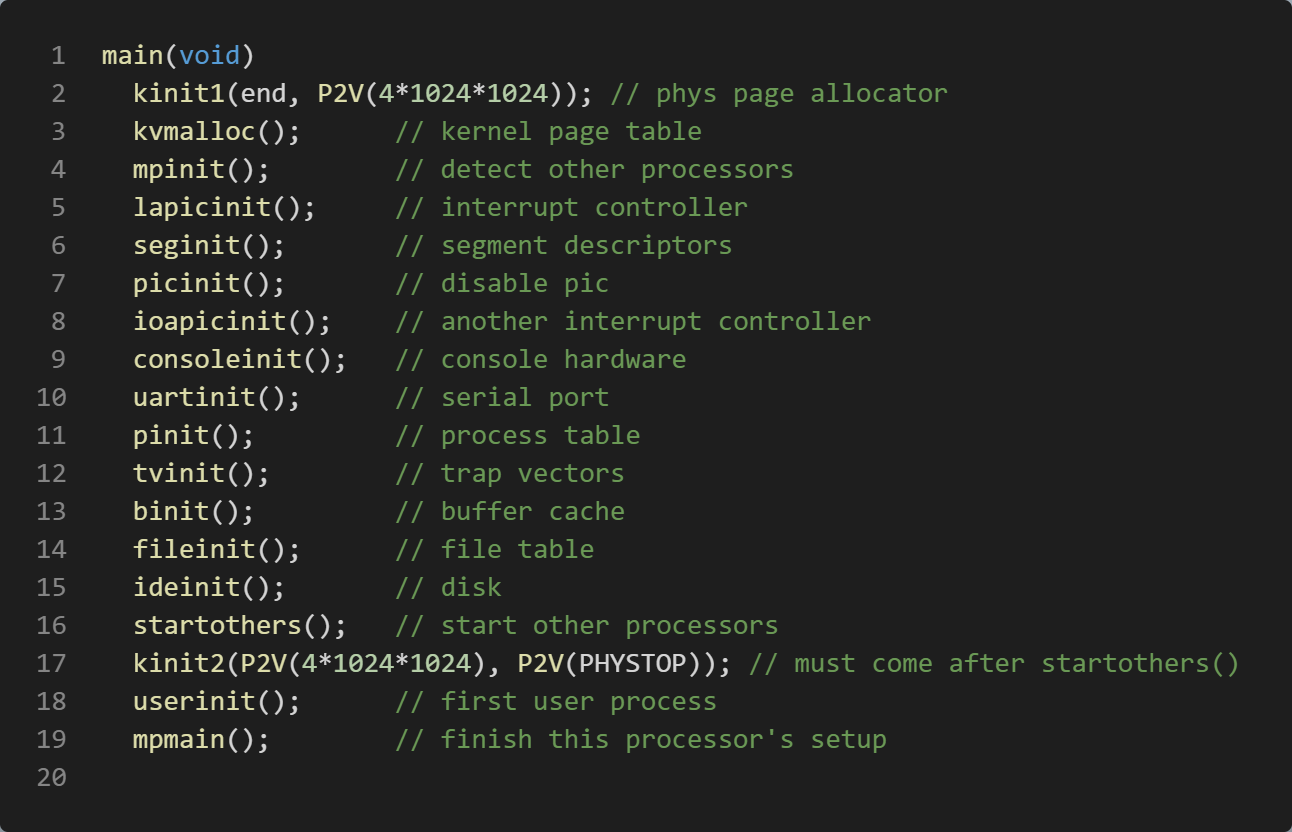
معادل این struct در لینوکس در لینک زیر و در استراکت task\_struct قرار دارد:

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

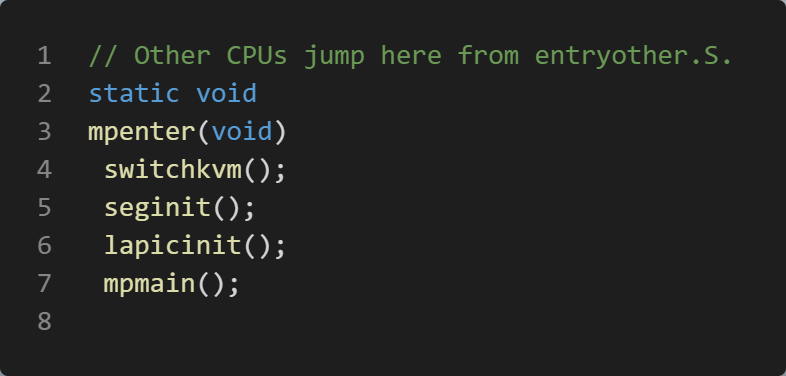
(Each element in the task list is a process descriptor of the type struct task\_struct, which is defined in <linux/sched. h>. The process descriptor contains all the information about a specific process.)

27-

هسته اول فرآیند بوت را انجام می دهد و توسط کد entry.s وارد تابع main می شود. توابع مورد نیاز برای آماده سازی سیستم در این تابع فراخوانده شده اند و توسط این هسته اجرا می شوند.



هسته های دیگر از طریق کد entryother.sوارد تابع mpenter می‌شوند. با توجه به اینکه این تابع در تابع main نیز فراخوانی میشود می توان گفت این 4 تابع بین تمامی هسته ها مشترک خواهند بود.



از موارد اختصاصی هسته اول می‌توان به تابع kvmalloc اشاره کرد که (با توجه به کد kpgdir=setupkvm();) یک page table ایجاد می‌کند که این مورد توسط هسته اول انجام می‌پذیرد یا تابع startothers که فقط پردازنده اول نیاز است بقیه پردازنده ها را start کند.

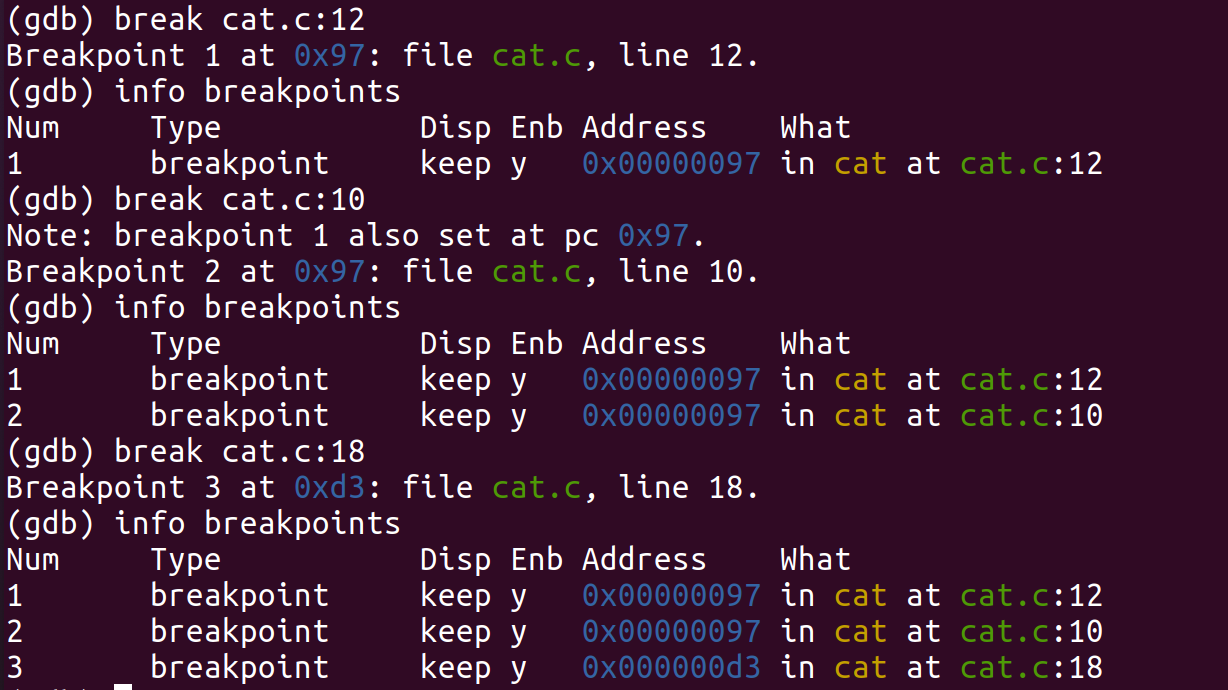
از توابع مشترک میتوان به mpmain اشاره کرد زیرا همه پردازنده ها باید کار خود را شروع کنند و آماده اجرای برنامه ها شوند که این مورد توسط این تابع انجام می پذیرد یا تابع switchkvmزیرا همه پردازنده ها باید آدرس table page که توسط پردازنده اول ایجاد شده را در رجیستر خود ذخیره کنند در نتیجه این تابع بین همه آن ها مشترک است.

زمانبند توسط تابع scheduler انجام می پذیرد در تابع mpmain صدا زده می‌شود که این تابع بین تمامی هسته ها مشترک است.

**اشکال زدایی:**



برای مشاهده breakpoint ها از دستور Info break points استفاده می کنیم.

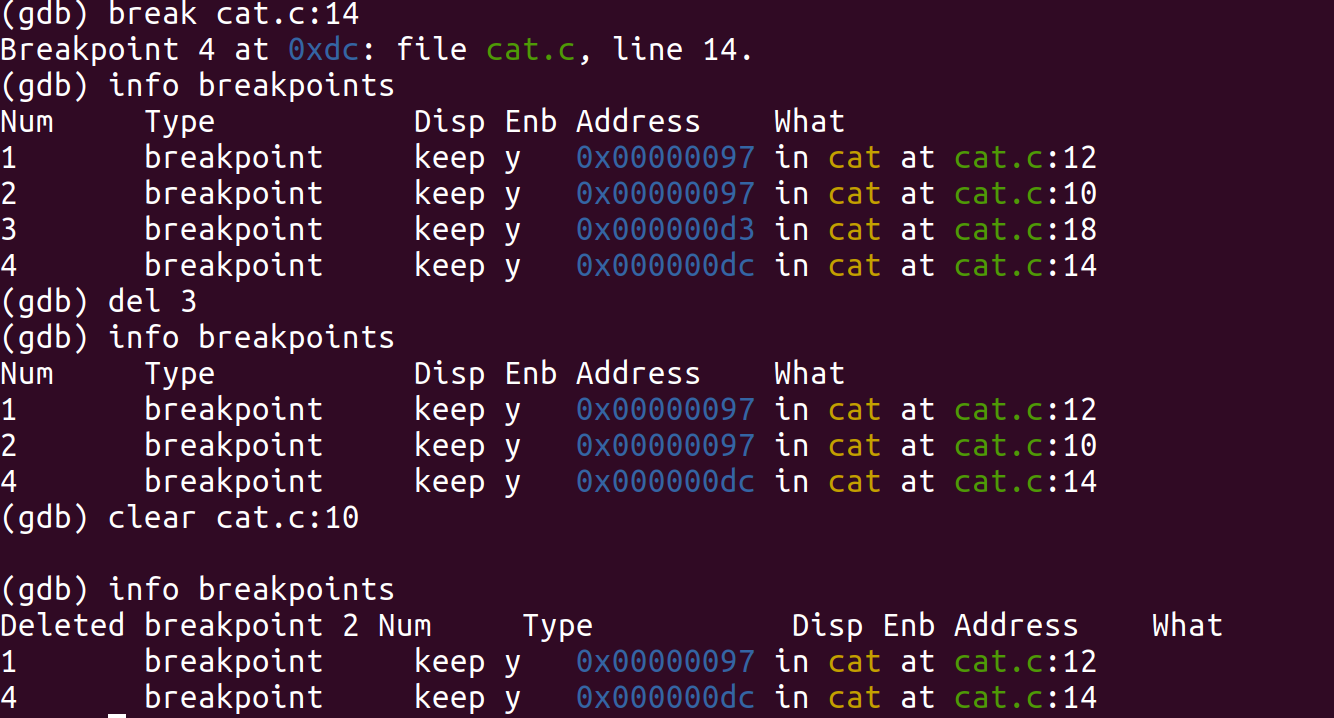




برای حذف break point ها از دو روش می توان استفاده کرد:

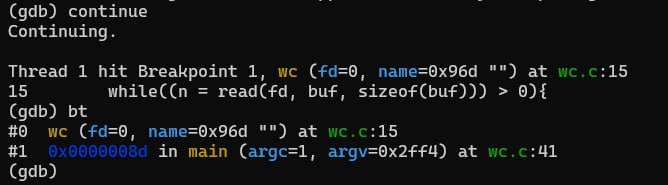
روش اول: استفاده از Del breakpoint\_number

روش دوم: استفاده از Clear file\_name:line



3-

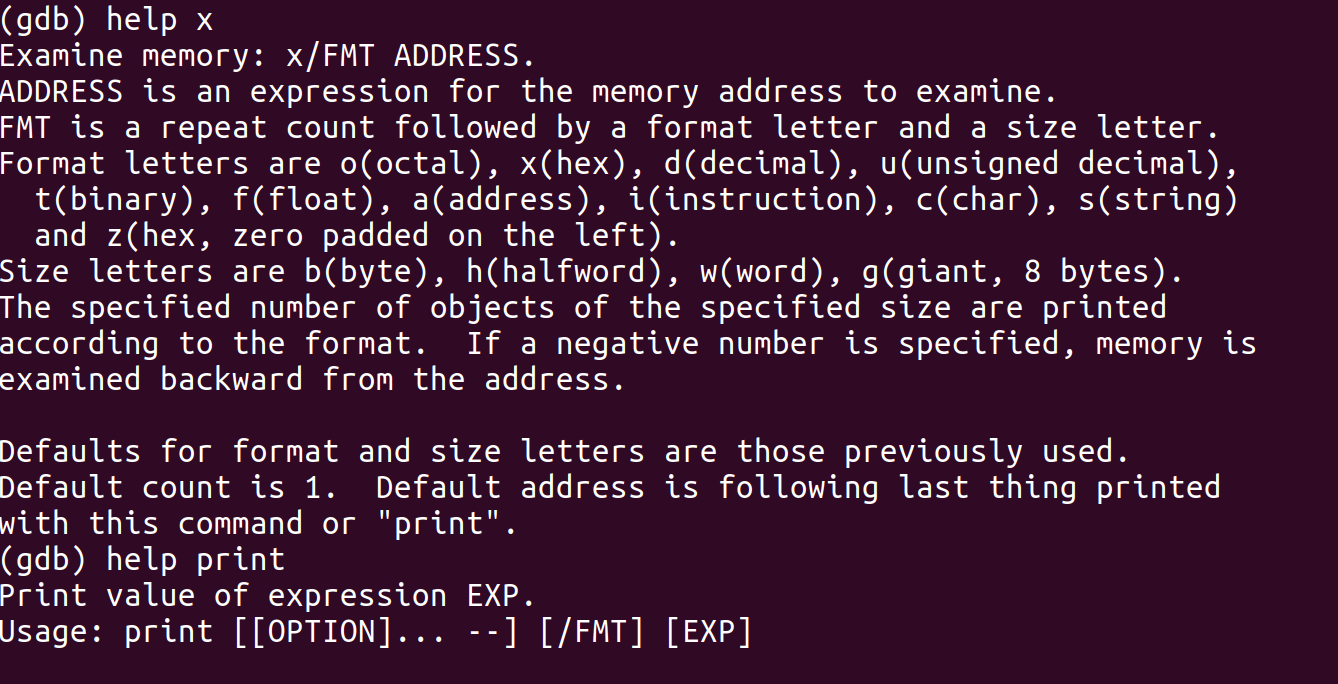
با اجرای bt سلسله از توابعی که تاکنون فراخوانی شده اند و در استک اضافه شده اند نمایش داده می شود. در حقیقت bt مخفف backtrace است.

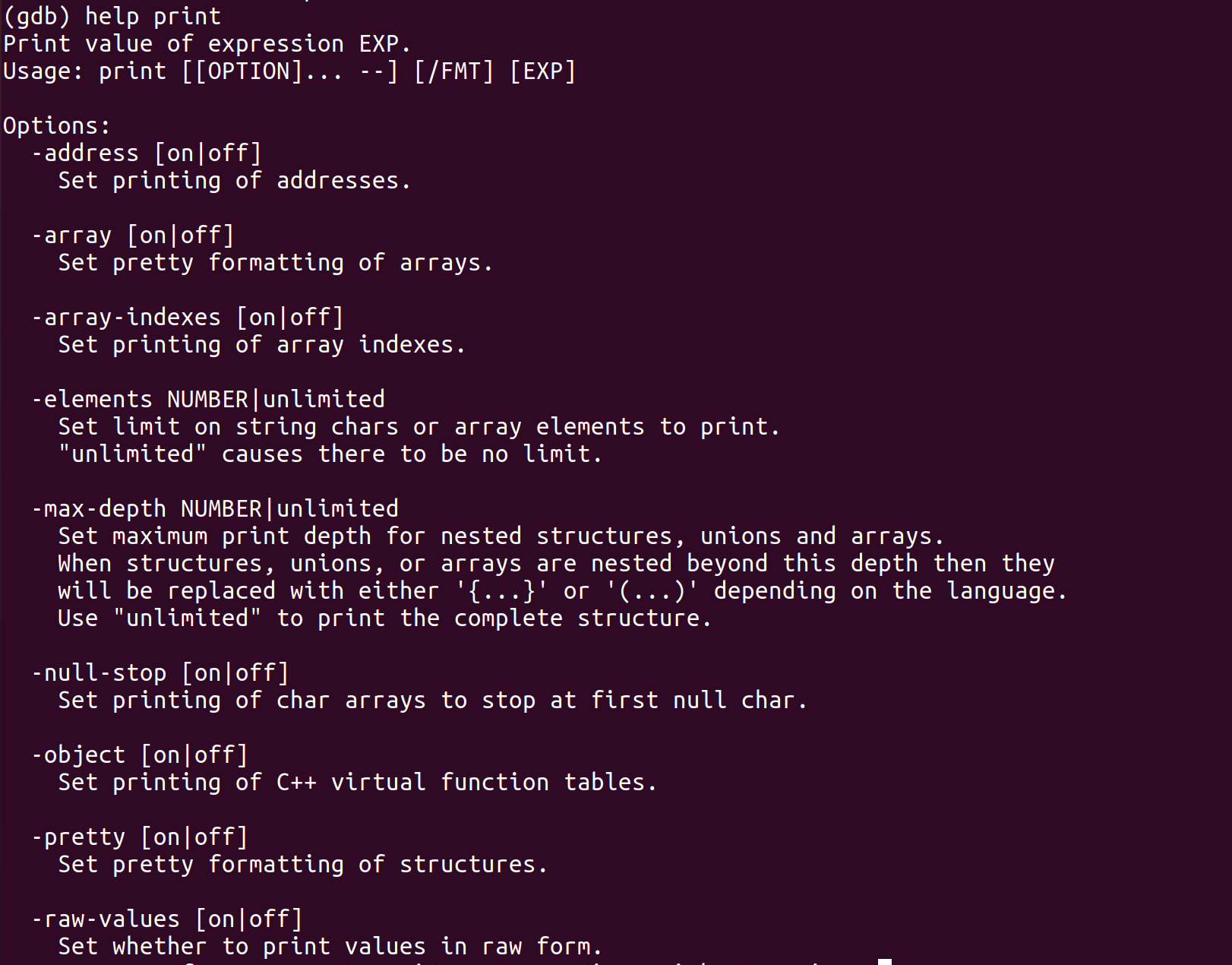


4 –

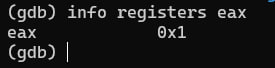
همان طور که در تصاویر ها مشاهده می کنیم ، نحوه دریافت ورودی این دو دستور با هم فرق می‌کند.

دستور x آدرس می گیرد اما دستورprint ، expressionمی‌گیرد. همچنین نحوه نمایش اطلاعات آن‌ها و موارد مختلفی که می توانند نمایش دهند با یکدیگر نیز متفاوت است.



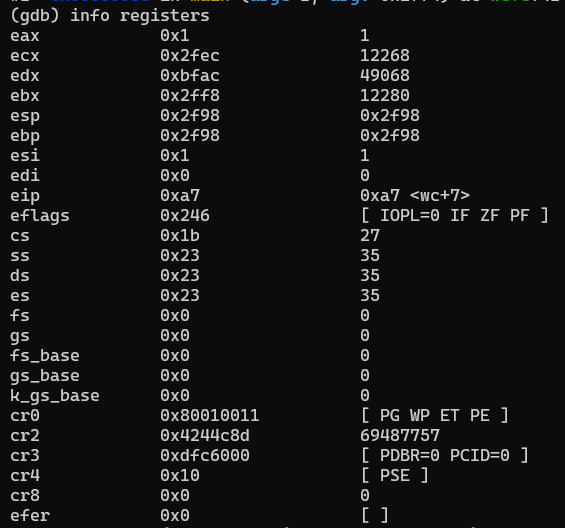


هم چنین با استفاده از دستور Info registers register\_name می‌توان یک ثبات خاص را نمایش داد:

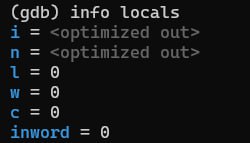


5-

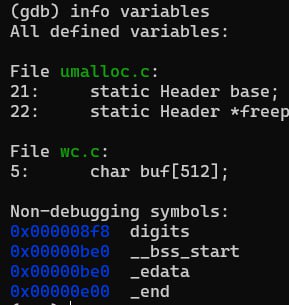
برای نمایش وضعیت ثبات ها از دستور info registers استفاده می شود:



برای وضعیت متغیر های محلی از دستور info locals استفاده می کنند.



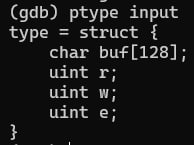
هم چنین اگر بخواهیم وضعیت تمام متغیر‌ها را ببینیم یعنی حتی غیر محلی ها از دستور info variables استفاده میکنیم.



رجیستر های esi , edi هر دو رجیستر های عام منظوره هستند. اما معمولا esi برای نگهداری آدرس source و edi برای نگهداری آدرس destination در زمان عملیات کپی استفاده می‌شوند.

6-

با استفاده از دستور ptype input اطلاعات استراکت input را از gdb نمایش می‌دهیم.



متغیرهای استراکت :

1. Buf: آرایه ای از کاراکتر هاست که ظرفیت آن 128 کاراکتر است. ورودی که در ترمینال می‌نویسیم (به صورت رشته) در این آرایه ذخیره می‌شود.
2. R: ایندکسی از بافر را نشان میدهد. در حقیقت آخرین جایی است که خوانده ایم.
3. w: ایندکسی از بافر که محل شروع نوشتن در خط ورودی buf است.
4. e: ایندکسی از بافر که محل کنونی cursor را نگه می‌دارد.

تمام زمان هایی که مکان cursor تغییر می کند مثل زمان هایی که در ترمینال چیزی تایپ می کنیم input.e تغییر می کند.

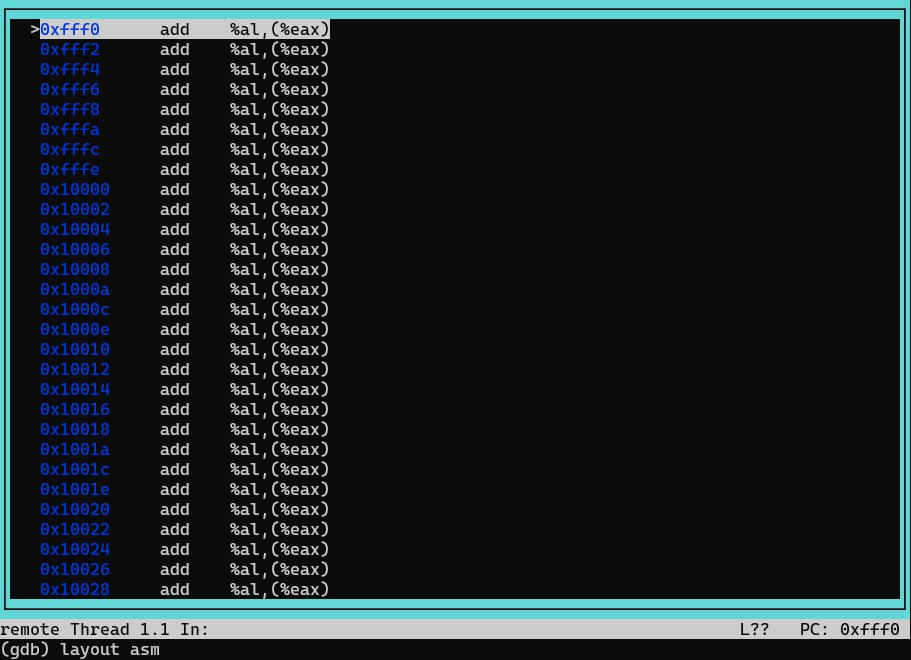
همچنین زمانی که چیزی می نویسم input.w به input.r می رسد.

7–

خروجی دستور ها:

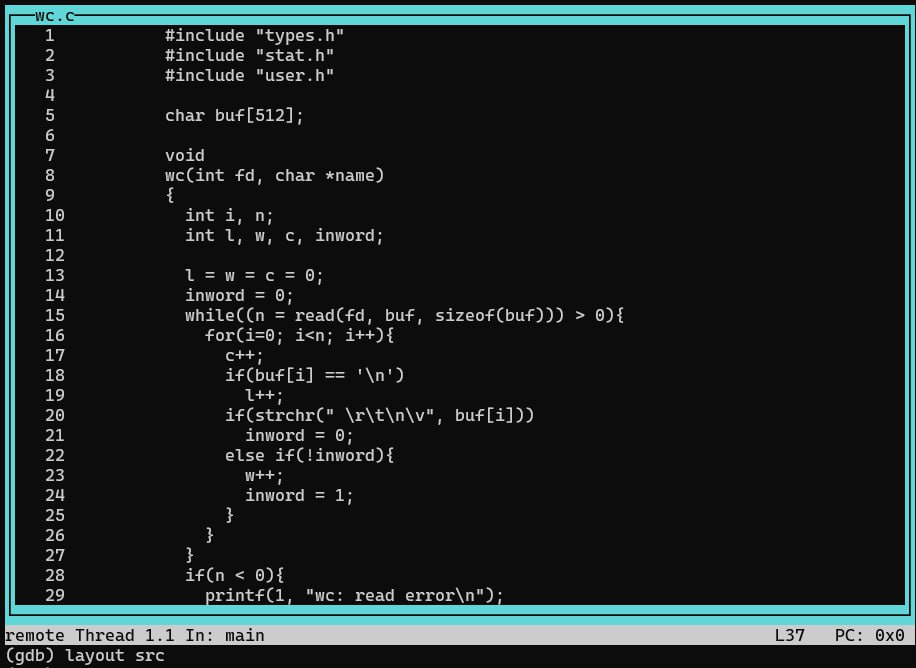
**layout asm:**

در این حالت می توان برنامه را به صورت کد اسمبلی دید.



**layout src:**

برنامه را در حالت کد سورسش نشان می دهد.



8–

برای جا به جاییی میان توابع از دستورات زیر استفاده می کنند:

1. دستور up n: با این دستور می توان به تعداد n، frameبه سمت بالای استک رفت. (اگر n را مقدار دهی نکنیم با n=1 دستور را اجرا می کند)
2. دستور down n: با این دستور می توان به تعدادn ، frameبه سمت پایین استک رفت. (اگر n را مقدار دهی نکنیم با n=1 دستور را اجرا می کند)
3. دستور frame[frame\_selection\_spec] :

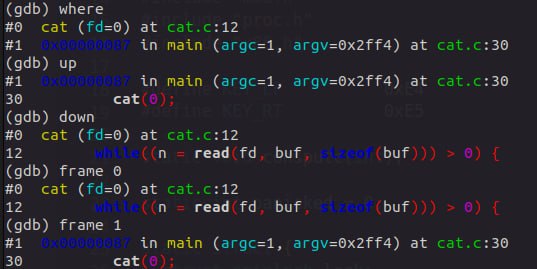
به جای frame\_selection\_spec مواردی مختلفی می توانیم قرار دهیم:

Function\_name: فریم استک متعلق به آن تابع را پیدا می کند و نمایش میدهد. اگر چند فریم متعلق به تابع باشد، داخلی ترین آن ها انتخاب می شود.

Level: منظور شماره فریمی از استک است که میخواهیم به آن برویم.

Stack\_address: آدرس فریمی از استک است که میخواهیم به آن برویم.

نمونه ای از استفاده از این دستورات در زیر نمایش داده شده است :

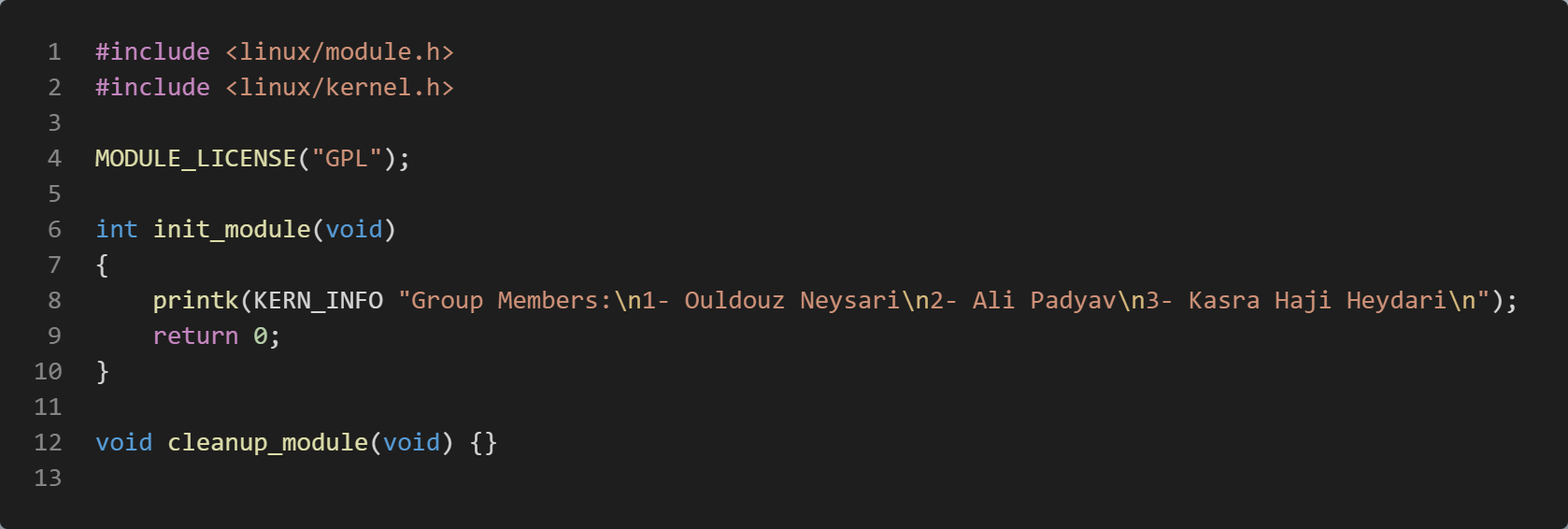


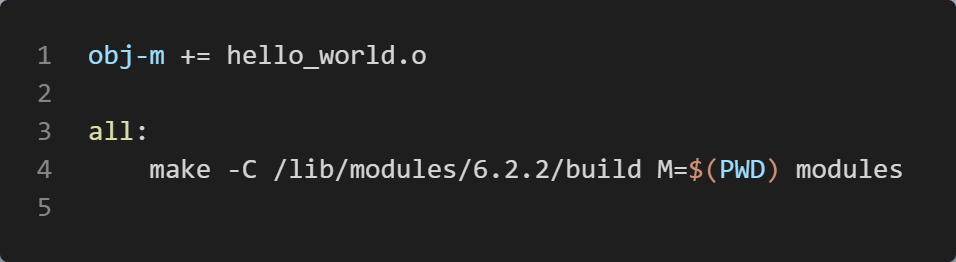
**بخش امتیازی (لینوکس)**

با دستور uname -a ورژن کرنل لینوکس نشان داده می‌شود.



برای نمایش اسم اعضای گروه یک فایل c و یک makefile نوشتیم:





با دستور make فایل hello\_world.ko ساخته میشود.

سپس با دستور sudo insmod hello\_world.ko و dmesg خروجی را مشاهد می‌کنیم:

