**به نام خدا**

گزارشکار آزمایشگاه یک آزمایشگاه سیستم عامل

اعضای گروه:

1. مهراد لیویان 810101501
2. بهراد بینایی حقیقی 810101392
3. مرضیه موسوی کانی 810101526

**آشنایی با سیستم عامل xv6**

1. **سیستم عامل xv6 یک مدل از پیاده سازی از سیستم عامل UNIX برای سیستم های چند پردازنده x86 و RISC\_V است. معماری این سیستم عامل از معماری UNIX پیروی میکند و میتوان به سادگی ساختار، پشتیبانی ازmultiprocess و مدیریت حافظه ساده اشاره کرد. معماری این سیستم عامل با وجود سادگی ،بسیاری از مفاهیم پیشرفته سیستم عامل را پوشش میدهد. این سیستم در واقع یک شبه یونیکس (UNIX Like ) است. و دسته بندی فایل های این سیستم عامل نیز مثل یونیکس calls system و program level user است. این سیستم عامل با AMSIC و برای multiprocessor x86 و سیستم های RISC\_V طراحی شده است؛ از دلایل این میتوان به وجود فایل asm.h که درآن از معماری 86x استفاده شده اشاره کرد.**
2. **در سیستم‌عامل XV6، پردازنده‌ها به وسیله‌ی scheduler به process های مختلف اختصاص می‌یابند. هر پردازنده وظیفه دارد به نوبت ، process ها را اجرا کند. بخش‌های تشکیل دهنده یک پردازنده در سیستم‌عامل XV6 شامل فضای حافظه کاربر از جمله دستورات ،داده ها و استک و وضعیت هر پردازنده که به صورت خصوصی در اختیار kernel قرار دارد میشود.**

**این سیستم عامل به شکل time sharing عمل میکند و در هر زمان CPU را در اختیار یک process قرار میدهد تا اجرا شود. و زمانی که یک process اجرا نمی شود سیستم عامل محتوای آن را در رجیستر CPU ذخیره میکند . اگر برنامه ای در حال اجرا باشد اما مقدار زمان اختصاص داده شده به آن تمام شود سیستم عامل محتوای رجیستر های این فرایند را در حافظه ذخیره میکند و CPU را در اختیار process بعدی قرار می دهد. و زمانی که فرایند بعدی اجرا شد محتوا مموری به رجیستر ها برگردانده میشود.**

1. **در سیستم** عاملهای مبتنی بر یونیکس **File descriptor یک عدد صحیح کوچک است که نشان دهنده object مدیریت شده توسط kernel است که یک فرایند در آن ممکن است بنویسد و یا از آن بخواند. File descriptor اطلاعاتی درباره فایل باز مانند نوع آن ، دسترسی های مجاز مثل مثلا خواندن و نوشتن و غیره و مکان فعلی را میدهد. عملکرد pipe به منظور ارتباط بین فرآیندها استفاده می‌شود و امکان انتقال داده‌ها بین دو فرآیند را فراهم می‌کند.**
2. تابع fork برای ایجاد یک process جدید استفاده می شود. این تابع یک نسخه کپی از پردازه ای می سازد که این تابع را صدا زده است. یعنی دیتا و دستورات پردازه فعلی در حافظه پردازه جدید (child)کپی می شوند. با وجود اینکه در لحظه ایجاد پردازه فرزند، داده های آن با پردازه پدر یکسان هستند، اما در واقع این دو پردازه حافظه جداگانه ای خواهند داشت و تغییر یک متغیر در پردازه پدر، آن متغیر در پردازه فرزند را تغییر نمی دهد. پردازه پدر پس از ایجاد پردازه فرزند، به caller تابع fork بازمی گردد که امکان اجرای همزمان دو پردازه را فراهم می سازد. 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 مقدار خروجی این تابع عدد 0 خواهد بود. پس اگر با استفاده از کد fork() = pid یک پردازه جدید درست کنیم، 3 حالت داریم:

|  |  |
| --- | --- |
| :0 = pid | در پردازه فرزند هستیم. |
| 0 < pid | در پردازه پدر هستیم و مقدار pid در واقع آی دی پردازه فرزند است. |
| 0 > pid | در زمان اجرای تابع fork و پردازه جدید اروری وجود داشته و پردازه فرزند ایجاد نشده است |

تابع exec حافظه پردازه فعلی را با یک حافظه جدید که در آن یک برنامه با فایل ELF لود شده است، جایگزین می کند. exec راهی برای اجرای یک برنامه در پردازه فعلی است. بر خالف تابع fork ، برنامه به caller تابع exec باز نمی گردد و برنامه جدید اجرا می شود مگر اینکه در زمان اجرای این تابع یک ارور رخ دهد. برنامه جدید اجرا شده در یک نقطه ای با استفاده از تابع exit اجرای پردازه را خاتمه می دهد.

مزیت ادغام نکردن این دو تابع در زمان redirection O/I خودش را نشان می دهد. زمانی که کاربر در shell یک برنامه را اجرا می کند، کاری که در پشت صحنه توسط shell انجام می شود به شرح زیر است:

* ابتدا دستور تایپ شده توسط کاربر در ترمینال خوانده میشود.
* با استفاده از تابع fork یک پردازه جدید ایجاد می کند.
* در پردازه فرزند با استفاده از تابع exec برنامه کاربر را جایگزین پردازه فعلی می کند.
* پس از اتمام پردازه فرزند به main باز می گردد و منتظر دستور جدید می شود.

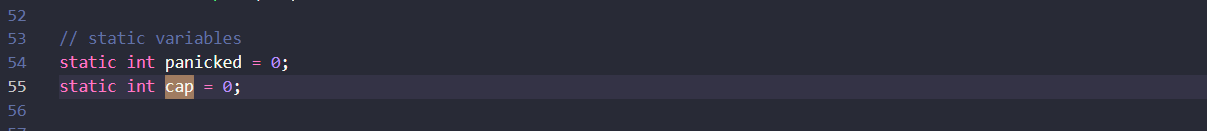
شرح پروژه :

1. در این قسمت برای تغییر curser به سمت چپ و راست در تابع consoleintr دو case جدید اضافه کردیم که 0xe4 مربوط به arrowkey چپ و oxe5 مربوط به arrowkey راست است.

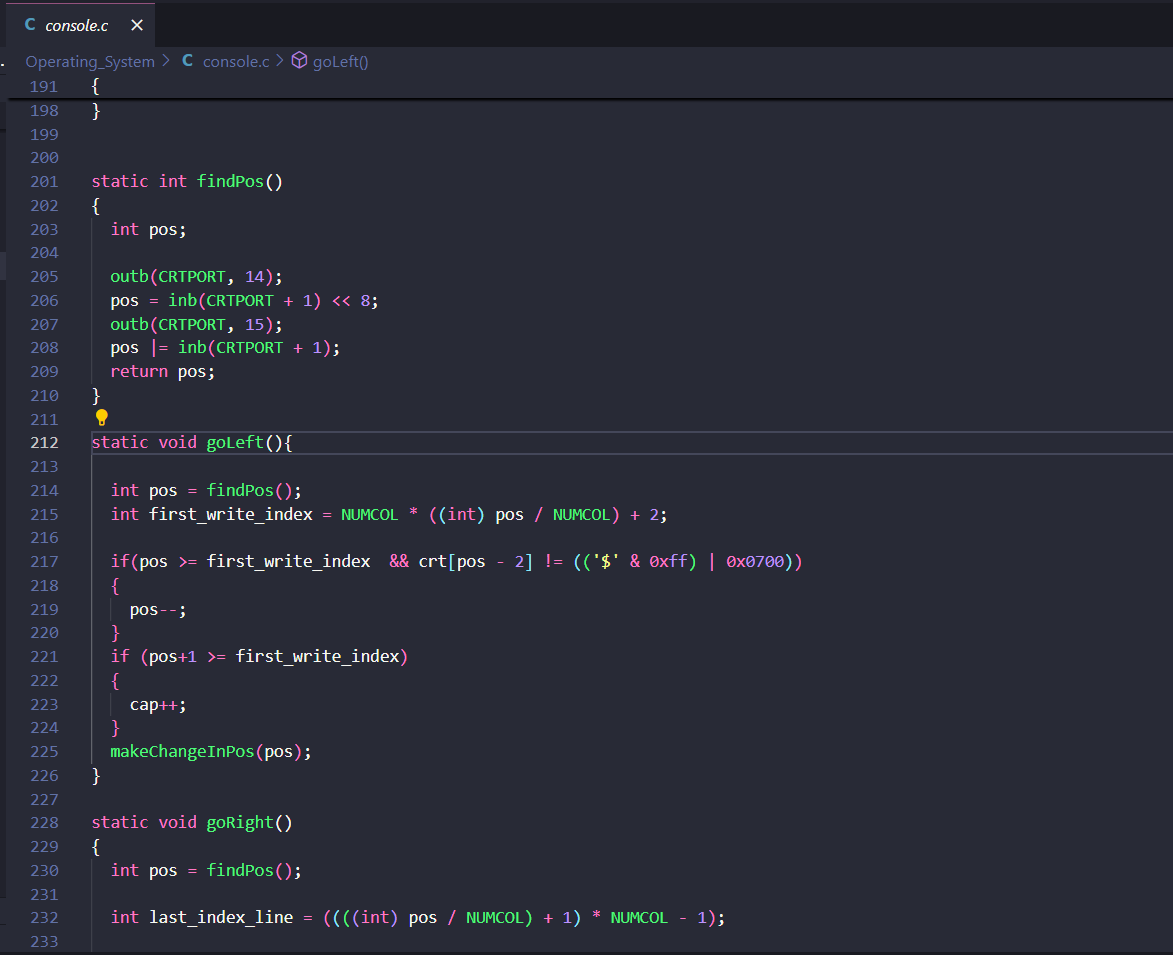
|  |  |
| --- | --- |
|  |  |

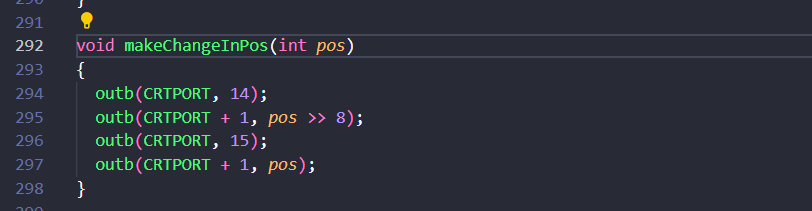
بعد از detect کردن arrow key ها حالا تابع را صدا میزنیم که goleft و goright هست. (قبل از صدا زدن goleft ابتدا چک می کنیم که از بازه مجاز خارج نشود.)

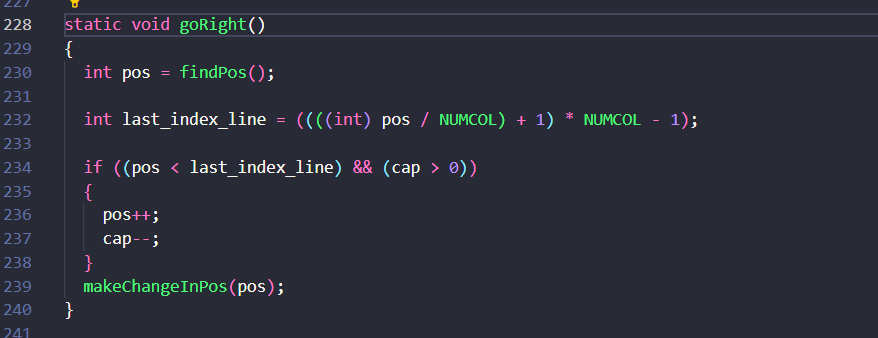
در این بخش یک متغیر به نام cap تعریف می کنیم که نشان دهنده تعدادی هست که از ته متن نوشته شده در کنسول فاصله گرفتیم.

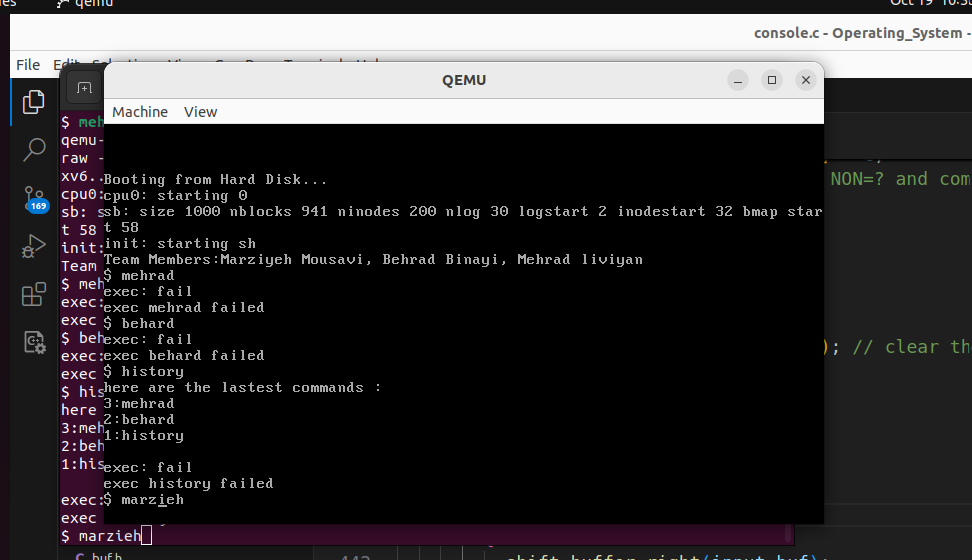


در این دو تابع ابتدا pos را میگیریم سپس تغییرات مناسب در آن را انجام می دهیم سپس pos را دوباره تعیین می کنیم.( گرفتن pos با findpos و تغییر در آن با changepos است .)

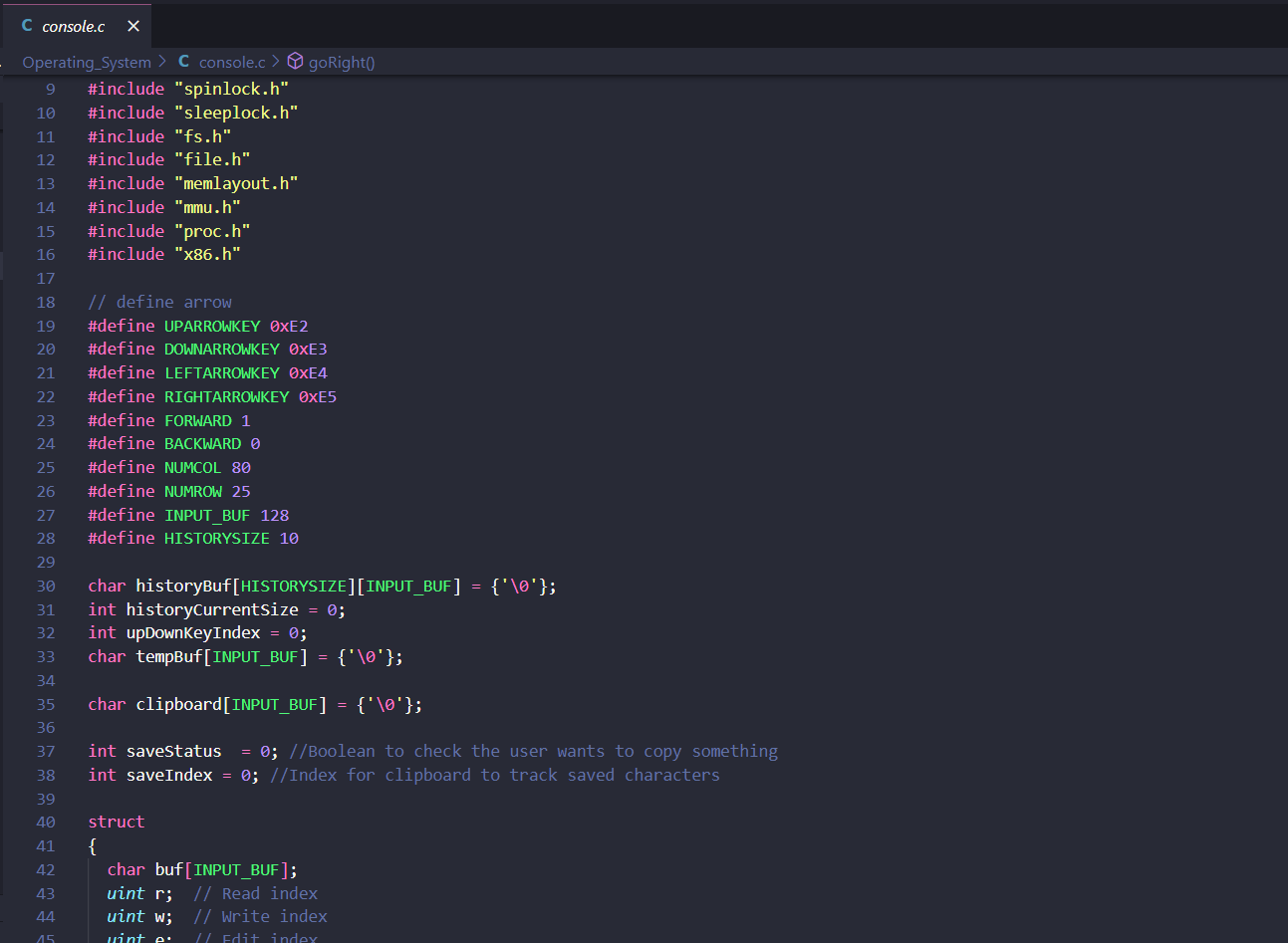








1. برای اضافه کردن history ابتدا یک آرایه با ابعاد 10 در 128 به نام historyBuf درست می کنیم.

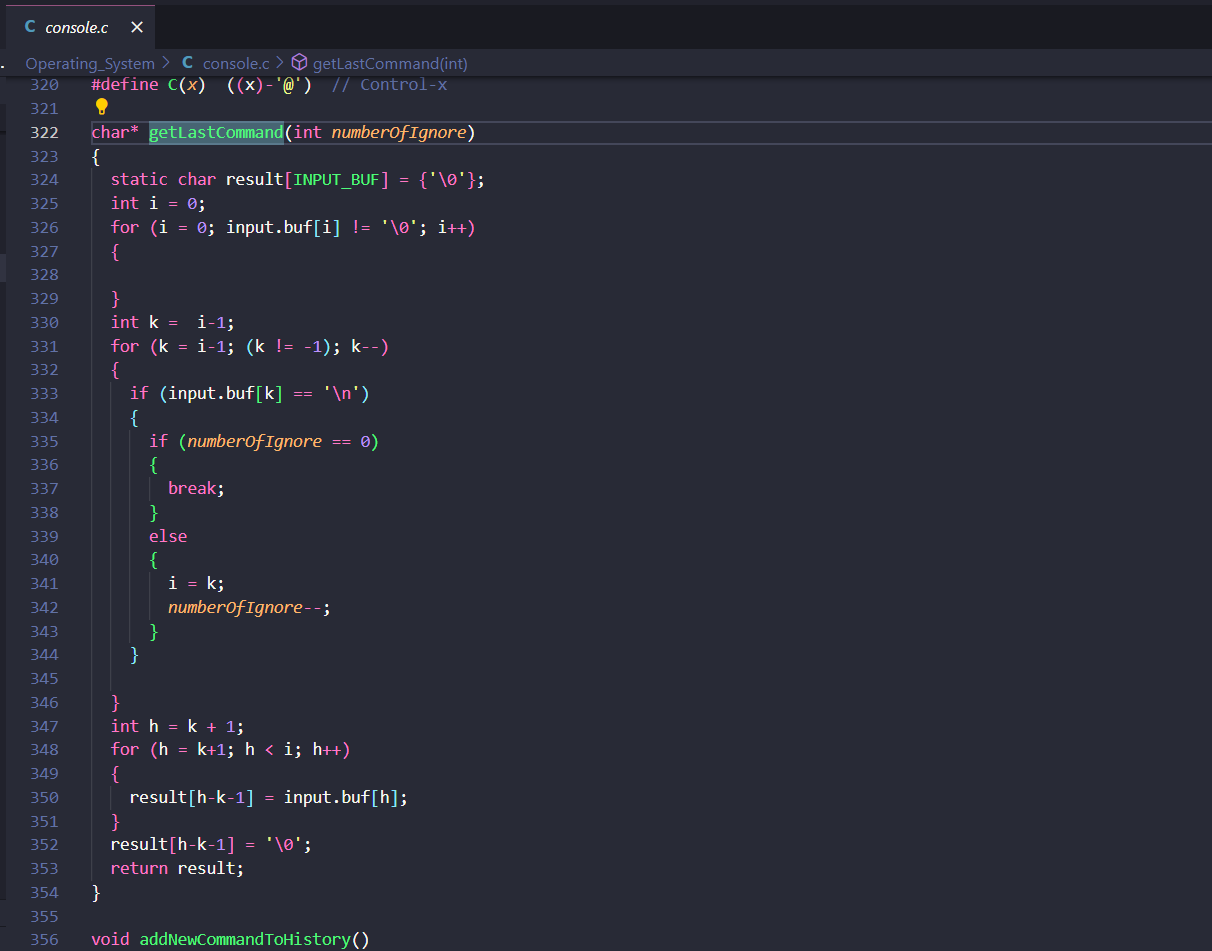


بعد از وارد شدن هر کامند و زده شدن \n در قسمت default در تابع " consoleintr" تابع addNewCommandToHistory را صدا میزنیم.

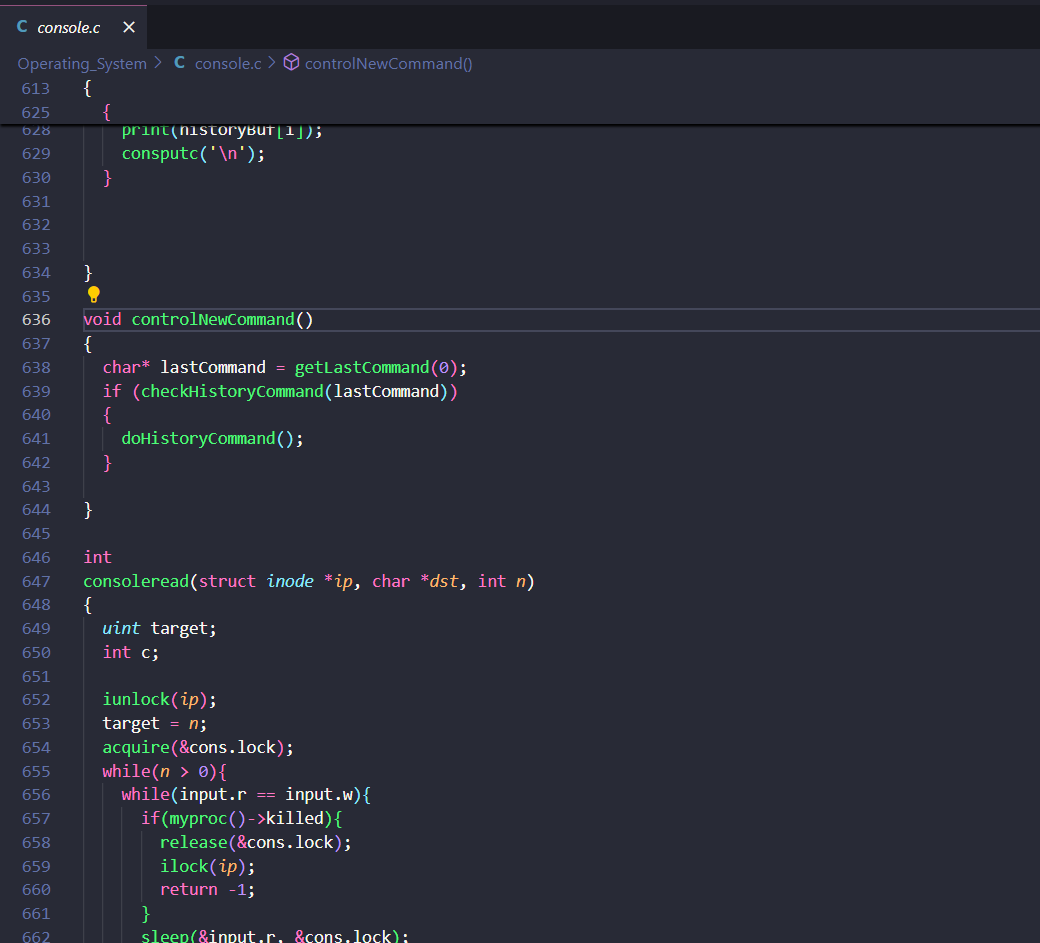
این تابع کاری که می کند این است که ابتدا تمام history های قبلی را یکی به عقب تر میفرستد سپس با استفاده از تابع getLastCommand آخرین کامند وارد شده ( که قبل از وارد شدن enter هست ) را میگیریم و در historyBuf[0] قرار می دهیم.

|  |  |
| --- | --- |
|  |  |

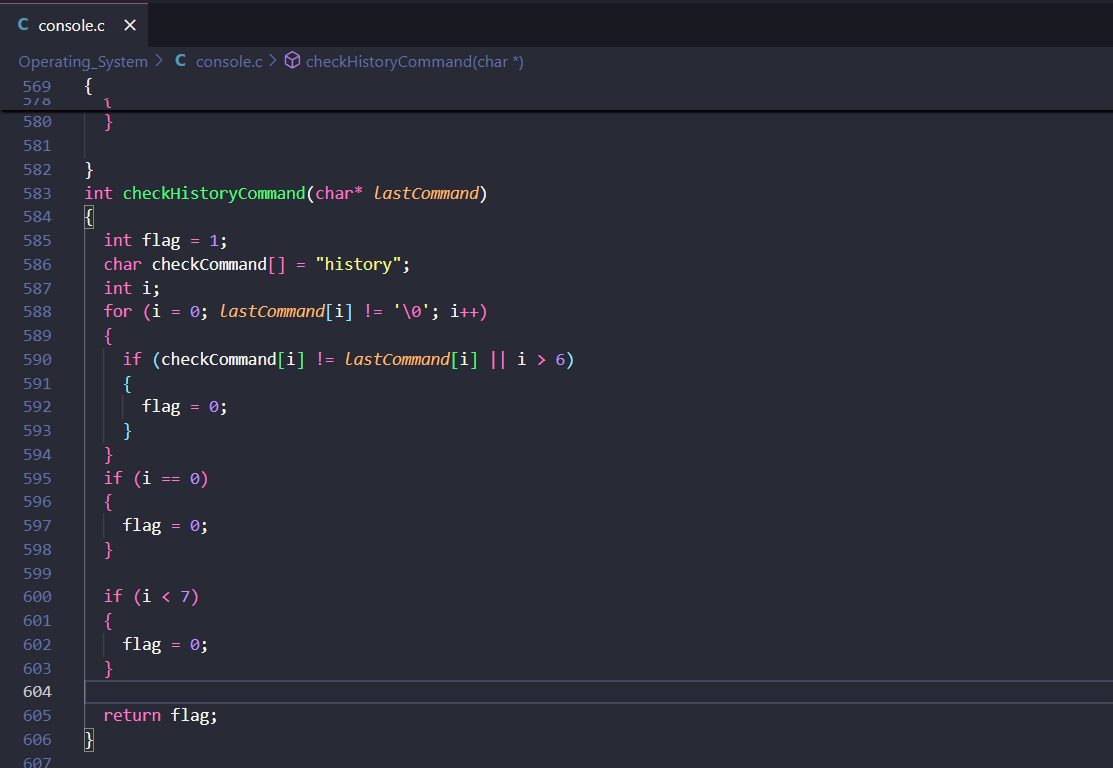
در تابع getLastCommand به این صورت عمل میکنیم که ایندکس آخر buffer را یعنی جایی که \0 می شود را به دست می آوریم و سپس ایندکس \n قبل از آن را بدست می اوریم سپس کاراکتر های بین این دو ایندکس را برمیگردانیم.



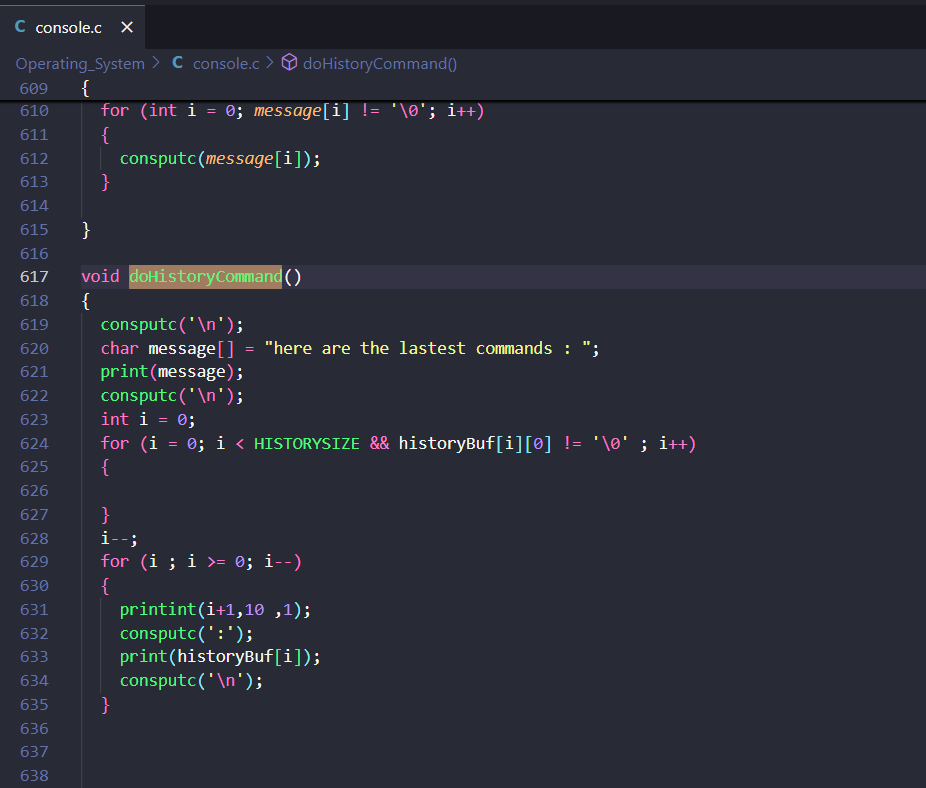
بعد از این کار به سراغ تابع controllnewCommand می رویم تا اگر آخرین دستور وارد شده history بود آن ده کامند قبلی ( اگر کمتر موجود بود که کمتر ) را چاپ کند. برای چک کردن آن که آخرین کامند history بوده یا نه از تابع checkHistoryCommand استفاده می کنیم. سپس با یک if اگر برابر بود تابع doHistoryCommand را صدا میزنیم که صرفا وظیفه چاپ کردن را دارد.



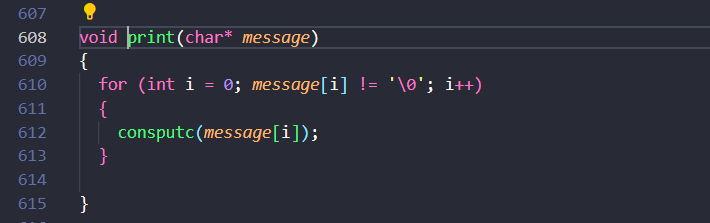
در تابع checkHistoryCommand فقط یک مقایسه ساده بین دو آرایه انجام می دهیم :



در تابع doHistoryCommand صرفا داریم مقادیر آن ها را چاپ می کنیم .



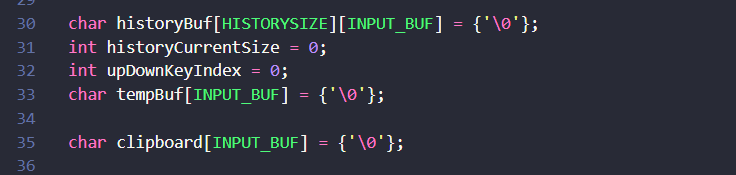
تابع پرینت که نوشتیم یک تابع هست که با استفاده از consputc کاراکتر ها را چاپ می کند.



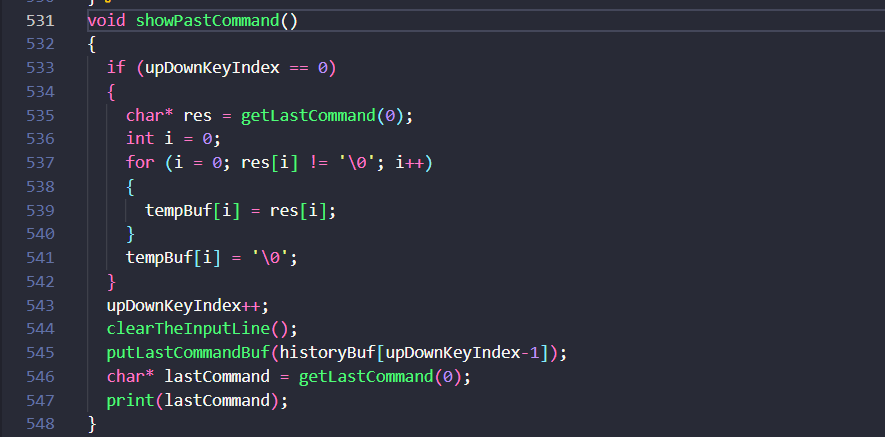
سپس برای این بخش باید در تابع consoleintr دو case جدید اضافه کنیم برای تشخیص arrow key بالا و پایین که 0xe2 برای بالا و 0xe3 برای پایین است.

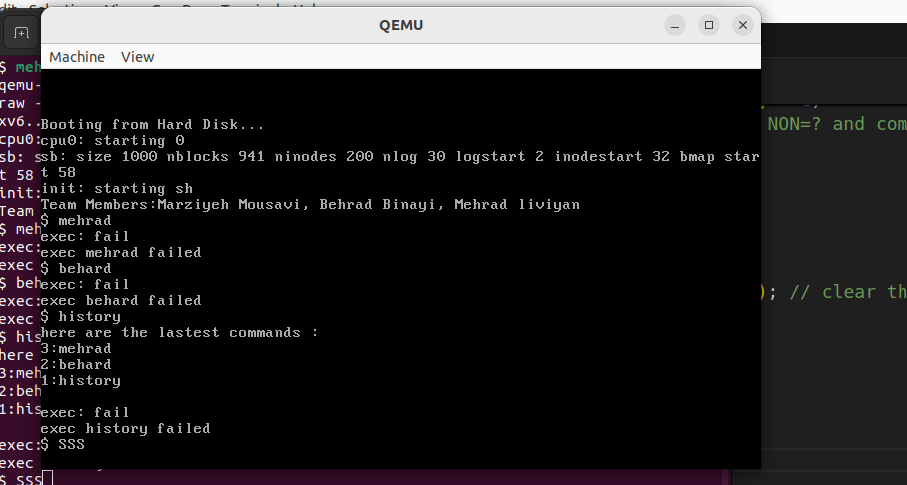
|  |  |
| --- | --- |
|  |  |

دو متغییر upDownKeyIndex که نشان دهنده تعداد بالا پایین رفتن کامند هست ( با هر بار بالا رفتن یکی زیاد می شود و با هر بار پایین رفتن یکی کم می شود.) که در ابتدا برابر 0 است و historyCurrentSize که نشان دهنده ایندکسی هست که آرایه historyBuf تا آنجا پر شده است.



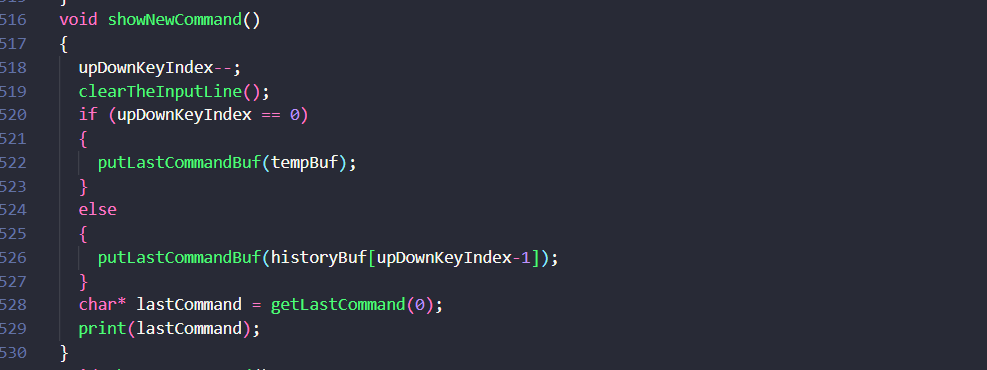
وقتی arrowkeyup زده شد دقت می کنیم که اگر upDownKeyIndex کوچکتر از historyCurrentSize می توان تابع showpastcommand را صدا می زنیم که در آن اول چک می کنیم که اگر upDownIndex برابر 0 بود مقدار کامند فعلی را در tempBuf ذخیره می کنیم ( یک آرایه از کاراکتر با طول 128 هست ) سپس یکی به upDownIndex اضافه می کنیم سپس تابع clearTheInputLine را صدا می زنیم تا خط کامند را پاک کند سپس خانه upDownIndex در historyBuf را به روی buffer می نویسیم سپس همان را هم چاپ می کنیم .



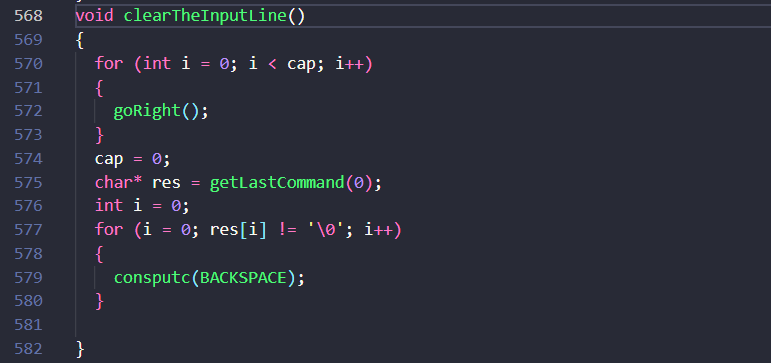


وقتی arrowkeydown زده شد ابتدا چک می کنیم که upDownKeyIndex بزرگتر از 0 باشد که بتوان پایین آمد سپس تابع showNewCommand را صدا می زنیم .

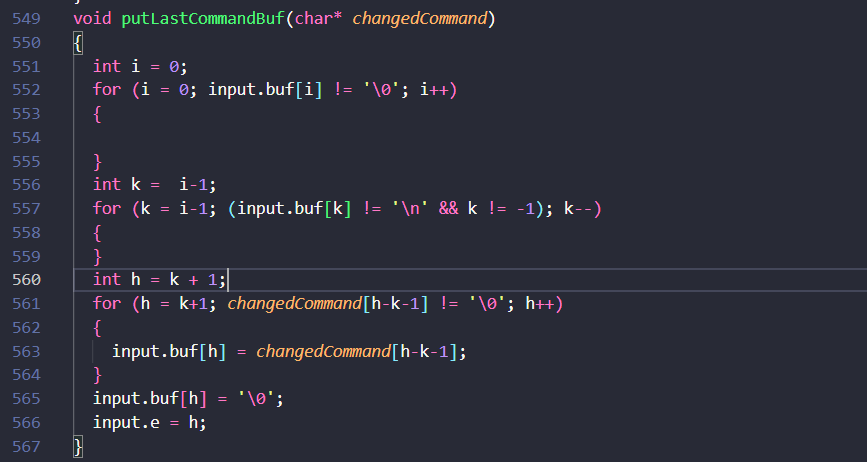
در این تابع ابتدا یکی از upDownIndex کم می کنیم سپس با استفاده از تابع clearTheInputLine کل خط را پاک می کنیم سپس چک می کنیم که آیا مقدار upDownIndex برابر 0 هست یا خیر . اگر برابر 0 بود باید tempBuf را بر روی buffer بزاریم ولی اگر برابر 0 نبود باید historyBuf را با upDownIndex چاپ کنیم و به روی Buffer قرار دهیم.



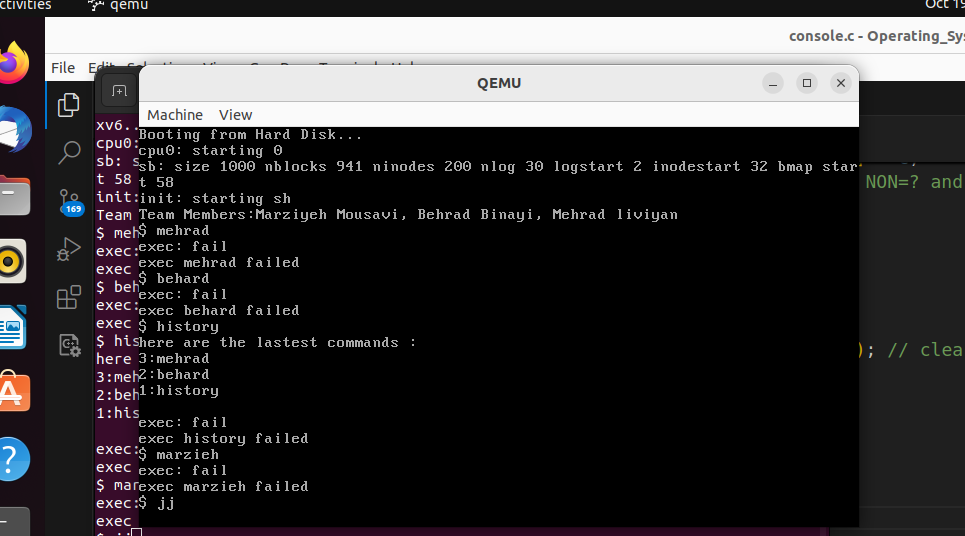
در تابع clearTheLine ابتدا باید به اندازه cap به راست برویم سپس cap را 0 کنیم سپس به اندازه آخرین کامند BackSpace بزنیم که پاک شود.



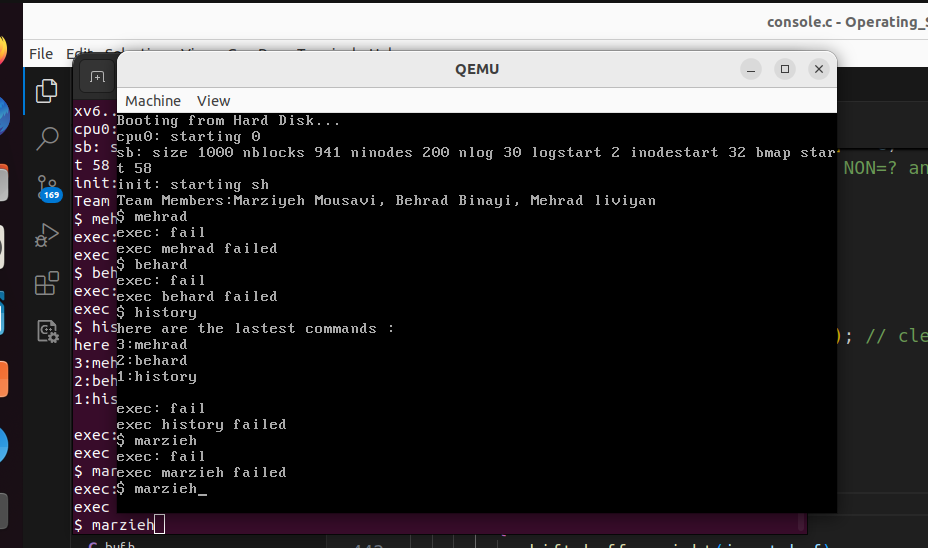
در تابع putLastCommand به این صورت عمل می کنیم که آخر buffer را پیدا می کنیم و آرایه ورودی را در آنجا چاپ می کنیم.



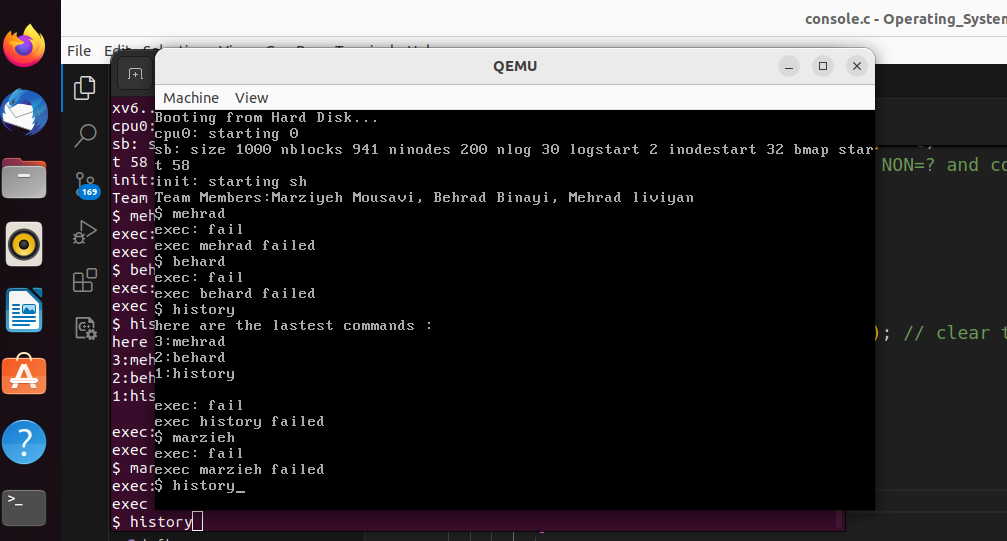
قبل از زدن دکمه up :



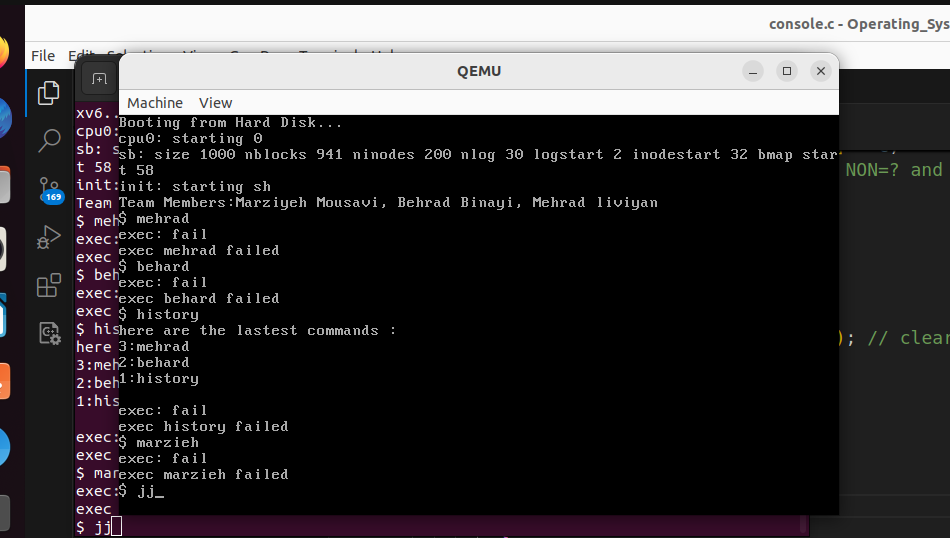
بعد از زدن یکبار دکمه up :



بعد زدن دکمه up دوباره :



زدن دکمه دوبار down :



1. برای دستور ctrl+S و crtrl+F از آرایه ای به سایز بافر به نام clipboard استفاده شده است. که بین ctrl+S و crtrl+F هر تغییری در console داده شده است را در خود بر اساس موقعیت زمانی آن یادداشت میکند.

|  |  |
| --- | --- |
|  |  |

1. الگوریتم کلی این دستور به این صورت میباشد که هر وقت user ، ؟ وارد میکند؛ سیستم عامل میرود از راست به چپ به ترتیب دنبال عدد، عملگر و عدد میگردد. از آنجایی که اعداد از سمت راست به چپ خوانده می شوند os اعداد را برعکس و به صورت string ذخیره می کند بنابراین ابتدا اعداد را به فرمت عدد صحیح تبدیل می کنیم سپس آن ها را برعکس میکنم و عملیات درخواست شده را انجام میدهیم

|  |  |
| --- | --- |
| 2. | 1. |
| 4. | 3. |

**سوالات گزارش کار:**

1-

* مدیریت منابع
* مدیریت پردازه ها
* واسط بین سخت افزار و نرم افزار

2-

Basic header:دارای constants و convention هایی که xv6 نیاز دارد

Entering xv6:کدی که xv6 را شروع می کند.xv6 برای لانچ شدن نیاز به یک سری ضروریات دارد که در فایل entry است

Locks:هماهنگ سازی دسترسی به ریسورس های مشترک

Processes:چیز هایی که برای انجام پراسس های یوزر و مولتی پروگرمینگ و مولتی تسکینگ نیاز داریم

System calls:کد های مربوط به سیستم کال ها که اینترفیس بین یوزر و کرنل است

File system:استراکچرهایی که برای مدیریت فایل ها دایرکتوری ها و مموری دیسک نیاز داریم در این قسمت است

Pipes:برای ارتباط بین پردازه ها استفاده می شود.پایپ ها به این صورت هستند که یک پراسس در یک طرف آن دیتا را می نویسد و پراسس دیگر در طرف دیگر آن

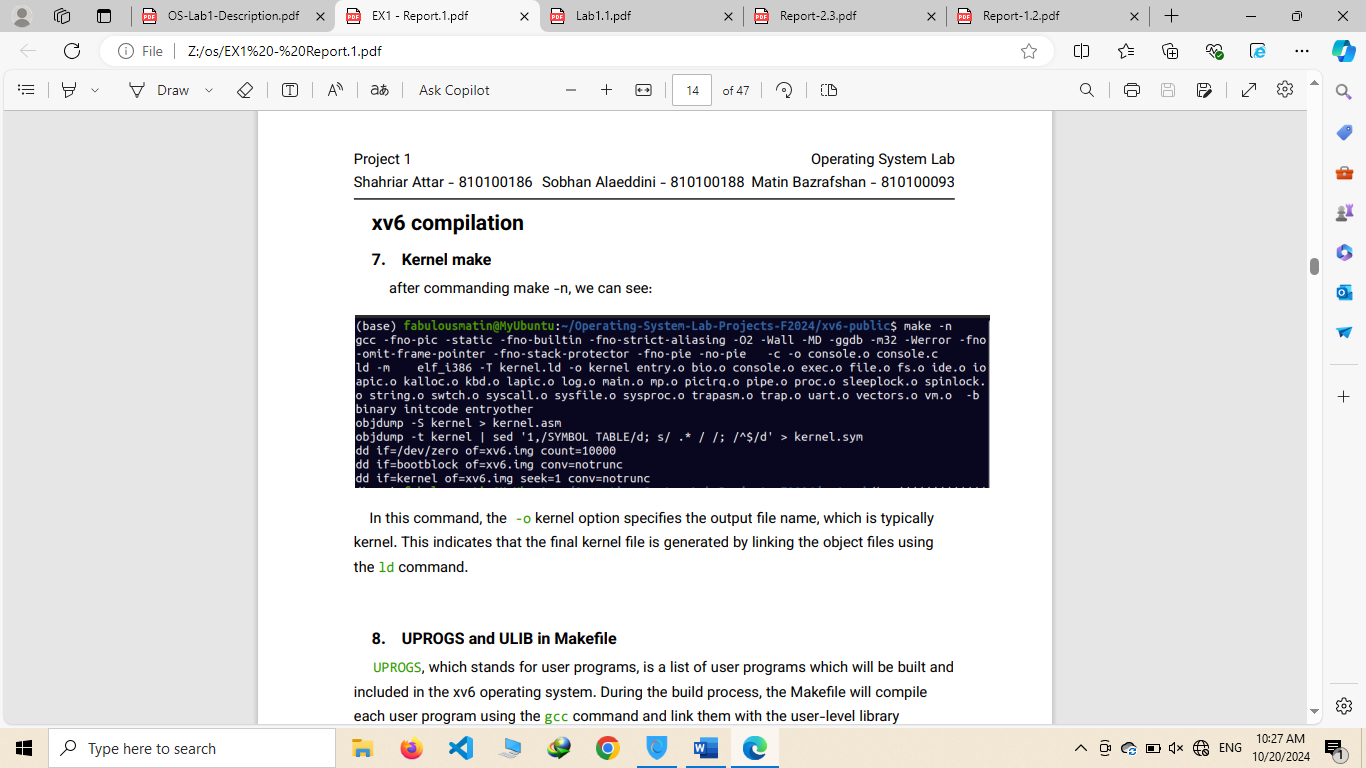
String operation:شامل عملیات هایی مانند strlen برای استرینگ ها

User level:کد init را دارد که همواره درحال اجرا می باشد.هدر user.h نیز دارای functionality های مربوط به یوزر است

Boot loader:کد هایی به زبان سی و اسمبلی که وظیفه ی آن ها لود کردن کرنل از دیسک به مموری است

Link:

3-



در این دستور، گزینه -o kernel نام فایل خروجی را مشخص می‌کند که معمولاً kernel است. این گزینه نشان می‌دهد که فایل نهایی کرنل با استفاده از دستور ld و لینک کردن فایل‌های objectتولید می‌شود.

4- در واقع UPROGS مخفف user program است که در آن قسمت می توانیم پروگرم های مربوط به یوزر را ببینیم و پروگرم های خودمان را اضافه کنیم.

ULIB مخفف یوزر لایبرری است.شامل utiliy function ها و سیستم کال هایی است که اجازه می دهند یوزر پروگرم ها با کرنل ارتباط داشته باشند

5- دستور -make qemu n برای آغاز به کار کردن OS در qemu استفاده می شود. فلگ n-برای لود کردن دو دیسک(که در

واقع drive hard های مجازی هستند ) به کار می رود که هر کدام محتوای متفاوتی دارند:

دیسک اول (System File Primary ): این دیسک شامل فایل سیستم 6xv و kernel آن است. این شامل و برنامه های پایه ای مانند ls و cat و shell می شود. در واقع این دیسک شامل سیستم نرم افزاری لازم برای اجرا شدن 6xv است. همچنین وقتی برنامه جدیدی مانند encode و decode را تعریف می کنیم، روی این دیسک ذخیره می شوند.

دیسک دوم : این دیسک معمولا خالی است و یا برای حافظه فایل اضافه استفاده می شود. این دیسک به ما کمک می کند که بتوانیم تست کردن os با چندین دیسک را تجربه کنیم و یا دیتا یا برنامه های مختلف را روی آن ذخیره کنیم.

8- از این دستور برای convert کردن object ها استفاده می شود.در xv6 بوت لودر برای لود کردن مستقیم فایل executable در مموری باید فرمت خاصی داشته باشد که این command آبجکت فایل را می گیرد و به فایل باینری مطلوب تبدیل می کند.

13- با توجه به اینکه بخشی از حافظه ی قبل این آدرس مربوط به bios است باید کرنل در بخشی قرار بگیرد که با bios تداخلی نداشته باشد

18-دلیل استفاده از seguser آن است که بین پردازه های سطح کاربر و پردازه های سطح هسته جداسازی انجام شود.

19-proc struct شامل attribute های زیر است:

Pid,state,sz,pgdir,context,chan,kstack,tf,pid,parent

Pid:

Int

یک شناسه ی یونیک برای هر پردازه

State:

Enum proc\_state

استیت فعلی پردازه را می گوید.برای اسکجلر مهم است که بداند آیا پردازه در حال ران شدن است یا منتظر است یا اصلا متوقف شده

Sz:

Int

سایز پردازه به بایت در مموری که شامل استک و هیپ اختصاص داده شده نیز هست

Pgdir:

Pde\_t\*

پوینتری که به دایرکتوری پیج یک پردازه اشاره دارد.این استراکچر کمک می کند پردازه آدرس های مجازی را به مموری فیزیکی مپ کند

Context:

Struct context

در هنگام context switch باید مقادیر بعضی از رجیستر ها و استیت پردازه ها سیو شود که وقتی دوباره به پردازه برگشتیم بدانیم باید از کجا شروع کنیم.این مقادیر در این متغیر است

Chan:

Void\*

وقتی پردازه ای به حالت sleep می رود کاربردی است که دلیل آن را بدانیم.مثلا دلیلش انتظار برای I/O بوده یا نه.این متغیر شامل این دلیل است.

Kstack:

Char[]

استک هسته برای پراسس که برای کال کردن توابع مورد نیاز است

Tf:

Struct trapframe

Trap frame هر پردازه را دارد که شامل رجیستر ها و بقیه اطلاعات مربوط به استیت پردازه است که برای هندل کردن اینتراپت ها و اکسپشن ها لازم است.

Parent:

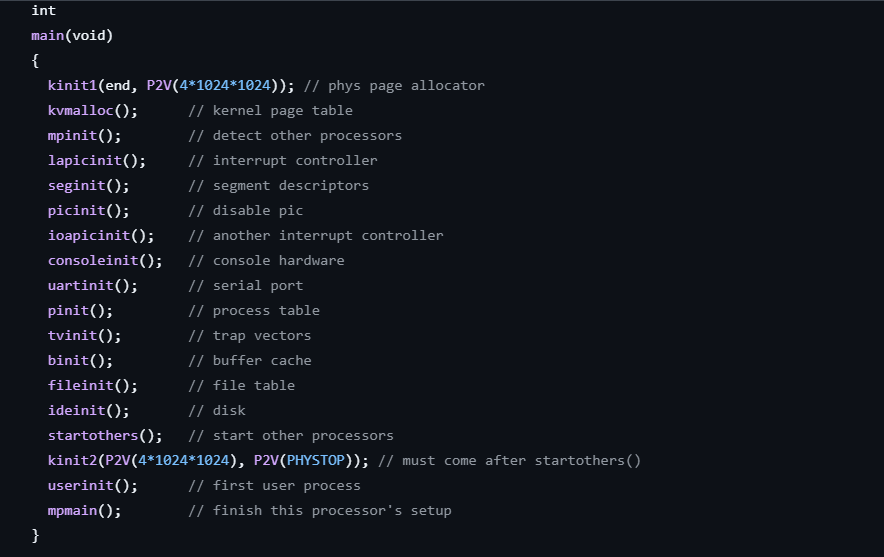
Struct proc\*

پوینتر به پردازه ی والد که برای سلسله مراتب پردازه ها لازم است.

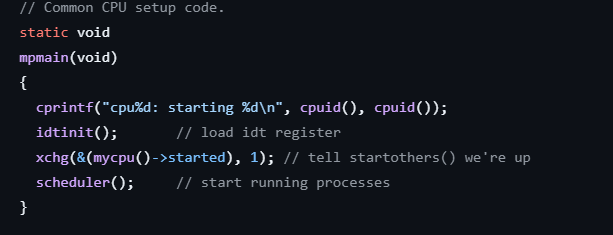
معادل این ساختار در لینوکس task struct است

این استراکچر به طور کلی در مدیریت پردازه ها نقش مهمی دارد.

23-



همانطورکه از بخشی از کد main.c مشخص است تا تابع startothers مختص یک هسته هست و بعد از این تابع بقیه هسته ها شروع به کار می کنند.بنابراین توابعی که در main هستند مختص هسته ی اول است.



همانطور که از کد مشخص است تابع mpmain که به تابع startothers می گوید آماده ی اجرای پردازه هست بین چند هسته مشترک است.

بنابراین:

مرحله ی boot به طور اختصاصی توسط هسته ی اول انجام می شود.بنابراین توابعی مانند kvmalloc,mpinit,startothers,picinit یعنی توابعی که برای آماده سازی سیستم هستند اختصاصا توسط این هسته انجام می شوند.

توابعی مانند mpmain,switchkvm مشترکا توسط همه ی هسته ها اجرا می شوند.

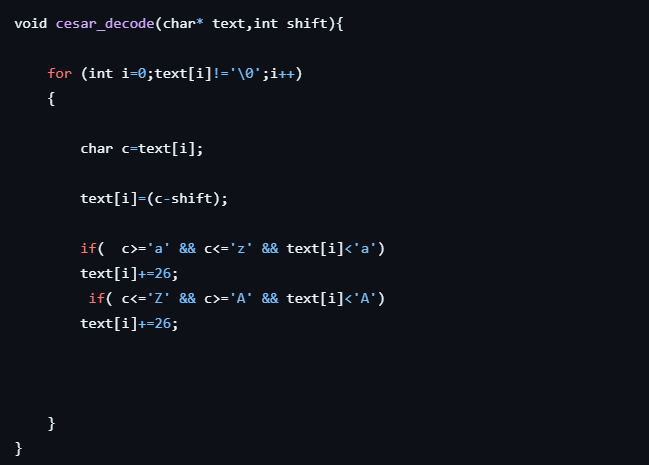
زمانبند یا scheduler در تابع mpmain است پس مشترکا توسط همه ی هسته ها اجرا می شود.

اضافه کردن برنامه ی سطح کاربر:

ابتدا توابع encode و decode را هر دو با اسم cesar encode در دو فایل جداگانه به اسم encode و decode گذاشتیم.در encode باید هر حرف + شیفت بخورد و باقیمانده به 26 سنجیده شود و از آن طور در decode باید -شیفت بخورد و دوباره برای اینکه در بازه باشد،باید باقیمانده به 26 سنجیده شود.

حرف های capital و small نیز جداگانه شیفت می دهیم.

تابع decode :



تابع encode:



شکل main برای هر دو فایل یکسان است.

با استفاده از argv و argc جمله را پاس می دهیم.

یعنی اسم فایل decode و encode را به عنوان دستور و جمله ها را در کنار آن می گذاریم.استرینگ ها در argv هستند.argc تعداد استرینگ های پاس شده است.اگر کمتر از 2 باشد(چون اولین متغیرargv مربوط به اسم برنامه است) یعنی جمله ای پاس داده نشده.در غیر این صورت کلمات را در یه آرایه ی جداگانه می ریزیم و انکود می کنیم.سپس تکست انکود شده را در فایل می نویسیم.

یک پارامتر flag هم تعریف می کنیم که اگر تعداد آرگومان ها کمتر از 2 بود اصلا فایلی باز نشود.

شکل کلی main:



شکل makefile به صورت زیر تغییر می کند و دو برنامه به سطح کاربر اضافه می شود

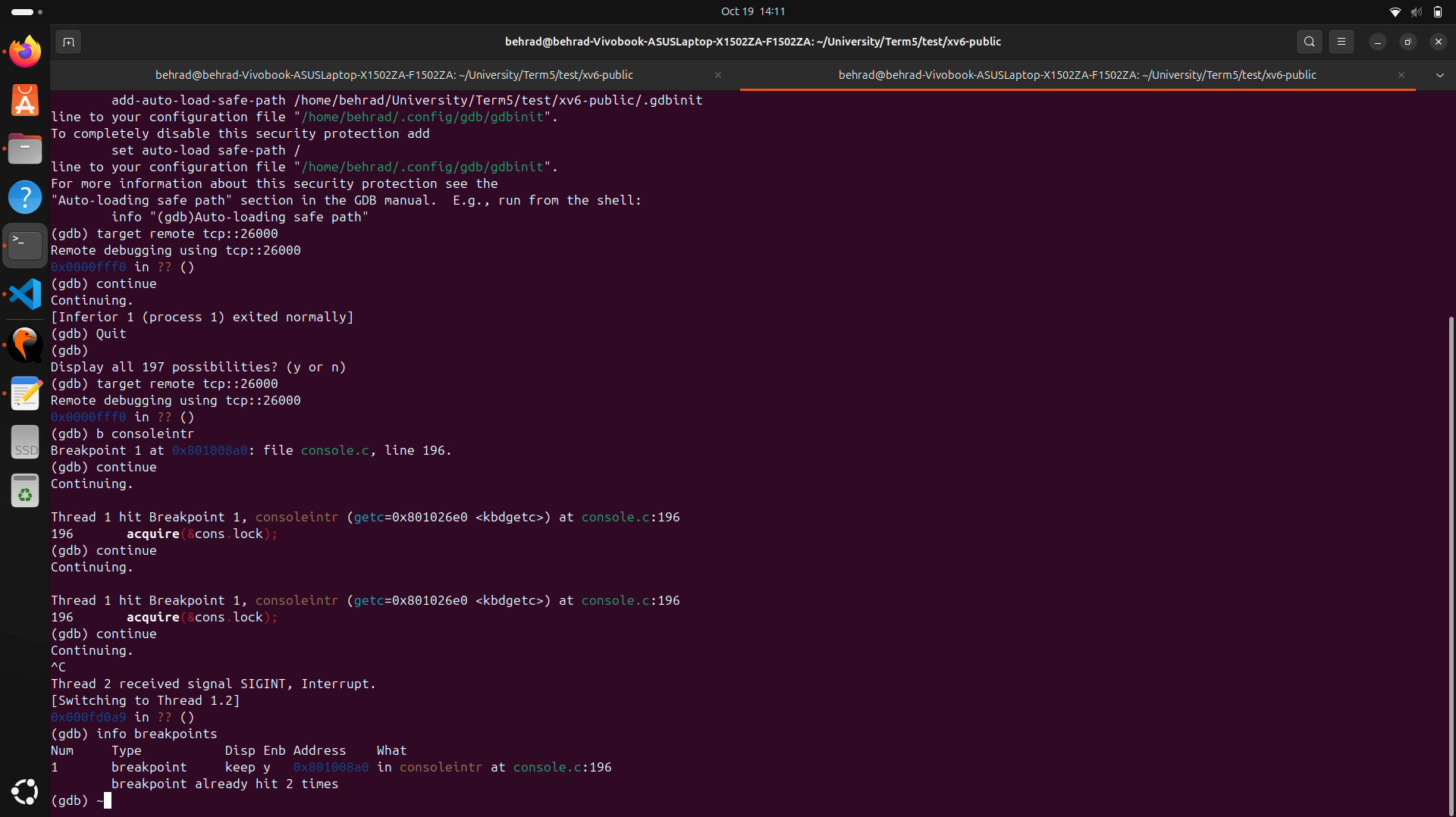


GDB:

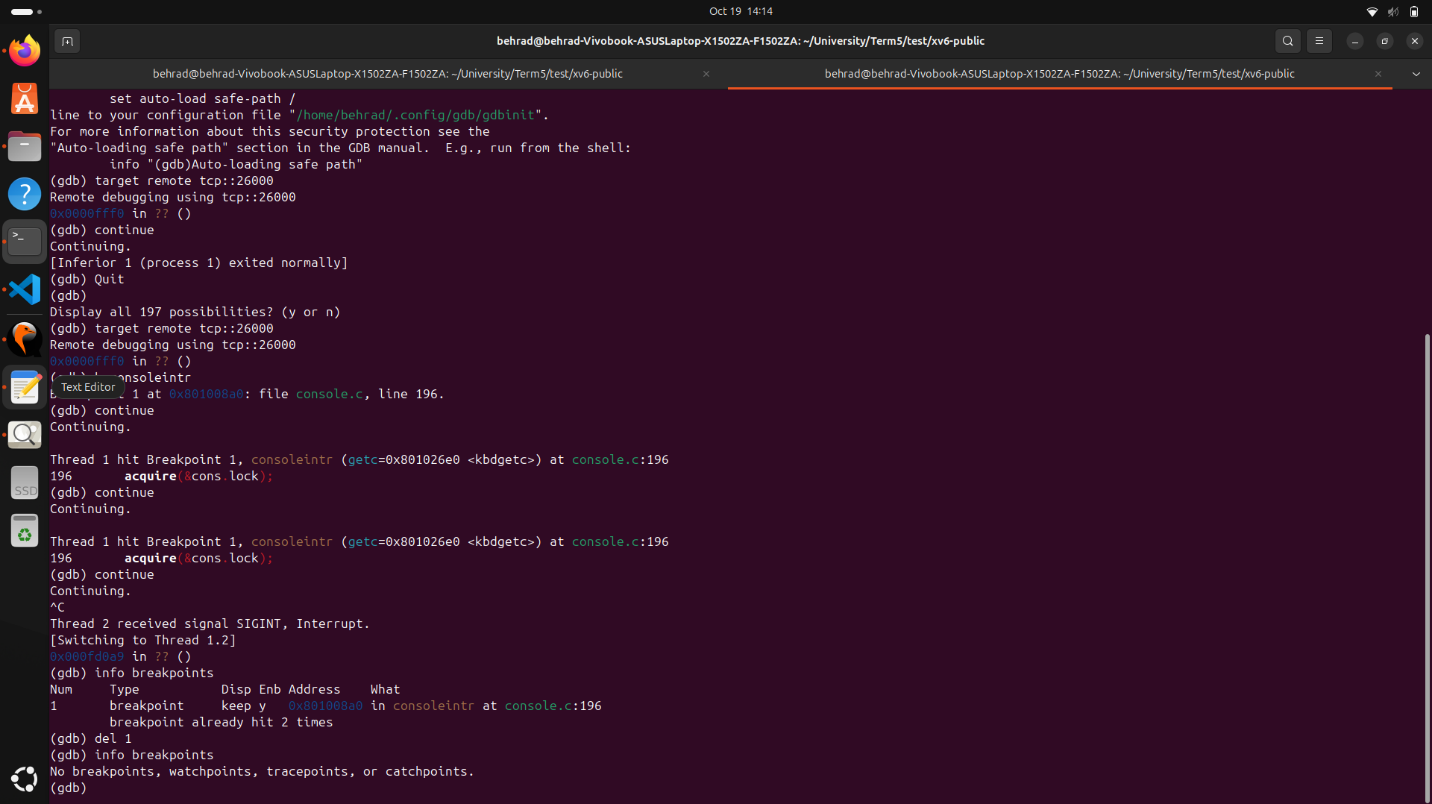
در همه ی مثال های مربوط به breakpoint ، break point سر تابع consoleintr در فایل console.c است.

1-

برای مشاهده breakpoints ها از info breakpoints ا استفاده می شود.



2-برای حذف breakpoint ها می توانیم از info breakpoints شماره هر break point را یافته و از دستور del برای حذف آن ها استفاده کنیم.



اگر del را با شماره بزنیم فقط breakpoint با شماره مربوطه حذف می شود و اگر خالی بزنیم همه ی breakpoint ها حذف می شود.

3-دستور bt مخفف backtrace است.این دستور بابت هر فریم استک یک لاین پرینت می کند.

4-

دستور X:

دستور x برای معاینه ی خانه های حافظه استفاده می شود.پارامتر address آدرس خانه ی حافظه و پارامتر FMT برای فرمت نمایش محتویات خانه ی حافظه می شود.

دستور print :

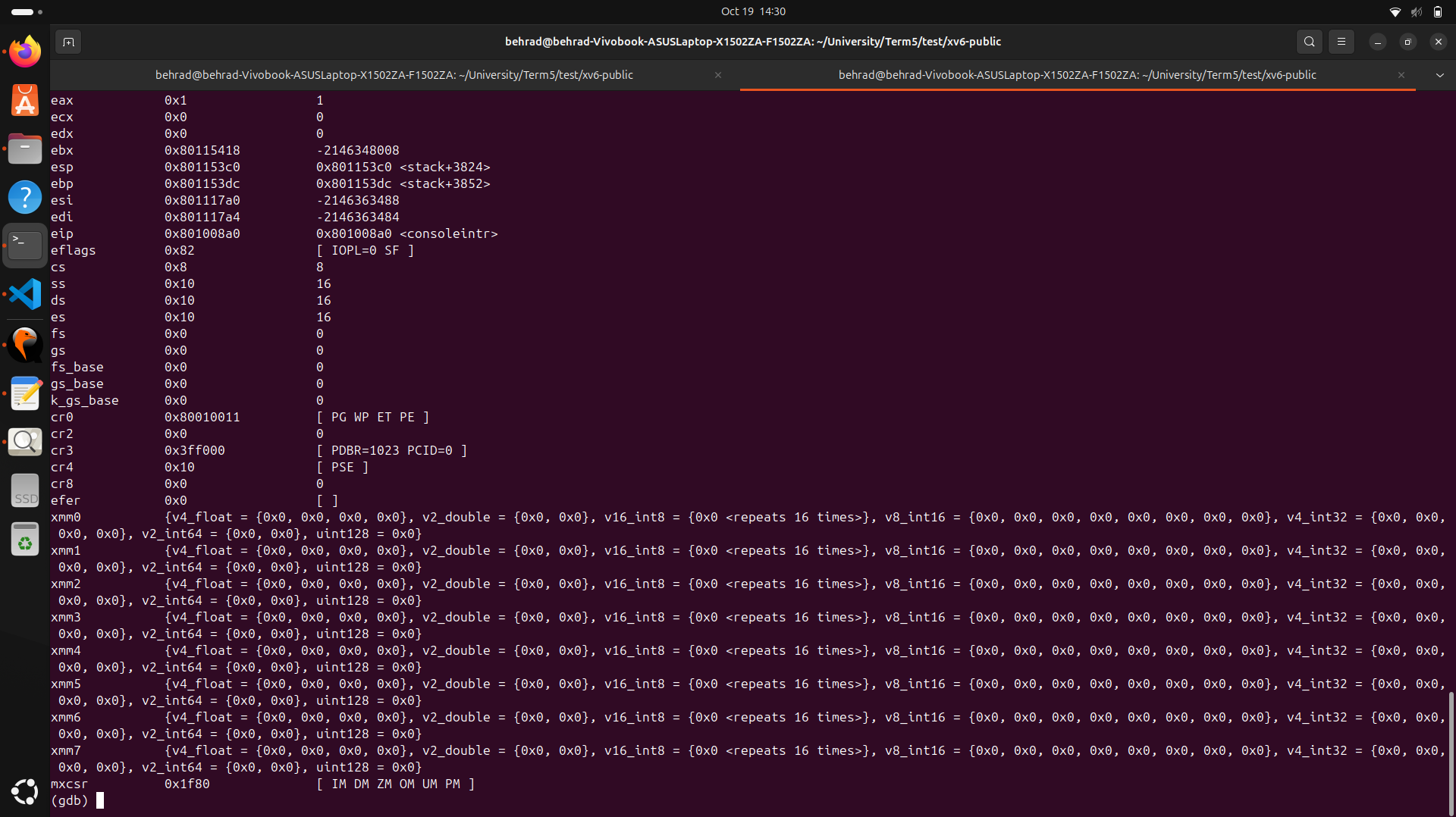
برای نشان دادن مقدار یک expression در gdb استفاده می شود.این expression مربوط به متغیر های local و رجیستر ها و هر گونه عبارتی استفاده می شود.

برای نشان دادن مقدار یک رجیستر می توان از دستور print استفاده می شود. برای نمایش یک رجیستر خاص می توان پشت اسم رجیستر علامت $ گذاشت و آن را پرینت کرد.

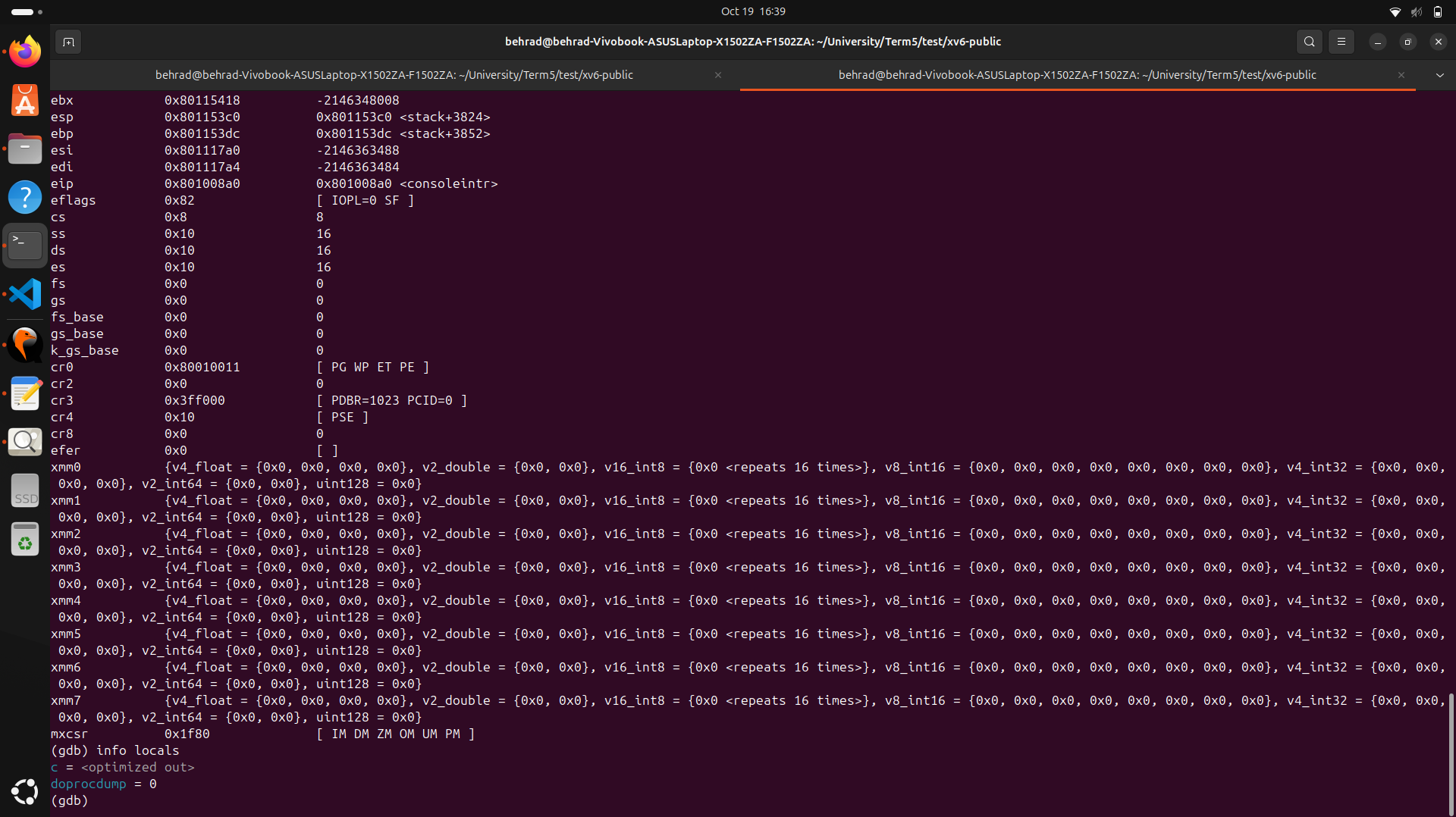
5-

برای نشان دادن محتویات register ها از دستور info registers می توان استفاده کرد و برای نشان دادن محتویات متغیر های local از info locals استفاده می شود.

وقتی info registers را می زنیم شکل رجیستر ها به صورت زیر است:



وقتی info locals را می زنیم شکل متغیر های محلی به صورت زیر است:



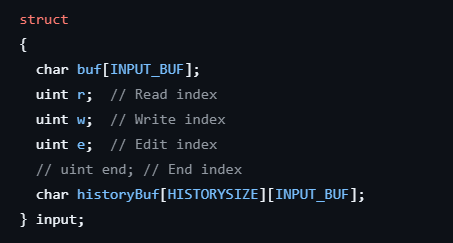
EDI:یکی از رجیستر های general purpose که در زبان اسمبلی استفاده می شود است.EDI مخفف extended destination index است.یک رجیستر 32 بیتی است که درعملیات های مربوط به آرایه ها،آدرس دادن مموری وتغییر دادن استرینگ ها استفاده می شود.

ESI:مخفف Extended source index است.مانند EDI یکی از رجیستر های general purpose است و کاربردش مانند EDI است.

به طور کلی در عملیات های مربوط به استرینگ ها، یا عملیات هایی مانند MOVSB، MOVESW ،MOVESD و .. ESI برای رفرنس کردن دیتای سورس و EDI پوینتری به دیتای مقصد است.

6-

استراکت اینپوت به شکل زیر است.



e که attribute این استراکت هست برای ادیت است.یعنی هر وقت کاراکتری در ورودی وارد می شود ++ می شود و هر وقت backspace زده می شود -- می شود.با زدن enter همه ی ورودی پردازش می شود .

با زدن enter ورودی تایپ شده در بافر ذخیره می شود و w به جلو حرکت می کند.هر وقت ورودی جدید وارد می شود نیز wدچار تغییر می شود و این متغیر همواره به اولین خانه بعد از آخرین ورودی اشاره می کند.

r مربوط به خواندن است.هر وقت کرنل بخواهد ورودی را از بافر بخواند r تغییر می کند و نشان می دهد چقد از بافر خوانده شده .

هیستوری باف اضافه شده توسط خود ما برای ذخیره ی کامند هاست.

خود باف ورودی خام را از یوزر می گیرد و ذخیره می کند.

7-

دستورات layout برای بررسی استیت های متفاوت از برنامه است.خروجی دو دستور زیر در tui نشان داده می شود.

حال layout asm کد اسمبلی که مربوط به نقطه ی اجرا را نشان می دهد.

دستور layout src سورس کد مربوط به دستور که در اینجا زبان سی است را نشان می دهد.

8-

از دستورات backtrace،up،down وframe می توان استفاده کرد.

فرمت هر دستور:

Backtrace:

Bt

لیست کامل توابعی که تا نقطه ی توقف فراخوانی شده را نشان می دهد.

Frame <N>

دستور را به فریم N در زنجیره ی فراخوانی منتقل می کند که می توان وضیعت متغیر ها را مشاهده کرد

Up و down برای جابجایی بین فریم هاست.UP برای رفتن به تابعی که تابع فعلی را فراخوانی کرده(caller) و down برای رفتن به توابع callee تابع ما می رود.