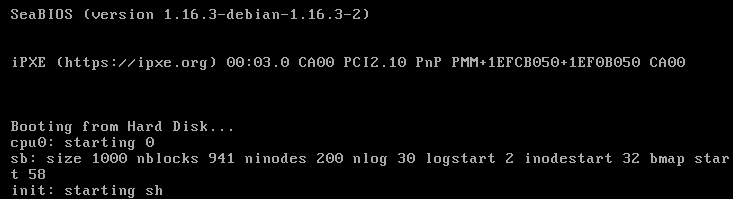
به نام خدا

گزارش آزمایشگاه آزمایش اول درس سیستم عامل

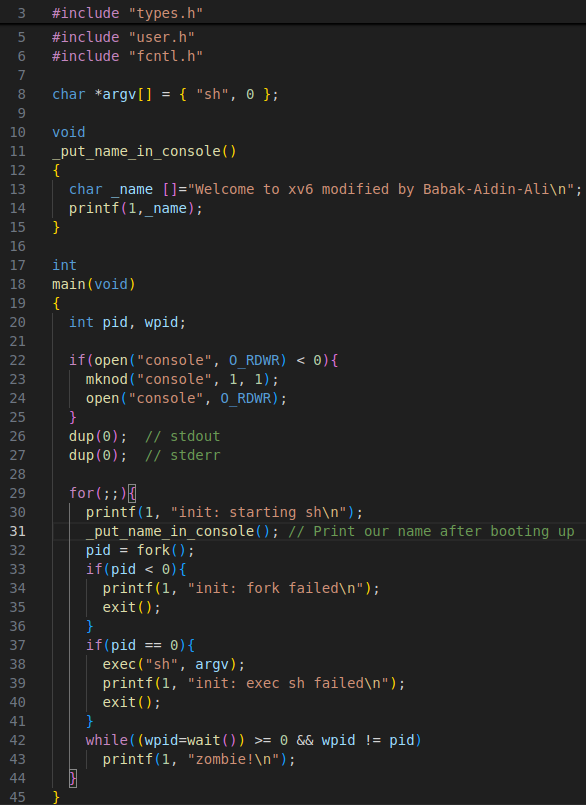
آیدین کاظمی: 810101561 علی زیلوچی: 810101560 بابک حسینی محتشم: 810101408

بخش اضافه کردن یک متن به Message Boot:

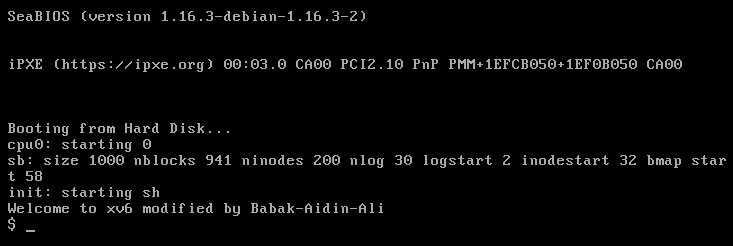
هربار که xv6 را اجرا می‌کنیم تصویر زیر را مشاهده می‌کنیم:



پس برای اضافه کردن متن به انتهای این پیام، متن آخرین خط را جستجو کردیم و متوجه شدیم که فایل init.c این پیام را چاپ می‌کند. پس از چاپ این پیام، نام اعضای گروه را چاپ می‌کنیم:



حال اگر سیستم‌عامل را اجرا کنیم تصویر زیر را مشاهده می‌کنیم:



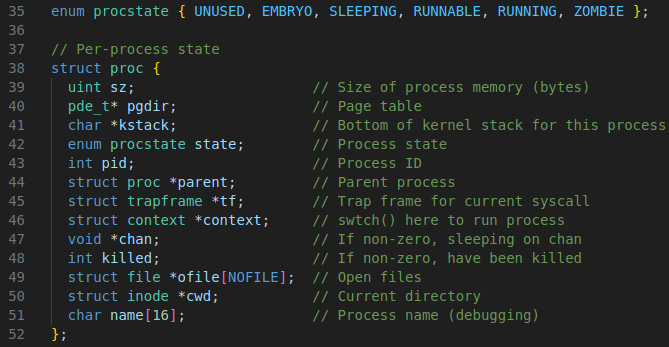
سوالات بخش آشنایی با سیستم عامل xv6:

1. معماری سیستم‌عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟

معماری این سیستم‌عامل monolithic است. در دفاع از نظر خود می‌دانیم که xv6 پیاده‌سازی مدرنی از سیستم‌عامل Unix است که خود معماری monolithic دارد. همچنین پس از کار با این سیستم‌عامل متوجه شدیم که تمام قسمت‌های سیستم‌عامل دسترسی کامل به سخت‌افزار دارند. برای مثال می‌توانیم از هر فایل سیستم‌عامل بنویسیم یا بخوانیم که دسترسی به سخت افزار برای همه قسمت‌های یک سیستم‌عامل از ویژگی‌های معماری monolithic است.

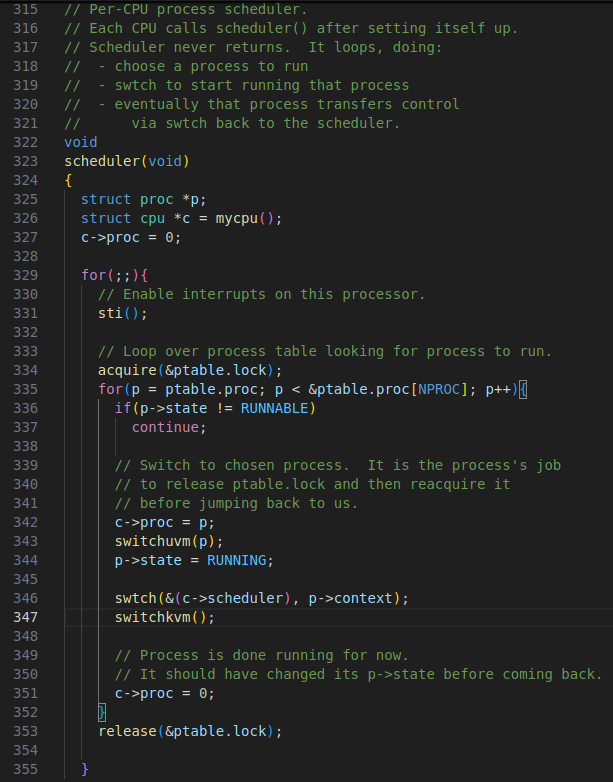
1. یک پردازه در سیست‌معامل xv6 از چه بخش‌هایی تشکیل شده است؟ این سیستم‌عامل به طور کلی چگونه پردازنده را به پردازه‌های مختلف اختصاص میدهد؟

در فایل proc.h ساختار داده زیر وجود دارد که همان ساختار داده یک پردازه در xv6 است:



هز پردازه در xv6 شامل حافظه‌ای در فضای کاربر که تشکیل شده از دستورات، داده‌ها و پشته است. همچنین kernel به هر پردازه یک pid می‌دهد که نشان دهنده آن پردازه خاص است. همچنین هر پردازه، استیتی که در آن قرار دارد، نام خود، پوینتری به پردازه پدر را هم نگه می‌دارد.

در فایل proc.c تابع scheduler قرار دارد:



می‌توان دید که در این سیستم‌عامل برای انتخاب پردازه‌ها از الگوریتم round-robin استفاده می‌شود و هر بار بین پردازه‌ها، پردازنده به پردازه‌ای که در استیت RUNNABLE است که یعنی قابل اجرا است داده می‌شود و استیت آن پردازه به RUNNING تغییر می‌کند و سپس تابع swtch صدا زده می‌شود تا محتوای پردازه فعلی را ذخیره و محتوای پردازه جدید را جدید را جایگزین کند. در این سیستم‌عامل در دو حالت بین پردازه‌ها جابه‌جا می‌شویم: یا پردازه‌ای از به دلایل مختلف مانند منتظر I/O بودن، از مکانیزم sleep استفاده کند و یا تغییر در پردازه‌ها به صورت تناوبی توسط xv6 وقتی که پردازه‌ای در حال اجرا کردن دستورات کاربر باشد.

1. مفهوم file decriptor در سیستم‌عامل‌های مبتنی بر UNIX چیست؟ عملکرد pipe در سیستم‌عامل xv6 چگونه است و به طور معمول برای چه هدفی استفاده میشود؟

File descriptor عدد صحیح کوچکی است که نشان دهنده شیئی است که یک پردازنده از آن می‌خواند یا در آن می‌نویسد. مفهوم File descriptor با انتزاعی کردن مفاهیم file، pipe، و device تفاوت بین آن‌ها را از بین می‌‌برد و ui یکسانی ارائه می‌دهد.

Pipe بافری کوچک در kernel است که بین چند پردازه قرار می‌گیرد و از آن هم برای خواندن و هم نوشتن استفاده می‌شود. نوشتن اطلاعات در یک سمت pipe، آن اطلاعات را در سمت دیگر در اختیار پردازه دیگر قرار می‌دهد. پردازه‌ها می‌توانند به کمک pipeها با هم ارتباط برقرار کنند. Pipeها مشابه فایل‌های موقتی عمل می‌کنند ولی 4 مزیت نسبت به یک فایل موقتی دارند: اول اینکه یک pipe خودش پس از پایان ارتباط، خودش را پاک می‌کند. دوم اینکه می‌شود با pipe مقدار زیادی داده فرستاد. سوم اینکه pipe اجازه اجرای موازی می‌دهد. چهارم اینکه در pipeها اگر پردازه‌ای سعی کند از pipe خالی بخواند، متوقف می‌شود و صبر می‌کند تا داده‌ای در pipe قرار گیرد.

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

پردازه‌ها به کمک فراخوانی سیستمی fork می‌توانند پردازه‌های جدیدی را بسازند. این فراخوانی پردازه فرزند را با محتوای یکسان با پردازه پدر تشکیل می‌دهد. این فراخوانی در پردازه پدر، pid پردازه فرزند را برمی‌گرداند و در پردازه فرزند، صفر برمی‌گرداند.

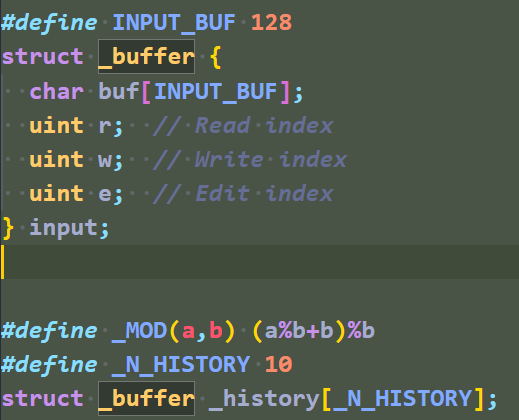
فراخوانی سیستمی exec محتوای حافظه پردازه را با محتوای جدیدی که در فایلی ذخیره شده جایگزین می‌کند و پردازه را با محتوای جدید اجرا می‌کند. فایلی که در آن دستورات ذخیره شده باید فرمت خاصی به نام ELFباشد.

مزیت یکسان نکردن این دو فراخوانی سیستمی در این است که اگر این دو یکسان نباشند، shell می‌تواند پردازه‌ای را با fork بسازد، سپس از دستورات open،close و dup در پردازه فرزند استفاده کند تا تغییراتی در file descriptorها ایجاد کند و سپس از دستور exec استفاده کند.

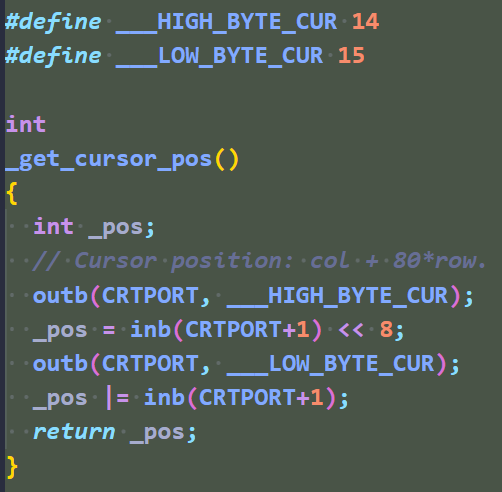
بخش شرح پروژه:

بخش 1:

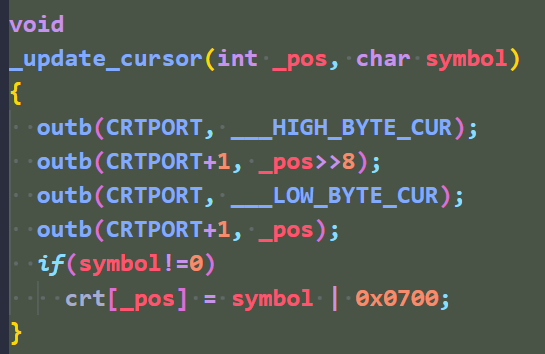
در ابتدا استراکت بافر را برای نگه داری محتویات یک بافر، شامل پوینتر های خواندن، نوشتن و ادیت و خود بافر که آرایه ای ازکاراکتر ها به طول 128 است تعریف میکنیم. از این استراکت هم در نگه داری history (که آرایه اس از استراکت ها به طول 10 خواهد بود) و هم در بافر ورودی استفاده خواهد شد:



در اینجا باید دو تابع مهم برای فهمیدن و آپدیت مقدار اشاره گر در ترمینال را معرفی کنیم. تابع \_get\_cursor\_pos با فرستادن چهار سیگنال به رجسیتر کنترلر vga، ابتدا دسترسی خواندن این رجیستر را گرفته و سپس یک بایت از آن را میخواند، که این کار را یک بار برای بایت بالایی و یک بار برای بایت پایینی انجام میدهد و پس از اتمام خواندن به همان شکل در متغیر \_pos ذخیره کرده، به عنوان محل قرار گیری اشاره گر باز میگرداند:

****

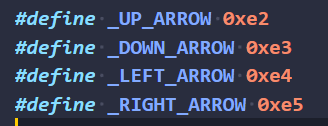
تابع \_update\_cursor وظیفه آپدیت کردن مکان اشاره گر با مقداری دلخواه را دارد. این تابع نیز ابتدا دسترسی نوشتن در رجیستر مورد نظر را پیدا کرده، سپس بایت بالایی و پایینی آن را با مقدار دلخواه ورودی تغییر میدهد. همچنین میتوان سیمبل قرار گیرنده بر روی ترمینال را نیز تغییر داد که در این پروژه استفاده ای از آن نمیشود:

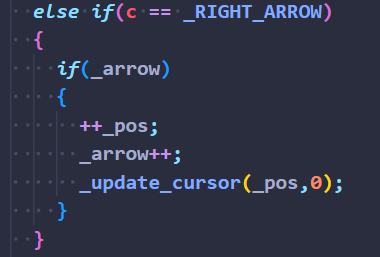
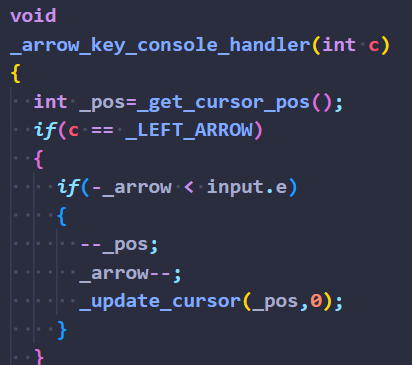


یک متغیر مهم \_arrow میباشد که به صورت گلوبال تعریف شده و وظیفه مشخص کردن مقدار عقب بودن اشاره گر از پوینتر e بافر را دارد. مقدار این متغیر در صورتی که وسط جمله باشد، منفی فاصله آن تا سر جمله و در صورتی که سر جمله باشد صفر است:

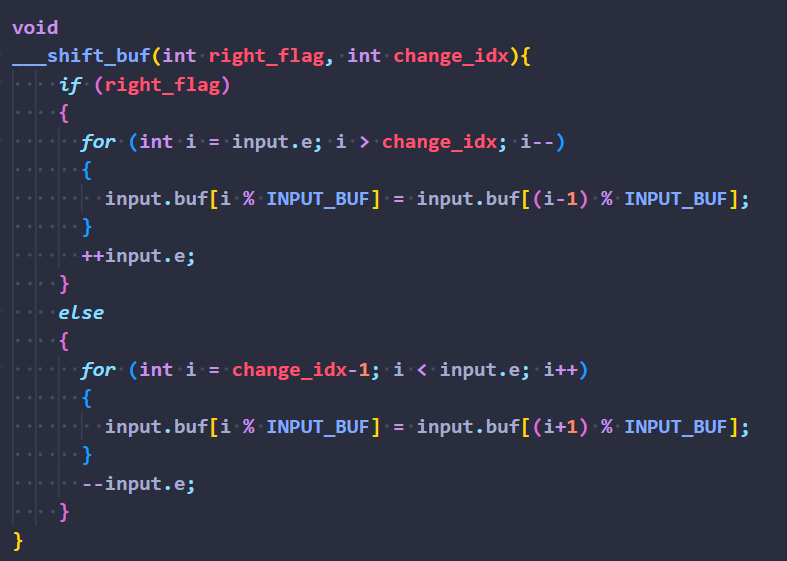


تابع بعدی \_arrow\_key\_console\_handlerمیباشد که در این قسمت فقط دو بخش ابتدایی آن توضیح داده میشود. این تابع با ورودی گرفتن یک کاراکتر ابتدا مقدار فعلی اشاره گر در صفحه را گرفته، سپس در صورتی که کلید سمت چپ باشد، بررسی میکند که به انتهای سمت چپ نرسیده باشیم و در صورت برقرار بودن شرط پوزیشین اشاره گر را یکی کم کرده، متغیر \_arrow را نیز یکی کم میکند و اشاره گر را روی صفحه آپدیت میکند. در صورتی که کلید راست فشرده شود و \_arrow صفر نباشد، همین کار را به بعلاوه یک کردن تکرار میکند:

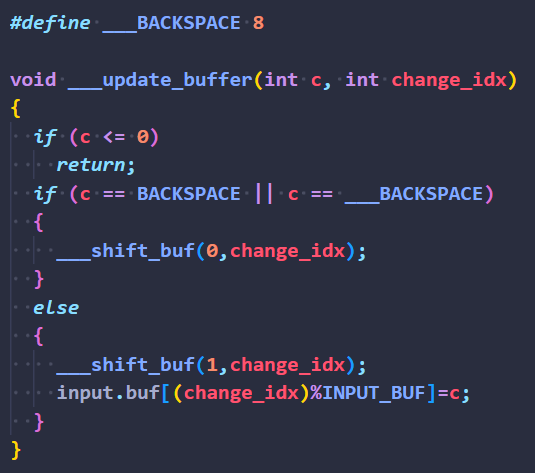
****



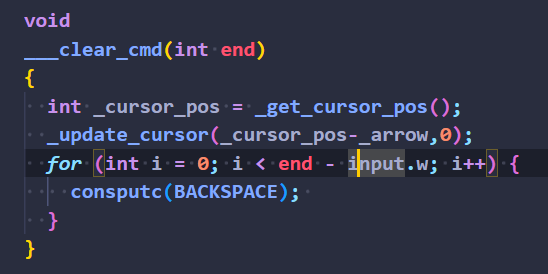
در اینجا باید چند تابع مهم و پر کاربرد معرفی شوند. ابتدا تابع \_\_\_ shift\_buf که محل شیفت پیدا کردن را گرفته و جهت شیفت را نیز میگیرد و به یکی از دو سمت چپ یا راست بافر را از محل مشخص شده شیفت میدهد و مقدار پوینتر e را یکی کم یا زیاد میکند (در شیفت راست خانه مشخص شده شیفت تغییر نمیکند و در شیفت چپ پاک میشود):



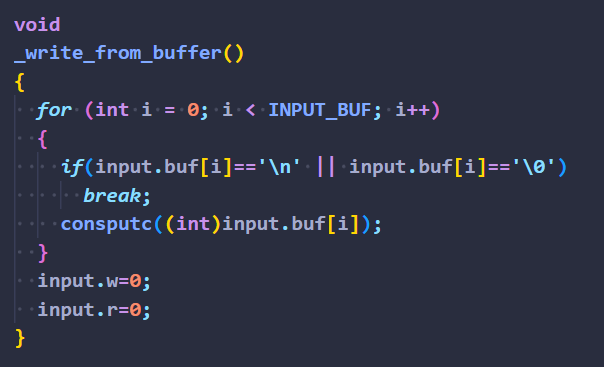
تابع بعدی \_\_\_update\_buffer میباشد که وظیفه قرار دادن یک کاراکتر جدید داخل بافر را دارد. در صورتی که کاراکتر بک اسپیس باشد، از محل تعیین شده بافر را به چپ شیفت میدهد، و در غیر این صورت از محل تعیین شده بافر را یکی به راست شیفت داده و کاراکتر جدید را جایگذاری میکند:



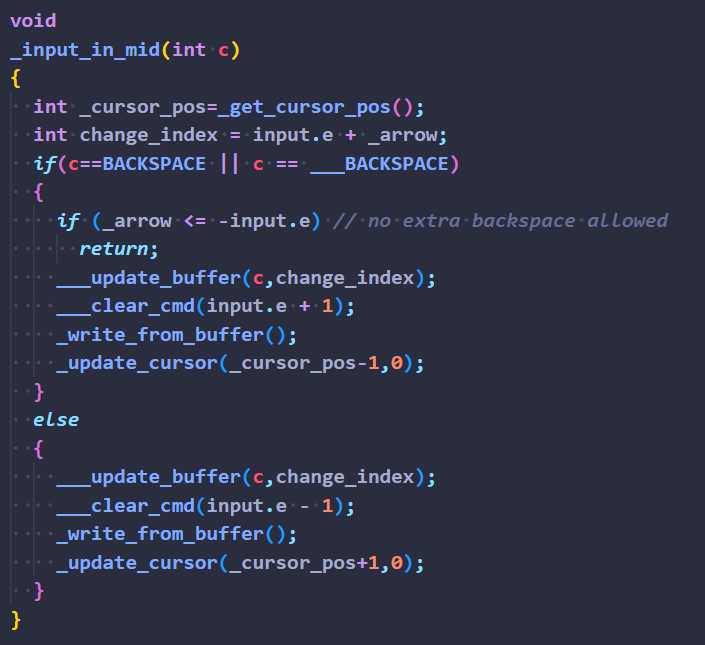
تابع بعدی \_\_\_clear\_cmd میباشد که وظیفه خالی کردن ترمینال را دارد. این تابع ابتدا اشاره گر را به انتهای راست جمله روی ترمینال برده، سپس به تعداد کاراکتر های جمله بک اسپیس وارد ترمینال میکند تا ترمینال خالی شود:

****

تابع بعدی \_write\_from\_buffer میباشد که مسئول نوشتن تمام کاراکتر های بافر به روی ترمینال است. این تابع تا رسیدن به کاراکتر نال یا نیو لاین کاراکتر های بافر را روی ترمینال قرار میدهد، و نهایتا پوینتر های خواندن و نوشتن از بافر را ریست میکند:

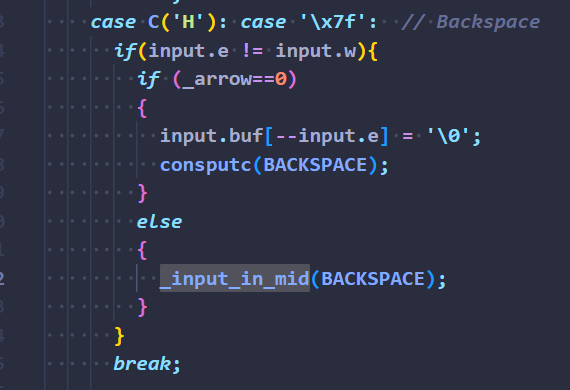


حال تابع بسیار مهم \_input\_in\_mid را میتوان از کنار هم قرار دادن این توابع بالا ساخت، که وظیفه آن مدیریت کاراکتر جدید وارد شده در وسط جمله میباشد. این تابع با گرفتن کاراکتر ورودی، مکان وارد کردن را با جمع کردن متغیر های e و \_arrow پیدا کرده و با توجه به بک اسپیس بودن یا نبودن آن، بافر را آپدیت کرده، ترمینال را پاک و بافر جدید را نوشته و موقعیت اشاره گر جدید را نیز آپدیت میکند:

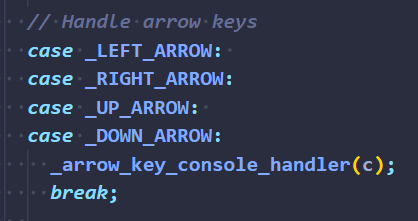
****

حال با کمک این تابع و تابع \_arrow\_key\_console\_handler ، تابع consoleintr که وظیفه مدیریت کلید های ورودی را دارد را در سه جا آپدیت میکنیم:

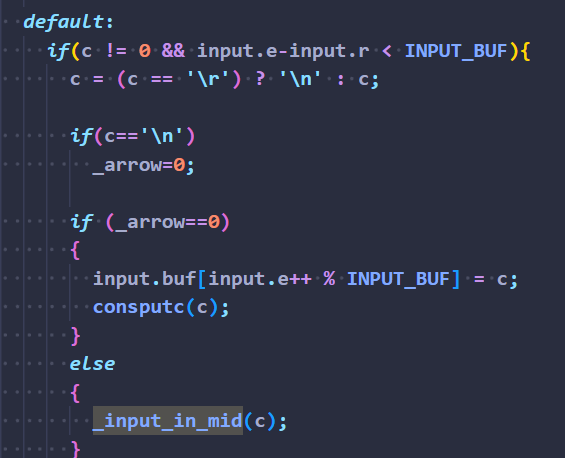
1- هنگامی که ورودی بک اسپیس است، در صورتی که در وسط جمله باشیم (\_arrow صفر نباشد) باید از \_input\_in\_mid استفاده شود:



2- در صورتی که کلید های چپ و راست (یا بالا و پایین که در قسمت های بعد توضیح داده خواهد شد) فشرده شوند باید از \_arrow\_key\_console\_handler استفاده شود:

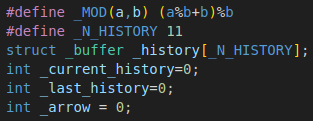


3- در صورتی که ورودی عادی باشد و در وسط جمله باشیم نیز باید از \_input\_in\_mid استفاده شود:



بدین ترتیب فانکشنالیتی های چپ و راست رفتن روی ترمینال و وارد کردن متن در وسط جمله کاملا مدیریت میشوند. 

بخش 2:



برای این بخش از متغیرهای بالا استفاده کردیم.

در زبان c اوپراتور % به صورت باقی‌مانده تعریف شده است اگر سمت چپ این اوپراتور عددی منفی باشد حاصل نیز منفی خواهد بود پس تابع \_MOD را به صورت بالا تعریف کردیم که مقدار a را در پیمانه همنهشتی b به ما می‌دهد.

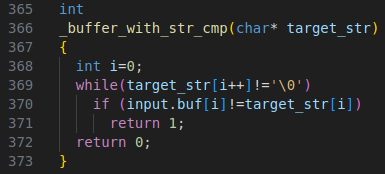
ثابت \_N\_HISTORY تعداد حداکثر دستوراتی است که با دستور history می‌توان به دست آورد که طبق صورت سوال برابر 11 گذاشتیم که شامل دستور فعلی و 10 دستور پیشین می‌شود.

متغیر \_history آرایه‌ای از جنس buffer\_ به طول 10 است که 10 input اخیر را در آن ذخیره می‌کنیم. با این متغیر به صورت حلقوی رفتار می‌کنیم یعنی بعد از 10 دوباره به خانه اول برمی2گردیم

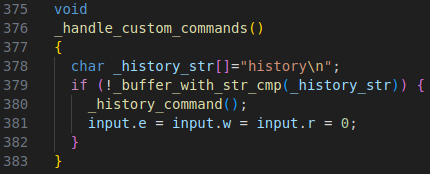
متغیر \_current\_history جایگاه خانه فعلی در \_history را نشان می‌دهد.

متغیر \_last\_history جایگاه آخرین دستور را نشان می‌دهد که اگر بخواهیم به دستورات گذشته جابه‌جا شویم، محتوای فعلی input را در این خانه از \_history ذخیره می‌کنیم.

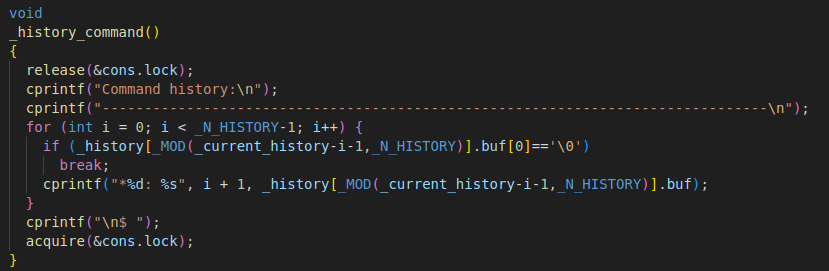
برای این بخش چند تابع مختلف تعریف کردیم. اولی تابع \_buffer\_with\_str\_cmp است. این تابع رشته‌ای را به عنوان ورودی می‌گیرد و محتوای بافر ورودی را با آن مقایسه می‌کند و در صورت یکسان بودن 0 وگرنه 1 برمی‌گرداند. از این تابع برای تشخیص اینکه محتوای بافر history است یا نه استفاده می‌کنیم.



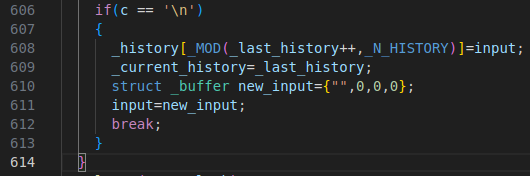
تابع بعدی \_handle\_custom\_commands است که در این تابع با کمک تابع قبل محتوای بافر ورودی با دستوراتی مشخص که در این پروژه فقط یک دستور history بود مقایسه می‌کند و در صورت یکسان بودن، توابع لازم را صدا می‌کند.



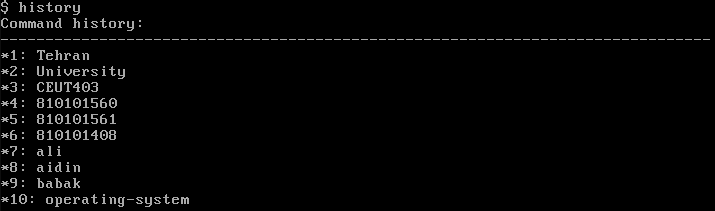
تابع بالا اگر بافر ورودی کلمه history بود، تابع \_history\_command را صدا می‌کند. این تابع محتوای تمام ورودی‌های ذخیره شده در \_history را چاپ می‌کند. البته در ابتدای کار کمتر از 10 ورودی در آرایه ذخیره شده است. پس این تابع هر دفعه بررسی می‌کند که اگر آن خانه آرایه خالی بود، متوقف شود.



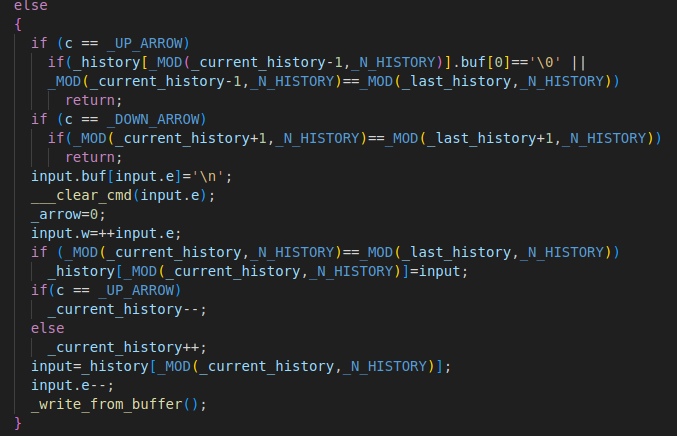
در تابع consoleread این شرط را تغییر دادیم. پس از این‌که حرف نیولاین وارد شد، این تابع ورودی فعلی را در خانه \_last\_history آرایه \_history ذخیره می‌کند و \_last\_history را یکی افزایش می‌دهد. همچنین با اضافه شدن input به \_history، \_current\_history آپدیت می‌شود تا نشان دهنده جایگاه ورودی جدید باشد و ورودی جدیدی با مقادیر خالی تشکیل می‌دهیم و input را برابر آن قرار می‌دهیم.



و بدین‌ترتیب دستور history به درستی کار می‌کند:



در بخش قبل قسمت مربوط به کلید فلش چپ و راست تابع \_arrow\_key\_handler را توضیح دادیم. حال قسمت مربوط به فلش بالا و پایین این تابع را توضیح می‌دهیم.

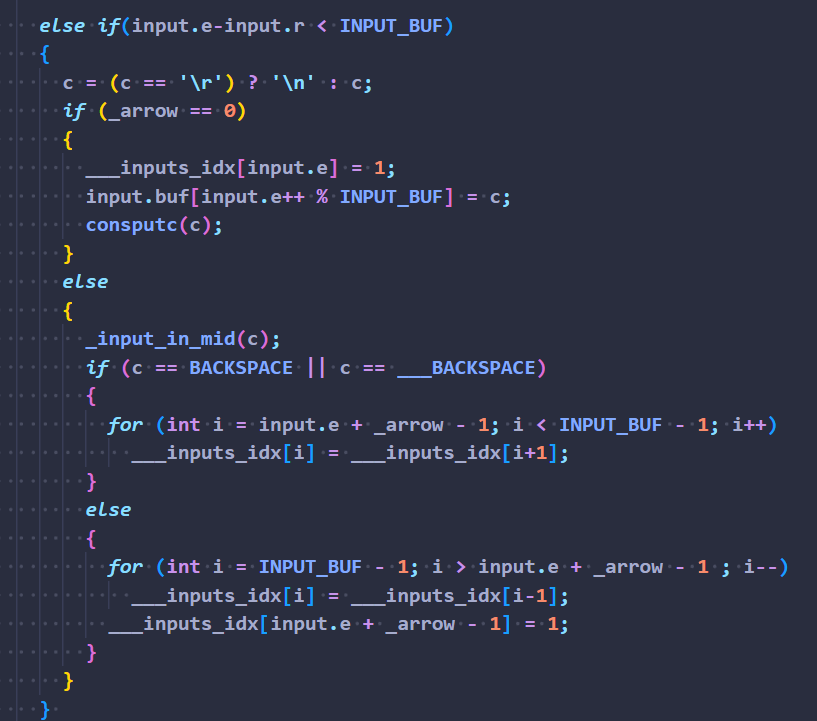
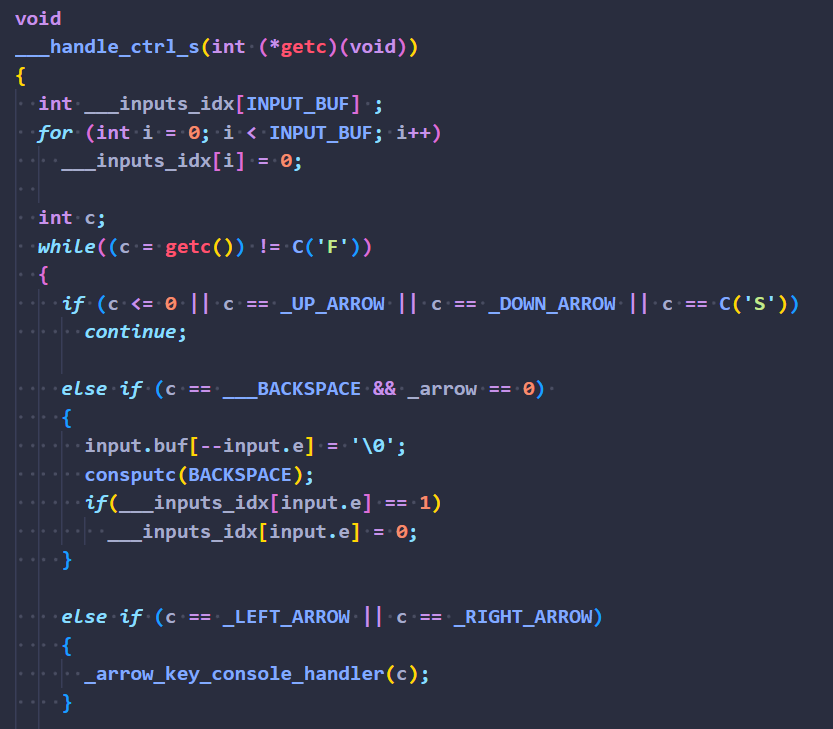


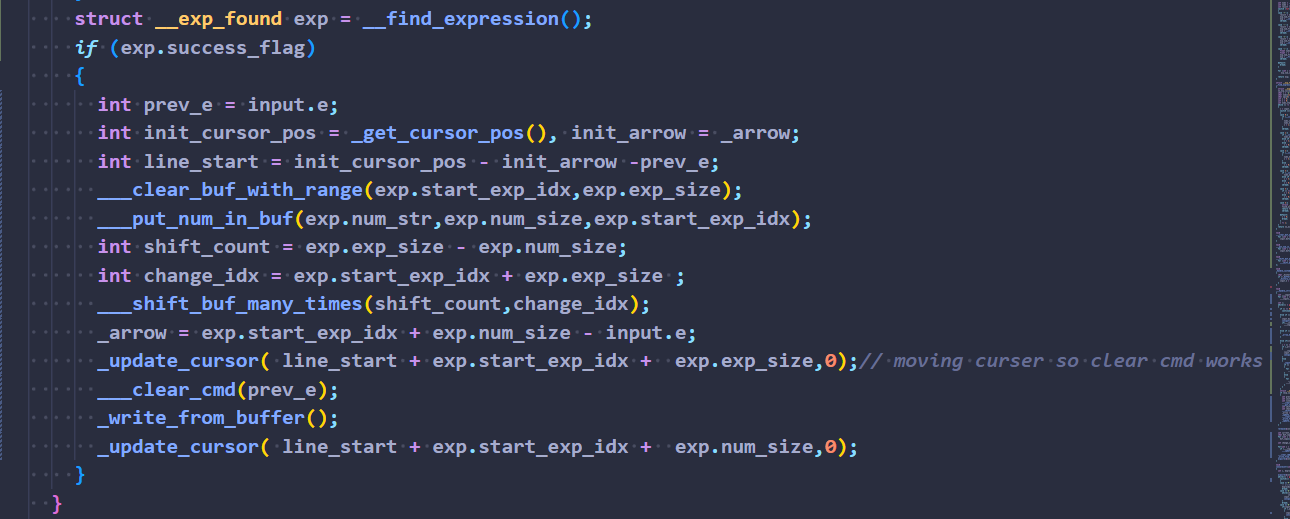
ابتدا بررسی می‌کنیم که خانه‌ای از \_history که سعی داریم به آن برویم خالی نباشد و همچنین به خانه انتهایی \_history نرسیده باشیم وگرنه نباید تغییری در ورودی فعلی بدهیم و همین‌جا return می‌کنیم. چون می‌خواهیم رشته فعلی کلا برود و تمام رشته‌های قبلی با نیولاین می‌رفتند و انتهایشان نیولاین قرار می‌گرفت، انتهای رشته نیز نیولاین می‌گذاریم. سپس صفحه را پاک می‌کنیم. بعد تنها در صورتی که مکان قبلی همان \_last\_history بود، تغییرات اعمال شده را در \_history ذخیره می‌کنیم. و در نهایت input را برابر input مورد نظر می‌گذاریم و محتوای آن را بر صفحه چاپ می‌کنیم.

بخش 3:

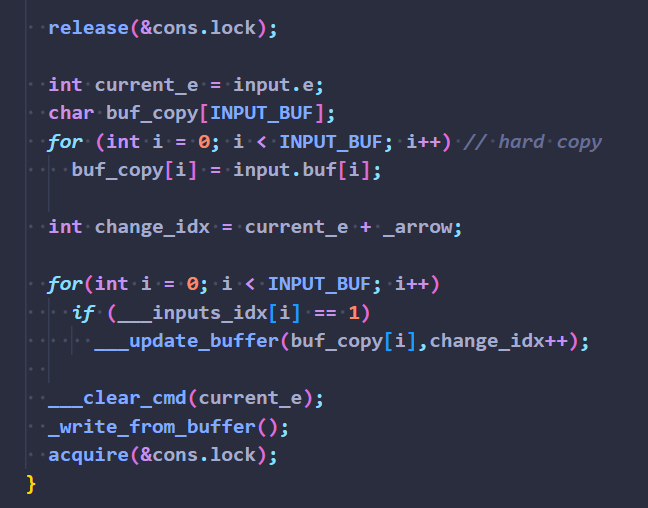
در این بخش باید ctrl + s و ctrl + f مدیریت شوند، که این کار توسط تابع \_\_\_handle\_ctrl\_s انجام میشود. این تابع از دو بخش تشکیل شده است:

1- بخش اول تا زمانی که ctrl + f نیامده وظیفه دارد کارایی عادی ترمینال را انجام داده و بافر را آپدیت کند. ابتدا آرایه ای برای ذخیره اندیس حروف ورودی تخصیص داده و سپس کاراکتر ورودی را تا رسیدن ctrl + f بررسی میکند. در صورتی که کاراکتر ورودی نامعتبر یا کلید های بالا یا پایین یا ctrl + s باشد کاری انجام نمیشود (چرا که در حین این عملیات نمیتوان به تاریخچه دسترسی پیدا کرد). در صورتی که کاراکتر وارد شده بک اسپیس باشد و اشاره گر در انتهای سمت راست قرار داشته باشد، یک کاراکتر بک اسپیس روی ترمینال نوشته، بافر را از انتها یکی کم میکند و در صورتی که کاراکتر آخر جزو کاراکتر های جدید الورود بود آن را از آرایه اندیس ها حذف میکند. در صورتی که کلید چپ و راست وارد شوند، مکان اشاره گر را آپدیت میکند و در غیر این صورت، کاراکتر وارد شده جدید را به انتهای راست یا در وسط جمله در حال نوشتن اضافه میکند و مقادیر اندیس های ورودی را یکی به چپ یا راست شیفت میدهد:

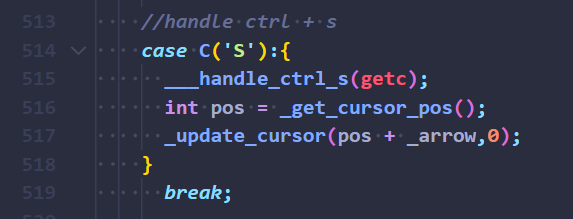
****

همچنین توابع سوال 4 نیز در انتهای این بلاک آورده شده تا ویژگی های آن قسمت نیز در اینجا قابل انجام باشد:

2- پس از دریافت ctrl + f، مقدار جدید e را در current\_e ذخیره میکند تا در آینده بعدا استفاده شود. سپس بافر را دیپ کپی کرده و در یک حلقه for مقادیر وارد شده توسط کاربر (که اندیس آنها در آرایه \_\_\_inputs\_idx ذخیره شده است) را از جایی که اشاره گر قرار دارد (با کمک \_arrow) در بافر وارد میکند. دست آخر ترمینال ورودی را پاک کرده (که سایز آن هنوز تغییر نکرده و برابر سایز ابتدای عملیات است) و مقدار بافر را دوباره روی ترمینال مینویسد:

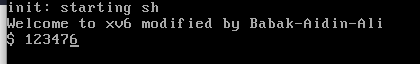


نهایتا این تابع هنگام دریافت ctrl + s توسط consoleintr صدا زده شده و پس از انجام عملیات، مقدار اشاره گر روی صفحه به اندازه \_arrow عقب میرود تا از ناهمگامی اشاره گر و بافر جلوگیری شود. بدین ترتیب کارایی های ctrl + s و ctrl + f به طور کامل مدیریت میشوند:



چند مثال از کارکرد این دستور:

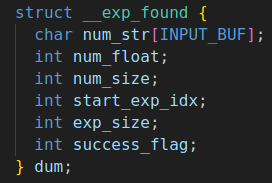
-> Ctrl + s -> ->

 Ctrl + f ->

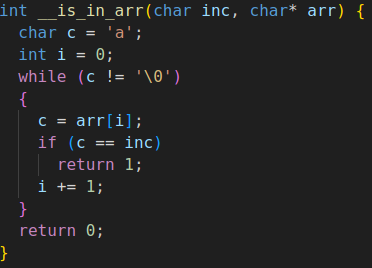
بخش 4:

در این بخش باید در صورت مشاهده یک الگوی خاص در ورودی محاسبات ریاضی را انجام دهیم که برای این منظور از موارد زیر کمک گرفته‌ایم:

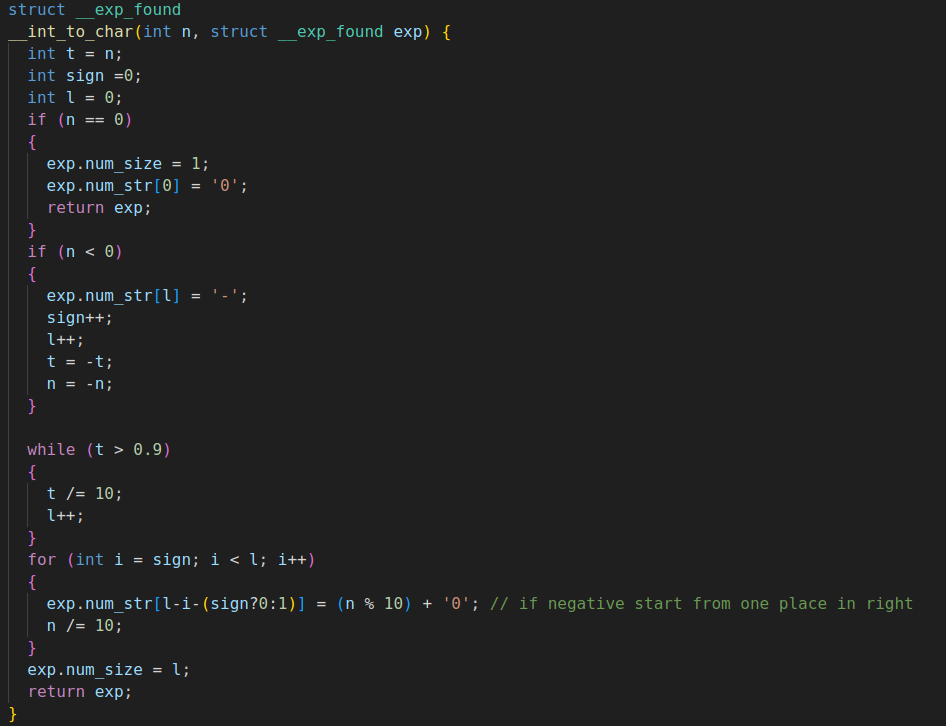
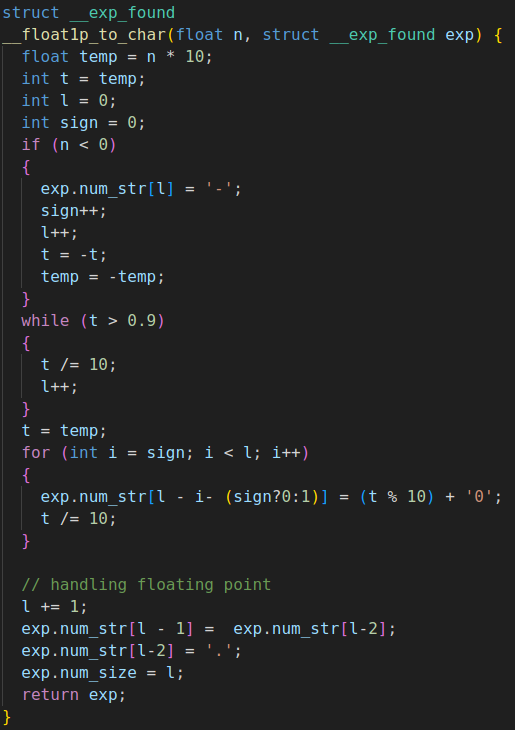
1. ساختار \_\_exp\_found که اگر الگوی مورد نظر توسط توابعی که در ادامه معرفی خواهند شد یافت شد، پاسخ آنرا بعد از محاسبه به صورت کاراکتر به همراه طول ورودی و خروجی و اندیس شروع الگو به توابع مربوط به انتقال به بافر تحویل میدهد تا آنها تغییرات لازم را ایجاد کنند.

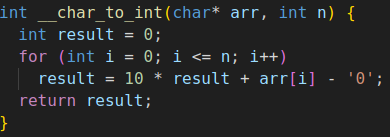


2. تابع \_\_is\_in\_arr وجود یک کاراکتر در یک آرایه از کاراکترها را تعیین میکند که در ادامه برای تشخیص اعداد و عملگرها به کار خواهد آمد

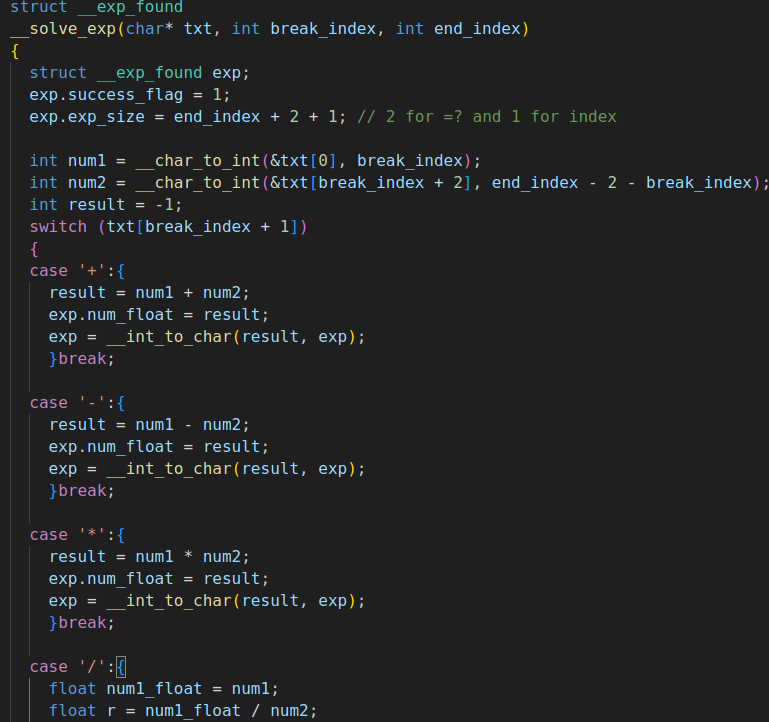
. 

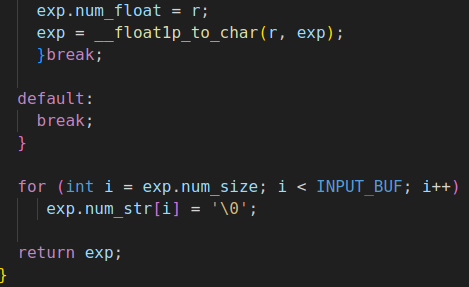
3. از آنجایی که محاسبات ما عددی هستند ولی ورودی و خروجی باید به صورت کاراکتر باشند؛ توابع \_\_char\_to\_int، \_\_int\_to\_char و \_\_float1p\_to\_char برای رسیدگی به این مورد تعریف شده‌اند.



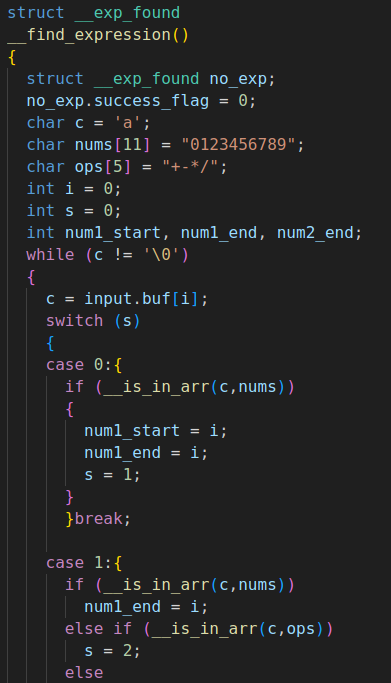
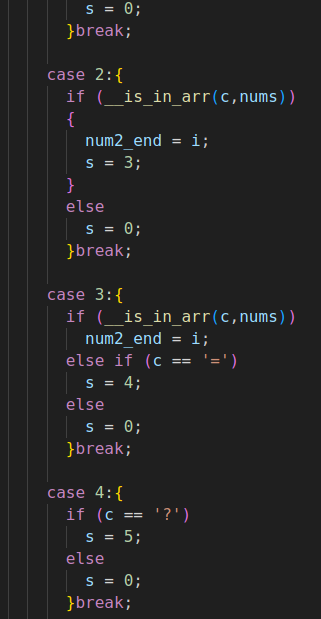


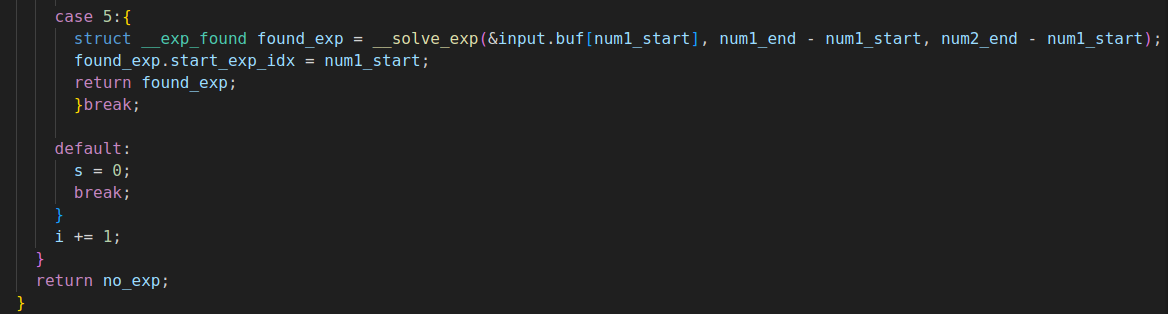
4. در ادامه تابع \_\_solve\_expرا برای انجام محاسبات بعد از کشف الگو داریم. این تابع کاراکتر اعداد را از الگو استخراج کرده و به کمک توابع بخش قبل آنها را به فرم int در می آورد؛ سپس با توجه به اینکه چه اپراتوری باید اجرا شود، محاسبه را انجام داده و حاصل را به کمک توابع بخش قبل به کاراکتر در فرم یک \_\_exp\_found خروجی میدهد.





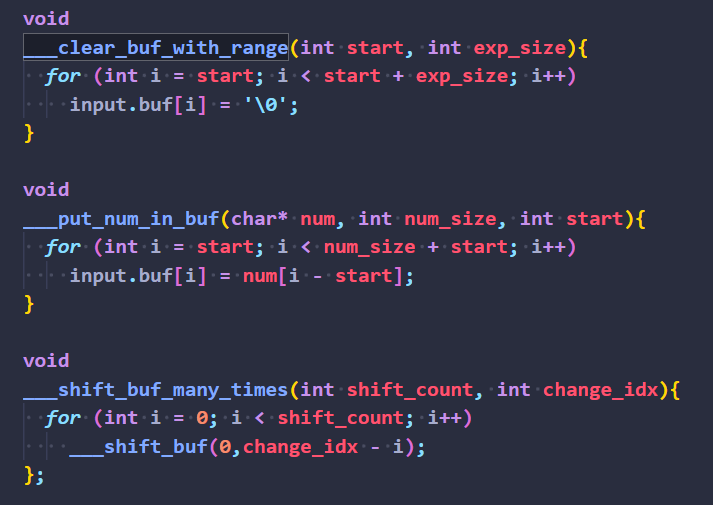
5. در نهایت تابع \_\_find\_expression را داریم این تابع روی input buffer به صورت یک state machine به دنبال الگوی مورد نظر سوال می گردد، هر زمان که به state آخر رسید(الگو را کامل مشاهده کرد)، اطلاعات الگو را به تابع \_\_solve\_exp که در بخش قبل گفته شد، میدهد و نتیجه این تابع را بعد تنظیم بعضی متغیر های ساختار \_\_exp\_found خروجی میدهد.





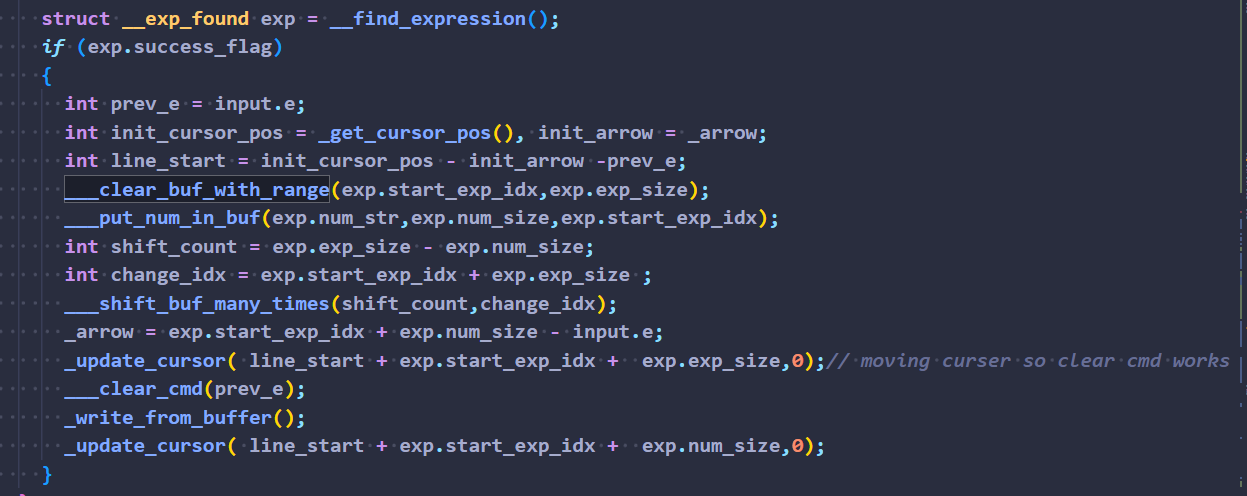
نهایتا با وارد کردن کد های استفاده شده به تابع consoelintr، پس از صدا کردن فانکشن ها میتوان نتایج دلخواه این بخش را گرفت. برای آنکه تغییرات اعمال شده بر روی اکسپرشن وابسته به توالی ورود کاراکتر ها نباشد، لازم است هر بار که تغییری روی بافر اعمال شد یک بار این توابع بافر را برای پیدا کردن اکسپرشن چک کنند.

ابتدا لازم است توابع مورد نیاز این عملیات توضیح داده شوند:



تابع اول وظیفه دارد بخشی از بافر را پاک کرده و با کاراکتر نال جایگذاری کند. تابع دوم وظیفه دارد یک عدد به فرم رشته را از یک مکان مشخص وارد بافر کند. تابع سوم نیز وظیفه دارد بافر را از یک نقطه مشخص شده به تعداد دلخواه به چپ شیفت دهد.

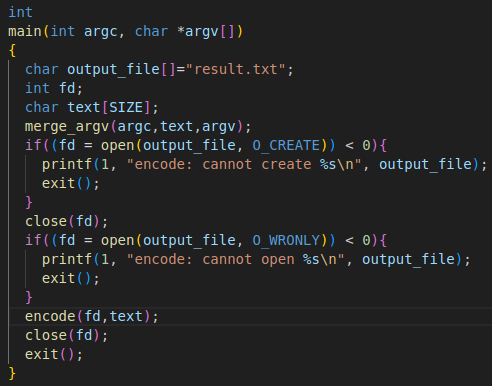
حال با کنار هم قرار دادن این توابع، عملیات مورد نیاز بر روی بافر را انجام میدهیم:



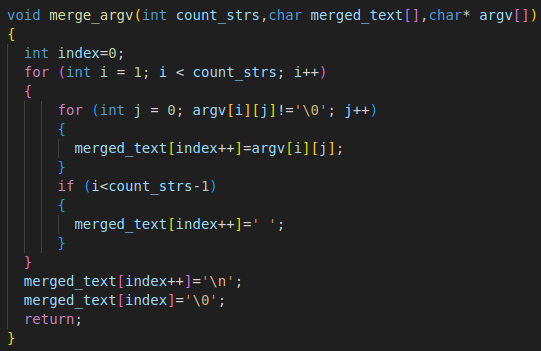
در ابتدا در صورتی که اکسپرشن با موفقیت پیدا شده بود، پوینتر e و مقدایر اولیه \_arrow و اشاره گر ذخیره شده، سپس مقدار ابتدای خط را حساب میکند تا بعد تر در جانمایی اشاره گر استفاده شود. در مرحله بعد بافر را از ابتدای شروع اکسپرشن به اندازه طول اکسپرشن خالی کرده و سپس عدد به دست آمده جواب را در ابتدای مکان خالی شده جایگذاری میکند. سپس از مکان انتهای اکسپرشن قبلی شروع کرده و فضای خالی بین انتهای عدد و مقدار خالی شده بافر را با کمک شیفت چپ پر میکند. سپس برای خالی کردن بافر، از آنجایی که در اثر عملیات های انجام شده ممکن است مکان اشاره گر به هم ریخته باشد، ابتدا مکان اشاره گر روی ترمینال را به انتهای قبلی اکسپرشن انتقال داده، سپس تابع \_\_\_clear\_cmd را صدا زده تا ترمینال را از مکان قبلی پوینتر e پاک کند (که هنوز روی صفحه دست نخورده اند). در مرحله بعد بافر آپدیت شده با تابع \_write\_from\_buffer بر روی ترمینال نوشته شده و نهایتا مکان اشاره گر به انتهای عدد جواب اکسپرشن منتقل میشود.

برنامه سطح کاربر:

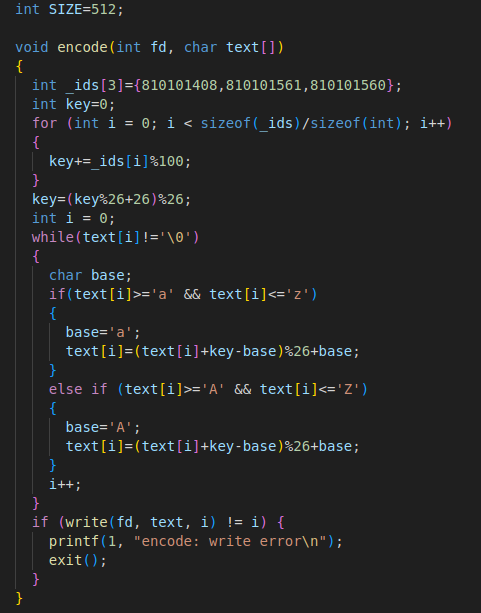
ابتدا فایل encode.c و decode.c را تشکلی دادیم. چون این دو فایل تقریبا یکسانند، تنها encode.c را توضیح می‌دهیم.



ابتدا تابع merge\_argv را صدا می‌زنیم. این تابع argv را می‌گیرد و آن را تبدیل به یک رشته می‌کند و برمی‌گرداند.

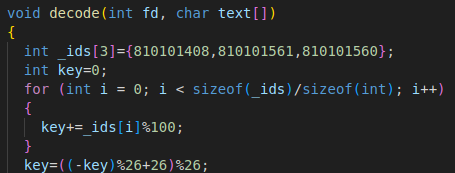


سپس فایل result.txt را تشکیل می‌دهیم و این فایل را باز می‌کنیم و تابع encode را فرا می‌خوانیم.

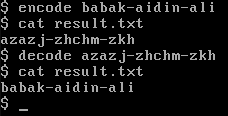


این تابع ابتدا کلید را مطابق الگوریتم خواسته شده حساب می‌کند. سپس به ازای هر حرف رشته، با توجه به بزرگ یا کوچک بودن حرف، آن را مطابق الگوریتم سزار تغییر می‌دهد و در پایان رشته حاصل را در فایل result.txt می‌نویسد.

فایل decode نیز دقیقا همین کار‌ها را انجام می‌دهد فقط key آن کمی متفاوت است.



و بدین ترتیب این دو برنامه سطح کاربر به درستی کار می‌کنند:



سوالات بخش مقدمه ای درباره سیستم عامل و xv6:

1- سه وظیفه اصلی سیستم عامل را نام ببرید؟

مدیریت منابع، قرار گرفتن میان کاربر و سخت افزار و مدیریت دسترسی ها و کاربران.

2- فایل های اصلی سیستم عامل xv6 در صفحه یک کتاب مربوطه لیست شده اند. به طور مختصر هر گروه را توضیح دهید. نام پوشه اصلی فایل های هسته سیستم عامل، فایل های سرایند و فایل سیستم در سیستم عامل لینوکس چیست؟ در مورد محتویات آن مختصرا توضیح دهید.

1. Basic Headers:

این بخش شامل تعاریف و ماکروهای مرتبط با سیستم‌عامل مورد نظر است. ماکروها برای ایجاد بخش‌های مختلف مانند بخش‌های قابل اجرا، خواندنی و نوشتنی استفاده می‌شوند، همچنین ساختارهای مختلفی مانند `rtcdate`، `buf`، `proc` و `segdesc` برای مدیریت زمان، بافرها و پردازش‌ها در این بخش تعریف شده‌اند.

2. Entering xv6:

شامل فایل های اسمبلی و فایل main.c میباشد که وظیفه آماده سازی سیستم و ورود به کرنل بعد از عملیات بوت شدن را داراست.

3. Locks:

شامل فایل spinlock میباشد که موجب همگام سازی میان پروسس ها و اینتراپت هندلر ها میشود، به طوری که با مدیریت دسترسی ها از خراب شدن یک پروسس توسط دیگری جلوگیری میکند.

4. Processes:

این بخش نقش مهمی در زمینه مدیریت پروسس ها ایفا میکند، به نحوی که عملیات های مهمی نظیر ایجاد و خاتمه پروسس ها، کانتکست سوییچ و حافظه مجازی در این بخش پیاده سازی شده اند که برای مولتی تسکینگ ضروری محسوب میشوند.

5. System Calls:

اینترفیس های سیستم کال، نحوه پیاده سازی آنها به همراه ثابت های مهم و نحوه پیاده سازی ترپ در این بخش مشخص شده اند.

6. File System:

این گروه وظیفه مدیریت عملیات های فایل (خواندن، نوشتن و...)، فایل سیستم و مدیریت دیسک را بر عهده دارد.

7. Pipes:

این بخش برای ارتباط میان پروسس ها طراحی شده و مسئول تبادل داده میان پروسس ها است.

8. String Operations:

این گروه توابعی برای عملیات بر روی رشته ها در سطوح پایین (از جمله کپی و مقایسه) ارائه میدهند.

9. Low-level Hardware:

این گروه مسئول رسیدگی به سخت افزار از جمله دستگاه های ورودی خروجی، رسیدگی به اینتراپت های سخت افزاری و رسیدگی به ملزومات مولتی پروسسور ها میباشد.

10. User-level:

شامل برنامه های سطح کاربر (از جمله shell ) بوده و مسئول تعامل برنامه های کاربر با کرنل میباشد.

11. Bootloader:

مسئول تعریف و پیاده سازی بوت لودر است که وظیفه لود کردن کرنل در مموری هنگام روشن شدن سیستم را داراست.

12. Link:

این بخش مسئول تعریف لینکر برای لینک کردن بخش های مختلف کرنل و جایگذاری هرکدام در فایل باینری نهایی کرنل میباشد.

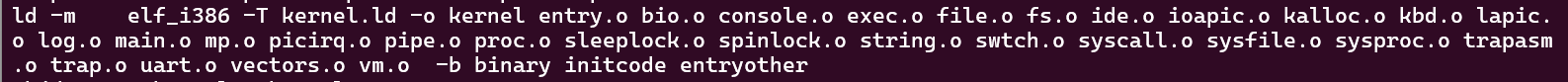
لینوکس:

فایل های سرایند، کرنل و فایل سیستم ها همگی داخل دیرکتوری /usr/src/linux قرار دارند. این دیرکتوری شامل بخش های مهمی از جمله documentation حاوی فایل های راهنما، source code حاوی فایل های اجرایی مهم از جمله درایور ها، کرنل و فایل سیستم، makefile، config حاوی فایل های کانفیگ کرنل و ... میباشد.

سوالات بخش کامپایل سیستم عامل:

3- دستور make -n را اجرا کنید. کدام دستور فایل نهایی هسته را میسازد؟

دستور زیر مسئول ساخت فایل نهایی است:



این دستور کلید اصلی در فرآیند تولید فایل نهایی کرنل است. این دستور لینکر میباشد که با استفاده از فایل kernel.ld که قوانین لینک کردن را تعریف می‌کند، تمام آبجکت فایل های تولید شده از مراحل قبلی را به فایل اجرایی کرنل سیستم‌عامل لینک میکند.

در makefile متغیر هایی به نام های UPROGS و ULIB تعریف شده است. کاربرد آنها چیست؟

UPROGS:

لیستی از برنامه های کاربر است که کامپایل و در تصویر نهایی فایل سیستم ها (fs.img) لینک میشوند. این برنامه ها در واقع همان برنامه هایی هستند که کاربر میتواند اجرا کند (مانند cat، ls، echo) و همانطور که توضیح داده شد، پس از تولید فایل سیستم این برنامه ها داخل تصویر فایل سیستم قرار گرفته و قابلیت اجرا شدن پیدا خواهند کرد.

ULIB

لیستی از کتابخانه های سطح کاربر است که توابع مورد نیاز برنامه های سطح کاربر (مخصوصا توابع نوشته شده در c )را داخل خود دارند. فایل هایی از جمله ulib، usys، printf و umalloc خدماتی از جمله سیستم کال ها، الوکیشن حافظه و ورودی خروجی سطح کاربر را ارائه میدهند که موجب میشود از نوشتن دوباره این توابع جلوگیری و نوشتن برنامه سطح کاربر بسیار آسان تر شود.

**سوالات اجرا بر روی شبیه ساز QEMU:**

5- دستور make qemu -n را اجرا نمایید. دو دیسک به عنوان ورودی به شبیه ساز داده شده است. محتوای آن ها چیست؟



دیسک های ورودی عبارتند از: xv6.img و fs.img. تصویر xv6.img شامل کد های اصلی کرنل سیستم عامل و بوت لودر برای بارگذاری سیستم عامل داخل حافظه میباشد. تصویر fs.img شامل فایل سیستم های مهم افزون بر ULIB و UPROGS است که عملیات های برنامه های سطح کاربر وابسته به آنها میباشد که در قسمت قبل توضیح داده شد.

**سوالات مراحل بوت سیستم عامل:**

8- علت استفاده از دستور objcopy در حین اجرای عملیات make چیست؟

این دستور برای تبدیل فایل های کامپایل شده به فرمت مناسب برای boot loader استفاده میشود. از آنجایی که boot loader انتظار مشاهده این فایل به صورت خام باینری دارد، این دستور تبدیل لازم را انجام میدهد.

13- کد bootmain.c هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در آدرس 0x100000 قرار می‌دهد. علت انتخاب این آدرس چیست؟

علت اینکه آنرا در آدرس های بالاتر قرار نمیدهیم این است که اگر ماشین کوچک بود و در حافظه محدودیت داشت نیز بتواند به آن دسترسی داشته باشد؛ همچنین آنرا در آدرس پایینتر نیز نمیتوان قرار داد چرا که آدرس های 0xa0000:0x100000 در اختیار دستگاه های ورودی و خروجی هستند.

**سوالات اجرای هسته xv6:**

18- همانطور که ذکر شد، ترجمه قطعه تأثیری 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؟

درست است که ترجمه قطعه تأثیری بر ترجمه آدرس ها نمیگذارد؛ اما همچنان پردازنده برای محافظت از آدرس های در انحصار هسته باید از سطح دسترسی دستورات مطلع باشد، به همین دلیل از این پرچم برای تعیین سطح دسترسی استفاده میشود.

**سوالات اجرای نخستین برنامه سطح کاربر:**

19- جهت نگهداری اطلاعات مدیریتی برنامه های سطح کاربر ساختاری تحت عنوان proc struct (خط ۲۳۳۶) ارائه شده است. اجزای آن را توضیح داده و ساختار معادل آن در سیستم عامل لینوکس را بیابید.

ما در این ساختار نام فرآیند، kernel stack، page table، وضعیت فرآیند ، id فرآیند، اندازه حافظه گرفته شده، فرآیند پدر، پرچمی برای تعیین خاتمه فرآیند، اشاره گری که اگر فرآیند معلق بود به آن اشاره کند، فایل های باز، پوشه فعلی، trap frame برای system call های فعلی و context برای تعویض را برای مدیریت برنامه ها داریم.

در linux، ساختار task\_struct را برای این منظور استفاده میکنیم.

23- 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 میشود؟

موارد مشترک بین هسته های پردازنده:

مدیریت حافظه، ساختار مدیریت پردازش، زمان بندی، ارتباطات بین پردازنده ها

در این سیستم زمان بند روی تمامی پردازنده ها مستقل اجرا میشود که منجر به موازی سازی و همزمانی پردازش میشود.

موارد اختصاصی:

Interrupt های محلی، context و structure های اختصاصی

هر پردازنده به interrupt های سخت افزار خودش جداگانه رسیدگی میکند.

**سوالات بخش اشکال زدایی:**

1. برای مشاهده Breakpointها از چه دستوری استفاده میشود؟

info breakpoint

1. برای حذف یک Breakpoint از چه دستوری و چگونه استفاده میشود؟

برای حذف breakpoint با شماره i تا شماره j می‌توانیم از دستور زیر استفاده کنیم:

del i-j

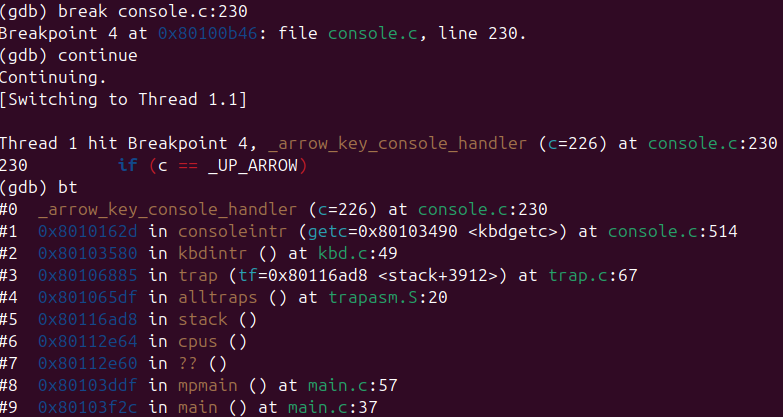
برای حذف breakpoint از خط i فایل f از دستور زیر استفاده می‌کنیم:

clear f:i

1. دستور زیر را اجرا کنید. خروجی آن چه چیزی را نشان میدهد؟

$ bt

برای بررسی دستور، یک breakpoint در تابع \_arrow\_key\_console\_handler فایل console.c گذاشتیم و وقتی به اینbreakpoint رسیدیم، دستور فوق را اجرا کردیم.

****

همان‌طور که می‌توان دید، این دستور، call stack برنامه را نشان می‌دهد. یعنی فراخوانی‌هایی که انجام شده به مکان فعلی برسیم.

1. دو تفاوت دستورهای x و print را توضیح دهید. چگونه می‌توان محتوای یک ثبات خاص را چاپ کرد؟ )راهنمایی: میتوانید از دستور help استفاده نمایید help x :و(help print

هدف این دو دستور متفاوت است. دستور x به ما اجازه می‌دهد که محتوای آدرس خاصی از حافظه را با فرمتی خاص بررسی کنیم. دستور print برای بررسی مقدار متغیرها و عبارات استفاده می‌شود.

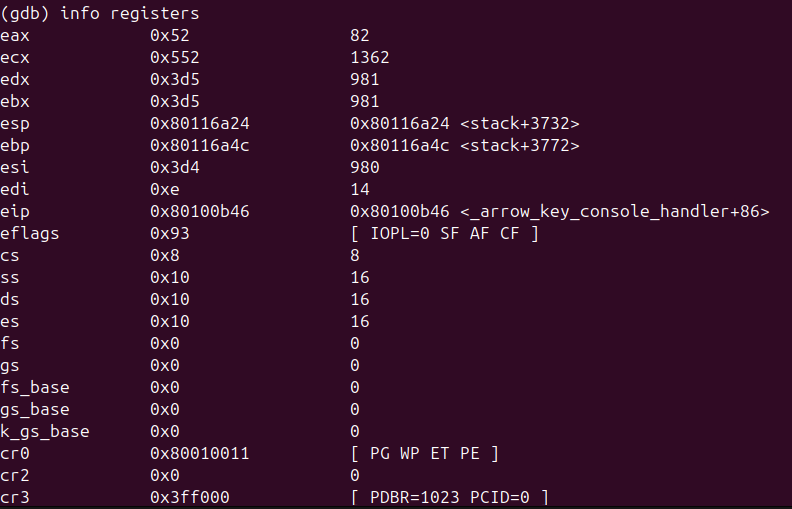
از لحاظ کاربرد نیز این دو دستور متفاوتند. از دستور x برای دیباگ بررسی سطح پایین خانه‌های حافظه استفاده می‌شود ولی از print برای دیباگ سطح بالا و مشاهده مقدار متغیرها استفاده می‌شود.

همچنین برای مشاهده محتوای یک ثبات خاص از دستور print با نام آن ثبات می‌توان استفاده کرد:

** **

1. برای نمایش وضعیت ثبات‌ها از چه دستوری استفاده می‌شود؟ متغیرهای محلی چطور؟ نتیجه این دستور را در گزارش‌کار خود بیاورید. همچنین در گزارش خود توضیح دهید که در معماری 86x رجیسترهای edi و esi نشانگر چه چیزی هستند؟

می‌توان از دستور info registers برای مشاهده وضعیت تمام ثبات‌ها استفاده کرد:

****

و از دستور info locals برای مشاهده متغیر‌های محلی استفاده می‌شود:

****

در معماری x86 این دو رجیستر به عنوان index registers شناخته می‌شوند. معمولا از این دو در انتقال داده و ایجاد تغییرات در رشته‌ها استفاده می‌شود.

معمولا از ثبات esi به عنوان پوینتر مبدا استفاده می‌شود. مثلا در کپی محتوا به خانه‌ای که از آن می‌خوانیم اشاره می‌کند.

edi به عنوان پوینتر مقصد استفاده می‌شود و به جایی که در آن می‌نویسیم اشاره می‌کند.

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

* توضیح کلی این struct و متغیرهای درونی آن و نقش آن‌ها
* نحوه و زمان تغییر مقدار متغیرهای درونی (برای مثال،e.input در چه حالتی تغییر میکند و چه مقداری میگیرد(

با gdb، watchpoint روی input قرار دادیم هر وقت تغییر کرد مقادیر آن را با کمک print بررسی کنیم. در ابتدا buf مقدار نال دارد و بقیه متغیرهای input هم صفر هستند. با هر حرفی که کاربر وارد می‌کند، آن حرف جدید به buf اضافه می‌شود و همچنینی یکی به مقدار e اضافه می‌شود. با هر backspace که کاربر می‌زند، حرف آخر از buf پاک می‌شود و یکی از e کم می‌شود و در تمام این مدت w و r صفر هستند. در نهایت وقتی کاربر enter بزند، نیولاین هم به buf اضافه می‌شود و e هم یکی اضافه می‌شود. سپس w برابر e می‌شود و بعد r هم یکی یکی زیاد می‌شود تا برابر e و wشود و در نهایت هر سه دوباره مثل ابتدا صفر می‌شوند و buf هم خالی می‌شود. با بررسی انجام شده این نتایج احتمالی را می‌شود گرفت:

متغیر buf آرایه‌ای است که حروف تایپ شده توسط کاربر در آن ذخیره‌ می‌شوند. با هر تغییری که کاربر ایجاد می‌کند، محتوای buf تغییر می‌کند.

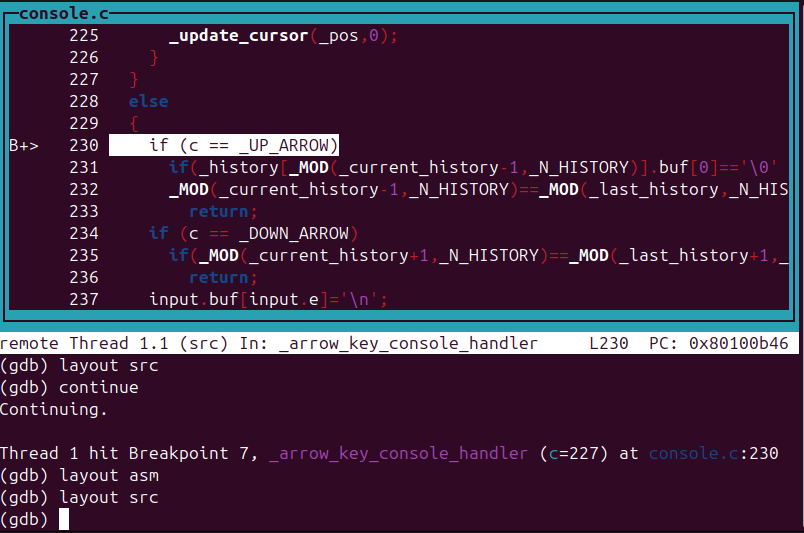
متغیر r اشاره به خانه بعدی از buf می‌کند که از آن می‌خواند. تنها موقع خواندن یعنی وقتی کاربر enter بزند افزایش میابد و بقیه اوقات صفر است.

متغیر w اشاره به اولین خانه از buf می‌کند. این متغیر هم همواره صفر است و فقط وقتی کاربر enter بزند، مقدارش برابر e می‌شود.

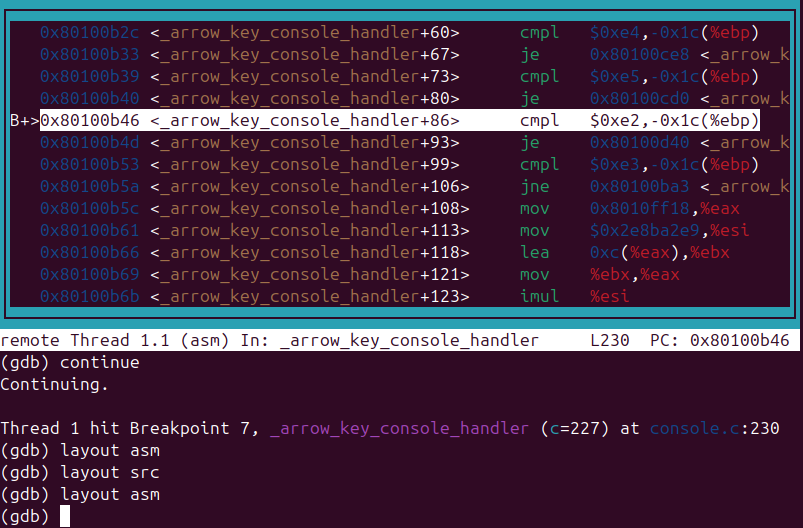
متغیر e اشاره به خانه‌ای از buf می‌کند که کاربر در حال ایجاد تغییراتی در آن است و با هر تغییر کاربر کم یا زیاد می‌شود و تغییر می‌کند.

1. خروجی دستورهای layout src و layout asm در TUI چیست؟

با استفاده از دستور layout src محیط terminal تغییر می‌کند به طوری‌که نیمه بالای ترمینال سورس کد برنامه و مکان فعلی را نشان دهد و نیمه پایین، مانند قبل بتوان دستورات gdb را وارد کرد:



دستور layout asm هم مانند دستور layout src است با این تفاوت که به جای سورس کد، کد اسمبلی نمایش داده می‌شود:

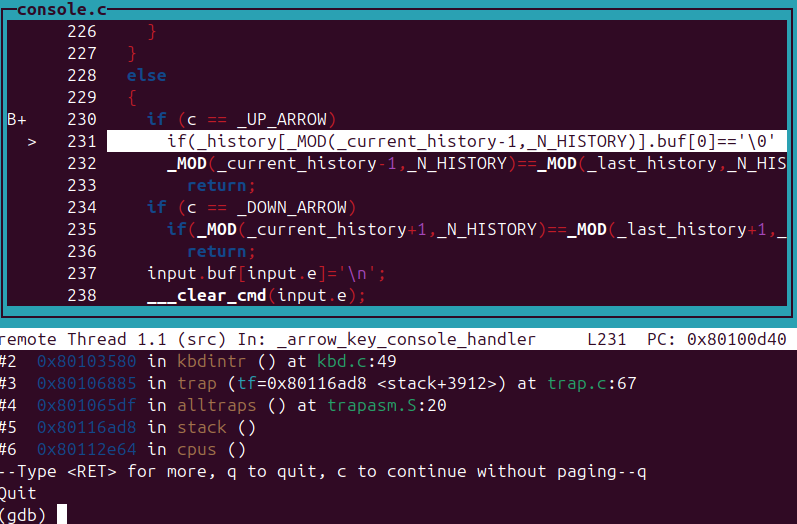


مشخص است که نمایش دستورات در این حالات به ردگیری برنامه کمک می‌کند چون می‌توانیم هم زمان کد را هم مشاهده کنیم.

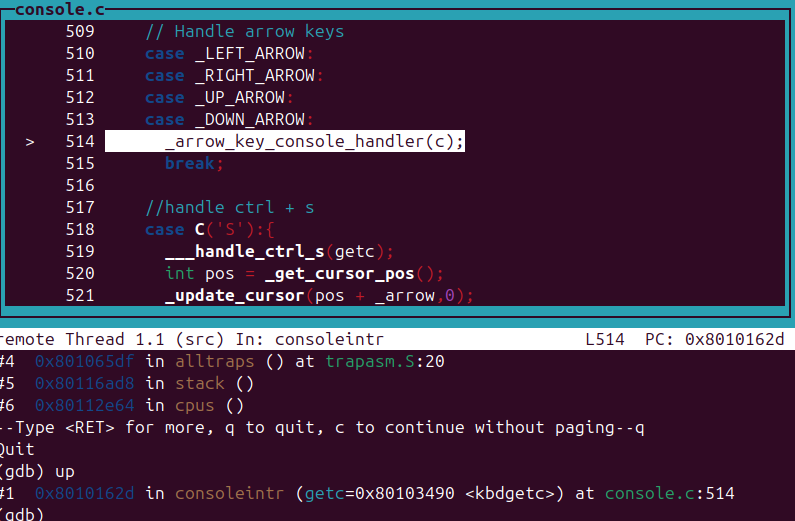
1. برای جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستوراتی استفاده میشود؟

در سوال 3 با دستور bt آشنا شدیم که زنجیره فراخوانی را به ما نمایش می‌داد. برای رفتن از تابع کنونی به تابع صدازننده از دستور up و برای برگشتن به تابع صدا زده شده از دستور down می‌شود استفاده کرد.

مکان فعلی:



برگشتن به تابع صدا زننده:



همچنین می‌توان با frame به تابع iام در زنجیره فراخوانی برویم:

