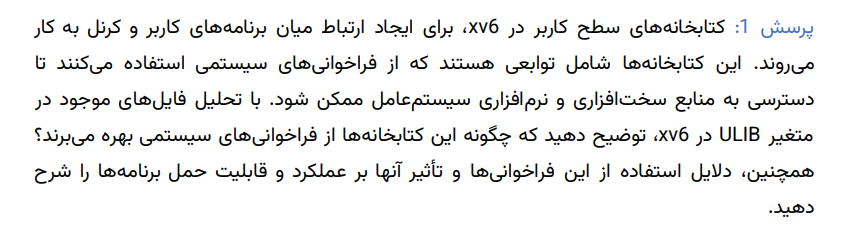
به نام خدا

اعضا گروه :

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

**پرسش ها :**



پاسخ:

در سیستم‌عامل 6v، فراخوانی‌های سیستمی معمولاً با استفاده از یک دستور trap یا وقفه‌ی نرم‌افزاری به کرنل ارسال می‌شوند. هر کدام از توابع کتابخانه سطح کاربر که به عملکردهای سیستمی نیاز دارند (مانند خواندن و نوشتن فایل‌ها، تخصیص حافظه، یا مدیریت پروسه‌ها)، یک فراخوانی سیستمی مناسب را اجرا کرده و سپس به کرنل منتقل می‌کنند. این توابع، شناسه‌ای منحصر به فرد (شماره‌ی فراخوانی سیستمی) و پارامترهای مورد نیاز را به کرنل ارسال می‌کنند، که کرنل این شناسه را تفسیر و سرویس مورد نظر را اجرا می‌کند. این نوع طراحی، به توابع کتابخانه‌های سطح کاربر اجازه می‌دهد تا بدون نیاز به دسترسی مستقیم به سخت‌افزار و جزئیات داخلی کرنل، به سرویس‌های سیستمی دسترسی پیدا کنند.



Ulib شامل آبجکت فایل های زیر است.به کد سی هر کدام مراجعه می کنیم و درباره آن ها توضیح می دهیم:

Ulib.c:

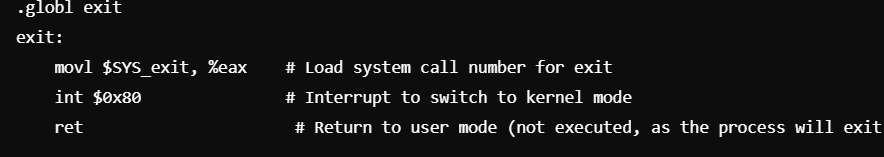
این فایل شامل توابع کاربردی برای کاربر است.این توابع شامل توابع مربوط به تغییر string ها مانند strlrn ، strcpy و strcmp است.گروه دیگر این توابع شامل توابع مربوط به مموری مانند memset و همچنین فانکشن های پرکاربردی برای تبدیل اینت به استرینگ و برعکس مانند atoi است.

در دو تابع gets و stat به ترتیب از syscall های read(برای خواندن از stdin) در اولی،open،close و fstat (اطلاعات فایل ها) استفاده شده.

Usys.s:

شامل یک سری interface برای استفاده از سیستم کال ها در سطح کاربر است.با استفاده از کد سیستم کال و اینتراپت ترنزیشن از سطح یوزر به سطح کرنل اتفاق می افتد.

این تصویر خلاصه ای از کد های در Usys.s است:



Printf.c:

این فایل شامل فانکشن هایی برای نوشتن است.با تابع printint در پروژه ی قبلی برای تغییر کنسول آشنا شدیم.توابعی مانند putc نیز در تابع console برای چاپ کاراکتر استفاده شده است.

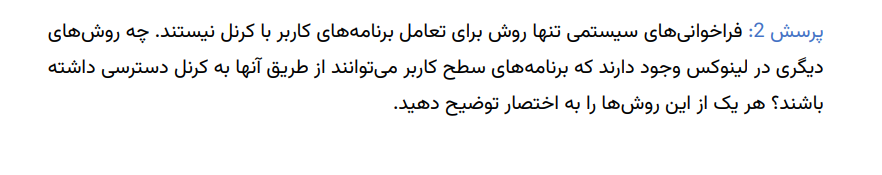
سیستم کال استفاده شده در اینجا سیستم کال write است که برای در فایل دیسکریپتور مورد نظر می نویسد.

Umalloc.c:

در اینجا تابع morecore را داریم که از سیستم کال sbrk برای افزایش سگمنت دیتای پراسس برای مقدار خاصی استفاده می کند.این سیستم کال ترمز برنامه(جایی که دیتا سگمنت تمام شده) را تغییر می دهد تا مموری اضافی فراهم شود. تابع mallocنیز از تابع morecore در مواردی استفاده می کند و غیر مسقیم با این سیستم کال سر و کار دارد.

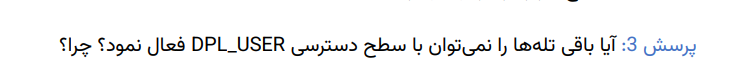
توابعی مانند free نیز وجود دارد که با کرنل کاری ندارد و صرفا حافظه را برای الوکیشن های بعد آزاد می کند.

سیستم کال ها یک اینترفیس فراهم می کنند که مستقل از معماری سخت افزار در صورت ساپورت سیستم عامل از این سیستم کال ها می توان از آن سیستم عامل استفاده کرد.از طرفی برنامه هایی که از سیستم کال ها استفاده می کنند لازم نیست جزییات دقیق مدیریتی سیستم عامل را بدانند و در صورت دانستن اینترفیس سیستم کال ها می توانند به راحتی با هر سیستم عاملی کار کنند.



در سیستم‌عامل لینوکس، روش‌های متعددی برای دسترسی از سطح کاربر به هسته وجود دارد. اگرچه فراخوانی‌های سیستمی (System Calls) یکی از رایج‌ترین این روش‌ها هستند، اما تنها روش نیستند. در ادامه به اختصار به توضیح روش‌های دیگر می‌پردازیم:

1. **فراخوانی‌های سیستمی (System Calls)**:
   * این روش اصلی‌ترین و مستقیم‌ترین راه ارتباط میان برنامه‌های کاربر و هسته است. در این روش، از طریق شماره‌ی فراخوانی سیستمی خاص و یک سری پارامترها، درخواست‌های سطح کاربر به هسته ارسال می‌شود و هسته به آن‌ها پاسخ می‌دهد. این فراخوانی‌ها اغلب شامل عملیات‌هایی مانند خواندن و نوشتن فایل، تخصیص حافظه، مدیریت پردازه‌ها و غیره هستند.
2. **فایل‌های دستگاه (Device Files)**:
   * در لینوکس، بسیاری از دستگاه‌ها و سخت‌افزارها به صورت فایل‌های دستگاه (مانند /dev/null، /dev/sda، و غیره) در سیستم فایل نمایش داده می‌شوند. از طریق خواندن و نوشتن این فایل‌ها، برنامه‌های کاربر می‌توانند با دستگاه‌ها و سخت‌افزارها ارتباط برقرار کنند. این روش امکان دسترسی غیرمستقیم به قابلیت‌های هسته را بدون استفاده مستقیم از فراخوانی‌های سیستمی فراهم می‌کند.
3. **فایل‌های مجازی در /proc و /sys**:
   * سیستم فایل‌های مجازی مانند /proc و /sys حاوی اطلاعات و پارامترهای تنظیمی درباره سیستم، هسته و پردازه‌ها هستند. این فایل‌ها به کاربران اجازه می‌دهند که اطلاعاتی از هسته بخوانند یا تنظیمات خاصی را تغییر دهند. برای مثال، کاربران می‌توانند با خواندن فایل‌های داخل /proc اطلاعاتی در مورد پردازه‌ها، مصرف حافظه، وضعیت CPU و دیگر موارد را بدست آورند.
4. **فراخوانی‌های کتابخانه‌ای (Library Calls)**:
   * در بسیاری از موارد، فراخوانی‌های کتابخانه‌ای (مانند توابع استاندارد libc در C) به عنوان واسطی برای دسترسی به سیستم عمل می‌کنند. این توابع در نهایت از فراخوانی‌های سیستمی استفاده می‌کنند، اما برای برنامه‌نویسان سطح کاربر، این روش به صورت مستقیم با فراخوانی‌های سیستمی درگیر نمی‌شود و از یک واسط سطح بالاتر بهره‌مند می‌شود.
5. **مکانیزم‌های ارتباط بین پردازه‌ای (IPC)**:
   * مکانیزم‌های IPC مانند صف‌های پیغام (Message Queues)، حافظه اشتراکی (Shared Memory) و لوله‌ها (Pipes) روش‌هایی برای ارتباط بین پردازه‌ها فراهم می‌کنند که گاهی به طور غیرمستقیم به هسته دسترسی دارند. هرچند برای استفاده از این مکانیزم‌ها معمولاً از فراخوانی‌های سیستمی نیز استفاده می‌شود اما همچنان برای کاربر ها مانند اینترفیس هستند.
   * . Message Queue (صف‌های پیغام): این روش اجازه می‌دهد که پردازه‌ها پیغام‌هایی را در صف‌هایی ذخیره کرده و پردازه‌های دیگر آن‌ها را دریافت کنند.
   * Shared Memory (حافظه اشتراکی): در این روش پردازه‌ها می‌توانند بخشی از حافظه را به اشتراک بگذارند و به سرعت داده‌ها را بین یکدیگر منتقل کنند.
   * Semaphores (سمافورها): برای هماهنگ‌سازی دسترسی به منابع مشترک بین پردازه‌ها استفاده می‌شود
6. **ioctl** 
   * این فراخوانی خاص برای اجرای عملیات‌های خاص و سفارشی در دستگاه‌ها و فایل‌های خاص استفاده می‌شود و دسترسی‌های ویژه‌ای به هسته و درایورهای دستگاه می‌دهد. فراخوانی ioctl به برنامه‌های کاربر اجازه می‌دهد که عملیات‌های سطح پایین و خاص دستگاه‌ها را اجرا کنند.
7. **سیگنال‌ها (Signals)**:
   * سیگنال‌ها یکی دیگر از روش‌های ارتباط بین کاربر و هسته هستند که معمولاً برای مدیریت اتفاقات ناگهانی و اعلان‌ها استفاده می‌شوند. هسته می‌تواند با ارسال سیگنال به پردازه‌ها، آن‌ها را از رخدادهایی مانند خطاها، وقفه‌ها و تکمیل عملیات‌ها مطلع سازد
   * • برخی از سیگنال‌های رایج عبارتند از:
   * SIGKILL: برای خاتمه دادن به یک پردازه.
   * SIGTERM: برای درخواست خاتمه پردازه به صورت نرم.
   * SIGSEGV: برای اطلاع از دسترسی غیرمجاز به حافظه.
   * • پردازه‌ها می‌توانند سیگنال‌ها را مدیریت کرده یا پردازش کنند، یا سیگنال‌ها می‌توانند به صورت پیش‌فرض در نظر گرفته شوند.
8. **کال‌بک‌ها (Callbacks) و توابع هوکینگ (Hooking Functions)**:
   * برخی مکانیزم‌ها در لینوکس به برنامه‌ها اجازه می‌دهند توابع خاصی را به عنوان "هوک" به وقایع خاصی از سیستم متصل کنند. برای مثال، برنامه‌های مدیریتی ممکن است به گونه‌ای تنظیم شوند که هنگام وقوع یک رویداد خاص (مانند اتصال یا قطع یک دستگاه) فراخوانی شوند.

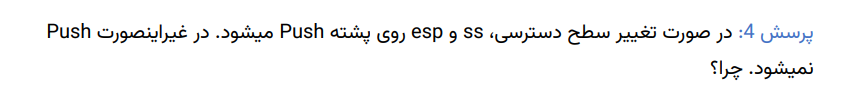


در سیستم‌عامل‌هایی مانند 6xv که از معماری‌های چند سطحی استفاده می‌کنند، مانند سطح 0 برای هسته (Kernel Mode) و سطح 3 برای کاربر (User Mode)، اصول امنیتی ایجاب می‌کند که دسترسی به هسته برای برنامه‌های سطح کاربر محدود شود. به عبارت دیگر، سطح دسترسی کاربر (DPL\_USER) که معادل سطح 3 است، نباید اجازه دسترسی مستقیم به هسته و اجرا کردن تله‌ها (traps) را بدهد.

اگر یک برنامه سطح کاربر به راحتی می‌توانست تله‌ها را فعال کند و به هسته دسترسی پیدا کند، این امر با اصول حفاظت از هسته در تضاد است. به این ترتیب، در معماری‌های مدرن، هسته سیستم‌عامل از پردازش‌ها و برنامه‌های سطح کاربر به طور منطقی جدا می‌شود تا امنیت سیستم حفظ شود.

برای مثال، اگر یک پردازنده بخواهد تله‌ای را فعال کند که منجر به ایجاد وقفه‌ای در سیستم شود، هسته سیستم‌عامل به طور مستقیم این دسترسی را به پردازش‌های سطح کاربر نمی‌دهد، چرا که چنین درخواست‌هایی باید توسط هسته و در سطح دسترسی بالاتر مدیریت شوند. این کار به این دلیل است که هسته از حملات احتمالی که ممکن است از سوی برنامه‌های سطح کاربر صورت گیرد، محافظت کند.

اگر چنین دسترسی‌هایی ممکن می‌شد، کاربر می‌توانست با بهره‌برداری از این تله‌ها به هسته حمله کند و به تمامی منابع سیستم، از جمله سخت‌افزار و نرم‌افزار، دسترسی پیدا کند. این موضوع باعث آسیب‌پذیری جدی در امنیت سیستم می‌شد و می‌توانست به هسته آسیب بزند یا دستکاری‌هایی در آن ایجاد کند. به همین دلیل، دسترسی مستقیم برنامه‌های سطح کاربر به هسته به طور عمدی مسدود می‌شود تا از این نوع حملات جلوگیری شود.



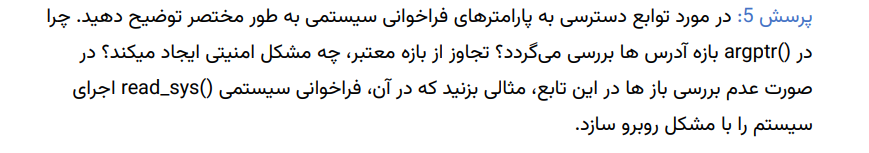
به طور کلی، در سیستم‌عامل‌هایی مانند 6xv که از سطوح مختلف دسترسی (کاربر و هسته) استفاده می‌کنند، دو پشته مجزا داریم: **پشته کاربر** و **پشته هسته** هر کدام از این پشته‌ها برای نگهداری اطلاعات مختلفی در زمان اجرا استفاده می‌شوند و برای جلوگیری از تداخل و محافظت از اطلاعات یکدیگر، از هم جدا هستند.

هنگامی که یک **trap** رخ می‌دهد و نیاز داریم سطح دسترسی از سطح کاربر به سطح هسته تغییر کند، نمی‌توانیم از پشته قبلی که مربوط به سطح کاربر است، استفاده کنیم. دلیل این امر این است که اطلاعات مربوط به اجرای کد در سطح کاربر ممکن است با اطلاعات مورد نیاز در سطح هسته تداخل کند و باعث از دست رفتن داده‌ها یا اجرای نادرست برنامه‌ها شود. بنابراین، باید اطلاعات مربوط به وضعیت اجرای برنامه، مانند **(SS)** و **پویشگر دستور (ESP)** را روی پشته هسته ذخیره کنیم. این کار به ما این امکان را می‌دهد که پس از پایان اجرای کد در سطح هسته و برگشت به سطح کاربر، بدانیم آخرین دستوراتی که اجرا کرده‌ایم چه بوده است و وضعیت سیستم را بازیابی کنیم.

این اطلاعات شامل مواردی چون آدرس دستور بعدی، وضعیت رجیسترها و پشته کاربر هستند که باید به دقت ذخیره و بعداً بازیابی شوند. اگر این اطلاعات ذخیره نشوند، پس از برگشت از حالت هسته به حالت کاربر، برنامه قادر نخواهد بود که به درستی ادامه دهد و می‌تواند باعث خطاهای جدی در عملکرد سیستم شود.

اما وقتی که سطح دسترسی تغییر نمی‌کند و از یک پشته واحد برای اجرای دستورات استفاده می‌شود (مثلاً زمانی که هنوز در سطح کاربر هستیم و به هیچ کدام از منابع سطح هسته دسترسی نداریم)، نیازی به ذخیره اطلاعات اضافی مانند SS و ESP نیست، چرا که وضعیت پشته تغییر نمی‌کند و می‌توانیم به سادگی ادامه‌ی اجرای برنامه را بدون نگرانی از تداخل با پشته‌های دیگر ادامه دهیم.

در نتیجه، ذخیره و بازیابی اطلاعات پشته زمانی که سطح دسترسی تغییر می‌کند، امری حیاتی برای جلوگیری از از دست رفتن اطلاعات در زمان تغییر سطح دسترسی است.

 **تابع argint**

* این تابع برای دسترسی به پارامترهای عددی فراخوانی‌های سیستمی استفاده می‌شود. تابع argint دو ورودی می‌گیرد:
  + **ورودی اول:** شماره پارامتر مورد نظر در لیست پارامترهای فراخوانی سیستمی.
  + **ورودی دوم:** یک متغیر از نوع int که به صورت reference by-pass به تابع داده می‌شود تا مقدار پارامتر در آن ذخیره شود.
* **عملکرد:** این تابع ابتدا با استفاده از تابع fetchint و با استفاده از مقادیر پشته (مانند esp, tf و ...) آدرس پارامتر مورد نظر را پیدا کرده و محتوای آن را در متغیر ورودی دوم ذخیره می‌کند. اگر عملیات با موفقیت انجام شود، مقدار صفر (0) برگشت داده می‌شود و اگر پارامتر مورد نظر وجود نداشته باشد یا عملیات با خطا مواجه شود، مقدار -1 برگشت داده می‌شود.

**تابع argptr**

* این تابع برای دسترسی به پارامترهایی استفاده می‌شود که به صورت اشاره‌گر (pointer) یا رشته‌ها (string) در فراخوانی‌های سیستمی به هسته ارسال می‌شوند. argptr سه ورودی دارد:
  + **ورودی اول:** شماره پارامتر مورد نظر.
  + **ورودی دوم:** یک اشاره‌گر که به صورت reference by-pass داده می‌شود و در آن آدرس پارامتر ذخیره خواهد شد.
  + **ورودی سوم:** سایز پارامتر که به تابع داده می‌شود تا در صورت نیاز اندازه آن چک شود.
* **عملکرد:** تابع argptr ابتدا از طریق تابع argint، آدرس خانه اول پارامتر (که معمولاً به صورت اشاره‌گر به یک آرایه یا رشته است) را به دست می‌آورد. سپس مقدار این اشاره‌گر را در ورودی دوم ذخیره می‌کند. اگر عملیات موفقیت‌آمیز باشد، مقدار صفر (0) برگشت داده می‌شود، در غیر این صورت، اگر پارامتر وجود نداشته باشد یا عملیات با خطا مواجه شود، مقدار -1 برگشت داده می‌شود.

**تابع argstr**

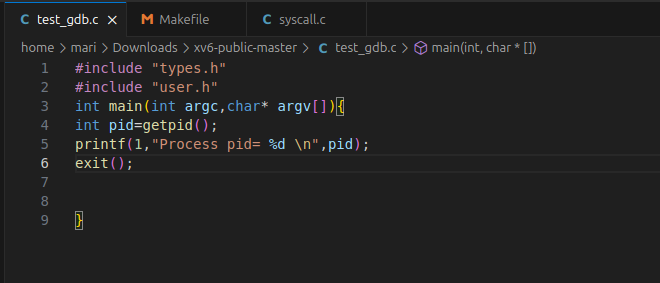
* این تابع به منظور دسترسی به پارامترهای رشته‌ای (string) فراخوانی‌های سیستمی استفاده می‌شود. در واقع، وقتی یک رشته (مثل نام فایل یا دستور) به عنوان پارامتر به هسته ارسال می‌شود، این تابع می‌تواند رشته را استخراج کند.
  + **ورودی اول:** شماره پارامتر (شماره‌ای که نشان می‌دهد کدام پارامتر از درخواست فراخوانی سیستم مدنظر است).
  + **ورودی دوم:** یک اشاره‌گر به char\* که در آن رشته ذخیره می‌شود.
* **عملکرد:** پس از بررسی صحت داده‌ها، رشته در اشاره‌گر char\* ذخیره می‌شود و باید به طور مناسب با کاراکتر null ('\0') پایان یابد (یعنی رشته باید با نال تمام شده باشد). در صورت موفقیت، مقدار صفر (0) برگشت داده می‌شود، در غیر این صورت، اگر پارامتر رشته‌ای وجود نداشته باشد، مقدار -1 برگشت داده می‌شود.

**argfd**

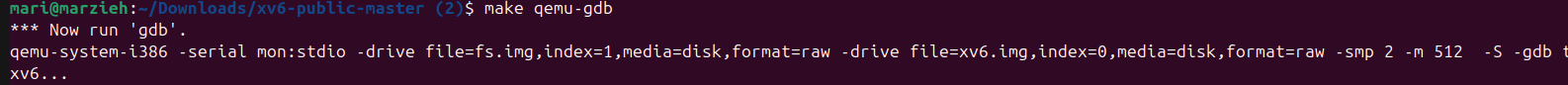
* این تابع برای استخراج اطلاعات مربوط به **فایل دسیپتور** (file descriptor) در فراخوانی‌های سیستمی استفاده می‌شود. این تابع در فایل sysfile.c تعریف شده است و معمولاً برای انجام عملیات‌های مربوط به فایل‌ها به کار می‌رود.
  + **ورودی اول:** شماره پارامتر که فایل دسیپتور مورد نظر را مشخص می‌کند.
  + **ورودی دوم:** یک اشاره‌گر به یک متغیر که دسیپتور فایل (file descriptor) را در آن ذخیره می‌کند.
  + **ورودی سوم:** یک اشاره‌گر به ساختار داده‌ای که اطلاعات فایل (مثل نام فایل، وضعیت فایل و غیره) را در خود نگه می‌دارد.
* **عملکرد:** این تابع با دریافت ورودی‌ها، اطلاعات مربوط به دسیپتور فایل و ساختار داده‌ای مرتبط را بازیابی کرده و آن‌ها را در مقادیر اشاره‌گرهای ورودی دوم و سوم ذخیره می‌کند. اگر عملیات موفقیت‌آمیز باشد، مقدار صفر (0) برگشت داده می‌شود. در غیر این صورت، اگر عملیات با خطا مواجه شود، مقدار -1 برگشت داده می‌شود.

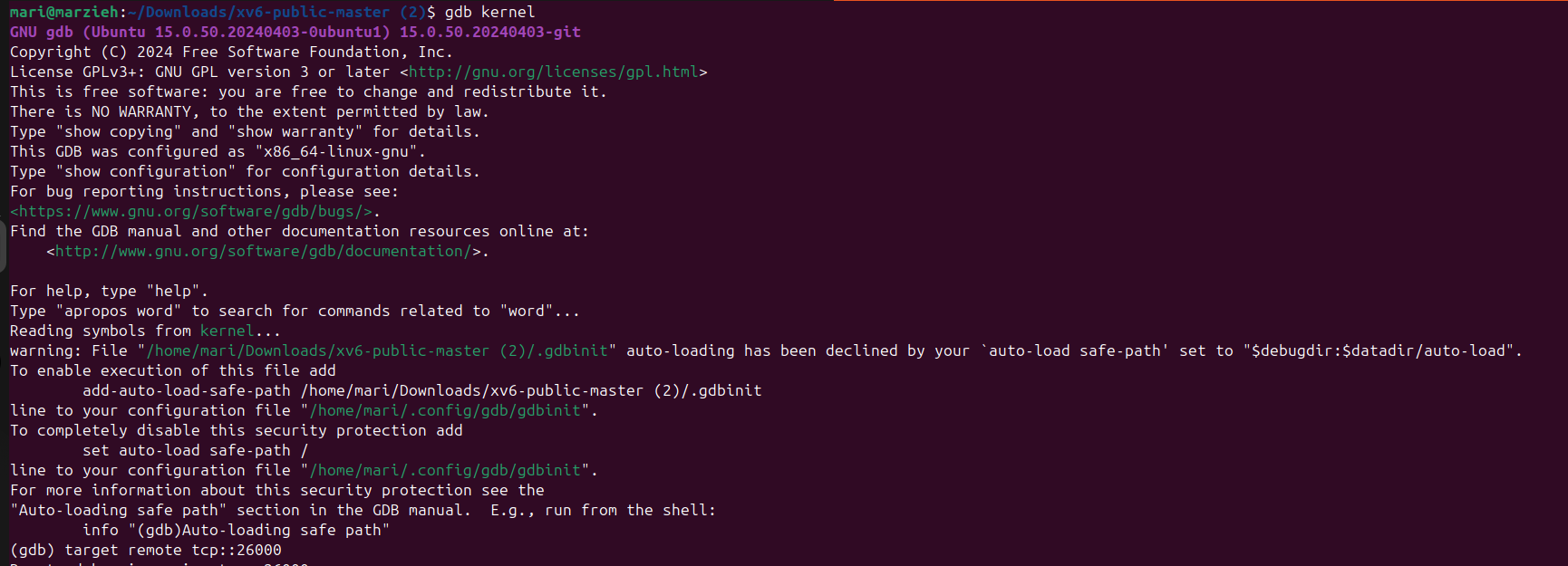
در هنگام انجام عملیات خواندن از یک فایل با استفاده از فراخوانی read\_sys، باید آدرس حافظه‌ای که قرار است داده‌ها در آن ذخیره شوند، به دقت چک شود. اگر این آدرس به اشتباه به منطقه‌ای از حافظه اختصاص یابد که متعلق به پردازه جاری نیست، داده‌های خوانده شده در بخش‌های نادرست از حافظه ذخیره خواهند شد..یا ممکن است در پردازش‌های هسته خطا ایجاد شود و سیستم به حالت trap وارد شود.



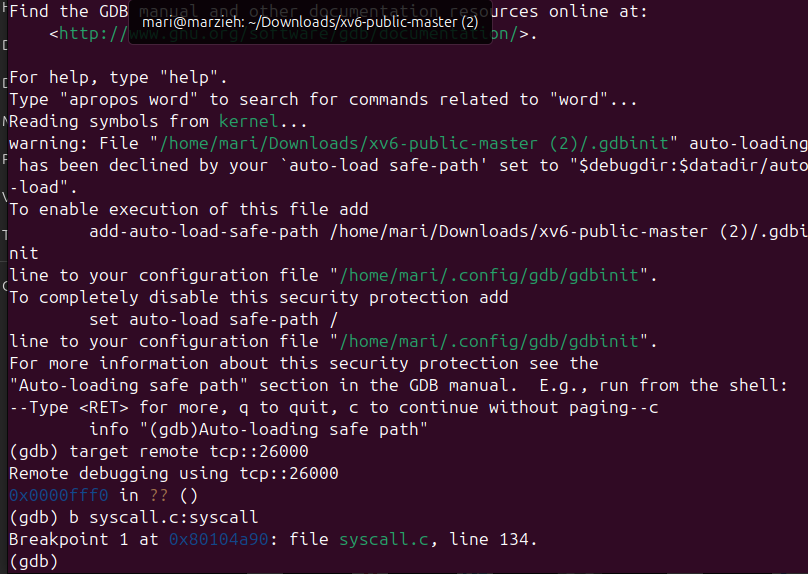
ابتدا برنامه ی سطح کاربر زیر را نوشته و طبق روال به xv6 اضافه می کنیم:  


حال دستور make qemu-gdb را در یک ترمینال ران کرده و در ترمینال دیگر دستور gdb kernel را ران می کنیم:

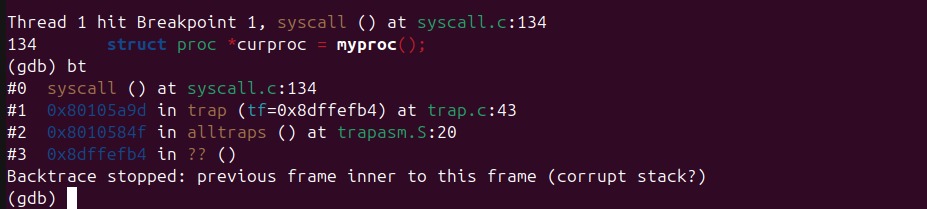




حال طبق روال دستور target remote tcp را برای اتصال به سیستم عامل می زنیم و سپس breakpoint را بر سر تابع syscall در فایل syscall.c می گذاریم:



برای دیدن استک فریم های مربوط به اجرا از دستور bt استفاده می کنیم:



زمانی که دستورbt در gdb اجرا می‌شود، اطلاعات موجود در **پشته فراخوانی توابع** (stack call) را نمایش می‌دهد. این پشته‌ی فراخوانی مانند یک استک (stack) عمل می‌کند و به ما امکان می‌دهد تا روند اجرای برنامه را از نقطه فعلی تا ابتدای اجرای برنامه بررسی کنیم. به عبارت دیگر، این پشته شامل لیستی از توابعی است که به ترتیب فراخوانی شده‌اند تا برنامه به وضعیت فعلی برسد.

**پشته فراخوانی** هر بار که یک تابع در برنامه فراخوانی می‌شود، یک بلاک حافظه‌ای به نام **stack frame** برای آن تابع در بالای استک اضافه می‌کند. این stack frame شامل اطلاعاتی از قبیل:

* **آدرس بازگشت**: آدرسی که پس از اجرای تابع، برنامه به آن بازمی‌گردد.
* **آرگومان‌های تابع**: مقادیری که در هنگام فراخوانی به تابع ارسال شده‌اند.
* **متغیرهای محلی تابع**: داده‌هایی که در درون تابع تعریف و ذخیره شده‌اند.
* **اطلاعات دیباگ**: آدرس تابع در حافظه و مسیر فایل سورس کد.

این فرآیند به ما کمک می‌کند تا مسیر دقیق اجرای برنامه را درک کنیم و بفهمیم کدام توابع و با چه ترتیب و پارامترهایی فراخوانی شده‌اند. با اجرای دستور bt، اطلاعات پشته فراخوانی از جمله:

* **نام توابع فراخوانی شده**
* **آدرس دقیق هر تابع در سورس کد**
* **آدرس حافظه هر تابع**
* **آرگومان‌های ورودی توابع و مقادیر آن‌ها**

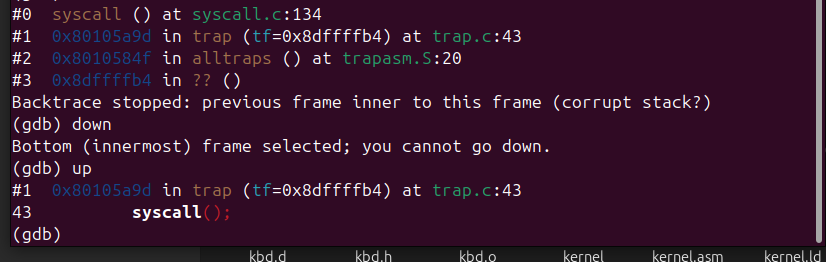
به ترتیب نمایش داده می‌شوند.

برای اضافه کردن یک سیستم کال (System Call) جدید در سیستم‌عامل، مراحل مختلفی طی می‌شود که به شرح زیر هستند:

1. **اختصاص شناسه و شماره‌ی سیستم کال**  
   هر سیستم کال در سیستم‌عامل با یک شناسه‌ی عددی یکتا شناسایی می‌شود. این شماره‌ها در فایل‌های syscall.h تعریف می‌شوند. اگر بخواهیم یک سیستم کال جدید اضافه کنیم، باید آن را در این لیست ثبت کنیم و یک شناسه‌ی یکتا به آن تخصیص دهیم.پرتوتایپ سیستم کال های مربوط به بخش کاربر نیز در user.h موجود است.
2. **تعریف سیستم کال در سطح اسمبلی**  
   سیستم کال‌ها برای تعامل مستقیم با سخت‌افزار به زبان اسمبلی تعریف می‌شوند و این تعاریف در فایل usys.s قرار دارند. پپ
3. **رجیستر eax و ذخیره مقدار بازگشتی**  
   در xv6، رجیستر eax برای ذخیره شماره‌ی سیستم کال استفاده می‌شود. هنگامی که سیستم کال فراخوانی می‌شود، شماره آن در eax ذخیره شده و قبل از اجرای دستور int 64 که سیگنال اجرای سیستم کال را صادر می‌کند، قرار می‌گیرد. پس از اجرای این دستور، مقدار یا وضعیت بازگشتی از سیستم کال نیز در eax ذخیره می‌شود.
4. **اجرای دستور int 64 و ورود به بردار 64**  
   پس از فراخوانی سیستم کال و اجرای دستور int 64، پردازنده به (interrupt vector) هدایت می‌شود. سپس در این مرحله (vector address) به‌صورت خودکار در استک (stack) ذخیره شده و به تابع alltraps که در فایل vector.s تعریف شده، (jump) می‌شود.
5. **ایجاد trap frame توسط alltraps**  
   در alltraps، ابتدا trap frame ساخته می‌شود که شامل اطلاعات لازم از وضعیت سیستم در زمان فراخوانی سیستم کال است. این trap frame در استک قرار می‌گیرد و سپس alltraps تابع trap را که در فایل trap.c قرار دارد، فراخوانی می‌کند.
6. **فراخوانی trap و تشخیص سیستم کال با شماره 64**  
   در تابع trap، trap frame که قبلاً در استک قرار داده شده بود به‌عنوان trap frame فرآیند جاری تنظیم می‌شود. با استفاده از شماره 64، مشخص می‌شود که یک سیستم کال فراخوانی شده است و سپس تابع syscall فراخوانی می‌شود.
7. **اجرای تابع syscall**  
   تابع syscall پردازش سیستم کال را آغاز می‌کند. این تابع، در واقع تمامی مراحل پردازش سیستم کال را انجام می‌دهد و پس از آن نتیجه به فراخواننده بازمی‌گردد.

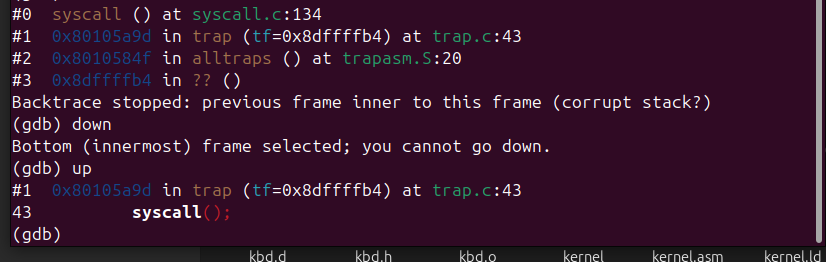
همان‌طور که در تصویر نیز قابل مشاهده است، مراحل مختلف سیستم کال شامل فراخوانی از طریق trap و alltraps هستند که به ترتیب در این مسیر قرار گرفته‌اند و تا اجرای کامل syscall طی می‌شوند.

حال دستور down را می زنیم:

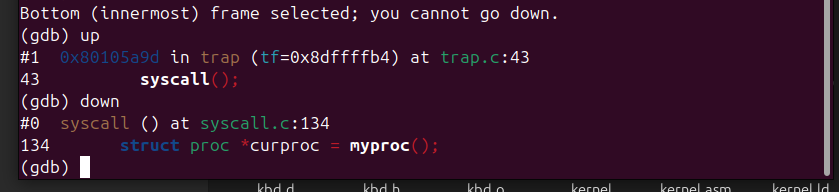


همانطور که مشاهده می کنید ما نمی توانیم به فریم درونی تر برویم.دلیل آن است که فریم درونی تر تابعی است که آخرین تابع ما صدا زده که بدیهتا آخرین تابع ما تابعی را صدا نزده بنابراین نمی توانیم پایین تر برویم.

دستور down و up همانطور که در آزمایش قبل نیز توضیح داده شد برای جابجای بین استک فریم ها می باشد. دستور down به استک فریم callee و دستور up به استک فریم caller می رود.



حال اگر بعد از دستور up دستور down را بزنیم مشکلی پیش نمی آید:

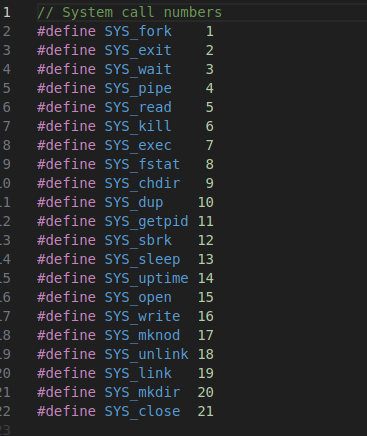


قبلا درباره دستور print که به ما محتوای اکسپرشن ها را نشان می دهد صحبت کردیم.از این دستور برای دیدن مقدار eax که در آن سیستم کال پراسس مورد نظر استفاده می کنیم:

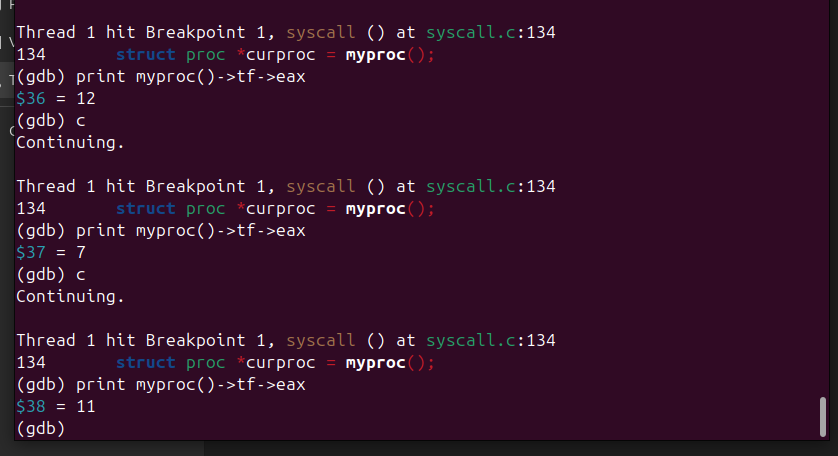


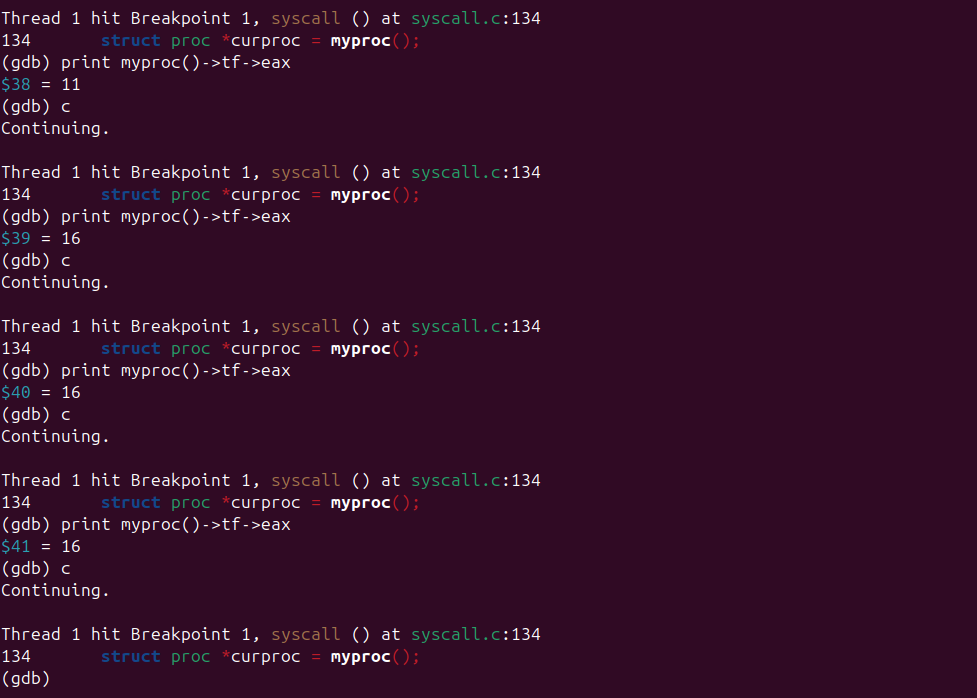
شاید این سوال پیش بیاید که چرا هیچ کدام از کد های سیستم کال ،مقداری برابر 11 ندارد ؟دلیل آن است که ابتدا پراسس ابتدایی که شامل بوت سیستم و پراسس های دیگری باید اجرا شوند تا نوبت به برنامه سطح کاربر ما برسد.

آیدی سیستم کال های ما در سیستم عامل xv6 در فایل syscall.h قرار دارد که به شرح زیر است:



بنابراین آن قدر دستور c که معادل continue است را می زنیم که به ورودی گرفتن از کاربر و برنامه ی سطح کاربر برسیم:



طبق برنامه باید انتظار داشته باشیم بعد از سیستم کال 11 برنامه شروع به نوشتن کند و کد 16 را ببینیم:  


که نتیجه مورد انتظار ماست.

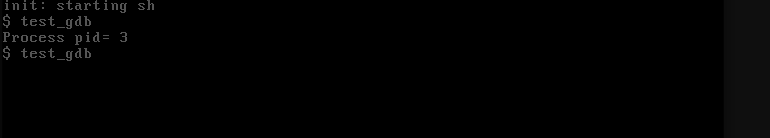
همچنین بعد از نوشتن همه ی کاراکتر ها باید برنامهexit کند که کد 2 را ببینیم:



همچنان نتیجه مورد انتظار است.

16 بعد برای نوشتن $ در خط بعد است.

خروجی نهایی به شرح زیر است:

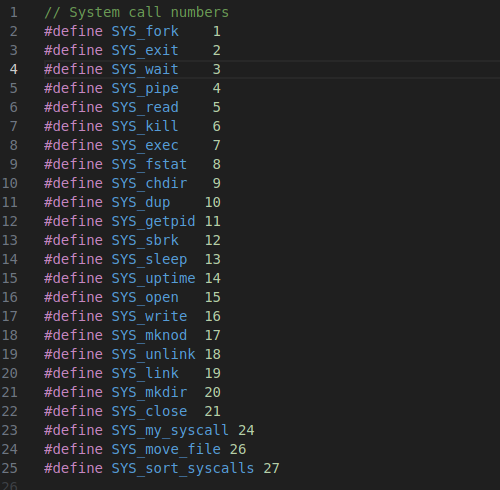


**اضافه کردن فراخوان های سیستمی:**

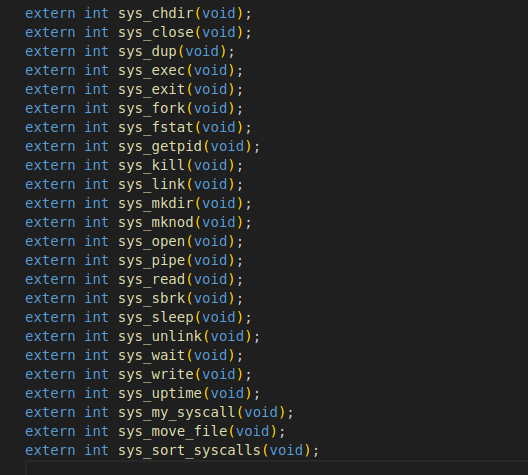
ابتدا تابع مورد نظر را به sysproc اضافه می کنیم:

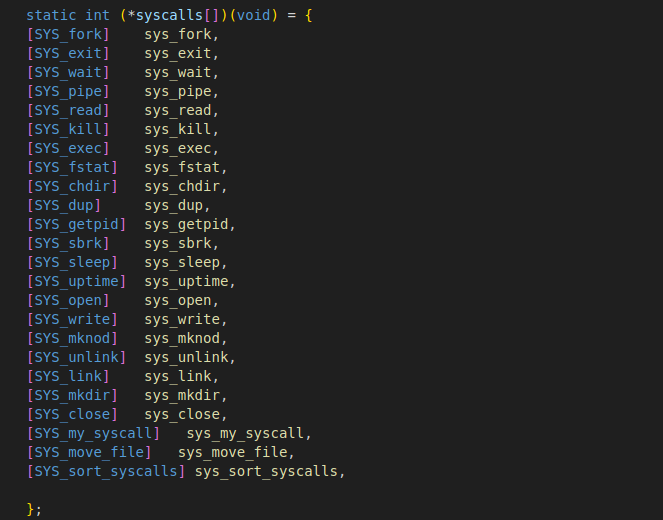


سپس کدی برای syscall مورد نظر در نظر می گیریم و در syscall.h می نویسیم:

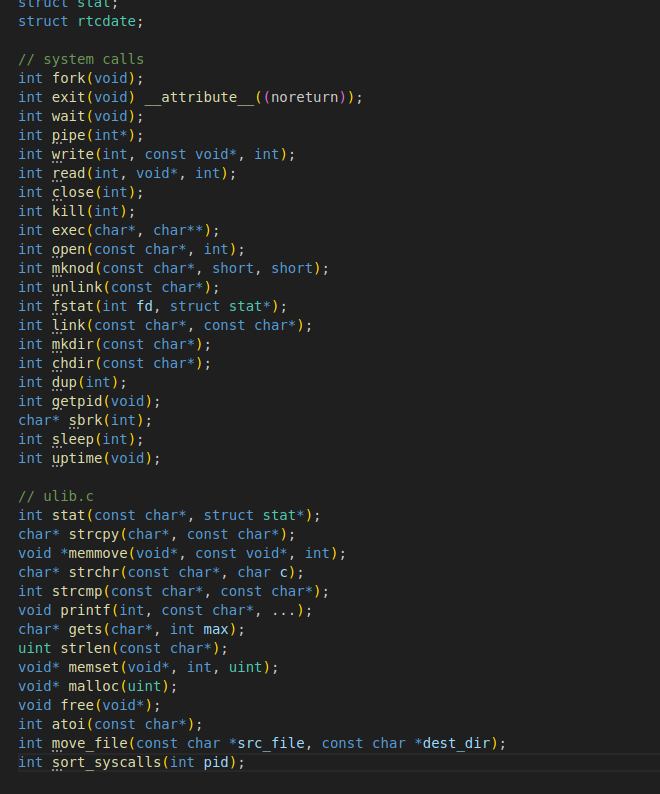


حال باید سیستم کال به وکتور سیستم کال ها در syscall.c باید اضافه شود و پروتوتایپ آن نیز باید به فایل اضافه شود:

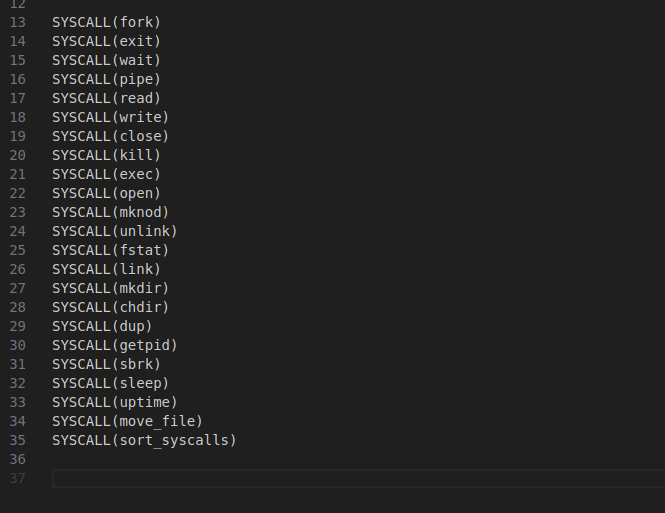
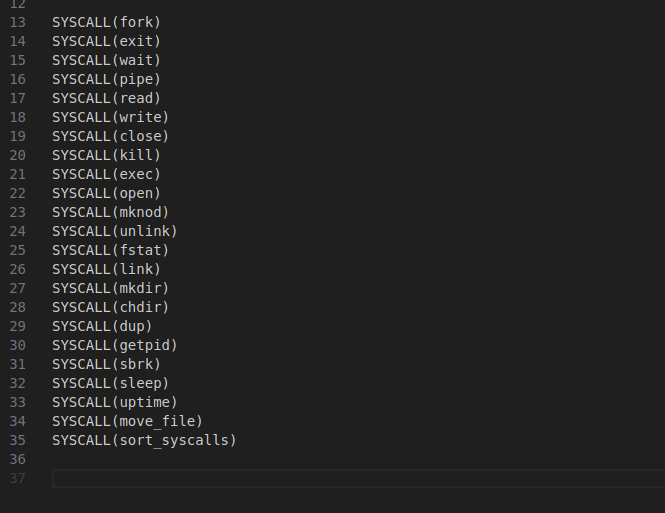




در user.h باید پروتوتایپ سیستم کال برای استفاده اضافه شود:



در آخر در usys.s باید سیستم کال اضافه شود:

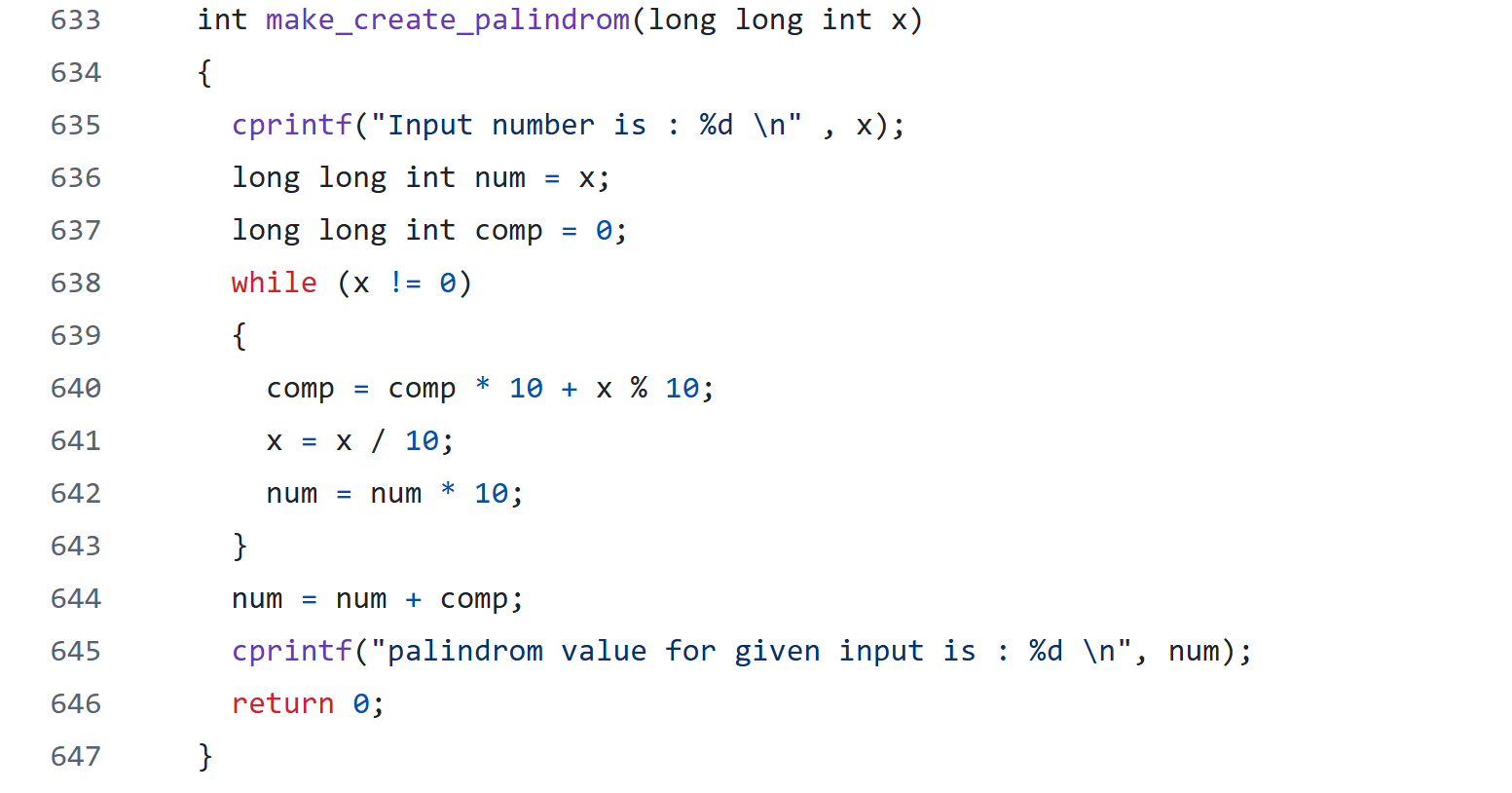


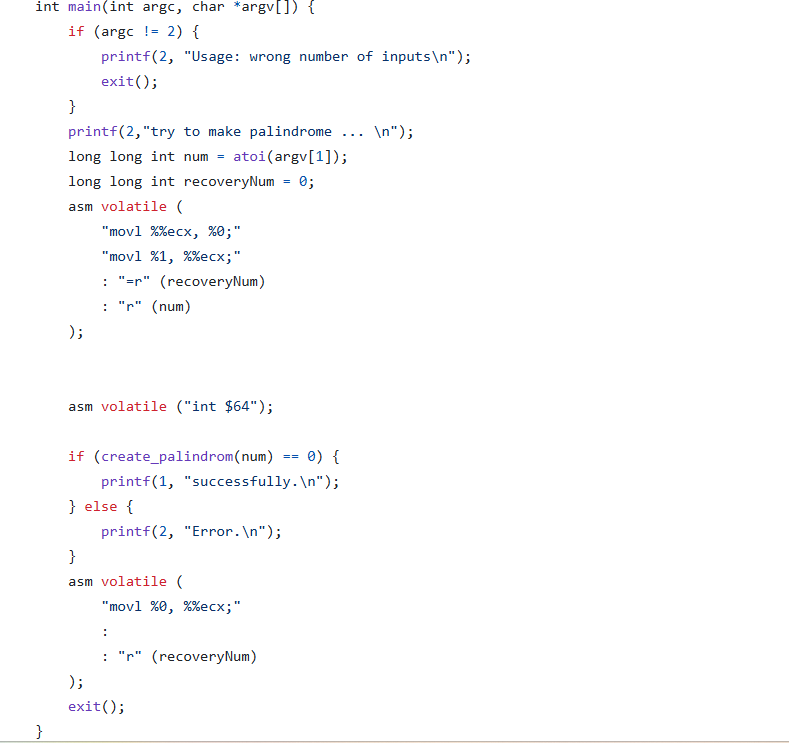
**توضیحات بخش create\_palindrome :**

توضیح بخش اول (تابع `make\_create\_palindrom`)

این تابع یک عدد ورودی `x` را دریافت می‌کند و یک نسخه‌ی قرینه از آن عدد ایجاد می‌کند. مراحل انجام این کار به شکل زیر است:

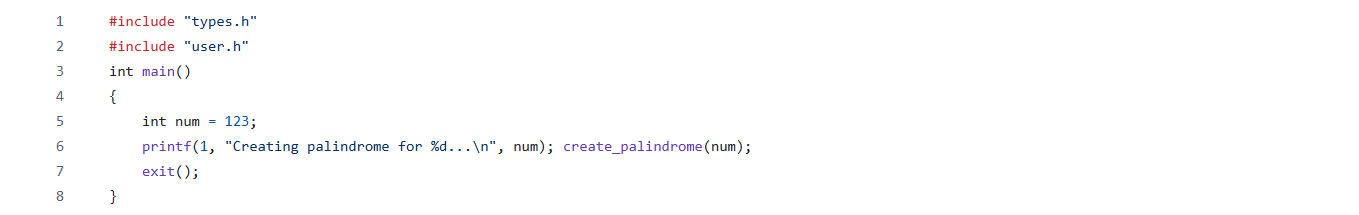
با استفاده از یک حلقه‌ی `while`، هر رقم از عدد `x` را گرفته و به `comp` اضافه می‌کند تا عدد `x` به صورت معکوس ساخته شود. در هر مرحله، رقم آخر `x` با استفاده از `x % 10` محاسبه و به `comp` اضافه می‌شود. سپس `x` بر 10 تقسیم می‌شود تا رقم بعدی را به دست آوریم.همچنین در هر مرحله `num` در 10 ضرب می‌شود تا برای ساختن عدد قرینه آماده شود. پس از پایان حلقه، `comp` که معکوس عدد اولیه است، به `num` اضافه می‌شود.

****

****

توضیح بخش دوم (تابع `main`)

بخش دوم کدی است که از تابع `make\_create\_palindrom` استفاده می‌کند. یک عدد به نام `num` تعریف شده و مقدار آن برابر با `123` قرار داده شده است. تابع `make\_create\_palindrom` با مقدار `num` فراخوانی می‌شود تا قرینه‌ی عدد تولید شود و در نهایت با استفاده از `exit()` برنامه را خاتمه می‌دهد.

****

**توضیحات بخش move\_file :**

این کد یک سیستم‌کال به نام `sys\_move\_file` را پیاده‌سازی می‌کند که برای جابجا کردن یک فایل از یک مسیر منبع (مبدا) به یک مسیر مقصد (هدف) استفاده می‌شود. در ادامه به توضیح کد می‌پردازیم:

1. تعریف متغیرها:

- `path\_src` و `path\_des`: مسیر مبدا و مسیر مقصد را ذخیره می‌کنند.

- `ip\_src`: اشاره‌گر به inode فایل مبدا.

- `dp\_src` و `dp\_des`: اشاره‌گر به inode دایرکتوری‌های مبدا و مقصد.

- `name`: آرایه ای که نام فایل در آن ذخیره می‌شود.

- `off`: متغیری برای ذخیره موقعیت (offset) فایل در دایرکتوری مبدا.

- `de`: ساختاری از نوع `dirent` که برای ذخیره ورودی‌های دایرکتوری استفاده می‌شود.

2. دریافت مسیر مبدا و مقصد:

- ابتدا مسیر منبع و مقصد از آرگومان‌های سیستم‌کال دریافت می‌شوند. اگر موفقیت‌آمیز نباشد، تابع -1 باز می‌گرداند که نشان‌دهنده خطا است.

3. یافتن فایل مبدا:

- با استفاده از تابع `namei`، inode فایل مبدا را جستجو می‌کند. اگر فایل یافت نشد، یا اگر فایل منبع از نوع `T\_FILE` (یعنی یک فایل عادی) نباشد، پیام خطا چاپ شده و تابع به اتمام میرسد.

4. یافتن دایرکتوری مبدا:

- با استفاده از `nameiparent`، دایرکتوری مبدا و نام فایل در آن مسیر شناسایی می‌شوند. اگر این دایرکتوری پیدا نشود، تابع با مقدار خطا باز به اتمام میرسد . `dirlookup` برای پیدا کردن inode فایل در دایرکتوری منبع استفاده می‌شود. اگر فایل پیدا نشود، تابع خاتمه می‌یابد.

5. یافتن دایرکتوری مقصد:

- با استفاده از `namei` دایرکتوری مقصد پیدا می‌شود. اگر پیدا نشود، تابع خاتمه میابد.

در غیر این صورت `dp\_des`، یا همان دایرکتوری مقصد، قفل می‌شود. سپس بررسی می‌کند که آیا دایرکتوری مقصد و فایل منبع روی یک دیسک قرار دارند یا نه. اگر اینگونه نباشد، قفل آزاد شده و تابع خاتمه می‌یابد. سپس فایل با نام `name` به دایرکتوری مقصد اضافه می‌شود (`dirlink`).

6. حذف فایل از دایرکتوری مبدا:

- دایرکتوری مبدا قفل می‌شود و ورودی فایل با استفاده از تابع `writei` حذف می‌شود (با پر کردن ورودی با صفر). در انتها، با `end\_op` تابع پایان می‌یابد و در صورت موفقیت مقدار `0` بازگردانده می‌شود.

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

**توضیحات بخش list\_all\_processes :**

