

در لینوکس فضای آدرس های مجازی یک فرایند به بخش های به نام vma تقسیم می شود.هر vma بازه ای از آدرس های مجازی و attribute هایی را توصیف می کند که می توانیم مثلا با استفاده از آن ها بفهمیم آیا ناحیه برای فایل خاصی است یا مربوط به دیتا های پویاست

همانطور که می دانید لینوکس از جدول صفحات برای نگاشت آدرس های مجازی به آدرس های فیزیکی استفاده می کند.vma شامل ورودی هایی از جدول صفحات می باشد که تعیین می کند هر آدرس مجازی به کدام بخش ازحافظه ی فیزیکی اشاره دارد.

در xv6 فضای آدرس هر فرایند به چند بخش دیتا استک وکد تقسیم می شود .مانند لینوکس ما اینجا نیز جدول صفحات داریم ولی سیاست های ساده تری برای مدیریت حافظه استفاده می گردد.



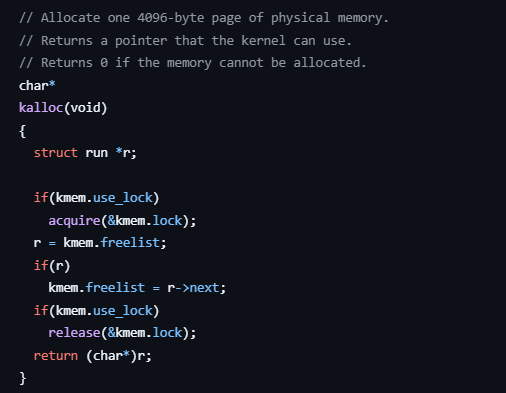
در ساختار سلسله مراتبی ما lazy allocation داریم.به این معنی که همه ی جداول لزوما در حافظه ی فیزیکی ایجاد یا نگهداری نمی شوند و فقط بخش هایی که مورد دسترس است در حافظه نگهداری می شود.

همچنین در ساختار خطی ما به فضای بزرگی برای ذخیره سازی page table نیازمندیم درحالی که وقتی ساختار سلسله مراتبی است هر جدول به چند سطح کوچکتر تقسیم می شود.



در سطح جدول صفحات 20 بیت داریم که مربوط به آدرس بیس حافظه ی فیزیکی است.12 بیت دیگر بیت های کنترلی هستند.مثلا بیتی به اسم present داریم که نشان می دهد آیا این صفحه در حافظه ی فیزیکی وجود دارد یا نه یا مثلا بیت read /write داریم که به ما می گوید اجازه ی نوشتن خواندن داریم یا نه.



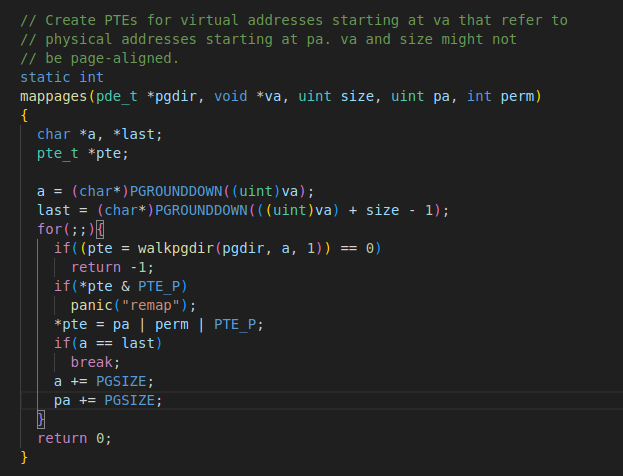


این تابع حافظه ی فیزیکی را تخصیص می دهد. این تابع از free list یک پیج خالی با اندازه 4096 بایت پیدا می کند و پوینتری به آن را بر می گرداند.

در این تابع ابتدا چک می کنیم که آیا از لاک استفاده شده و اگر شده آن راaqcuire می کنیم.سپس در free list دنبال صفحه ی خالی می گردد و در صورت وجود پوینتر را به صفحه ی خالی بعدی می دهد.اگر از لاک استفاده کردیم آن را ریلیز می کنیم و سپس پوینتر را برمی گردانیم.



این تابع حافظه ی مجازی را به حافظه ی فیزیکی نگاشت می کند.بنابراین این تابع دو کار می کند.هم کار نگاشت را انجام می دهد و هم مدخل های جدول صفحات را درست می کند.



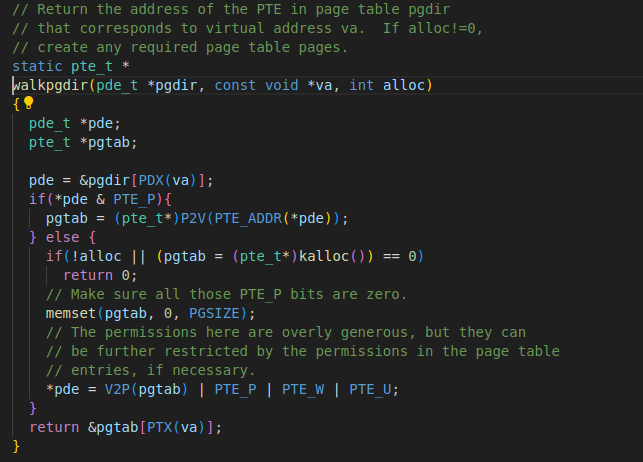
این تابع پیج دایرکتوری که بالاترین لول در سلسله مراتب جدول صفحات است را می گیرد آدرس مجازی سایز مموری رجیون اختصاص یافته و اجازه های مربوط به این بخش از مموری را می گیرد.

این تابع ابتدا از اول virtual address تا virtual address+size می رود .اگر صفحات هنوز تخصیص داده نشده اند صفحات فیزیکی را تخصیص داده و جدول صفحات را آپدیت می کند.همچنین پرمیشن های مربوط به هر مدخل را نیز تنظیم می کند.

این تابع هنگامی که پردازه ی جدید ساخته می شود یا کرنل می خواهد حافظه ی جدید اختصاص دهد استفاده می شود



این تابع در جدول صفحات پیمایش می کند و مدخل مربوط به یک آدرس مجازی مشخص را می یابد.



این تابع پیج دایرکتوری مربوط آدرس مجازی و یک فلگ که فقط در صورت یک بودن پیج تیبل جدید می سازد را می

گیرد

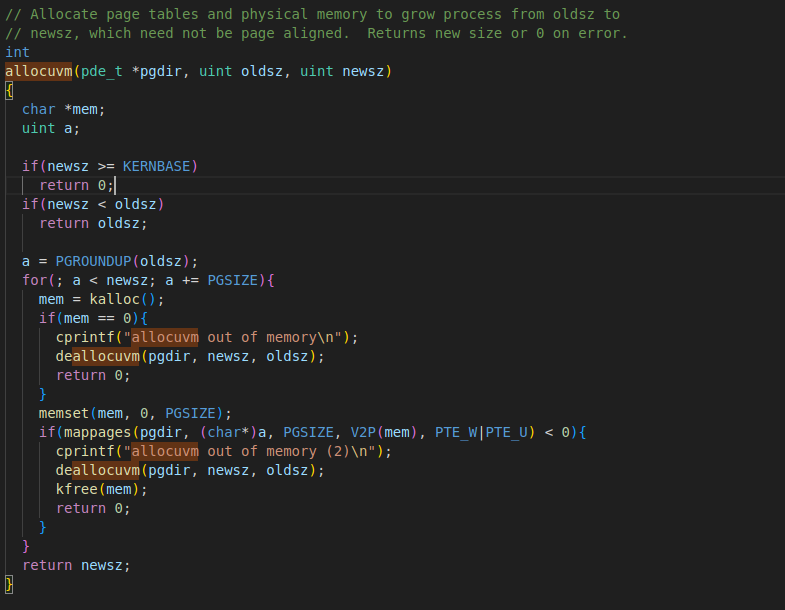
در این تابع ابتدا pde ( page directory entry) را پیدا می کند.اگر پیج تیبل مربوط به این دایرکتوری وجود داشته باشد فلگ pte-p حاضر است.آن را در pgtabمی ریزد.در غیر این صورت اگر فلگ ما 0 بود یا اینکه نتوانست پیج تیبل را بسازد 0 ریترن می کند

اندازه pgtab را به اندازه سایز پیج می گذارد.آدرس مجازی را به آدرس فیزیکی تبدیل می کند.پرمیشن های مربوط به این pdeرا تنظیم کرده و آن را برمی گرداند.

این تابع انگار کار cpu که تبدیل آدرس مجازی به فیزیکی است را انجام می دهد.



تابع mppages به طور کامل توضیح داده شده است.در رابطه با تابع allocuvm این توضیحات را می دهیم.



این تابع مموری را در یک پردازه ی سطح کاربر تخصیص می دهد و آدرس مجازی را گسترش می دهد.

این تابع پوینتری به پیج دایرکتوری سایز قبلی و سایز جدید را می گیرد.

اگر سایز جدید بیشتر از آدرس شروع کرنل اسپیس بود 0 باز می گرداند که پردازه بخش کرنل را وایولیت نکند.

اگر سایز جدید کوچکتر از سایز قبلی بود چون این تابع کوچک کردن حافظه را ساپورت نمی کند سایز قبلی را باز می گرداند.

ابتدا نزدیک ترین آدرس پیج را که می شود تخصیص حافظه را از آن شروع کرد را پیدا می کنیم.تا جایی که می شود از آنجا شروع می کنیم و مموری را اختصاص می دهیم.اگر مموری ما تمام شد همه ی عملیات را revertکرده و 0 باز می گردانی.

در غیر این صورت هر مموری به اندازه سایز پیجمان می گذاریم و با استفاده از تابع mappage آن را به آدرس های فیزیکی مپ می کنیم.

Pte\_w اجازه ی مربوط به نوشتن و pte\_u اجازه ی مربوط به سطح کاربر است.



با استفاده از namei یک inode مربوط به آدرس فایل می سازیم.وقتی فایل ساخته شد،سیستم elf header که شامل اطلاعات اساسی درباره فایل و لوکیشن آن بر روی دیسک است.

سیستم روی سگمنت های در elf header پیمایش می کند و هرکدام prog\_load شان اکتیو بود برایش ویرچوال مموری اختصاص می دهد(از allocuvm استفاده می کند) و آن ها را به آدرس فیزیکی مپ می کند.

با استفاده از loaduvm محتویات برنامه را از inode می خوانیم و اختصاص می دهیم.

سپس دو صفحه برای userstack اختصاص داده می شود.یکی از آن ها غیر قابل دسترسی است که اگر از آن یکی کامل استفاده کردیم بک آپ پلن داشته باشیم.

آرگومان های برنامه به صورت استرینگ روی استک گذاشته می شوند.

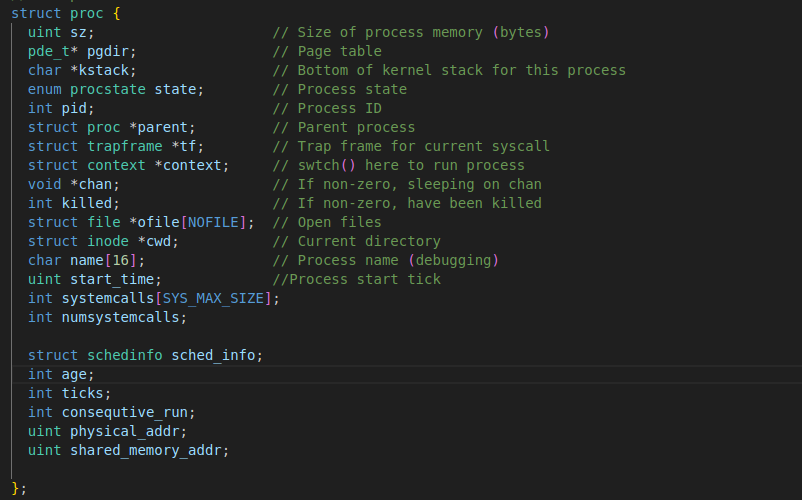
جدول صفحات آپدیت و سایز کامل مموری آپدیت می شود.eip که رجیستر مربوط entry point برنامه است و esp که مربوط به سر استک یوزر است نیز آپدیت می شوند.

با استفاده از switchuvm به پیج تیبل جدید سویچ می کنیم.

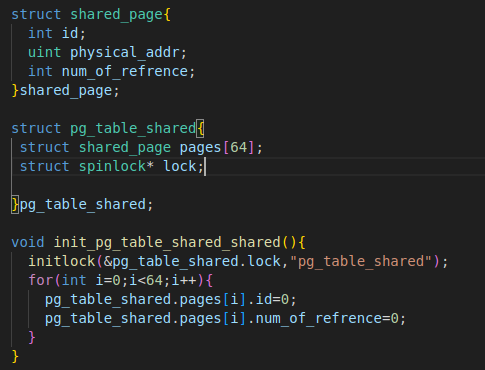
پیاده سازی پروژه:

1-

پیاده سازی استراکت های جدید و اضافه کردن فیلد های جدید به استراکت ها:



به استراکت proc دو فیلد جدید physical address و shared memory address اضافه کردیم.

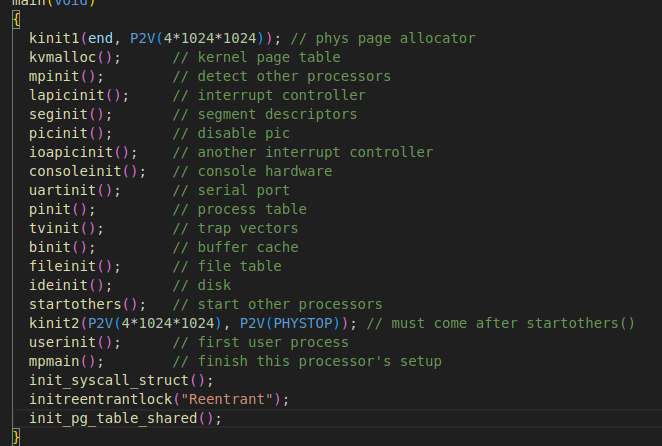


استراکت جدید shared\_page برای صفحات مشترک است که شامل id تعداد پراسس هایی که به آن دسترسی دارند و آدرس فیزیکی صفحه است.

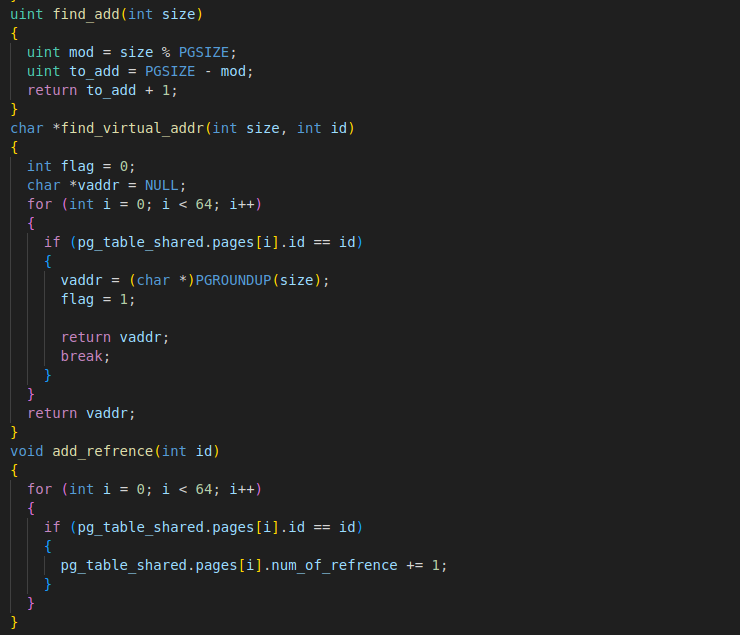
در pgtable\_shared ما 64 پیج داریم و یک لاک که جلوی race condition را بگیریم.

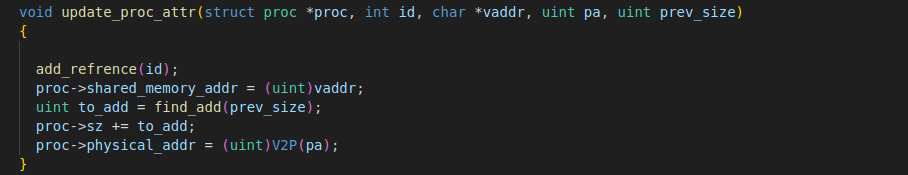
در تابع init\_pg\_table آیدی همه ی پیج هارا 0 می گذاریم که به معنی آن است که هنوز کسی از آن ها استفاده نکرده و تعداد پراسس هایی که به این صفحه دسترسی دارند را نیز 0 می گذاریم

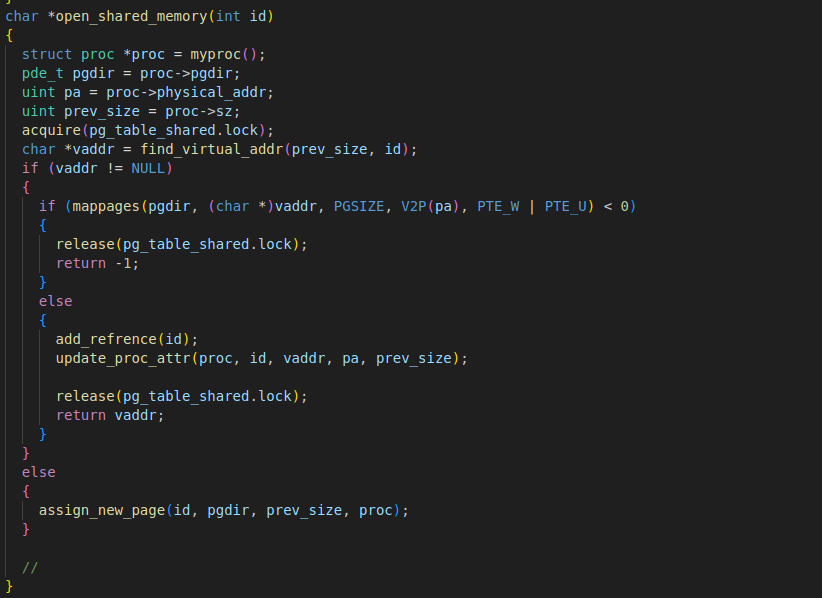
این تابع را در هنگام initialize کردن در main.c صدا می کنیم:



2-بررسی تابع open shared memory :







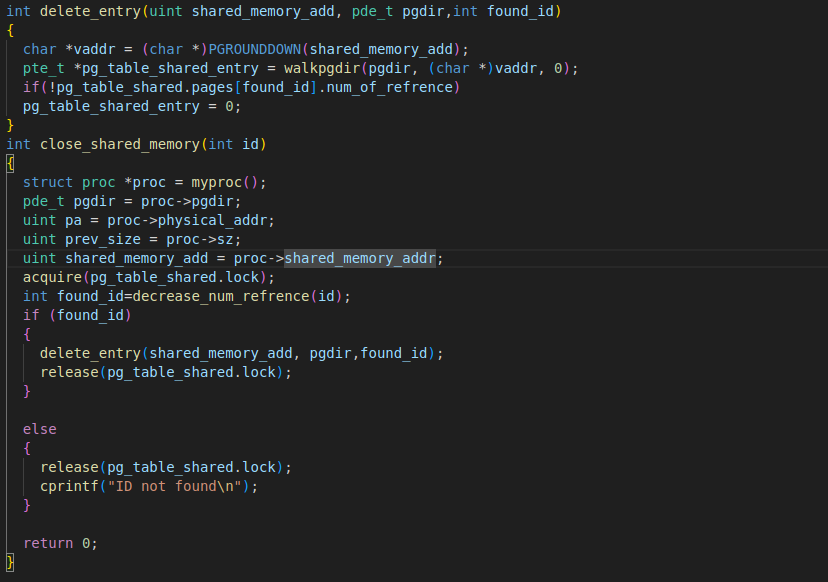
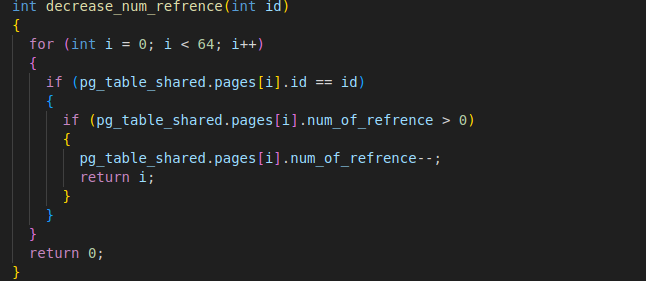
در این تابع ابتدا یک بار جدولمان می گردیم.اگر id موجود بود برای آن با توجه به سایز پراسس یک ویرچوال آدرس با تابع find\_virtual\_address اختصاص می دهیم.اگر یافتیم مقدارغیر نال بازگردانده شده را به آدرس فیزیکال مپ می کنیم.در صورت موفق بودن تعداد دسترسی ها را افزایش می دهیم.سپس آدرس ها و سایز پراسس را در آپدیت پراسس تغییر می دهیم و در آخر قفل را آزاد می کنیم.

حال فرض کنید آی دی مورد نظر در جدول نبود.پس باید یک بخش از جدول اختصاص داده شود.اولین بخش خالی را پیدا می کنیم و همان مراحل بالا را تکرار می کنیم.با این تفاوت که ابتدا با استفاده از تابع kalloc بخشی از حافظه ی فیزیکی را اختصاص می دهیم.در صورت اختصاص درست مانند دفعه ی قبل اولین بخش خالی را پیدا کرده و در صورت موفق بودن نگاشت تعداد دسترسی ها را آپدیت می کنیم.به جای آدرس shared پراسس آدرس مجازی را می گذاریم و سایز پراسس را آپدیت می کنیم.

در صورت موفق نبودن هر یک از مراحل منفی یک را برمی گردانیم

3-

ساختار تابع close shared memory

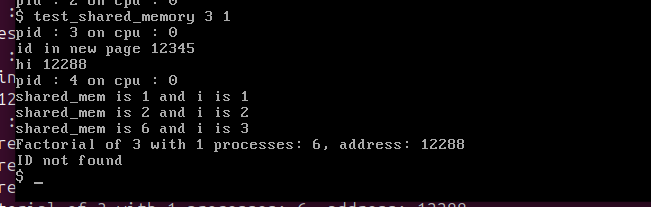


ابتدا به دنبال آی دی می گردیم.در صورت پیدا کردن آی دی آن را باز می گردانیم و در غیر این صورت 0 برمی گردانیم.

در صورت که آی دی را پیدا کردیم در یک تابع مدخل مربوط را با استفاده از تابع walkpgdir پیدا می کنیم.در صورت 0 بودن تعداد رفرنس ها مدخل را 0 می کنیم و حذف می کنیم.

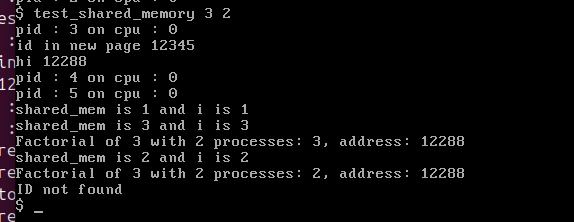
نتیجه ی تست:

تک پردازه ای بدون لاک



که چون یک پردازه داریم نبود لاک تاثیری در جواب مسئله ندارد

چند پردازه ای بدون لاک:



به دلیل race condition جواب مسئله مشکل دارد.