نیما تاجیک
810100104

علی قهاری
810100201

محمد صادقی
810100175

سوال1) مفهوم VMA در لینوکس

حافظه مجازی راهی برای نمایش حافظه است که از حافظه فیزیکی دستگاه abstract شده است. VMA هم از رم و هم از فضای ذخیره سازی شما استفاده می کند، خواه روی هارد دیسک سنتی یا SSD باشد.

در لینوکس این کار در سطح هسته و سخت افزار انجام می شود. CPU دارای یک قطعه سخت افزاری به نام Memory Management Unit (MMU) است که آدرس های حافظه فیزیکی را به مجازی تبدیل می کند. این آدرس ها مستقل از محل قرارگیری فیزیکی آنها در دستگاه هستند. این فضاهای آدرس به عنوان "صفحه" شناخته می شوند و می توانند در RAM یا در هارد دیسک یا SSD شما باشند. سیستم عامل این آدرس ها را به عنوان یک مجموعه بزرگ از حافظه می بیند که به عنوان "فضای آدرس" شناخته می شود.

حافظه مجازی از این واقعیت استفاده می‌کند که تمام حافظه‌هایی که در تئوری استفاده می‌شوند همیشه استفاده نمی‌شوند. برنامه‌های موجود در حافظه به صفحات تقسیم می‌شوند و بخش‌هایی که هسته آن را غیرضروری می‌داند ، تعویض می‌شوند یا به هارد دیسک منتقل می‌شوند. در صورت نیاز، می‌توان آن‌ها را معاوضه کرد یا به رم بازگرداند.

VM در xv6 و Linux

هر پردازه فضای آدرس و جدول صفحه مخصوص به خود را دارد. هسته جداول صفحه را هنگام تعویض پردازه ها تغییر می دهد. تفاوت این دو سیستم عامل در ویژگی های مدیریت حافظه است.

در xv6 ویژگی های مبنایی مثل demand paging پشتیبانی میشود اما سیاست هایی مثل کپی در نوشتن یا جایگزینی را پشتیبانی نمی کند و مدیریت حافظه بسیار ساده ای دارد.

در جهت دیگر سیستم عامل Linux از ویژگی های نسبتا پیچیده تری برای مدیریت حافظه بهره می برد.

مثل shared memory و memory-mapped files و سیاست های جایگزینی مختلفی که در صورت علاقه می تواند مورد مطالعه قرار گیرد. ضمن اینکه معماری ساختار صفحه بندی در Linux به سادگی xv6 که دو سطحی است نبوده و نسبتا پیچیده تر است.

نیما تاجیک
810100104

علی قهاری
810100201

محمد صادقی
810100175

ساختار دو سطحی مدیریت حافظه در XV6

در سیستم عامل xv6، استفاده از سلسله مراتب مدیریت حافظه دو سطحی برای کاهش سربار حافظه و بهبود کارایی در مدیریت فضای حافظه مجازی طراحی شده است. بررسی میکنیم که چگونه این سلسله مراتب دو سطحی به این مزایا دست می یابد:

کاهش سربار حافظه: ساختار جدول صفحه دو سطحی به سیستم عامل اجازه می دهد تا فضای آدرس مجازی را به قطعات کوچکتر و قابل مدیریت تر تقسیم کند. xv6 به جای داشتن یک جدول صفحه بزرگ برای کل فضای آدرس، جداول صفحه را در یک سلسله مراتب سازماندهی می کند، که اندازه هر جدول را کاهش می دهد. با یک سلسله مراتب دو سطحی، جدول صفحه سطح بالا، جداول صفحه سطح دوم را indexing می کند. جداول سطح دوم به نوبه خود مناطق کوچکتری از فضای آدرس مجازی را ترسیم می کنند. این ساختار سلسله مراتبی از نیاز به جدول صفحات مجزا و بزرگ مجزا جلوگیری می کند و نیاز به حافظه را کاهش می دهد.

استفاده از فضای آدرس پراکنده: در بسیاری از برنامه ها، همه مناطق فضای آدرس مجازی به طور همزمان استفاده نمی شوند یا به حافظه فیزیکی نگاشت نمی شوند. یک سلسله مراتب دو سطحی به سیستم عامل اجازه می دهد تا حافظه را به طور موثرتری اختصاص دهد، به خصوص زمانی که با فضاهای آدرس پراکنده سروکار دارد.سلسله مراتب سیستم عامل را قادر می سازد تا حافظه را برای یک منطقه خاص فقط در صورت نیاز تخصیص دهد، نه اینکه مجبور باشد یک قطعه بزرگ به هم پیوسته را برای کل فضای آدرس تخصیص دهد و مدیریت کند.

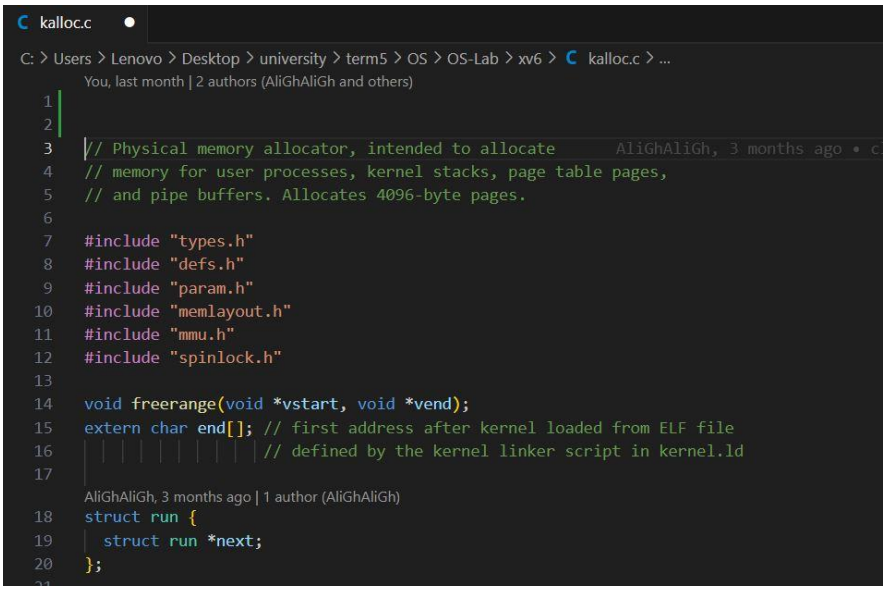
صفحه بندی تقاضا:ساختار جدول صفحه دو سطحی اجرای صفحه‌بندی تقاضا را تسهیل می‌کند، تکنیکی که در آن صفحات تنها زمانی در حافظه فیزیکی بارگذاری می‌شوند که به آن‌ها دسترسی داشته باشید نه اینکه کل فرآیند را به یکباره در حافظه بارگیری کنید.هنگامی که به یک صفحه دسترسی پیدا می‌شود، ورودی‌های جدول صفحه مربوطه به صورت پویا ایجاد می‌شوند و به سیستم عامل اجازه می‌دهند تنها بخش‌های لازم از فضای آدرس را در صورت نیاز در حافظه فیزیکی بارگذاری کند.

بهبود مدیریت جدول صفحه : مدیریت یک سلسله مراتب دو سطحی انعطاف پذیرتر و کارآمدتر است. ورودی های جدول صفحه سطح بالا به جداول صفحه سطح دوم و ورودی های سطح دوم به فریم های واقعی صفحه در حافظه فیزیکی اشاره می کنند. این سازماندهی امکان به روز رسانی و تغییرات کارآمدتر در جداول صفحه را می دهد.

- ورودی جدول صفحه سطح اول (PDX) : 10 بیت پر ارزش آدرس مجازی برای یافتن ورودی مربوطه در جدول صفحه سطح اول استفاده می شود. در Xv6، این ورودی 32 بیت است و آدرس فیزیکی که به آن اشاره می کند، آدرس پایه جدول صفحه سطح دوم است.
- ورودی جدول صفحه سطح دوم (PTX) : 10بیت بعدی آدرس مجازی برای یافتن ورودی مربوطه در جدول صفحه سطح دوم استفاده می شود. این ورودی 32 هم بیت است و آدرس فیزیکی که به آن اشاره می کند، آدرس واقعی فریم در حافظه است.
- 12 بیت باقی مانده در آدرس مجازی به عنوان افست در فریم استفاده می شود.

کد مربوط به ایجاد فضاهای آدرس درxv6

با توجه به تابع موجود در فایل kalloc.c آدرس فیزیکی است(کامنت خط سوم)



تابع mappage وظیفه نگاشت بازه آدرس مجازی به فیزیکی را در page table به عهده دارد که 5 ورودی مطابق شکل زیر دارد:

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int      AliGhAliGh, 3 months ago • cloned file added
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

- pgdir : اشاره گر به پیج دایرکتوری پردازه است
- va : آدرس مجازی جایی است که نگاشت باید از آن شروع شود
- pa : آدرس فیزیکی جایی در حافظه اصلی است که نگاشت به آن میرود.
- size : اندازه قطعه نگاشت شده از حافظه است
- perm : این نشان دهنده اجازه دسترسی به ناحیه نگاشت شده است (خواندن و نوشتن و ...)

این تابع آدرس های شروع و پایان تراز صفحه را برای محدوده آدرس مجازی محاسبه می کند، در این محدوده تکرار می شود و ورودی های مناسب را در جداول صفحه تنظیم می کند. این تضمین می کند که جداول صفحه لازم تخصیص داده شده و در صورت عدم وجود آنها مقداردهی اولیه می شوند.

نیما تاجیک
810100104

علی قهاری
810100201

محمد صادقی
810100175

تابع walkpgdir

ابتدا نگاهی به بدنه این تابع در فایل vm.c می اندازیم:

```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];        AliGhAliGh, 3 months ago • cloned file
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

این تابع برای پیمایش جداول صفحه برای یافتن ورودی جدول صفحه مربوط به یک آدرس مجازی داده شده استفاده می شود.

این تابع نیز 3 ورودی دارد که ورودی اول و دوم آن مشابه تابع قبلی است و متغیر سوم آن پرچمی است که نشان می‌دهد آیا تابع باید جداول صفحه جدیدی را در صورتی که قبلاً وجود ندارند، اختصاص دهد یا خیر.

این تابع کاری را که در سخت افزار با دیسک ، چرخش آن و حرکت head انجام میدهند تا جایی از حافظه اصلی را بر اساس آدرس مجازی پیدا کنند عهده دار می شود و در صورت عدم وجود چنین حافظه ای آن را تخصیص می دهد.

تابع allocuvm

در xv6، *allocuvm* و *mappages* توابع مربوط به مدیریت حافظه مجازی، به ویژه در زمینه ایجاد پردازه و تنظیم جداول صفحه اولیه هستند. بیایید نگاهی دقیق تر به هر یک از این توابع بیندازیم:

```
// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
  return newsz;
}
```

هدف *allocuvm* تخصیص طیف وسیعی از صفحات حافظه مجازی برای یک پردازه است و معمولا وقتی استفاده می شود که پردازه نیاز به حافظه بیشتری داشته باشد. اغلب در حین ایجاد پردازه برای تنظیم طرح اولیه حافظه مجازی فراخوانی می شود. تابع یک اشاره گر را به حافظه اختصاص داده شده جدید برمی گرداند.

در داخل *allocuvm* از *kalloc* برای تخصیص صفحات حافظه فیزیکی و از *mappage* برای نگاشت این صفحات در فضای آدرس پردازه استفاده می کند.

نیما تاجیک
810100104
علی قهاری
810100201
محمد صادقی
810100175

بارگذاری برنامه توسط exec

در xv6، فراخوانی سیستم *exec* مسئول بارگذاری یک برنامه جدید در حافظه و جایگزینی تصویر فرآیند فعلی با برنامه جدید است. فرآیند بارگذاری شامل چندین مرحله است که درینجا یک نمای کلی از نحوه عملکرد فراخوانی سیستم *exec* از نظر بارگذاری یک برنامه در حافظه ارائه خواهیم کرد:

تجزیه فایل اجرایی: فراخوانی سیستم *exec* یک نام فایل را به عنوان آرگومان می گیرد که نام فایل اجرایی است که قرار است بارگذاری شود.هسته فایل را در سیستم فایل جستجو می کند و هدر فایل اجرایی را می خواند تا ساختار و الزامات آن را بفهمد.

پاکسازی حافظه: حافظه فرآیند موجود پاک می شود یا اختصاص داده می شود تا جایی برای برنامه جدید ایجاد شود. این شامل انتشار جداول صفحه قدیمی و هر منبع مرتبط با برنامه قبلی است.

بارگذاری برنامه:هسته یک فضای آدرس جدید را برای فرآیند اختصاص می دهد که معمولاً از تابع *allocuvm* استفاده می کند. این تابع وظیفه تخصیص جداول صفحات جدید و نگاشت صفحات لازم در فضای حافظه مجازی فرآیند را بر عهده دارد.

کپی کردن داده ها و کد: هسته محتویات فایل اجرایی را که شامل هر دو بخش کد و داده است می خواند و آنها را در فضای جدید حافظه اختصاص داده شده کپی می کند. این ممکن است شامل استفاده از تابع *Readi* برای خواندن از فایل و نقشه‌ها برای نگاشت صفحات در حافظه مجازی فرآیند باشد.

راه اندازی پشته:هسته پشته را برای برنامه جدید مقداردهی اولیه می کند. این شامل تخصیص فضا برای پشته و تنظیم نشانگر پشته بر اساس آن است.

تنظیم رجیسترها و شمارنده برنامه:هسته مقادیر ثبت فرآیند، از جمله شمارنده برنامه (PC) را تنظیم می کند تا به نقطه ورودی برنامه تازه بارگذاری شده اشاره کند.

پرش به برنامه جدید: هسته یک context switch را انجام می دهد و کنترل را به برنامه تازه بارگذاری شده منتقل می کند. این شامل استفاده از دستورالعمل‌هایی برای تنظیم پشته حالت کاربر و انتقال کنترل به حالت کاربر است.