آزمایشگاه سیستم عامل

گزارش پروژه پنجم

- **آرین باستانی** - 810100088
- **مهدیار هرندی** - 810199596
- **محمدرضا نعمتی** - 810100226

- https://github.com/mmd-nemati/OS-Lab5
- Last commit hash: 2ada9b11643924072cadaa8f54784b7e639eec72

1- VMA ساختاری است که توسط هسته لینوکس برای مدیریت مناطق حافظه فضای آدرس یک پراسس استفاده می شود. هر VMA نشان دهنده یک ناحیه پیوسته در فضای آدرس یک پراسس است و اطلاعاتی مانند آدرس های شروع و پایان منطقه، مجوزهای دسترسی (مانند خواندن، نوشتن، execute) و خصوصی بودن یا اشتراک گذاری منطقه در بین پراسس ها را در خود نگه می دارد.

VMA ها نقش مهمی در مدیریت فایل های memory-mapped ، صفحه بندی تقاضا و مکانیزم های copy-on-write دارند. هنگامی که یک پراسس به بخشی از فضای آدرس خود دسترسی پیدا می کند، هسته VMA مربوطه را برای تعیین نحوه مدیریت دسترسی، از جمله بارگیری داده ها از دیسک، اشتراک گذاری حافظه با سایر فرآیندها، یا تخصیص صفحات حافظه جدید بررسی می کند.

xv6 بسیار ساده تر از لینوکس است. VMA های لینوکس بخشی از یک سیستم مدیریت حافظه پیچیده هستند که از ویژگی های پیشرفته مانند memory-mapped files و استراتژی های تخصیص حافظه پیچیده پشتیبانی می کنند. از سوی دیگر، xv6 یک رویکرد مدیریت حافظه ساده‌تر دارد، که برای مدیریت سناریوهای متنوعی که لینوکس انجام می‌دهد ساخته نشده است.

در لینوکس، VMAها برای مدیریت جنبه های مختلف حافظه یک پراسس، از جمله مجوزهای دسترسی به حافظه فیزیکی استفاده می شوند. xv6 دارای یک سیستم مدیریت حافظه ساده‌تر است که جزئیات یا انعطاف‌پذیری کمتری را در مدیریت مناطق مختلف حافظه ارائه می‌کند. لینوکس از ویژگی های

مدیریت حافظه پیشرفته مانند صفحه بندی تقاضا، که در آن تنها بخش هایی از برنامه در صورت نیاز در حافظه بارگذاری می شود، و copy-on-write ، که در آن فورک های پردازش می توانند حافظه فیزیکی یکسانی را به اشتراک بگذارند، پشتیبانی می کند. xv6، در فرم استاندارد خود از این ویژگی های پیشرفته پشتیبانی نمیکند ، یا اگر پشتیبانی می کند، آنها را به شکل بسیار ابتدایی تری پیاده سازی می کند.

2- در سیستم عامل ها، استفاده از ساختار سلسله مراتبی برای مدیریت حافظه می تواند مصرف حافظه را به چند دلیل کاهش دهد:

در یک ساختار سلسله مراتبی، اجزا یا کتابخانه های مشترک را می توان بین برنامه ها یا ماژول های مختلف به اشتراک گذاشت. این بدان معنی است که به جای اینکه هر برنامه کپی مخصوص به خود از یک جزء داشته باشد، یک نمونه واحد می تواند توسط چندین برنامه استفاده شود.

سیستم های سلسله مراتبی اغلب ماژولار هستند، به این معنی که هر ماژول یا لایه فقط باید اطلاعات خود را پیگیری کند و می تواند برای عملیات اساسی تر به لایه های پایین تر تکیه کند. این ماژولار بودن امکان استفاده کارآمدتر از حافظه را فراهم می کند، زیرا هر لایه فقط آنچه را که برای عملکرد خاص خود لازم است بارگیری یا حفظ می کند.

ساختارهای سلسله مراتبی می توانند در تخصیص حافظه کارآمدتر موثر باشند. به عنوان مثال، حافظه را می توان بر اساس نیاز به استفاده تخصیص داد. لایه‌های بالایی می‌توانند در صورت نیاز از لایه‌های پایین‌تر حافظه درخواست کنند، و زمانی که دیگر نیازی به آن نیست، آن را آزاد کنند.

مدیریت حافظه سلسله مراتبی اغلب به خوبی با مکانیسم های کش و پیجینگ هماهنگ می‌شوند. این ساختارها سازماندهی حافظه را به شکلی تسهیل می‌کنند که پیاده‌سازی سیاست‌های کشینگ کارآمدتر شود، به طوری که داده‌های پرکاربرد دردسترس‌تر نگهداری می‌شوند، و در نتیجه نیاز به حافظه بزرگتر برای حفظ عملکرد کاهش می‌یابد.

با جداسازی ساختاری بخش‌های مختلف سیستم، طرح‌های سلسله مراتبی همچنین امکان کنترل دقیق‌تری بر دسترسی به حافظه را فراهم می‌کنند. این می تواند از بارگذاری غیر ضروری داده های امن جلوگیری کند و مصرف حافظه را کاهش دهد.

3- در یک سیستم صفحه بندی 32 بیتی، هر ورودی در page table، سی و دو بیت است و معمولاً شامل موارد زیر است:

Physical frame address: اکثر بیت ها نشان دهنده آدرس حافظه فیزیکی هستند که داده های واقعی در آن ذخیره می شوند.

Present Bit: نشان می دهد که صفحه در حافظه فیزیکی است یا خیر.

Read/Write Bit: نشان می دهد که آیا صفحه قابل خواندن و نوشتن است یا خیر.

User/Supervisor Bit: سطح دسترسی (کاربر یا هسته) را تعیین می کند.

Accessed Bit: نشان می‌دهد که آیا به صفحه دسترسی پیدا شده است یا خیر.

Dirty Bit: نشان می‌دهد که آیا صفحه از زمان آخرین پاکسازی توسط سیستم‌عامل تغییر کرده است یا خیر.

تفاوت اصلی بین ورودی‌ها در سطوح مختلف پیجینگ هدف آن‌ها است:

Higher level tables: به جداول صفحه دیگر اشاره میکند.

Lower level tables: مستقیماً به فریم های حافظه فیزیکی اشاره میکند.

هر دو سطح دارای بیت های کنترل و وضعیت مشابه برای کنترل دسترسی و مدیریت حافظه هستند.

4- از تابع kalloc برای تخصیص حافظه فیزیکی استفاده می شود. این تابع یک پوینتر را به یک صفحه آزاد از حافظه فیزیکی برمی گرداند. این تابع برای مدیریت حافظه سیستم عامل بسیار مهم است، زیرا حافظه را برای پراسس ها و برای خود هسته اختصاص می دهد. سپس حافظه فیزیکی اختصاص داده شده توسط kalloc را می توان با استفاده ازPage table ها در فضای آدرس مجازی یک پراسس یا هسته مپ کرد.

درxv6 از تابع kalloc برای تخصیص حافظه فیزیکی استفاده می شود. این یک پوینتر را به یک صفحه آزاد از حافظه فیزیکی برمی گرداند. این تابع برای مدیریت حافظه سیستم عامل بسیار مهم است، زیرا حافظه را برای پراسس ها و برای خود هسته اختصاص می دهد. سپس حافظه فیزیکی اختصاص

داده شده توسط kalloc را می توان با استفاده از page table ها در فضای آدرس مجازی یک فرآیند یا هسته مپ کرد.

5- تابع mappages در xv6 برای ایجاد نگاشت در page table استفاده می شود. این تابع محدوده ای از آدرس های مجازی را به آدرس های فیزیکی مپ می کند. همچنین این تابع چندین آرگومان از جمله دایرکتوری صفحه (page table)، آدرس مجازی، اندازه حافظه برای map، آدرس فیزیکی و فلگ های مجوز را می گیرد. هدف اصلی mappages تنظیم ورودی های page table است تا یک آدرس مجازی به درستی به آدرس فیزیکی مربوطه با مجوزهای مناسب (مانند read, write, execute) نگاشت شود. این تابع درتنظیم page table های کرنل هسته و هم در تنظیم page table ها برای پراسس های کاربر استفاده می شود. دلیل حیاتی بودن mappages این است که سیستم با استفاده از این تابع حافظه مجازی را با پیوند دادن آدرس‌های مجازی به مکان‌های حافظه فیزیکی فعال میکند.

7- تابع walkpgdir در xv6 بخش مهمی از سیستم پیجینگ آن است. این تابع page table را پیمایش می کند تا PTE را برای یک آدرس مجازی مشخص پیدا کند. این تابع اساساً فرآیند ترجمه یک آدرس مجازی به یک آدرس فیزیکی را شبیه سازی می کند، وظیفه ای که معمولاً توسط Memory Management Unit در سخت افزار انجام می شود.همچنین این تابع در سلسله مراتب page table، با استفاده از بخش هایی از آدرس مجازی برای فهرست بندی در جدول و یافتن PTE مربوطه که حاوی آدرس فیزیکی است، پیمایش میکند.

8- allocuvm: با گسترش فضای آدرس مجازی یک پراسس، حافظه مجازی را به آن اختصاص می دهد. از kalloc برای تخصیص حافظه فیزیکی استفاده می‌کند و سپس این صفحات فیزیکی را با استفاده از تابع mappages به آدرس‌های مجازی در فضای آدرس پراسس مپ می‌کند.

mappages: آدرس های مجازی را به آدرس های فیزیکی در page table نگاشت می کند. ورودی های page table را آپدیت می کند تا طیفی از آدرس های مجازی را به آدرس های فیزیکی خاص پیوند دهد. این تابع برای

اتصال حافظه مجازی به حافظه فیزیکی ضروری است و در کارهای مختلف مدیریت حافظه از جمله توسط تابع allocuvm استفاده می شود.

9- در xv6، سیستم کال exec مسئول بارگذاری یک برنامه در حافظه و اجرای آن است و تصویر پراسس فعلی را با برنامه جدید جایگزین می کند.

وقتی exec فراخوانی می شود، ابتدا فرمت فایل اجرایی را تأیید می کند، که معمولاً یک ELF در xv6 است. با نگاه کردن به هدرها بررسی می کند که آیا فایل یک فایل اجرایی معتبر است یا خیر.

اگر فایل معتبر باشد، exec حافظه را برای کد، داده ها و بخش های استک برنامه تخصیص می دهد. این کار با فراخوانی allocuvm برای گسترش فضای آدرس پراسس و تخصیص صفحات مورد نیاز انجام می شود. سپس کد و دیتای فایل اجرایی را می خواند و آنها را در حافظه اختصاص داده شده جدید در فضای آدرس پراسس کپی می کند. این شامل خواندن از سیستم فایل و نوشتن در حافظه مجازی اختصاص داده شده برای پراسس است.

یک استک در فضای آدرس پراسس تنظیم می شود. exec استک را با آرگومان های برنامه و هر متغیر محیطی لازم مقداردهی اولیه می کند. فراخوانی exec طرح حافظه پراسس را آپدیت می کند تا برنامه جدید را منعکس کند به صورتی که پوینتر دستور برای اشاره به نقطه ورودی برنامه جدید را آپدیت کرده و سایر رجیسترها را در صورت نیاز به روز میکند. سپس چیدمان حافظه پراسس را آپدیت می کند و پوینتر دستور را روی نقطه ورودی برنامه جدید قرار میدهد. در نهایت، از کرنل مود به یوزر مود رفته و کنترل را به برنامه تازه بارگذاری شده منتقل می‌کند و پراسس شروع به اجرای برنامه جدید از نقطه ورود آن می‌کند.

کل فضای آدرس قدیمی پراسس با فضای آدرس برنامه جدید جایگزین می شود. این بدان معنی است که تمام حافظه مرتبط با برنامه قدیمی، از جمله کد، داده ها و استک آن حذف شده و با حافظه برنامه جدید جایگزین می شود.

پیاده سازی shared memory

ابتدا فیلد shared_addr را به استراکت proc اضافه میکنیم تا آدرس shared memory را ذخیره کنیم.

xv6-public - proc.h

```
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52    uint shared_addr;            // Address of shared memory
53  };
54
```

حال طبق راهنمایی صورت پروژه برای پیاده سازی shared memory دو استراکت زیر را تعریف میکنیم.

xv6-public - vm.c

```
11  // shared memory
12  #define NUM_SHAREDPAGE   64
13  struct sharedmem_page {
14      int id;
15      int num_access;
16      uint physical_addr;
17  } sharedmem_page;
18
19  struct sharedmem_table {
20      struct spinlock lock;
21      struct sharedmem_page pages[NUM_SHAREDPAGE];
22
23  } sharedmem_table;
24  // end shared memory
```

استراکت sharedmem_table تعدادی page به همراه قفل به سبک spinlock دارد که از دسترسی چند پراسس به طور همزمان به shared memory جلوگیری میکند.

استراکت sharedmem_page هم اطلاعات مورد نیاز در مورد shared memory را ذخیره میکند.

سیستم کال open_sharedmem

از این سیستم کال برای ساختن یا استفاده از یک shared memory استفاده میکنیم. به عنوان ورودی یک id میگیرد که نشان دهنده shared memory مورد نظر است. این آیدی در sharedmem_page ذخیره می شود.

اگر برای id داده شده shared memory وجود داشته باشد ابتدا یک مموری مجازی را با سایز مورد نیاز پراسس در اختیار میگیریم. سپس این مموری مجازی را به یک ادرس مموری فیزیکی با استفاده از مقادیر موجود در sharedmem_page و sharedmem_table مپ میکنیم. باید page و id متناظر را از این دو استراکت بدست آوریم. همچنین در زمان افزایش سایز پراسس باید از آدرس مجازی بعد از shared memory استفاده کنیم تا به remap panic نخوریم.

اگر برای id داده شده memory shared وجود نداشته باشد باید ابتدا برای آن به مقدار مورد نیاز مموری با استفاده از تابع kalloc اختصاص دهیم. سپس با استفاده از ماکرو xv6 یعنی V2P آن آدرس مجازی را به یک آدرس فیزیکی مپ میکنیم. بعد از آن تعداد دسترسی ها shared memory را یک عدد زیاد میکنیم. سپس ادامه فرایند مانند حالت قبل است.

xv6-public - vm.c

```
412  char* open_sharedmem(int id) {
413      struct proc* proc = myproc();
414      acquire(&sharedmem_table.lock);
415      int size = PGSIZE;
416
417      for (int i = 0; i < NUM_SHAREDPAGE; i++) {
418          if (sharedmem_table.pages[i].id == id) {
419              sharedmem_table.pages[i].num_access++;
420              char* vaddr = (char*)PGROUNDUP(proc->sz);
421              if (mappages(proc->pgdir, vaddr, PGSIZE, sharedmem_table.pages[i].physical_addr, PTE_W | PTE_U) < 0)
422                  cprintf("error in mappages\n");
423
424              proc->shared_addr = (uint)vaddr;
425              proc->sz += size;
426              release(&sharedmem_table.lock);
427              return vaddr;
428          }
429      }
430
431      int page_index = -1;
432      for (int i = 0; i < NUM_SHAREDPAGE; i++) {
433          if (sharedmem_table.pages[i].id == 0) {
434              sharedmem_table.pages[i].id = id;
435              page_index = i;
436              break;
437          }
438      }
439      if (page_index == -1) {
440          cprintf("all pages have been used\n");
441          release(&sharedmem_table.lock);
442          return 0;
443      }
```

```
443     }
444
445     char* paddr;
446     if ((paddr = kalloc())== 0) {
447         cprintf("memory is full\n");
448         release(&sharedmem_table.lock);
449         return 0;
450     }
451
452     memset(paddr, 0, PGSIZE);
453     char* vaddr = (char*)PGROUNDUP(proc->sz);
454     sharedmem_table.pages[page_index].physical_addr = (uint)V2P(paddr);
455     if (mappages(proc->pgdir, vaddr, PGSIZE, sharedmem_table.pages[page_index].physical_addr, PTE_W | PTE_U) < 0) {
456         cprintf("error in mapping\n");
457     }
458
459     sharedmem_table.pages[page_index].num_access++;
460     proc->shared_addr = (uint)vaddr;
461     proc->sz += size;
462
463     release(&sharedmem_table.lock);
464     return vaddr;
465 }
466
```

سیستم کال close_sharedmem

این سیستم کال در لیست sharedmem_table س shared memory با آیدی داده شده را جستجو میکند.
در صورت پیدا شدن آن num access آن کم می شود. سپس با استفاده از تابع `walkpgdir` آدرس PTE آن shared memory را پیدا میکنیم و مساوی صفر قرار میدهیم.
در صورتی که shared memory پیدا نشد -1 ریترن میکند.

xv6-public - vm.c

```
468 int close_sharedmem(int id) {
469     struct proc* proc = myproc();
470     acquire(&sharedmem_table.lock);
471     for (int i = 0; i < NUM_SHAREDPAGE; i++) {
472         if (sharedmem_table.pages[i].id == id) {
473             sharedmem_table.pages[i].num_access--;
474             pte_t *PTE_addr = walkpgdir(proc->pgdir, (char*)(uint)PGROUNDUP(proc->shared_addr), 0);
475             *PTE_addr = 0;
476
477             if(sharedmem_table.pages[i].num_access == 0)
478                 sharedmem_table.pages[i].id = 0;
479
480             release(&sharedmem_table.lock);
481             return 0;
482         }
483     }
484     release(&sharedmem_table.lock);
485     cprintf("shared memory not found\n");
486     return -1;
487 }
```

برنامه تست

در برنامه تست ابتدا یک shared memory ایجاد میکنیم. سپس مقدار موجود در آن را برابر صفر قرار میدهیم. سپس 5 پراسس child ایجاد میکنیم که هر کدام به آن shared memory دسترسی پیدا میکنند و مقدار آن را تغییر میدهند.

در آخر مشاهده میکنیم مقدار shared memory به طور واقعی در پراسس parent تغییر کرده است.

xv6-public - shared_test.c

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char* argv[]) {
    char* shared = open_sharedmem(1);
    shared[0] = 0;
    printf(1, "proc: %d --- value: %d\n", 0, shared[0]);

    for (int i = 1; i <= 5; i++) {
        if (fork() == 0) {
            char* shmem = open_sharedmem(1);
            shmem[0] += 1;
            printf(1, "proc: %d --- value: %d\n", i, shmem[0]);
            close_sharedmem(1);
            exit();
        }
    }
    for (int i = 0; i < 5; i++)
        wait();

    printf(1, "\nfinal value:\nproc: %d --- value: %d\n", 0, shared[0]);
    close_sharedmem(1);
    exit();
}
```

```
qemu-system-i386 -serial mon:stdio -c
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200
init: starting sh
$ shared_test
proc: 0 --- value: 0
proc: 1 --- value: 1
proc: 2 --- value: 2
proc: 3 --- value: 3
proc: 4 --- value: 4
proc: 5 --- value: 5

final value:
proc: 0 --- value: 5
$
```