آزمایشگاه سیستم عامل **پروژه 5**

سهیل حاجیان منش 810100119

امیرعلی رحیمی 810100146

مهدی نوری 810100231

github repo: https://github.com/MahdiNoori2003/OS-Lab-5

last commit: 7086543a1992d8a8814f8c5cf31d235b149bbbb3

مدیریت حافظه در xv6

1. راجع به مفهوم ناحیه مجازی (VMA[1]) در لینوکس به طور مختصر توضیح داده و آن را با xv6 مقایسه کنید.

در لینوکس، هسته از مناطق حافظه مجازی (VMAs) برای پیگیری نگاشت‌های حافظه یک فرآیند استفاده می‌کند. یک فرآیند یک VMA برای کد خود، یک VMA برای هر نوع داده و یک VMA برای هر نگاشت حافظه مجزا دارد. VMA ها ساختارهای مستقل از پردازنده با مجوزها[2] و پرچم های کنترل دسترسی[3] هستند. هر VMA یک آدرس شروع و یک طول دارد و اندازه آنها همیشه مضربی از اندازه

[1] Virtual memory address
[2] Permissions
[3] Access control flags

صفحه است. مناطق حافظه توصیف شده توسط VMA همیشه به صورت مجازی به هم پیوسته هستند و نه از نظر فیزیکی.

حال این را با xv6 مقایسه میکنیم:

در xv6 مفهومی از VMA وجود ندارد. در عوض، xv6 از یک جدول صفحه دو سطحی ساده برای مدیریت تخصیص حافظه خود استفاده می کند. تمام حافظه فیزیکی قابل استفاده توسط kpgdir در فضای آدرس مجازی نگاشت می شود، بنابراین تمام حافظه می تواند توسط آدرس های مجازی آدرس دهی شود و توسط واحد مدیریت حافظه (MMU) ترجمه شود. بخشی از جدول صفحه که با صفحات هسته سروکار دارد در تمام process ها یکسان است. با این حال، بر خلاف لینوکس، xv6 از demand paging استفاده نمیکند، بنابراین مفهومی از حافظه مجازی وجود ندارد.

به طور خلاصه لینوکس از VMA برای مدیریت نگاشت حافظه برای یک فرآیند استفاده می کند در حالی که xv6 از ساختار جدول صفحه دو سطحی ساده تری استفاده می کند.

2. چرا ساختار سلسله مراتبی منجر به کاهش مصرف حافظه می گردد؟

در بسیاری از برنامه‌ها، کل فضای آدرس به طور کامل به کار نمی‌رود. ممکن است بخش‌های بزرگی از فضای آدرس استفاده نشده یا پراکنده باشد. با صفحه‌بندی سلسله‌مراتبی، فقط بخش‌هایی از جداول صفحات مرتبط با فضای آدرس استفاده‌شده باید اختصاص داده و پر شوند، که به کاهش کل مصرف حافظه نسبت به یک جدول صفحات مسطح[4] که باید فضای لازم برای کل فضای آدرس را رزرو کند، منجر می‌شود.

با توجه به اینکه هر منطقه ای از حافظه که برای ناحیه ای که توسط شماره صفحه سطح بالا پوشش داده نشده است، نیازی به تخصیص جدول صفحه سطح پایین تر ندارد، از این رو از اختصاص جدول صفحه سطح پایین برای آن ناحیه جلوگیری می کند و در نتیجه منجر به کاهش حافظه می شود.

[4] Flat page table

3. محتوای هر بیت یک مدخل (32 بیتی) در هر سطح چیست؟ چه تفاوتی میان آنها وجود دارد؟

20 بیت در هر دو سطح وجود دارد که کارایی تقریبا یکسانی دارند هر کدام به سطح بالاتر خود اشاره دارند. در Page Directory به آدرس شروع جدول صفحه مورد نظر در حافظه اشاره میکند و در Page Table به آدرس قاب[5] مورد نظر در آدرس فیزیکی[6] اشاره دارد.

همچنین 12 بیت در هر دو سطح به عنوان سطح دسترسی نگه داری میشود.

Page Directory و Page Table هر دو مدخل های یکسانی دارند و تنها تفاوتشان در بیت D (dirty) می باشد که این بیت برای Page Table کاربردی ندارد اما Page Directory مشخص میکند که صفحه باید در دیسک نوشته شود تا تغییرات اعمال شود.

4. تابع kalloc چه حافظه ای تخصیص می دهد؟(فیزیکی یا مجازی)

قطع کد زیر مربوط به تابع kalloc است که در فایل kalloc.c قرار دارد :

[5] frame

[6] Physical memory

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

همانطور که در توضیحات خود xv6 در بالای کد کامنت شده است این تابع یک صفحه 4096 بایتی از حافظه فیزیکی را تخصیص می دهد. در صورتی که بتواند این مقدار از حافظه را تخصیص دهد اشاره گری به آن صفحه برمیگرداند که کرنل می تواند از آن استفاده کند و در صورتی که به هر دلیلی نتواند حافظه تخصیص دهد مقدار صفر را برمیگرداند.

5.تابع mappages چه کاربردی دارد؟

قطع کد زیر مربوط به تابع mappages است که در فایل vm.c قرار 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
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;
}
```

با توجه به توضیحات xv6 در کامنت بالای کد این تابع برای آدرس مجازی که از va شروع میشود PTE تشکیل می دهد و صفحه جدید را به آدرس فیزیکی که از pa شروع میشود اضافه می کند.(pgdir)

اگر این نگاشت موفقیت آمیز باشد، این تابع 0 و در غیر اینصورت 1- برمیگرداند.

7.راجع به تابع walkpgdir توضیح دهید. این تابع چه عمل سخت افزاری را شبیه سازی میکند؟

قطع کد زیر مربوط به تابع walkpgdir است که در فایل vm.c قرار دارد :

```
// 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)];
  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)];
}
```

با توجه به توضیحاتی که در بالای کد بصورت کامنت آورده شده است این تابع آدرس PTE مرتبط با آدرس مجازی va را از جدول صفحات pgdir مشخص می کند و برمیگرداند.هم چنین در صورت لزوم page table لازم را می سازد.

همانطور که مشخص است این تابع عملکردی مشابه عمل سخت افزاری ترجمه آدرس مجازی به آدرس فیزیکی دارد.

8.توابع allocuvm و mappages که در ارتباط با حافظه مجازی هستند را توضیح دهید.

تابع mappages در سوال 5 توضیح داده شد.

قطع کد زیر مربوط به تابع allocuvm است که در فایل vm.c قرار دارد :

```
// 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;
}
```

این تابع allocuvm در سیستم عامل Unix برای اختصاص دادن صفحات حافظه و حافظه فیزیکی به یک فرآیند استفاده می‌شود. این تابع برای توسعه اندازه فضای حافظه‌ی یک فرآیند، از oldsz به newsz صدا می‌شود. الگوریتم allocuvm در ابتدا محدودیت‌هایی را بررسی کرده و سپس برای هر صفحه از

oldsz تا newsz، یک صفحه حافظه فیزیکی تخصیص داده و به آن مقداردهی اولیه می‌کند. سپس نقشه‌های مورد نیاز بین فضای حافظه مجازی و فضای حافظه فیزیکی ایجاد می‌کند. در صورتی که این عملیات با مشکل مواجه شود، حافظه‌ی تخصیص یافته را آزاد کرده و مقدار صفر خروجی می دهد. در غیر اینصورت، اندازه جدید فضای حافظه را بازمی‌گرداند.

9.شیوه بارگذاری برنامه در حافظه توسط فراخوانی exec را شرح دهید.

ابتدا در این قسمت inode مربوط به path داده شده را با استفاده از تابع namei پیدا میکنیم و آن را در ip ذخیره میکنیم.

```
10      int
11      exec(char *path, char **argv)
12      {
13        char *s, *last;
14        int i, off;
15        uint argc, sz, sp, ustack[3+MAXARG+1];
16        struct elfhdr elf;
17        struct inode *ip;
18        struct proghdr ph;
19        pde_t *pgdir, *oldpgdir;
20        struct proc *curproc = myproc();
21
22        begin_op();
23
24        if((ip = namei(path)) == 0){
25          end_op();
26          cprintf("exec: fail\n");
27          return -1;
28        }
```

در ادامه، ELF header فایل مربوطه را چک میکنیم تا اطمینان حاصل کنیم که یک فایل اجرایی معتبر باشد.

```
32        // Check ELF header
33        if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
34          goto bad;
35        if(elf.magic != ELF_MAGIC)
36          goto bad;
```

سپس از تابع setupkvm برای ایجاد مجموعه جدیدی از جدول های صفحه برای process استفاده می کنیم.

```
38        if((pgdir = setupkvm()) == 0)
39          goto bad;
```

حال به قطعه کد مربوط به بارگذاری برنامه در حافظه میرسیم. در این قسمت، یک حلقه میزنیم که روی هدر های برنامه در ELF file پیمایش میکند.

توسط readi، هر هدر را خوانده و در ph ذخیره میکنیم. سپس توسط تابع allocuvm یک فضای حافظه ی جدید برای این بخش از process متناسب با مقدار نیاز یعنی به سایز `ph.vaddr + ph.memsz` تخصیص می دهیم و سپس توسط تابع loaduvm این بخش را به حافظه ای که گرفتیم لود میکنیم.

در تمامی این فرایند توضیح داده شده، ممکن است به ارور هایی بخوریم که در کد زیر، تمامی این ارور ها را میتوان مشاهده کرد.

```
41      // Load program into memory.
42      sz = 0;
43      for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
44        if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
45          goto bad;
46        if(ph.type != ELF_PROG_LOAD)
47          continue;
48        if(ph.memsz < ph.filesz)
49          goto bad;
50        if(ph.vaddr + ph.memsz < ph.vaddr)
51          goto bad;
52        if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
53          goto bad;
54        if(ph.vaddr % PGSIZE != 0)
55          goto bad;
56        if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
57          goto bad;
58      }
59      iunlockput(ip);
60      end_op();
61      ip = 0;
```

پیاده سازی

در ابتدا برای حافظه اشتراکی دو struct در فایل shm.h اضافه می‌کنیم. در استراکچر shared_page اطلاعات یک صفحه مشترک نگهداری می‌شود که شامل id آن صفحه، تعداد رفرنس‌ها و فریم فیزیکی آن است.

```c
#define MAX_SHARED_PAGES 16

struct shared_page
{
    int id;
    uint ref_count;
    char *frame;
};
```

استراکچر دوم شامل آرایه‌ای از این صفحات و یک قفل برای جلوگیری از race condition می‌باشد.

```c
struct shared_memory
{
    struct shared_page shared_pages[MAX_SHARED_PAGES];
    struct spinlock slk;
};
```

حال برای استفاده از این حافظه مشترک آن را در فایل proc.c تعریف می‌کنیم.

```
// shared memory definition
struct shared_memory shm;
```

مشخصا باید تابعی برای مقداردهی اولیه این حافظه داشته باشیم.

```
// shared memory initialize function
void shm_init()
{
  for (int i = 0; i < MAX_SHARED_PAGES; i++)
  {
    shm.shared_pages[i].id = -1;
    shm.shared_pages[i].frame = (char *)0;
  }

  initlock(&shm.slk, "spin lock");
}
```

این تابع تمام آیدی ها را برابر منفی یک می‌گذارد و قفل موجود در صفحه را مقداردهی می‌کند.

در ادامه برای اینکه از ابتدای اجرای os به این حافظه دسترسی داشته باشیم تابع shm_init را در تابع main سیستم عامل فراخوانی می‌کنیم.

```
binit();                         // buffer cache
fileinit();                      // file table
ideinit();                       // disk
shm_init();
startothers();                                  // sta
kinit2(P2V(4 * 1024 * 1024), P2V(PHYSTOP)); // mus
userinit();                                     // fir
```

حال تابع shm_open را تعریف می‌کنیم. عملکرد این تابع به شرح زیر است :

این تابع ابتدا بررسی می‌کند که آیا صفحه‌ای که به دنبال آنیم وجود دارد یا خیر. در صورتی که وجود نداشت به دنبال اولین جای خالی در آرایه صفحات می‌گردد و آن را با مقادیر مناسب مقداردهی می‌کند. در ادامه این آدرس فیزیکی را یه فضای آدرس مجازی process فعلی مپ می‌کند.

حال در صورتی که این صفحه وجود داشت بررسی می‌کند که آیا پردازه فعلی این صفحه را در اختیار دارد یا خیر .

توجه شود که این کار به وسیله آرایه shared_addresses که در استراکچر process تعریف کرده‌ایم استفاده می‌کند.

```
struct rtcdate init_time;
struct bjf_info bjf_info;
enum MLFQ queue;
int last_run;
int last_in_lcfs;
int shared_addresses[MAX_SHARED_PAGES];
;
```

```
for (int i = 0; i < MAX_SHARED_PAGES; i++)
{
  p->shared_addresses[i] = -1;
}

return p;
```

(مقداردهی اولیه در allocproc)

اگر این صفحه را در اختیار نداشت آن را به فضای آدرس مجازی process مپ می‌کند. و این آدرس مجازی را بازمی‌گرداند.

```c
// open the shared memory region
void *shm_open(int id)
{
  struct proc *proc = myproc();
  pde_t *pgdir = proc->pgdir;
  int page_index = -1;

  acquire(&shm.slk);

  for (int i = 0; i < MAX_SHARED_PAGES; i++)
  {
    if (shm.shared_pages[i].id == id)
    {
      page_index = i;
      break;
    }
  }

  if (page_index == -1)
  {
    int free_index = -1;
    for (int i = 0; i < MAX_SHARED_PAGES; i++)
    {
      if (shm.shared_pages[i].id == -1)
      {
        free_index = i;
        break;
      }
    }
```

```
  if (free_index == -1)
  {
    release(&shm.slk);
    return (char *)-1;
  }

  char *page;
  page = kalloc();
  memset(page, 0, PGSIZE);
  mappages(pgdir, (void *)PGROUNDUP(proc->sz), PGSIZE, V2P(page), PTE_W | PTE_U);
  proc->shared_addresses[free_index] = PGROUNDUP(proc->sz);
  shm.shared_pages[free_index].frame = page;
  shm.shared_pages[free_index].ref_count++;
  shm.shared_pages[free_index].id = id;
  release(&shm.slk);
  return (void *)proc->shared_addresses[free_index];
}
else
{
  if (proc->shared_addresses[page_index] != -1)
  {
    release(&shm.slk);
    return (void *)-1;
  }

  mappages(pgdir, (void *)PGROUNDUP(proc->sz), PGSIZE, V2P(shm.shared_pages[page_index].frame), PTE_W | PTE_U);
  proc->shared_addresses[page_index] = PGROUNDUP(proc->sz);
  shm.shared_pages[page_index].ref_count++;
  release(&shm.slk);
  return (void *)proc->shared_addresses[page_index];
}
}
```

تابع بعدی ، تابع shm_close است. این تابع برای بستن فضای اشتراکی ابتدا بررسی می‌کند که آیا آن فضا موجود باشد و توسط process فعلی باز شده باشد. در ادامه این فضا را از فضای آدرس مجازی process حذف کرده و همچنین بررسی می‌کند که درصورتی که دیگر رفرنسی به آن صفحه فیزیکی نبود آن صفحه را از حافظه فیزیکی نیز حذف می‌کند و entry آن در جدول صفحات مشترک را خالی می‌کند.

```c
// close the shared memory region
int shm_close(int id)
{
  struct proc *proc = myproc();

  int page_index = -1;
  acquire(&shm.slk);

  for (int i = 0; i < MAX_SHARED_PAGES; i++)
  {
    if (shm.shared_pages[i].id == id)
    {
      page_index = i;
      break;
    }
  }

  if (page_index == -1 || proc->shared_addresses[page_index] == -1)
  {
    release(&shm.slk);
    return -1;
  }
```

```
  uint a = PGROUNDUP(proc->shared_addresses[page_index]);
  pte_t *pte = walkpgdir(proc->pgdir, (char *)a, 0);

  if (!pte)
    a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
  else if ((*pte & PTE_P) != 0)
  {
    *pte = 0;
  }

  proc->shared_addresses[page_index] = -1;

  shm.shared_pages[page_index].ref_count--;
  if (shm.shared_pages[page_index].ref_count == 0)
  {
    kfree(shm.shared_pages[page_index].frame);
    shm.shared_pages[page_index].id = -1;
    shm.shared_pages[page_index].frame = (void *)0;
  }

  release(&shm.slk);
  return 0;
}
```

حال برای استفاده از دو تابع بالا فراخوانی‌های سیستمی زیر را تعریف می‌کنیم :

```
void *sys_open_sharedmem()
{
  int id;
  char *pointer;

  if (argint(0, &id) < 0)
    return -1;

  pointer = (char *)shm_open(id);

  if (pointer == (char *)-1)
  {
    cprintf("Failed to open shared memory region!\n");
  }
  return pointer;
}
```

```
int sys_close_sharedmem()
{
  int id;

  if (argint(0, &id) < 0)
    return -1;

  int res = shm_close(id);
  if (res == -1)
  {
    cprintf("Failed to close shared memory region!\n");
  }
  return res;
}
```

در ادامه برنامه تست shm_test را طراحی می‌کنیم :

```
int main(int argc, char const *argv[])
{

    char *shared_mem = (char *)open_sharedmem(SHM_ID);
    char *value = (char *)shared_mem;
    *value = 0;

    for (int i = 0; i < NUM_CHILDREN; i++)
    {
        int pid = fork();
        if (pid < 0)
        {
            printf(1, "Error forking child process.\n");
            exit();
        }
        else if (pid == 0)
        {
            char *shared_mem = (char *)open_sharedmem(1);
            acquire_prioritylock();
            char *value = (char *)shared_mem;
            *value += 1;
            printf(1, "Child proc with pid %d shared memory value is : %d\n", getpid(), *value);
            release_prioritylock();
            close_sharedmem(SHM_ID);
            exit();
        }
    }
```

```
    for (int i = 0; i < NUM_CHILDREN; i++)
    {
        wait();
    }

    printf(1, "Final shared memory value is : %d\n", *value);
    close_sharedmem(SHM_ID);
    exit();
}
```

خروجی برنامه باید ۴ باشد که مطابق شکل زیر صحیح است :