آزمایشگاه سیستم عامل

پروژه پنج

اعضای گروه:

الهه خداوردی - 810100132

فرشته باقری - 810100089

عاطفه میرزاخانی - 810100220

Repository: https://github.com/elahekhodaverdi/Operating-System-Lab-Projects

Latest Commit:

مقدمه

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

در لینوکس، هسته از نواحی virtual memory با پیگیری memory mapping های پردازه استفاده می‌کند. برای مثال یک پردازه یک VMA برای کد، یک VMA برای هر نوع دیتا و یک VMA برای هر memory mapping دارد. هر VMA شامل تعدادی page است که هر کدام از این page ها یک entry به page table دارد اما xv6 از آدرس‌های مجازی 32 بیتی استفاده می‌کند که فضای آدرسی مجازی 4 گیگابایتی ایجاد می‌کند. همچنین xv6 از جدول دو سطحی استفاده می‌کند و مفهومی از حافظه مجازی ندارد.

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

اگر ساختار سلسله مراتبی نداشته باشیم، در این صورت در یک کامپیوتر 32-بیتی که اندازه هر page آن 4KB است به 2^{20} صفحه نیاز داریم یعنی page table باید دارای تقریبا 1M مدخل باشد و از آنجا که هر مدخل 32 بیتی است یعنی به حدودا 4MB حافظه برای نگه‌داری page table های هر پردازه نیاز داریم. ولی اگر از ساختار سلسله مراتبی استفاده کنیم هر کدام از page table ها و جدول page directory فقط 4KB هستند و کافی است برای هر پردازه جدول page directory را نگه‌داری کنیم و هر کدام از page table ها را در صورت نیاز ایجاد کنیم.

3) محتوای هر بیت یک مدخل (۳۲ بیتی) در هر سطح چیست؟ چه تفاوتی میان آن ها وجود دارد؟

در هر دو سطح 12 بیت برای سطح دسترسی وجود دارد. 20 بیت دیگر در سطح page table برای آدرس صفحه فیزیکی استفاده می‌شود و در سطح page directory برای اشاره به سطح بعدی استفاده می‌شود.
همچنین در بیت D یعنی بیت dirty تفاوت دارند. در page directory این بیت به معنای آن است که صفحه باید در دیسک نوشته شود تا تغییرات اعمال شود اما در page table این بیت معنایی ندارد.

مدیریت حافظه در xv6

4) تابع kalloc چه نوع حافظه‌ای تخصیص می‌دهد؟ (فیزیکی یا مجازی)

همانطور که در ابتدای فایل kalloc.c در کامنت ها میبینیم میتوانیم متوجه شویم که این تابع برای اختصاص دادن حافظه فیزیکی است.

```
// Physical memory allocator, intended to allocate
// memory for user processes, kernel stacks, page table pages,
// and pipe buffers. Allocates 4096-byte pages.
```

این تابع برا اختصاص حافظه به ساختار های هسته همانند استک، pagetable و اختصاص حافظه به پردازه‌ها استفاده می‌شود.
در کد این تابع :

```
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 میبینیم ابتدا قفل مربوط به حافظه خالی کسب میشود و سپس اولین فضای خالی موجود گرفته می‌شود و سپس اولین فضای خالی را نیز اپدیت میکنیم و در نهایت پوینتر به حافظه را برمیگردانیم.
در xv6 حافظه به صورت پیج های 4096 بایتی تقسیم بندی می‌شود و این پوینتر به این فضاهای خالی در استراکت kmem ذخیره می‌شود.
در ابتدای شروع در تابع main با فراخوانی های مربوط به kinit سیستم عامل حافظه بعد کرنل تا انتها را free میکند و آن هارا در kmem نگهداری می‌کند.

```
kinit1(end, P2V(4*1024*1024));
kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
```

که در کد بالا end به صورت اولین حافظه خالی در حافظه بعداز کرنل تعریف شده است و PHYSTOP نیز پایان حافظه است.

5) تابع mappages() چه کاربردی 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;
}
```

همانطور که کامنت های مربوط به تابع نشان می‌دهد این تابع به منظور ساخت نگاشت از آدرس مجازی به فیزیکی استفاده می‌شود. به این صورت کار میکند که ابتدا یک page table entry می‌سازد و برای این کار از تابع walkpgdir استفاده می‌کند.

تابع walkpgdir به این صورت کار میکند که ادرس page table entry که به آدرس مجازی va در pgdir نگاشت شده را برمی‌گرداند و اگر همچین نگاشتی وجود نداشته باش page table page میسازد.

از این تابع در توابع زیر استفاده می‌شود:

- Copyuvm: *Given a parent process's page table, create a copy of it for a child.*
- Allocuvm: *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.*
- Inituvm: *Load the initcode into address 0 of pgdir.*
- Setupkvm: *Set up kernel part of a page table.*

همچنین فلگ های استفاده شده در تابع mappages به صورت زیر تعریف شده‌اند:

```
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size
```

7) راجع به تابع walkpgdir() توضیح دهید. این تابع چه عمل سخت‌افزاری را شبیه‌سازی می‌کند؟

به کارکرد این تابع در سوال پیشین نیز اشاره شد.

این تابع به منظور نگاشت آدرس مجازی به آدرس فیزیکی استفاده می‌شود. به این صورت که اگر PTE در pgdir وجود داشت که به آدرس مجازی با شروع از va اشاره داشت آن را برمیگرداند و اگر وجود نداشت جدولی میسازد و آدرس آن را برمیگرداند.

می‌توان گفت که این تابع عمل سخت افزاری ترجمه آدرس مجازی به فیزیکی را شبیه‌سازی می‌کند.

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

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

mappages : این تابع مسئول برقراری یک نگاشت بین محدوده ای از آدرس های مجازی و آدرس های فیزیکی است. این تابع زمانی استفاده می شود که سیستم عامل نیاز به ایجاد یک ارتباط بین حافظه مجازی که فرآیندها می بینند و حافظه فیزیکی که سخت افزار استفاده می کند دارد.

پارامترها: این تابع یک دایرکتوری صفحه، محدوده ای از آدرس های مجازی، آدرس های فیزیکی متناظر و permission fllags را به عنوان پارامتر می گیرد.

نگاشت: این تابع بر روی محدوده آدرس های مجازی حلقه می زند. برای هر آدرس، آن را در دایرکتوری صفحه متناظر می یابد.

ورودی جدول صفحه (PTE): اگر یک Page Table Entry برای آدرس مجازی فعلی وجود نداشته باشد، mappages یکی ایجاد می کند. سپس آدرس فیزیکی PTE را به آدرس فیزیکی متناظر تنظیم می کند و flag را اعمال می کند.

مدیریت خطا: اگر mappages هنگام ایجاد یک PTE با خطا مواجه شود (برای مثال، اگر حافظه آزادی برای یک جدول صفحه جدید وجود نداشته باشد)، یک خطا برمی گرداند.

در اصل، mappages مسئول اطمینان از این است که وقتی یک فرآیند به یک آدرس مجازی دسترسی پیدا می کند، سخت افزار به درستی آن دسترسی را به آدرس فیزیکی مناسب مپ می کند.

allocuvm: این تابع کوتاه شده "Allocate User Virtual Memory" است. این تابع مسئول افزایش حافظه مجازی کاربر در یک دایرکتوری صفحه خاص است. دو حالت وجود دارد که این تابع می تواند fail شود :

حالت 1: اگر تابع kalloc شکست بخورد. این تابع مسئول بازگرداندن آدرس یک صفحه جدید، در حال حاضر استفاده نشده، در RAM است. اگر 0 برگرداند، به این معنی است که در حال حاضر صفحه استفاده نشده ای وجود ندارد.

حالت 2: اگر تابع mappages شکست بخورد. این تابع مسئول این است که صفحه تازه تخصیص داده شده را با استفاده از دایرکتوری صفحه داده شده برای فرآیند که از آن استفاده می کند، با مپ کردن آن صفحه با آدرس مجازی بعدی موجود در دایرکتوری صفحه، قابل دسترسی کند. اگر این تابع شکست بخورد، احتمالا به این معنی است که دایرکتوری صفحه در حال حاضر پر است.

در هر دو حالت، allocuvm موفق به افزایش حافظه کاربر به اندازه درخواست شده نشده است، بنابراین تمام تخصیص ها را تا نقطه شکست، برگردانده و حافظه مجازی تغییر نکرده و خودش یک خطا برمی گرداند.

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

تابع exec در سیستم عامل های شبیه به Unix، از جمله xv6، برای جایگزین کردن فرآیند در حال اجرا با یک فرآیند جدید استفاده می شود.

پاک کردن حالت حافظه فعلی: تابع exec ابتدا حالت حافظه فرآیند فراخواننده را پاک می کند.

یافتن فایل برنامه: سپس به سیستم فایل می رود تا فایل برنامه درخواستی را پیدا کند.

کپی کردن فایل برنامه: exec این فایل را در حافظه برنامه کپی می کند.

مقداردهی اولیه وضعیت register ها : از جمله شمارنده برنامه (PC)، که به دستور بعدی برای اجرا اشاره می کند.

بارگذاری برنامه جدید: برنامه جدید در فضای آدرس همان، جایگزین قابل اجرا قبلی می شود. این عمل همچنین به عنوان یک پوشش معرفی می شود.

شناسه فرآیند (PID) همان است: از آنجا که فرآیند جدیدی ایجاد نمی شود، شناسه فرآیند (PID) تغییر نمی کند.

تابع exec مگر در صورت وجود خطا، return نمی کند. این به این دلیل است که فرآیند در حال اجرا کاملاً توسط فرآیند جدید جایگزین می شود.

Shared Memory

در این پروژه برای ایجاد فضای اشتراکی بین پردازه ها نیازمند تعریف سه struct جدید بودیم:

```
// vm.c
struct shmRegion
{
  uint key, size;
  int shmid;
  void *physicalAddr[SHAREDREGIONS];
  uint shm_segsz;
  int shm_nattch;
};

struct shmTable
{
  struct spinlock lock;
  struct shmRegion allRegions[SHAREDREGIONS];

} shmTable;

// proc.h
typedef struct sharedPages {
  uint key, size;
  int shmid;
  void *virtualAddr;
} sharedPages;
```

استراکت اول مربوط به ناحیه اشتراکی است که شامل آیدی ناحیه، سایز (تعداد page های ناحیه)، سایز ناحیه و تعداد پردازه های مرتبط به ناحیه و همچنین آدرس فیزیکی مربوط به هر page را ذخیره می‌کند.
استراکت دوم مربوط به جدولی است که تمام ناحیه های اشتراکی موجود را ذخیره سازی می‌کند و همچنین یک قفل برای کنترل دسترسی‌ها و تغییرات وجود دارد.
استراکت سوم مربوط به فضای اشتراکی است که هر پردازه می‌تواند در اختیار داشته باشد. در هر پردازه لیستی از این استراکت تعریف کردیم.

```
  sharedPages pages[SHAREDREGIONS];
```

Open Shared Memory

در این سیستم کال ابتدا چک میکنیم که اگر آن حافظه اشتراکی از قبل موجود نبود ابتدا ان را با استفاده از تابع create_shm میسازیم و سپس مراحل attach کردن حافظه به پردازه انجام می‌شود.

پس از attach شدن هر پردازه نیز nattch مربوط به ناحیه اشتراکی را یکی زیاد می‌کنیم.

```
void *
open_sharedmem(int shmid)
{
  if (shmid < 0 || shmid > 64)
     // return failure
  acquire(&shmTable.lock);
  int index = -1, idx;
  void *va = (void *)HEAPLIMIT, *least_va;
  struct proc *process = myproc();
  index = shmTable.allRegions[shmid].shmid;
  if (index == -1)
      // create_shared_memory
  if (index == -1)
      // return failure

  for (int i = 0; i < SHAREDREGIONS; i++)
  {
    idx = getLeastvaidx(va, process);
    if (idx != -1)
    {
      least_va = process->pages[idx].virtualAddr;
      if ((uint)va + shmTable.allRegions[index].size * PGSIZE <= (uint)least_va)
        break;
      else
        va = (void *)((uint)least_va + process->pages[idx].size * PGSIZE);
    }
    else
      break;
  }

  if ((uint)va + shmTable.allRegions[index].size * PGSIZE >= KERNBASE)
      // return failure
  idx = -1;
```

```
  for (int i = 0; i < SHAREDREGIONS; i++)
  {
    if (process->pages[i].key != -1 && (uint)process->pages[i].virtualAddr +
process->pages[i].size * PGSIZE > (uint)va && (uint)va >=
(uint)process->pages[i].virtualAddr)
    {
      idx = i;
      break;
    }
  }
  if (idx != -1)
      // return failure
  for (int k = 0; k < shmTable.allRegions[index].size; k++)
  {
    if (mappages(process->pgdir, (void *)((uint)va + (k * PGSIZE)), PGSIZE,
(uint)shmTable.allRegions[index].physicalAddr[k], 06) < 0)
    {
      deallocuvm(process->pgdir, (uint)va, (uint)(va +
shmTable.allRegions[index].size));
      // return failure
    }
  }
  idx = -1;
  for (int i = 0; i < SHAREDREGIONS; i++)
      // get index of the first unused sharedpages in proc
  if (idx != -1)
      // attaches shared region to process and update shared region
  else
      // return failure
  return va;
}
```

تابع create_shm نیز به نحو زیر پیاده سازی شده است:

```
int create_shm(uint size, int index)
{
  acquire(&shmTable.lock);
  if (size <= 0)
  {
    release(&shmTable.lock);
    return -1;
  }
  int num_of_pages = (size / PGSIZE) + 1;
  if (num_of_pages > SHAREDREGIONS)
  {
    release(&shmTable.lock);
    return -1;
  }
  for (int i = 0; i < num_of_pages; i++)
  {
    char *new_page = kalloc();
    if (new_page == 0)
    {
      cprintf("Create_shm: failed to allocate a page (out of memory)\n");
      release(&shmTable.lock);
      return -1;
    }
    memset(new_page, 0, PGSIZE);
    shmTable.allRegions[index].physicalAddr[i] = (void *)V2P(new_page);
  }
  shmTable.allRegions[index].size = num_of_pages;
  shmTable.allRegions[index].key = 0;
  shmTable.allRegions[index].shm_segsz = size;
  shmTable.allRegions[index].shmid = index;

  release(&shmTable.lock);
  return index;
}
```

Close Shared Memory

در این تابع نیز آدرس مربوط به ناحیه اشتراکی را گرفته و آن را از لیست ناحیه های اشتراکی پردازه حذف میکنیم و به موجب آن از مقدار nattch ناحیه کم می‌کنیم و اگر این مقدار به 0 رسید ناحیه اشتراکی را از جدول حافظه اشتراکی حذف میکنیم.

```
int close_sharedmem(void *shmaddr)
{
  acquire(&shmTable.lock);
  struct proc *process = myproc();
  void *va = (void *)0;
  uint size;
  int index, shmid;
  for (int i = 0; i < SHAREDREGIONS; i++){
       // find index(and another data) of sharedpages with this address
       }
  if (va)
  {
    for (int i = 0; i < size; i++)
    {
      pte_t *pte = walkpgdir(process->pgdir, (void *)((uint)va + i * PGSIZE), 0);
      if (pte == 0)
      {
        release(&shmTable.lock);
        return -1;
      }
      *pte = 0;
    }
    // detach shared memory from process sharedpages
       ...
    if (shmTable.allRegions[shmid].shm_nattch > 0)
      shmTable.allRegions[shmid].shm_nattch -= 1;
    if (shmTable.allRegions[shmid].shm_nattch == 0)
    {
      for (int i = 0; i < shmTable.allRegions[index].size; i++)
      {
        char *addr = (char *)P2V(shmTable.allRegions[index].physicalAddr[i]);
        kfree(addr);
        shmTable.allRegions[index].physicalAddr[i] = (void *)0;
      }
      shmTable.allRegions[index].size = 0;
      shmTable.allRegions[index].key = shmTable.allRegions[index].shmid = -1;
```

```
      shmTable.allRegions[index].shm_nattch = 0;
      shmTable.allRegions[index].shm_segsz = 0;
      cprintf("number of refrences is 0, shared memory is freed.\n");
    }
    release(&shmTable.lock);
    return 0;
  }
  else
      // return failure
}
```

برنامه آزمون

```
void test_sharedmem_increment() {
  int shmid = 0;
  void *addr = (void *)open_sharedmem(shmid);

  for (int i = 0; i < NCHILD; i++) {
    int pid = fork();
    if (pid < 0) {
      printf(1, "fork failed\n");
      return;
    } else if (pid == 0) {

      (*(int *)addr)++;
      exit();
    }
  }
  for (int i = 0; i < NCHILD; i++) {
    wait();
  }
  printf(1, "Final value in shared memory: %d\n", *(int *)addr);
}
int main(void) {
  test_sharedmem_increment();
  exit();
}
```

نتیجه اجرای برنامه آزمون :