آزمایشگاه سیستم عامل

تمرین کامپیوتری ۵

اعضای گروه :

علی حمزه پور- ۸۱۰۱۰۰۱۲۹

نرگس سادات سیدحائری – ۸۱۰۱۰۰۱۶۵

مینا شیرازی - ۸۱۰۱۰۰۲۵۰

مقدمه

سوال یک: راجع به مفهوم ناحیه مجازی در لینوکس به طور مختصر توضیح داده و آن را با xv٦ مقایسه کنید.

در لینوکس، نواحی حافظه مجازی (VMA) برای نمایش مناطق مختلف فضای آدرس مجازی یک پردازه استفاده می‌شود که جزئیاتی چون حفاظت حافظه، تخصیص حافظه پویا و نقشه‌برداری حافظه را مدیریت می‌کند.

در سیستم عامل لینوکس از مفهوم VMA به صورت گسترده برای مدیریت حافظه مجازی پردازه ها استفاده می‌شود و از page table برای ایجاد تناظر بین آدرس مجازی و فیزیکی استفاده می شود. هر VMA شامل تعدادی entry از page tableمتناظر است، زمانی که یک پردازه به یک آدرس مجازی دسترسی پیدا می کند entry- های متناظر آدرس مجازی را به فیزیکی ترجمه می کنند. درxv٦ از VMA استفاده نمی شود بلکه هسته آن از یک مکانیزم مدیریت ساده تر که به صورت مستقیم آدرس مجازی را به فیزیکی تبدیل می کند استفاده می کند .

سوال دو: چرا ساختار سلسله مراتبی منجر به کاهش مصرف حافظه می‌گردد؟

در ساختار سلسله‌مراتبی، processها و task ها به راحتی میتوانند با به اشتراک گذاشتن کدها و داده‌ها توسط mapping بخش مناسب به صفحات فیزیکی از مصرف اضافی حافظه جلوگیری کنند. مپ کردن به صفحات به ساختار اجازه می‌دهد که صفحات مختلف حافظه به ترتیب دسترسی قرار گیرند و به صورت دینامیک مدیریت شوند، که باعث بهره‌وری در استفاده از حافظه و کاهش زمان دسترسی به داده‌ها می‌شود. این ویژگی‌ها باعث بهبود کارایی و کاهش مصرف حافظه در سیستم می‌شوند.

سوال سه: محتوای هر بیت یک مدخل (۳۲ بیتی)در هر سطح چیست؟ چه تفاوتی میان آنها وجود دارد؟

مدخل ۳۲ بیتی از دو بخش تشکیل شده است؛ بخشی برای اشاره به سطح بعدی حافظه و بخشی دیگر برای تعیین سطح دسترسی به داده‌ها. از کل تعداد ۳۲ بیت، ۲۰ بیت برای اشاره به سطح بعدی حافظه (به عنوان مثال، جدول صفحه) اختصاص داده شده‌اند. باقی مانده ۱۲ بیت برای تعیین سطح دسترسی به داده‌ها در هر سطر حافظه می‌باشند. در سطح جدول صفحه، از این ۲۰ بیت برای تشخیص آدرس فیزیکی استفاده می‌شود. علاوه بر این، در این مدل حافظه یک بیت به نام Dirty (D) وجود دارد که در سطوح مختلف دارای تفاوت است. در سطح Page Table، این بیت معنای خاصی ندارد. اما در سطح Page Directory به این معناست که صفحه باید در دیسک نوشته شود. به عبارت دیگر، اگر بیت Dirty فعال باشد، نشانگر است که تغییراتی روی صفحه

انجام شده است و باید این تغییرات در دیسک ذخیره شوند. این ویژگی به عنوان شرطی در نظر گرفته می‌شود که برای اعمال تغییرات در داده‌ها دارای اهمیت است.

کد مربوط به ایجاد فضاهای آدرس درxv٦

سوال چهار: تابع ()kalloc چه نوع حافظه‌ای تخصیص می‌دهد؟

تابع kalloc به‌صورت زیر تعریف شده است:

```
// Allocate one ٤٠٩٦-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns ٠ 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;
}
```

در سیستم عامل xv٦، این تخصیص حافظه به صورت فیزیکی انجام می‌شود. در واقع kalloc درxv٦ صفحات حافظه‌ای فیزیکی به طول ٤٠٩٦ بایت را اختصاص می‌دهد، که مستقیماً در حافظه فیزیکی سیستم قرار دارد. این تابع برای تخصیص حافظه در kernel heap برای ذخیره سازی ساختمان‌های پویا استفاده می‌شود.بدین صورت که در لیستی از فضاهای خالی به دنبال block memory-ای خالی که به اندازه کافی بزرگ باشد می گردد و سپس آن را از لیست فضاهای خالی خارج می کند و اگر نتواند آن را پیدا کند مقدار صفر را برمی‌گرداند.

سوال پنج: تابع ()mappages چه کاربردی دارد؟

تابع ()mappages به صورت مقابل است:

```
// 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 - ١);
  for(;;){
    if((pte = walkpgdir(pgdir, a, ١)) == ٠)
      return -١;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return ٠;
}
```

این تابع به منظور ساخت نگاشت از آدرس مجازی به فیزیکی استفاده می‌شود.

در اینجا چند فلگ به شرح زیر تعریف شده‌اند:

- PTE_P: نشان‌دهنده حاضر بودن صفحه (Present) در حافظه.

- PTE_W: نشان‌دهنده امکان نوشتن (Writeable) در صفحه.
- PTE_U: نشان‌دهنده امکان دسترسی توسط کاربر (User) به صفحه.
- PTE_PS: نشان‌دهنده اندازه بزرگی صفحه (Page Size)، به معنای استفاده از صفحات بزرگ.

تابع mappages ابتدا آدرس مجازی (va) و اندازه مورد نظر (size) را به آدرسی که به صفحه‌بندی شده است تبدیل می‌کند. سپس با استفاده از حلقه، از تابع walkpgdir برای پیدا کردن یا ایجاد PTE مربوط به هر آدرس مجازی استفاده می‌کند. اگر PTE قبلاً تعریف شده باشد (با بررسی بیت PTE_P)، با یک panic به خطا می‌افتد؛ در غیر این صورت، یک PTE جدید با مشخصات مربوط به آدرس فیزیکی (pa)، مجوزهای دسترسی (perm) و بیت PTE_P (نشان‌دهنده فعال بودن PTE) ایجاد می‌شود. حلقه تا زمانی ادامه پیدا می‌کند که به آخرین آدرس مجازی (last) برسد. در نهایت، تابع باز می‌گردد و ۰ را ارجاع می‌دهد تا نشان دهد که عملیات موفقیت‌آمیز بوده است، مگر اینکه در طول اجرا با مشکلی مواجه شود که در آن صورت ۱- برگردانده می‌شود.

سوال هفت: راجع به تابع walkpgdir توضیح دهید. این تابع چه عمل سخت‌افزاری را شبیه‌سازی می‌کند؟

این تابع PTE مربوط به یک آدرس مجازی را از page table پیدا می‌کند. در حقیقت این تابع یک آدرس مجازی را به آدرس فیزیکی‌اش تبدیل می‌کند. در صورتی که PTE مورد نظر وجود نداشته باشد و پارامتر alloc غیر صفر باشد نیز یک PTE برای آن آدرس می‌سازد. تعریف و پیاده‌سازی این تابع به شکل زیر است:

```
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*)P۲V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == ۰)
      return ۰;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, ۰, PGSIZE);
```

```
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V۲P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

سوال هشت: توابع allocuvm و mappages که در ارتباط با حافظه‌ی مجازی هستند را توضیح دهید.

تابع mappages همانطور که به تفصیل در سوال ۵ توضیح داده شد یک حافظه‌ی مجازی را به حافظه‌ی فیزیکی متصل می‌کند.

تابع allocuvm ناحیه‌ی حافظه‌ی مجازی یک پردازه را از oldsz به newsz افزایش می‌دهد. این تابع تا زمانی که اندازه‌ی حافظه‌ی اولیه به مقدار خواسته شده برسد، حافظه‌ی فیزیکی allocate می‌کند و آن را با استفاده از تابع mappages به یک حافظه‌ی مجازی در پردازه متصل می‌کند.

پیاده‌سازی این تابع به شکل زیر است:

```
// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or ۰ on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return ۰;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
```

```
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == ٠){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return ٠;
    }
    memset(mem, ٠, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V۲P(mem), PTE_W|PTE_U) < ٠){
      cprintf("allocuvm out of memory (۲)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return ٠;
    }
  }
  return newsz;
}
```

سوال نه: شیوه‌ی بارگذاری برنامه در حافظه توسط فراخوانی سیستمی exec را شرح دهید.

تابع exec ابتدا فایل محتوای برنامه‌ای که باید بارگذاری شود را باز می‌کند.

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[۳+MAXARG+۱];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
```

```
struct proc *curproc = myproc();

begin_op();

if((ip = namei(path)) == ٠){
  end_op();
  cprintf("exec: fail\n");
  return -١;
}
ilock(ip);
pgdir = ٠;
```

سپس هدرهای آن فایل چک می‌شود و بعد با صدا زده شدن تابع setupkvm بخش هسته‌ی page table برای برنامه‌ی جدید ساخته می‌شود.

```
// Check ELF header
 if(readi(ip, (char*)&elf, ٠, sizeof(elf)) != sizeof(elf))
   goto bad;
 if(elf.magic != ELF_MAGIC)
   goto bad;

 if((pgdir = setupkvm()) == ٠)
   goto bad;
```

سپس در یک حلقه محتوای برنامه در حافظه‌ی پردازه ذخیره می‌شود. این حلقه هربار قسمتی از فایلی که برنامه در آن قرار دارد را می‌خواند و با استفاده از تابع allocuvm حافظه‌ی پردازه را زیاد می‌کند تا بتواند قسمتی را که خوانده است در حافظه‌ی پردازه ذخیره کند که با استفاده از تابع loaduvm این‌کار را انجام می‌دهد.

```
// Load program into memory.
 sz = ٠;
 for(i=٠, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
```

```
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == ۰)
    goto bad;
  if(ph.vaddr % PGSIZE != ۰)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < ۰)
    goto bad;
 }
```

در ادامه دو page ساخته می‌شود که اولی غیر قابل دسترسی می‌شود(برای گذاشتن فاصله و عدم رخ دادن مشکلات اورفلو و نظیر آن) و دومی برای پشته برنامه در نظر گرفته می‌شود. پارامترهای ورودی (args) مربوط به برنامه نیز در همین استک ذخیره می‌شوند. در نهایت page table قبلی آن پردازه آزاد می‌شود.

```
// Allocate two pages at the next page boundary.
 // Make the first inaccessible.  Use the second as the user stack.
 sz = PGROUNDUP(sz);
 if((sz = allocuvm(pgdir, sz, sz + ۲*PGSIZE)) == ۰)
   goto bad;
 clearpteu(pgdir, (char*)(sz - ۲*PGSIZE));
 sp = sz;

 // Push argument strings, prepare rest of stack in ustack.
 for(argc = ۰; argv[argc]; argc++) {
   if(argc >= MAXARG)
     goto bad;
   sp = (sp - (strlen(argv[argc]) + ۱)) & ~۳;
```

```
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + ١) < ٠)
    goto bad;
  ustack[٣+argc] = sp;
}
ustack[٣+argc] = ٠;

ustack[٠] = ٠xffffffff;  // fake return PC
ustack[١] = argc;
ustack[٢] = sp - (argc+١)*٤;  // argv pointer

sp -= (٣+argc+١) * ٤;
if(copyout(pgdir, sp, ustack, (٣+argc+١)*٤) < ٠)
  goto bad;

// Save program name for debugging.
for(last=s=path; *s; s++)
  if(*s == '/')
    last = s+١;
safestrcpy(curproc->name, last, sizeof(curproc->name));

// Commit to the user image.
oldpgdir = curproc->pgdir;
curproc->pgdir = pgdir;
curproc->sz = sz;
curproc->tf->eip = elf.entry;  // main
curproc->tf->esp = sp;
switchuvm(curproc);
freevm(oldpgdir);
init_num_syscalls();
return ٠;
```

پیاده‌سازی حافظه‌ی اشتراکی

ابتدا ساختار جدیدی به نام shared_page می‌سازیم که اطلاعات هر صفحه از حافظه‌ی اشتراکی را شامل می‌شود:

```
struct shared_page{
    int id;
    int num_of_refs;
    char* frame;
};
```

این ساختار شامل متغیرهای زیر است:

- id: شناسه‌ی آن صفحه
- num_of_refs: تعداد رفرنس‌هایی که به آن صفحه دسترسی دارند.
- frame: پوینتر به فریم فیزیکی شروع صفحه

سپس یک ساختار به نام shared_memory می‌سازیم که شامل یک آرایه از shared_page و یک قفل برای دسترسی به آن است:

```
struct{
    struct shared_page table[NUM_OF_SHARED_PAGES];
    struct spinlock lock;
} shared_memory;
```

تابعی برای init کردن این ساختار می‌نویسیم و آن را در pinit اجرا می‌کنیم:

```
void
init_shared_mem(){
    acquire(&shared_memory.lock);
    for (int i = ۰; i < NUM_OF_SHARED_PAGES; i++){
        shared_memory.table[i].num_of_refs = ۰;
    }
```

```
    release(&shared_memory.lock);
}
```

برای open_sharedmem ابتدا بررسی می‌کنیم که صفحه با id خواسته شده از قبل حافظه دارد یا خیر. در صورتی که حافظه موجود نبود، حافظه به آن اختصاص می‌دهیم و سپس با استفاده از mappages آن را به یک حافظه‌ی مجازی در پردازه متصل می‌کنیم. همچنین در ساختار proc متغیری اضافه می‌کنیم تا آدرس آن حافظه‌ی مجازی که به صفحه‌ی اشتراکی متصل شده را در آن ذخیره کنیم. در نهایت حافظه‌ی مجازی که به صفحه‌ی اشتراکی متصل شده را خروجی می‌دهیم.

```
char*
open_shared_mem(int id){
    struct proc* proc = myproc();
    pde_t *pgdir = proc->pgdir;
    acquire(&shared_memory.lock);
    int index = id;
    if (shared_memory.table[index].num_of_refs == ۰){
        shared_memory.table[index].frame = kalloc();
        memset(shared_memory.table[index].frame, ۰, PGSIZE);
    }
    char* start_mem = (char*)PGROUNDUP(proc->sz);
    //cprintf("start_mem: %d\n", start_mem);

    mappages(pgdir, start_mem, PGSIZE, V۲P(shared_memory.table[index].frame),
PTE_W|PTE_U);
    shared_memory.table[index].num_of_refs++;
    shared_memory.table[index].id = id;
    proc->shm = start_mem;

    release(&shared_memory.lock);
    return start_mem;
}
```

برای پیاده‌سازی close_sharedmem ابتدا حافظه‌ی مجازی که به صفحه‌ی اشتراکی متصل شده را از page table حذف می‌کنیم(زیرا در تابع wait حافظه‌های فیزیکی پردازه‌ی فرزند آزاد می‌شوند و اگر صفحه‌ی اشتراکی به پردازه متصل بماند، محتوای آن از بین می‌رود.) سپس بررسی می‌کنیم اگر تعداد رفرنس‌ها به صفحه‌ی اشتراکی صفر بشود، حافظه‌ی آن را آزاد می‌کنیم.

```
void
close_shared_mem(int id){
    struct proc* proc = myproc();
    pde_t *pgdir = proc->pgdir;
    acquire(&shared_memory.lock);
    int index = id;
    shared_memory.table[index].num_of_refs--;

    uint a = PGROUNDUP((uint)proc->shm);
    pte_t *pte = walkpgdir(pgdir, (char*)a, ٠);
    if(!pte)
    a = PGADDR(PDX(a) + ١, ٠, ٠) - PGSIZE;
    else if((*pte & PTE_P) != ٠){
        uint pa = PTE_ADDR(*pte);
        *pte = ٠;
    }

    if (shared_memory.table[index].num_of_refs == ٠){
        kfree(shared_memory.table[index].frame);

    }

    release(&shared_memory.lock);
}
```

در برنامه‌ی سطح کاربر ابتدا حافظه‌ی اشتراکی با شناسه‌ی ۱ را باز می‌کنیم و مقدار آن را ۰ قرار می‌دهیم. سپس ۱۰ فرزند می‌سازیم و هر فرزند این حافظه را باز می‌کنند و مقدار آن را یکی زیاد می‌کنیم. همچنین قبل از تغییر

دادن محتوای حافظه قفل سطح کاربری که در آزمایش قبل پیاده‌سازی کردیم را درخواست می‌کنیم تا مطمئن شویم که Race Condition رخ نمی‌دهد:

```
int main(int argc, char* argv[]){
    char* shared_mem = open_sharedmem(١);
    char* value = (char*) shared_mem;
    *value = ٠;
    for (int i = ٠; i < ١٠; i++){
        if (fork() == ٠){
            char* shared_mem = open_sharedmem(١);
            char* value = (char*) shared_mem;
            acquire_user_lock();
            *value += ١;
            printf(١, "Child: %d\n", *value);
            release_user_lock();
            close_sharedmem(١);
            exit();
        }
    }
    while (wait() != -١);

    printf(١, "Parent: %d\n", *value);

    close_sharedmem(١);

    exit();
}
```

در صورتی که این برنامه را اجرا کنیم مشاهده می‌کنیم که هر فرزند مقدار حافظه‌ی اشتراکی را یکی زیاد می‌کند و در نهایت مقدار حافظه‌ی اشتراکی که پردازه‌ی پدر چاپ می‌کند به تعداد پردازه‌های فرزند است:

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #9
1. Narges Haeri
2. Ali Hamzehpour
3. Mina Shirazi
$ test_shared_mem
Child: 1
Child: 2
Child: 3
Child: 4
Child: 5
Child: 6
Child: 7
Child: 8
Child: 9
Child: 10
Parent: 10
$
```