

آزمایشگاه سیستم عامل

تمرین کامپیوتری 5

اعضای گروه :

محمد سوری – 810100249

صفورا علوی پناه – 810100254

ریحانه حاجبی – 810100116

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

در لینوکس، نواحی حافظه مجازی (VMA) برای نمایش مناطق مختلف فضای آدرس مجازی یک پردازنده استفاده میشود که جزئیاتی چون حفاظت حافظه، تخصیص حافظه پویا و نقشه برداری حافظه را مدیریت می کند.

در سیستم عامل لینوکس از مفهوم VMA به صورت گسترده برای مدیریت حافظه مجازی پردازنده 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 table متناظر است، زمانی که یک پردازنده به یک آدرس مجازی دسترسی پیدا می کند entry های متناظر آدرس مجازی را به فیزیکی ترجمه می کنند. در xv6 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 .

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

در ساختار سلسله مراتبی، process ها و task ها به راحتی میتوانند با به اشتراک گذاشتن کدها و داده ها 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 بهره وری در استفاده از حافظه و کاهش زمان دسترسی به داده ها میشود. این ویژگی ها باعث بهبود کارایی و کاهش مصرف حافظه در سیستم می شوند.

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

- Directory Page : برای اشاره به سطح بعدی از 20 بیت استفاده میشود. این به این معناست که هر دفترچه (directory page) میتواند به حداکثر 20^2 صفحه دیگر اشاره کند 12 بیت نیز برای نگهداری اطلاعات مربوط به سطح دسترسی (access level) استفاده می شود.

- Table Page : اینجا نیز برای اشاره به صفحه فیزیکی (physical page) از 20 بیت استفاده میشود. این به این معناست که هر صفحه جدول (table page) میتواند به حداکثر 20^2 صفحه داده اشاره کند. بر خلاف directory page ، اینجا اطلاعات مربوط به سطح دسترسی از اهمیت کمتری برخوردار هستند. چرا که جدول صفحات فقط یک لینک به صفحه فیزیکی دارد و نه خود داده ها را در خود ذخیره می کند.

- بیت (Dirty Bit) : در directory page ، بیت D به معنای آن است که اگر یک صفحه تغییر کرده باشد، باید تغییرات آن را در دیسک ذخیره کنیم (نوشته شود). در جدول صفحات، این بیت معنای خاصی ندارد. چرا که جدول صفحات فقط به عنوان یک لینک برای موقعیت فیزیکی داده ها عمل می کند و خود داده ها را در خود نگهداری نمی کند.

کد مربوط به ایجاد فضاهای آدرس در xv6

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

تابع kalloc به صورت زیر تعریف شده است:

```
// 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;
}
```

در سیستم عامل xv6 ، این تخصیص حافظه به صورت فیزیکی انجام می شود. در واقع kalloc در xv6 صفحات حافظه ای فیزیکی به طول 4096 بایت را اختصاص می دهد، که مستقیماً در حافظه فیزیکی سیستم قرار دارد. این تابع برای تخصیص حافظه در kernel heap برای ذخیره سازی ساختمان های پویا استفاده می شود. 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 برمی گرداند.

5. 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 - 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;
}
```

این تابع به منظور ساخت نگاشت از آدرس مجازی به فیزیکی استفاده میشود.
در اینجا چند فلگ به شرح زیر تعریف شده اند:

- PTE_P: نشان دهنده حاضر بودن صفحه (Present) در حافظه.
- PTE_W: نشان دهنده امکان نوشتن (Writeable) در صفحه.
- PTE_U: نشان دهنده امکان دسترسی توسط کاربر (User) به صفحه.
- PTE_PS: نشان دهنده 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 صفحه بندی شده است تبدیل می کند. سپس با استفاده از حلقه، از تابع walkpgdir برای پیدا کردن یا ایجاد PTE مربوط به هر آدرس تعریف شده باشد (با بررسی بیت PTE_P)، با یک panic به خطا می افتد؛ در غیر این صورت، یک PTE جدید با مشخصات مربوط به آدرس فیزیکی (pa)، مجوزهای دسترسی (perm) و بیت PTE_P (نشان دهنده فعال بودن PTE) ایجاد می شود. حلقه تا زمانی ادامه پیدا میکند که به آخرین آدرس مجازی (last) برسد. در نهایت، تابع باز می گردد و 0 را ارجاع میدهد تا نشان دهد که عملیات موفقیت آمیز بوده است، مگر اینکه در طول اجرا با مشکلی مواجه شود که در آن صورت 1- برگردانده میشود.

7. راجع به تابع `walkpgdir()` توضیح دهید. این تابع چه عمل سخت افزاری را شبیه سازی 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 فیزیکی اش تبدیل می کند. در صورتی که PTE مورد نظر وجود نداشته باشد و پارامتر `alloc` غیر صفر باشد نیز یک PTE برای آن آدرس می سازد. تعریف و پیاده سازی این تابع به شکل زیر است:

```
pte_t *
walkpgdir(pte_t *pgdir, const void *va, int alloc)
{
    pte_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 `mappages` و `allocvm` که در ارتباط با حافظه ی مجازی هستند را توضیح دهید.

تابع `mappages` همانطور که به تفصیل در سوال 5 توضیح داده شد یک حافظه ی مجازی را به حافظه ی فیزیکی متصل میکند. تابع `allocvm` ناحیه ی حافظه ی مجازی یک پردازنده را از `oldsz` به `newsz` افزایش میدهد. این تابع تا زمانی که اندازه ی حافظه ی اولیه به مقدار خواسته شده برسد، حافظه ی فیزیکی `allocate` میکند و آن را با استفاده از تابع `mappages` به یک حافظه ی مجازی در پردازنده متصل میکند. پیاده سازی 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 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;
}

```

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

تابع **exec** ابتدا فایل محتوای برنامه ای که باید بارگذاری شود را باز می کند.

```

int
exec(char *path, char **argv)
{
    char *s, *last;
    int i, off;
    uint argc, sz, sp, ustack[3+MAXARG+1];
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pde_t *pgdir, *oldpgdir;
    struct proc *curproc = myproc();
    begin_op();
    if((ip = namei(path)) == 0){
        end_op();
        cprintf("exec: fail\n");
        return -1;
    }
    ilock(ip);
    pgdir = 0;
}

```

سپس هدر های آن فایل چک میشود و بعد با صدا زده شدن تابع `setupkvm` بخش هسته ی `page table` برای برنامه ی جدید ساخته میشود.

```

// Check ELF header
if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
if(elf.magic != ELF_MAGIC)
    goto bad;
if((pgdir = setupkvm()) == 0)
    goto bad;

```

سپس در یک حلقه محتوای برنامه در حافظه ی پردازنده ذخیره میشود. این حلقه هربار قسمتی از فایلی که برنامه در آن قرار دارد را میخواند و با استفاده از تابع `allocuvvm` حافظه ی پردازنده را زیاد میکند تا بتواند قسمتی را که خوانده است در حافظه ی پردازنده ذخیره کند که با استفاده از تابع `loaduvvm` این کار را انجام می دهد.

```

// Load program into memory.
sz = 0;
for(i=0, 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)) == 0)
        goto bad;
    if(ph.vaddr % PGSIZE != 0)
        goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
        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 می شود. پارامترهای ورودی (**args**) مربوط به برنامه نیز در همین استک ذخیره میشوند. در نهایت **page table** قبلی آن پردازنده 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 = allocvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp = (sp - (strlen(argv[argc]) + 1) & ~3);
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
        goto bad;
    ustack[3+argc] = sp;
}
ustack[3+argc] = 0;
ustack[0] = 0xffffffff; // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4; // argv pointer
sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;
// Save program name for debugging.
for(last=s=path; *s; s++)
    if(*s == '/')
        last = s+1;
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;
switchvm(curproc);
freevm(oldpgdir);
init_num_syscalls();
return 0;

```

نمایش اطلاعات درباره ی حافظه:

در این بخش چند سیستم کال پیاده سازی میشود که در ادامه توضیح میدهیم:
printvir: باید تعداد صفحات مجازی در بخش کاربری فضای آدرس پردازش را برگرداند .

```
94
95 void
96 printvir(void) {
97     struct proc *p = myproc();
98     pde_t *pgdir = p->pgdir;
99     int virtual_pages = 0;
100
101     for (int i = 0; i < NPDETRIES; i++) {
102         if (!(pgdir[i] & PTE_U))
103             break;;
104
105         pde_t *pde = &pgdir[i];
106         if (*pde & PTE_P) {
107
108             pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
109             for (int j = 0; j < NPTETRIES; j++) {
110                 if (pgtab[j] & PTE_U) {
111                     virtual_pages++;
112                 }
113             }
114         }
115     }
116
117     cprintf("Number of virtual pages: %d\n", virtual_pages);
118 }
119
```

printphy: تعداد صفحات فیزیکی در بخش کاربری فضای آدرس پردازش را برمیگرداند .

```
void printphy(void)
{
    struct proc *p = myproc();
    pde_t *pgdir = p->pgdir;
    int physical_pages = 0;

    for (int i = 0; i < NPDETRIES; i++) {
        if (!((pgdir[i] & PTE_U) && (pgdir[i] & PTE_P)))
            break;

        pde_t *pde = &pgdir[i];
        pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));

        for (int j = 0; j < NPTETRIES; j++) {
            if ((pgtab[j] & PTE_U) && (pgtab[j] & PTE_P)){
                physical_pages++;
            }
        }
    }

    cprintf("Number of physical pages: %d\n", physical_pages);
}
```

:mapex

```
void*
mapex(int size)
{
    char *a, *last;
    pte_t *pte;
    void* va = myproc()->sz;

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

چک کردن آرگومان های mapex:

```
int
sys_mapex(void)
{
    int size;

    if (argint(0, &size) < 0)
        return 0;

    if (size <= 0 || size % PGSIZE != 0)
        return 0;

    uint s_z = myproc()->sz;

    mapex(size);

    return s_z;
}
```

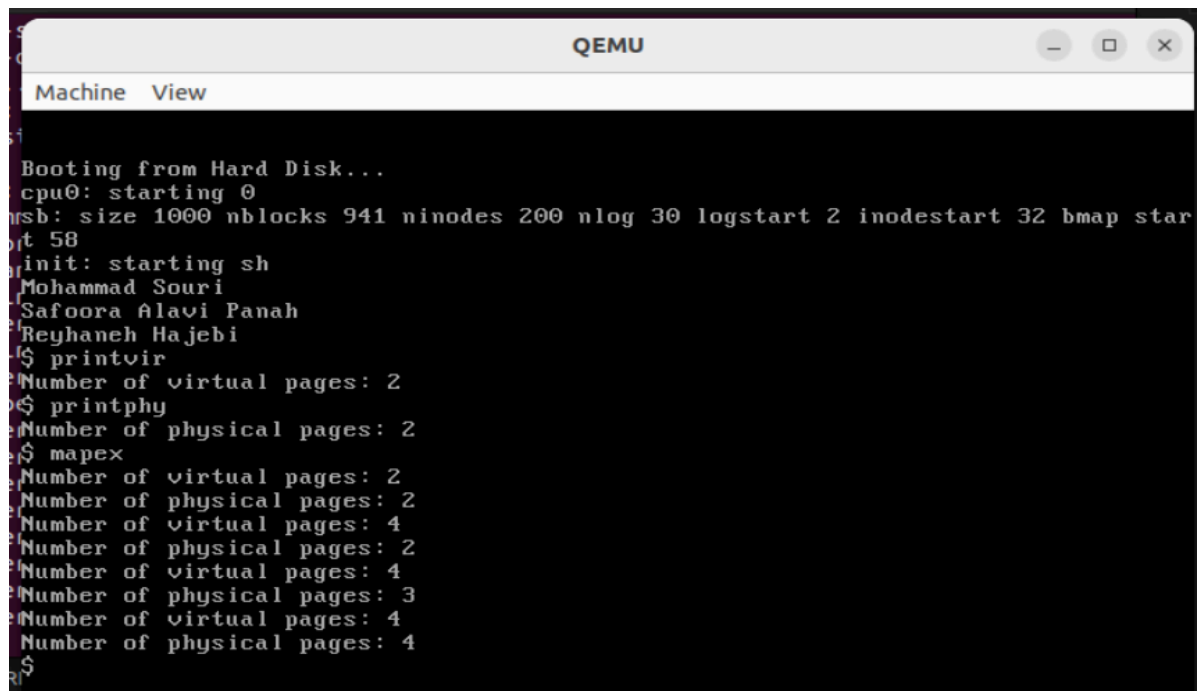
این ها در فایل vm.c که مربوط به حافظه است پیاده سازی شده اند. همچنین برای هندل کردن page : fault

```

case T_PGFLT:
    allocuvm(myproc()->pgdir, PGROUNDDOWN(rcr2()), PGROUNDDOWN(rcr2())+PGSIZE);
    switchuvm(myproc());
    break;

```

خروجی:



```

QEMU
Machine View
Booting from Hard Disk...
cpu0: starting 0
rsb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Mohammad Sour
Safoora Alavi Panah
Reyhaneh Hajebi
$ printv
Number of virtual pages: 2
$ printphy
Number of physical pages: 2
$ mapex
Number of virtual pages: 2
Number of physical pages: 2
Number of virtual pages: 4
Number of physical pages: 2
Number of virtual pages: 4
Number of physical pages: 3
Number of virtual pages: 4
Number of physical pages: 4
$

```

Github : [link](#)