

به نام خدا

گزارش پروژه «۵» آزمایشگاه سیستم عامل

استاد: دکتر کارگهی

گروه ۱۱

امیرارسلان شهبازی ۸۱۰۱۰۱۴۵۱

سید محمدحسین مظهری ۸۱۰۱۰۱۵۲۰

محمد مهدی صمدی ۸۱۰۱۰۱۴۶۵

لینک مخزن https://github.com/AMIRSH1383/OS-SMS_LAB5.git

مقدمه

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

در لینوکس، هسته از نواحی حافظه مجازی با پیگیری memory mapping های پردازش استفاده می کند. برای مثال یک پردازش یک vma برای کد، یک vma برای هر نوع دیتا و یک vma برای هر memory mapping دارد. هر vma شامل تعدادی پیچ است که هر کدام از این پیچ ها یک ورودی به page table دارد اما xv6 از آدرس های مجازی ۳۲ بیتی استفاده می کند که فضای آدرسی مجازی ۴ گیگابایتی ایجاد می کند. همچنین xv6 از جدول دو سطحی استفاده می کند که مفهومی از حافظه مجازی ندارد.

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

در ساختار سلسله مراتبی، پردازش ها و تسک ها به راحتی می توانند با به اشتراک گذاشتن کدها و داده ها توسط mapping بخش مناسب به صفحات فیزیکی از مصرف اضافی حافظه جلوگیری کنند.

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

در مدخل سطح page directory برای اشاره به سطح بعدی از ۲۰ بیت استفاده شده است. همچنین ۱۲ بیت برای سطح دسترسی نگهداری می شود.

این ۱۲ بیت در هر دو سطح وجود دارد اما در سطح page table از ۲۰ بیت برای آدرس صفحه فیزیکی استفاده می شود.

در بیت D(Dirty) با هم تفاوت دارند.

در page directory این بیت به این معناست که صفحه باید در دیسک نوشته شود تا تغییرات اعمال شود. اما در page table این بیت معنایی ندارد.

(۴) تابع kalloc چه نوع حافظه ای اختصاص می دهد؟ (فیزیکی یا مجازی)

این تابع یک حافظه فیزیکی به فضای ۴۰۹۶ بایتی اختصاص می دهد. پوینتری را return می کند که کرنل می تواند از آن استفاده کند و در صورتی که نتواند حافظه ای تخصیص دهد 0 را return می کند.

(۵) تابع mappages چه کاربردی دارد؟

این تابع آدرس `directory page` و آدرس یک خانه حافظه مجازی و آدرس یک خانه حافظه فیزیکی و سائز را می‌گیرد و صفحه موجود در حافظه فیزیکی را با توجه به آدرس و سائزی که به آن داده ایم در آدرسی که در حافظه مجازی به آن داده ایم بارگذاری می‌کند. این کار برای دسترسی به متغیرهای پردازش در حال اجراست تا صفحه آن بتواند به درستی بارگذاری شود و تغییر داده شود. همچنین این تابع صفحه جدید را به `pgdir` اضافه می‌کند و کلا حافظه مجازی را به فیزیکی متصل می‌کند.

(۶) این سوال وجود نداشت و شماره گذاری سوالات اشتباه شده بود.

(۷) راجع به تابع `walkpgdir` توضیح دهید. این تابع چه عمل سخت افزاری را شبیه سازی می‌کند؟

تابع فوق آدرس `directorypage` و همچنین خانه ای از حافظه مجازی را گرفته و آدرس `pagetable` ای که در حافظه مجازی داریم را از `directory page` برمی‌گرداند و در صورت نیاز جدول مورد نظر را می‌سازد.

این تابع عمل سخت افزاری ترجمه آدرس مجازی به فیزیکی را شبیه سازی می‌کند.

(۸) توابع `allocvm` و `mappages` که در ارتباط با حافظه 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 منظر و `permission flags` 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 می کند.

`allocvm`: 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 ۱: اگر تابع `kalloc` شکست بخورد. این تابع مسئول برگرداندن آدرس یک صفحه جدید و در حال حاضر استفاده نشده در رم است. اگر ۰ برگرداند یعنی در حال حاضر صفحه استفاده نشده ای وجود ندارد.

حالت ۲: 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 فرایند 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 احتمالاً به این معنی است که دایرکتوری صفحه در حال حاضر پر است.

در هر دو حالت `allocvm` 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.

۹) شیوه بارگذاری برنامه در حافظه توسط فراخوانی سیستمی **exec** را شرح دهید.

تابع **exec** برای جایگزین کردن فرآیند در حال اجرا با یک فرآیند جدید استفاده می شود:

مراحل:

پاک کردن حالت فعلی حافظه: این تابع 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: سپس این فایل را در حافظه برنامه کپی می کند.

مقدار دهی وضعیت اولیه وضعیت رجیسترها از جمله PC که به دستور بعدی اشاره می کند.

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

شناسه فرآیند یا **pid** همان قبلی است: از آنجا که فرآیند جدید ایجاد نمی شود، شناسه فرآیند **pid** تغییر نمی کند. تابع **exec** مگر در صورت خطا ریترن نمی کند. این به این دلیل است که فرآیند در حال اجرا کاملاً توسط فرآیند جدید جایگزین می شود.

شرح پروژه

توضیحات مربوط به پیاده سازی **close shared memory** و **open shared memory**

Make File

```
259 EXTRA=\
260 mkfs.c ulib.c user.h cat.c echo.c decode.o encode.o move_file.o history.o create_palindrome.o sort_syscalls.o get_most_invoked_syscall.o list_all_processes.o forkttest.c grep.c kill.c\
261 ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c sham_test.c\
262 printf.c umalloc.c\
263 README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
264 .gdbinit.tmpl gdbutil\
265
```

```

168     UPROGS=\
169         _cat\
170         _decode\
171         _echo\
172         _encode\
173         _history\
174         _forktest\
175         _gdb_test\
176         _grep\
177         _init\
178         _kill\
179         _ln\
180         _ls\
181         _mkdir\
182         _move_file\
183         _rm\
184         _sh\
185         _stressfs\
186         _usertests\
187         _wc\
188         _zombie\
189         _create_palindrome\
190         _list_all_processes\
191         _sort_syscalls\
192         _get_most_invoked_syscall\
193         _sham_test\

```

Defs.h

```

180     // vm.c
181     void      seginit(void);
182     void      kvmalloc(void);
183     pde_t*    setupkvm(void);
184     char*     uva2ka(pde_t*, char*);
185     int       allocvm(pde_t*, uint, uint);
186     int       deallocvm(pde_t*, uint, uint);
187     void      freevm(pde_t*);
188     void      initvm(pde_t*, char*, uint);
189     int       loadvm(pde_t*, char*, struct inode*, uint, uint);
190     pde_t*    copyvm(pde_t*, uint);
191     void      switchvm(struct proc*);
192     void      switchkvm(void);
193     int       copyout(pde_t*, uint, void*, uint);
194     void      clearpteu(pde_t *pgdir, char *uva);
195     char*     open_sharedmem(int id);
196     int       close_sharedmem(int id);

```

Proc.h

```
43 // Per-process state
44 ✓ struct proc {
45     uint sz;                // Size of process memory (bytes)
46     pde_t* pgdir;           // Page table
47     char *kstack;           // Bottom of kernel stack for this process
48     enum procstate state;    // Process state
49     int pid;                 // Process ID
50     struct proc *parent;     // Parent process
51     struct trapframe *tf;    // Trap frame for current syscall
52     struct context *context; // swtch() here to run process
53     void *chan;              // If non-zero, sleeping on chan
54     int killed;              // If non-zero, have been killed
55     struct file *ofile[NOFILE]; // Open files
56     struct inode *cwd;        // Current directory
57     char name[16];           // Process name (debugging)
58     struct syscall_info syscalls[1000]; // List of system calls used in process
59     int syscall_count;        // Number of system calls used in process
60     uint shmemaddr;           // address of shared-memory
61 };
```

Syscall.c

```
142 extern int sys_open_sharedmem(void);
143 extern int sys_close_sharedmem(void);
144
```

```
172 [SYS_open_sharedmem] sys_open_sharedmem,
173 [SYS_close_sharedmem] sys_close_sharedmem,
174 };
```

Syscall.h

```
28 #define SYS_open_sharedmem 27
29 #define SYS_close_sharedmem 28
```

Sysproc.c

```

156 char* sys_open_sharedmem(void) {
157     int id;
158     if (argint(0, &id) < 0)
159         return (char*)(-1); // error but we don't have equivalent in char*
160
161     return open_sharedmem(id);
162 }
163
164 int sys_close_sharedmem(void) {
165     int id;
166     if (argint(0, &id) < 0)
167         return -1;
168
169     return close_sharedmem(id);
170 }

```

User.h

```

32 char* open_sharedmem(int);
33 int close_sharedmem(int);
34

```

U.sys.s

```

37 SYSCALL(open_sharedmem)
38 SYSCALL(close_sharedmem)

```

Vm.c

```

397 // shared memory
398 #define NSHPAGE 64
399 #define HEAPLIMIT 0x7F000000 // 2GB - 128MB
400 struct shpage {
401     int id;
402     int n_access;
403     uint physicalAddr;
404 };
405
406 struct shmemtable {
407     struct shpage pages[NSHPAGE];
408     struct spinlock lock;
409 } shmemtable;
410

```

برای پیاده سازی حافظه اشتراکی ۲ استراکت بالا را تعریف می کنیم .

یک استراکت برای نگهداری shared page که شامل شماره صفحه و تعداد دسترسی ها و آدرس فیزیکی صفحه می باشد.
و استراکتی دیگر جهت نگهداری shared memory table که شامل تعدادی صفحه و قفلی جهت نگهداری و دسترسی به table است .

Open shared memory

```
411 char* open_sharedmem(int id) {
412     struct proc* proc = myproc();
413     acquire(&shmemtable.lock);
414     int size = PGSIZE;
415
416     for (int i = 0; i < NSHPAGE; i++) {
417         if (shmemtable.pages[i].id == id) {
418             shmemtable.pages[i].n_access++;
419             char* vaddr = (char*)PGROUNDUP(proc->sz);
420             if (mappages(proc->pgdir, vaddr, PGSIZE, shmemtable.pages[i].physicalAddr, PTE_W | PTE_U) < 0)
421                 return (char*)(-1);
422             proc->shmemaddr = (uint)vaddr;
423             proc->sz += size;
424             release(&shmemtable.lock);
425             return vaddr;
426         }
427     }
428
429     int pgidx = -1;
430     for (int i = 0; i < NSHPAGE; i++) {
431         if (shmemtable.pages[i].id == 0) {
432             shmemtable.pages[i].id = id;
433             pgidx = i;
434             break;
435         }
436     }
```

```

437
438     if (pgidx == -1) {
439         cprintf("shared memory: pages are full\n");
440         release(&shmemtable.lock);
441         return (char*)(-1);
442     }
443
444     char* paddr = kalloc();
445     if (paddr == 0) {
446         cprintf("shared memory: out of memory\n");
447         release(&shmemtable.lock);
448         return (char*)(-1);
449     }
450
451     memset(paddr, 0, PGSIZE);
452     char* vaddr = (char*)PGROUNDUP(proc->sz);
453     shmemtable.pages[pgidx].physicalAddr = (uint)V2P(paddr);
454
455     if (mappages(proc->pgdir, vaddr, PGSIZE, shmemtable.pages[pgidx].physicalAddr, PTE_W | PTE_U) < 0)
456         return (char*)(-1);
457
458     shmemtable.pages[pgidx].n_access++;
459     proc->shmemaddr = (uint)vaddr;
460     proc->sz += size;
461
462     release(&shmemtable.lock);
463     return vaddr;
464 }

```

این تابع برای باز کردن یک بخش از حافظه مشترک است و دو حالت اصلی دارد:

(۱) اگر حافظه مشترک با شناسه مشخص (**id**) وجود داشته باشد:

- حافظه پیدا شده و به فضای آدرس مجازی فرآیند فعلی متصل می‌شود.
- تعداد دسترسی‌ها (**n_access**) افزایش می‌یابد.

(۲) اگر حافظه مشترک وجود نداشته باشد:

- یک صفحه جدید از حافظه فیزیکی با استفاده از **kalloc** تخصیص داده می‌شود.
- حافظه به صفر مقداردهی شده و به فرآیند متصل می‌شود.
- جدول حافظه مشترک با اطلاعات صفحه جدید به‌روزرسانی می‌شود.

Close shared memory


```

466     int close_sharedmem(int id) {
467         struct proc* proc = myproc();
468         acquire(&shmemtable.lock);
469
470         for (int i = 0; i < NSHPPAGE; i++) {
471             if (shmemtable.pages[i].id == id) {
472                 shmemtable.pages[i].n_access--;
473
474                 uint a = (uint)PGROUNDUP(proc->shmemaddr);
475                 pte_t* pte = walkpgdir(proc->pgdir, (char*)a, 0);
476                 *pte = 0;
477
478                 if (shmemtable.pages[i].n_access == 0)
479                     shmemtable.pages[i].id = 0;
480
481                 release(&shmemtable.lock);
482                 return 0;
483             }
484         }
485
486         release(&shmemtable.lock);
487         cprintf("No shared memory with this ID.\n");
488         return -1;
489     }

```

این تابع برای بستن یک بخش از حافظه مشترک است:

- ابتدا صفحه مرتبط با شناسه داده شده (id) را پیدا می‌کند.
 - تعداد دسترسی‌ها (n_access) کاهش می‌یابد.
- اگر هیچ فرآیند دیگری از حافظه استفاده نکند (n_access == 0)، شناسه حافظه پاک می‌شود و صفحه از جدول آزاد می‌شود.

برنامه آزمون

در بخش آخر ما فایلی برای تست کردن سیستم کال‌هایی که نوشتیم می‌نویسیم. فایل sham_test.c

کد این فایل به صورت زیر است.

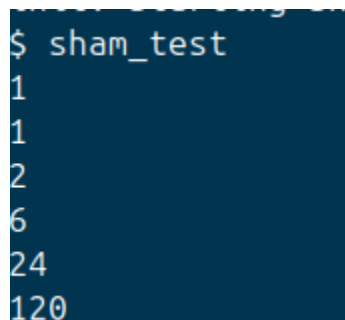
```

4
5     #define NPROCESS 5
6     #define ID      30
7
8     void test_sharedmem(void) {
9         char* adr = open_sharedmem(ID);
10        adr[0] = 1;
11        printf(1, "%d\n", adr[0]);
12
13        for (int i = 0; i < NPROCESS; i++) {
14            if (fork() == 0) {
15                sleep(100 * i);
16                char* adrs = open_sharedmem(ID);
17                adrs[0] *= (i+1);
18                printf(1, "%d\n", adrs[0]);
19                close_sharedmem(ID);
20                exit();
21            }
22        }
23        for (int i = 0; i < NPROCESS; i++)
24            wait();
25        close_sharedmem(ID);
26    }
27
28    int main(int argc, char* argv[]) {
29        test_sharedmem();
30        exit();
31    }

```

این فایل به این صورت است که یک متغیر مشترک در نظر می گیریم. سپس آن را با مقدار یک مقدار اولیه می دهیم. با توجه به اینکه فاکتوریل از ما خواسته شده بود، هرپدازه ای که می آید بسته به ایندکسی که دارد اجرا می شود(مثلا اینجا ۵ تا پردازش داریم و ۵ ایندکس) عددی را در مقدار مشترک قبلی ضرب کرده و همانجا ذخیره می کند. در هرگام نتیجه متغیر مشترک را چاپ می کنیم و به نتیجه زیر می رسم.

همانطور که مشاهده می کنید نتیجه فاکتوریل ما درست است.



```

$ sham_test
1
1
2
6
24
120

```