گزارش پروژه 5 آزمایشگاه سیستم عامل

مدیریت حافظه در XV6

اعضای گروه 18:
پریا پاسه‌ورز - 810101393
کوثر شیری جعفرزاده - 810101456
پریسا یحیی‌پور فتیده - 810101551
لینک مخزن - هش آخرین کامیت: 9f35f84adf1b30e90c9a15ff9edc3803549a23b1

مقدمه

سوال 1

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

هر فرایند در سیستم‌عامل لینوکس فضای آدرس مجازی مختص به خود دارد که از طریق جدول صفحات به حافظه فیزیکی نگاشت می‌شود.
شامل نواحی مختلفی است، مانند: VMA

- متن برنامه: حاوی کد اجرایی
- داده‌ها: شامل متغیرهای استاتیک و داده‌های تخصیص‌یافته در زمان اجرا
- Stack: برای ذخیره‌سازی متغیرهای محلی و بازگشت توابع
- Heap: برای تخصیص پویا
- Mapped Regions: مانند فایل‌های نگاشت‌شده به حافظه

این نواحی در ساختاری مانند mm_struct و vm_area_struct مدیریت می‌شوند.
مدیریت حافظه xv6 بسیار ساده‌تر است و پیچیدگی‌های VMA را ندارد.
فضای آدرس یک فرآیند شامل بخش‌های زیر است:

- کد (متن برنامه)
- داده (متغیرها و داده‌های استاتیک)

- پشته

مدیریت حافظه در xv6 به صورت خطی و ساده انجام می‌شود و از ساختارهای پیچیده مانند VMA استفاده نمی‌کند. این سیستم از جدول دو سطحی استفاده می‌کند و مفهومی از حافظه مجازی ندارد، بلکه از آدرس‌های مجازی 32 بایتی استفاده می‌کند که فضای حافظه 3 گیگابایتی را تشکیل می‌دهند.

سوال 2

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

در یک کامپیوتر 32 بیتی با اندازه هر صفحه برابر KB4، اگر از ساختار سلسله‌مراتبی استفاده نکنیم، برای مدیریت حافظه نیاز به یک جدول صفحه بسیار بزرگ داریم. این جدول باید حدود 1 میلیون (20^2) مدخل داشته باشد و چون هر مدخل 32 بیت است، کل جدول حدود 4MB حافظه مصرف می‌کند.
اما اگر از ساختار سلسله‌مراتبی استفاده کنیم، جدول‌ها به دو بخش تقسیم می‌شوند: page table & page directory. در این حالت، هر کدام از این جدول‌ها فقط4KB حافظه نیاز دارند. به این ترتیب، به جای نگهداری کل جدول صفحه برای هر پردازه، فقط جدول page directory ذخیره می‌شود و بخش‌های لازم از page table ها در صورت نیاز ساخته می‌شوند. این روش مصرف حافظه را به شدت کاهش می‌دهد.

سوال 3

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

بیت ها	توضیح
0-11	Physical Frame Address: آدرس پایه صفحه در حافظه فیزیکی
12	"Present" (وجود): نشان می‌دهد که آیا صفحه در حافظه فیزیکی موجود است یا خیر.
13	بیت "Read/Write" (خواندن/نوشتن): مشخص می‌کند که صفحه فقط قابل خواندن است یا قابلیت نوشتن هم دارد.
14	بیت "User/Supervisor" (حالت کاربر/کرنل): تعیین می‌کند که دسترسی به صفحه فقط برای کرنل مجاز است یا کاربران عادی هم می‌توانند به آن دسترسی داشته باشند.

15	بیت "Page Write Through": مشخص می‌کند که عملیات نوشتن به کش و حافظه به صورت همزمان انجام شود یا نه.
16	بیت "Page Cache Disable": اگر تنظیم شود، کش کردن این صفحه غیر فعال می‌شود.
17	بیت "Accessed" (دسترسی): نشان می‌دهد که آیا صفحه اخیراً دسترسی داشته شده است یا خیر.
18	بیت "Dirty" (تغییر یافته): نشان می‌دهد که آیا صفحه تغییر داده شده است (برای نوشتن در حافظه فیزیکی نیاز است).
19	بیت "Page Attribute Table" یا PAT: برای تنظیمات خاص کش استفاده می‌شود.
20-31	باقی‌مانده آدرس صفحه یا بیت‌های رزرو شده برای ویژگی‌های دیگر.

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

سوال 4

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

با توجه به کامنت های ابتدایی فایل kalloc.c این تابع برای اختصاص دادن حافظه های فیزیکی است که می تواند به ساختار های هسته همانند استک، pipe buffer، page table و پردازه های کاربر حافظه فیزیکی اختصاص دهد.

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

در کد مربوط به این بخش ابتدا قفل حافظه خالی را می گیرد و سپس اولین فضای خالی موجود در حافظه را می گیرد و آن را آپدیت می کند و در نهایت این پوینتر به حافظه را بر میگرداند.

این پوینتر به حافظه خالی در استراکت kmem ذخیره می شود. لازم به ذکر است طبق کامنت بالا در xv6 حافظه به پیج های 4096 بایتی تقسیم بندی شده اند .

```
// 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 هنگام شروع تابع main ابتدا با فراخوانی دستور زیر سیستم عامل، حافظه بعد کرنل را تا انتها آزاد می کند و آن را درون kmem ذخیره می کند.

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
```

```
  mpmain();        // finish this processor's setup
}
```

سوال 5

تابع mappages چه کاربردی دارد؟

این تابع به منظور ساخت نگاشت از آدرس مجازی به فیزیکی استفاده می شود و در فایل vm.c قرار دارد.
این تابع ابتدا یک page table entry می سازد و برای این کار از تابع walkpgdir استفاده می کند.
تابع walkpgdir به این صورت کار می کند که آدرس page table entry که به آدرس مجازی va در pagdir نگاشت شده را برمی گرداند و اگر همچین نگاشتی وجود نداشته باشد page table یک پیج می سازد.

```
// 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;
}
```

از این تابع در توابع زیر استفاده شده است:

```
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

```
// Load the initcode into address 0 of pgdir.
// sz must be less than a page.
void
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```

```
// 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;
}
```

```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
```

```
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      kfree(mem);
      goto bad;
    }
  }
  return d;

bad:
  freevm(d);
  return 0;
}
```

فلگ هایی که در این تابع استفاده می شوند در فایل mmu.h قرار دارند که عبارت است از :

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size
```

سوال 7

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

تابع walkpgdir به منظور نگاشت آدرس مجازی به آدرس فیزیکی استفاده می شود. این تابع عمل سخت افزاری ترجمه آدرس مجازی به فیزیکی را شبیه سازی می کند.
اگر PTE در pgdir وجود داشت که به آدرس مجازی با شروع از va اشاره داشت آن را برمی گرداند و اگر وجود نداشت جدولی ساخته و آدرس آن را برمیگرداند.

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

سوال 8

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

تابع allocuvm

این تابع در فایل vm.c قرار دارد که کد آن در ادامه آورده شده است:

```
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;
}
```

مسئولیت این تابع این است که صفحه های جدیدی از حافظه را به بخش سمت کاربر پردازه اختصاص دهد. این تابع با ایجاد یک نگاشت بین حافظه فیزیکی و آدرس های مجازی، فضای آدرس دهی مجازی پردازه را گسترش می دهد. این تابع سه ورودی دارد:

- pde_t *pgdir: مربوط به directory صفحه های پردازه است.
- uint oldsz: اندازه فعلی فضای آدرس دهی مجازی پردازه به بایت
- uint newsz: اندازه فضای آدرس دهی مجازی جدید پردازه که باید از oldsz بزرگتر باشد

اگر مشکلی در این تابع رخ دهد 0 برمی گرداند و در صورت موفقیت آمیز بودن اندازه جدید یا newsz را بر می گرداند.

این تابع معمولا در حالات زیر استفاده می شود:

- هنگام مقدار دهی اولیه پردازه ها تا به آنها فضای آدرس بدهیم
- زمانی که هیپ یا استک پردازه قرار است گسترش یابد
- مدیریت حافظه هنگام استفاده از exec یا اختصاص حافظه به شکل پویا

این تابع در ابتدا درست بودن مقادیر ورودی را بررسی می کند. ابتدا مطمئن می شود که پردازه از فضای آدرس دهی هسته درخواست حافظه نکند. سپس مقدار اندازه جدید را با اندازه قدیمی مقایسه می کند و در صورتی که اندازه جدید از اندازه قبلی کوچکتر باشد خطا می دهد. حلقه موجود در این تابع، در فضای آدرس مجازی که [oldsz,newsz) است می چرخد و به ازای هر صفحه، فضای فیزیکی حافظه می گیرد و نگاشت را انجام می دهد. در صورتی که kalloc موفقیت آمیز نباشد یعنی اختصاص حافظه فیزیکی موفقیت آمیز نبوده در این صورت با استفاده از deallocuvm هر صفحه ای که تا اینجا گرفته بوده را پاک می کند.

تابع mappages

این تابع هم در فایل vm.c قرار دارد که کد آن در ادامه آورده شده است:

```
 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;
}
```

مسئولیت این تابع این است که یک بازه ای از آدرس های مجازی را به آدرس فیزیکی در page table پردازه نگاشت دهد. این تابع نقش مهمی در مدیریت فضای مجازی دارد چرا که ترجمه آدرس مجازی به آدرس فیزیکی مربوط به این تابع است. این تابع سه ورودی دارد:

- pde_t *pgdir: مانند تابع قبلی پوینتر به دایرکتوری صفحه های پردازه است
- void *va: شروع آدرس مجازی برای نگاشت
- uint size: اندازه حافظه که باید نگاشت شود (به بایت)
- uint pa: شروع آدرس فیزیکی برای نگاشت
- int perm: اجازه یا permission مربوط به page table

در صورتی که عملیات موفقیت آمیز باشد 0 برمی گرداند و در غیر این صورت -1 که یعنی خطایی رخ داده است و نمی تواند یک مدخل از page table را اختصاص دهد. این تابع ابتدا شروع و پایان آدرس مجازی که باید نگاشت شود را نزدیک ترین lower page boundary تبدیل می کند تا مطمئن شود بازه مدنظر صفحه به صفحه بررسی می شود. سپس روی صفحات می چرخد و ابتدا برای آن را PTE پیدا می کند. در صورتی که PTE برای این آدرس از قبل وجود داشته باشد panic می کند چون به مشکل double mapping خورده است. سپس مقدار های لازم را در PTE قرار می دهد که شامل آدرس فیزیکی، مجوز ها و یک فلگ برای نشان دادن حضور داشتن page است. در صورتی که به آخرین صفحه نرسیده باشد ادامه می دهد.

سوال 9

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

همان طور که می دانیم فراخوانی سیستمی exec تصویر پردازه از حافظه را به طور کامل جایگزین می کند. گام هایی که باید برای این هدف طی شوند به شکل زیر هستند:

1. فایل اجرایی را parse می کنیم
2. حافظه اختصاص می دهیم و نگاشت را انجام می دهیم
3. کد را کپی می کنیم
4. استک را راه اندازی می کنیم
5. تعویض متن انجام می دهیم

کد این فراخوانی سیستمی در exec.c قرار دارد که بخش مربوط به کپی کردن کد در پایین آورده شده است:

```
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;
  }
```

این بخش از فایل مربوط به کپی کردن کد در حافظه است که از فایلی که آدرس آن از طریق آرگومان به تابع exec پاس داده شده خوانده می شود.

شرح پروژه

برای پیاده سازی صفحه اشتراکی ابتدا یک استراکت برای هر صفحه تعریف می کنیم تا بتوانیم داده های مربوط به هر صفحه را در آن نگهداری کنیم:

```
#define MAX_SHARED_MEM 10

struct shared_memory
{
  int id;
  int ref_count;
  struct sleeplock lock;
  uint physical_memory;
};

struct shared_memory shared_memory_table[MAX_SHARED_MEM];
```

در این استراکت:

- متغیر id: شناسه صفحه
- متغیر ref_count: تعداد رفرنس هایی که به این صفحه دسترسی دارند
- متغیر lock: قفل برای مدیریت دسترسی
- متغیر physical_memory: پوینتر به شروع فریم فیزیکی

بخش open_sharedmem و close_sharedmem

ابتدا یک تابع برای مقدار دهی اولیه shared memory table داریم که بررسی می کند آیا این جدول قبلا مقداردهی اولیه شده است یا خیر (به کمک متغیر گلوبال). اگر این متغیر ست نشده بود جدول را مقداردهی اولیه می کند و این متغیر را ست می کند:

```
int is_init = 0;

void init_shared_memory_table()
{
  if (is_init == 0)
  {
      cprintf("init_shared_memory_table called \n");
      for (int i = 0; i < MAX_SHARED_MEM; i++)
      {
      shared_memory_table[i].id = i;
      shared_memory_table[i].ref_count = 0;
```

```
      shared_memory_table[i].physical_memory = 0;
      initsleeplock(&shared_memory_table[i].lock, "shared_memory_table");
      }
      is_init = 1;
  }
}
```

در ادامه یک تابع پیاده سازی می کنیم که یک بخش حافظه اشتراکی در اختیار ما قرار دهد:

```
void get_sharedmem(int id, char **pointer)
{
  init_shared_memory_table();
  if (shared_memory_table[id].ref_count == 0)
  {
      char *memory = kalloc();
      if (memory == 0)
      {
      panic("get_sharedmem: kalloc failed");
      }
      struct proc *curproc = myproc();
      void *address = (void *)(curproc->top - PGSIZE);
      curproc->top -= PGSIZE;
      memset(memory, 0, PGSIZE);
      mappages(curproc->pgdir, address, PGSIZE, V2P(memory), PTE_W |
PTE_U);
      *pointer = address;
      shared_memory_table[id].physical_memory = V2P(memory);
  }
  else
  {
      uint physical_address = shared_memory_table[id].physical_memory;
      struct proc *curproc = myproc();
      void *address = (void *)(curproc->top - PGSIZE);
      curproc->top -= PGSIZE;
      mappages(curproc->pgdir, address, PGSIZE, physical_address, PTE_W |
PTE_U);
      *pointer = address;
  }
  shared_memory_table[id].ref_count++;
}
```

اگر آیدی داده شده قبلا اختصاص داده نشده باشد، یک صفحه جدید از حافظه فیزیکی اختصاص می دهد و آن با به فضای آدرس مجازی نگاشت می دهد. اگر این آیدی قبلا اختصاص داده شده باشد، صفحه فیزیکی موجود را

به فضای آدرس مجازی نگاشت می دهد و در نهایت رفرنس صفحه آپدیت می شود.

برای اینکه بتوانیم رفرنس به صفحه را کم کنیم یک تابع می نویسیم:

```
void dump_sharedmem(int id)
{
  init_shared_memory_table();
  if (id >= MAX_SHARED_MEM)
  {
      cprintf("dump_sharedmem: shared mem id out of index\n");
      return;
  }
  if (shared_memory_table[id].ref_count > 0)
  {
      shared_memory_table[id].ref_count--;
  }
  else
  {
      cprintf("dump_sharedmem: ref_count is 0");
      return;
  }
  if (shared_memory_table[id].ref_count == 0)
  {
      cprintf("ref count hit 0\n");
      char *virtual_address = P2V(shared_memory_table[id].physical_memory);
      memset(virtual_address, 1, PGSIZE);
      kfree(virtual_address);
      shared_memory_table[id].physical_memory = 0;
  }
}
```

این تابع یکی از رفرنس های صفحه کم می کند و در صورتی که تعداد رفرنس های صفحه به صفر رسید حافظه را آزاد می کند. این تابع آیدی مربوط به صفحه را می گیرد و بررسی می کند که واقعا در جدول وجود دارد یا خیر و سپس از رفرنس ها کم می کند و صفر شدن رفرنس را بررسی می کند و اگر صفر شده باشد آن را آزاد می کند.

برای مدیریت دسترسی به این صفحات نیازمند قفل هستیم که ابتدا تابعی برای گرفتن این قفل می نویسیم:

```
void get_sharedmem_lock(int id)
{
  acquiresleep(&shared_memory_table[id].lock);
}
```

همچنین برای رها کردن قفل به تابع دیگری نیاز داریم:

```
void let_sharedmem_lock(int id)
{
  releasesleep(&shared_memory_table[id].lock);
}
```

کاربرد این توابع در در سیستم کال هایی است که برای پیاده سازی برنامه آزمون استفاده شده اند.

```
int sys_open_sharedmem(void)
{
  int id;
  argint(0, &id);
  if (id < 0)
  {
      return -1;
  }
  char **val;
  if (argptr(1, (void *)&val, sizeof(char **)) < 0)
  {
      return -1;
  }
  get_sharedmem(id, val);

  return 0;
}
```

```
int sys_close_sharedmem(void)
{
  int id;
  argint(0, &id);
  if (id < 0)
  {
      return -1;
  }
  dump_sharedmem(id);
  return 0;
}
```

```
int sys_acquire_sharedmem_lock()
{
  int id;
```

```
  argint(0, &id);
  if (id < 0)
  {
      return -1;
  }
  get_sharedmem_lock(id);
  return 0;
}
```

```
int sys_release_sharedmem_lock()
{
  int id;
  argint(0, &id);
  if (id < 0)
  {
      return -1;
  }
  let_sharedmem_lock(id);
  return 0;
}
```

برنامه آزمون

در برنامه آزمون متغیری که مقدار فاکتوریل به ازای هر ایندکس در آن آپدیت می شود یک متغیر اشتراکی است و هر پردازه فرزند آن را آپدیت می کند. با توجه به اینکه تعداد فرزند ها برابر عددی است که می خواهیم فاکتوریل آن را حساب کنیم، تعداد فرزند ها پویا است:

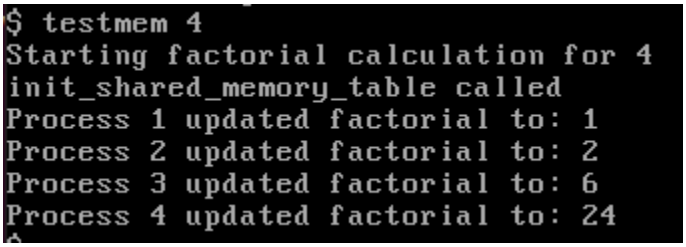
```
for (int i = 1; i <= number; i++)
{
	int pid = fork();
	if (pid == 0)
	{
		calculate_factorial(i, i);
		sleep(10);
		exit();
	}
}
```

همچنین برای جلوگیری از ایجاد race condition قفلی که پیاده سازی کردیم برای اعمال تغییرات در ناحیه اشتراکی ابتدا گرفته می شود و پس از اتمام کار رها می شود:

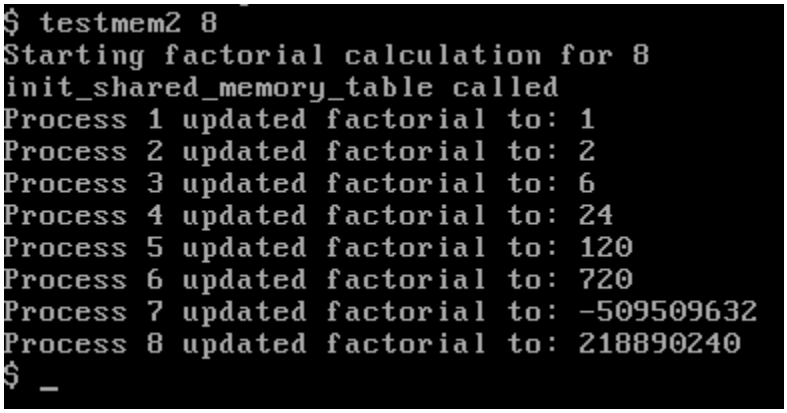
```
void calculate_factorial(int index, int num)
{
	char *val = 0;

	open_sharedmem(MEM_ID, &val);
	acquire_sharedmem_lock(MEM_ID);

	int *shared_val = (int *)val;
	*shared_val *= num;
	printf(1, "Process %d updated factorial to: %d\n", index,
shared_val);

	release_sharedmem_lock(MEM_ID);
	close_sharedmem(MEM_ID);
}
```

```c
int main(int argc, char *argv[])
{
	if (argc != 2)
	{
	printf(1, "Usage: %s <number>\n", argv[0]);
	exit();
	}
	int number = atoi(argv[1]);
	if (number < 0)
	{
	printf(1, "Error: Factorial is not defined for negative numbers.\n");
	exit();
	}

	printf(1, "Starting factorial calculation for %d\n", number);

	char *val = 0;

	open_sharedmem(MEM_ID, &val);
	acquire_sharedmem_lock(MEM_ID);

	int *shared_val = (int *)val;
	*shared_val = 1;

	release_sharedmem_lock(MEM_ID);
	for (int i = 1; i <= number; i++)
	{
	int pid = fork();
	if (pid == 0)
	{
		calculate_factorial(i, i);
		sleep(10);
		exit();
	}
	}
	for (int i = 1; i <= number; i++)
	{
	wait();
	}
	exit();
}
```

ابتدا برای حالتی که برنامه قفل دارد، تست را ران می کنیم:



چهار پردازه فرزند ایجاد شده اند که به ترتیب متغیر اشتراکی را آپدیت کرده اند و حاصل فاکتوریل 4 که 24 می باشد حساب شده است.



در این حالت چون قفلی نداریم ممکن است یکی از پردازه های فرزند زودتر از موعد اقدام به آپدیت متغیر اشتراکی بکند و در این صورت مقدار درستی نخواهیم داشت و طبق تصویر فاکتوریل 8 اشتباه محاسبه شده است.