آزمایشگاه سیستم عامل

گزارش کار پروژه پنجم

اعضای گروه:

امیرحسین عارف زاده 810101604

مهدی نایینی 810101536

کیارش خراسانی 810101413

Repository : https://github.com/Amir-rfz/OS-Lab

Latest Commit : d28336000177ac230ff1ee97d929e9b28cc485f5

مقدمه

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

در لینوکس، هسته از مناطق حافظه مجازی (VMAs)برای پیگیری نگاشت های حافظه یک فرآیند استفاده میکند. یک فرآیند یک VMA برای کد خود، یک VMA برای هر نوع داده و یک VMA برای هر نگاشت حافظه مجزا دارد. VMA ها ساختارهای مستقل از پردازنده با مجوزها (Permissions) و پرچم های کنترل دسترسی (Access control flags) هستند.
هر VMA یک آدرس شروع و یک طول دارد و اندازه آنها همیشه مضربی از اندازه صفحه است. مناطق حافظه توصیف شده توسط VMA همیشه به صورت مجازی به هم پیوسته هستند و نه از نظر فیزیکی.
حال این را با xv6 مقایسه میکنیم:
در xv6 مفهومی از VMA وجود ندارد. در عوض، xv6 از یک جدول صفحه دو سطحی ساده برای مدیریت تخصیص حافظه خود استفاده می کند. تمام حافظه فیزیکی قابل استفاده توسط kpgdir در فضای آدرس مجازی نگاشت می شود، بنابراین تمام حافظه می تواند توسط آدرس های مجازی آدرس دهی شود و توسط واحد مدیریت حافظه (MMU) ترجمه شود. بخشی از جدول صفحه که با صفحات هسته سروکار دارد در تمام process ها یکسان است.

با این حال، بر خالف لینوکس، xv6 از demand paging استفاده نمیکند، بنابراین مفهومی از حافظه مجازی وجود ندارد. به طور خالصه لینوکس از VMA برای مدیریت نگاشت حافظه برای یک فرآیند استفاده می کند در حالی که xv6 از ساختار جدول صفحه دو سطحی ساده تری استفاده می کند.

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

در بسیاری از برنامه ها، کل فضای آدرس به طور کامل به کار نمیرود. ممکن است بخشهای بزرگی از فضای آدرس استفاده نشده یا پراکنده باشد. با صفحه بندی سلسله مراتبی، فقط بخشهایی از جداول صفحات مرتبط با فضای آدرس استفاده شده باید اختصاص داده و پر شوند، که به کاهش کل مصرف حافظه نسبت به یک جدول صفحات مسطح (Flat page table) که باید فضای لازم برای کل فضای آدرس را رزرو کند، منجر میشود.
با توجه به اینکه هر منطقه ای از حافظه که برای ناحیه ای که توسط شماره صفحه سطح بالا پوشش داده نشده است، نیازی به تخصیص جدول صفحه سطح پایین تر ندارد، از این رو از اختصاص جدول صفحه سطح پایین برای آن ناحیه جلوگیری می کند و در نتیجه منجر به کاهش حافظه می شود.

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

20 بیت در هر دو سطح وجود دارد که کارایی تقریبا یکسانی دارند هر کدام به سطح بالاتر خود اشاره دارند. در Page Directory به آدرس شروع جدول صفحه مورد نظر در حافظه اشاره میکند و در Page Table به آدرس قاب (frame) مورد نظر در آدرس فیزیکی (Physical memory) اشاره دارد.
همچنین 12 بیت در هر دو سطح به عنوان سطح دسترسی نگه داری میشود.
Page Directory و Page Table هر دو مدخل های یکسانی دارند و تنها تفاوتشان در بیت D(dirty) می باشد که این بیت برای Table Page کاربردی ندارد اما Directory Page مشخص میکند که صفحه باید در دیسک نوشته شود تا تغییرات اعمال شود.

کد مربوط به ایجاد فضاهای آدرس در xv6

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

همانطور که در ابتدای فایل c.kalloc 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.
```

قطع کد زیر مربوط به تابع kalloc است که در فایل c.kalloc قرار دارد :

```
// 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;
}
```

همانطور که در توضیحات خود 6xv در باالی کد کامنت شده است این تابع یک صفحه 4096 بایتی از حافظه فیزیکی را تخصیص می دهد. در صورتی که بتواند این مقدار از حافظه را تخصیص دهد اشاره گری به آن صفحه برمیگرداند که کرنل می تواند از آن استفاده کند و در صورتی که به هر دلیلی نتواند حافظه تخصیص دهد مقدار صفر را برمیگرداند.

5. تابع mappages چه کاربردی دارد؟

قطع کد زیر مربوط به تابع mappages است که در فایل c.vm قرار 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;
}
```

با توجه به توضیحات 6xv در کامنت بالای کد این تابع برای آدرس مجازی که از va شروع میشود PTE تشکیل می دهد و صفحه جدید را به آدرس فیزیکی که از pa شروع میشود اضافه می کند.(pgdir)

اگر این نگاشت موفقیت آمیز باشد، این تابع 0 و در غیر اینصورت -1 برمیگرداند.

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

قطع کد زیر مربوط به تابع walkpgdir است که در فایل c.vm قرار دارد :

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

با توجه به توضیحاتی که در بالای کد به صورت کامنت آورده شده است این تابع آدرس PTE مرتبط با آدرس مجازی va را از جدول صفحات pgdir مشخص می کند و برمیگرداند. همچنین در صورت لزوم page table لازم را می سازد.

همانطور که مشخص است این تابع عملکردی مشابه عمل سخت افزاری ترجمه آدرس مجازی به آدرس فیزیکی دارد

8. توابع allocuvm و mappages که در ارتباط با حافظه‌ی مجازی هستند را توضیح دهید.

قطع کد زیر مربوط به تابع mappages است که در فایل c.vm قرار 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 در سوال 5 گفته شد 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.

قطع کد زیر مربوط به تابع allocuvm است که در فایل c.vm قرار دارد :

```
// 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;
}
```

این تابع allocuvm در سیستم عامل Unix برای اختصاص دادن صفحات حافظه و حافظه فیزیکی به یک فرآیند استفاده میشود. این تابع برای توسعه اندازه فضای حافظه ی یک فرآیند، از oldsz به newsz صدا میشود.
الگوریتم allocuvm در ابتدا محدودیتهایی را بررسی کرده و سپس برای هر صفحه از oldsz تا newsz، یک صفحه حافظه فیزیکی تخصیص داده و به آن مقداردهی اولیه میکند.
سپس نقشه های مورد نیاز بین فضای حافظه مجازی و فضای حافظه فیزیکی ایجاد میکند. در صورتی که این عملیات با مشکل مواجه شود، حافظهی تخصیص یافته را آزاد کرده و مقدار صفر خروجی می دهد. در غیر اینصورت، اندازه جدید فضای حافظه را بازمیگرداند.

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

ابتدا در این قسمت inode مربوط به path داده شده را با استفاده از تابع namei پیدا میکند و آن را در ip ذخیره میکند.

```
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;
  change_queue(myproc()->pid, UNSET);
```

در ادامه، header ELF فایل مربوطه را چک میکند تا اطمینان حاصل کند که یک فایل اجرایی معتبر باشد.
همچنین از تابع setupkvm برای ایجاد مجموعه جدیدی از جدول های صفحه برای process استفاده می کند.

```
// 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;
```

حال به قطعه کد مربوط به بارگذاری برنامه در حافظه میرسیم. در این قسمت، یک حلقه میزنیم که روی هدر های برنامه در ELF file پیمایش میکند.
توسط readi، هر هدر را خوانده و در ph ذخیره میکنیم.
سپس توسط تابع allocuvm یک فضای حافظه ی جدید برای این بخش از process متناسب با مقدار نیاز یعنی به سایز ph.vaddr + ph.memsz تخصیص می دهیم و سپس توسط تابع loaduvm این بخش را به حافظه ای که گرفتیم لود میکنیم.
در تمامی این فرایند توضیح داده شده، ممکن است به ارور هایی بخوریم که در کد زیر، تمامی این ارور ها را میتوان مشاهده کرد.

```
// 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;
}
iunlockput(ip);
end_op();
ip = 0;
```

شرح پروژه

در این پروژه قصد داریم قابلیت فضای حافظه اشتراکی را به سیستم‌عامل xv6 اضافه کنیم تا دو یا چند پردازه بتوانند به یک فضای حافظه مشترک دسترسی داشته باشند. برای این منظور، دو فراخوانی سیستمی جدید به هسته اضافه خواهد شد و یک سیستم مدیریت حافظه اشتراکی طراحی می‌شود که شامل سه ساختار اصلی است. ساختار SharedMemory فضای اشتراکی هر پردازه را با ذخیره اطلاعاتی مانند شناسه حافظه، کلید، اندازه، و آدرس مجازی مدیریت می‌کند. ساختار SharedMemoryRegion ناحیه‌های حافظه اشتراکی را با ویژگی‌هایی مانند کلید ناحیه، اندازه (تعداد صفحات)، تعداد پردازه‌های مرتبط و آدرس فیزیکی صفحات کنترل می‌کند. در نهایت، ساختار SharedMemoryTable جدولی برای مدیریت تمامی نواحی اشتراکی سیستم است که با استفاده از یک قفل، دسترسی‌های همزمان را کنترل می‌نماید. این تغییرات امکان اشتراک‌گذاری داده‌ها در حافظه و ارتباط بین پردازه‌ها را فراهم می‌آورد.

```c
typedef struct SharedMemory {
  int mem_id;
  uint key;
  uint size;
  void *virtual_address;
} SharedMemory;
```

```c
struct SharedMemoryRegion {
  uint key;
  uint size;
  uint shared_memory_part_size;
  int mem_id;
  int shared_memory_nattch;
  void *physical_address[NUM_SHARED_MEMORY];
};

struct SharedMemoryTable {
  struct spinlock lock;
  struct SharedMemoryRegion shaared_mem[NUM_SHARED_MEMORY];
} SharedMemoryTable;
```

تابع create_shared_memory مسئول تخصیص و مقدار دهی اولیه به یک ناحیه حافظه اشتراکی در سیستم‌عامل xv6 است. این تابع اندازه مورد نظر برای حافظه اشتراکی و یک اندیس برای شناسایی ناحیه در جدول حافظه اشتراکی را به عنوان ورودی دریافت می‌کند. ابتدا تعداد صفحات مورد نیاز را محاسبه کرده، ورودی‌ها را بررسی می‌کند و سپس با استفاده از تابع kalloc صفحات فیزیکی مورد نیاز را تخصیص می‌دهد. این صفحات قبل از استفاده پاک‌سازی شده و به ناحیه حافظه اشتراکی تخصیص داده می‌شوند. پس از موفقیت در تخصیص، جدول حافظه اشتراکی با ویژگی‌های ناحیه مانند اندازه، آدرس‌های فیزیکی، و شناسه منحصربه‌فرد به‌روزرسانی می‌شود. همچنین، برای جلوگیری از مشکلات همزمانی، قفل‌هایی جهت مدیریت دسترسی به کار گرفته می‌شوند. در صورت بروز خطا در تخصیص یا نامعتبر بودن ورودی‌ها، منابع به صورت ایمن آزاد شده و خطا بازگردانده می‌شود.

```
int
create_shared_memory(uint size, int given_index)
{
  acquire(&SharedMemoryTable.lock);

  int num_of_pages = (size / PGSIZE) + 1;
  if (size <= 0 || num_of_pages > NUM_SHARED_MEMORY) {
    release(&SharedMemoryTable.lock);
    return -1;
  }

  for (int i = 0; i < num_of_pages; i++) {
    char *new_page = kalloc();

    if (new_page == 0) {
      cprintf("memory limit: failed to allocate a page\n");
      release(&SharedMemoryTable.lock);
      return -1;
    }

    memset(new_page, 0, PGSIZE);
    SharedMemoryTable.shaared_mem[given_index].physical_address[i] = (void *)V2P(new_page);
  }

  SharedMemoryTable.shaared_mem[given_index].key = 0;
  SharedMemoryTable.shaared_mem[given_index].mem_id = given_index;
  SharedMemoryTable.shaared_mem[given_index].size = num_of_pages;
  SharedMemoryTable.shaared_mem[given_index].shared_memory_part_size = size;

  release(&SharedMemoryTable.lock);
  return given_index;
}
```

Open_sharedmem

تابع open_shared_memory برای دسترسی به یک ناحیه حافظه اشتراکی موجود یا ایجاد و اتصال آن به پردازه در سیستم‌عامل xv6 طراحی شده است. در ابتدا بررسی می‌شود که آیا شناسه حافظه اشتراکی معتبر است و آیا این ناحیه قبلاً ایجاد شده است یا خیر. در صورت عدم وجود، تابع با استفاده از create_shared_memory ناحیه حافظه مورد نظر را ایجاد می‌کند. سپس، تابع آدرس مجازی مناسبی برای اتصال حافظه به فضای آدرس پردازه تعیین می‌کند، با اطمینان از اینکه تداخل یا تجاوز به سایر نواحی وجود ندارد. پس از نگاشت صفحات حافظه فیزیکی به آدرس مجازی پردازه با استفاده از mappages، اطلاعات مربوط به این ناحیه در ساختار پردازه ثبت شده و شمارنده تعداد پردازه‌های متصل به ناحیه اشتراکی (nattch) افزایش می‌یابد. در صورت بروز هرگونه خطا، منابع تخصیص داده شده آزاد شده و مقدار خطا بازگردانده می‌شود.

```
void*
open_shared_memory(int mem_id)
{
  if (mem_id < 0 || mem_id > NUM_SHARED_MEMORY) {
    return (void *)-1;
  }

  acquire(&SharedMemoryTable.lock);

  int foound_index;
  void *least_virtual_address;
  void *virtual_address = (void *)HEAPLIMIT;
  struct proc *process = myproc();

  int index = SharedMemoryTable.shaared_mem[mem_id].mem_id;
  if (index == -1) {
    release(&SharedMemoryTable.lock);
    index = create_shared_memory(2565, mem_id);
    acquire(&SharedMemoryTable.lock);
  }

  if (index == -1) {
    release(&SharedMemoryTable.lock);
    return (void *)-1;
  }

  for (int i = 0; i < NUM_SHARED_MEMORY; i++) {
    foound_index = get_least_index(virtual_address, process);
    if (foound_index != -1) {
      least_virtual_address = process->pages[foound_index].virtual_address;

      if ((uint)virtual_address + SharedMemoryTable.shaared_mem[index].size * PGSIZE <= (uint)least_virtual_address) {
        break;
      }
      else {
        virtual_address = (void *)((uint)least_virtual_address + process->pages[foound_index].size * PGSIZE);
      }
    }
    else {
      break;
    }
  }

  if ((uint)virtual_address + SharedMemoryTable.shaared_mem[index].size * PGSIZE >= KERNBASE) {
    release(&SharedMemoryTable.lock);
    return (void *)-1;
  }

  foound_index = -1;
  for (int i = 0; i < NUM_SHARED_MEMORY; i++) {
    if (process->pages[i].key != -1 &&
      (uint)process->pages[i].virtual_address + process->pages[i].size * PGSIZE > (uint)virtual_address &&
      (uint)virtual_address >= (uint)process->pages[i].virtual_address)
    {
      foound_index = i;
      break;
    }
  }

  if (foound_index != -1) {
    release(&SharedMemoryTable.lock);
    return (void *)-1;
  }

  for (int k = 0; k < SharedMemoryTable.shaared_mem[index].size; k++) {
    if (mappages(process->pgdir, (void *)((uint)virtual_address + (k * PGSIZE)), PGSIZE, (uint)SharedMemoryTable.shaared_mem[index].physical_address[k], 06) < 0)
    {
      deallocuvm(process->pgdir, (uint)virtual_address, (uint)(virtual_address + SharedMemoryTable.shaared_mem[index].size));
      release(&SharedMemoryTable.lock);
      return (void *)-1;
    }
  }

  foound_index = -1;
  for (int i = 0; i < NUM_SHARED_MEMORY; i++) {
    if (process->pages[i].key == -1) {
      foound_index = i;
      break;
    }
  }

  if (foound_index != -1) {
    process->pages[foound_index].mem_id = mem_id;
    process->pages[foound_index].virtual_address = virtual_address;
    SharedMemoryTable.shaared_mem[index].shared_memory_nattch += 1;
    process->pages[foound_index].key = SharedMemoryTable.shaared_mem[index].key;
    process->pages[foound_index].size = SharedMemoryTable.shaared_mem[index].size;
  }

  else {
    release(&SharedMemoryTable.lock);
    return (void *)-1;
  }

  release(&SharedMemoryTable.lock);
  return virtual_address;
}
```

Close_sharedmem

این تابع با نام close_shared_memory برای مدیریت حافظه اشتراکی در یک سیستم طراحی شده است. هدف اصلی این تابع، بررسی و کاهش تعداد مراجع (references) به یک حافظه اشتراکی خاص است. در صورتی که تعداد مراجع به صفر برسد، حافظه اشتراکی از جدول مدیریت حافظه حذف می‌شود، اما آزادسازی حافظه انجام نمی‌گیرد زیرا هنوز در جدول تبدیل حافظه مجازی به فیزیکی به عنوان یک حافظه معتبر در نظر گرفته می‌شود. عملکرد این تابع شامل جستجوی آدرس مجازی مرتبط در ساختار داده‌های پردازه، به‌روزرسانی اطلاعات مربوط به حافظه اشتراکی، کاهش تعداد مراجع، و در صورت نیاز آزادسازی منابع مرتبط با حافظه است. این فرآیند با استفاده از قفل‌ها جهت اطمینان از هماهنگی و جلوگیری از مشکلات هم‌زمانی انجام می‌شود.

```
int
close_shared_memory(void *shmaddr)
{
  acquire(&SharedMemoryTable.lock);
  void *virtual_address = (void *)0;
  struct proc *process = myproc();
  int mem_id;
  int index;
  uint size;

  for (int i = 0; i < NUM_SHARED_MEMORY; i++) {
    if (process->pages[i].key != -1 && process->pages[i].virtual_address == shmaddr) {
      index = i;
      size = process->pages[index].size;
      mem_id = process->pages[i].mem_id;
      virtual_address = process->pages[i].virtual_address;
      break;
    }
  }

  if (virtual_address) {
    for (int i = 0; i < size; i++) {
      pte_t *pte = walkpgdir(process->pgdir, (void *)((uint)virtual_address + i * PGSIZE), 0);
      if (pte == 0) {
        release(&SharedMemoryTable.lock);
        return -1;
      }

      *pte = 0;
    }

    process->pages[index].size = 0;
    process->pages[index].key = -1;
    process->pages[index].mem_id = -1;
    process->pages[index].virtual_address = (void *)0;

    if (SharedMemoryTable.shaared_mem[mem_id].shared_memory_nattch > 0) {
      SharedMemoryTable.shaared_mem[mem_id].shared_memory_nattch -= 1;
    }

    if (SharedMemoryTable.shaared_mem[mem_id].shared_memory_nattch == 0) {
      for (int i = 0; i < SharedMemoryTable.shaared_mem[index].size; i++) {
        char *addr = (char *)P2V(SharedMemoryTable.shaared_mem[index].physical_address[i]);
        kfree(addr);
        SharedMemoryTable.shaared_mem[index].physical_address[i] = (void *)0;
      }

      SharedMemoryTable.shaared_mem[index].size = 0;
      SharedMemoryTable.shaared_mem[index].shared_memory_nattch = 0;
      SharedMemoryTable.shaared_mem[index].shared_memory_part_size = 0;
      SharedMemoryTable.shaared_mem[index].key = SharedMemoryTable.shaared_mem[index].mem_id = -1;
    }

    release(&SharedMemoryTable.lock);
    return 0;
  }

  else {
    release(&SharedMemoryTable.lock);
    return -1;
  }
}

void close_shared_memory_wrapper(void *address) {
  close_shared_memory(address);
}
```

برنامه آزمون

این برنامه با هدف آزمودن عملکرد صحیح حافظه اشتراکی در کرنل xv6 طراحی شده است. در این برنامه، یک پردازه والد فضایی اشتراکی در حافظه ایجاد کرده و چندین پردازه فرزند برای محاسبه توزیع‌شده فاکتوریل یک عدد مشخص ایجاد می‌شوند. هر پردازه فرزند مقدار فعلی ذخیره‌شده در حافظه اشتراکی را خوانده و آن را با استفاده از عدد جدید به‌روزرسانی می‌کند. برای جلوگیری از وقوع شرایط رقابتی (Race Conditions)، از یک فایل قفل ساده برای همگام‌سازی عملیات پردازه‌ها استفاده شده است. این برنامه به صورت دینامیک تعداد پردازه‌های فرزند را تعیین می‌کند و امکان مقایسه خروجی با یک یا چند پردازه را فراهم می‌سازد تا عملکرد و صحت حافظه اشتراکی به خوبی ارزیابی شود.

```c
#include "types.h"
#include "user.h"
#include "fcntl.h"

void acquire_user() {
  while ((open("lockfile", O_CREATE  | O_WRONLY)) < 0);
}

void release_user() {
  unlink("lockfile");
}

void test_shared_memory_with_factorial(int input_factorial) {
  int mem_id_num = 0;
  int mem_id_fact = 1;
  void *addr_num = (void *)open_shared_memory(mem_id_num);
  void *addr_factorial = (void *)open_shared_memory(mem_id_fact);

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

      int pre_num = (*(int *)addr_num);
      int pre_fact = (*(int *)addr_factorial);
      (*(int *)addr_num)++;

      if (pre_fact == 0) {
        (*(int *)addr_factorial) = (pre_num + 1);
      }
      else {
        (*(int *)addr_factorial) = pre_fact * (pre_num + 1);
      }

      release_user();
      exit();
    }
  }

  for (int i = 0; i < input_factorial; i++) {
    wait();
  }

  printf(1, "Factorial of %d = %d\n", input_factorial, *(int *)addr_factorial);
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf(1, "Usage: factorial <input_number>\n");
    exit();
  }

  int input_factorial = atoi(argv[1]);
  test_shared_memory_with_factorial(input_factorial);
  exit();
}
```

خروجی اجرای برنامه آزمون:

```
Booting from Hard Disk...
cpu1: starting 1
cpu2: starting 2
cpu0: starting 0
sb: size 1000 nblocks 954 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 45
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ factorial 5
Factorial of 5 = 120
$ factorial 6
Factorial of 6 = 720
$
```

```c
void test_shared_memory_with_factorial(int input_factorial) {
  int mem_id_num = 0;
  int mem_id_fact = 1;
  void *addr_num = (void *)open_shared_memory(mem_id_num);
  void *addr_factorial = (void *)open_shared_memory(mem_id_fact);

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

      int pre_num = (*(int *)addr_num);
      int pre_fact = (*(int *)addr_factorial);
      (*(int *)addr_num)++;

      if (pre_fact == 0) {
        (*(int *)addr_factorial) = (pre_num + 1);
      }
      else {
        (*(int *)addr_factorial) = pre_fact * (pre_num + 1);
      }

      // release_user();
      exit();
    }
  }
```

```
cpu1: starting 1
cpu2: starting 2
cpu3: starting 3
cpu4: starting 4
cpu5: starting 5
cpu6: starting 6
cpu7: starting 7
cpu0: starting 0
sb: size 1000 nblocks 954 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 45
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ factorial 10
Factorial of 10 = 3628800
$ factorial 11
Factorial of 11 = 39916800
$ factorial 12
Factorial of 12 = 479001600
$ factorial 13
Factorial of 13 = 1932053504
```