گزارش ازمایشگاه سیستم عامل پروژه 5

سبحان کوشکی محمد جزایری نوید هاشمی
810101496 810101399 810101549

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

VMA مخفف Virtual Memory Area است و به بازه‌ای از فضای حافظه مجازی اشاره دارد که ویژگی‌ها و خصوصیات یکسانی دارد. در سیستم عامل لینوکس، هر فرآیند از یک فضای آدرس مجازی جداگانه استفاده می‌کند، و این فضا به بخش‌هایی تقسیم می‌شود که هر بخش یک VMA است.
از کاربرد virtual memory area میتوان به موارد زیر اشاره کرد:
مدیریت حافظه:
VMAها به سیستم عامل کمک می‌کنند تا حافظه فرآیندها را به صورت مجزا مدیریت کند. هر VMA می‌تواند ویژگی‌های متفاوتی داشته باشد، مانند:

- نوع دسترسی (خواندن، نوشتن، اجرا).
- منبع داده (مثلاً یک فایل روی دیسک، یا یک منطقه از حافظه مشترک).

جداسازی و حفاظت حافظه:
با استفاده از VMAها، سیستم عامل می‌تواند از دسترسی غیرمجاز به حافظه جلوگیری کند و امنیت فرآیندها را تضمین کند.
پیاده‌سازی بهینه‌تر تخصیص حافظه:
برای عملیات‌هایی مانند تخصیص یا آزاد کردن حافظه، VMAها به سیستم عامل این امکان را می‌دهند که سریع‌تر عمل کند.

در سیستم‌عامل xv6 ، مدیریت حافظه مجازی بسیار ساده‌تر از لینوکس است و مفهومی مشابه Virtual Memory Area به صورت کامل پیاده‌سازی نشده است. با این حال، می‌توان نحوه مدیریت حافظه مجازی در xv6 را بررسی کرد و مشابهتی با VMAهای لینوکس پیدا کرد.
در xv6، هر فرآیند از یک فضای آدرس مجازی جداگانه استفاده می‌کند. این فضا توسط یک جدول صفحه (Page Table) مدیریت می‌شود که نگاشت بین آدرس‌های مجازی و فیزیکی را برقرار می‌کند.
برخلاف لینوکس که از VMAها استفاده می‌کند، xv6 از یک رویکرد ساده‌تر برای مدیریت بازه‌های حافظه استفاده می‌کند.

چرا ساختار سلسله مراتبی منجر به کاهشمصرف حافظه می گردد؟

ساختار سلسله‌مراتبی و کاهش نیاز به ذخیره کل جدول صفحات

اگر سیستم تنها از یک جدول صفحه استفاده کند، باید برای هر آدرس مجازی ممکن (فضای آدرس مجازی) یک ورودی در جدول صفحه ذخیره شود، حتی اگر بخشی از این آدرس‌ها توسط فرآیند استفاده نشود. این منجر به اتلاف حافظه برای ذخیره اطلاعات غیرضروری می‌شود.

صرفه‌جویی در حافظه برای فرآیندهای با فضای آدرس پراکنده

بسیاری از فرآیندها از کل فضای آدرس مجازی خود استفاده نمی‌کنند. معمولاً حافظه فرآیندها به صورت پراکنده تخصیص داده می‌شود.

در طرح مسطح ، باید برای کل فضای آدرس ورودی تعریف شود، حتی اگر بخش عمده‌ای از این فضا استفاده نشود.

کاهش تعداد ورودی‌ها با استفاده از تقسیم‌بندی سلسله‌مراتبی
کاهش سربار ناشی از نگهداری جدول‌های بزرگ

در سیستم‌های با فضای آدرس بزرگ استفاده از یک جدول صفحات مسطح غیرعملی است. زیرا اندازه جدول صفحات با افزایش فضای آدرس بسیار بزرگ می‌شود. در طرح سلسله‌مراتبی، ساختار به بخش‌های کوچک‌تر تقسیم می‌شود که ذخیره‌سازی و دسترسی به آن‌ها ساده‌تر و بهینه‌تر است.

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

همانطور که در تصویر صورت پروژه میتوان دید هر دو سطح 32 بیتی هستند که هر دو یک قسمت 20 بیتی PPN و flag دارند و در تصویر پایین تر دقیق نوشته شده که در ان قسمت flag چه چیز هایی نوشته شده است تنها تفاوتی که به چشم میخورد ان یک بیت dirty وجود دارد که در page table معنایی ندارد ولی در بخش page directory به معنای این است که این داده ای که در کش هست باید در حافظه اصلی نوشته شود و چیزی که در حافظه اصلی است معتبر نیست.

تابع kalloc چه نوع حافظه ای تخصیص می دهد؟

تابع `kalloc` در سیستم‌عامل xv6 حافظه‌ای از نوع فیزیکی تخصیص می‌دهد.
این تابع یک بلاک از حافظه فیزیکی با اندازه یک صفحه (معمولاً 4 کیلوبایت) را به فرآیند اختصاص می‌دهد. حافظه تخصیص داده‌شده توسط این تابع در مدیریت نگاشت حافظه مجازی به فیزیکی استفاده می‌شود.

تابع mappages چه کاربردی دارد؟

تابع `mappages` برای نگاشت آدرس‌های مجازی به آدرس‌های فیزیکی در جدول صفحه استفاده می‌شود.

- وظیفه اصلی این تابع، تنظیم جدول صفحه (Page Table) برای مشخص کردن نگاشت آدرس‌ها است.
- این تابع آدرس‌های مجازی یک فرآیند را به صفحات فیزیکی تخصیص داده‌شده توسط `kalloc` متصل می‌کند.
- کاربرد آن در پیاده‌سازی مدیریت حافظه مجازی است و به سیستم امکان می‌دهد که از فضای آدرس مجازی مستقل از حافظه فیزیکی استفاده کند.

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

تابع `walkpgdir` برای پیمایش و شبیه‌سازی عملیات سخت‌افزاری ترجمه آدرس مجازی به فیزیکی در جدول صفحات (Page Table) استفاده می‌شود.

وظایف ان به شکل زیر است:

1. جستجوی آدرس مجازی:
   - ابتدا دایرکتوری صفحه (Page Directory) را بررسی می‌کند.
   - سپس به جدول صفحه (Page Table) متناظر دسترسی پیدا می‌کند.
2. ایجاد ورودی جدید (در صورت نیاز):
   - اگر ورودی مربوط به آدرس وجود نداشته باشد و فلگ مخصوص تنظیم شده باشد، یک Page Table جدید ایجاد می‌کند.
3. بازگرداندن نتیجه:
   - اشاره‌گری به ورودی پیدا شده (یا ایجاد شده) را برمی‌گرداند.

کاربردهای ان ان عبارت اند از:

- استفاده در توابع مدیریت حافظه مثل `mappages`.
- نگاشت آدرس‌های مجازی به فیزیکی.
- پشتیبانی از عملیات‌هایی مثل `fork` و `exec`.
- شبیه‌سازی عمل سخت‌افزاری MMU برای ترجمه آدرس‌ها.

این تابع بهینه‌سازی مصرف حافظه را با ساختار سلسله‌مراتبی جدول صفحات ممکن می‌سازد.

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

تابع `allocuvm`:

- این تابع برای افزایش فضای آدرس مجازی یک فرآیند استفاده می‌شود.
- با تخصیص یک یا چند صفحه فیزیکی به فرآیند و نگاشت آن‌ها به آدرس‌های مجازی جدید، اندازه فضای آدرس مجازی را گسترش می‌دهد.
- ابتدا از `kalloc` برای تخصیص صفحات فیزیکی استفاده می‌کند و سپس از `mappages` برای نگاشت آدرس‌های مجازی به آدرس‌های فیزیکی بهره می‌گیرد.

تابع `mappages`:

- این تابع مستقیماً مسئول انجام نگاشت بین آدرس‌های مجازی و صفحات فیزیکی تخصیص‌یافته است.
- جدول صفحات فرآیند را تنظیم می‌کند و آدرس‌های مجازی جدید را به صفحات فیزیکی اختصاص‌یافته متصل می‌کند.
- در واقع، `mappages` ابزار اصلی برای پیاده‌سازی نگاشت حافظه مجازی است که توسط `allocuvm` فراخوانی می‌شود.

ارتباط این دو تابع:
تابع `allocuvm`، حافظه فیزیکی را تخصیص می‌دهد و از `mappages` استفاده می‌کند تا این حافظه را در فضای آدرس مجازی فرآیند قرار دهد. این همکاری کلید پیاده‌سازی مدیریت حافظه مجازی در سیستم عامل است.

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

شیوه بارگذاری برنامه در حافظه توسط فراخوانی سیستم `exec` به صورت زیر است:

1. پاک‌سازی فضای آدرس قدیمی:

- وقتی یک فرآیند جدید توسط `exec` اجرا می‌شود، ابتدا فضای آدرس قدیمی فرآیند پاک می‌شود.

2. ایجاد فضای آدرس جدید:
   - یک فضای آدرس کاملاً جدید با استفاده از توابعی مانند `setupkvm` و `allocuvm` ایجاد می‌شود.
   - کد و داده‌های برنامه جدید (مانند فایل اجرایی ELF) در این فضای آدرس جدید بارگذاری می‌شوند.
3. بارگذاری بخش‌های برنامه از فایل اجرایی:
   - بخش‌های مختلف برنامه (مانند کد، داده‌ها و ...) از فایل ELF خوانده و در آدرس‌های مناسب حافظه نگاشت می‌شوند.
   - این کار توسط توابعی که فایل ELF را تفسیر می‌کنند (مانند `loadseg`) انجام می‌شود.
4. راه‌اندازی پشته:
   - یک پشته جدید برای فرآیند تخصیص داده می‌شود و اشاره‌گر پشته تنظیم می‌شود.
5. تنظیم شروع اجرا:
   - شمارنده برنامه (Program Counter) به آدرس ورود برنامه (Entry Point) تنظیم می‌شود تا اجرا از آنجا شروع شود.

در `exec`، فضای آدرس قدیمی فرآیند تخریب می‌شود و یک فضای آدرس جدید ایجاد می‌شود. سپس کد و داده‌های برنامه از فایل اجرایی بارگذاری و ساختارهای مورد نیاز (مانند پشته) تنظیم می‌شوند.

بخش کد پروژه 5

ابتدا استراکت هایی که نیاز داریم را میسازیم
اولین استراکتی که درست میکنم برای sharedPages است در اینجا چیز هایی که نگه میداریم ادرس virtual است کلید و سایز و shared memory id و permishion که به ان داریم که به صورت زیر میشود.
بعد ازتعریف این ها در هر pcb هرprocess یک ارایه ای از این shared memory ها نگه میداریم. چیز هایی که در زیر میبینید در proc.h است.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int syscall_count;
  int syscall_history[SYSCALL_NUM];
  int level_queue;
  int arrival;
  int last_exec;
  int burst;
  int certainty;
  int wait_cycles;
  int consecutive;
  int queue_arrival;
  shared_pages pages[SHAREDREGIONS];
};
```

```
#define SHAREDREGIONS 64

typedef struct shared_pages {
  void *virtual_addr;
  int shared_mem_id,perm;
  uint key, size;
} shared_pages;
```

حالا به سراغ استراکت های بعدی رفته و ان ها را تعریف میکنیم بعد از اینکه برای هر process ان استراکت را درست کرده ایم حالا باید برای هر مکانی که در memory به صورت مشترک استفاده میکنیم یک اطلاعاتی را ذخیره کنیم مثل کلید و تعداد صفحه ها و id و فیزیکال ادرس ها و ...

```
struct shared_mem_region {
  uint key, size;
  int shared_mem_id;
  int to_be_deleted;
  void *physical_addr[SHAREDREGIONS];
  struct shared_mem_id_DS buffer;
};

struct shared_mem_table {
  struct spinlock lock;
  struct shared_mem_region all_regions[SHAREDREGIONS];
} shared_mem_table;
```

در استراکت پایین ما یک لایه بالا تر می آییم و برای هر shared memory یک table نگه میداریم که در انجا یک قفل هم میگذاریم.

```
int get_shared_mem(uint key, uint size, int shared_mem_flag) {
  int lower_bits = shared_mem_flag & 7, permission = -1;

  acquire(&shared_mem_table.lock);
  if(lower_bits == (int)READ_SHM) {
    permission = READ_SHM;
    shared_mem_flag ^= READ_SHM;
  }
  else if(lower_bits == (int)RW_SHM) {
    permission = RW_SHM;
    shared_mem_flag ^= RW_SHM;
  } else {
    if(!((shared_mem_flag == 0) && (key != IPC_PRIVATE))) {
      release(&shared_mem_table.lock);
      return -1;
    }
  }
  if(size <= 0) {
    release(&shared_mem_table.lock);
    return -1;
  }
  int num_of_pages = (size / PGSIZE) + 1;
  if(num_of_pages > SHAREDREGIONS) {
    release(&shared_mem_table.lock);
    return -1;
  }
  int index = -1;
  for(int i = 0; i < SHAREDREGIONS; i++) {
    if(shared_mem_table.all_regions[i].key == key) {
      if(shared_mem_table.all_regions[i].size != num_of_pages) {
        release(&shared_mem_table.lock);
        return -1;
      }
      if(shared_mem_flag == (IPC_CREAT | IPC_EXCL)) {
        release(&shared_mem_table.lock);
        return -1;
      }
      int check_perm = shared_mem_table.all_regions[i].buffer.shared_mem_perm.mode;
      if(check_perm == READ_SHM || check_perm == RW_SHM) {
        if((shared_mem_flag == 0) && (key != IPC_PRIVATE)) {
          release(&shared_mem_table.lock);
          return shared_mem_table.all_regions[i].shared_mem_id;
        }
        if(shared_mem_flag == IPC_CREAT) {
          release(&shared_mem_table.lock);
          return shared_mem_table.all_regions[i].shared_mem_id;
        }
      }
      release(&shared_mem_table.lock);
      return -1;
    }
  }
```

```
  for(int i = 0; i < SHAREDREGIONS; i++) {
    if(shared_mem_table.all_regions[i].key == -1) {
      index = i;
      break;
    }
  }
  if(index == -1) {
    release(&shared_mem_table.lock);
    return -1;
  }
  if((key == IPC_PRIVATE) || (shared_mem_flag == IPC_CREAT) || (shared_mem_flag == (IPC_CREAT | IPC_EXCL))) {
    for(int i = 0; i < num_of_pages; i++) {
      char *new_page = kalloc();
      if(new_page == 0){
        cprintf("shmget: failed to allocate a page (out of memory)\n");
        release(&shared_mem_table.lock);
        return -1;
      }
      memset(new_page, 0, PGSIZE);
      shared_mem_table.all_regions[index].physical_addr[i] = (void *)V2P(new_page);
    }
    shared_mem_table.all_regions[index].size = num_of_pages;
    shared_mem_table.all_regions[index].key = key;
    shared_mem_table.all_regions[index].buffer.shared_mem_segment_size = size;
    shared_mem_table.all_regions[index].buffer.shared_mem_perm.__key = key;
    shared_mem_table.all_regions[index].buffer.shared_mem_perm.mode = permission;
    shared_mem_table.all_regions[index].buffer.shared_mem_creator_pid = myproc()->pid;
    shared_mem_table.all_regions[index].shared_mem_id = index;

    release(&shared_mem_table.lock);
    return index;
  } else {
    release(&shared_mem_table.lock);
    return -1;
  }
}
```

این کد تابعی به نام `get_shared_mem` است حافظه مشترک اجازه می‌دهد چندین فرآیند بتوانند به یک ناحیه حافظه مشترک دسترسی داشته باشند.

توضیح مختصر عملکرد:

1. پارامترهای ورودی:
   - `key`: کلیدی که برای شناسایی ناحیه حافظه مشترک استفاده می‌شود.
   - `size`: اندازه درخواست‌شده برای ناحیه حافظه.
   - `shared_mem_flag`: فلگ‌هایی که نوع عملیات (ایجاد یا دسترسی به حافظه مشترک) و مجوزهای دسترسی (فقط خواندن یا خواندن و نوشتن) را مشخص می‌کنند.
2. بررسی اولیه:
   - فلگ‌ها و مجوزهای دسترسی بررسی می‌شوند تا مقادیر معتبر باشند.

- اندازه حافظه درخواست‌شده بررسی می‌شود (باید بزرگ‌تر از صفر و مطابق محدودیت باشد).

3. جستجو برای ناحیه موجود:
   - بررسی می‌شود که آیا ناحیه حافظه مشترک با کلید مشخص (`key`) از قبل وجود دارد.
   - اگر ناحیه موجود باشد:
     - اندازه آن بررسی می‌شود تا مطابقت داشته باشد.
     - مجوزهای دسترسی و فلگ‌ها نیز بررسی می‌شوند.
     - اگر شرایط برقرار باشد، شناسه حافظه مشترک بازگشت داده می‌شود.
4. ایجاد ناحیه جدید:
   - اگر ناحیه‌ای با کلید مشخص وجود نداشته باشد و درخواست ایجاد (با استفاده از `IPC_CREAT`) داده شده باشد:
     - یک ناحیه حافظه جدید تخصیص داده می‌شود.
     - صفحات حافظه (`()kalloc`) به تعداد موردنیاز ایجاد و مقداردهی اولیه می‌شوند.
     - اطلاعات مربوط به ناحیه جدید در جدول حافظه مشترک ذخیره می‌شود (مانند اندازه، کلید، مجوزها و شناسه فرآیند ایجادکننده).
5. خطاها:
   - در مواردی مانند نامعتبر بودن اندازه، پر شدن نواحی حافظه، یا درخواست‌های ناسازگار، تابع مقدار `-1` را بازمی‌گرداند.
6. بازگشت شناسه حافظه:
   - اگر همه شرایط درست باشد، شناسه حافظه مشترک (`shared_mem_id`) به عنوان خروجی بازگشت داده می‌شود.

```
void* attach_shared_mem(int shared_mem_id, void* shared_mem_addr, int shared_mem_flag) {
  if(shared_mem_id < 0 || shared_mem_id > 64) {
    return (void*)-1;
  }
  acquire(&shared_mem_table.lock);
  int index = -1,idx, perm_flag;
  uint segment,size = 0;
  void *va = (void*)HEAPLIMIT, *least_virtual_addr;
  struct proc *process = myproc();
  index = shared_mem_table.all_regions[shared_mem_id].shared_mem_id;
  if(index == -1) {
    // shared_mem_id not found
    release(&shared_mem_table.lock);
    return (void*)-1;
  }
  if(shared_mem_addr) {
    if((uint)shared_mem_addr >= KERNBASE || (uint)shared_mem_addr < HEAPLIMIT) {
      release(&shared_mem_table.lock);
      return (void*)-1;
    }
    // round down to nearest multiple of SHMLBA
    uint rounded = ((uint)shared_mem_addr & ~(SHMLBA-1));

    if(shared_mem_flag & SHM_RND) {
      if(!rounded) {
        release(&shared_mem_table.lock);
        return (void*)-1;
      }
      va = (void*)rounded;
    } else {

      // page aligned address
      if(rounded == (uint)shared_mem_addr) {
        va = shared_mem_addr;
      }
    }

  } else {
    for(int i = 0; i < SHAREDREGIONS; i++) {
      idx = get_least_virtual_addr_index(va,process);
      if(idx != -1) {
        least_virtual_addr = process->pages[idx].virtual_addr;
        if((uint)va + shared_mem_table.all_regions[index].size*PGSIZE <=  (uint)least_virtual_addr)
          break;
        else
          va = (void*)((uint)least_virtual_addr + process->pages[idx].size*PGSIZE);
      } else
        break;
    }
  }
```

```
if((uint)va + shared_mem_table.all_regions[index].size*PGSIZE >= KERNBASE) {
  // size exceeded
  release(&shared_mem_table.lock);
  return (void*)-1;
}
idx = -1;
for(int i = 0; i < SHAREDREGIONS; i++) {
  if(process->pages[i].key != -1 && (uint)process->pages[i].virtual_addr + process->pages[i].size*PGSIZE > (uint)va && (uint)va >= (uint)process->pages[i].virtual_addr)
    idx = i;
    break;
  }
}
if(idx != -1) {
  if(shared_mem_flag & SHM_REMAP) {
    segment = (uint)process->pages[idx].virtual_addr;
    // repeat till all conflicting mappings are removed
    while(segment < (uint)va + shared_mem_table.all_regions[index].size*PGSIZE) {
      size = process->pages[idx].size;
      release(&shared_mem_table.lock);
      if(deattach_shared_mem((void*)segment) == -1) {
        return (void*)-1;
      }
      acquire(&shared_mem_table.lock);
      idx = get_least_virtual_addr_index((void*)(segment + size*PGSIZE),process);
      if(idx == -1)
        break;
      segment = (uint)process->pages[idx].virtual_addr;
    }
  } else {
    release(&shared_mem_table.lock);
    return (void*)-1;
  }

}
if((shared_mem_flag & SHM_RDONLY) || (shared_mem_table.all_regions[index].buffer.shared_mem_perm.mode == READ_SHM)){
  perm_flag = PTE_U;
}
else if (shared_mem_table.all_regions[index].buffer.shared_mem_perm.mode == RW_SHM) {
  perm_flag = PTE_W | PTE_U;
} else {
  //permission mismatch between get and attach
  release(&shared_mem_table.lock);
  return (void*)-1;
}
for (int k = 0; k < shared_mem_table.all_regions[index].size; k++) {
  if(mappages(process->pgdir, (void*)((uint)va + (k*PGSIZE)), PGSIZE, (uint)shared_mem_table.all_regions[index].physical_addr[k], perm_flag) < 0) {
    deallocuvm(process->pgdir,(uint)va,(uint)(va + shared_mem_table.all_regions[index].size));
    release(&shared_mem_table.lock);
    return (void*)-1;
  }
}
```

```
  idx = -1;
  for(int i = 0; i < SHAREDREGIONS; i++) {
    if(process->pages[i].key == -1) {
      idx = i;
      break;
    }
  }
  if(idx != -1) {
    process->pages[idx].shared_mem_id = shared_mem_id;
    process->pages[idx].virtual_addr = va;
    process->pages[idx].key = shared_mem_table.all_regions[index].key;
    process->pages[idx].size = shared_mem_table.all_regions[index].size;
    process->pages[idx].perm = perm_flag;
    shared_mem_table.all_regions[index].buffer.shared_mem_number_of_attch += 1;
    shared_mem_table.all_regions[index].buffer.shared_mem_last_pid = process->pid;
  } else {
    release(&shared_mem_table.lock);
    return (void*)-1; // all page regions exhausted
  }
  release(&shared_mem_table.lock);
  return va;
}
```

این کد تابعی به نام attach_shared_mem است که برای اتصال یک ناحیه حافظه مشترک به فضای آدرس مجازی فرآیند فعلی استفاده می‌شود. به عبارت دیگر، این تابع به فرآیند اجازه می‌دهد تا یک بخش از حافظه مشترک (که قبلاً ایجاد شده است) را به فضای آدرس خود متصل کند تا بتواند به داده‌های آن دسترسی داشته باشد.

توضیح مختصر عملکرد:

1. پارامترهای ورودی:
   - shared_mem_id: شناسه ناحیه حافظه مشترک که باید متصل شود.
   - shared_mem_addr: آدرسی مجازی که کاربر پیشنهاد می‌کند حافظه مشترک در آن قرار گیرد (در صورت NULL، سیستم بهترین آدرس را پیدا می‌کند).
   - shared_mem_flag: فلگ‌هایی که رفتار اتصال را کنترل می‌کنند، مانند اینکه آیا حافظه فقط خواندنی باشد (SHM_RDONLY) یا باید با همپوشانی (remap) انجام شود (SHM_REMAP).
2. بررسی اولیه:
   - بررسی می‌شود که shared_mem_id معتبر باشد (بین محدوده مجاز).
   - اگر آدرس پیشنهادی کاربر (shared_mem_addr) وجود داشته باشد:
     - بررسی می‌شود که آیا این آدرس در محدوده فضای کاربر (بین HEAPLIMIT و KERNBASE) قرار دارد.
     - اگر فلگ SHM_RND تنظیم شده باشد، آدرس به نزدیک‌ترین مرز صفحه (براساس SHMLBA) گرد می‌شود.
3. پیدا کردن آدرس مناسب:
   - اگر کاربر آدرس مشخص نکرده باشد، سیستم با استفاده از ساختار فضای آدرس فرآیند، بهترین آدرس موجود را پیدا می‌کند که اندازه ناحیه حافظه مشترک در آن جا شود.
   - در صورت وجود همپوشانی (conflict) با صفحات دیگر:
     - اگر فلگ SHM_REMAP تنظیم شده باشد، نقشه‌های متناقض حذف می‌شوند.
     - در غیر این صورت، عملیات با خطا متوقف می‌شود.
4. بررسی مجوزها:
   - مجوزهای اتصال بررسی می‌شوند:
     - اگر ناحیه حافظه مشترک فقط خواندنی باشد (SHM_RDONLY) یا در زمان ایجاد با مجوز فقط خواندنی تنظیم شده باشد، مجوز اتصال به صورت فقط خواندنی تنظیم می‌شود.

- اگر مجوز خواندن و نوشتن (RW_SHM) وجود داشته باشد، اجازه خواندن و نوشتن داده می‌شود.
- اگر مجوزهای اتصال با مجوزهای ناحیه ناسازگار باشند، عملیات با خطا متوقف می‌شود.

5. اتصال حافظه:
   - با استفاده از `mappages`، آدرس‌های فیزیکی صفحات حافظه مشترک به فضای آدرس مجازی فرآیند نگاشت می‌شوند.
   - اگر نگاشت موفق باشد، اطلاعات ناحیه حافظه مشترک در ساختار فرآیند (مثلاً لیست `pages`) ذخیره می‌شود.
   - شمارنده‌های مربوط به تعداد اتصال‌ها (`shared_mem_number_of_attach`) و آخرین فرآیند متصل به این ناحیه به‌روزرسانی می‌شوند.
6. خطاها:
   - در مواردی مانند پر بودن فضای حافظه، تناقض در مجوزها، یا تخصیص آدرس نامعتبر، تابع مقدار `-1` را بازمی‌گرداند.

```
int deattach_shared_mem(void* shared_mem_addr) {
  acquire(&shared_mem_table.lock);
  struct proc *process = myproc();
  void* va = (void*)0;
  uint size;
  int index,shared_mem_id;
  for(int i = 0; i < SHAREDREGION (int)1 ]
    if(process->pages[i].key != -1 && process->pages[i].virtual_addr == shared_mem_addr) {
      va =  process->pages[i].virtual_addr;
      index = i;
      shared_mem_id = process->pages[i].shared_mem_id;
      size = process->pages[index].size;
      break;
    }
  }
  if(va) {
    for(int i = 0; i < size; i++) {
      pte_t* pte = walkpgdir(process->pgdir, (void*)((uint)va + i*PGSIZE), 0);
      if(pte == 0) {
        release(&shared_mem_table.lock);
        return -1;
      }
      *pte = 0;
    }
    process->pages[index].shared_mem_id = -1;
    process->pages[index].key = -1;
    process->pages[index].size =  0;
    process->pages[index].virtual_addr = (void*)0;
    if(shared_mem_table.all_regions[shared_mem_id].buffer.shared_mem_number_of_attch > 0) {
      shared_mem_table.all_regions[shared_mem_id].buffer.shared_mem_number_of_attch -= 1;
    }
    if(shared_mem_table.all_regions[shared_mem_id].buffer.shared_mem_number_of_attch == 0 && shared_mem_table.all_regions[shared_mem_id].to_be_deleted == 1) {
      for(int i = 0; i < shared_mem_table.all_regions[index].size; i++) {
        char *addr = (char *)P2V(shared_mem_table.all_regions[index].physical_addr[i]);
        kfree(addr);
        shared_mem_table.all_regions[index].physical_addr[i] = (void *)0;
      }
      shared_mem_table.all_regions[index].size = 0;
      shared_mem_table.all_regions[index].key = shared_mem_table.all_regions[index].shared_mem_id = -1;
      shared_mem_table.all_regions[index].to_be_deleted = 0;
      shared_mem_table.all_regions[index].buffer.shared_mem_number_of_attch = 0;
      shared_mem_table.all_regions[index].buffer.shared_mem_segment_size = 0;
      shared_mem_table.all_regions[index].buffer.shared_mem_perm.__key = -1;
      shared_mem_table.all_regions[index].buffer.shared_mem_perm.mode = 0;
      shared_mem_table.all_regions[index].buffer.shared_mem_creator_pid = -1;
      shared_mem_table.all_regions[index].buffer.shared_mem_last_pid = -1;
    }
    shared_mem_table.all_regions[shared_mem_id].buffer.shared_mem_last_pid = process->pid;
    release(&shared_mem_table.lock);
    return 0;
  } else {
    release(&shared_mem_table.lock);
    return -1;
  }
```

این تابع با نام `deattach_shared_mem` برای جدا کردن (detach) یک بخش حافظه مشترک از فضای آدرس فرآیند استفاده می‌شود. به عبارت دیگر، اگر فرآیند به یک بخش حافظه مشترک متصل شده باشد، این تابع اتصال را حذف می‌کند و در صورت نیاز منابع مرتبط با آن را آزاد می‌کند.

توضیح مختصر عملکرد:

1. پارامتر ورودی:
   - `shared_mem_addr`: آدرس مجازی‌ای که حافظه مشترک به آن متصل شده است.
2. جستجوی حافظه مشترک متصل‌شده:
   - با پیمایش آرایه `pages` در ساختار فرآیند:
     - بررسی می‌شود که آیا آدرسی که فرآیند داده، به یکی از حافظه‌های مشترک متصل شده است.
     - اگر پیدا شد، اطلاعات مربوط به آن (اندکس، شناسه حافظه مشترک و اندازه) استخراج می‌شود.
3. حذف نگاشت صفحات:
   - برای هر صفحه از حافظه مشترک که به فضای آدرس فرآیند نگاشت شده:
     - از طریق `walkpgdir` و تنظیم صفر در ورودی جدول صفحه (PTE)، نگاشت آن حذف می‌شود.
4. حذف اطلاعات از ساختار فرآیند:
   - اطلاعات مربوط به اتصال حافظه مشترک در آرایه `pages` فرآیند به حالت پیش‌فرض (غیرمعتبر) تنظیم می‌شود.
5. به‌روزرسانی اطلاعات جدول حافظه مشترک:
   - تعداد اتصال‌ها (`shared_mem_number_of_attach`) در جدول حافظه مشترک کاهش می‌یابد.
   - اگر تعداد اتصال‌ها به صفر برسد و فلگ `to_be_deleted` تنظیم شده باشد:
     - حافظه فیزیکی مرتبط با این ناحیه آزاد می‌شود (از طریق `kfree`).
     - تمام فیلدهای ناحیه حافظه مشترک در جدول (`shared_mem_table`) به حالت پیش‌فرض (غیرمعتبر) بازنشانی می‌شوند.
6. بازگشت مقدار:
   - اگر عملیات موفقیت‌آمیز باشد، مقدار `0` بازگردانده می‌شود.
   - در صورت خطا (مثلاً اگر آدرس داده‌شده متصل نباشد)، مقدار `-1` بازگردانده می‌شود.

```
int remove_shared_mem(int shared_mem_id, int cmd, void *buf) {
  if(shared_mem_id < 0 || shared_mem_id > 64){
    return -1;
  }
  acquire(&shared_mem_table.lock);
  struct shared_mem_id_DS *buffer = (struct shared_mem_id_DS *)buf;
  int index = -1;
  index = shared_mem_table.all_regions[shared_mem_id].shared_mem_id;
  if(index == -1) {
    release(&shared_mem_table.lock);
    return -1;
  } else {
    int check_perm = shared_mem_table.all_regions[index].buffer.shared_mem_perm.mode;
    switch(cmd) {
      case IPC_SET:
        if(buffer) {
          if((buffer->shared_mem_perm.mode == READ_SHM) || (buffer->shared_mem_perm.mode == RW_SHM)) {
            shared_mem_table.all_regions[index].buffer.shared_mem_perm.mode = buffer->shared_mem_perm.mode;
            release(&shared_mem_table.lock);
            return 0;
          } else {
            release(&shared_mem_table.lock);
            return -1;
          }
        } else {
          release(&shared_mem_table.lock);
          return -1;
        }
        break;
      case SHM_STAT:
      case IPC_STAT:
        if(buffer && (check_perm == READ_SHM || check_perm == RW_SHM)) {
          buffer->shared_mem_number_of_attch = shared_mem_table.all_regions[index].buffer.shared_mem_number_of_attch;
          buffer->shared_mem_segment_size = shared_mem_table.all_regions[index].buffer.shared_mem_segment_size;
          buffer->shared_mem_perm.__key = shared_mem_table.all_regions[index].buffer.shared_mem_perm.__key;
          buffer->shared_mem_perm.mode = check_perm;
          buffer->shared_mem_creator_pid = shared_mem_table.all_regions[index].buffer.shared_mem_creator_pid;
          buffer->shared_mem_last_pid = shared_mem_table.all_regions[index].buffer.shared_mem_last_pid;
          release(&shared_mem_table.lock);
          return 0;
        } else {
          release(&shared_mem_table.lock);
          return -1;
        }
        break;
```

```
      case IPC_RMID:
        if(shared_mem_table.all_regions[index].buffer.shared_mem_number_of_attch == 0) {
          for(int i = 0; i < shared_mem_table.all_regions[index].size; i++) {
            char *addr = (char *)P2V(shared_mem_table.all_regions[index].physical_addr[i]);
            kfree(addr);
            shared_mem_table.all_regions[index].physical_addr[i] = (void *)0;
          }
          shared_mem_table.all_regions[index].size = 0;
          shared_mem_table.all_regions[index].key = shared_mem_table.all_regions[index].shared_mem_id = -1;
          shared_mem_table.all_regions[index].to_be_deleted = 0;
          shared_mem_table.all_regions[index].buffer.shared_mem_number_of_attch = 0;
          shared_mem_table.all_regions[index].buffer.shared_mem_segment_size = 0;
          shared_mem_table.all_regions[index].buffer.shared_mem_perm.__key = -1;
          shared_mem_table.all_regions[index].buffer.shared_mem_perm.mode = 0;
          shared_mem_table.all_regions[index].buffer.shared_mem_creator_pid = -1;
          shared_mem_table.all_regions[index].buffer.shared_mem_last_pid = -1;
        } else {
          shared_mem_table.all_regions[index].to_be_deleted = 1;
        }
        release(&shared_mem_table.lock);
        return 0;
        break;
      default:
        release(&shared_mem_table.lock);
        return -1;
        break;
    }
  }
}
```

تابع `remove_shared_mem` برای مدیریت و حذف یک بخش حافظه مشترک (shared memory) با توجه به شناسه (`shared_mem_id`) و دستور (`cmd`) مشخص‌شده استفاده می‌شود. این تابع می‌تواند دستورات مختلفی را برای تغییر تنظیمات، خواندن وضعیت، یا حذف کامل حافظه مشترک اجرا کند.

توضیح عملکرد:

1. ورودی‌ها:
   - `shared_mem_id`: شناسه حافظه مشترک که باید روی آن عملیات انجام شود.
   - `cmd`: دستوری که مشخص می‌کند چه عملیاتی باید انجام شود.
   - `buf`: اشاره‌گری به ساختاری که اطلاعات لازم برای دستورات خاص را نگهداری می‌کند یا نتایج را بازمی‌گرداند.
2. اعتبارسنجی اولیه:
   - بررسی می‌شود که `shared_mem_id` معتبر است (در بازه مشخصی باشد).
   - اگر `shared_mem_id` وجود نداشته باشد یا معتبر نباشد، تابع `-1` بازمی‌گرداند.
3. عملیات براساس دستور (`cmd`):

الف. `IPC_SET`

   - تنظیم مجدد دسترسی‌ها (`permissions`) به حافظه مشترک.
   - اگر مقادیر موجود در `buf` معتبر باشند (فقط `READ_SHM` یا `RW_SHM`)، مقادیر به‌روزرسانی می‌شوند.
   - در صورت موفقیت، مقدار `0` بازمی‌گرداند؛ در غیر این صورت، `-1`.

4. ب. `SHM_STAT` یا `IPC_STAT`
   - این دستورات وضعیت حافظه مشترک را به کاربر گزارش می‌دهند.
   - اطلاعاتی مانند تعداد فرآیندهای متصل، اندازه حافظه، دسترسی‌ها، شناسه کاربری و PIDهای مرتبط درون ساختار `buf` کپی می‌شود.
   - اگر مجوزها معتبر باشند، مقدار `0` بازمی‌گرداند؛ در غیر این صورت، `-1`.
5. ج. `IPC_RMID`
   - حافظه مشترک را حذف می‌کند:
     - اگر هیچ فرآیندی به حافظه مشترک متصل نباشد (`shared_mem_number_of_attach == 0`)، منابع آن آزاد و ورودی مربوطه در جدول حافظه مشترک به حالت پیش‌فرض بازنشانی می‌شود.
     - اگر هنوز فرآیندهایی متصل باشند، فقط فلگ `to_be_deleted` تنظیم می‌شود تا حذف حافظه به تأخیر بیفتد.

- مقدار `0` برای موفقیت و `-1` برای خطا بازمی‌گرداند.

4. پیش‌فرض (`default`):
   - اگر `cmd` ناشناخته باشد، تابع `-1` بازمی‌گرداند.
5. خروجی:
   - `0`: عملیات موفقیت‌آمیز.
   - `-1`: در صورت بروز خطا (مانند شناسه نامعتبر، دسترسی‌های نامعتبر، یا داده‌های نادرست در `buf`).

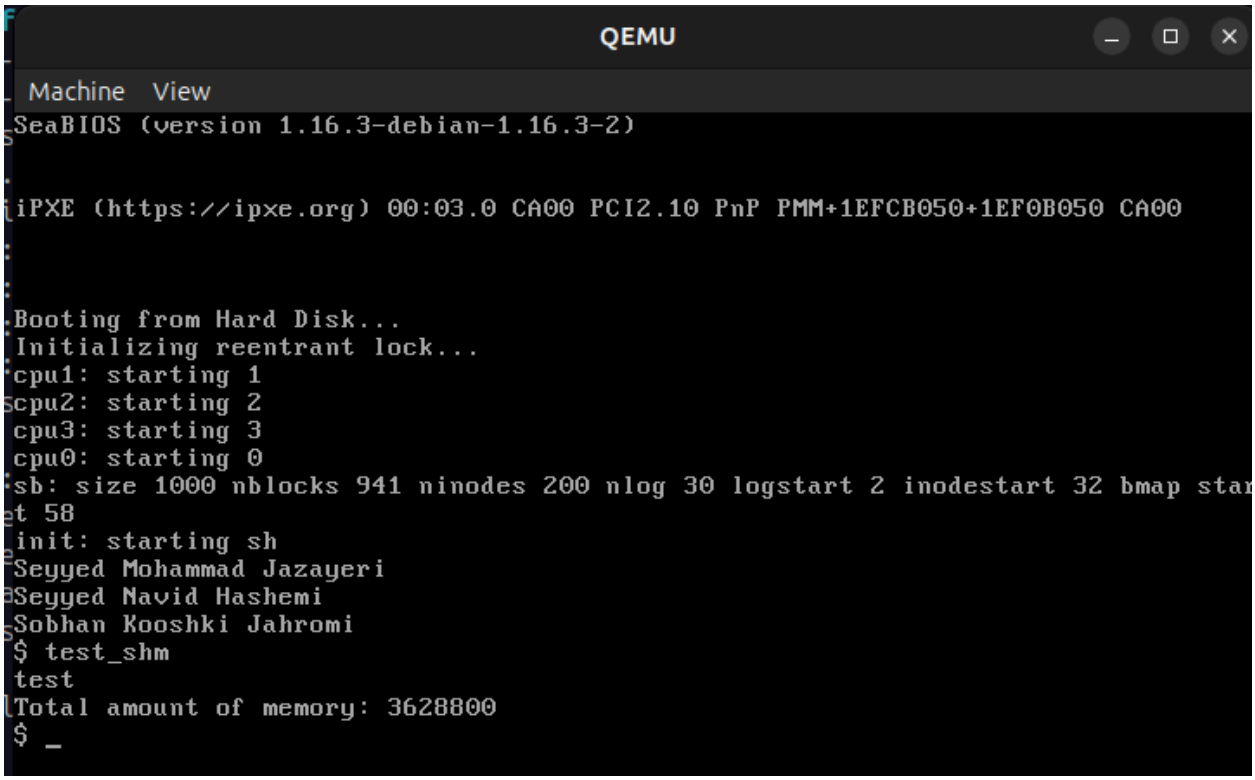
حالا باید یک برنامه سطح کاربر بنویسیم و اینکار را امتحان کنیم.

```c
int main(int argc, char *argv[]) {
    int shared_mem_id = get_shared_mem(SHM_KEY, sizeof(int), 0);
    if (shared_mem_id < 0) {
        shared_mem_id = get_shared_mem(SHM_KEY, sizeof(int), 06 | IPC_CREAT);
        if (shared_mem_id < 0) {
            printf(1, "Failed to create shared memory segment\n");
            exit();
        }
        int *shared_mem_ptr = (int *)attach_shared_mem(shared_mem_id, 0, 0);
        if ((int)shared_mem_ptr < 0) {
            printf(1, "Failed to attach shared memory segment\n");
            exit();
        }
        *shared_mem_ptr = 0;
        deattach_shared_mem(shared_mem_ptr);
    }
    for (int i = 0; i < NUM_CHILD_PROCESSES; i++) {
        int pid = fork();
        if (pid < 0) {
            printf(1, "Fork failed\n");
            exit();
        } else if (pid == 0) {
            int child_shared_mem_id = get_shared_mem(SHM_KEY, sizeof(int), 0);
            if (child_shared_mem_id < 0) {
                printf(1, "Failed to get shared memory segment\n");
                exit();
            }
            int *child_shared_mem_Ptr = (int *)attach_shared_mem(child_shared_mem_id, 0, 0);
            if ((int)child_shared_mem_Ptr < 0) {
                printf(1, "Failed to attach shared memory segment\n");
                exit();
            }
            if(i != 0)
                *child_shared_mem_Ptr *= i + 1;
            else
                *child_shared_mem_Ptr = 1;
            deattach_shared_mem(child_shared_mem_Ptr);
            exit();
        }
    }
    for (int i = 0; i < NUM_CHILD_PROCESSES; i++) {
        wait();
    }
    int *parent_shared_mem_Ptr = (int *)attach_shared_mem(shared_mem_id, 0, 0);
    if ((int)parent_shared_mem_Ptr < 0) {
        printf(1, "Failed to attach shared memory segment\n");
        exit();
    }
    printf(1, "Total amount of memory: %d\n", *parent_shared_mem_Ptr);
    deattach_shared_mem(parent_shared_mem_Ptr);

    remove_shared_mem(shared_mem_id, IPC_RMID, 0);

    exit();
```

اول کار میآییم ان shared memory را بدست میآوریم برای اینکار از دستور گت استفاده میکنیم که یکی از ان ماکرو های shared را به ما میدهد و و اگر میتوانستیم یک پوینتر به ما میدهد

سپس در ادامه در یک for ما میاییم بچه هایی میسازیم و fork میکنیم ان ها میروند و یک shared memory را میگیرند سپس کار خود را که همان محاسبه فاکتوریل است را انجام میدهند و در ادامه ان حافظه را ول میکنند

در ادامه parent ان ها برای بچه ها wait میکند تا ان ها کارشان تمام شود سپس خودش ان حافظه را attach میکند تا بتواند نتیجه را بخواند و ان را پرینت میکند وقتی کار خود را انجام داد ان حافظه را deattach میکند و سپس ان را با استفاده از remove به ماکرو ها میدهد.



اینم نتیجه کار.