«گزارش کار پروژه 5 آزمایشگاه سیستم های عامل»

کسری کاشانی نژاد 810101490

البرز محمودیان 810101514

نرگس بابالار810101557

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

در لینوکس، مفهوم "ناحیه حافظه مجازی (Virtual Memory Area) "به فضایی اشاره دارد که برای هر فرآیند به صورت مجزا تخصیص داده می‌شود. در این روش، برنامه‌ها با آدرس‌های مجازی کار می‌کنند که توسط سیستم‌عامل و سخت‌افزار به آدرس‌های فیزیکی تبدیل می‌شوند. این رویکرد مزایای متعددی دارد:

1. ایزوله‌سازی فرآیندها :هر فرآیند فضای آدرس مجازی مستقل دارد، بنابراین یک فرآیند نمی‌تواند به داده‌های فرآیند دیگر دسترسی پیدا کند.
2. ساده‌سازی رابط برنامه‌نویسی :فرآیندها تصور می‌کنند که حافظه‌ای پیوسته و بزرگ در اختیار دارند، در حالی که این حافظه ممکن است به صورت تکه‌تکه و پراکنده در حافظه فیزیکی قرار گرفته باشد.
3. استفاده کارآمدتر از حافظه :با استفاده از مکانیزم‌هایی مانند صفحه‌بندی (Paging) و تعویض(Swapping) ، می‌توان برنامه‌ها را حتی در صورتی که کل حافظه آن‌ها در RAMجا نشود اجرا کرد.

مقایسه با:xv6

xv6نیز از مفهوم آدرس‌دهی مجازی استفاده می‌کند، اما به دلیل ساده بودن این سیستم‌عامل، قابلیت‌های آن نسبت به لینوکس محدودتر است. در:xv6

- آدرس‌دهی مجازی و صفحه‌بندی استفاده می‌شود، اما مدیریت پیشرفته‌ای مانند swapping یا تخصیص پویا به پیچیدگی لینوکس وجود ندارد.
- فضای آدرس مجازی فرآیندها ساده‌تر و معمولا در دو بخش اصلی (کاربر و هسته) تقسیم می‌شود.
- ساختار جدول صفحه در xv6 کمتر پیچیده است و عمدتاً برای آموزش و درک اصول اولیه طراحی سیستم‌عامل طراحی شده است.

لینوکس در این زمینه به مراتب پیچیده‌تر و کاربردی‌تر است و قابلیت‌هایی مانند ASLR (تصادفی‌سازی فضای آدرس) و مدیریت حافظه پویا را ارائه می‌دهد که در xv6 وجود ندارند.

سوال 2:چرا ساختار سلسله‌مراتبی منجر به کاهش مصرف حافظه می‌گردد؟

ساختار سلسله‌مراتبی (Hierarchical Structure) در جدول صفحه (Page Table) به دلیل سازماندهی داده‌ها به صورت سلسله‌مراتبی، مصرف حافظه را کاهش می‌دهد. این روش به

جای ذخیره تمام اطلاعات مربوط به نگاشت آدرس‌ها در یک جدول بزرگ، اطلاعات را در چند جدول کوچک‌تر و مرتبط ذخیره می‌کند. این مزیت‌ها شامل موارد زیر است:

1. **صرفه‌جویی در حافظه** :نیازی به تخصیص یک جدول صفحه کامل برای هر فرآیند نیست، بلکه فقط جداولی که به طور فعال استفاده می‌شوند تخصیص داده می‌شوند.
2. **مدیریت پویا :**بخش‌هایی از جدول که استفاده نمی‌شوند، نیازی به ذخیره یا نگهداری ندارند، که این باعث کاهش مصرف حافظه می‌شود.

سوال 3:محتوای هر بیت (32 بیت) در هر سطح چیست؟ چه تفاوتی میان آن‌ها وجود دارد؟

محتوای هر ورودی (Entry) در هر سطح جدول سلسله‌مراتبی شامل اطلاعات زیر است:

1. P (Present): نشان‌دهنده این است که صفحه در حافظه اصلی قرار دارد یا خیر.
2. W (Writable): مشخص می‌کند که صفحه قابل نوشتن است یا فقط خواندنی.
3. U (User): نشان‌دهنده این است که آیا این صفحه برای فرآیندهای سطح کاربر قابل دسترسی است یا فقط برای هسته.
4. WT (Write-through): نحوه مدیریت کش را تعیین می‌کند.
5. CD (Cache Disabled): مشخص می‌کند که آیا این صفحه از کش استفاده می‌کند یا خیر.
6. A (Accessed): نشان می‌دهد که آیا صفحه دسترسی شده است.
7. D (Dirty): برای صفحه‌هایی که تغییر داده شده‌اند استفاده می‌شود.
8. AVL: بیت‌های رزرو شده برای استفاده در آینده یا ویژگی‌های خاص سیستم‌عامل.

تفاوت میان سطوح:

- در سطح بالاتر(Page Directory) ، ورودی‌ها به جدول‌های صفحه پایین‌تر اشاره می‌کنند.
- در سطح پایین‌تر(Page Table) ، ورودی‌ها مستقیماً به آدرس فیزیکی صفحات حافظه اشاره می‌کنند.

سطح بالاتر به عنوان یک ایندکس یا نقشه برای دسترسی سریع‌تر عمل می‌کند، در حالی که سطح پایین‌تر نگاشت دقیق‌تری از آدرس‌های مجازی به فیزیکی ارائه می‌دهد.

به شکل دقیق تر :

ساختار بیت‌ها در هر ورودی:

1. **بیت 0:(P - Present)**
   - اگر این بیت مقدار 1 داشته باشد، نشان می‌دهد که صفحه در حافظه اصلی (RAM) وجود دارد.
   - اگر مقدار 0 باشد، صفحه در حافظه اصلی نیست (مثلاً در دیسک ذخیره شده یا هنوز تخصیص داده نشده است).
2. **بیت 1:(W - Writable)**
   - اگر مقدار 1 باشد، صفحه قابل نوشتن است.
   - اگر مقدار 0 باشد، صفحه فقط خواندنی است.
3. **بیت 2:(U - User)**
   - اگر مقدار 1 باشد، صفحه برای کدهای سطح کاربر قابل دسترسی است.
   - اگر مقدار 0 باشد، فقط کدهای سطح هسته به آن دسترسی دارند.
4. **بیت 3:(WT - Write-through)**
   - مشخص می‌کند که آیا کش برای این صفحه به صورت Write-through کار می‌کند یا نه.
   - مقدار 0) Write-back :پیش‌فرض، عملکرد کش بهینه‌تر است.(
   - مقدار 1) Write-through :مستقیم روی حافظه اصلی می‌نویسد.(
5. **بیت 4:(CD - Cache Disabled)**
   - اگر مقدار 1 باشد، استفاده از کش برای این صفحه غیرفعال می‌شود.
   - اگر مقدار 0 باشد، کش فعال است.
6. **بیت 5:(A - Accessed)**
   - نشان می‌دهد که آیا این صفحه توسط پردازنده دسترسی پیدا کرده است یا خیر.
   - این بیت به‌طور خودکار توسط سخت‌افزار به‌روزرسانی می‌شود.

7. **بیت 6:(D - Dirty)**
   - این بیت زمانی مقدار 1 می‌شود که صفحه تغییر یافته باشد (مثلاً داده‌های جدید روی آن نوشته شده باشد).
   - برای مدیریت کش و نوشتن به دیسک استفاده می‌شود.
8. **بیت‌های 7 تا 8:(AVL - Available for System Use)**
   - برای استفاده‌های خاص سیستم‌عامل یا سخت‌افزار رزرو شده‌اند.
9. **بیت‌های 9 تا 11 (گسترده در بعضی معماری‌ها):**
   - این بیت‌ها ممکن است برای اهداف خاصی مانند ویژگی‌های اضافی کش استفاده شوند.
10. **بیت‌های 12 تا 31:**
    - این بیت‌ها حاوی **Physical Page Number (PPN)** هستند.
    - این عدد آدرس فیزیکی صفحه را مشخص می‌کند (ابتدای صفحه در حافظه فیزیکی).

سوال 4:تابع kallocچه نوع حافظه‌ای تخصیص می‌دهد؟ (فیزیکی یا مجازی)

تابع kallocحافظه فیزیکی را تخصیص می‌دهد. این تابع برای اختصاص دادن یک صفحه (Page) از حافظه فیزیکی استفاده می‌شود. این صفحات به طور معمول اندازه استانداردی برابر با اندازه یک صفحه (معمولاً 4 کیلوبایت) دارند و برای استفاده در نگاشت آدرس‌های مجازی به آدرس‌های فیزیکی به کار می‌روند.

سوال 5:تابع mappagesچه کاربردی دارد؟

تابع mappagesبرای نگاشت (Mapping) آدرس‌های مجازی به آدرس‌های فیزیکی در جدول صفحه استفاده می‌شود. این تابع ورودی‌هایی مانند آدرس مجازی، آدرس فیزیکی و تعداد صفحات را می‌گیرد و آن‌ها را در جدول صفحه قرار می‌دهد. کاربرد اصلی این تابع، مدیریت نگاشت حافظه در سیستم است، به گونه‌ای که پردازنده بتواند از آدرس‌های مجازی استفاده کند و آن‌ها را به آدرس‌های فیزیکی ترجمه کند.

سوال 7:راجع به تابع walkpgdirتوضیح دهید. این تابع چه عمل سخت‌افزاری را شبیه‌سازی می‌کند؟

تابع walkpgdirبرای یافتن ورودی‌های مرتبط با یک آدرس مجازی در جدول صفحه (Page Table)استفاده می‌شود. این تابع، آدرس مجازی داده شده را تجزیه می‌کند تا به ترتیب به ورودی‌های Page Directoryو سپس Page Tableدسترسی پیدا کند. اگر ورودی موردنظر در جدول صفحه موجود نباشد، می‌تواند یک ورودی جدید ایجاد کند.
عملکرد این تابع، شبیه‌سازی مکانیزم ترجمه آدرس در سخت‌افزار (MMU - Memory Management Unit)است که آدرس‌های مجازی را به آدرس‌های فیزیکی ترجمه می‌کند. این شبیه‌سازی به سیستم‌عامل کمک می‌کند تا بدون نیاز مستقیم به سخت‌افزار، جدول صفحه را مدیریت و اطلاعات لازم را ثبت کند.

سوال 8:توابع allocuvmو mappagesکه در ارتباط با حافظه‌ی مجازی هستند را توضیح دهید.

- **تابع allocuvm:**

این تابع برای تخصیص حافظه‌ی مجازی به یک فرآیند در فضای کاربر استفاده می‌شود. این تابع از حافظه فیزیکی برای ایجاد صفحات جدید استفاده کرده و آن‌ها را در فضای آدرس مجازی فرآیند نگاشت می‌کند. به‌طور خاص، این تابع از kallocبرای تخصیص حافظه‌ی فیزیکی بهره می‌برد و سپس از mappagesبرای نگاشت آدرس‌های مجازی به حافظه فیزیکی استفاده می‌کند.

- **تابع mappages:**

این تابع به طور مستقیم برای ایجاد نگاشت (Mapping) بین آدرس مجازی و آدرس فیزیکی در جدول صفحه استفاده می‌شود. وظیفه اصلی این تابع، تنظیم ورودی‌های جدول صفحه (Page Table)برای آدرس‌دهی حافظه است. این کار شامل مشخص کردن ویژگی‌هایی مثل بیت‌های Present، Writable، و Userنیز می‌شود.

ارتباط:

این دو تابع در کنار هم برای مدیریت و نگاشت حافظه فرآیندها استفاده می‌شوند؛ به‌طوری که allocuvmحافظه را تخصیص می‌دهد و mappagesوظیفه نگاشت دقیق را بر عهده دارد.

سوال 9:شیوه‌ی بارگذاری برنامه در حافظه توسط فراخوانی سیستمی execرا شرح دهید.

فراخوانی سیستمی execبرای جایگزینی فضای آدرس فعلی یک فرآیند با فضای آدرس برنامه جدید استفاده می‌شود. فرآیند اجرای execشامل مراحل زیر است:

1. **باز کردن فایل اجرایی مشخص‌شده توسط path:**
فایل اجرایی (Executable File) مشخص‌شده توسط مسیر (path) باز می‌شود و بررسی می‌شود تا اطمینان حاصل شود که یک فایل اجرایی معتبر است.
2. **خواندن هدر:ELF**
هدر ELF (Executable and Linkable Format) فایل خوانده می‌شود تا اطلاعات مربوط به سگمنت‌ها و سکشن‌های برنامه به‌دست آید. این اطلاعات مشخص می‌کند که کدام بخش از برنامه باید در حافظه بارگذاری شود.
3. **آزادسازی فضای آدرس قبلی:**
فضای آدرس فعلی فرآیند شامل Page Directory و جداول صفحه به‌طور کامل تخریب و آزاد می‌شود تا فضای کافی برای برنامه جدید فراهم شود.
4. **کپی کردن سگمنت‌های مشخص‌شده در ELF به حافظه:**
   - هر سگمنت برنامه مشخص‌شده در فایل ELF به ترتیب در حافظه کپی می‌شود.
   - برای هر سگمنت، با استفاده از تابع allocuvm، آدرس فیزیکی حافظه تخصیص داده می‌شود.
   - داده‌های مربوط به سگمنت‌ها از فایل اجرایی به حافظه منتقل می‌شوند.
5. **ایجاد پشته فرآیند جدید:**
   - یک پشته جدید برای فرآیند ایجاد می‌شود.
   - آرگومان‌های برنامه (argv) و مقادیر محیطی (Environment Variables) روی پشته قرار داده می‌شوند.

6. **تنظیم رجیسترهای CPU:**
   - رجیسترهای CPU برای شروع اجرای برنامه جدید مقداردهی می‌شوند.
   - Program Counter (PC) روی نقطه ورود (Entry Point) برنامه تنظیم می‌شود.
7. **آزادسازی منابع اضافی:**
   - منابعی که دیگر موردنیاز نیستند، مانند فایل‌های موقت یا حافظه استفاده نشده، آزاد می‌شوند.
8. **شروع اجرای برنامه جدید:**
   کنترل پردازنده به برنامه جدید منتقل می‌شود و اجرای آن آغاز می‌شود.

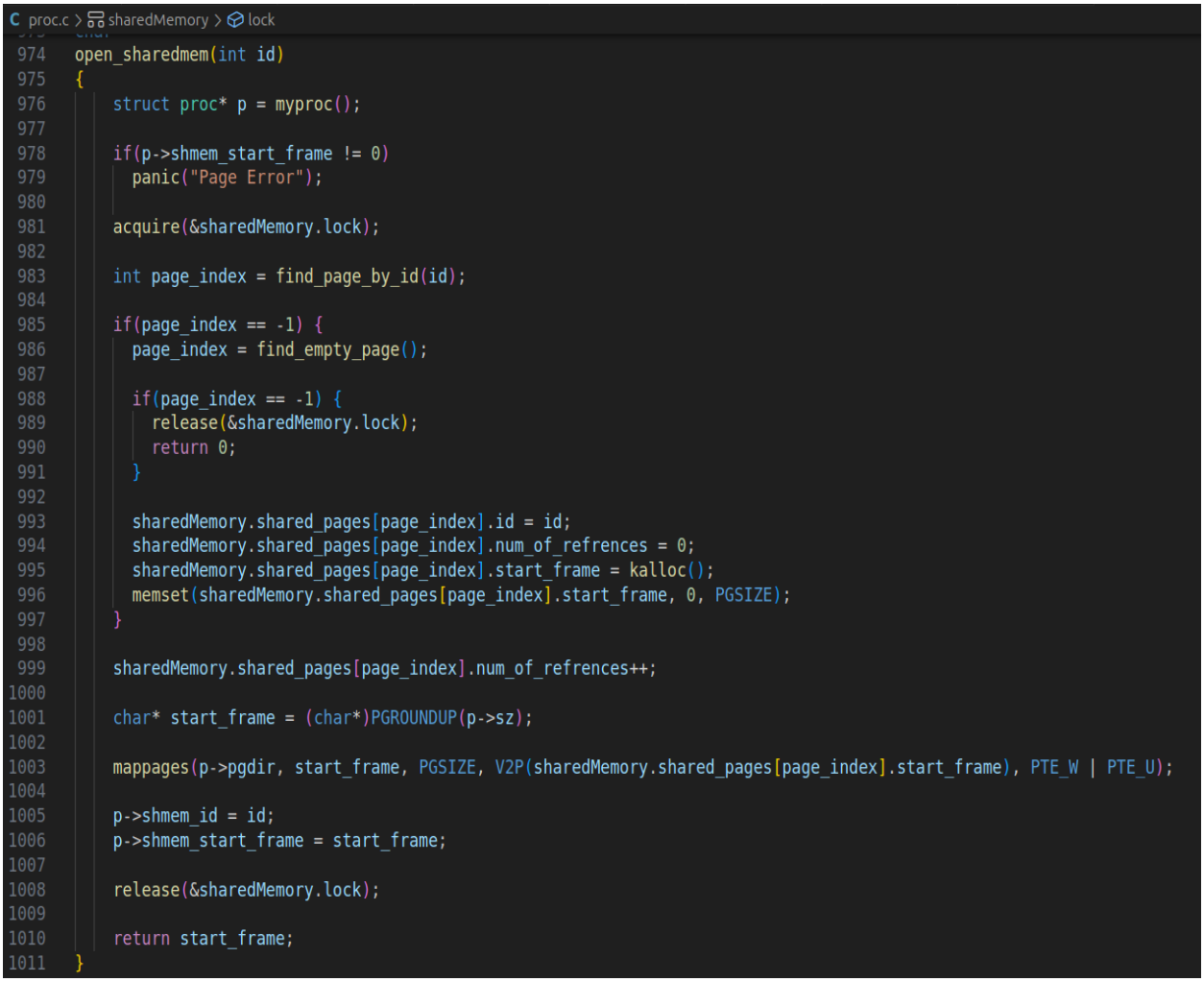
این فرآیند به طور کامل فضای آدرس قبلی فرآیند را با فضای آدرس جدید جایگزین کرده و برنامه جدید را برای اجرا آماده می‌کند. این عملکرد یکی از حیاتی‌ترین بخش‌های سیستم‌عامل است که امکان اجرای برنامه‌های مختلف را فراهم می‌کند.

- حال نحوه پیاده سازی سیستم کال های خواسته شده را توضیح می دهیم:

ابتدا استراکچر اصلی یعنی sharedMemory را می سازیم. این استراکچر شامل یک قفل برای مدیریت دسترسی همزمان چند پراسس به جدول و یک لیستی از sharedPage ها با حداکثر سایز برابر با MAX_SHARED_PAGES که در memlayout.h برابر با 8 تعریف کردیم می باشد. این استراکچر sharedPage شامل id هر صفحه، تعداد refrence های به آن و پوینتری به فریم فیزیکی شروع صفحه می باشد.

```
25  struct sharedPage {
26    int id;
27    int num_of_refrences;
28    char* start_frame;
29  };
30
31  struct sharedMemory {
32    struct sharedPage shared_pages[MAX_SHARED_PAGES];
33    struct spinlock lock;
34  } sharedMemory;
35
```

حال سیستم کال open_sharedmem را پیاده سازی می کنیم. بدین صورت که درصورت وجود داشتن صفحه با آن id، فقط تعداد refrence هایش را ++ می کند و mappages را انجام می دهد. اما در صورت وجود نداشتن آن id، ابتدا یک صفحه با تعداد refrence های برابر با 0 پیدا می کنیم و آن صفحه را به آن پراسس اختصاص می دهیم و سپس mappages را انجام می دهیم.



```c
proc.c > sharedMemory > lock
open_sharedmem(int id)
{
    struct proc* p = myproc();

    if(p->shmem_start_frame != 0)
      panic("Page Error");

    acquire(&sharedMemory.lock);

    int page_index = find_page_by_id(id);

    if(page_index == -1) {
      page_index = find_empty_page();

      if(page_index == -1) {
        release(&sharedMemory.lock);
        return 0;
      }

      sharedMemory.shared_pages[page_index].id = id;
      sharedMemory.shared_pages[page_index].num_of_refrences = 0;
      sharedMemory.shared_pages[page_index].start_frame = kalloc();
      memset(sharedMemory.shared_pages[page_index].start_frame, 0, PGSIZE);
    }

    sharedMemory.shared_pages[page_index].num_of_refrences++;

    char* start_frame = (char*)PGROUNDUP(p->sz);

    mappages(p->pgdir, start_frame, PGSIZE, V2P(sharedMemory.shared_pages[page_index].start_frame), PTE_W | PTE_U);

    p->shmem_id = id;
    p->shmem_start_frame = start_frame;

    release(&sharedMemory.lock);

    return start_frame;
}
```

حال برای پیاده سازی سیستم کال close_sharedmem، ابتدا صفحه با آن id را پیدا می کنیم و اگر وجود نداشت، ارور می دهد. سپس در صورت وجود آن صفحه، ابتدا تعداد refrence هایش را - - می کند و حافظه مجازی که به آن صفحه اختصاص داده شده بود را از page table حذف می کنیم. در انتها نیز در صورت 0 شدن تعداد refrence های آن صفحه، حافظه ی آن را آزاد می کنیم.

```
1013  int
1014  close_sharedmem(int id)
1015  {
1016      struct proc* p = myproc();
1017
1018      if (p->shmem_id != id || p->shmem_start_frame == 0)
1019          panic("Illegal access to this page\n");
1020
1021      acquire(&sharedMemory.lock);
1022
1023      int page_index = find_page_by_id(id);
1024
1025      if(page_index == -1) {
1026        release(&sharedMemory.lock);
1027        cprintf("The page id doesnt exist!\n");
1028        return 0;
1029      }
1030
1031      sharedMemory.shared_pages[page_index].num_of_refrences--;
1032
1033      p->shmem_start_frame = 0;
1034      p->shmem_id = -1;
1035
1036      uint a = PGROUNDUP((uint)p->shmem_start_frame);
1037      pte_t* pte = walkpgdir(p->pgdir, (char*)a, 0);
1038      if(!pte)
1039          a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
1040      else if((*pte & PTE_P) != 0){
1041          *pte = 0;
1042      }
1043
1044      if (sharedMemory.shared_pages[page_index].num_of_refrences == 0) {
1045          kfree(sharedMemory.shared_pages[page_index].start_frame);
1046      }
1047
1048      release(&sharedMemory.lock);
1049
1050    return 0;
1051  }
```

همچنین یک تابع init_sharedmem را نیز تعریف کرده و در main.c اجرایش می کنیم تا در ابتدا، تعداد refrence های تمامی صفحات sharedMemory را برابر با 0 قرار دهد.

```
1053  int
1054  init_sharedmem()
1055  {
1056      acquire(&sharedMemory.lock);
1057
1058      for (int i = 0; i < MAX_SHARED_PAGES; i++){
1059          sharedMemory.shared_pages[i].num_of_refrences = 0;
1060      }
1061
1062      release(&sharedMemory.lock);
1063
1064      return 0;
1065  }
```

در انتها نیز یک برنامه سطح کاربر برای تست این حافظه ی اشتراکی نوشتیم که ابتدا پراسس والد، یک صفحه با id برابر با 1 درست می کند و مقدار 1 را در آن قرار می دهد. سپس 5 پراسس فرزند ایجاد می کند و هرکدام نیز به طور مستقل و موازی، به کمک دو سیستم کال زده شده، مقدار همان حافظه ی مشترک را در شماره ی آن پراسس به اضافه 1 ضرب می کند تا در این صورت مقدار فاکتوریل شماره ی آن پراسس به اضافه 1 محاسبه شود. در انتها نیز با نمایش آن مقدار در خانه ی حافظه مشترک پراسس والد، باید مشاهده شود که مقدار برابر با فاکتوریل تعداد پراسس های فرزند یعنی 5! = 120 می باشد.

test_sharedmem.c > main()

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

#define NUM_OF_CHILDS 5

int main()
{
    int *shared_mem_parent = (int*)open_sharedmem(1);
    shared_mem_parent[0] = 1;

    for(int i = 0; i < NUM_OF_CHILDS; i++) {
        int pid = fork();

        if(pid == 0) {
            int *shared_mem_child = (int*)open_sharedmem(1);
            int num = shared_mem_child[0] * (i + 1);
            shared_mem_child[0] = num;

            printf(1, "Child: factorial of %d is %d\n", i + 1, num);

            close_sharedmem(1);

            exit();
        }
    }

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

    printf(1, "Parent: factorial of %d is %d\n", NUM_OF_CHILDS, shared_mem_parent[0]);

    close_sharedmem(1);

    exit();
}
```

همچنین نتیجه ی اجرای این برنامه به صورت زیر است. همانطور که انتظار می رفت، مقدار نهایی داخل shared memory که پراسس والد آن را پرینت کرده است، برابر با فاکتوریل تعداد پراسس های فرزند یعنی 120 = !5 می باشد.