گزارش پروژهٔ پنجم آزمایشگاه سیستم عامل

به تدریس دکتر کارگهی

محمد امین یوسفی

مبینا مهرآذر

متین نبی‌زاده

زمستان 1402

اطلاعات مربوط به گیت‌هاب، در آخرین صفحهٔ گزارش، درج شده است.

مقدمه

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

مفهوم ناحیهٔ مجازی[1] در سیستم عامل لینوکس، برای مدیریت فضای آدرس مجازی مربوط به پردازه‌ها استفاده می‌شود. در واقع این فضا به چندین زیرفضا تقسیم می‌شود که هرکدام یک محدودهٔ پیوسته از حافظهٔ مجازی با مشخصهٔ منحصر به فردی را نمایان می‌کنند. هر فضا، یک آدرس شروع و یک طول دارد و اندازهٔ هر کدام نیز مضربی از اندازهٔ صفحات می‌باشد. خوانندهٔ محترم در نظر دارد که پیوستگی بین فضاها از نوع فیزیکی نبوده و از مجازی است.
بر خلاف سیستم‌عامل لینوکس، در سیستم عامل xv6، مفهوم ناحیهٔ مجازی معنایی ندارد بلکه جدول صفحهٔ دوسطحی‌ای وجود دارد که برای مدیریت تخصیص حافظه استفاده می‌شود. نگاشت حافظه فیزیکی قابل استفاده توسط پردازه‌ها، به کمک تابع kpgdir در فضای آدرس مجازی صورت می‌گیرد. به کمک این نگاشت صورت گرفته، همه‌ی حافظه به کمک آدرس‌های مجازی تعریف‌شده، آدرس‌دهی می‌شوند. این نگاشت آدرس هم توسط واحد سخت‌افزاری MMU یا همان واحد مدیریت حافظه، ترجمه می‌شود. این ساز و کار امکان تصور فضای آدرسی بزرگ‌تر از حافظهٔ فیزیکی موجود را برای پردازه‌ها فراهم می‌سازد.

به طور خلاصه، لینوکس از مفهومی به نام VMA برای مدیریت فضای حافظهٔ پردازه‌ها استفاده می‌کند ولی در سیستم‌عامل xv6، از جدول صفحهٔ دوسطحی استفاده می‌شود.

[1] Virtual Memory Area (VMA)

2) چرا ساختار سلسله‌مراتبی منجر به کاهش مصرف حافظه می‌گردد؟

پیش از بحث و بررسی بهینگی‌ای که ساختار سلسله‌مراتبی حافظه برای سیستم به ارمغان می‌آورد، بایستی یک تعریف اجمالی از ساختار سلسله‌مراتبی حافظه در ذهن، داشته باشیم:

- **طراحی سلسله‌مراتب‌وار[2] حافظه به تفکیک انواع مختلف آن بر اساس سرعت، ظرفیت و هزینه، با هدف تسریع دسترسی به داده‌ها اشاره دارد؛ به گونه‌ای که ضمن اولویت‌بندی مذکور، حافظهٔ کوچک‌تر و سریع‌تر، در نظر گرفته می‌شود برای اطلاعاتی که اغلب مورد استفاده قرار می‌گیرند، در حالی که ظرفیت‌های ذخیره‌سازی بزرگ‌تر برای داده‌هایی که کمتر در دسترس هستند، در نظر گرفته می‌شوند.**

با درک معنای طراحی سلسله‌مراتب‌وار حافظه، نقش محوری آن را در افزایش پاسخگویی، کارایی و عملکرد کلی سیستم، نمایان می‌گردد. این ساختار، به طور خودکار دسترسی به داده‌ها را مدیریت می کند و از اصل محلی بودن[3] استفاده می کند تا اطمینان حاصل کند که CPU به سرعت، اطلاعات مورد نیاز خود را بازیابی می کند و تاخیر[4]ها و گلوگاه[5]ها را کاهش می دهد. در اصل، سلسله‌مراتب حافظه فقط یک آرایش سلسله‌مراتبی مختص به انواع حافظه نیست؛ بلکه یک معماری است که استفاده از منابع، متعادل کردن سرعت، ظرفیت و هزینه را برای ارائهٔ محاسبات، بهینه می کند.

به طور خاص، در سیستم‌عامل xv6، وجود سلسله‌مراتب دو سطحی از آن جهت مفید است که وجود یک جدول صفحه، موجب اجتناب از وجود کل صفحهٔ جدول صفحات و در نتیجه، موجب صرفه‌جویی در مصرف و تخصیص حافظه، می‌گردد.[6]

---

[2] Hierarchical
[3] Localization
[4] Latency
[5] Bottleneck
[6] R. Cox, F. Kaashoe and R. Morris, *xv6: a simple, Unix-like teaching operating system*, MIT, Page 29

3) محتوای هر بیت یک مدخلِ 32 بیتی در هر سطح چیست؟ چه تفاوتی میان آنها وجود دارد؟

- در جدول دوسطحی موجود در xv6، هر آدرس فیزیکی، به سه بخش زیر تقسیم می‌شود:

- **یک آفست در صفحه**
- **یک اندیس در جدول صفحه**
- **یک اندیس در دایرکتوری صفحه**

12 بیت، مربوط به آفست است. از آنجایی که اندازهٔ هر صفحه عموما 4KB می‌باشد، این آفست توسط 12 بیت کم ارزش آدرس مجازی مشخص می‌شود. 10 بیت از آدرس، اندیس نشان‌دهندهٔ جدول صفحه، یک مدخل از آرایهٔ یک بعدی جدول صفحه را نشانه‌گذاری می‌کند.همچنین 10 بیت از آدرس، اندیس نشان‌گذارندهٔ دایرکتوری صفحه را مشخص می‌کند. آرایهٔ تک بعدی‌ای که حکم دایرکتوری صفحه را دارد، از چندین مدخل تشکیل شده است که که هر کدام از این مدخل‌ها، به یک جدول صفحه اشاره می‌کنند.

در مجموع، از 32 بیت مربوط به آدرس مجازی، 12 بیت کم ارزش مربوط به آفست، 10 بیت بعدی نمایانگر اندیس جدول صفحه و 10 بیت نهایی نیز نمایانگر اندیس دایرکتوری صفحه می‌باشد.

مدخل‌های مربوط به دایرکتوری صفحه و جدول صفحه، تا حدودی مشابه می‌باشند و تفاوت در با ارزش بودن پرچم بیت کثیف[7] در دایرکتوری صفحه می‌باشد. این پرچم، مشخص می‌کند که برای اعمال تغییرات، صفحات مورد نظر بایستی در دیسک نوشته شوند.

[7] Dirty Bit

مدیریت حافظه در xv6

4) تابع ()kalloc چه نوع حافظه‌ای تخصیص می‌دهد؟ (فیزیکی یا مجازی)

به دور از اطناب کلام، پاسخ این سؤال در یک جمله، آن است که هستهٔ سیستم‌عامل xv6، توابع ()kalloc و ()kfree را به ترتیب برای تخصیص و آزادسازی **حافظهٔ فیزیکی** در زمان اجرا، فراخوانی می‌کند. لازم به ذکر است که این هسته، برای تخصیص حافظه به داده‌ساختارهای به خصوصی (از جمله پشته[8]ها، ثبات‌های مختص به لوله‌ها[9] و جدول صفحه[10])، در زمان اجرا اقدام می‌کند. همچنین، تخصیص‌دهندهٔ مذکور یک فهرست از صفحات آزاد حافظهٔ فیزیکی، نگهداری می‌کند که در زمان لزوم، آمادهٔ تخصیص داده شدن هستند.

5) تابع ()mappages چه کاربردی دارد؟

پیش از بررسی کاربرد این تابع، خالی از لطف نیست اگر به فرآیند مقداردهی جدول صفحه‌ها رجوع کنیم. در ابتدای امر، تابع mainc (موجود در خط 1354) یک جدول صفحه برای استفادهٔ هسته با فراخوانی به تابع kvmalloc ایجاد می کند و تابع mpmain نیز موجب می‌شود تا سخت‌افزار صفحه‌بندی معماری x86، ضمن فراخوانی تابع vmenable، شروع کند به استفاده از جدول صفحهٔ به تازگی ایجاد شده. همچنین، می‌دانیم که جدول صفحهٔ موجود در معماری x86، در حافظهٔ فیزیکی و به صورت یک دایرکتوری 4096-بایتی ذخیره می‌شود که هر جدول صفحه، در اصل آرایه‌ای از 1024 مدخل 32-بیتی است. از طرف دیگر، سخت‌افزار صفحه‌بندی به منظور انتخاب یک مدخل از دایرکتوری صفحه، از 10 بیتِ پرارزش یک آدرس مجازی استفاده می کند. در صورتی که مدخلی خاص از دایرکتوری صفحه با پرچم PTE_P علامت‌گذاری شده باشد، سخت‌افزار صفحه‌بندی از 10 بیت بعدی استفاده می‌کند تا مدخلی از جدول صفحه را بیابد که مدخل دایرکتوری صفحه به آن، اشاره می‌کند. همچنین، به عنوان یک حالت خاص، اگر مدخل دایرکتوری صفحه یا مدخل جدول صفحه شامل هیچ‌گونه پرچم PTE_P نباشند، سخت‌افزار صفحه‌بندی یک خطا[11] را گزارش می‌کند. در این جا است که نوبت به تابع mappages می‌رسد تا در یک جدول صفحه، برای محدوده‌ای از آدرس مجازی (که متناظر با محدودهٔ مشخصی از فضای حافظهٔ فیزیکی هستند) نگاشت‌هایی را ایجاد کند. همان‌طور که در کُد پیاده‌سازی

---

[8] Stack
[9] Pipe Buffers
[10] Page Table
[11] Fault

این تابع می‌بینید، این تابع به ازای هر آدرس مجازی‌ای که قرار است نگاشته شود، تابع walkpagedir را فراخوانی می‌کند تا آدرس آن مدخلی از جدول صفحه که قرار است ترجمه (نگاشت) این آدرس باشد را بیابد. در نهایت نیز مدخل مذکور از جدول صفحه را با آدرس فیزیکی مربوطه، اختیارات مورد نیاز (از جمله PTE_W و/یا PTE_U) و PTE_P، مقداردهی کرده تا این مدخل جدول صفحه، معتبر شناخته شود[12].

```
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;
}
```

[12] R. Cox, F. Kaashoe and R. Morris, *xv6: a simple, Unix-like teaching operating system*, MIT

6) راجع به تابع walkpgdir() توضیح دهید. این تابع چه عملیات سخت‌افزاری‌ای را شبیه‌سازی می‌کند؟[13]

تابع walkpagedir، در اصل نقش سخت‌افزار صفحه‌بندی را بازی می‌کند؛ بدین صورت که میان مداخل جدول صفحه برای یافتن یک آدرس مجازی به خصوص، جستجو می‌کند و پس از یافتن آن، از 10 بیت پرارزش این آدرس مجازی، استفاده می‌کند تا مدخل دایرکتوری صفحه را بیابد. در صورتی که مدخل دایرکتوری صفحه (که در قسمت پیشین بدست آمده است)، یک مدخل معتبر نباشد، بدان معناست که صفحهٔ جداول صفحه‌ها[14] مورد نیاز هنوز ساخته نشده است. در صورتی که پرچم create، تعیین شده باشد، خود تابع walkpagedir دست به کار شده و جدول صفحهٔ ناموجود را می‌سازد. در نهایت نیز با استفاده از 10 بیت بعدی آدرس مجازی، آدرس مدخل جدول صفحه را در صفحهٔ جدول صفحات، بیابد. لازم به ذکر است که کُد پیاده‌سازی این قسمت، از آدرس فیزیکی موجود در مداخل دایرکتوری صفحه به عنوان آدرس مجازی استفاده می‌کند؛ زیرا هسته، صفحات دایرکتوری صفحه[15] و صفحات جدول صفحه را از ناحیه‌ای از حافظهٔ فیزیکی تخصیص می‌دهد که برای آن، نگاشت‌هایِ از مجازی به فیزیکیِ مستقیم دارد[16].

---

[13] خوانندهٔ محترم در نظر دارد که صورت فعلی پروژه (که این گزارش نیز بر مبنای آن نگاشته شده است)، از نبود سؤال شمارهٔ 6 رنج برده (!) و از سؤال پنجم، به طور ناگهانی، به سؤال هفتم می‌پرد! لذا پاسخ به این سؤال (یعنی سؤال ششم موجود در این گزارش)، در اصل متناظر با سؤال هفتم صورت پروژه است.

[14] Page table page

[15] Page directory pages

[16] R. Cox, F. Kaashoe and R. Morris, *xv6: a simple, Unix-like teaching operating system*, MIT

7) توابع allocuvm و mappages[17] که در ارتباط با حافظهٔ مجازی هستند را توضیح دهید.

به طور خلاصه، می‌توان گفت که تابع allocuvm، مسئولیت تخصیص حافظهٔ مجازی سطح کاربر به یک پردازه را بر عهده دارد. همان‌طور که از دید خوانندهٔ محترم پنهان نیست، این تابع پارامتری به نام pgdir گرفته و با استفاده از آن، جدول صفحهٔ پردازهٔ مورد نظر را قفل می‌کند تا از دسترسی انحصاری به آن، اطمینان حاصل شود. در ادامه، ضمن پیمایش بر روی تعداد صفحهٔ مورد نیاز، با استفاده از تابع kalloc، به هر کدام، حافظهٔ فیزیکی اختصاص دهد. در ادامه، همین تابع، با فراخوانی تابع mappages، آدرس‌های فیزیکی تخصیص داده شده را به آدرس‌های مجازی، می‌نگارد.

همچنین، همان‌طور که پیش‌تر (در سؤال 5) پیرامون تابع mappages، توضیحات مفصلی داده شد، در اینجا نیز به طور مختصر، این نکته را مرور می‌کنیم که این تابع، در اصل به عنوان تابع کمکی allocuvm، به آن تابع، این امکان را می‌دهد تا آدرس‌های فیزیکی را به آدرس‌های مجازی، بنگارد.

```
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");
```

[17] متناظر با سؤال شمارهٔ 8 در صورت پروژه؛ به پاورقی ابتدای سؤال 6 وجوع شود.

```
      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;
}
```

8) شیوهٔ[18] بارگذاری برنامه در حافظه توسط فراخوانی سیستمی exec را شرح دهید.

در ابتدای امر، تعدادی متغیر تعریف می‌شود. در ادامه مقدار inode مربوط به ورودی از جنس char* به اسم path را به کمک تابع namei و در متغیر ip ذخیره می‌کنیم.
در شرط‌های تعریف شده در ادامه، چک می‌شود که فایل اجرایی، معتبر می‌باشد یا خیر. این کار با بررسی ELF header انجام می‌شود.
برای ایجاد جداول صفحه جدید مربوط به پردازه، از تابع setupkvm استفاده می‌شود. خروجی غیرصفر این تابع، نشان می‌دهد که این اختصاص جداول به درستی صورت گرفته است.
در ادامه کد لود کردن برنامه در حافظه را داریم که کد یک حلقه روی هدرهای برنامه یا همان سگمنت‌های مربوط به ELF می‌باشد که هر سگمنت در فضای حافظه‌ی هر برنامه لود می‌شود.
تابع readi موجود در شرط، هدر مربوط به برنامه را از ELF فایل خوانده و در داده‌ساختار ph ذخیره می‌کند. اگر سایز خروجی تابع readi با سایز ph یعنی سایز مورد انتظار یکسان نبود، به نشانه‌ی ارور به لیبل bd می‌پرد.
در ادامه روی حلقه فقط هدرهای برنامه‌ای که از نوع ELF_PROG_LOAD هستند اجرا می‌شود.
سپس چک می‌شود که سایز سگمنت در ممووری، از سایز فایل کوچک‌تر نباشد.
چک کردن اورفلو در حافظه مجازی آدرس، در ادامه به کمک شرط if صورت می‌گیرد.
در ادامه با اجرا شدن تابع allocuvm، حافظه را در جدول صفحه کاربر، اختصاص می‌دهد.
سپس چک می‌شود که آدرس مجازی، قابلیت وفق یافتن بر روی صفحه را دارند یا خیر.
کد تابع موجود در فایل exec.c به شرح زیر می‌باشد:

```
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;
```

[18] متناظر با سؤال شمارهٔ 9 در صورت پروژه؛ به پاورقی ابتدای سؤال 6 وجوع شود.

```
  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;

  // 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;

  // 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;

  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;

  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc],
strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
```

```
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(curproc->name, last,
sizeof(curproc->name));

  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;

 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

پیاده‌سازی پروژه

در ابتدا، طبق راهنمایی صورت پروژه، دو داده ساختار به نام‌های SHM_table و SHM_info ایجاد کرده و یک نمونه از آن را نیز در فایل proc.c ایجاد می‌کنیم.

- **SHM_info** : شناسهٔ یکتا، تعداد ارجاع‌ها و آدرس فریم فیزیکی مربوط به یک صفحه از حافظهٔ اشتراکی را در خود ذخیره می‌کند.
- **SHM_table** : اطلاعات مربوط به تمامی صفحات مربوط به حافظه اشتراکی را در قالب یک آرایه از SHM_infoها ذخیره می‌کند. همچنین یک قفل چرخشی[19] برای جلوگیری از ایجاد شرایط رقابت[20] در دسترسی به صفحات حافظهٔ اشتراکی ایجاد می‌کنیم.

```
1   #define SHM_TABLE_SIZE 16
2
3   struct SHM_info
4   {
5     int id;
6     uint ref_counter;
7     char* frame;
8   };
9
10  struct SHM_table
11  {
12    struct SHM_info shm_infos[SHM_TABLE_SIZE];
13    struct spinlock slk;
14  };
```

همچنین در داده‌ساختار proc در فایل proc.h، یک آرایه از uintها ایجاد می‌کنیم تا آدرس‌های مجازی صفحات اشتراکی مربوط به هر پردازه را در خود ذخیره کند.

```
87    uint shared_mem[SHM_TABLE_SIZE];                    // Shared memory
```

[19] Spinlock
[20] Race Condition

- برای مقداردهی اولیهٔ حافظهٔ اشتراکی، تابع shm_init را در فایل proc.c تعریف کرده و آن را در تابع main در فایل main.c فراخوانی می‌کنیم.

```
888   void shm_init(void)
889   {
890     initlock(&shm_table.slk, "spin lock");
891     for (int i = 0; i < SHM_TABLE_SIZE; i++)
892     {
893       shm_table.shm_infos[i].id = -1;
894       shm_table.shm_infos[i].frame = 0;
895     }
896   }
```

```
17  int
18  main(void)
19  {
20    kinit1(end, P2V(4*1024*1024)); // phys page allocator
21    kvmalloc();      // kernel page table
22    mpinit();        // detect other processors
23    lapicinit();     // interrupt controller
24    seginit();       // segment descriptors
25    picinit();       // disable pic
26    ioapicinit();    // another interrupt controller
27    consoleinit();   // console hardware
28    uartinit();      // serial port
29    pinit();         // process table
30    tvinit();        // trap vectors
31    binit();         // buffer cache
32    fileinit();      // file table
33    ideinit();       // disk
34    shm_init();      // shared memory
35    startothers();   // start other processors
36    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
37    userinit();      // first user process
38    mpmain();        // finish this processor's setup
39  }
```

- حال، دو فراخوانی سیستمی open_sharedmem و close_sharedmem را تعریف می‌کنیم.

```
37  #define SYS_open_sharedmem 34
38  #define SYS_close_sharedmem 35
```

```
138  extern void* sys_open_sharedmem(void);
139  extern int sys_close_sharedmem(void);
```

```
175  [SYS_open_sharedmem] sys_open_sharedmem,
176  [SYS_close_sharedmem] sys_close_sharedmem,
```

```
180  void*
181  sys_open_sharedmem(void)
182  {
183    int id;
184    argint(0, &id);
185    if (id < 0)
186    {
187      cprintf("Invalid arguments\n");
188      return (void*)-1;
189    }
190    return open_sharedmem(id);
191  }
192
193  void
194  sys_close_sharedmem(void)
195  {
196    int id;
197    argint(0, &id);
198    if (id < 0)
199    {
200      cprintf("Invalid arguments\n");
201      return;
202    }
203    close_sharedmem(id);
204  }
```

- سپس، تابع open_sharedmem را در فایل proc.c، به صورت زیر پیاده‌سازی می‌کنیم:

```
918   void* open_sharedmem(int id)
919   {
920     struct proc* proc = myproc();
921     pde_t *pgdir = proc->pgdir;
922     char* start_vm_addr = (char*)PGROUNDUP(proc->sz);
923
924     acquire(&shm_table.slk);
925
926     int index = find_shm_index(id);
927     if (index == -1)
928     {
929       int free_shm_index = find_free_shm();
930       if(free_shm_index > -1)
931       {
932         char* mem = kalloc();
933         memset(mem, 0, PGSIZE);
934         mappages(pgdir, start_vm_addr, PGSIZE, V2P(mem), PTE_W|PTE_U);
935         shm_table.shm_infos[free_shm_index].frame = mem;
936         shm_table.shm_infos[free_shm_index].ref_counter++;
937         shm_table.shm_infos[free_shm_index].id = id;
938         proc->shared_mem[free_shm_index] = (uint)start_vm_addr;
939       }
940       else
941       {
942         cprintf("No free shared memory\n");
943         release(&shm_table.slk);
944         return (void*)-1;
945       }
946     }
```

```
947     else
948     {
949       if (proc->shared_mem[index] != -1)
950       {
951         release(&shm_table.slk);
952         return (void*)-1;
953       }
954       mappages(pgdir, start_vm_addr, PGSIZE, V2P(shm_table.shm_infos[index].frame), PTE_W|PTE_U);
955       shm_table.shm_infos[index].ref_counter++;
956       proc->shared_mem[index] = (uint)start_vm_addr;
957     }
958     release(&shm_table.slk);
959     return (void*)start_vm_addr;
960   }
```

این تابع، در ابتدا وجود صفحه با شناسهٔ خواسته شده را بررسی می‌کند. در صورت وجود، اگر این پردازه از قبل این صفحه را در اختیار نداشت، صفحه را به حافظهٔ اشتراکی پردازه می‌افزاید و تعداد ارجاع‌ها به آن صفحه را یک عدد اضافه می‌کند. اما در صورتی که صفحه‌ای با شناسهٔ مورد نظر وجود نداشت، یک فریم فیزیکی را با آخرین آدرس مجازی استفاده شده در پردازه نگاشته کرده و این صفحه را به حافظهٔ اشتراکی پردازه می‌افزاییم.

توجه داشته باشید که آرایهٔ آدرس‌های مجازی هر پردازه در تابع allocproc با 1- مقداردهی اولیه شده است.

- سپس، تابع close_sharedmem را نیز در فایل proc.c پیاده‌سازی می‌کنیم.

```
962  void close_sharedmem(int id)
963  {
964    struct proc *proc = myproc();
965
966    acquire(&shm_table.slk);
967
968    int index = find_shm_index(id);
969    shm_table.shm_infos[index].ref_counter--;
970
971    if (index != -1 && proc->shared_mem[index] != -1)
972    {
973      uint a = PGROUNDUP(proc->shared_mem[index]);
974      pte_t *pte = walkpgdir(proc->pgdir, (char *)a, 0);
975
976      if (!pte)
977        a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
978      else if ((*pte & PTE_P) != 0)
979        *pte = 0;
980      proc->shared_mem[index] = -1;
981
982      if (shm_table.shm_infos[index].ref_counter == 0)
983      {
984        kfree(shm_table.shm_infos[index].frame);
985        shm_table.shm_infos[index].id = -1;
986        shm_table.shm_infos[index].frame = (void *)0;
987      }
988    }
989    release(&shm_table.slk);
990  }
```

- این تابع، پس از بررسی وجود صفحه در فضای اشتراکی پردازهٔ فراخواننده، حافظه را از فضای پردازه حذف می‌کند و در صورت نبود هیچ ارجاع به آن صفحه توسط پردازه‌های دیگر، صفحه فیزیکی و ورودی آن را در جدول فضای اشتراکی حذف می‌کند.

همچنین برای پیاده‌سازی توابع بالا از دو تابع کمکی به صورت زیر استفاده شده است.

```
898  int find_shm_index(int id)
899  {
900    for (int i = 0; i < SHM_TABLE_SIZE; i++)
901    {
902      if (shm_table.shm_infos[i].id == id)
903        return i;
904    }
905    return -1;
906  }
907
908  int find_free_shm()
909  {
910    for (int i = 0; i < SHM_TABLE_SIZE; i++)
911    {
912      if (shm_table.shm_infos[i].id == -1)
913        return i;
914    }
915    return -1;
916  }
```

- تابع find_shm_index با دریافت یک شناسه، اندیس مربوط به صفحه با شناسهٔ مورد نظر را در صورت وجود باز می‌گرداند.

- تابع find_free_shm اندیس اولین ورودی خالی جدول حافظهٔ اشتراکی را باز می‌گرداند.

برنامهٔ آزمون

- دو برنامه جهت آزمودن فراخوانی‌های سیستمی پیاده‌سازی شده، نوشته شده است که در دو صفحهٔ پیش رو، پیش چشم خوانندهٔ محترم قرار گرفته‌اند:

```c
int main(int argc, char *argv[])
{
    char *shared_mem = open_sharedmem(SHMID);
    *shared_mem = 0;
    printf(1, "Shared memories initial value in parent process(%d) is : %d\n\n", getpid(), *shared_mem);
    for (int i = 0; i < NUM_OF_PROCESSES; i++)
    {
        int pid = fork();
        if (pid == 0)
        {
            acquire_prioritylock();
            char *shared_mem = open_sharedmem(SHMID);
            *shared_mem += 1;
            printf(1, "Child process %d modified shared value. new value is: %d\n", getpid(), *shared_mem);
            release_prioritylock();
            close_sharedmem(SHMID);
            exit();
        }
        else if(pid < 0)
        {
            printf(1, "fork failed\n");
            exit();
        }
    }
    for (int i = 0; i < NUM_OF_PROCESSES; i++)
    {
        wait();
    }
    printf(1, "\nShared memories final value in parent process(%d) is : %d\n", getpid(), *shared_mem);
    close_sharedmem(SHMID);
    exit();
}
```

```c
int main(int argc, char *argv[])
{
    char *sharedMemory = open_sharedmem(SHMID);
    if ((int)sharedMemory < 0)
        return -1;

    for (int i = 0; i < NUM_OF_PROCESSES; i++)
    {
        if (fork() == 0)
        {
            acquire_prioritylock();
            char *childSharedMemory = open_sharedmem(SHMID);
            childSharedMemory[i] += 1;
            printf(1, "Child process (pid: %d) - value: %d\n", getpid(), childSharedMemory[i]);
            release_prioritylock();

            close_sharedmem(SHMID);
            exit();
        }
    }
    for (int i = 0; i < NUM_OF_PROCESSES; i++)
        wait();

    int extractedChildCount = 0;
    for (int i = 0; i < 5; i++)
        extractedChildCount += sharedMemory[i];

    printf(1, "Parent process (pid: %d) - value: %d\n", getpid(), extractedChildCount);
    close_sharedmem(SHMID);
    exit();
}
```

- خروجی تولید شده نیز به شرح زیر می‌باشد:

```
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ shm_test
Shared memories initial value in parent process(3) is : 0

Child process 4 modified shared value. new value is: 1
Child process 5 modified shared value. new value is: 2
Child process 6 modified shared value. new value is: 3
Child process 7 modified shared value. new value is: 4
Child process 8 modified shared value. new value is: 5

Shared memories final value in parent process(3) is : 5
$ shm_test
Shared memories initial value in parent process(9) is : 0

Child process 10 modified shared value. new value is: 1
Child process 11 modified shared value. new value is: 2
Child process 12 modified shared value. new value is: 3
Child process 13 modified shared value. new value is: 4
Child process 14 modified shared value. new value is: 5

Shared memories final value in parent process(9) is : 5
$ _
```

- **Github Hash: c8921309fd2dfb30d237214059632d62d5a490a2**