گزارش آزمایشگاه سیستم عامل: پروژه شماره ۵

گروه ۲۲

پردیس زندکریمی - محمد امانلو - شهزاد ممیز

۸۱۰۱۰۱۰۸۱ - ۸۱۰۱۰۰۰۸۴ - ۸۱۰۱۰۰۲۷۲

Last commit code:

bde38cccf99f92d71f27a74bda04004f5921e147

Github repo:

https://github.com/MohammadAmanlou/OS_lab

۱.

در لینوکس ساختمان داده ای به نام VMA داریم که وظیفه آن توصیف بازه ای از آدرس های مجازی است که این آدرس مجازی اتریبیوت هایی دارد که محتویات آن قسمت از حافظه را توصیف میکند و با استفاده از آنها میتوانیم بفهمیم که این ناحیه با دیتا پر شده است یا خیر. با استفاده از ساختمان داده ی ذکر شده حافظه مجازی پردازه ها مدیریت میشوند. در سیستم عامل لینوکس از pagetable برای مپ کردن آدرس مجازی و فیزیکی استفاده میشود. هر VMA شامل تعدادی ورودی از pagetable است. زمانیکه یک پردازه به یک آدرس مجازی دسترسی پیدا میکند ورودی های متناظر آدرس مجازی را به فیزیکی ترجمه میکند. در سیستم عامل xv6 از VMA استفاده نمیشود و هسته از سیاست مدیریت دیگری برای انجام این کار استفاده میکند.

.۲

زمانی که ساختار سلسله مراتبی داریم پردازه ها و تسک ها توانایی ایجاد تناظر یا map کردن کد ها و داده ها را خواهند داشت و از مصرف حافظه تا حدی کاسته میشود.
همینطور این ساختار به سیستم اجازه میدهد که داده های که بیشتر از آنها استفاده میکند را در حافظه cache که سریعتر است ذخیره کند و همین امر باعث میشود تعداد دسترسی ها به حافظه کاهش یابد که در نهایت باعث پرفورمنس بهتر سیستم میشود.

.۳

۳۲ بیت مدخل دو قسمت دارد: اشاره گر به سطح بعدی حافظه و سطح دسترسی. که برای بخش اول ۲۰ بیت و ۱۲ بیت باقیمانده برای بخش دوم است. بخش ۱۲ بیتی در هر سطر دسترسی وجود دارد. در سطح page table از ۲۰ بیت برای آدرس فیزیکی استفاده می شود. بیتی به نام dirty (D) وجود دارد که در سطوح تفاوت دارد. در page table معنای خاصی ندارد اما در page directory به معنای این است که صفحه باید در دیسک نوشته شود. این شرطی است که برای اعمال تغییرات دارد.

.۴

در میان فایل های xv6 فایلی به نام kalloc.c وجود دارد که پیاده سازی تابع ذکر شده در این فایل است و تخصیص حافظه فیزیکی توسط این تابع انجام میشود. در xv6 این تابع برای تخصیص حافظه در kernel heap برای ذخیره سازی در ساختمان های پویا استفاده میشود. این تابع در لیستی از فضاهای خالی برای قسمتی از مموری (block) که به اندازه کافی فضا داشته باشد میگردد و در صورت پیدا کردن آن آنرا از لیست فضاهای خالی خارج میکند.

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

```
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;
}
```

.۵

این تابع به منظور ساختن mapping از آدرس مجازی به آدرس فیزیکی استفاده میشود. یک page table entry میسازد در این حین از walkpgdir استفاده میکند که آدرس مجازی با شروع از va را به آدرس فیزیکی با شروع از pa نگاشت دهد. از این تابع در توابع inituvm و setupkvm , copyuvm, allocuvm استفاده شده است.

```
// 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;
}
```

همچنین فلگ هایی برای PTE تعریف شده است که به شکل زیر است:

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

.۷

در اصل این تابع عمل map کردن آدرس مجازی به فیزیکی را بر عهده دارد.در این تابع در صورت وجود PTE که به آدرس مجازی با شروع از va اشاره دارد در pgdir آدرس آن را

برمیگرداند و همچنین اگر وجوود نداشت جدولی برای آن میسازد و آدرس آن را برمیگرداند. موارد کاربرد این تابع در توابعی است که به PTE متناظر با یک آدرس مجازی نیاز دارند.

mappages, loaduvm, deallocuvm, Clearpteu, copyuvm, uva2ka

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

.۸

تابع **allocuvm** در مفهوم خود allocate user virtual memory را جای داده است. هر دوی این توابع در فایل vm.c تعریف شده اند. افزایش حافظه مجازی برای کاربر در یک دایرکتوری صفحه خاص از مسئولیت های این تابع است. البته ممکن است فراخوانی این تابع با شکست مواجه شود. (مثلا در حالتی که Kalloc به درستی انجام نشود)

همانطور که پیش تر ذکر شد تابع **mappages** برای نگاشت آدرس مجازی به فیزیکی است.
با استفاده تابع allocuvm و فراخوانی آن حافظه اختصاص یافته در page directory مشخص به مقداری که میخواهیم زیاد میشود و اگر مشکلی جهت انجام این عملیات وجود داشته باشد صفر و در غیر اینصورت و درست اجرا شدن این عملیات عدد سایز بعنوان مقدار برگشتی ریترن میشود. پس میتوان گفت این تابع در زمانی استفاده میشود که پردازه به حافظه بیشتر یا جدید تر نیاز داشته باشد (در توابع exec یا growproc).
در تابع Allocuvm اول مقدار newsz و oldsz بررسی میشود و سپس با توجه به سایر تعدادی page در نظر گرفته میشود . در نهایت این صفحات با استفاده از mappages صفحات ساخته شده به آدرس مجازی آزاد در pagedir مپ میشوند تا پردازه دارنده pagedir بتواند از آن استفاده کند.

.۹
در مرحله اول فایل مشخص شده توسط پارامتر PATH با استفاده از namei (path) باز میشود و داخل استراکت inode قرار میگیرد. سپس ELF header برنامه خوانده و داخل استراکت elfhdr قرار میدهد. این هدر شامل اطلاعاتی درباره قطعه های برنامه می باشد. سپس تابع setupknm فراخوانی شده تا یک پیج تیبل جدید که pgdir نام دارد برای پردازه بسازد. حلقه ای روی ELF header های برنامه زده میشود و برای آنهایی که تایپ prog load دارند مراحل زیر طی میشود:
در ابتدا فضای حافظه برنامه با استفاده از تابع allocvm افزایش می یابد.این تابع صفحات مورد نیاز برنامه را در حافظه مجازی پردازه تخصیص میدهد و نگاشت آن را انجام میدهد. سپس تابع loaduvm فراخوانی می شود و محتوای برنامه را از inode (ip) خوانده و داخل حافظه پردازه قرار میدهد.

پس از انجام این مراحل که inode که قفل شده بود ازاد میشود. سپس در ادامه دو صفحه با استفاده از تابع allocuvm به هدف userstack تخصیص داده مشود که این صفحات بعد از مقادیر بخش های قبلی در حافظه قرار گرفته اند و همچینین صفحه اول غیرقابل دسترسی و صفحه دوم بعنوان userstack استفاده میشود. سپس ارگومان هایی با دیتاتایب استرینگ داخل استک قرار میگیرینند و باقی مقادیر استک در ارایه ustack تعبیه میشود. همچنین نام برنامه به منظور دیباگ ذخیره میشود. درمرحله بعد فیلد های پردازه آپدیت میشوند. (pgir , sz)

دو متغیر eip و esp به ترتیب مقادیر نقطه ورود برنامه و بالای userstack را در خود ذخیره میکنند.

با استفاده از تابع Switchuvm مقدار pagetable ذخیره شده به صورت سخت افزاری را اپدیت میکند تا از جدول جدید برای این پردازه استفاده شود. همچنین با استفاده از تابع freevm هم pagetable قبلی از حافظه پاک میشود.

همچنین اگر در اجرای این تابع به مشکلی بربخوریم به لیبل bad میرویم و در آنجا مقدار pgdir از حافظه پاک میشود و قفل inode نیز آزاد میشود.

پیاده سازی مسئله:

در این بخش به پیاده سازی بخش دستور پروژه می پردازیم:

ابتدا برای ایجاد یک فضای اشتراکی بین پردازه ها نیازمند تعریف 3 struct جدید هستیم.

1. ابتدا استراکت زیر را ایجاد می کنیم. در این استراکت که مربوط به فضای مشترکی است که هر پردازه می تواند در اختیار داشته باشد، attribute های زیر را مطابق شکل قرار داده و همچنین تعداد بخش های مشترک را مساوی با همین مقدار در memlayout.h تعریف میکنیم. این استراکت در proc.h تعریف می شود. در نهایت نیز بایستی در استراکت پراسس ها یک لیستی از این استراکت داشته

باشیم.

```c
#define SHAREDREGIONS 64 // same as marco in memlayout.h

typedef struct sharedPages {
  uint key, size;
  int shmid,perm;
  void *virtualAddr;
} sharedPages;
```

2و3. سپس دو استراکت زیر را در vm.c تعریف می کنیم. استراکت اول اطلاعات هر region را ذخیره می کند و در استراکت دوم تمامی ناحیه های موجود ذخیره سازی شده و برای آن قفل ایجاد می شود. که مدیریت دسترسی به shared memory به درستی انجام شود.

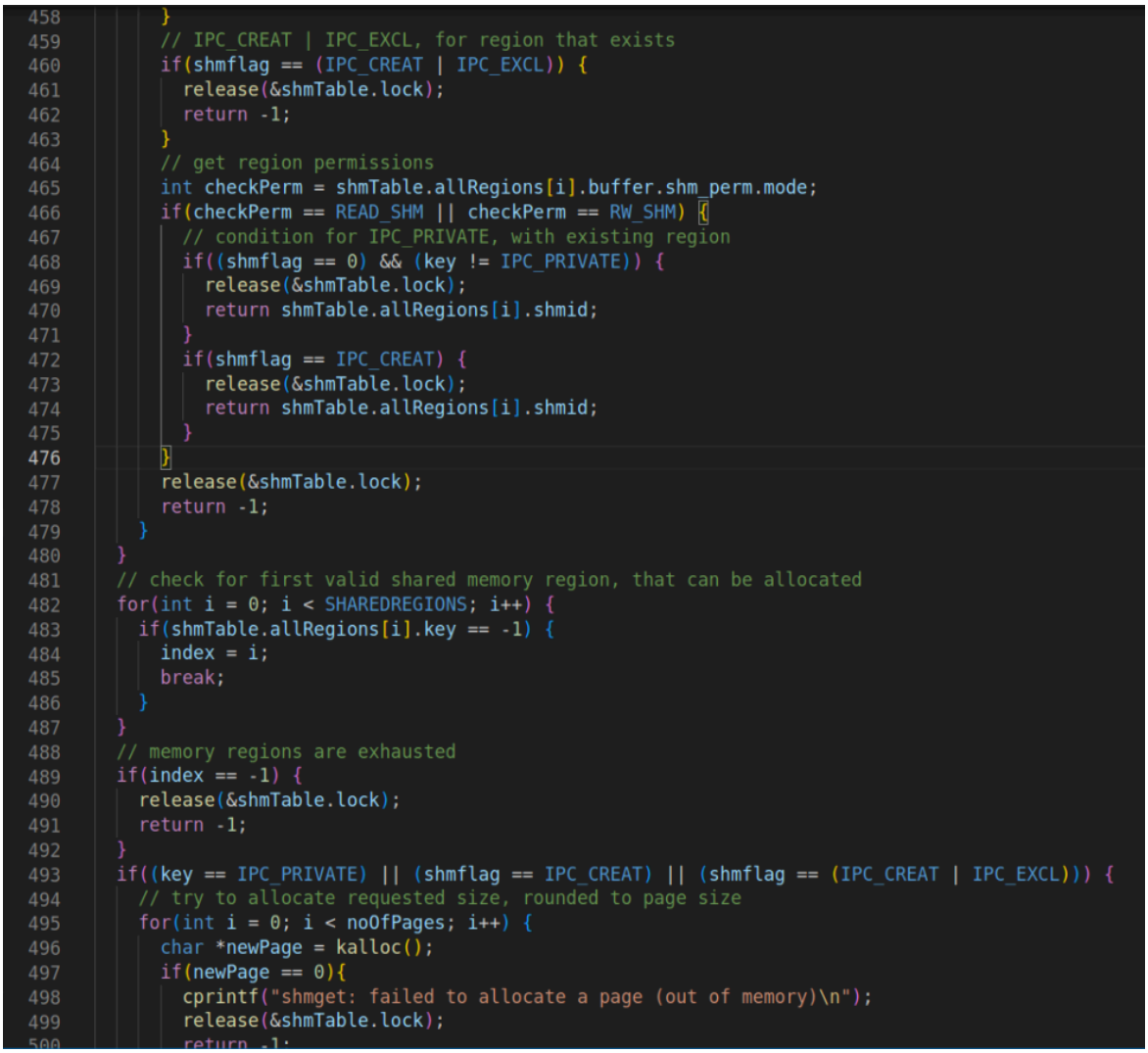
```c
// structure of single shared memory region
struct shmRegion {
  uint key, size; // key = region key; size = number of pages, e.g. requested size = 4096 (PGSIZE), then size = 1
  int shmid;  // shmid
  int toBeDeleted;  // flag to check if the region is marked for deletion or not. 1 = marked for deletion, 0 = not marked (default)
  void *physicalAddr[SHAREDREGIONS];  // store V2P of pages
  struct shmid_ds buffer; // kernel shmid_ds data structure associated with a region
};

// shared memory table
struct shmTable {
  // lock for table
  struct spinlock lock;
  // total shared memory regions
  struct shmRegion allRegions[SHAREDREGIONS];
} shmTable;
```

در گام بعدی می بایست دستورات مرتبط با سیستم کال open_shared_memory را پیاده سازی کنیم. در این سیستم کال ابتدا باید بررسی شود که آیا حافظه اشتراکی مد

نظر از قبل موجود بوده است یا خیر؟ در صورتی که این حافظه موجود نباشد بایستی با

```
int
shmget(uint key, uint size, int shmflag) {
  // as Xv6 has only single user, else lower 9 bits would be considered
  int lowerBits = shmflag & 7, permission = -1;

  acquire(&shmTable.lock);

  // separate correct permissions and shmflag
  if(lowerBits == (int)READ_SHM) {
    permission = READ_SHM;
    shmflag ^= READ_SHM;
  }
  else if(lowerBits == (int)RW_SHM) {
    permission = RW_SHM;
    shmflag ^= RW_SHM;
  } else {
    if(!((shmflag == 0) && (key != IPC_PRIVATE))) {
      release(&shmTable.lock);
      return -1;
    }
  }
  // check for requested size
  if(size <= 0) {
    release(&shmTable.lock);
    return -1;
  }
  // calculate no of requested pages, from entered size
  int noOfPages = (size / PGSIZE) + 1;
  // check if no of pages is more than decided limit
  if(noOfPages > SHAREDREGIONS) {
    release(&shmTable.lock);
    return -1;
  }
  int index = -1;
  // check if key already exists
  for(int i = 0; i < SHAREDREGIONS; i++) {
    if(shmTable.allRegions[i].key == key) {
      // if wrong size is requested with existing region
      if(shmTable.allRegions[i].size != noOfPages) {
        release(&shmTable.lock);
        return -1;
```

استفاده از دستور shmget آن را ایجاد می کنیم و سپس آن را attach می کنیم.



```
458       }
459       // IPC_CREAT | IPC_EXCL, for region that exists
460       if(shmflag == (IPC_CREAT | IPC_EXCL)) {
461         release(&shmTable.lock);
462         return -1;
463       }
464       // get region permissions
465       int checkPerm = shmTable.allRegions[i].buffer.shm_perm.mode;
466       if(checkPerm == READ_SHM || checkPerm == RW_SHM) {
467         // condition for IPC_PRIVATE, with existing region
468         if((shmflag == 0) && (key != IPC_PRIVATE)) {
469           release(&shmTable.lock);
470           return shmTable.allRegions[i].shmid;
471         }
472         if(shmflag == IPC_CREAT) {
473           release(&shmTable.lock);
474           return shmTable.allRegions[i].shmid;
475         }
476       }
477       release(&shmTable.lock);
478       return -1;
479     }
480   }
481   // check for first valid shared memory region, that can be allocated
482   for(int i = 0; i < SHAREDREGIONS; i++) {
483     if(shmTable.allRegions[i].key == -1) {
484       index = i;
485       break;
486     }
487   }
488   // memory regions are exhausted
489   if(index == -1) {
490     release(&shmTable.lock);
491     return -1;
492   }
493   if((key == IPC_PRIVATE) || (shmflag == IPC_CREAT) || (shmflag == (IPC_CREAT | IPC_EXCL))) {
494     // try to allocate requested size, rounded to page size
495     for(int i = 0; i < noOfPages; i++) {
496       char *newPage = kalloc();
497       if(newPage == 0){
498         cprintf("shmget: failed to allocate a page (out of memory)\n");
499         release(&shmTable.lock);
500         return -1;
501       }
502       // zero out
503       memset(newPage, 0, PGSIZE);
504       shmTable.allRegions[index].physicalAddr[i] = (void *)V2P(newPage);
505     }
506     // mark rest of the fields in structure
507     shmTable.allRegions[index].size = noOfPages;
508     shmTable.allRegions[index].key = key;
509
510     // store data for shmid_ds data structure
511     shmTable.allRegions[index].buffer.shm_segsz = size;
512     shmTable.allRegions[index].buffer.shm_perm.__key = key;
513     shmTable.allRegions[index].buffer.shm_perm.mode = permission;
514
515     // store creator pid
516     shmTable.allRegions[index].buffer.shm_cpid = myproc()->pid;
517
518     // store shmid in not yet shared region
519     shmTable.allRegions[index].shmid = index;
520
521     release(&shmTable.lock);
522     return index; // valid shmid
523   } else {
524     release(&shmTable.lock);
525     return -1;
526   }
527 }
```

در زیر پیاده سازی تابع shmat برای attach کردن و در واقع همان open_sharedmem

```
615  void*
616  shmat(int shmid, void* shmaddr, int shmflag) {
617    if(shmid < 0 || shmid > 64) {
618      return (void*)-1;
619    }
620    acquire(&shmTable.lock);
621    int index = -1,idx, permflag;
622    uint segment,size = 0;
623    void *va = (void*)HEAPLIMIT, *least_va;
624    struct proc *process = myproc();
625    index = shmTable.allRegions[shmid].shmid;
626    if(index == -1) {
627      // shmid not found
628      release(&shmTable.lock);
629      return (void*)-1;
630    }
631    if(shmaddr) {
632      if((uint)shmaddr >= KERNBASE || (uint)shmaddr < HEAPLIMIT) {
633        release(&shmTable.lock);
634        return (void*)-1;
635      }
636      // round down to nearest multiple of SHMLBA
637      uint rounded = ((uint)shmaddr & ~(SHMLBA-1));
638
639      if(shmflag & SHM_RND) {
640        if(!rounded) {
641          release(&shmTable.lock);
642          return (void*)-1;
643        }
644        va = (void*)rounded;
645      } else {
646
647        // page aligned address
648        if(rounded == (uint)shmaddr) {
649          va = shmaddr;
650        }
651      }
652
```

مشاهده می شود.

```
653     } else {
654       for(int i = 0; i < SHAREDREGIONS; i++) {
655         idx = getLeastvaidx(va,process);
656         if(idx != -1) {
657           least_va = process->pages[idx].virtualAddr;
658           if((uint)va + shmTable.allRegions[index].size*PGSIZE <=  (uint)least_va)
659             break;
660           else
661             va = (void*)((uint)least_va + process->pages[idx].size*PGSIZE);
662         } else
663           break;
664       }
665     }
666     if((uint)va + shmTable.allRegions[index].size*PGSIZE >= KERNBASE) {
667       // size exceeded
668       release(&shmTable.lock);
669       return (void*)-1;
670     }
671     idx = -1;
672     for(int i = 0; i < SHAREDREGIONS; i++) {
673       if(process->pages[i].key != -1 && (uint)process->pages[i].virtualAddr + process->pa
674         idx = i;
675         break;
676       }
677     }
678     if(idx != -1) {
679       if(shmflag & SHM_REMAP) {
680         segment = (uint)process->pages[idx].virtualAddr;
681         // repeat till all conflicting mappings are removed
682         while(segment < (uint)va + shmTable.allRegions[index].size*PGSIZE) {
683           size = process->pages[idx].size;
684           release(&shmTable.lock);
685           if(shmdt((void*)segment) == -1) {
686             return (void*)-1;
687           }
688           acquire(&shmTable.lock);
689           idx = getLeastvaidx((void*)(segment + size*PGSIZE),process);
690           if(idx == -1)
691             break;
692           segment = (uint)process->pages[idx].virtualAddr;
693         }
```

```
699     }
700     if((shmflag & SHM_RDONLY) || (shmTable.allRegions[index].buffer.shm_perm.mode == READ_SHM)){
701       permflag = PTE_U;
702     }
703     else if (shmTable.allRegions[index].buffer.shm_perm.mode == RW_SHM) {
704       permflag = PTE_W | PTE_U;
705     } else {
706       //permission mismatch between get and attach
707       release(&shmTable.lock);
708       return (void*)-1;
709     }
710     for (int k = 0; k < shmTable.allRegions[index].size; k++) {
711       if(mappages(process->pgdir, (void*)((uint)va + (k*PGSIZE)), PGSIZE, (uint)shmTable.allRegions[index].physicalAddr[k], permflag) < 0) {
712         deallocuvm(process->pgdir,(uint)va,(uint)(va + shmTable.allRegions[index].size));
713         release(&shmTable.lock);
714         return (void*)-1;
715       }
716     }
717     idx = -1;
718     for(int i = 0; i < SHAREDREGIONS; i++) {
719       if(process->pages[i].key == -1) {
720         idx = i;
721         break;
722       }
723     }
724     if(idx != -1) {
725       process->pages[idx].shmid = shmid;
726       process->pages[idx].virtualAddr = va;
727       process->pages[idx].key = shmTable.allRegions[index].key;
728       process->pages[idx].size = shmTable.allRegions[index].size;
729       process->pages[idx].perm = permflag;
730       shmTable.allRegions[index].buffer.shm_nattch += 1;
731       shmTable.allRegions[index].buffer.shm_lpid = process->pid;
732     } else {
733       release(&shmTable.lock);
734       return (void*)-1; // all page regions exhausted
735     }
736     release(&shmTable.lock);
737     return va;
738   }
```

در مرحله بعدی بایستی دستور close_sharedmem را پیاده سازی کنیم به طریق مشابه در این قسمت، آدرس آن ناحیه مشترکی که پیش از این ایجاد شده بود را از آن گرفته یا در واقع detach می کنیم، یا در واقع آن را از لیست ناحیه های اشتراکی داخل پردازه حذف میکنیم. و در صورتی که هیچ پراسسی این ناحیه را در اختیار نداشته باشد این ناحیه را از جدول حافظه های اشتراکی نیز حذف می کنیم. این دستور را در تابع shmdt پیاده سازی کرده ایم.

```
552  int
553  shmdt(void* shmaddr) {
554    acquire(&shmTable.lock);
555    struct proc *process = myproc();
556    void* va = (void*)0;
557    uint size;
558    int index,shmid;
559    for(int i = 0; i < SHAREDREGIONS; i++) {
560      // find the index from pages array which is attached at the provided shmaddr
561      if(process->pages[i].key != -1 && process->pages[i].virtualAddr == shmaddr) {
562          va =  process->pages[i].virtualAddr;
563          index = i;
564          shmid = process->pages[i].shmid;
565          size = process->pages[index].size;
566          break;
567      }
568    }
569    if(va) {
570      for(int i = 0; i < size; i++) {
571        pte_t* pte = walkpgdir(process->pgdir, (void*)((uint)va + i*PGSIZE), 0);
572        if(pte == 0) {
573          release(&shmTable.lock);
574          return -1;
575        }
576        *pte = 0;
577      }
578      process->pages[index].shmid = -1;
579      process->pages[index].key = -1;
580      process->pages[index].size =  0;
581      process->pages[index].virtualAddr = (void*)0;
582      if(shmTable.allRegions[shmid].buffer.shm_nattch > 0) {
583        // decrement attaches
584        shmTable.allRegions[shmid].buffer.shm_nattch -= 1;
585      }
586      if(shmTable.allRegions[shmid].buffer.shm_nattch == 0 && shmTable.allRegions[shmid].toBeDeleted == 1) {
587        // remove the segments
588        for(int i = 0; i < shmTable.allRegions[index].size; i++) {
589          char *addr = (char *)P2V(shmTable.allRegions[index].physicalAddr[i]);
590          kfree(addr);
591          shmTable.allRegions[index].physicalAddr[i] = (void *)0;
```

```
588         for(int i = 0; i < shmTable.allRegions[index].size; i++) {
589           char *addr = (char *)P2V(shmTable.allRegions[index].physicalAddr[i]);
590           kfree(addr);
591           shmTable.allRegions[index].physicalAddr[i] = (void *)0;
592         }
593         shmTable.allRegions[index].size = 0;
594         shmTable.allRegions[index].key = shmTable.allRegions[index].shmid = -1;
595         shmTable.allRegions[index].toBeDeleted = 0;
596         shmTable.allRegions[index].buffer.shm_nattch = 0;
597         shmTable.allRegions[index].buffer.shm_segsz = 0;
598         shmTable.allRegions[index].buffer.shm_perm.__key = -1;
599         shmTable.allRegions[index].buffer.shm_perm.mode = 0;
600         shmTable.allRegions[index].buffer.shm_cpid = -1;
601         shmTable.allRegions[index].buffer.shm_lpid = -1;
602       }
603       shmTable.allRegions[shmid].buffer.shm_lpid = process->pid;
604       release(&shmTable.lock);
605       return 0;
606     } else {
607       release(&shmTable.lock);
608       return -1;
609     }
610
611   }
```

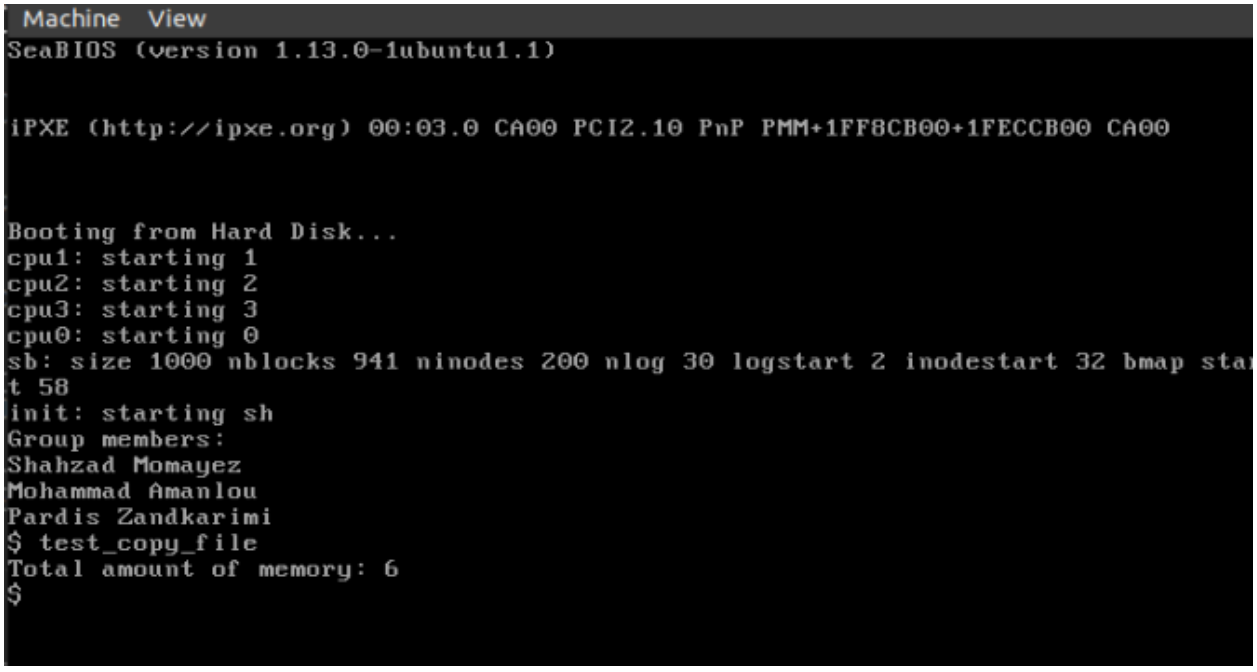
در نهایت نیز می بایست یک برنامه تست ایجاد کنیم که در این برنامه تست یک حافظه اشتراکی با مقدار صفر ایجاد میکنیم و تعدادی پردازه ایجاد میکنیم هر پردازه فرزند یکی به مقدار این حافظه مشترک اضافه می کند، در نهایت که مقدار این حافظه مشترک را از حافظه پدر بخواهیم باید عددی معادل با تعداد پردازه های فرزند داشته باشیم.

OS_lab > xv6-public > C test_copy_file.c > main(int, char * [])

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "ipc.h"
#include "shm.h"
#include "memlayout.h"

#define SHM_KEY 1000
#define NUM_CHILD_PROCESSES 6

int main(int argc, char *argv[]) {
    int shmid = shmget(SHM_KEY, sizeof(int), 0);
    if (shmid < 0) {
        shmid = shmget(SHM_KEY, sizeof(int), 06 | IPC_CREAT);
        if (shmid < 0) {
            printf(1, "Failed to create shared memory segment\n");
            exit();
        }
        int *ptr = (int *)shmat(shmid, 0, 0);
        if ((int)ptr < 0) {
            printf(1, "Failed to attach shared memory segment\n");
            exit();
        }
        *ptr = 0;
        shmdt(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 childShmid = shmget(SHM_KEY, sizeof(int), 0);
            if (childShmid < 0) {
                printf(1, "Failed to get shared memory segment\n");
                exit();
            }
            int *childPtr = (int *)shmat(childShmid, 0, 0);
            if ((int)childPtr < 0) {
                printf(1, "Failed to attach shared memory segment\n");
                exit();
```

```
    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 childShmid = shmget(SHM_KEY, sizeof(int), 0);
            if (childShmid < 0) {
                printf(1, "Failed to get shared memory segment\n");
                exit();
            }
            int *childPtr = (int *)shmat(childShmid, 0, 0);
            if ((int)childPtr < 0) {
                printf(1, "Failed to attach shared memory segment\n");
                exit();
            }
            *childPtr = *childPtr + 1;
            shmdt(childPtr);
            exit();
        }
    }

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

    // Report the amount of memory
    int *parentPtr = (int *)shmat(shmid, 0, 0);
    if ((int)parentPtr < 0) {
        printf(1, "Failed to attach shared memory segment\n");
        exit();
    }
    printf(1, "Total amount of memory: %d\n", *parentPtr);
    shmdt(parentPtr);

    // Remove shared memory segment
    shmctl(shmid, IPC_RMID, 0);

    exit();
}
```

همان طور که در تصویر زیر مشاهده می شود، مقدار خروجی داده شده مساوی با تعداد فرزندانی که ایجاد کرده ایم که مطابق کد بالا به مقدار define 6 شده است می باشد.


Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00
Booting from Hard Disk...
cpu1: starting 1
cpu2: starting 2
cpu3: starting 3
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
t 58
init: starting sh
Group members:
Shahzad Momayez
Mohammad Amanlou
Pardis Zandkarimi
$ test_copy_file
Total amount of memory: 6
$