آزمایشگاه سیستم عامل

پروژه شماره 5

علی پادیاو – اولدوز نیساری – کسری حاجی حیدری

بهار 1402

Repository Link: https://github.com/alumpish/OS-Lab-Projects

Latest Commit Hash: d0162d5bfc19653c9d4c09c3c9cd62af0ee34338

-1

در لینوکس VMA یک ساختمان داده است که ناحیه پیوسته ای را از فضاهای آدرس virtual توصیف می‌کند. در حقیقت هر VMA در حقیقت یک بازه از آدرس‌های مجازی را توصیف می‌کند که خود شامل attribute هایی برای توضیف محتویان آن ناحیه است. این attribute ها کمک می‌کنند که متوجه شویم که ناحیه شامل داده است یا خیر.

در سیستم عامل لینوکس از مفهوم VMA به صورت گسترده برای مدیریت حافظه مجازی پردازه‌ها استفاده می‌شود. همان طور که می‌دانیم در لینوکس از page table برای ایجاد تناظر بین آدرس مجازی و فیزیکی استفاده می‌شود. هر VMA شامل تعدادی entry از page table متناظر است، زمانی که یک پردازه به یک آدرس مجازی دسترسی پیدا می‌کند entry-های متناظر آدرس مجازی را به فیزیکی ترجمه می‌کنند.

در xv6 به صورت واضح از VMA استفاده نمی‌شود بلکه هسته آن از یک مکانیزم مدیریت ساده تر که به صورت مستقیم آدرس مجازی را به فیزیکی تبدیل می‌کند استفاده می‌کند.

-2

زمانی که از ساختار سلسه مراتبی استفاده می‌کنیم ، پردازه‌ها و تسک‌ها توانایی این را پیدا می‌کنند که با استفاده از mapping کدها و داده‌ها را به اشتراک بگذارند و با این کار از مصرف حافظه اضافی جلوگیری کنند.

هم چنین این ساختار به سیستم کامپیوتری اجازه میدهد که داده‌هایی که از آن‌ها بیشتر استفاده می‌کند را در حافظه سریع تر و کوچک cache ذخیره کند. این باعث می‌شود تعداد memory access-ها کاهش یابد که به صورت فاحشی باعث بهبود عملکرد سیستم می‌شود.

-3

مدخل 32 بیتی از دو بخش تشکیل شده است بخشی برای اشاره به سطح بعدی حافظه و بخشی دیگر برای سطح دسترسی.

تعداد بیت‌های اختصاص داده شده برای اشاره به سطح بعدی 20 است و 12 بیت باقی مانده برای سطح دسترسی است.

میتوان گفت بخش 12 بیتی در هر سطر دسترسی وجود دارد ،در سطح page table از 20 بیت برای آدرس فیزیکی استفاده می‌شود.

بیتی به نام dirty(D) وجود دارد که در سطوح دارای تفاوت است. در page table معنای خاصی ندارد اما در page directory به این معناست که صفحه باید در دیسک نوشته شده شود ، این شرطی است که برای اعمال تغییرات دارد.

-4

اگر در میان فایل‌های xv6 دقت کنیم فایلی به نام kalloc.c وجود دارد که پیاده سازی این تابع در این فایل است. در ابتدا این برای توضیح کامنت‌هایی به صورت زیر نوشته شده است:

```
// 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;
}
```

این کامنت‌ها نشان می‌دهد که تخصیص حافظه فیزیکی توسط این تابع انجام می‌شود.

در کل در xv6 این تابع برای تخصیص حافظه در kernel heap برای ذخیره سازی ساختمان‌های پویا استفاده می‌شود. این تابع در لیستی از فضاهای خالی به دنبال memory block-ای خالی که به اندازه کافی بزرگ باشد می‌گردد. زمانی که پیدا کرد آن را از لیست فضاهای خالی خارج می‌کند.

-5

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

همانطور که در کامنت های تکه کد روبرو نوشته شده است این تابع به منظور ساختن نگاشت از آدرس مجازی به فیزیکی استفاده میشود.

(یک page table entry می‌سازد) در این حین از تابع walkpgdir استفاده میکند (که آدرس مجازی با شروع از va را به آدرس فیزیکی با شروع از pa نگاشت میدهد).

این تابع در توابع setupkvm, inituvm, allocuvm, copyuvm استفاده شده است.

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

```
// 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
```

-7

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 وجود نداشت جدولی برای آن میسازد و آدرس آن را برمیگرداند. در اصل این تابع عمل سخت افزاری ترجمه آدرس مجازی به فیزیکی را شبیه سازی می‌نماید.

این تابع در توابع زیر که به PTE متناظر با یک آدرس مجازی نیازمند هستند استفاده شده است:

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)];
}
```

-8

این دو تابع در فایل vm.c تعریف شده اند. همانطور که در سوال 5 گفته شد، تابع mappages به منظور ایجاد نگاشت از آدرس مجازی به آدرس فیزیکی استفاده میشود. allocuvm کوتاه شده عبارت allocate user virtual memory می‌باشد که به تعریف این تابع می‌پردازیم:

```
// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
  return newsz;
}
```

هنگامی که پردازه به حافظه بیشتر یا جدید نیاز داشته باشد (در توابع exec یا growproc) با استفاده از این تابع و فراخوانی آن، حافظه اختصاص داده شده در page directory مشخص به مقدار خواسته شده افزایش می‌یابد و اگر به اروری برنخوریم مقدار سایز و در غیر اینصورت صفر بازمی‌گردد.

در این تابع ابتدا مقادیر newsz و oldsz چک می‌شود. سپس با توجه به سایز تعدادی صفحه در نظر گرفته می‌شود و در نهایت با استفاده از تابع mappages صفحه ساخته شده به آدرس مجازی آزاد در pagedir نگاشت داده می‌شود تا پردازه دارنده pagedir بتواند از آن استفاده نماید.

-9

شیوه ی بارگذاری برنامه در حافظه توسط فراخوانی سیستمی exec به صورت زیر است:

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 می‌گیرد. این هدر شامل اطلاعاتی درباره قطعه‌های برنامه می‌باشد. سپس تابع setupkvm فراخوانی میشود تا یک جدول صفحه جدید که pgdir نام دارد برای پردازه بسازد.

حلقه ای بر روی ElF header های برنامه زده میشود و برای هدرهایی که تایپ ELF_PROG_LOAD دارند مراحل زیر طی میشود:

در ابتدا فضای حافظه برنامه با استفاده از تابع allocuvm افزایش میابد.این تابع صفحات مورد نیاز برنامه را در حافظه مجازی پردازه تخصیص میدهد و نگاشت آن را نیز انجام میدهد.سپس تابع loaduvm فراخوانی میشود و محتوای برنامه را از inode(ip) میخواند و داخل حافظه پردازه قرار میدهد.

بعد از انجام مراحل بالا inode که قفل شده بود آزاد میشود.در ادامه دو صفحه با استفاده از تابع allocuvm به هدف userstack تخصیص داده میشود که این صفحات بعد از مقادیر بخش‌های قبلی در حافظه قرار گرفته اند و همچنین صفحه اول غیر قابل دسترسی و صفحه دوم به عنوان userstack استفاده میشود.

در مرحله بعد ارگومان‌های از جنس استرینگ داخل استک قرار میگیرند و باقی مقادیر استک در آرایه ustack تعبیه میشود.همچنین نام برنامه به منظور دیباگ کردن ذخیره میشود.

در مرحله بعد فیلدهای پردازه اپدیت میشوند(pgdir, sz).

eip به نقطه ورود برنامه مقدار دهی میشود esp نیز به بالای userstack اشاره می‌کند.

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، pagetable قدیمی‌از حافظه پاک میشود.

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

تغییر ساختار حافظه

به استراکت proc فیلد st که نشان‌دهنده top of stack است اضافه میکنیم.

فایل vm.c:

در تابع copyuvm که page table پردازه پدر را در فرزند کپی می‌کند، آرگومان st را اضافه می‌کنیم و تکه کد زیر را به این تابع اضافه می‌کنیم:

```
if (st == 0)
{
  return d;
}

for (i = st; i < KERNBASE; i += PGSIZE)
{
  if ((pte = walkpgdir(pgdir, (void *)i, 0)) == 0)
  {
    panic("copyuvm: pte should exist 2");
  }
  if (!(*pte & PTE_P))
  {
    panic("copyuvm: page not present 2");
  }
  pa = PTE_ADDR(*pte);
  flags = PTE_FLAGS(*pte);
  if ((mem = kalloc()) == 0)
  {
    goto bad;
  }
  memmove(mem, (char *)P2V(pa), PGSIZE);
  if (mappages(d, (void *)i, PGSIZE, V2P(mem), flags) < 0)
  {
    goto bad;
  }
}
```

اگر پردازه پدر فضای مخصوص استک داشته باشد، از top of stack تا KERNBASE محتویاتش را در پردازه فرزند کپی می‌کنیم.

فایل exec.c:

در xv6 معمولی در تابع exec به صورت زیر حافظه به استک اختصاص داده می‌شود.

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

اما از آنجایی که میخواهیم استک به ته حافظه برود، از KERNBASE شروع می‌کنیم و به اندازه PGSIZE 2 بالاتر از آن را به استک می‌دهیم.

```
st = KERNBASE - 2*PGSIZE;
if((sp = allocuvm(pgdir, st, KERNBASE)) == 0)
  goto bad;
```

و در آخر هم مقدار top of stack را در پردازه ذخیره می‌کنیم.

```
curproc->st = st;
```

فایل proc.c:

در تابع fork curproc->st را هم به copyuvm پاس می‌دهیم و مقدار st در np را هم برابر این مقدار قرار می‌دهیم.

فایل syscall.c:

در 3 تابع fetchint، fetchstr و argptr برای چک کردن محدوده آدرس از sz استفاده می‌شد که با تغییرات انجام شده به جای sz از KERNBASE استفاده می‌کنیم.

فایل trap.c:

یک کیس برای هندل کردن ترپ‌های pagefault اضافه می‌کنیم:

```
case T_PGFLT:
  unmapped_addr = rcr2();
  stackTop = myproc()->st;
  new_stackTop = stackTop - PGSIZE;
  if (unmapped_addr < stackTop && unmapped_addr > new_stackTop)
  {
    if (allocuvm(myproc()->pgdir, new_stackTop, stackTop) == 0)
    {
      cprintf("Page fault detected...\n"
              " pid %d %s: trap %d err %d on cpu %d eip 0x%x add 0x%x--kill proc, top_stack 0x%x\n",
              myproc()->pid, myproc()->name, tf->trapno,
              tf->err, cpuid(), tf->eip, unmapped_addr, stackTop);
      myproc()->killed = 1;
      break;
    }
    else
    {
      myproc()->st = new_stackTop;
      break;
    }
  }
```

در اینجا اگر تله مربوط به page fault باشد به این کیس می‌رویم. با استفاده از تابع rcr2 آدرس حافظه‌ای که نتوانستیم اختصاص دهیم را دریافت می‌کنیم. اگر این آدرس مربوط به page-ای که دقیقا بعد از top of stack ما قرار دارد باشد، یک page به استکمان با استفاده از allocuvm اضافه می‌کنیم. اگر allocuvm نتواند این کار را انجام دهد، یعنی حافظه فیزیکی‌ای برایمان نمانده و اخطار pagefault می‌دهیم.

برنامه سطح کاربر

برنامه memtest به صورت زیر نوشته شد که گسترش stack و heap راتست می‌کند.

```
int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        display_help();
        exit();
    }
    if (!strcmp(argv[1], "stack"))
    {
        stack(1);
    }
    else if (!strcmp(argv[1], "heap"))
    {
        heap();
    }
    else if (!strcmp(argv[1], "both"))
    {
        if (fork() == 0)
        {
            stack(1);
            exit();
        }
        if (fork() == 0)
        {
            heap();
            exit();
        }

        wait();
        wait();
    }
    else
        display_help();
    exit();
}
```

با دستور memset stack تابع stack به صورت recursive کال می‌شود تا جایی که دیگر نشود مموری‌ای برای استک اختصاص داد. دستور memset heap تابع heap را صدا می‌کند که در یک حلقه هر بار به اندازه chunk که 4096 بایت باشد، فضا به هیپ اختصاص می‌دهد. اگر هم دستور memset both را بزنید 2 پردازه ساخته می‌شود که یکی stack و دیگری heap را کال می‌کند تا هر دو همزان به سمت هم رشد کنند.

```c
void stack(int count)
{
    if (count % 1000000 == 0)
        printf(1, "stack: %d\n", count);
    stack(count + 1);
    printf(1, "End");
    exit();
}

void heap()
{
    int *ptr;

    for (int i = 0;; i++)
    {
        ptr = (int *)sbrk(CHUNK_SIZE);

        if (ptr == (int *)-1)
        {
            printf(1, "Memory allocation failed. Exiting...\n");
            exit();
        }

        if (i % 10000 == 0)
            printf(1, "Heap: %d\n", i);
    }
}
```

تعداد باری که stack به صورت ریکرسیو کال شد و همینطور تعداد دفعاتی که heap با استفاده از دستور sbrk، allocate شد هم به صورت تقریبی پرینت می‌شود. (هر بار کال کردن تابع stack-ای که نوشتیم، 32 بایت داده را به stack اضافه می‌کند.)

خروجی برنامه:

پیش از تغییر ساختار حافظه با اجرای این برنامه نتیجه زیر به دست آمد:

```
Count: 110
Count: 111
Count: 112
Count: 113
Count: 114
Count: 115
Count: 116
Count: 117
Count: 118
Count: 119
Count: 120
Count: pid 5 stack_test: trap 14 err 7 on cpu 0 eip 0x45c addr 0x1ff8--kill proc
```

مشاهده میشود که استک ظرفیت 120 بار کال شدن تابع stack را داشت. یعنی تقریبا 3840 بایت.

اما پس از بردن استک به پایین حافظه و دادن قابلیت گسترش به آن:

```
$ memtest stack
stack: 1000000
stack: 2000000
stack: 3000000
stack: 4000000
stack: 5000000
stack: 6000000
stack: 7000000
allocuvm out of memory
Page fault detected...
 pid 3 memtest: trap 14 err 6 on cpu 0 eip 0x129 add 0x72269ff0--kill proc, top_stack 0x7226a000
```

بیشتر از 7000000 بار stack کال شده و آدرس top_stack هم مشاهده می‌شود (تقریبا 220 مگابایت). پس استک دیگر محدود نیست و می‌تواند رشد کند.

با اجرای دستور memtest heap خروجی به صورت زیر است:

```
$ memtest heap
Heap: 0
Heap: 10000
Heap: 20000
Heap: 30000
Heap: 40000
Heap: 50000
allocuvm out of memory
Memory allocation failed. Exiting...
```

بیشتر از 50000 بار به اندازه chunk size فضا به هیپ اختصاص داده شده که تقریبا می‌شود 200 مگابایت. این مقدار نزدیک به مقداری است که به استک در دستور قبلی داده شد. یعنی هیپ و استک به یک اندازه می‌توانند از کل حافظه استفاده کنند.

در نهایت هم با دستور memtest both خروجی زیر را می‌بینیم:

```
$ memtest both
Heap: 0
stack: 1000000
Heap: 10000
stack: 2000000
stack: 3000000
Heap: 20000
stack: 4000000
allocuvm out of memory
allocuvm out of memory
MPage fault detected...
 pid 10 memtest: trap 14 err 6 on cpu 1 eip 0x129 add 0x78203ff0--kill proc, top_stack 0x78204000
emory allocation failed. Exiting...
```

می‌بینیم که هیپ و استک همزمان رشد کرده‌اند و به دلیل این همزمانی، نسبت به 2 دستور قبل به هر کدام حافظه کمتری رسیده.