به نام خدا

پروژه پنجم سیستم عامل

اعضای گروه:

سارا رضایی منش 810198576

نرگس غلامی 810198447

-1

در single level page table شما حتی برای دسترسی به یک دیتای کوچک هم به تمام table نیازمند هستید. مثلا اگر فرض بکنیم 2 به توان بیست پیج داریم که هر کدام 4 بایت را اشغال کرده اند کل مقداری که در مموری نیاز داریم وجود داشته باشد 2 به توان 22 یا 4MB است.

در multi-level paging می توان با استفاده از سازماندهی multi-level تصمیم بگیریم که داده ما در کدام پیج میان دو به توان بیست پیج موجود وجود دارد و آن را انتخاب بکنیم. پس در این منطق ما تنها به یک پیج مشخص هنگام ران کردن پراسس نیازمند هستیم. در اینجا مقدار مموریی که نیاز داریم یک پیج جدول سطح 1 و سپس 1 پیج از 2 به توان 10 صفحه در سطح دوم است. بنابراین، اندازه سطح 1 برابر است با 4bytes = 4KB * 2 ^ 10 همچنین در سطح دوم فقط به 1 صفحه در بین 2 ^ 10 صفحه جدول نیاز داریم . بنابراین اندازه 4bytes = 4KB * 2 ^ 10 است.

در کل سایزی که نیاز داریم برابر 8KB است که باید این مقدار را با 4MB مقایسه نمایید و مشاهده می‌کنیم که مقدار کمتری در این مرحله اشغال شده است.

-2

Access bit در لینوکس نشان می دهد که آیا نرم افزار به پیج ۴ کیلوبایتی اشاره شده توسط page table entry دسترسی پیدا کرده است یا خیر. حال با توجه به این موضوع، می توان فرکانس دسترسی به صفحه های حافظه را اینگونه تعریف کرد:

هنگام تغییر access bit به یک، متغیری با نام frequency که مقدار اولیه آن صفر است را یک واحد افزایش می دهیم. پس از گذشت یک ثانیه مقدار فرکانس بدست آمده است!

-3

هسته xv6 از توابع kalloc و kfree برای تخصیص دادن و آزاد کردن حافظه فیزیکی در run-time استفاده می کند. Kalloc یک صفحه ۴۰۹۶ بایتی از حافظه فیزیکی را تخصیص می دهد و یک پوینتر برمی گرداند که کرنل می تواند از آن استفاده کند و اگر نتواند حافظه ای تخصیص دهد، صفر برمی گرداند.

```
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;
}
```

-4

```
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;
}
```

mappages یک آدرس پیج (page table سطح اول)، یک آدرس مجازی، یک آدرس فیزیکی و یک اندازه را می گیرد و آن آدرس پیج را تغییر می دهد تا بایت های حافظه مجازی را اندازه کند و آدرس مجازی مشخص شده را به آدرس فیزیکی مشخص شده تبدیل کند. اگر آدرس مجازی مشخص شده قبلاً به یک آدرس فیزیکی نگاشت شده باشد panic می کند.

-5

```
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)];
}
```

این تابع یک آدرس page table سطح اول را می گیرد و یک اشاره‌گر به page table entry برای یک آدرس مجازی خاص بر می‌گرداند. به صورت اختیاری page tables های سطح دوم را نیز اختصاص خواهد داد.

walkpgdir از x86 paging hardware تقلید می کند. که PTE را برای یک آدرس مجازی جستجو می کند. walkpgdir از 10 بیت بالای آدرس مجازی برای یافتن page directory entry استفاده می کند. اگر page directory entry وجود نداشته باشد، page directory entry مورد نیاز هنوز تخصیص داده نشده است. اگر آرگومان alloc تنظیم شده باشد، walkpgdir آن را تخصیص می دهد و آدرس فیزیکی آن را در page directory قرار می دهد.

در نهایت از 10 بیت بعدی آدرس مجازی برای یافتن آدرس PTE در صفحه جدول صفحه استفاده می کند.

-6

عیب های نگاشت فایل در حافظه نسبت به حالت عادی به شرح زیر می باشد:

۱) بزرگترین عیب نگاشت فایل ها به حافظه این است که RAM مصرف می کنند و باعث می شوند که file system کمتر تاثیرگذار باشد.

۲) در بعضی شرایط اجرای عملیات I/O بر روی فایل های نگاشته شده در حافظه می تواند به طرز چشمگیری کندتر از خواندن عادی از فایل باشد.

۳) یکی دیگر از عیب های فایل های نگاشته شده در حافظه مربوط به فضای حافظه معماری است که می خواهد فایل را نگاشت کند. اگر حجم فایل بیشتر از حافظه قابل آدرس دهی باشد، می توان هر دفعه فقط بخش هایی از آن را نگاشت کرد که باعث می شود خواندن از آن پیچیده تر شود.

توضیحات در مورد کدهای برنامه:

توضیحات مربوط به تابع get_free_pages_count2:

این تابع به این صورت عمل می کند که ابتدا پوینتری که به اولین صفحه خالی page table اشاره می کند را در r ذخیره می کند. Struct r یک نود یک لیست پیوندی است. هر صفحه در r به صفحه بعدی خودش اشاره می کند. در صورتی که به پایان صفحات برسیم، مقدار next خالی یا null خواهد بود. به همین خاطر در این تابع از حلقه while استفاده کردیم تا هنگامی که next خالی باشد در page table جلو می رود و تعداد خانه های خالی را هر دفعه یک واحد افزایش می دهد. در پایان مقدار ذخیره شده در num_of_free مقدار مورد نظر ما خواهد بود.

```
int get_free_pages_count2() {
  struct run *r;
  int num_of_free = 0;
  r = kmem.freelist;
  while(r) {
    r = r->next;
    num_of_free++;
  }
  return num_of_free;
}
```

توضیحات مربوط به تابع mmap: در فراخوانی سیستمی mmap استراک mappedFile می‌سازیم. این استراکت تعریف شده توسط خودمان است و دارای مقادیر روبرو است. در این تابع این مقادیر را که عملا همان مقادیر ورودی تابع mmap است تکمیل می‌کنیم تا بعدا از این موارد در تابع trap.c استفاده بکنیم. یک ویژگی files هم به استراکت proc اضافه شد که آرایه‌ای هشت تایی از mmapedFileها می‌باشد. در این تابع mmapedFiles نیز آپدیت خواهد شد و این استراکت جدید به آن اضافه خواهد شد. lastAddr نیز آدرسی است که حتما پیج باید بعد از آن شروع به نوشتن بکند.

```
struct mappedFile{
  int start_addr;
  int fd;
  int valid;
  int length;
  int prot;
  int flag;
};
```

```
int lastAdrr = 0x40000000;

int mmap(int addr, int lenght, int prot, int flag, int fd, int offset) {

  struct proc* p = myproc();
  struct mappedFile mmapStruct;
  mmapStruct.length = lenght;
  mmapStruct.fd = fd;
  lastAdrr = PGROUNDUP(lastAdrr+1);
  mmapStruct.start_addr = (addr+lastAdrr);
  mmapStruct.flag = flag;
  mmapStruct.prot = prot;
  p->files[p->fileNum++] = mmapStruct;

  return lastAdrr;
}

int get_free_pages_count(void)
{
  return get_free_pages_count2();
}
```

توضیحات مربوط به تابع pageFaultHandler در trap.c

هنگامی که page fault رخ می دهد به جای اینکه مستقیما panic انجام شود ابتدا چک می کنیم که آیا این page fault به این علت بوده که در mmap صفحه اختصاص داده ایم یا خیر که اگر علت این باشد باید پیج های لازم را اختصاص بدهیم و panic انجام نشود. این موارد توسط تابع pageFaultHandler انجام می شوند. به این صورت که در ابتدا آدرسی که موجب ایجاد خطا شده است را از تابع rcr2 دریافت می کنیم. سپس چک می کنیم که آیا این آدرس در میان آدرس صفحه های mmap شده پردازه‌ای که منجر به رخ دادن خطا شده هست یا خیر. این فایل ها در files ذخیره می شوند. در صورتی که باشد، یعنی برنامه به آدرس مجازی دسترسی پیدا کرده است و فایل مورد نیاز برنامه باید لود شود. بخش از حافظه را برای خواندن فایل اختصاص می دهیم و به تابع mappages می دهیم تا فضای اختصاص داده شده به mem را به page table پردازه map کند. در صورتی که این عملیات موفقیت آمیز نباشد، تابع mappages مقدار -1 برمی گرداند و فضای اختصاص داده شده به mem را خالی می کنیم. در غیر اینصورت باید فایل مورد نظر را داخل mem بخوانیم. خانه با ایندکس توصیف کننده فایل هایی که توسط پردازه open می شوند در ofile مقدار یک را به خود می گیرند. پس اگر خانه با ایندکس fd فایل ذخیره شده در files که پردازه می خواهد به آن دسترسی پیدا کند توسط پردازه باز شده بود یعنی پردازه اجازه دسترسی به آن را دارد و عملیات خواندن فایل داخل mem آغاز می شود. در غیر اینصورت به ارور می خوریم و مقدار 0 به نشانه موفقیت آمیز نبودن عملیات برمی گردد. در پایان آدرسی که منجر به ایجاد خطا شده نیز چاپ می شود.

```
int pageFaultHandler(void)
{
  struct proc *p = myproc();
  uint addr = rcr2();
  // uint new_index = 0;
  int i = 0;
  for(i = 0; i < 8; i++) {
    if((addr >= p->files[i].start_addr) && (addr < p->files[i].start_addr+p->files[i].length))
      break;
  }

  if(i == 8) {
    return 0;
  }

  char* mem;
  mem = kalloc();
  memset(mem, 0, PGSIZE);

  if(mappages(p->pgdir, (char*)PGROUNDDOWN(addr), PGSIZE, V2P(mem), PTE_W|PTE_U) < 0) {
    cprintf("Access Denied\n");
    kfree(mem);
    return 0;
  }
  // read from file
  if(p->ofile[p->files[i].fd] == 0) {
    cprintf("File is not opened\n");
    return 0;
  }

  struct file* fd = p->ofile[p->files[i].fd];
  fileread(fd, mem, PGSIZE);

  cprintf("Address: %x\n", addr);
  return 1;
}
```

برنامه تست mmap: ابتدا یک فایل test.txt باز می‌شود و در این مرحله تابع mmap صدا زده می‌شود و خروجی text دریافت می‌شود. هنگامی که می‌خواهیم یک خانه آرایه text را تغییر دهیم page_fault رخ می‌دهد و به trap مربوطه در فایل trap.c مراجعه می‌کند و در آنجا موارد را هندل می‌کند.

```c
int main(int argc, char *argv[])
{
    int fd = open("test.txt", O_CREATE|O_RDWR);

    int length = 10;
    char* text = (char*)mmap(0, length, PROT_READ, MAP_PRIVATE, fd, 0);
    text[0] = '1';
    text[1] = '2';

    exit();
}
```