南京航空航天大学

操作系统实践

Proj4

班级： 1618001
学号： 161840230
姓名： 王可

一、 题目简述

1. 写时复制的fork（90%）

fork产生的子进程和父进程有很大的相似性：代码段一样，数据段一样，栈段一样，堆段也一样。xv6在实现fork时，会为子进程的代码、数据、堆、栈均分配物理内存，这有一些浪费，尤其是，若子进程接着调用了exec，则刚分配的物理内存又需要释放。你需要完成以下任务（请参阅教材9.3节的内容）。

- fork执行时，只为子进程创建页表，这个页表完全复制了父进程的页表（包括页框号）。同时，将子进程和父进程的可写页面标记为只读，这些页面称为写时复制页面，它们本来可以被写，只是因为写时复制技术才标记为只读，需要将这些页面与本来就是只读的页面做区分（缺页中断时需要区分处理）。可以通过页表项中的第9-11位来做标记（这几位硬件未使用，是供操作系统用的）
- 当父进程（或子进程）试图写这些页面时，会触发缺页中断，核实原因为写内存（通过error code判断）并且所写页面为写时复制页面时，再为其分配一个页框，并将原有的内容复制到此页框中，然后修改当前进程的页表项为可写。提示：将页表项的权限改为只读 *pte&=~PTE_W ； 将其设置为可写 *pte|=PTE_W 。
- 当进程消亡时，需要释放页框。但如果两个进程共享一个页框，则提前消亡的进程在释放页框后会造成存活进程无法正常运行。一种策略是为每个已分配的页框记录一个引用数，记录该页框被几个进程的页表所指，每次释放会将引用数减1，当引用数为0时再真正释放页框。**你需要在哪些地方维护这个引用数？请写到课设报告中。**
- xv6最多支持64个进程，所以引用数用1个字节表示即可（c语言中可用 char 类型）。Makefile中指明物理内存大小为256M（QEMUOPTS中的 -m 256 ），据此计算出页框总数，从而得出需要多少字节来存储引用数。为简单起见，你可以在内核里定义一个全局的数组来记录系统启动后页框被进程引用的数量（被内核引用多少次倒不必记录，这里记录的引用数以能满足copy-on-write fork的要求为准）
- 由于一个页框可能分配给两个进程，所以有可能出现两个进程同时释放页框的情况，即，在对页框的引用数做修改时可能出现竞争条件。复用kmem.lock来解决这个问题。

---

2. 开放任务(10%，未实现)

二、 实验过程

1. 写时复制的fork

1.1 全局考虑

1. 在进行fork时，会调用一个函数进行页目录的创建，包括四个部分

- 建立页目录表（setupkvm）
- 建立页目录项（walkpgdir）
- 分配页框（kalloc， mappages）
- 复制页框内容（memmove）

  实际上复制页框内容的memmove出现在分配页框的mappages之前，但为了方便理解，做如上排序。

  其中，为了实现写时复制，即是要修改后面两个部分的内容，取消新页框的分配，也就不需要复制页框内容，而为了满足页表要求，需要将页目录项重定向到父进程的页框上，并且相对修改一些Flag标志位，并且根据要求修改Val空闲位，以供后续中断时识别使用。

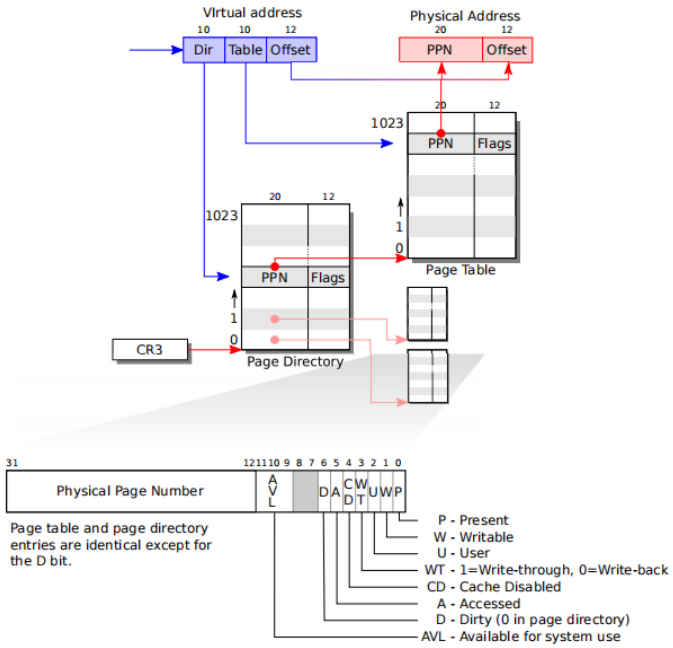
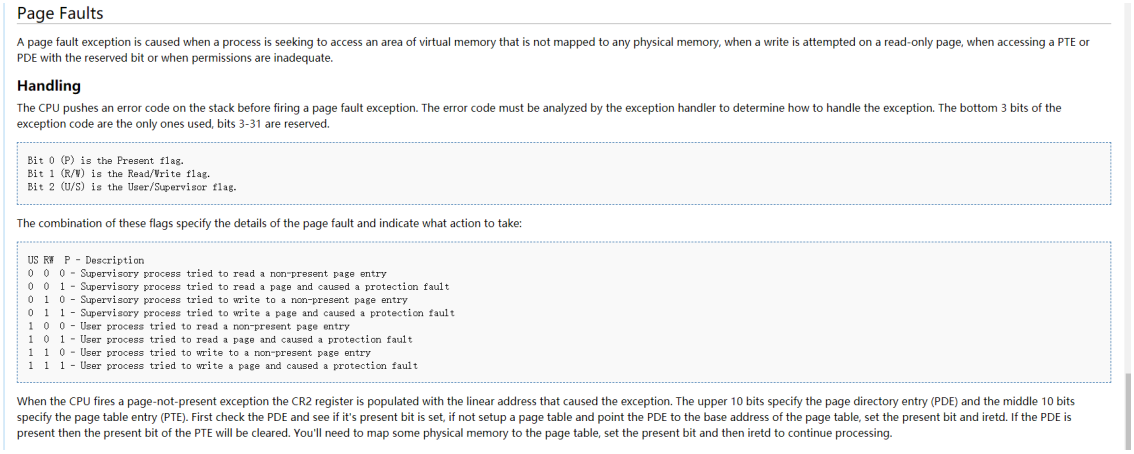


Figure 2-1. x86 page table hardware.

---

2. 关于写时复制，在实验前，我以为是在父进程或者子进程修改时，为子进程重新分配一个页框并将原本内容复制进去，同时把父进程的不可写状态改为可写。而在实现时，构想如何区分父进程和子进程，发现太过困难，因为对于树结构的进程结构而言，若出现三层以上的继承关系，父进程同时也是子进程，在其被写时很难判断修改的先后，因此重新阅读题目和书后发现是在某个写时复制进程时（无论父子）给其分配一个新的页框，而若父子都失去对原有页框的引用时，将原页框释放即可。
3. 中断处理时，通过查阅题目中给的网页链接，可以发现error code中，用户进程访问不可写页面的特征为末三位为111，因此需要在trap中对其进行判断，同时，我将写时复制的val设置为001，这一部分也需要进行判断。同时，引起错误的地址在CR2中获得，使用rcr2函数。



Page Faults

A page fault exception is caused when a process is seeking to access an area of virtual memory that is not mapped to any physical memory, when a write is attempted on a read-only page, when accessing a PTE or PDE with the reserved bit or when permissions are inadequate.

Handling

The CPU pushes an error code on the stack before firing a page fault exception. The error code must be analyzed by the exception handler to determine how to handle the exception. The bottom 3 bits of the exception code are the only ones used, bits 3-31 are reserved.

```
Bit 0 (P) is the Present flag.
Bit 1 (R/W) is the Read/Write flag.
Bit 2 (U/S) is the User/Supervisor flag.
```

The combination of these flags specify the details of the page fault and indicate what action to take:

```
US RW  P - Description
0  0   0 - Supervisory process tried to read a non-present page entry
0  0   1 - Supervisory process tried to read a page and caused a protection fault
0  1   0 - Supervisory process tried to write to a non-present page entry
0  1   1 - Supervisory process tried to write a page and caused a protection fault
1  0   0 - User process tried to read a non-present page entry
1  0   1 - User process tried to read a page and caused a protection fault
1  1   0 - User process tried to write to a non-present page entry
1  1   1 - User process tried to write a page and caused a protection fault
```

When the CPU fires a page-not-present exception the CR2 register is populated with the linear address that caused the exception. The upper 10 bits specify the page directory entry (PDE) and the middle 10 bits specify the page table entry (PTE). First check the PDE and see if it's present bit is set, if not setup a page table and point the PDE to the base address of the page table, set the present bit and iretd. If the PDE is present then the present bit of the PTE will be cleared. You'll need to map some physical memory to the page table, set the present bit and then iretd to continue processing.

4. 在维护页框引用量时需要注意的是

- 页框数量为物理存储空间大小除以页面大小即256M/4KB，即为0x10000
- 当使用pa作为下表时，应该注意pa需要向右移动12位，否则将会获得基地址的数值（远远大于数组大小）
- 访问页框引用数组应当使用物理地址，若获得的是逻辑地址，需要使用V2P函数转为物理地址。

1.2 fork

1.2.1 主要思路

在对进程进行fork时，调用了copyuvm函数进行页表的创建，因此修改此函数，使之符合上面第一条全局考虑的要求即可。

我仿照PTE_ADDR宏定义了四个辅助函数于mmu.h，用于管理pte的符号位，其实现如下：

```
// modify the flags into unwritable,  retain other flags
#define PTE_Clear_writable(pte) ((uint)(pte) & 0xFFFFFFFD)

// clear the flags units of AVL, and get ready to set these units to show "Copy on Writing"
#define PTE_Clear_Val(pte) ((uint)(pte) & 0xFFFFF1FF)

// set the flags units of AVL to be normal
#define PTE_Set_Val_Normal(pte) ((uint)(pte) | 0x0)

// set the flags units of AVL to be "Copy on Writing"
#define PTE_Set_Val_Copy_on_Writing(pte) ((uint)(pte) | 0x200)
```

通过使用以上函数，可以将原有的flag更改为我们需要的，去除了Writable位并且赋值了Val的标记，将新的标记与原有物理地址合并，可以获得新的父进程PTE，而对于子进程，则通过mappages进行映射，生成新的pte指向父进程的页框。

1.2.2 实现代码

```
pde_t*
  copyuvm_new(pde_t *pgdir, uint sz)
  {
    pde_t *d;
    pte_t *pte;
    uint pa, i, flags;
    // make a new kernal page table directory
    if((d = setupkvm()) == 0)
      return 0;
```

```
10
11      // copy all the PTE from parent
12      for(i = 0; i < sz; i += PGSIZE){
13        if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
14          panic("copyuvm: pte should exist");
15        if(!(*pte & PTE_P))
16          panic("copyuvm: page not present");
17        /* ---------------------------------------do not allocate new pages ---
    ---*/
18
19
20        pa = PTE_ADDR(*pte);
21
22        flags = PTE_FLAGS(*pte);
23
24        flags = PTE_Clear_writable(flags);
25        flags = PTE_Clear_Val(flags);
26        flags = PTE_Set_Val_Copy_on_Writing(flags);
27        // if((mem = kalloc()) == 0)
28        //   goto bad;
29        // memmove(mem, (char*)P2V(pa), PGSIZE);
30        // *pte = V2P(pa) | flags;
31        *pte = pa | flags;
32
33        if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0)
34          goto bad;
35
36        // add the reference number of the page
37        kaddRefer(pa);
38
39        lcr3(V2P(pgdir));
40
41      }
42      return d;
43
44    bad:
45      lcr3(V2P(pgdir));
46      freevm(d);
47      return 0;
48    }
```

1.3 trap

1.3.1 主要思路

首先需要判断自陷类型是否为目标类型

- PageFault 页访问错误
- 通过查询tf->err得知是用户访问不可写页面
- 通过查询pte得知是写时复制的页面报错

确认目标类型后即可展开操作，先获取当前指向的目标页，将其引用删除，同时创建新页表，复制原页表内容并重定向pte到新的页框，并且更改flags位，使得可写（因为可写之后不会进入二重判断，所以不修改Val也可以）。增加新页面的引用。

在这个过程中，我本来使用mappages重新定向，但报错remap，仔细检查代码后发现，mappages禁止向以及存在并PTE_P位有效的页面重定向。因此改为手动修改PTE的值。

1.3.2 实现代码

```
// trap.c
case T_PGFLT:
    // cprintf("get pgflt\n");
    if((tf->err & 0b111) == 0b111){   // a user process are to write on a protected page
      char *a = (char*)rcr2();
      pde_t* pgdir = myproc()->pgdir;
      Handle_trap_copy_on_writing(pgdir, a);
      // cprintf("write err end in pid: %d \n", myproc()->pid);
      }
      break;


// vm.c
int Handle_trap_copy_on_writing(pde_t *pgdir, char *a){
  pte_t *pte;
  uint pa, flags;
  char *mem;
  if((pte = walkpgdir(myproc()->pgdir, a, 0)) == 0){
        return -1;
  }

  if((((*pte) & PTE_W) == 0 ) && ((((*pte) >> 9) & 0b111) == 0b001)){  // unwritable && copy_on_writing

    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    flags = PTE_Clear_Val(flags);
    flags = flags | PTE_W;

    if((mem = kalloc()) == 0){
      kfree(mem);
      return 0;
    }
    memmove(mem, (char*)P2V(pa), PGSIZE);
    kfreeRefer(pa);
    //   remap
    *pte = V2P(mem) | flags | PTE_P;
    kaddRefer(PTE_ADDR(*pte));
    // kshowRefer(PTE_ADDR(*pte));
    lcr3(V2P(pgdir));
    return 1;
  }
  return 0;
}
```

1.4 Reference Number

使用一个字符型数组char refer[65540]来记录引用量，在kalloc.c文件中分装四个相关函数，分别用于增加引用量，减少饮用量，检查引用量为0时释放页框，和显示引用量（用于调试）代码如下：

其中，对于refer数组的争用，复用kmem.lock进行加锁处理，在初期debug时发现会卡死，经过修改尝试后判断可能是kmem.lock还没生成的时候就使用了，因此导致出错，因此模仿其他kalloc中的函数加上了对kmem.use_lock的值判断。同时kfree自带有加锁，所以提前解锁。

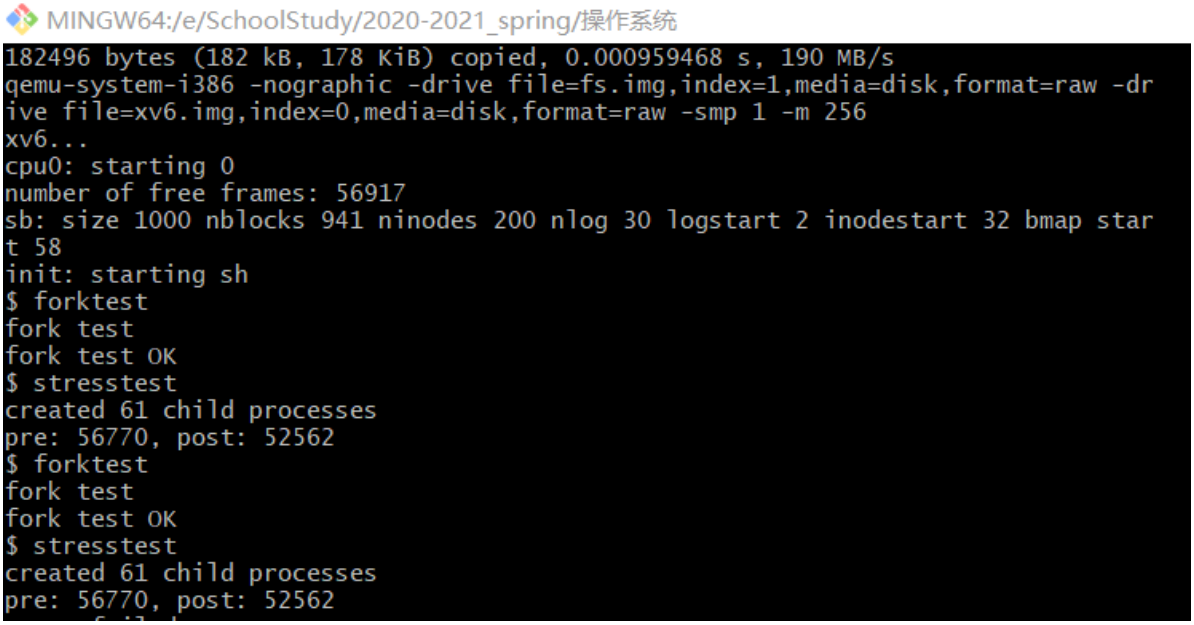
```
int kaddRefer(uint pa){
  if(kmem.use_lock)
    acquire(&kmem.lock);
  pa = (pa >> 12);
  refer[pa]++;
  if(kmem.use_lock){
    // if(refer[pa]>2)
    // cprintf("the ++ number of %d is %d\n",pa,refer[pa]);
    release(&kmem.lock);
  }

  return 1;
}

// Minus the reference number of a page
int kfreeRefer(uint pa){
  uint paa = (pa >> 12);
  if(kmem.use_lock)
    acquire(&kmem.lock);
  refer[paa]--;

  if(kmem.use_lock){
    // cprintf("the -- number of %d is %d\n",paa,refer[paa]);
    release(&kmem.lock);
  }


  return kcheckPage(pa);
}

// Check if the reference number of page is zero, which means the page
should be released.
int kcheckPage(uint pa){
  if(kmem.use_lock)
    acquire(&kmem.lock);
  if(refer[pa >> 12] <= 0){
    char *v = P2V(pa);
    if(kmem.use_lock)
      release(&kmem.lock);
    kfree(v);

    return 1;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return 0;
}

int kshowRefer(uint pa){
  pa = (pa >> 12);
  if(kmem.use_lock)
    acquire(&kmem.lock);
  cprintf("the reference number of %d is %d\n",pa,refer[pa]);
```

```
53    if(kmem.use_lock)
54      release(&kmem.lock);
55    return 1;
56  }
```

其中引用量添加的操作，实现于inituvm,allocuvm,copyuvm_new以及trap处理函数中。 而减少操作则在trap处理函数和deallocuvm中出现。

三、 实验结果

在xv6中先后输入forktest和stresstest，按要求输出了指定内容：



四、实验心得

本次实验中，在debug上花了很多多余的时间，因为一开始的错在初始化部分，什么报错信息都无法显示，后期出现很多死循环和页引用错，用了gdb和输出信息来确定错误位置。通过本次实验，我对操作系统如何分配页面有了进一步了解。