第 4 次操作系统实验报告

161810129 董世晨

一、 写时复制的 fork

1. 主要思路

使用页表项的第9位作为copy-on-write的标志位，记作PTE_COW。在copyuvm函数中，将当前的pte中的PTE_W位移动到PTE_COW位，表示该页框被多个进程共享。同时需要刷新页表并增加该页框的引用计数。

```
313  // Given a parent process's page table, create a copy
314  // of it for a child.
315  pde_t*
316- copyuvm(pde_t *pgdir, uint sz)
317  {
318    pde_t *d;
319    pte_t *pte;
320    uint pa, i, flags;
321-   char *mem;
322
323    if((d = setupkvm()) == 0)
324      return 0;
325    for(i = 0; i < sz; i += PGSIZE){
326      if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
327        panic("copyuvm: pte should exist");
328      if(!(*pte & PTE_P))
329        panic("copyuvm: page not present");
330      pa = PTE_ADDR(*pte);
331      flags = PTE_FLAGS(*pte);
332-     if((mem = kalloc()) == 0)
333        goto bad;
334-     memmove(mem, (char*)P2V(pa), PGSIZE);
335-     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
336-       goto bad;
337    }
338    return d;
```

```
321  // Given a parent process's page table, create a copy
322  // of it for a child.
323  pde_t*
324+ copyuvm_on_write(pde_t *pgdir, uint sz)
325  {
326    pde_t *d;
327    pte_t *pte;
328    uint pa, i, flags;
329
330    if((d = setupkvm()) == 0)
331      return 0;
332    for(i = 0; i < sz; i += PGSIZE){
333      if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
334        panic("copyuvm: pte should exist");
335      if(!(*pte & PTE_P))
336        panic("copyuvm: page not present");
337+     if (*pte & PTE_W) {
338+       *pte |= PTE_COW;
339+       *pte &= ~PTE_W;
340+       lcr3(V2P(myproc()->pgdir));
341+     }
342      pa = PTE_ADDR(*pte);
343      flags = PTE_FLAGS(*pte);
344+     if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0)
345        goto bad;
346+     if(kmem.use_lock)
347+       acquire(&kmem.lock);
348+     ++kmem.ref_count[pa >> PGSHIFT];
349+     if(kmem.use_lock)
350+       release(&kmem.lock);
351    }
352    return d;
```

在 trap 函数中新增对 page fault 的捕获，调用 handle_pgflt 函数：

```
78     lapiceoi();
79     break;
80
81   //PAGEBREAK: 13
82   default:
83     if(myproc() == 0 || (tf->cs&3) == 0){
```

```
79     lapiceoi();
80     break;
81
82+  case T_PGFLT:
83+    if (handle_pgflt(tf))
84+      break;
85+
86   //PAGEBREAK: 13
87   default:
88     if(myproc() == 0 || (tf->cs&3) == 0){
```

handle_pgflt 函数中，先通过错误码是否为 7 和当前页框是否有 PTE_COW 标志位来判断是否是因为写时复制造成的 page fault，如果不是，转到默认处理方式。如果是，将当前页框的 PTE_COW 位移动到 PTE_W 位，如果有超过一个进程正在使用该页表，则还需要将该页框的内容复制一份。最终无论哪种情况都需要刷新页表。

```
int handle_pgflt(struct trapframe *tf)
{
  if (tf->err != 7)
    return 0;
  struct proc *pproc = myproc();
  pte_t* pte = walkpgdir(pproc->pgdir, (void *)rcr2(), 0);
  if (!(*pte & PTE_COW))
    return 0;

  uint pa = PTE_ADDR(*pte);
  if (kmem.use_lock)
    acquire(&kmem.lock);
  char refc = kmem.ref_count[pa >> PGSHIFT];
  if (kmem.use_lock)
    release(&kmem.lock);
```

```
  if (refc == 0)
    panic("handle_pgflt: ref count == 0");

  *pte &= ~PTE_COW;
  *pte |= PTE_W;
  if (refc > 1) {
    char *mem;
    if((mem = kalloc()) == 0) {
      cprintf("Out of memory when handling copy-on-write page fault\n");
      pproc->killed = 1;
      return 1;
    }
    memmove(mem, (char*)P2V(pa), PGSIZE);
    *pte = V2P(mem) | PTE_FLAGS(*pte);

    if (kmem.use_lock)
      acquire(&kmem.lock);
    --kmem.ref_count[pa >> PGSHIFT];
    if (kmem.use_lock)
      release(&kmem.lock);
  }

  lcr3(V2P(pproc->pgdir));
  return 1;
}
```

为了解决共享页框的问题，每个页框都需要使用一个引用计数，共需要 PHYSTOP/PGSIZE 个引用计数，即 57344 个。在 kinit1 中全部初始化为-1，表示初始化时调用 freerange 做标志。

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;

void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;

  freerange(vstart, vend);
}
```

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
  char ref_count[PHYSTOP / PGSIZE];
} kmem;

void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  for (int i = 0; i < PHYSTOP / PGSIZE; ++i)
    kmem.ref_count[i] = -1;
  freerange(vstart, vend);
}
```

在 kfree 中，如果引用计数不是-1，也就是说不是初始化 freelist 的话，将引用计数减一，只有当引用计数为 0，也就是没有进程使用这个页框的时候再释放页框。在 kalloc 中将当前分配的页框引用计数置 1。

```
  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(r){
    kmem.freelist = r->next;
    free_frame_cnt--;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
```

```
  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  char* prefc = &kmem.ref_count[V2P(v) >> PGSHIFT];
  if (kmem.use_lock)
    acquire(&kmem.lock);
  char refc = *prefc;
  if (kmem.use_lock)
    release(&kmem.lock);

  if (refc != -1) {
    if (refc == 0)
      panic("kfree: ref count == 0");
    if (kmem.use_lock)
      acquire(&kmem.lock);
    refc = --*prefc;
    if (kmem.use_lock)
      release(&kmem.lock);
    if (refc > 0)
      return;
  }

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(r){
    kmem.freelist = r->next;
    free_frame_cnt--;
    kmem.ref_count[V2P(r) >> PGSHIFT] = 1;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
```

2. 细节

下面这些细节或者需要经过细致的思考，或者总结自我很长一段时间调试的教训：

1) 在每次更改页表之后都需要刷新TLB，可以通过重置CR3寄存器来刷新：

```
lcr3(V2P(myproc()->pgdir));
```

2) 在copyuvm中，需要将父进程和子进程的页表都改为copy-on-write的，因为如果父进程更改了，子进程的内容应当不变。具体而言，应当修改*pte而不是flags变量。
3) 如果kmem.use_lock为真，在每次访问和修改kmem.ref_count时都需要使用锁，反之不用。
4) 需要同时满足以下三个条件才能说明时写时复制导致的page fault：tf->trapno为T_PGFLT、tf->err为7、页框有PTE_COW标志位。
5) 如果出现写时复制导致的page fault时，该页框引用计数为1，则不需要重新分配再把该页框释放，可以直接将该页框的PTE_COW标志位移动到PTE_W位即可。
6) 如果一个地址在内核中，需要使用V2P宏转换成物理地址再获取页框号；如果地址在用户进程中，需要先获取pte，再通过PTE_ADDR(*pte)>>PGSHIFT来获取页框号。
7) kfree函数除了正常的释放页框的功能，当被freerange调用时，它需要将所有页框连接到freelist中。我通过设置引用计数为-1来判断是否由freerange调用。

3. 实验结果

```
init: starting sh
$ forktest
fork test
fork test OK
$ stresstest
created 61 child processes
pre: 56772, post: 52564
```

二、 其他节省物理内存的技术

1. 复用内核页表

和setupkvm类似，我新建了一个setupuvm函数，该函数不复制所有内核页表，而是直接将内核页表kpgdir中的内容复制过来。值得注意的是，需要将内核页框引用计数都增加一，来防止进程结束时释放内核页框。将除了初始化kpgdir的其余setupkvm函数调用改为setupuvm函数，就完成了复用内核页表的功能。

```
pde_t* setupuvm(void)
{
  pde_t *pgdir = (pde_t*)kalloc();
  if(!pgdir)
    return 0;
  memmove(pgdir, kpgdir, PGSIZE);
  for (int i = 0; i < NPTENTRIES; ++i)
    if (pgdir[i] & PTE_P) {
      if (kmem.use_lock)
        acquire(&kmem.lock);
      ++kmem.ref_count[pgdir[i] >> PGSHIFT];
      if (kmem.use_lock)
        release(&kmem.lock);
    }
  return pgdir;
}
```

2. 分析实验结果

```
init: starting sh
$ forktest
fork test
fork test OK
$ stresstest
created 61 child processes
pre: 56964, post: 56659
```

如上图所示，运行前空闲页框数为56964，创建61个子进程之后空闲页框数位56659个，共用去305个页框，即正好**每个子进程需要使用5个页框**。经过思考和分析，这五个页框分别用于：

1) 子进程的页目录表pgdir；
2) 子进程的第一个页目录表项对应的页表；
3) 子进程修改了data数组导致分配的页框；
4) 在创建子进程时allocproc函数中，会分配一个页框用于子进程的内核栈；
5) 父进程在执行完fork后，它的所有可读的页框都被标记上了PTE_COW，当从fork返回时，第一件事做的是将返回值等参数出栈，这个操作会修改父进程的用户栈内容，因此立刻会触发copy-on-write page fault，从而复制一个新的父进程的用户栈。

3. 展望

我逐一思考了以上五个页框能否有进一步节省的空间：

1) 对于页目录表，我认为，由于子进程和父进程的页表不可能完全相同，它的页目录表也不可能完全相同，因此无法共享页目录表；
2) 对于第一个页目录表项对应的页表，我认为子进程和父进程必然不可能完全相同，只要出现一次copy-on-write page fault，页表就发生了改变，因此，延迟分配页表的意义不大；
3) 对于修改了data数组导致分配的页框，观察到所有的子进程虽然修改了内容，但是都修改了相同的内容，修改之后的页框内容完全一致，可以将所有的子进程修改后的页表项指向同一个页框。虽然这可能比较难以实现，但是不失为一种思路；
4) 对于子进程的内核栈，fork创建的子进程的内核栈应该不和父进程一致，而且，陷入内核时trapframe会保存在子进程的内核栈中，如果此时发生缺页可能会有无法预料的问题；
5) 对于父进程的用户栈，我认为，这个页框无法节省，因为只要父进程或子进程在执行时修改了栈，就会导致分配新页框，比如从fork返回。

另外，也可以选择节省内核的内存使用来节省物理内存，比如用6个比特而不是char的8个比特来保存页框的引用计数，但是，xv6现有的设计已经处于空间和时间取舍的完美的平衡了，牺牲效率来强行节省内存得不偿失。

综上，以上的方案，或者难以实现，或者无法实现，或者意义不大，因此我认为我提交的代码对页框的节省已经到了某个极限。