**实验五 xv6 lazy page allocation**

**5.1 实验目的**

1. 了解惰性空间分配

2. 对惰性空间分配进行学习

**5.2 实验内容**

5.2.1 Eliminate allocation from sbrk()（easy）

您的第一个任务是从sbrk(n)系统调用实现中删除页面分配，这是sysproc.c中的函数sys\_sbrk()。Sbrk(n)系统调用将进程的内存大小增加n个字节，然后返回新分配区域的开始（即旧大小）。新的sbrk(n)应该将进程的大小（myproc()->sz）增加n，然后返回旧的大小。它不应该分配内存——因此您应该删除对growproc()的调用（但仍然需要增加进程的大小！）。

5.2.2 Lazy allocation（moderate）

修改trap.c中的代码，通过将新分配的物理内存页映射到出错地址，然后返回到用户空间，让进程继续执行，从而从用户空间响应页面错误。您应该在生成“usertrap（）：…”消息的printf调用之前添加代码。修改您需要的任何其他xv6内核代码，以使echo hi正常工作。

5.2.3 Lazytests and Usertests（moderate）

我们为您提供了lazytests，这是一个xv6用户程序，用于测试一些可能会给您的惰性内存分配器带来压力的特定情况。修改内核代码，使所有lazytests和usertests都通过。。

5.2.4 submit

提交实验。

**5.3 实验过程**

5.3.1 eliminate allocation from sbrk()

（1）在sysproc.c中修改sys\_sbrk()函数，删除sbrk系统调用中原有的分配物理地址机制，即删除growproc，只单纯地增加进程的sz变量，当用户要求新增大小为负时需要调用uvmdealloc函数释放物理地址。

uint64

sys\_sbrk(void)

{

  int addr;

  int n;

  struct proc \*p = myproc();

  if (argint(0, &n) < 0)

    return -1;

  addr = p->sz; // old sz

  p->sz += n;

  if (n < 0)

  {

    uvmdealloc(p->pagetable, addr, p->sz);

  }

  return addr;

}

5.3.2 lazy allocation

（1）前面我们修改了sbrk系统调用使得它可以迅速地处理用户的空间申请，但是用户申请的地址在页表中依然是invalid的状态，当用户试图访问这些地址时就会触发缺页异常陷入内核，而我们就需要在usertrap函数中处理缺页。在RISC-V中，缺页异常号是13和15（分别对应读和写），它们会保存在scause寄存器中，而导致缺页的地址会保存在stval寄存器中。具体的处理代码如下：

void

usertrap(void){

  .......

if (r\_scause() == 8) {

  .......

} else if (r\_scause() == 13 || r\_scause() == 15)

   // page fault

  {

    uint64 va = r\_stval();

    if (va >= p->sz || va < p->trapframe->sp) {

      // page-faults on a virtual memory address higher than any allocated with sbrk()

      // or lower than the stack. In xv6, heap is higher than stack

      p->killed = 1;

    } else {

      uint64 ka = (uint64) kalloc();

      if (ka == 0){

        p->killed = 1;

      } else {

        memset((void \*)ka, 0, PGSIZE);

        va = PGROUNDDOWN(va);

        if(mappages(p->pagetable, va, PGSIZE, ka, PTE\_W|PTE\_R|PTE\_U) != 0) {

          kfree((void \*)ka);

          p->killed = 1;

        }

      }

    }

  } else {

  .......

}

（2）若进程需要释放申请会因为没有真正的空间报错，修改uvmunmap使其不报错。

void

uvmunmap(pagetable\_t pagetable, uint64 va, uint64 npages, int do\_free)

{

  uint64 a;

  pte\_t \*pte;

  if((va % PGSIZE) != 0)

    panic("uvmunmap: not aligned");

  for(a = va; a < va + npages\*PGSIZE; a += PGSIZE){

    if((pte = walk(pagetable, a, 0)) == 0){

      //panic("uvmunmap: walk");

      continue;

    }

    if((\*pte & PTE\_V) == 0){

      //panic("uvmunmap: not mapped");

      //it is ok for pagefault

      continue;

    }

    if(PTE\_FLAGS(\*pte) == PTE\_V)

      panic("uvmunmap: not a leaf");

    if(do\_free){

      uint64 pa = PTE2PA(\*pte);

      kfree((void\*)pa);

    }

    \*pte = 0;

  }

}

（4）运行lazy allocation



5.3.3 lazytests and usertests

（1）这部分完善lazy allocation的鲁棒性，之前处理的usertrap中lazy allocation机制只能满足从用户态触发的缺页，但read和write最终会调用内核中copyin和copyout函数，需要进行处理。

（2）copyin函数的功能是从用户地址空间复制数据到内核地址空间（两者页表不同），而它的实现就是通过walkaddr函数手动实现地址翻译的过程，得到用户数据的物理地址，然后将用户物理地址空间的数据复制到内核地址。若用户提供的虚拟地址还未分配的话就需要先分配物理页，然后重复上面的操作。

int

copyin(pagetable\_t pagetable, char \*dst, uint64 srcva, uint64 len)

{

  uint64 n, va0, pa0;

  struct proc \*p = myproc();

  while(len > 0){

    va0 = PGROUNDDOWN(srcva);

    pa0 = walkaddr(pagetable, va0);

    if(pa0 == 0) {

      // allow for lazy allocation

      if (va0 >= p->sz || va0 < p->trapframe->sp) {

        return -1;

      } else {

        pa0 = (uint64) kalloc();

        if (pa0 == 0) {

          p->killed = 1;

        } else {

          memset((void \*)pa0, 0, PGSIZE);

          va0 = PGROUNDDOWN(va0);

          if(mappages(p->pagetable, va0, PGSIZE, pa0, PTE\_W|PTE\_R|PTE\_U) != 0) {

            kfree((void \*)pa0);

            p->killed = 1;

          }

        }

      }

    }

    n = PGSIZE - (srcva - va0);

    if(n > len)

      n = len;

    memmove(dst, (void \*)(pa0 + (srcva - va0)), n);

    len -= n;

    dst += n;

    srcva = va0 + PGSIZE;

  }

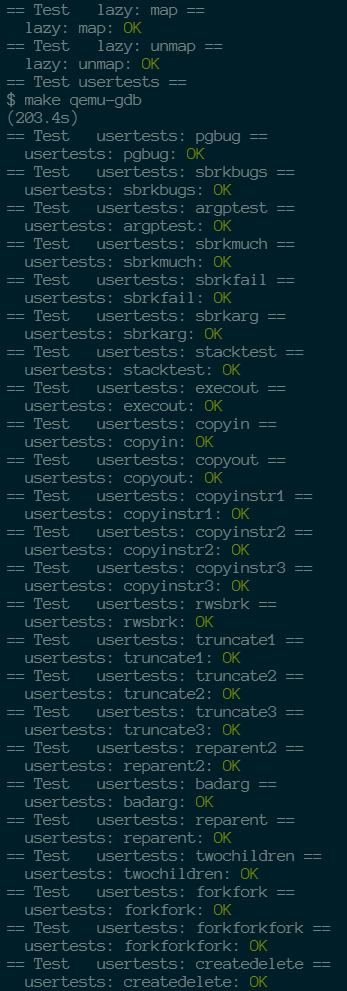
  return 0;

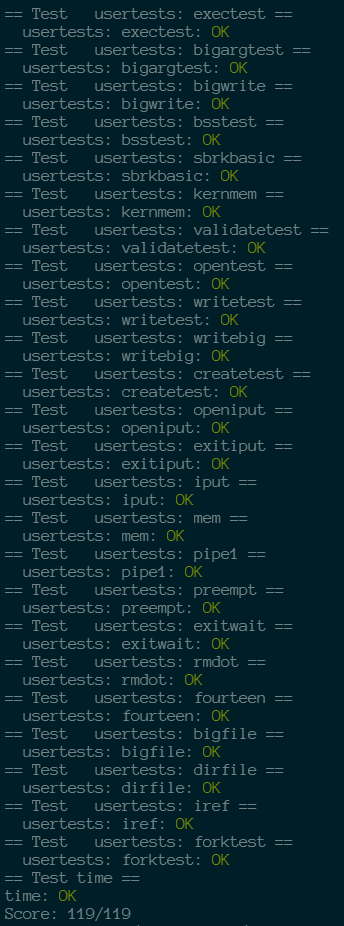
}

（3）copyout函数修改与copyin类似。

**5.4 实验结果**

通过make grade对所有实验内容进行测试，结果如下图所示。





**5.5 实验小结**

本次实验是对操作系统的一种优化，即对提高效率采用的lazy allocation，惰性空间分配是直到真正需要的时候再进行空间的分配以减少响应时间，而如果从来没有用到这个对象的情况下可以减少分配。这是一种效率较高的优化方法，可以部分简化空间的分配，使操作系统更加高效。