# Lab5 xv6 lazy page allocation

### 实验提要：

O/S可以在页表硬件上使用的许多巧妙的技巧之一是延迟分配用户空间堆内存。Xv6应用程序使用`sbrk（）`系统调用向内核请求堆内存。在我们给您的内核中，`sbrk（）`分配物理内存并将其映射到进程的虚拟地址空间。但是，有些程序使用`sbrk（）`来请求大量内存，但从未使用大部分内存，例如实现大型稀疏数组。为了对这种情况进行优化，复杂的内核延迟地分配用户内存。也就是说，`sbrk（）`不分配物理内存，只是记住分配了哪些地址。当进程第一次尝试使用任何给定的内存页时，CPU会生成一个页错误，内核通过分配物理内存、将其归零并映射它来处理该错误。在本实验中，我们要向xv6添加这个延迟分配特性。

## Eliminate allocation from sbrk()

### 实验目的

Your first task is to delete page allocation from the sbrk(n) system call implementation, which is the function sys\_sbrk() in sysproc.c. The sbrk(n) system call grows the process's memory size by n bytes, and then returns the start of the newly allocated region (i.e., the old size). Your new sbrk(n) should just increment the process's size (myproc()->sz) by n and return the old size. It should not allocate memory -- so you should delete the call to growproc() (but you still need to increase the process's size!).

您的第一个任务是从`sbrk（n）`系统调用实现中删除页面分配，该实现是`sysproc.c`中的`sys\_sbrk（）`函数。

### 实验步骤

1. 在`kernel/sysproc.c`修改 `sys\_sbrk()` 函数, 删去原本调用的 `growproc()` 函数, 增加 `myproc()->sz`

uint64

sys\_sbrk(void)

{

  int addr;

  int n;

  if(argint(0, &n) < 0)

    return -1;

  addr = myproc()->sz;

  // 释放内存

  // if(growproc(n) < 0)

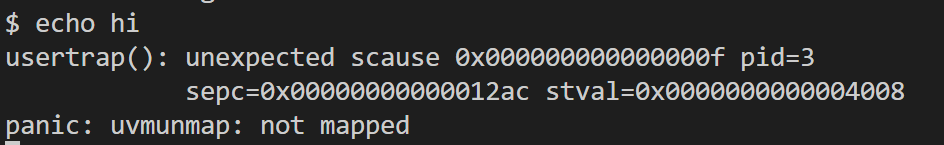
  //   return -1;

  myproc()->sz += n;

  return addr;

}

1. 实验验证结果，在终端输入make qemu启动xv6后echo hi，得到的结果符合预期



### 实验中遇到的问题和解决办法

比较简单的一个小实验，弄明白sbrk()函数的具体功能就好。

### 实验感想

先通过一个小实验让我们逐步去接触懒分配的方法，这个实验的设置层次深入，效果非常好。

## Lazy allocation

### 实验目的

Modify the code in trap.c to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing. You should add your code just before the printf call that produced the "usertrap(): ..." message. Modify whatever other xv6 kernel code you need to in order to get echo hi to work.

修改trap.c中的代码，以通过在故障地址处映射新分配的物理内存页面，然后返回用户空间以使进程继续执行，来响应用户空间中的页面错误。 您应该在产生“ usertrap（）：...”消息的printf调用之前添加代码。 修改您需要的其他xv6内核代码，以使echo hi起作用。

### 实验步骤

1. 在 `kernel/trap.c` 的 `usertrap()` 中添加对 page fault 的处理.

根据实验指导书提示, 当 r\_scause() 的值为 13 和 15 时为需要处理的 page fault 情况。系统调用的中断码是 8，page fault 的中断码是 13 和 15。因此，这里我们增加对r\_scause()中断原因进行判断，如果是 13 或是 15，则说明没有找到地址。错误的虚拟地址被保存在了 STVAL 寄存器中，我们中断判断时，如果出错（虚拟地址不合法或者没有成功映射到物理地址），也要终止进程。

    syscall();

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

    char \*pa;

    if((pa = kalloc()) != 0) {

      uint64 va = PGROUNDDOWN(r\_stval());

      memset(pa, 0, PGSIZE);

      if(mappages(p->pagetable, va, PGSIZE, (uint64)pa, PTE\_W|PTE\_R|PTE\_U) != 0) {

          kfree(pa);

          printf("usertrap(): mappages() failed\n");

          p->killed = 1;

      }

    } else {

      printf("usertrap(): kalloc() failed\n");

      p->killed = 1;

    }

  } else if((which\_dev = devintr()) != 0){

    // ok

  } else {

2.处理 `uvmunmap` 的报错

因为`uvmunmap` 是用来释放内存调用的，释放内存时，页表内有些地址并没有实际分配内存，因此没有进行映射。如果在 `uvmunmap` 中发现了没有映射的地址，不需要 panic.

具体方法是修改`kernal/vm.c`的`uvmunmap()`函数，注释掉那两行`panic`，并continue。

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");

      continue;

    if(PTE\_FLAGS(\*pte) == PTE\_V)

      panic("uvmunmap: not a leaf");

    if(do\_free){

      uint64 pa = PTE2PA(\*pte);

      kfree((void\*)pa);

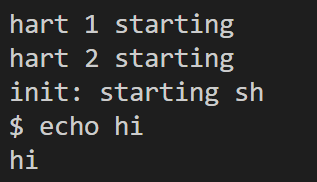
    }

    \*pte = 0;

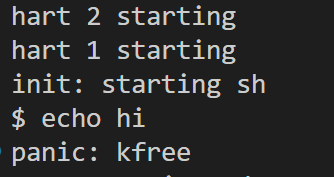
  }

}

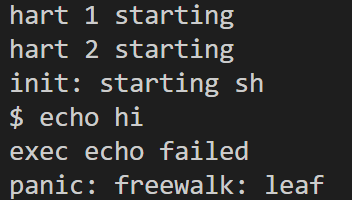
实验结果：



### 实验中遇到的问题和解决办法：



遇到kfree的panic，解决办法是在注释uvmunmap()`函数的那两行`panic`后加上continue防止执行完该函数后面的代码。



遇到freewalk：leaf的panic，原因是 `usertrap()` 中处理 page fault 时未对 `va` 使用 `PGROUNDDOWN()` 向下取整，这会导致多映射一页, 从而页面释放时多映射的一页未被取消映射, 从而引发` panic`。

### 实验心得

遇到各种bug并debug的过程太艰难，还是得把函数的含义先理解清楚再写代码，这可以省去大量的debug时间。

## Lazytests and Usertests

### 实验目的：

We've supplied you with lazytests, an xv6 user program that tests some specific situations that may stress your lazy memory allocator. Modify your kernel code so that all of both lazytests and usertests pass.

我们为您提供了lazytests，这是一个xv6用户程序，它可以测试可能会给您的惰性内存分配器带来压力的某些特定情况。

### 实验步骤：

1.修改uvmcopy函数，处理当父进程向子进程拷贝时出现的 Lazy allocation。`fork()` 是通过 `uvmcopy()` 来进行父进程页表即用户空间向子进程拷贝的. 而对于 `uvmcopy()` 的处理和 `uvmunmap()` 是一致的, 只需要将 PTE 不存在和无效的两种情况由引发 panic 改为 continue 跳过即可。

  for(i = 0; i < sz; i += PGSIZE){

    if((pte = walk(old, i, 0)) == 0)

      // panic("uvmcopy: pte should exist");

      continue;

    if((\*pte & PTE\_V) == 0)

      // panic("uvmcopy: page not present");

      continue;

    pa = PTE2PA(\*pte);

    flags = PTE\_FLAGS(\*pte);

    if((mem = kalloc()) == 0)

      goto err;

    memmove(mem, (char\*)pa, PGSIZE);

    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){

      kfree(mem);

      goto err;

    }

  }

  return 0;

2.参照growproc()函数，对sysy\_sbrk()函数进行修改，对‘add+n’的值要求大于等于PGROUNDUP(p->trapframe->sp)，而且还要对’add+n.=addr’进行防溢出操作，避免’add+n’由正变符造成’p->sz’变小。

uint64

sys\_sbrk(void)

{

  int addr;

  int n;

  struct proc \*p;

  if(argint(0, &n) < 0)

    return -1;

  p = myproc();

  addr = p->sz;

  if(n >= 0 && addr + n >= addr){

    p->sz += n;

  } else if(n < 0 && addr + n >= PGROUNDUP(p->trapframe->sp)){

    p->sz = uvmdealloc(p->pagetable, addr, addr + n);

  } else {

    return -1;

  }

//  if(growproc(n) < 0)

//    return -1;

  return addr;

}

3.修改usertrap()，当page fault的虚拟地址超过p->sz或者低于用户栈时直接终止进程.

void

usertrap(void)

{

  ......

//new code

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

    char \*pa;

 uint64 va = r\_stval();

    // 虚拟地址超过 `p->sz` 或低于用户栈时，杀死进程

    if(va >= p->sz){

      printf("usertrap(): invalid va=%p higher than p->sz=%p\n",

             va, p->sz);

      p->killed = 1;

      goto end;

    }

    if(va < PGROUNDUP(p->trapframe->sp)) {  // new code

      printf("usertrap(): invalid va=%p below the user stack sp=%p\n",

             va, p->trapframe->sp);

      p->killed = 1;

      goto end;

    }

    if ((pa = kalloc()) == 0) {

        printf("usertrap(): kalloc() failed\n");

        p->killed = 1;

        goto end;

    }

    memset(pa, 0, PGSIZE);

    if (mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64) pa, PTE\_W | PTE\_R | PTE\_U) != 0) {

        kfree(pa);

        printf("usertrap(): mappages() failed\n");

        p->killed = 1;

        goto end;

    }

  } else if((which\_dev = devintr()) != 0){

    // ok

  } else {

    printf("usertrap(): unexpected scause %p pid=%d\n", r\_scause(), p->pid);

    printf("            sepc=%p stval=%p\n", r\_sepc(), r\_stval());

    p->killed = 1;

  }

  end:    //new code

  if(p->killed)

    exit(-1);

  // give up the CPU if this is a timer interrupt.

  if(which\_dev == 2)

    yield();

  usertrapret();

}

4.修改’kernel/vm.c’中的’walkaddr’函数。

因为read()/write()函数在使用未分配物理内存时最终会调用 `copyin()` 和 `copyout()` 来完成用户空间到内核空间的读写，而这两个函数本质都要调用`walkaddr()` 函数来得到物理地址。

当PTE无效或不存在，PTE\_U标志位缺失的情况都是可行的，现在要使他们成立。

具体逻辑实现如下：

uint64

walkaddr(pagetable\_t pagetable, uint64 va)

{

  pte\_t \*pte;

  uint64 pa;

  struct proc \*p=myproc();

  if(va >= MAXVA)

    return 0;

  pte = walk(pagetable, va, 0);

  if(pte == 0 || (\*pte & PTE\_V) == 0) {

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

        char \*pa;

        if ((pa = kalloc()) == 0) {

            return 0;

        }

        memset(pa, 0, PGSIZE);

        if (mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE,

                     (uint64) pa, PTE\_W | PTE\_R | PTE\_U) != 0) {

            kfree(pa);

            return 0;

        }

    } else {

        return 0;

    }

  }

  if((\*pte & PTE\_U) == 0)

    return 0;

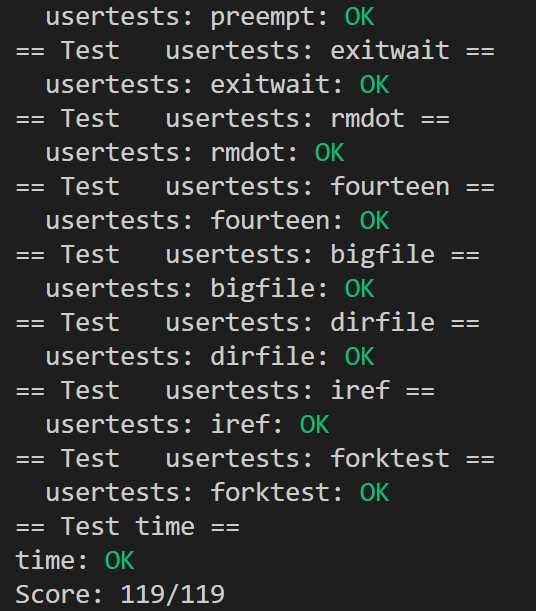
  pa = PTE2PA(\*pte);

  return pa;

}

### 实验结果：

退出xv6后在终端输入./grade-labs-lazy



符合预期

### 实验中遇到的问题和解决办法

头文件冲突：需要先明确项目的结构，不能看到有有文件报错就直接添加声明。

### 实验心得

如果在某一步遇到了一些关于free的panic，可能是某些内存没有释放干净，并且当我们调试的时候可以不用直接usertests，可以直接在这条命令之后跟上我们要调试的模块，可以大大缩短调试的时间。