Lab5: xv6 lazy page allocation 实验报告

- xv6 中使用`sbrk()`系统调用向内核请求堆内存。在原有代码中，`sbrk()`直接分配物理内存并将其映射到进程的虚拟地址空间，因此内核可能需要很长时间才能为大型请求分配和映射内存。为了让`sbrk()`在这些情况下更快地完成，复杂的内核延迟分配用户内存。也就是说，`sbrk()`不分配物理内存，而是只记住分配了哪些用户地址，并在用户页表中将这些地址标记为无效。当进程首次尝试使用懒分配内存的任何给定页面时，CPU 会生成page fault interrupt（缺页中断），此时内核再通过分配物理内存、将其置零和映射来处理该页面错误。

---

1. 首先切换分支：

```
$ git fetch
$ git checkout lazy
$ make clean
```

Eliminate allocation from sbrk() (easy)

1. 删除`sbrk(n)`系统调用中的页面分配代码（位于*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部分（即旧的大小）。新的`sbrk(n)`应该只将进程的大小（`myproc()->sz`）增加n，然后返回旧的大小。它不应该分配内存——因此应该删除对`growproc()`的调用（但是仍然需要增加进程的大小！）。

```
uint64
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(n < 0) {
    // 如果缩小空间，则立刻释放
    myproc()->sz = uvmdealloc(myproc()->pagetable, addr, addr + n
  } else {
    // 懒分配：只修改地址空间大小但不分配内存
    myproc()->sz += n;
  }
  return addr;
}
```

2. 启动xv6，键入 `echo hi`

```
220110512@comp1:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive fil
e=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x0000000000001272 stval=0x0000000000004008
panic: uvmunmap: not mapped
```

- “`usertrap(): …`”这条消息来自*trap.c*中的用户陷阱处理程序；它捕获了一个不知道如何处理的异常。
- panic 显示出现 `uvmunmap: not mapped` 错误，这是由于 uvmunmap 函数在释放内存时，发现内存页无效。
  - 因为此时我们只是修改了 sz，并未将分配的用户地址进行标记，也并未真正分配物理内存。

Lazy allocation (moderate)

修改kernel/trap.c 中的代码，通过在故障地址映射新分配的物理内存页面来响应用户空间的缺页错误，然后返回用户空间让进程继续执行。

这个任务即真正实现Lazy allocation：当系统发生缺页异常时，就会进入到`usertrap`函数中，此时`scause`寄存器保存的是异常原因（13为page load fault，15为page write fault），`stval`是引发缺页异常的地址

1. 在 *kernel/trap.c* 中的usertrap函数中，找到中断处理的逻辑进行更改。

- 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**，我们取出该地址进行懒分配。

```
    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if (r_scause() == 13 || r_scause() == 15){
    // 获取错误页面地址
    uint64 addr = r_stval();
    // 判断是否是懒分配引起的（如果是，则错误的地址在 p->sz 内）
    if(lazy_allocate(addr)<0){
        p->killed = 1;
    }
  } 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;
  }
```

2. 在kernel/proc.c中定义lazy_allocate函数：

```
int
lazy_allocate(uint64 va){
  uint64 pa = (uint64)kalloc();
  struct proc *p = myproc();
  if (pa == 0) {
    return -1;
  } else if (va >= p->sz || va <= PGROUNDDOWN(p->trapframe->sp)) {
    // 不是由懒分配引起
    kfree((void*)pa);
    return -1;
  } else {
    va = PGROUNDDOWN(va);
    // 分配新的内存块之前，使用memset函数将其初始化为0
    memset((void*)pa, 0, PGSIZE);
    // 建立映射：将一个物理内存页面映射到指定进程的虚拟地址空间中
    if (mappages(p->pagetable, va, PGSIZE, pa, PTE_W | PTE_U | PTE_R) != 0) {
      kfree((void*)pa);
      return -1;
    }
  }
  return 0;
}
```

3. 在kernel/defs.h中添加声明：

```
106  void            procdump(void);
107  int             lazy_allocate(uint64 va);
```

4. 更改 kernel/vm.c中uvmunmap 函数的内容，由于释放内存时，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，直接跳过就行，不需要 panic

```
182   for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
183     if((pte = walk(pagetable, a, 0)) == 0) // 页表项不存在
184       continue;
185       // panic("uvmunmap: walk");
186     if((*pte & PTE_V) == 0) //页表项无效
187       continue;
188       // panic("uvmunmap: not mapped");
189     if(PTE_FLAGS(*pte) == PTE_V)
190       panic("uvmunmap: not a leaf");
191     if(do_free){
192       uint64 pa = PTE2PA(*pte);
193       kfree((void*)pa);
194     }
195     *pte = 0;
196   }
```

5. 启动xv6，键入 `echo hi`

```
220110512@comp1:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=
x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ echo hi
hi
```

正常执行

Lazytests and Usertests (moderate)

完善初级版本的lazy allocation的各项功能，解决以下问题：

- 处理负的 sbrk() 参数
- 如果导致缺页错误的虚拟地址高于任何 sbrk() 分配的内存地址，则杀死进程
- 在 fork() 中正确处理父进程到子进程的内存复制
- 处理以下情况：一个进程给系统调用（如 read / write）传入了一个合法的地址，但是地址的内存还没有分配
- 正确处理超出内存的情况：如果 kalloc() 在缺页错误处理中失败了，则杀死进程
- 处理缺页错误中访问用户栈之下的非法空间

1. 针对第一点，即在 `sys_sbrk` 函数中的 `if(n < 0)` 部分调用 `uvmdealloc` 取消分配，在任务一的写法中已经实现
2. 针对第二、五、六点，加入 `p-> killed =1` 即可，在任务二中已经完成

3. 针对第三点，fork 函数在创建进程时会调用 uvmcopy 函数。由于没有实际分配内存，因此，在这里，忽略 pte 无效和不存在，继续执行代码

```
for(i = 0; i < sz; i += PGSIZE){
  if((pte = walk(old, i, 0)) == 0)  //页表项不存在
    continue;
    // panic("uvmcopy: pte should exist");
  if((*pte & PTE_V) == 0) //页表项无效
    continue;
    // panic("uvmcopy: page not present");
  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;
```

4. 针对第四点，当进程通过 `read` 或 `write` 等系统调用时，由于进程利用系统调用已经到了内核中，页表已经切换为内核页表，无法直接访问虚拟地址。因此，需要通过 `walkaddr` 将虚拟地址翻译为物理地址。这里如果没找到对应的物理地址，就调用 `lazy_allocate` 分配一个

```
  pte = walk(pagetable, va, 0);
  // if(pte == 0)
  //   return 0;
  // if((*pte & PTE_V) == 0)
  //   return 0;
  if(pte == 0 || (*pte & PTE_V) == 0){
    if(lazy_allocate(va)<0){
      return 0;
    }
    pte = walk(pagetable, va, 0);
  }
  if((*pte & PTE_U) == 0)
    return 0;
  pa = PTE2PA(*pte);
  return pa;
}
```

5. 执行测试命令：`lazytests`

```
220110512@comp1:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=
x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
test lazy unmap: OK
running test out of memory
test out of memory: OK
ALL TESTS PASSED
$ usertests
```

执行测试命令：`usertests`

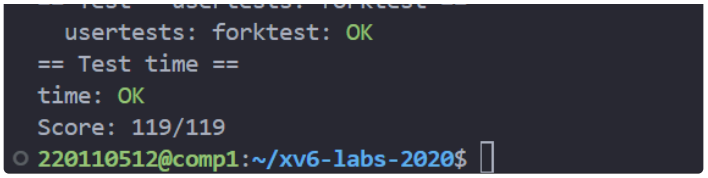
```
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

结果截图

执行命令`make grade`

```
$ make qemu-gdb
(7.2s)
== Test   lazy: map ==
  lazy: map: OK
== Test   lazy: unmap ==
  lazy: unmap: OK
== Test usertests ==
$ make qemu-gdb
(138.0s)
== Test   usertests: pgbug ==
  usertests: pgbug: OK
== Test   usertests: sbrkbugs ==
  usertests: sbrkbugs: OK
== Test   usertests: argptest ==
  usertests: argptest: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: sbrkfail ==
  usertests: sbrkfail: OK
== Test   usertests: sbrkarg ==
  usertests: sbrkarg: OK
== Test   usertests: stacktest ==
  usertests: stacktest: OK
== Test   usertests: execout ==
  usertests: execout: OK
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: rwsbrk ==
  usertests: rwsbrk: OK
== Test   usertests: truncate1 ==
  usertests: truncate1: OK
== Test   usertests: truncate2 ==
  usertests: truncate2: OK
== Test   usertests: truncate3 ==
  usertests: truncate3: OK
== Test   usertests: reparent2 ==
  usertests: reparent2: OK
== Test   usertests: badarg ==
  usertests: badarg: OK
== Test   usertests: reparent ==
  usertests: reparent: OK
== Test   usertests: twochildren ==
  usertests: twochildren: OK
== Test   usertests: forkfork ==
  usertests: forkfork: OK
== Test   usertests: forkforkfork ==
  usertests: forkforkfork: OK
== Test   usertests: createdelete ==
  usertests: createdelete: OK
== Test   usertests: linkunlink ==
  usertests: linkunlink: OK
```

```
== Test   usertests: linktest ==
  usertests: linktest: OK
== Test   usertests: unlinkread ==
  usertests: unlinkread: OK
== Test   usertests: concreate ==
  usertests: concreate: OK
== Test   usertests: subdir ==
  usertests: subdir: OK
== Test   usertests: fourfiles ==
  usertests: fourfiles: OK
== Test   usertests: sharedfd ==
  usertests: sharedfd: OK
== Test   usertests: exectest ==
  usertests: exectest: OK
== Test   usertests: bigargtest ==
  usertests: bigargtest: OK
== Test   usertests: bigwrite ==
  usertests: bigwrite: OK
== Test   usertests: bsstest ==
  usertests: bsstest: OK
== Test   usertests: sbrkbasic ==
  usertests: sbrkbasic: OK
== Test   usertests: kernmem ==
  usertests: kernmem: OK
== Test   usertests: validatetest ==
  usertests: validatetest: OK
== Test   usertests: opentest ==
  usertests: opentest: OK
== Test   usertests: writetest ==
  usertests: writetest: OK
== Test   usertests: writebig ==
  usertests: writebig: OK
== Test   usertests: createtest ==
  usertests: createtest: OK
== Test   usertests: openiput ==
  usertests: openiput: OK
== Test   usertests: exitiput ==
  usertests: exitiput: OK
== Test   usertests: iput ==
  usertests: iput: OK
== Test   usertests: mem ==
  usertests: mem: OK
== Test   usertests: pipe1 ==
  usertests: pipe1: OK
== Test   usertests: preempt ==
  usertests: preempt: OK
== Test   usertests: exitwait ==
  usertests: exitwait: OK
== Test   usertests: rmdot ==
  usertests: rmdot: OK
== Test   usertests: fourteen ==
  usertests: fourteen: OK
== Test   usertests: bigfile ==
  usertests: bigfile: OK
== Test   usertests: dirfile ==
  usertests: dirfile: OK
== Test   usertests: iref ==
  usertests: iref: OK
== Test   usertests: forktest ==
```


usertests: forktest: OK
== Test time ==
time: OK
Score: 119/119
220110512@comp1:~/xv6-labs-2020$