# 2023/09/26~2023/09/27

## XV6第四章

依然是第四章的内容，lazy allocation。从我的角度看来，懒分配是对进程总是倾向于多分配内存这一问题的应对措施，而并不是一个优化系统的方案。当进程总是按需申请内存时，懒分配频繁地触发pagefault中断将极大地降低系统的运行效率。

然而，在进程总是倾向于多分配内存的情况下，降低那么一点运行效率总比内存爆满而导致系统崩溃好的多。

## Lab5 Lazy

实验还是非常简单，一个easy两个moderate，而且大部分内容Kasshoek教授在网课上都已经提前讲解过了，这里就三合一直接上代码了。

对于每一个调用sbrk以增长自己的内存空间的进程，都先仅仅扩大进程的sz，而暂时搁置对内存的分配。对于每个缩小自己内存空间的请求，sz缩小的同时，内存也将被立即释放。这里处于系统调用本身尽量简洁的原则，更改sbrk调用的growproc以达成目的。

int

growproc(int n)

{

  uint sz;

  struct proc \*p = myproc();

  sz = p->sz;

  if(n > 0){

    p->sz+=n;

  } else if(n < 0){

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

    p->sz = sz;

  }

  return 0;

}

那么进程调用懒分配增长的内存显然会触发一个pagefault，在这个实验中表现为scause的值为0xf或0xd。对于这样的pagefault，判断触发pagefault的va是否在p->sz范围内，并且为了通过usertests中的stacktest，还要阻止给因访问guardpage触发的pagefault分配内存。这一点通过检测va和sp的关系完成，sp以下为用户栈剩余空间，guardpage和data&context，显然仅有guardpage有可能触发pagefault，所以对于低于sp的pagefault一概杀死进程。

另外在做这一部分的时候，我发现了一个mappage非常误导人的特性，mappage的注释说明了传入的va不需要对齐。但这样的后果是，如果传入一个不对齐的va，那么该va所在的页和紧邻的下一个页都会被分配，而不是只分配va所在的页！而已经申请好的物理内存又仅仅只有一页，这显然会出错。

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

    uint64 va=r\_stval();

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

      char\* mem=kalloc();

      if(mem==0){

        p->killed=1;

      }else{

        memset(mem,0,PGSIZE);

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

          kfree(mem);

          p->killed=1;

        }

      }

    }

    else p->killed=1;

  }

做好这两部分的时候，会发现uvmunmap在每个进程输出结果后报错。显然这是在进程内存回收的时候发生的。对于那些p->sz已覆盖但还未分配的页，它们会触发pannic：uvmunmap：not mapped或panic：uvmunmap: walk。把这两个panic注解掉并且加上continue即可。

做到这里已经可以运行一些不那么“刁难”的测试了。第一个出问题的测试便在于fork，fork对于子进程内存的分配也是基于sbrk的，也会触发懒分配。而在子进程返回用户空间之前，父进程还需要使用uvmcopy将自己内存中的内容复制给子进程，这会触发panic：uvmcopy：pte should exist或page not present。处理方案和uvmunmap相同。

下一个是对于懒分配下，write/read所访问未分配的虚拟地址触发的错误。这个错误的难点在于，它和fork一样，是在内核中发生的，所以并不会走用户中断中pagefault的处理；并且它不会抛出任何的panic，只会单纯地返回-1。这迫使我得自己去追踪write调用……最后发现是在copyin/copyout 在解引用用户空间传入的va时，walkaddr找不到对应的地址。

于是自定义walkaddr，其实处理流程和用户pagefault几乎一致。

uint64

walkaddr(pagetable\_t pagetable, uint64 va)

{

  pte\_t \*pte;

  uint64 pa;

  if(va >= MAXVA)

    return 0;

  pte = walk(pagetable, va, 0);

  if(pte==0||(\*pte&PTE\_V)==0){

    struct proc\* p=myproc();

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

      char\* mem=kalloc();

      if(mem==0){

        return 0;

      }else{

        memset(mem,0,PGSIZE);

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

          kfree(mem);

          return 0;

        }

      }

    }else return 0;

  }

……

}

到这里所有测试就均能通过了。我奇怪的点在于我是如何通过sbrkmuch测试的？我并没有对p->sz的增长做出任何的限制，它完全是可能超过MAXVA的……

### 总结

我隐约感觉懒分配是危险的，它注解掉了大量的系统原有的panic。这在大规模编程时很容易留下漏洞。

实验内容简简单单，洒洒水