# 2024/04/02~2024/04/03

连着两天磕了十来小时给做完了，体验下来就是引入一点文件操作的cow lab。大量地和页表打交道。

与上一次cow和lazy的实验经历不同的是，这一次有了vscode debug的加成，debug快多了。

### mmap

提出mmap的背景在于：在内存中缓存文件块的bcache仅存在于内核地址空间，如果用户想要使用就必须通过内核与用户之间的写入写出来和bcache交互，每一次的读/写操作都将经过磁盘->内核,内核->用户两次拷贝。

Mmap实现了这样的功能：将一些可能常用文件内容拷贝到一块物理地址上，并且建立这段物理地址和用户虚拟地址的映射（写入进程页表中）。这样用户对特定内容的访问就只需要通过一次拷贝，即磁盘->某一段物理地址。

mmap函数原型如下:

void \*mmap(void \*addr, size\_t length, int prot, int flags,int fd, off\_t offset);

其中prot为保护位，用于约定这块文件内容映射之后的读写，执行等权限。

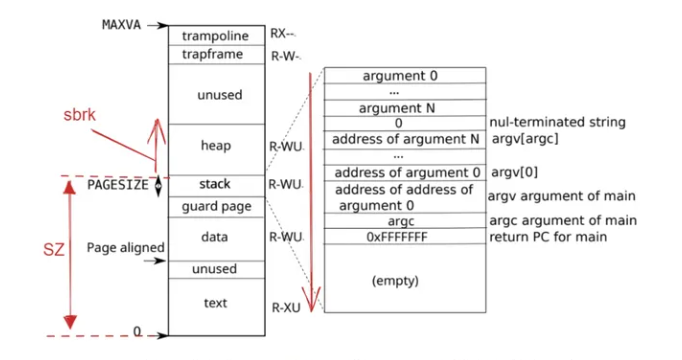
flags定义了map方式，在本实验中，有private和shared两种map方式。private方式下，进程可以无视文件在被open调用时所定义的读写权限，因为进程所做的任何修改都不会写回到磁盘中去。而在share方式下，进程是否能对文件进行读写操作决定于文件打开时所定义的权限。

fd传入了被进程打开的文件的文件描述符。

允许传入未对齐的length，会在后续操作中对齐。

在xv6的实现方式下，addr在调用过程中总是为0，即由内核来决定应该把这一部分文件映射到进程中的哪一段地址中去。offset在本lab中也约定总是为0。

为了决定将文件映射到哪一段虚拟地址上，首先得寻找进程地址中可用的部分，考察进程地址空间：



可以注意到，在heap之上存在着unused段。而heap是向上生长的，由p->sz指向heap顶端地址。所以映射文件的空间(VAM)可以从TRAMFRAME地址开始，向下生长直到触碰到p->sz。

如下的代码将用于自上向下查找可用的地址段。

对于找到的任何空闲页（地址为start），向下继续寻找空闲页直到地址i，使得start-i+PGSIZE的值等于PGROUNDUP(length)。+1页的原因在于start地址的页内所有地址都不在[i，start]区间内。ismapped用于判断该段地址是否已经用于mmap但还处于懒分配状态,即判断该地址是否属于任何一个vma(在下文介绍vma)。

int

ismapped(struct proc\* p,uint64 va){

  for(int i=0;i<16;i++){

    if(p->vmas[i].used==1&&(uint64)va>=p->vmas[i].addr&&(uint64)va<p->vmas[i].addr+PGROUNDUP(p->vmas[i].length)){

      return 1;

    }

  }

  return 0;

}

uint64

finduregin(pagetable\_t pagetable,int length,int psz){

  pte\_t \*pte;

  struct proc\* p=myproc();

  uint64 start=0;

  int flag=0;

  for(uint64 i=TRAPFRAME;i>psz;i-=PGSIZE){

    if(ismapped(p,i))continue;

    pte=walk(pagetable,i,0);

    if(pte==0&&flag==0){

      if(PGROUNDUP(length)==PGSIZE){

        flag=2;

        start=i;

        break;

      }

      flag=1;

      start=i;

    }

    else if(pte==0&&flag==1){

      if(start-i+PGSIZE==PGROUNDUP(length)){

        start=i;

        flag=2;

        break;

      }

    }

    else if(pte!=0&&flag==1){

      flag=0;

      start=0;

    }

  }

  if(flag!=2){

    return 0;

  }

  return start;

}

为了维护进程的mmap信息，定义VMA结构体来保存每一次mmap的信息。

struct vma{

  int used;//该vma是否已经被使用

  uint64 addr;//起始地址

  uint length;//长度，向上对齐

  int flags;//SHARED或PRIVATE

  int prot;//权限位

  struct file\* f;//文件指针

};

当然，一个进程通常会mmap多个文件，所以静态地在proc结构体中添加vmas数组

struct proc {

    ...

  struct vma vmas[NVMA];//NVMA=16,实验的Hint建议的规模。

    ...

};

mmap系统调用是以懒分配的方式分配地址，即在vma中作对应记录但并不建立映射，也不读取文件内容到物理地址中。这样将允许进程mmap很大的文件。

对于读写规则，即prot位的确定如下:按照测试用例总结如下:对于RW模式打开的文件，读写都被允许。对于R模式打开的文件，SHARED MAP模式下的写操作将被禁止,PRIVATE MAP模式下的读写操作都被允许。

mmap系统调用的主要工作就是找到一块未被使用的vma记录，利用传入的参数将其初始化，不做任何操作。

void\* sys\_mmap(void){

  struct proc \*p = myproc();

  uint64 addr;

  int i;

  int length;

  int prot;

  int flags;

  int fd;

  int offset;

  if(argint(1,&length)<0||argint(2,&prot)<0||argint(3,&flags)<0||argint(4,&fd)<0||argint(5,&offset)<0){

    return (void\*)-1;

  }

  if((addr=finduregin(p->pagetable,length,p->sz))==0){

    return (void\*)-1;

  }

    if(p->ofile[fd]->writable==0&&p->ofile[fd]->readable==1&&(prot&PROT\_WRITE)&&(flags&MAP\_SHARED)){

    return (void\*)-1;

  }

  for(i=0;i<16;i++){

    if(p->vmas[i].used==0){

      p->vmas[i].used=1;

      break;

    }

  }

  if(i==16){

    return (void\*)-1;

  }

  p->vmas[i].addr=addr;

  p->vmas[i].length=PGROUNDUP(length);

  p->vmas[i].flags=flags;

  p->vmas[i].prot=prot;

  p->vmas[i].f=p->ofile[fd];

  p->vmas[i].f->ref+=1;

  return (void\*)addr;

}

而真正的mmap将在进程访问到mmap的虚拟地址时发生，即在缺页中断中处理。

进入到usertrap:

根据r\_stval中保存的va来确定是对哪一块VMA的访问造成了缺页中断，并为文件分配内存，在该块内存的物理地址写入文件内容，建立用户虚拟地址和该物理地址的映射，建立映射时需要额外添加PTE\_U位。

当然，va可能对应的是文件的中间部分，(va-p->vmas[i].addr)/PGSIZE\*PGSIZE即为对应的文件偏移量

  else if(r\_scause()==13){

    uint64 va=r\_stval();

    int i;

    for(i=0;i<16;i++){

      if(va>=p->vmas[i].addr&&va<p->vmas[i].addr+PGROUNDUP(p->vmas[i].length)){

        break;

      }

    }

    if(i!=16){

      struct file\* f=p->vmas[i].f;

      struct inode\* ip=f->ip;

      char\* mem=kalloc();

      if(mem==0){

        p->killed=1;

      }else{

        memset(mem,0,PGSIZE);

        if(mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)mem, p->vmas[i].prot<<1|PTE\_U)!=0){

          kfree(mem);

          p->killed=1;

        }

        ilock(ip);

        readi(ip,1,PGROUNDDOWN(va),(va-p->vmas[i].addr)/PGSIZE\*PGSIZE,PGSIZE);

        iunlock(ip);

      }

    }

    else p->killed=1;

  }

有mmap，那肯定就有munmap:

munmap做这样的工作:寻找传入的va对应的vma，如果是以SHARED方式进行的mmap，调用filewrite将内存中的文件修改写回到磁盘中，并且解绑该页，对应地修改vma中的addr和length字段。

当length为0时，将文件的引用计数-1。

当然，还需要处理懒分配后一直没有建立映射的地址（本实验中默认这部分地址连续且位于mmap地址段的末尾），当传入一个mmap懒分配但并未建立映射的地址时，持续地前推vma的addr和减少length，直到length为0。

int sys\_munmap(void){

  struct proc\* p=myproc();

  int i;

  uint64 addr;

  int length;

  if(argaddr(0,&addr)<0||argint(1,&length)<0){

    return -1;

  }

  for(i=0;i<16;i++){

    if(p->vmas[i].used==1&&(uint64)addr>=p->vmas[i].addr&&(uint64)addr<p->vmas[i].addr+PGROUNDUP(p->vmas[i].length)){

      break;

    }

  }

  if(i==16){

    return -1;

  }

  while(walkaddr(p->pagetable,(uint64)addr)==0){

      p->vmas[i].addr+=PGSIZE;

      p->vmas[i].length-=PGSIZE;

      if(p->vmas[i].length==0){

        p->vmas[i].f->ref-=1;

        p->vmas[i].used=0;

        return 0;

      }

  }

  if(p->vmas[i].flags&MAP\_SHARED){

    for(uint64 va=PGROUNDDOWN((uint64)addr);va<PGROUNDDOWN((uint64)addr)+length;va+=PGSIZE){

      filewrite(p->vmas[i].f,va,PGSIZE);

    }

  }

  uvmunmap(p->pagetable,PGROUNDDOWN((uint64)addr),PGROUNDUP(length)/PGSIZE,0);

  p->vmas[i].addr+=PGROUNDUP(length);

  p->vmas[i].length-=PGROUNDUP(length);

  if(p->vmas[i].length==0){

    p->vmas[i].f->ref-=1;

    p->vmas[i].used=0;

  }

  return 0;

}

根据提示，还需要修改exit和fork。

修改exit，保证进程退出时释放所有mmap占用的物理内存。

这里采用的方案是从这段地址的末尾开始向前释放，即先释放懒分配的页，为了避免调用uvmunmap的时候不缺定在length范围内究竟多少页被真正分配了。

  for(int i=0;i<NVMA;i++){

    if(p->vmas[i].used==1){

      uint64 uaddr=PGROUNDDOWN(p->vmas[i].addr)+p->vmas[i].length-PGSIZE;

      while(walkaddr(p->pagetable,uaddr)==0){

        uaddr-=PGSIZE;

        p->vmas[i].length-=PGSIZE;

        if(p->vmas[i].length==0){

          p->vmas[i].f->ref-=1;

          p->vmas[i].used=0;

          break;

        }

      }

      uvmunmap(p->pagetable,PGROUNDDOWN(p->vmas[i].addr),p->vmas[i].length/PGSIZE,0);

    }

    p->vmas[i].used=0;

  }

修改fork，以保证子进程和父进程有同样的mmap。因为懒分配的页信息都保存在vma中，在拷贝vma后，对于walkaddr无法解析的va（即未实际分配的页）直接跳过即可。

  for(int k=0;k<NVMA;k++){

    if(p->vmas[k].used==1){

      uint64 pa;

      p->vmas[k].f->ref+=1;

      memmove(&np->vmas[k],&p->vmas[k],sizeof(struct vma));

      p->vmas[k].f->ref+=1;

     for(uint64 j =p->vmas[k].addr; j<p->vmas[k].addr+p->vmas[k].length; j += PGSIZE){

        if((pa=walkaddr(p->pagetable,p->vmas[k].addr))==0)

          continue;

        if(mappages(np->pagetable, j, PGSIZE, (uint64)pa, p->vmas[k].prot<<1|PTE\_U) != 0){

          return -1;

        }

      }

    }

  }

至此全部内容完成。

### 总结

在vscode加持和之前cowlab的经验下，mmap实验进行的是相当的顺利。我在完成实验后查阅网上的代码，发现和自己的结构大相径庭，虽然都能通过make grade就是了。尤其是对于可用地址段的寻找部分，网上大部分给出的方案是维护一个VMA\_BOTTOM变量，记录VMA生长的最下端地址，每次VMA就从这个位置添加，每次释放VMA都检查是否释放的是最低端的VMA,如果是则更新VMA\_BOTTOM。显然这种实现方式是有缺陷的。我采用的则是从TRAMFRAME开始向下搜索空余空间的方案，个人认为更优。