**实验十 mmap**

**10.1 实验目的**

1. 了解使用虚拟内存的原因

2. 理解创建和释放文件映射的方式

**10.2 实验内容**

10.2.1 mmap（hard）

您应该实现足够的mmap和munmap功能，以使mmaptest测试程序正常工作。如果mmaptest不使用mmap特性，则不需要实现该特性。

10.2.2 submit

提交实验。

**10.3 实验过程**

10.3.1 mmap

（1）mmap实现了内核映射文件并返回文件的虚拟地址，与此相对，munmap将删除指定地址范围内的mmap映射。首先定义虚拟内存区域结构体VMA，即一段连续的虚拟地址，与页表共同构成了虚拟空间。

#define MAXVMA 16

struct VMA

{

   int used;   // if this vma is used or

   uint64 addr;//address

   uint64 len; //length

   int prot;   //permissions

   int flags;  // flags

   struct file \*f; //the file being mapped

   uint64 start\_point;//starting piont in the file at which to map

};

// Per-process state

struct proc {

  ···

  struct VMA vma[MAXVMA];      // keep track of what mmap has mapped for proc

};

（2）在sysfile.c中添加系统调用mmap，首先读取参数然后在vma中虚招空位并填充相应字段。

uint64

sys\_mmap(void)

{

   int len;

   int prot,flags,fd;

   struct file \*f;

   if(argint(1, &len) < 0 ||  argint(2, &prot) < 0 || argint(3, &flags) < 0  || argfd(4, &fd,&f) < 0  )

    return -1;

   // if file is read-only,but map it as writable.return fail

   if(!f->writable && (prot & PROT\_WRITE) && (flags & MAP\_SHARED ) )

   {

      return -1;

   }

   struct proc\* p=myproc();

   for(uint i=0;i<MAXVMA;i++)

   {

      struct VMA \*v=&p->vma[i]  ;

      if(!v->used) //find an unsed vma

      {

         // store relative auguments

   v->used=1;

         v->addr=p->sz;//use p->sz to p->sz+len to map the file

   len= PGROUNDUP(len);

   p->sz+=len;

   v->len=len;

   v->prot=prot;

   v->flags=flags;

   v->f= filedup(f);//increase the file's ref cnt

   v->start\_point=0;//staring point in f to map is 0

   return v->addr;

      }

   }

   return -1;

}

（3）由于使用了懒加载方式，所以在访问未加载界面时会产生缺页中断。在usertrap中对缺页中断进行处理。

void

usertrap(void)

{

  int which\_dev = 0;

  if((r\_sstatus() & SSTATUS\_SPP) != 0)

    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),

  // since we're now in the kernel.

  w\_stvec((uint64)kernelvec);

  struct proc \*p = myproc();

  // save user program counter.

  p->trapframe->epc = r\_sepc();

  if(r\_scause() == 8){

    // system call

    if(p->killed)

      exit(-1);

    // sepc points to the ecall instruction,

    // but we want to return to the next instruction.

    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,

    // so don't enable until done with those registers.

    intr\_on();

    syscall();

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

    // ok

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

  {

      // check whether lazy allocation is needed

      uint64 va=r\_stval();

      if(va>=p->sz) goto a;

      if(va<p->trapframe->sp) goto a;

      uint lazy=0;

      for(uint i=0;i<MAXVMA;i++)

      {

   struct VMA \*v=&p->vma[i];

         //注意区间是左闭右开的

         if(v->used && va>=v->addr && va<v->addr+v->len)//find corresponding vma

   {

            // lazy allocation

      char \* mem;

      mem=kalloc();

      memset(mem,0,PGSIZE);

      if(mem==0) goto a;

            va=PGROUNDDOWN(va);

      uint64 off=v->start\_point+va-v->addr;// starting point + extra offset

      // PROT\_READ=1 PROT\_WRITE=2 PROT\_EXEC=4

      // PTE\_R=2     PTE\_W=4      PTE\_X=8

      // 所以需要将vma[i]->prot 左移一位

      if(mappages(p->pagetable,va,PGSIZE,(uint64)mem,(v->prot<<1) |PTE\_U  )!=0)

      {

         kfree(mem);

         goto a;

      }

            // read 4096 bytes of relevant file into allocated page

      ilock(v->f->ip);

      readi(v->f->ip,1,va,off,PGSIZE);

      iunlock(v->f->ip);

      lazy=1;

      break;

   }

      }

      if(!lazy)  goto a;// no lazy allocation is needed

  }

  else {

   a:

    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;

  }

  if(p->killed)

    exit(-1);

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

  if(which\_dev == 2)

    yield();

  usertrapret();

}

（4）munmap即释放mmap空间，当mmap是MAP\_SHARED时需要进行写回，除此之外不用写回。同时需要在数组中找到对应的vma结构体，且需要设定偏移量，所以需要重写一个支持自定义文件offset的函数。

int file\_write\_new(struct file \*f, uint64 addr,int n ,uint off)//自定义一个写文件函数 和filewrite主要的区别是可以设定off，也就是从哪里开始写

{

   int r=0;

   if(f->writable==0) return -1;

   int max= ((MAXOPBLOCKS-1-1-2) / 2)\* BSIZE;

   int i=0;

   while(i<n)

   {

      int n1=n-i;

      if(n1>max) n1=max;

      begin\_op();

      ilock(f->ip);

      if((r=writei(f->ip , 1 , addr +i,off,n1)) >0 )

          off+=r;

      iunlock(f->ip);

      end\_op();

      if(r!=n1)  break;

      i+=r;

   }

   return 0;

}

uint64

sys\_munmap(void)

{

  uint64 addr;

  int len;

  int close=0;

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

    return -1;

   struct proc\* p=myproc();

   for(uint i=0;i<MAXVMA;i++)

   {

      struct VMA \*v=&p->vma[i];

      //only unmap at start,end or the whole region

      if(v->used && addr>=v->addr && addr <=v->addr+v->len)

      {

   uint64 npages=0;

         uint off=v->start\_point;

         if(addr==v->addr) // unmap at start

   {

       if(len >= v->len) //unmap whole region

       {

     len=v->len;

           v->used=0;

     close=1;

       }

       else//unmap from start but not whole region

       {

    v->addr+=len;

    v->start\_point=len;//update start point at which to map

       }

   }

   len=PGROUNDUP(len);

      npages=len/PGSIZE;

   v->len-=len;

   p->sz-=len;

         if(v->flags & MAP\_SHARED) // need to write back pages

   {

      file\_write\_new(v->f, addr , len , off );

   }

         uvmunmap(p->pagetable,PGROUNDDOWN(addr),npages,0);

         // decrease ref cnt of v->f

   if(close) fileclose(v->f);//注意先写文件再fileclose 否则 f->ref=0 就没办法写了

   return 0;

      }

   }

  return -1;

}

（5）修改fork函数使子进程复制父进程的文件映射，修改exit函数实现当退出进程时写回并释放相应的文件映射。

int

fork(void)

{

  int i, pid;

  struct proc \*np;

  struct proc \*p = myproc();

  // Allocate process.

  if((np = allocproc()) == 0){

    return -1;

  }

  for(int i=0;i<MAXVMA ; i++)

  {

     struct VMA \*v=&p->vma[i];

     struct VMA \*nv=&np->vma[i];

      //only unmap at start,end or the whole region

      if(v->used)

      {

         memmove(nv,v,sizeof(struct VMA));

   filedup(nv->f);

      }

  }

 void

exit(int status)

{

  struct proc \*p = myproc();

  if(p == initproc)

    panic("init exiting");

  // Close all open files.

  for(int fd = 0; fd < NOFILE; fd++){

    if(p->ofile[fd]){

      struct file \*f = p->ofile[fd];

      fileclose(f);

      p->ofile[fd] = 0;

    }

  }

  for(int i=0;i<MAXVMA ; i++)

  {

     struct VMA \*v=&p->vma[i];

      //only unmap at start,end or the whole region

      if(v->used)

      {

         uvmunmap(p->pagetable,v->addr,v->len / PGSIZE,0);

         memset(v,0,sizeof(struct VMA));

      }

  }

  begin\_op();

  iput(p->cwd);

  end\_op();

  p->cwd = 0;

  // we might re-parent a child to init. we can't be precise about

  // waking up init, since we can't acquire its lock once we've

  // acquired any other proc lock. so wake up init whether that's

  // necessary or not. init may miss this wakeup, but that seems

  // harmless.

  acquire(&initproc->lock);

  wakeup1(initproc);

  release(&initproc->lock);

  // grab a copy of p->parent, to ensure that we unlock the same

  // parent we locked. in case our parent gives us away to init while

  // we're waiting for the parent lock. we may then race with an

  // exiting parent, but the result will be a harmless spurious wakeup

  // to a dead or wrong process; proc structs are never re-allocated

  // as anything else.

  acquire(&p->lock);

  struct proc \*original\_parent = p->parent;

  release(&p->lock);

  // we need the parent's lock in order to wake it up from wait().

  // the parent-then-child rule says we have to lock it first.

  acquire(&original\_parent->lock);

  acquire(&p->lock);

  // Give any children to init.

  reparent(p);

  // Parent might be sleeping in wait().

  wakeup1(original\_parent);

  p->xstate = status;

  p->state = ZOMBIE;

  release(&original\_parent->lock);

  // Jump into the scheduler, never to return.

  sched();

  panic("zombie exit");

}

（6）因为懒分配的存在，所以p->sz范围之内的虚拟地址可能没有全部映射到p->pagetable中，所以修改uvmunmap和uvmcopy函数。

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");

    if((\*pte & PTE\_V) == 0) return;

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

    if(PTE\_FLAGS(\*pte) == PTE\_V)

      panic("uvmunmap: not a leaf");

    if(do\_free){

      uint64 pa = PTE2PA(\*pte);

      kfree((void\*)pa);

    }

    \*pte = 0;

  }

}

int

uvmcopy(pagetable\_t old, pagetable\_t new, uint64 sz)

{

  pte\_t \*pte;

  uint64 pa, i;

  uint flags;

  char \*mem;

  for(i = 0; i < sz; i += PGSIZE){

    if((pte = walk(old, i, 0)) == 0)

      panic("uvmcopy: pte should exist");

    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;

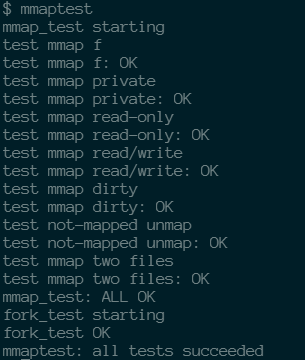
 err:

  uvmunmap(new, 0, i / PGSIZE, 1);

  return -1;

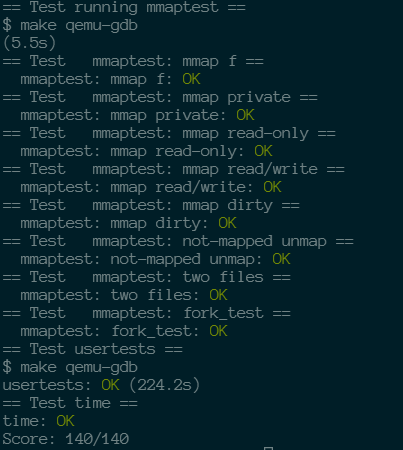
}

（7）运行mmap



**10.4 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。



**10.5 实验小结**

本次实验覆盖的内容较多，包含了虚拟内存、文件地址、懒加载等等，这不仅是单纯的对于某个知识点的运用，而是将其综合起来进行考察。这让我体会到写一个完整的操作系统需要兼顾多个方面，不仅需要完成各个部分的编写，也要看与其相关的其他函数与之相应的修改。