操作系统课设实验报告

- 学院：泰山学堂
- 专业：计算机
- 年级：2021级
- 姓名：袁胜利
- 学号：202100130139

我们完成了 MIT 6.828: Operating System Engineering 课程实验 (xv6)，并已经验收完毕。这里对于实验给出初验报告。

需要注意的是，我在做实验的时候备注了很多注释，这些注释对实验和代码进行了很多解释，已经注释的部分在代码下方的具体讲解中就不在赘述。

Lab Utilities

sleep

```
//sleep.c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
//暂停指定的嘀嗒数（与OS有关）
int main(int argc, char **argv) {
    if(argc < 2) {
        printf("usage: sleep <ticks>\n");//错误信息
        exit(1);
    }
    sleep(atoi(argv[1]));//变整数
    exit(0);
}
```

借用 `sleep` 系统调用接口实现用户端的 `sleep` 程序，考察对于主函数参数 `argc` 及 `argv` 的使用，其中 `argc` 代表参数个数、 `argv` 代表参数的字符串表示，值得注意的是程序名也包括在参数之中。

pingpang

```c
// pingpong.c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char **argv) {
    // 0读 1写
    int ptc[2], ctp[2];
    pipe(ptc); // 父→ 子
    pipe(ctp); // 子→ 父

    int flag=fork();//创建子'进'程
    if(flag<0) //失败
    {
        printf("fail\n");
        exit(1);
    }

    if(flag != 0)
    { // 父进程
        write(ptc[1], "Y", 1); //写字符串Y的前1位
        char buf;
        read(ctp[0], &buf, 1); //buf=ctp[0] 读
        printf("%d: received pong\n", getpid()); // 成功
        wait(0);
    } else { //子进程
        char buf;
        read(ptc[0], &buf, 1); //读
        printf("%d: received ping\n", getpid());//打印进程id
        write(ctp[1], &buf, 1); // 写
    }
    exit(0);
}
```

本函数的功能是，将一个字节的数据在父子进程之间传输，当父进程或子进程收到该字节时，打印其PID和一串消息。考察了对于进程通信方式（IPC）——管道的理解及使用。事实上 `pipe` 函数的作用仅为构造一对双工相连的文件描述符，在调用 `fork` 后，这一对文件描述符在父子进程中被共享，因此即可通过这对文件描述符实现父子进程的进程间通信。

具体实现方式比较简单，在 `pipe` 和 `fork` 被调用后，在父子进程中分别关闭不同的文件描述符后，父子进程即可通过这对文件描述符进行通信，父子进程分别调用 `read` 和 `write` 传递信息即可。

具体顺序为：父进程先向子进程传输数据，子进程收到后给父进程应答，最后父进程收到应答后，结束进程。

prime

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

// 一次 dfs 筛一个素数
void dfs(int L[2]) {
    int p;
    read(L[0], &p, sizeof(p)); // 素
    if(p == -1) exit(0); // flag
    printf("prime %d\n", p);
    int R[2]; pipe(R);//新管道
    if(fork()==0)
    {// 子进程
        close(R[1]); // 操作系统对每个进程有文件描述符数量的限制，如果不关闭不需要的文件描述符
        close(L[0]); // ，可能会导致文件描述符用尽的问题，进而导致程序异常。
        dfs(R);
        exit(0);
    } else {// 父进程
        close(R[0]);
        int buf;
        while(read(L[0], &buf, sizeof(buf)) && buf != -1)
            if(buf % p != 0) //重写
                write(R[1], &buf, sizeof(buf));
        buf = -1;
        write(R[1], &buf, sizeof(buf)); //-1
        wait(0); // 等待任何子进程终止,不能等待子进程的子进程
        exit(0);
    }
}

int main(int argc, char **argv) {
    int que[2];
    pipe(que);

    if(fork()==0)
    {//子进程
        close(que[1]);
        dfs(que);
        exit(0);//终止一个进程
    } else {
        close(que[0]); // 同上
        int i;
```

```
        for(i=2;i<=35;i++) write(que[1], &i, sizeof(i));//先进先出
        i=-1;
        write(que[1], &i, sizeof(i)); // 末尾输入 -1，用于标识输入完成
        wait(0);//因为if有exit，所以放外面一样
        exit(0);
    }
}
```

注意stage之间的管道 pleft 和 pright，要关闭不需要用到的文件描述符，因为xv6中的资源非常有限，不这么做会报错。

具体解释一下代码：首先将 2—35 的整数放入管道中，每次递归取出的最左边的数p一定是质数，然后再管道中去除掉p的所有倍数，最后将新管道交给子进程处理即可。递归的结束条件为取完所有数，此时所有质数已经找到，执行 `exit(0)` 。

find

```
//find.c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/fs.h"

void find(char *path, char *target) {
    char buf[512], *p;//完整路径长度，经验选择512
    int fd;//用于标识打开文件或其他 I/O 设备的句柄,以便后续的文件操作可以使用它来引用打开的文件或设备。
    struct dirent de;//表示目录中的条目（文件和子目录）
    struct stat st;
    if((fd = open(path, 0)) < 0)
    {//打不开
        printf("find: cannot open %s\n", path);
        return;
    }
    if(fstat(fd, &st) < 0)//权限检查、文件大小计算、时间戳的管理等
    {//无法获取状态信息
        printf("find: cannot stat %s\n", path);
        close(fd);
        return;
    }

    switch(st.type){//类型-文件/文件夹
    case T_FILE:// 文件名后缀匹配
        if(strcmp(path+strlen(path)-strlen(target), target) == 0)
```

```
            printf("%s\n", path);
        break;
    case T_DIR://文件夹
        if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){//超过512，寄
            printf("find: path too long\n");
            break;
        }
        strcpy(buf, path);//buf=path
        p = buf+strlen(buf);//指向buf后
        *p++ = '/';//放入/
        while(read(fd, &de, sizeof(de)) == sizeof(de))//sizeof(de)=2+14
        {//从目录文件fd中读取一个目录项，并存在de中。每次调用read会读取下一个目录项，确保了每次迭代都处理一层路径。
            if(de.inum == 0) continue;//该目录项无效
            memmove(p, de.name, DIRSIZ);//memcpy 14
            p[DIRSIZ] = 0;//确保字符串以 null 结尾
            // Don't recurse into "." and ".." ! ! ! ! !
            if(strcmp(buf+strlen(buf)-2, "/.") != 0 && strcmp(buf+strlen(buf)-3, "/..") != 0)
                find(buf, target); // 递归
        }
        break;
    }
    close(fd);
}

int main(int argc, char *argv[])
{
    if(argc < 3) exit(0);
    char target[512];//要查找的目标文件
    target[0] = '/'; // 后缀匹配，防止前面还有//默认根目录下运行
    strcpy(target+1, argv[2]);//要查找的目标文件的名称
    find(argv[1], target);//argv[1]要搜索的起始目录的路径
    exit(0);
}
```

该程序的功能是，通过输入**查询根目录**和**目标文件名**，寻找并打印查询根目录下所有与目标文件名的名称相同的文件。

利用深度优先方式，递归访问当前路径名下的所有文件，包括普通文件和目录文件。如当前路径名所对应的文件为普通文件时，如该文件名与目标文件名相同，则将当前路径打印；如当前路径名所对于的文件为目录文件时，对该目录下除 `.` 和 `..` 的所有文件递归的运行 `find` 函数（如果不 `.` 和 `..` 会造成死循环）。具体的递归方式为，将文件名压入用于储存当前路径名 `path` 的栈中，当以该文件作为根目录的 `find` 函数运行完后，将该文件名从栈中弹出。

更详细的讲解已在代码中的注释处给出。

xargs

该程序的功能为：将输入该程序的 `argv` 数组的除 `argv[0]` 的其余字符串数组，作为新的命令行参数数组。每当程序从标准输入接受到一行字符串时，将该行字符串作为这个数组的最后一个参数，并生成子进程运行该命令。

```
//xargs.c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/fs.h"

// 带参数列表，执行某个程序
void run(char *program, char **args) {
    if(fork() == 0)
    { // 必须在子进程中调用
        exec(program, args);//路径-参数
        exit(0);
    }//一旦调用了 exec，当前进程的代码和数据都会被新程序替代，原有的程序逻辑将失效。
    return; // parent return
}

int main(int argc, char *argv[]){
    char buf[2048]; // 读入时使用的内存池,足够大的缓冲区
    char *p = buf, *last_p = buf; // 当前参数的结束、开始指针
    char *argsbuf[128]; // 全部参数列表，字符串指针数组，包含 argv 传进来的参数和
stdin 读入的参数
    char **args = argsbuf; // 指向指针的指针
    for(int i=1;i<argc;i++)
    {//程序启动时传递的参数。
        *args = argv[i];
        args++;
    }
    char **pa = args; // 后面开始逐行读命令
    while(read(0, p, 1) != 0) {//标准输入stdin，循环到EOF
        if(*p == ' ' || *p == '\n')
        {//  `echo zxf ptx`，则 zxf 和 ptx 各为一个参数
            *p = '\0';     //分割
            *(pa++) = last_p;//表这儿文法出现的字符串指针
            last_p = p+1;
            if(*p == '\n')
            {// 读入一行完成
                *pa = 0; // 参数列表末尾用 null 标识列表结束
```

```
                run(argv[1], argsbuf); // 执行最后一行
                pa = args; // 重置读入参数指针，准备读入下一行
            }
        }
        p++;
    }
    if(pa != args)
    { // 最后一行非空
        *p = '\0';
        *(pa++) = last_p;// 收尾最后一行
        *pa = 0; // 参数列表末尾用 null 标识列表结束
        run(argv[1], argsbuf);// 执行最后一行指令
    }
    while(wait(0) != -1) {}; // 循环等待所有子进程完成，每一次 wait(0) 等待一个
    //如果调用进程没有子进程，wait(0) 会失败，并返回 -1
    exit(0);
}
```

具体实现比较简单，首先从输入参数中提取 `args` 。并以 `argsbuf` 作为存储标准输入的缓存区，当读入非 `\n` 字符时，将其加入 `argsbuf` ；当读入 `\n` 时，将当前缓冲区作为 `args` 的最后一个元素，并执行 `args` 所对应的命令，并清空缓冲区。

注意在 `run` 函数当中必须创建子进程，因为调用了 `exec` 后，当前进程的代码和数据都会被新程序替代，为了保护当前环境，只能派子进程去 `exec` 。

具体的写法使用了很多C语言字符串的特性，我在具体的代码中写了详细的注释。

其他工作

Optional challenges

我在这里实现了 `uptime` 。

```
// uptime.c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main() {
    printf("%d\n", uptime());
    exit(0);
}
```

输出当前打印时间即可。

Makefile

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_sleep\
    $U/_pingpong\
    $U/_primes\
    $U/_find\
    $U/_xargs\
    $U/_uptime\
```

这样使得我们的程序能够正确通过编译。以后的很多实验同样需要修改 `Makefile` 文件，这里不再重复。

测试结果

```
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (4.8s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (1.1s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.1s)
== Test pingpong ==
```

```
$ make qemu-gdb
pingpong: OK (1.2s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.2s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.3s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.6s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.7s)
== Test time ==
time: OK
Score: 100/100
```

Lab: system calls

系统调用

本实验要求实现两个不同的系统调用，由于都是系统调用，所以我们先讲它们的相同点，之后再分别讲解不同点。

系统调用的具体流程如下

```
user/user.h:        用户态程序调用跳板函数
user/usys.S:        跳板函数陷入到内核态
kernel/syscall.c    到达内核态统一系统调用处理函数 syscall()，所有系统调用都会跳到这里来处理。
kernel/? .c           执行具体内核操作
```

这样说可能会比较抽象，下面我们依次解释说明

Makefile

```
UPROGS=\
    ……
    $U/_trace\
    $U/_sysinfotest\
```

和上一个实验一样，先吧文件添加到路径

user/user.h

```
int trace(int);
struct sysinfo;
int sysinfo(struct sysinfo *);
```

在用户态声明函数

usys.pl

```
entry("trace");  # 实验1
entry("sysinfo");#实验2
```

这个脚本在运行后会生成 $usys.S$ 汇编文件，里面定义了每个 $systemcall$ 的用户态跳板函数

syscall.h

```
#define SYS_trace  22 // 实验1
#define SYS_sysinfo 23//实验2
```

内核态系统调用的命名规范，用字符串表示一个下标

syscall.c

```
extern uint64 sys_trace(void);    // 实验1
extern uint64 sys_sysinfo(void);  //实验2

static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
```

```
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_trace]   sys_trace,     //实验1
[SYS_sysinfo] sys_sysinfo,//实验2
};
```

前面是声明函数，后面这个是一个指针（函数）数组，存放不同的系统调用函数。

这些工作都做完后，剩下的就是在内核中合适的位置实现我们的内核调用，理论上放的位置无所谓，但是要保证代码的可维护性。

这么繁琐的调用流程的主要目的是实现用户态和内核态的良好隔离。

System call tracing

这个实验需要增加一个叫做trace的系统调用，它会让进程在调用被打上mask的系统调用时，输出对应的进程号、系统调用名和返回值。以下为具体过程

proc.h

```
struct proc {
    ..........
  uint64 syscall_trace;         // 我们新加的一行Mask for syscall tracing（新添加的
用于标识追踪哪些 system call 的 mask）
};
```

我们先在“进程”这个结构体中增加掩码，表示要追踪的进程

proc.c

```
static struct proc*
allocproc(void)
{
  ..........
  p->syscall_trace = 0; // 实验1 为 syscall_trace 设置一个 0 的默认值
  return p;
}

int
fork(void)
{
  ......

  np->syscall_trace = p->syscall_trace; // HERE!!! 子进程继承父进程的
syscall_trace

   ......
}
```

这里有两个修改，第一个修改是让第一个进程的掩码初始化为0，防止乱码报错

第二个修改是在新建进程时，让子进程继承父进程的掩码，这是因为题目中有要求

`The trace system call should enable tracing for the process that calls it and any children that it subsequently forks`

即`trace()`系统调用应启用跟踪对于调用它的进程以及它随后产生的任何子进程

syscall.c

```
const char *syscall_names[] = {
[SYS_fork]    "fork",
[SYS_exit]    "exit",
[SYS_wait]    "wait",
[SYS_pipe]    "pipe",
[SYS_read]    "read",
[SYS_kill]    "kill",
[SYS_exec]    "exec",
[SYS_fstat]   "fstat",
[SYS_chdir]   "chdir",
[SYS_dup]     "dup",
[SYS_getpid]  "getpid",
```

```
[SYS_sbrk]    "sbrk",
[SYS_sleep]   "sleep",
[SYS_uptime]  "uptime",
[SYS_open]    "open",
[SYS_write]   "write",
[SYS_mknod]   "mknod",
[SYS_unlink]  "unlink",
[SYS_link]    "link",
[SYS_mkdir]   "mkdir",
[SYS_close]   "close",
[SYS_trace]   "trace",//实验1
[SYS_sysinfo] "sysinfo"//实验2
};



void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;//将相应的系统调用号存储至寄存器a7
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) { // 如果系统调用编号有效
    p->trapframe->a0 = syscalls[num](); // 调用，并用a0存储返回值
    // 如果当前进程设置了对该编号系统调用的 trace，则打出 pid、系统调用名称和返回值。
    if((p->syscall_trace >> num) & 1) {//实验1新增的if
      printf("%d: syscall %s -> %d\n",p->pid, syscall_names[num], p-
>trapframe->a0); //发现实验要求追踪的进程
    }// syscall_names[num]: 从 syscall 编号到 syscall 名的映射表
  } else {
    printf("%d %s: unknown sys call %d\n",p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

由于一个进程的每一个系统调用都需要经过 `syscall.c`，所以我们选择从这里入手。

`*syscall_names[]` 数组是为 `syscall` 函数服务的，因此直接讲解 `syscall` 函数。

记录一下调用的是第几个系统调用，如果是第i个，则通过掩码查看掩码第 i位是否为1，若是，则表示被追踪，应该输出进程号，系统调用名称，以及返回值。

其中，在输出系统调用名称时，我们需要创建一个字符串数组进行映射，即 `*syscall_names[]` 数组。

sysproc.c 主函数

```
uint64//新增代码
sys_trace(void)
{
  int mask;//若用户程序调用了trace(5),我们要想办法得到5
//虚拟地址在两种特权级别下可能对应不同的物理内存地址,需要使用 argaddr、argint、argstr 等系列函数
  if(argint(0, &mask) < 0)//  , 从进程的 trapframe 中读取用户进程寄存器中的mask参数。
      return -1;//0 表示要提取第一个参数，索引从 0 开始。如果读取失败，它将返回-1，表示出现了错误。
  myproc()->syscall_trace = mask;// 设置调用进程的 syscall_trace mask
  return 0;
}
```

在经过上述操作后，主函数的代码就好写了，只需要给进程的掩码打一个标记即可。

需要注意由于内核与用户进程的页表不同，寄存器也不互通，所以参数无法直接通过 C 语言参数的形式传过来，而是需要使用 $argaddr$、$argint$、$argstr$ 等系列函数，从进程的 $trapframe$ 中读取用户进程寄存器中的参数。

Sysinfo

这个实验需要增加一个新的SysInfo的系统调用，获取内存当前剩余的空闲页大小以及UNUSED的进程。

def.h

```
……
void            ramdiskinit(void);
void            ramdiskintr(void);
void            ramdiskrw(struct buf*);
uint64             count_free_mem(void); // 实验2
……
int             either_copyout(int user_dst, uint64 dst, void *src, uint64
len);
int             either_copyin(void *dst, int user_src, uint64 src, uint64
len);
void            procdump(void);
uint64            count_process(void); // 实验2
```

首先声明计算空闲页大小的函数和计算进程数的函数。

kalloc.c 获取空闲页表

```
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;// 直接分配空闲页链表的根节点(空闲链表法)
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r; // 把空闲页链表的根节点返回出去，作为内存页使用（长度是 4096）
}
// 实 验 2
uint64
count_free_mem(void) // 空闲页大小
{
  acquire(&kmem.lock); // 必须先锁内存管理结构，防止竞态条件出现
  //空闲内存页的记录方式是，将空虚内存页本身直接用作链表节点，形成一个空闲页链表，每次需要分配，就把链表根部对应的页分配出去。
  uint64 mem_bytes = 0;//每次需要回收，就把这个页作为新的根节点，把原来的 freelist 链表接到后面。
  struct run *r = kmem.freelist;
  while(r)
  {//直接使用空闲页本身作为链表节点
    mem_bytes += PGSIZE;//页大小
    r = r->next;
  }
  release(&kmem.lock);
  return mem_bytes;
}
```

$xv6$ 使用空闲链表法管理空闲页，即将空虚内存页本身直接用作链表节点，形成一个空闲页链表，每次需要分配，就把链表根部对应的页分配出去。每次需要回收，就把这个页作为新的根节点，把原来的 freelist 链表接到后面。

因此在 `count_free_mem` 函数中，我们只需要从空闲页头开始，循环即可找到所有空闲页，需要注意的是由于我们正在读页表，因此需要上锁，防止页表更改造成错误。

proc.c 获取进程数

```
uint64//struct proc proc[NPROC];是全局变量。   NPROC 是一个常量，表示操作系统支持的最大进程数量。
count_process(void)
{ // added function for counting used process slots (lab2)
  uint64 cnt = 0;
  for(struct proc *p = proc; p < &proc[NPROC]; p++)
   {//不需要锁进程 proc 结构，因为我们只需要读取进程列表，不需要写
    if(p->state != UNUSED)  cnt++;
  }
  return cnt;
}
```

由于 $xv6$ 中进程都保存在 `proc[]` 数组中，上限个数为$NPROC$，因此只需遍历这个数组即可，如果不为空，就表示有一个进程。

sysproc.c

```
#include "sysinfo.h"//别忘了!


uint64   //实验2
sys_sysinfo(void)
{
  // 从用户态读入一个指针，作为存放 sysinfo 结构的缓冲区
  uint64 addr;
  if(argaddr(0, &addr) < 0)//需要使用 argaddr、argint、argstr 等系列函数
    return -1;
  struct sysinfo sinfo;
  sinfo.freemem = count_free_mem(); // kalloc.c
  sinfo.nproc = count_process(); // proc.c
  // 使用 copyout，结合当前进程的页表，获得进程传进来的指针（逻辑地址）对应的物理地址]
  // 然后将 &sinfo 中的数据复制到该指针所指位置，供用户进程使用。
  if(copyout(myproc()->pagetable, addr, (char *)&sinfo, sizeof(sinfo)) < 0)//从内核地址空间拷贝数据到用户地址空间
    return -1;
  return 0;
}
```

首先，主函数里别忘了 `#include`

有了前面的支撑，主函数的工作并不多，只需要调用这两个函数即可。

不过需要注意的是同时由于页表不同，指针也不能直接互通访问，也就是内核不能直接对用户态传进来的指针进行解引用，而是需要使用 copyin、copyout 方法结合进程的页表，才能顺利找到用户态指针（逻辑地址）对应的物理内存地址。

测试结果

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (4.2s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.1s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.1s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (24.3s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (4.6s)
== Test time ==
time: OK
Score: 35/35
```

Lab: page tables

Speed up system calls

本实验中要求实验者在创建进程时，在物理内存中创建一个页面以保存进程的PID，并在进程的页表中将 `USYSCALL` 虚拟地址映射至上述页面。通过上述操作，进程即可在用户空间内直接访问其PID，以节省了系统调用的开销。

proc.h

```
// Per-process state
struct proc {
……
  struct usyscall *usyscall;    // <加入一个指向加速页面的指针>
……
};
```

我们首先要修改的是对于进程结构体的定义，在其中添加一枚指针指向usyscall结构体，这枚指针本质上也指向了加速页面的起始物理地址。

proc.c

这里需要修改多个函数，我们分别解释。

allocproc

```
// 查找进程表，找到UNUSED状态的进程
// 如果找到了UNUSED状态的进程，那么初始化它的状态，使其能在内核态下运行
// 并且返回时不占用锁
// 如果没有空闲进程，或内存分配失败，则返回0
static struct proc*
allocproc(void)
{
  struct proc *p;

  // 遍历进程组，寻找处于UNUSED状态的进程
  for(p = proc; p < &proc[NPROC]; p++) {
    // 首先获取进程的锁，保证访问安全
    acquire(&p->lock);

    // 如果进程状态为UNUSED，则跳转至found
    if(p->state == UNUSED) {
      goto found;
    } else {
      // 否则释放锁，检查下一个进程是否为UNUSED状态
      release(&p->lock);
    }
  }
  return 0;

// 以下是初始化一个进程的代码
found:
  // 为当前进程分配PID，并将当前进程状态改为USED
  p->pid = allocpid();
```

```
  p->state = USED;

  // Allocate a trapframe page.
  // 分配一个trapframe页，如果不成功则释放当前进程和锁
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // <照葫芦画瓢，我们也像trapframe一样申请一个空闲物理页用来存储变量>
  // <那么在进程信息里也得保留一份指向这个页面的指针>，这里记为p->usyscall
  // 因为内核执行的是直接映射，所以kalloc返回的指针可以直接当作物理地址映射到页表中
  if((p->usyscall) == (struct usyscall *)kalloc() == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // <将pid信息放入结构体，其实也就是放到了这个物理页面开头处>
  p->usyscall->pid = p->pid;

  // An empty user page table.
  // 为当前进程申请一个页表页，并将trapframe和trampoline页面映射进去
  // 注意trampoline代码作为内核代码的一部分，不用额外分配空间，只需要建立映射关系
  // <我们后面将要修改这个函数，将加速页面一并映射进去>
  p->pagetable = proc_pagetable(p);

  // 如果上述函数执行不成功，则释放当前进程和锁
  if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // 设置新的上下文并从forkret开始执行
  // 这会回到用户态下
  // 这里涉及trap的返回过程，我们在后面研究源码时再深入解读这里的行为
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  // 返回新的进程
  return p;
}
```

由于 `uyscall` 和 `trapframe` 相同，都需要映分配一个页进行映射，因此可以对照着使用 `kalloc` 分配物理页，并创建页表。

proc_pagetable

```
pagetable_t
proc_pagetable(struct proc *p)
{
……
    // <仿照上面的格式将加速页面映射到页表中>
  // 使用mappages将此页映射到页表中
  // 如果出错要释放映射trampoline和trapframe的映射关系
  // 并释放pagetable的内存，返回空指针
  if(mappages(pagetable, USYSCALL, PGSIZE, //将一个页面映射到进程的页表中
              (uint64)(p->usyscall), PTE_R | PTE_U) < 0){//PTE_U用户权限位
      uvmunmap(pagetable, TRAMPOLINE, 1, 0);//从一个执行环境（通常是用户态）切换到另一个执行环境（通常是内核态）的平滑过渡
      uvmunmap(pagetable, TRAPFRAME, 1, 0);//取消映射一个区域的页面
      uvmfree(pagetable, 0);
      return 0;
  }
  // <添加代码结束>
  return pagetable;
}
```

上文中提到了分配页表，具体分配方案在此处。

`PTE_U` 表示用户态可访问，这是加速用户态访问内核信息的关键一步， `PTE_R` 表示可读。

注意到我们已经提前分配了 `TRAMPOLINE` 和 `TRAPFRAME` ，因此若 `USYSCALL` 分配失败，需要释放前二者。

freeproc

```
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;

   // <仿照trapframe内存释放的代码，在出错时将p->usyscall页面释放，并将指针置空>
  if(p->usyscall)//将之前通过 kalloc 分配的内存块标记为可用，以便后续可以重新分配给其他需要内存的部分。
    kfree((void*)p->usyscall);//上面有allocproc
```

```
  p->usyscall = 0;
    // 释放页表
  // proc_freepagetable会调用uvmunmap解除trampoline和trapframe的映射关系
  // 并最终调用uvmfree释放内存和页表
  // <我们后面要修改此处的函数>
  if(p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}
```

由于我们之前为 `usycall` 申请了物理页，因此需要用 `kfree` 释放掉。

此外还要消除地址映射，因此还需要修改 `proc_freepagetable` 函数

proc_freepagetable

```
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{//释放进程页表
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
    // <释放USYSCALL函数的映射关系>
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmfree(pagetable, sz);
}
```

非常简单，模仿 `TRAMPOLINE` 和 `TRAPFRAME` 释放 `USYSCALL` 即可。

总结

首先，allocproc函数会在调用kalloc分配一个物理页面专门用来存储一些不用进入内核态就可以直接读取的信息。kalloc会返回一个指针，虽是虚拟地址，但由于内核页表的直接映射机制，我们也可以将此地址作为物理地址直接使用。随后，我们将想要加速访问的数据(pid)直接放入此页面中，并在进程信息结构体中维持一个指向此页面的指针。注意直到此时，这个页面只是存在了，但是进程还没有将其映射到自己的地址空间中，因而在页表中是访问不到这个页面的。

接下来allocproc会调用proc_pagetable函数，这个函数会创建进程的页表，并将特殊的几个页面(trampoline、trapframe、我们的加速页面)映射到页表中。这几个特殊页面必须已经分配好，即在物理内存中已经存在，我们在这里只是使用建立映射关系。trampoline是内核代码的一部分，始终在内存中存在，trapframe和我们的加速页面在之前的allocproc中已经分配完毕，故它们也是存在的，所以在proc_pagetable中我们使用mappages一一将它们映射到进程页表中。

在上述步骤都完成之后，一个进程在用户态下，直接访问所谓的USYSCALL虚拟地址，这个地址经过多级页表的翻译就会索引到我们分配的加速页面中，进而直接获取到了想要的数据。

Print a page table

在本部分中，需要对页表中所有有效的PDE及PTE进行输出，利用DFS方式的递归方法即可完成。

defs.h

```
void            vmprint(pagetable_t);    // <插入vmprint的函数签名>
```

首先声明函数。

exec.c

```
int
exec(char *path, char **argv)
{
……
  if(p->pid == 1){
    printf("page table %p\n", p->pagetable);
    vmprint(p->pagetable);
  }
  ……
}
```

根据题目要求，在 `exec.c` 中调用我们的 `vmprint` 函数。

vm.c

```
// 此函数借鉴了walkaddr的写法，用来递归地打印页表
void
raw_vmprint(pagetable_t pagetable, int Layer)
{
```

```
  // 遍历页表的每一项
  for(int i = 0 ; i < 512 ; ++i){
    pte_t pte = pagetable[i];

    // 如果当前的pte指向的是更低一级的页表
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){//不能 可读可写可执行 ，说明是子页表
      // 从PTE中解析出物理地址，并打印指定数量的缩进符
      // 注意解析物理地址时，不能只是简单地将权限位移除出去，还应该左移12位，让出页内偏移量
      uint64 phaddr = (pte >> 10) << 12;//这是因为页表项中的物理地址只占用了低 52 位，需要左移以清除权限位和页内偏移。
      for(int i=Layer ; i != 0 ; --i)
        printf(".. ");

      // 打印本级页表信息，向孩子页表递归，注意层数+1
      printf("..%d: pte %p pa %p\n", i, pte, phaddr);//索引 i、页表项的内容 pte 以及解析得到的物理地址 phaddr。
      uint64 child = PTE2PA(pte);
      raw_vmprint((pagetable_t)child, Layer + 1);
    }

    // 如果当前PTE指向的是叶级页表
    // 取出物理地址并打印信息，随后返回
    else if (pte & PTE_V){//PTE_V: 有效位 (Valid)
      uint64 phaddr = (pte >> 10) << 12;
      printf(".. .. ..%d: pte %p pa %p\n", i, pte, phaddr);
    }
  }
}

void vmprint(pagetable_t pagetable)
{
  raw_vmprint(pagetable, 0);
}
```

这是一个递归函数。

首先，当递归到当前页表时，首先访问页表中所有的512项，如果访问到一项有效，则说明需要打印信息。

我们首先判断它是更低一级的子页表还是叶级页表，如果它不可读不可写不可执行，则为低一级页表，输出后需要继续递归，如果可以，说明是叶级页表，输出即可。

输出时，我们要先右移10位，去除掉所有的权限位，再左移12位，因为页表项中的物理地址只占用了低 52 位。

Detecting which pages have been accessed

在本部分中，需要增加一个系统调用，以返回用户空间虚拟地址中的“脏页”分布情况，脏页即被访问过的页面。具体的调用方式为，用户输入检测的起始虚拟地址和检测页面数，内核在计算后通过指针返回用于表示脏页分布情况的位掩码。

riscv.h

```
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_A (1L << 6) // <加入对访问位的支持>
```

通过查询资料，得到PTE_A的值。

需要注意的是，PTE_A的含义为是否被访问过。

sysproc.c

```
#ifdef LAB_PGTBL

extern pte_t * walk(pagetable_t, uint64, int);
//在本部分中，需要增加一个系统调用，以返回用户空间虚拟地址中的“脏页”分布情况，脏页即被访问过的页面。
int//具体的调用方式为，用户输入检测的起始虚拟地址和检测页面数，内核在计算后通过指针返回用于表示脏页分布情况的位掩码。
sys_pgaccess(void)//简单来说，就是检测某个页面是否被访问的
{
  // lab pgtbl: your code here.

      uint64 va,dst;
    int n;
    if(argint(1, &n) < 0 || argaddr(0, &va) < 0 || argaddr(2, &dst) < 0)
        return -1;//va起始地址
    if(n > 64 || n < 0)
        return -1;
    uint64 bitmask = 0,mask = 1;
    pte_t *pte;
    pagetable_t pagetable = myproc()->pagetable;
    while(n > 0){//n检查页数
        pte = walk(pagetable,va,1);//指向页表项（Page Table Entry，PTE）的指针
        if(pte){
            if(*pte & PTE_A)//1<<6 表示访问过
                bitmask |= mask;
            *pte = *pte & (~PTE_A);// 清除访问标记
```

```
            }
            mask <<= 1;
            va = (uint64)((char*)(va)+PGSIZE);
            n--;
        }
        if(copyout(pagetable,dst,(char *)&bitmask,sizeof(bitmask)) < 0)//将结果传给
    用户态
            return -1;

      return 0;
    }
    #endif
```

系统调用采用三个参数。首先，它需要检查的第一个用户页面的起始虚拟地址va；其次，它需要检查的页数n；最后，它会获取一个用户地址到缓冲区dst，将结果存储到位掩码中。

我们实现它的具体思路为循环枚举。从起始地址va开始循环n次，每次使va跳转到下一个页，并记录当前页掩码左移一位。

需要注意，如果当前页的PTE_A被设置，我们要清除PTE_A。否则，将无法确定上次调用pgaccess() 后是否访问过页面。

其他工作

answers-pgtbl.txt

本实验还要求回答若干问题，我们附在此处。

> Which other xv6 system call(s) could be made faster using this shared page? Explain how.

可能受益的系统调用以及它们如何受益的解释：

1. `read` 和 `write` 系统调用：这些系统调用用于文件的读取和写入。通过共享页面，内核可以将文件数据的某些部分映射到用户进程的地址空间中，允许用户进程直接读取或写入内核中的数据，而无需进行额外的数据拷贝。这可以减少系统调用的次数和数据传输的开销，提高性能。
2. `mmap` 系统调用：`mmap` 用于将文件或设备映射到进程的地址空间中。同理，允许进程以页面的形式访问文件内容。这可以加快文件访问速度，因为文件数据可以在需要时直接从内核中读取，而不是通过系统调用进行文件读取。
3. `exec` 系统调用：`exec` 用于加载新的程序映像并替换当前进程的地址空间。同样的，内核可以在新程序映像中包含一些常用的库或数据，这些库或数据可以在多个进程之间共享，而不需要每次都加载新的副本。这可以减少内存占用和加载时间。

Explain the output of `vmprint` in terms of Fig 3-4 from the text. What does page 0 contain? What is in page 2? When running in user mode, could the process read/write the memory mapped by page 1? What does the third to last page contain?

关于 `vmprint` 输出的解释，可以参考教材中的图 3-4：

- 页面 0 包含内核的代码和数据，通常不允许用户进程直接读取或写入它。
- 页面 2 包含 trampoline 代码，这是用户态和内核态之间切换的桥梁。它允许用户进程进入内核执行系统调用，但不允许用户进程直接访问或修改它。
- 当在用户模式下运行时，用户进程通常不能读取或写入由页面 1 映射的内核内存，因为它受到保护。只有在系统调用的上下文中，内核才能访问页面 1。
- 倒数第三页通常包含 trapframe，这是在系统调用时用于保存用户进程状态的数据结构。它包含了用户进程的寄存器值和其他状态信息。它也是内核执行系统调用所必需的。

总之，`vmprint` 输出反映了 xv6 的内存布局，包括内核代码、用户代码、共享页面和其他重要数据结构。页面 0 和页面 2 包含了一些关键的内容，而页面 1 通常由内核保护，只有在系统调用上下文中才能访问。倒数第三页包含了 trapframe，用于保存用户进程的状态。这些页面的内容在 xv6 的运行过程中起着重要作用。

测试结果

```
== Test pgtbltest ==
$ make qemu-gdb
(4.3s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (1.1s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(148.4s)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

Lab: traps

RISC-V assembly

这是一个学习性质的实验，回答对应的问题即可。

1. 哪些寄存器储存了函数的参数？例如，哪些寄存器存储了在主函数被调用的 `printf` 的参数13

   根据RISCV手册，a0-a7寄存器保存了函数的参数，在 `printf` 中，参数13被存储在a2寄存器中。

2. 主函数中对 `f` 和 `g` 的调用在哪里（提示：编译器可能会内联函数

很显然，主函数中没有调用这两个函数，而是有编译器直接计算出调用结果12。 p 3. printf函数位于哪个地址

主函数中对 `printf` 间接寻址，其地址位于 `30 + 1536 = 0x630` 处。

1. 在jalr跳转至主函数的printf时，寄存器ra中有什么值

`jalr offset(rd)` 的功能为： `t =pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t` 。在使用 `0x80000332 jalr -1298(ra)` 跳入 `printf` 后， `ra` 的值为 `0x80000332+4 = 0x80000336` 。

1. 运行以下代码：

```
unsigned int i = 0x00646c72;
printf("H%x Wo%s", 57616, &i);
```

输出是什么？上述输出是基于RISC-V为小端法的事实的。如果RISC-V为大端法，i应该被设置为什么，57616需要被改变吗？

上述代码输出为 `HE110 World` 。如果为大端法，i需要被更改为 `0x726c6400` ，57616不需要被改变，硬件会处理具体的存储和读取方式。

1. 在下面的代码中，会打印 `'y='` 什么？为什么会发生这种情况？

```
printf("x=%dy=%d", 3);
```

会打印出 `a2` 寄存器当前的值，因为在调用 `printf` 时，并未给 `a2` 寄存器赋值，但在 `printf` 被执行的时候却无法检查当前 `a2` 中的值是否为用户传入的值，而只是单纯的对其进行打印。

Backtrace

在本部分中，需要实现可以跟踪当前函数的函数调用情况的函数 `backtrace()` 。即用 `backtrace()` 来访问那些编译器在每个堆栈帧中放置的存放调用方的帧指针来浏览堆栈并打印在每个堆栈帧中保存的返回地址。

具体实验步骤如下

defs.h

```
void            backtrace(void); // new
```

首先声明函数。

riscv.h

```
static inline uint64
r_fp()//添加获取当前 fp (frame pointer) 寄存器
{//fp 指向当前栈帧的开始地址，sp 指向当前栈帧的结束地址。
  uint64 x;//栈从高地址往低地址生长，所以 fp 虽然是帧开始地址，但是地址比 sp 高
  asm volatile("mv %0, s0" : "=r" (x));
  return x;
}
```

fp指向当前栈帧的开始地址，sp 指向当前栈帧的结束地址，从栈底走向栈顶的过程就可以模拟回溯过程。

需要注意，栈从高地址往低地址生长，所以 fp 虽然是帧开始地址，但是地址比 sp 高。

printf.c

```
void backtrace() {
  uint64 fp = r_fp();
  while(fp != PGROUNDUP(fp)) { // 如果已经到达栈底
    uint64 ra = *(uint64*)(fp - 8); //栈帧中从高到低第一个 8 字节 fp-8 是 return address，也就是当前调用层应该返回到的地址
    printf("%p\n", ra);
    fp = *(uint64*)(fp - 16); // 栈帧中从高到低第二个 8 字节 fp-16 是 previous address，指向上一层栈帧的 fp 开始地址
  }
}
```

栈帧中从高到低第一个 8 字节 `fp-8` 是 return address，也就是当前调用层应该返回到的地址，这是我们需要输出的东西。
栈帧中从高到低第二个 8 字节 `fp-16` 是 previous address，指向上一层栈帧的 fp 开始地址，这是我们回溯的下一个目标。
剩下的为保存的寄存器、局部变量等。

不难发现一个栈帧的大小不固定，但是至少 16 字节。
在 xv6 中，使用一个页来存储栈，如果 fp 已经到达栈页的上界，则说明已经到达栈底，此时终止循环即可。

sysproc.c

```
uint64
sys_sleep(void)
{
  int n;
  uint ticks0;
backtrace();
……
}
```

最后根据题目要求在 `sysproc.c` 文件的 `sys_sleep()` 函数中调用 `bracktrace()` 即可。

Alarm

在本实验中，需要实现新的系统调用 `sys_sigalarm` 和 `sys_sigreturn` ，以实现定时器功能函数 `sigalarm` 。当用户进程调用 `sigalarm(n,fn)` 时，进程将以 `n` 个CPU时间滴答为周期调用定时器处理函数 `fn` ，当函数返回后进程应当从被中断的地方继续。

如何在xv6系统中增加系统调用已经在 `syscall Lab` 中被详细介绍，在这里不加赘述。直接讲解系统调用的实现。

proc.h

```
struct proc {
……
  int alarm_interval;          // Alarm interval (0 for disabled)
  void(*alarm_handler)();      // Alarm handler
  int alarm_ticks;             // How many ticks left before next alarm goes
off
  struct trapframe *alarm_trapframe;  // A copy of trapframe right before
running alarm_handler
  int alarm_goingoff;          // Is an alarm currently going off and hasn't
not yet returned? (prevent re-entrance of alarm_handler)
}
```

alarm_interval：时钟周期，0 为禁用

alarm_handler：时钟回调处理函数

alarm_ticks：下一次时钟响起前还剩下的 ticks 数

alarm_trapframe：时钟中断时刻的 trapframe，用于中断处理完成后恢复原程序的正常执行

alarm_goingoff：是否已经有一个时钟回调正在执行且还未返回（用于防止在 alarm_handler 中途闹钟到期再次调用 alarm_handler，导致 alarm_trapframe 被覆盖）

sysproc.c

```
int sigalarm(int ticks, void(*handler)()) {//进程将以 n 个CPU时间滴答为周期调用定时
器处理函数 fn
  // 设置 myproc 中的相关属性
  struct proc *p = myproc();
  p->alarm_interval = ticks;
  p->alarm_handler = handler;
  p->alarm_ticks = ticks;
  return 0;
}

int sigreturn() {//解锁
  // 将 trapframe 恢复到时钟中断之前的状态，恢复原本正在执行的程序流
  struct proc *p = myproc();
  *p->trapframe = *p->alarm_trapframe;
  p->alarm_goingoff = 0;
  return 0;
}
```

```
uint64 sys_sigalarm(void) {//设置警报器
  int n;
  uint64 fn;
  if(argint(0, &n) < 0)
    return -1;
  if(argaddr(1, &fn) < 0)
    return -1;//读入两个数

  return sigalarm(n, (void(*)())(fn));
}

uint64 sys_sigreturn(void) {
    return sigreturn();
}
```

在 `sys_sigalarm` 函数中，读入用户态的n和fn，然后当前进程开始倒计时。

`sys_sigreturn` 函数是一个解锁的的过程，主要目的是恢复因中断而改变的 `trapframe` 中的信息，以及重新开始一轮新的倒计时。

proc.c

```
static struct proc*
allocproc(void)
{
  // ......

found:
  p->pid = allocpid();

  // Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    release(&p->lock);
    return 0;
  }

  // Allocate a trapframe page for alarm_trapframe.
  if((p->alarm_trapframe = (struct trapframe *)kalloc()) == 0){
    release(&p->lock);
    return 0;
  }

  p->alarm_interval = 0;
```

```
  p->alarm_handler = 0;
  p->alarm_ticks = 0;
  p->alarm_goingoff = 0;

  // ......

  return p;
}

static void
freeproc(struct proc *p)
{
  // ......

  if(p->alarm_trapframe)
    kfree((void*)p->alarm_trapframe);
  p->alarm_trapframe = 0;

  // ......

  p->alarm_interval = 0;
  p->alarm_handler = 0;
  p->alarm_ticks = 0;
  p->alarm_goingoff = 0;
  p->state = UNUSED;
}
```

由于我们在 `proc.h` 中加入了新的变量，所以的申请和释放进程需要进行相应的修改。

在新进程建立时，除 `alarm_trapframe` 类型为 `trapframe *`,需要 `kalloc` 外，其他变量都是整数，直接赋值为0即可。

同样的，在释放进程时， `alarm_trapframe` 需要 `kfree` ，其余变量直接赋0即可。

trap.c

```
void
usertrap(void)
{
……
   if(which_dev == 2) {//表示中断源是CPU时钟中断
    if(p->alarm_interval != 0) { // 如果设定了时钟事件
      if(--p->alarm_ticks <= 0) { // 时钟倒计时 -1 tick，如果已经到达或超过设定的
tick 数
        if(!p->alarm_goingoff) { // 确保没有时钟正在运行
```

```
            p->alarm_ticks = p->alarm_interval;
            // jump to execute alarm_handler
            *p->alarm_trapframe = *p->trapframe; // backup trapframe
            p->trapframe->epc = (uint64)p->alarm_handler;
            p->alarm_goingoff = 1;
          }
          // 如果一个时钟到期的时候已经有一个时钟处理函数正在运行，则会推迟到原处理函数运行完成后的下一个 tick 才触发这次时钟
        }
      }
      yield();
    }
    usertrapret();
}
```

在 `usertrap` 函数中，内核处理所有来自用户的中断，其中包括CPU时钟中断。当 `which_dev = devintr()) == 2` 时，当前中断类型为CPU时钟中断。

此时若此进程被设置过警报器，则我们减小倒计时器，如果减值0，说明应发出警报，但若上次的锁还未打开，则为了程序安全不进行操作，锁是开的则先上锁，再执行回调函数。

测试结果

```
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (2.8s)
== Test running alarmtest ==
$ make qemu-gdb
(3.5s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (113.0s)
== Test time ==
time: OK
Score: 85/85
```

Lab: Copy-on-Write Fork for xv6

Implement copy-on write

本实验要求为xv6系统增加 `fork` 函数写时复刻功能，即当 `fork` 完成时父子进程共享同一组只读物理内存页。当进程试图读写这些物理内存时，为该进程重新分配新的物理内存页，并将共享物理内存页的内容拷贝至新内存页。

defs.h

```
.....
void            krefpage(void *);
void            *kcopy_n_deref(void *pa);
.....
int             uvmcheckcowpage(uint64 va);
int             uvmcowcopy(uint64 va);
.....
```

首先定义函数

riscv.h

```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_COW (1L << 8) // 是否为懒复制页，使用页表项 flags 中保留的第 8 位表示
//用于标示一个映射对应的物理页是否是懒复制页
```

首先定义 `PTE_COW` 标志位，用于标示一个映射对应的物理页是否是懒复制页。

vm.c

```
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");
    pa = PTE2PA(*pte);
    if(*pte & PTE_W) {//清除父进程的写入标志并设置 COW 标志，以便后续写操作会引发页错误，从而执行复制。
      // clear out PTE_W for parent, set PTE_COW
      *pte = (*pte & ~PTE_W) | PTE_COW;
    }
    flags = PTE_FLAGS(*pte);
// 将父级的物理页直接映射到子级（写时复制）
      // 因为父级的写标志已经被清除
      // 子映射也不会具有写入标志。
      // 对于父级已经只读的页面，它将被读取-
      // 也仅适用于子
      // 对于也是一个cow页面的只读页面，PTE_COW标志将
      // 复制到子页面，自动使其成为cow页面。
    if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){//函数将物理页映射到新进程的页表中
      goto err;
    }
    // for any cases above, we created a new reference to the physical
    // page, so increase reference count by one.
    krefpage((void*)pa);//增加引用计数
  }
  return 0;
 err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

首先修改 `uvmcopy()`，在复制父进程的内存到子进程的时候，不立刻复制数据，而是建立指向原物理页的映射，并将父子两端的页表项都设置为不可写。

下面有两个地方需要检测是否需要复制：

```
//vm.c
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){//copyout() 由于是软件访问页表，不会触发缺页异常
    if(uvmcheckcowpage(dstva)) // 检查每一个被写的页是否是 COW 页
    uvmcowcopy(dstva);
    ……
  }
}
```

`copyout()` 由于是软件访问页表，表示复制，需要手动添加监测代码，检测接收的页是否是一个懒复制页，若是，执行实复制操作。

```
//trap.c
void
usertrap(void)
{
  ……
    // ok            13 表示页访问错误（Page Fault），15 表示访问权限错误。
  } else if((r_scause() == 13 || r_scause() == 15) &&
uvmcheckcowpage(r_stval())) { // 并且是复制页
    if(uvmcowcopy(r_stval()) == -1){ // 如果内存不足，则杀死进程
      p->killed = 1;
    }
  } else {
    ……
}
```

如果尝试修改懒复制的页，会出现 `page fault` 被 `usertrap()` 捕获。接下来需要在 `trap.c` 的 `usertrap()` 中捕捉这个 `page fault`，并在尝试修改页的时候，执行实复制操作。

具体的检测和复制过程如下：

```
//新函数，  检查一个地址指向的页是否是懒复制页
int uvmcheckcowpage(uint64 va) {
  pte_t *pte;
  struct proc *p = myproc();

  return va < p->sz // 在进程内存范围内
    && ((pte = walk(p->pagetable, va, 0))!=0)// 页表能访问到va
```

```
    && (*pte & PTE_V) // 页表项存在（Valid有效）
    && (*pte & PTE_COW); // 页是一个懒复制页
}

// 实复制一个懒复制页，并重新映射为可写
int uvmcowcopy(uint64 va) {
  pte_t *pte;
  struct proc *p = myproc();

  if((pte = walk(p->pagetable, va, 0)) == 0)
    panic("uvmcowcopy: walk");

  // 调用 kalloc.c 中的 kcopy_n_deref 方法，复制页
  // （如果懒复制页的引用已经为 1，则不需要重新分配和复制内存页，只需清除 PTE_COW 标记并标记
PTE_W 即可）
  uint64 pa = PTE2PA(*pte);// 物理地址
  uint64 new = (uint64)kcopy_n_deref((void*)pa); // 将一个懒复制的页引用变为一个实复
制的页
  if(new == 0)
    return -1;

  // 重新映射为可写，并清除 PTE_COW 标记
  uint64 flags = (PTE_FLAGS(*pte) | PTE_W) & ~PTE_COW;
  uvmunmap(p->pagetable, PGROUNDDOWN(va), 1, 0);
  if(mappages(p->pagetable, va, 1, new, flags) == -1) {
    panic("uvmcowcopy: mappages");
  }
  return 0;
}
```

检查是否为懒页的核心在于查看 `COW` 标志位，当然不能忘记其他边界条件

实复制时后记得清除 `COW` 位再从页表中映射，更具体的 `kcopy_n_deref` 函数的细节我们在下面讲解。

kalloc.c

```
// 用于访问物理页引用计数数组
#define PA2PGREF_ID(p) (((p)-KERNBASE)/PGSIZE)//将物理地址减去内核的基地址
KERNBASE，然后除以页面大小 PGSIZE，以确定在 pageref 数组中的位置。
#define PGREF_MAX_ENTRIES PA2PGREF_ID(PHYSTOP)//确定引用计数数组的最大条目数，也就是
数组的大小。（物理内存的结束地址）

struct spinlock pgreflock; // 用于 pageref 数组的锁，防止竞态条件引起内存泄漏
```

```
int pageref[PGREF_MAX_ENTRIES]; // 从 KERNBASE 开始到 PHYSTOP 之间的每个物理页的引用计数
// note:  reference counts are incremented on fork, not on mapping. this means that
//        multiple mappings of the same physical page within a single process are only
//        counted as one reference. 引用计数在 fork 时递增，而不是在映射时递增。
//        this shouldn't be a problem, though. as there's no way for a user program to map
//        a physical page twice within it's address space in xv6.

// 通过物理地址获得引用计数
#define PA2PGREF(p) pageref[PA2PGREF_ID((uint64)(p))]//用于将给定的物理地址 p 转换为对应物理页的计数

void
kinit()
{
  initlock(&kmem.lock, "kmem");
  initlock(&pgreflock, "pgref"); // 初始化锁
  freerange(end, (void*)PHYSTOP);
}
```

首先定义 `PA2PGREF(p)` 表示物理地址 `p` 表示的页的引用计数。

这里的锁的作用是防止竞态条件 `(race-condition)` 下导致的内存泄漏,下同。

```
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  acquire(&pgreflock);
  if(--PA2PGREF(pa) <= 0) {
    // 当页面的引用计数小于等于 0 的时候，释放页面

    // pa will be memset multiple times if race-condition occurred.
  // Fill with junk to catch dangling refs.
```

```
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
  }//keme指向空闲内存页列表的指针,分配内存页
  release(&pgreflock);
}
```

释放物理页 `kfree()` 使引用计数减1；如果计数变为0，则释放回收物理页

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
  {
    memset((char*)r, 5, PGSIZE); // fill with junk
    // 新分配的物理页的引用计数为 1
    // （这里无需加锁）
    PA2PGREF(r) = 1;//计数为1
  }
  return (void*)r;
}
```

分配物理页时吗，将其引用计数置为1

```
// Decrease reference to the page by one if it's more than one, then
// allocate a new physical page and copy the page into it.
// (Effectively turing one reference into one copy.)
//
```

```
// Do nothing and simply return pa when reference count is already
// less than or equal to 1.
//
// 当引用已经小于等于 1 时，不创建和复制到新的物理页，而是直接返回该页本身
void *kcopy_n_deref(void *pa) {
  acquire(&pgreflock);

  if(PA2PGREF(pa) <= 1) { // 只有 1 个引用，无需复制
    release(&pgreflock);
    return pa;
  }

  // 分配新的内存页，并复制旧页中的数据到新页
  uint64 newpa = (uint64)kalloc();
  if(newpa == 0) {//MLE
    release(&pgreflock);
    return 0; // out of memory
  }
  memmove((void*)newpa, (void*)pa, PGSIZE);

  // 旧页的引用减 1
  PA2PGREF(pa)--;

  release(&pgreflock);
  return (void*)newpa;
}

// 为 pa 的引用计数增加 1
void krefpage(void *pa) {
  acquire(&pgreflock);
  PA2PGREF(pa)++;
  release(&pgreflock);
}
```

这两个函数是我们自己定义的函数

`krefpage` 表示使引用计数加一

`kcopy_n_deref` 表示将物理页的一个引用实复制到一个新物理页上（引用计数为1），返回得到的副本页；并将本物理页的引用计数减1

测试结果

```
== Test running cowtest ==
$ make qemu-gdb
```

```
(8.8s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(119.3s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 110/110
```

Lab: Multithreading

Uthread: switching between threads

在本部分中，需要我们实现xv6用户级线程包。

uthread_switch.S

这里需要实现上下文切换的代码，借鉴 `swtch.S` 中的写法

```
    .text

    /*
         * save the old thread's registers,
         * restore the new thread's registers.
         */

    .globl thread_switch
thread_switch:
    /* YOUR CODE HERE */
    sd ra, 0(a0)
    sd sp, 8(a0)
```

```
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)
/*  Store Double, 用于将一个双字（64位数据）存储到内存中。它将指定寄存器中的数据存储到内存的指定地址。 */
/*ld 表示 Load Double, 用于从内存中加载一个双字（64位数据）到寄存器中。它从内存的指定地址加载数据并将其放入指定寄存器。*/
/*其中, a0 为当前线程的 context 结构地址, a1 为被调度线程的context 结构地址。*/
/*只需保存 callee-saved 寄存器, 以及 返回地址 ra、栈指针 sp 即可*/
    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)

    ret    /* return to ra */
```

需要注意的是，在调用本函数 `uthread_switch()` 的过程中， `caller-saved registers` 已经被调用者保存到栈帧中了，所以这里无需保存这一部分寄存器。

具体来说，内核调度器无论是通过时钟中断进入，还是线程自己主动放弃 CPU，最终都会调用到 yield 进一步调用 `swtch` 。 由于上下文切换永远都发生在函数调用的边界，恢复执行相当于是 `swtch` 的返回过程，因此不需要向中断可能在任何地方发生的 `trapframe` 那样，所以恢复的时候需要靠 pc 寄存器来定位。 并且由于后者切换位置不一定是函数调用边界，所以几乎所有的寄存器都要保存，才能保证正确的恢复执行。

uthrade.c

```
// Saved registers for thread context switches.
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};


struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct context ctx; // 在 thread 中添加 context 结构体
};
struct thread all_thread[MAX_THREAD];
struct thread *current_thread;
extern void thread_switch(struct context* old, struct context* new); // 修改
thread_switch 函数声明
```

从 `proc.h` 中借鉴一下 `context` 结构体，用于保存 `ra` 、 `sp` 以及 `callee-saved registers`

```
void
thread_schedule(void)
{

……

  if (current_thread != next_thread) {         /* switch threads?  */
    next_thread->state = RUNNING;
```

```
    t = current_thread;
    current_thread = next_thread;
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
     */
    thread_switch(&t->ctx, &next_thread->ctx); // 切换线程
  } else
    next_thread = 0;
}
```

`thread_schedule` 用于调度线程执行，其遍历所有线程，寻找可执行线程，并调用 `thread_switch` 实施具体的线程切换

```
void
thread_create(void (*func)())
{
  struct thread *t;

  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  t->ctx.ra = (uint64)func; // 返回地址     这意味着当线程开始执行时，它将从 func 函数的地址处开始执行。这是线程的入口点。
  // thread_switch 的结尾会返回到 ra，从而运行线程代码
  t->ctx.sp = (uint64)&t->stack + (STACK_SIZE - 1);  // 栈指针
  // 将线程的栈指针指向其独立的栈，注意到栈的生长是从高地址到低地址，所以
  // 要将 sp 设置为指向 stack 的最高地址

}
```

设置上下文中 `ra` 指向的地址为线程函数的地址，这样在第一次调度到该线程，执行到 `thread_switch` 中的 `ret` 之后就可以跳转到线程函数从而开始执行了。设置 `sp` 使得线程拥有自己独有的栈，也就是独立的执行流。

Using threads

answers-thread.txt

Why are there missing keys with 2 threads, but not with 1 thread? Identify a sequence of events with 2 threads that can lead to a key being missing. Submit your sequence with a short explanation in answers-thread.txt

```
假设键 k1、k2 属于同个bucket

thread 1: 尝试设置 k1
thread 1: 发现 k1 不存在，尝试在bucket末尾插入 k1
切换到 thread 2
thread 2: 尝试设置 k2
thread 2: 发现 k2 不存在，尝试在bucket末尾插入 k2
thread 2: 分配 entry，在bucket插入 k2
切换回 thread 1
thread 1: 分配 entry，没有意识到 k2 的存在，在其认为的 “bucket”（实际为 k2 所处位置）插入 k1

k1 被插入，但是由于被 k1 覆盖，k2 消失，引发了键值丢失
```

因此需要在代码执行中进行适当地上锁。

ph.c

```
int
main(int argc, char *argv[])
{
  pthread_t *tha;
  void *value;
  double t1, t0;
  for(int i=0;i<NBUCKET;i++) {
    pthread_mutex_init(&locks[i], NULL);
  }
  ……
}
```

首先在 `main` 函数中进行初始化

```
pthread_mutex_t locks[NBUCKET];//两个线程不会同时操作同一个 哈希表 即可，并不需要确保不会同时操作整个哈希表。
static
void put(int key, int value)
```

```
{
  int i = key % NBUCKET;

  // is the key already present?
  pthread_mutex_lock(&locks[i]);
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  if(e){
    // update the existing key.
    e->value = value;
  } else {
    // the new is new.
    insert(key, value, &table[i], table[i]);
  }
  pthread_mutex_unlock(&locks[i]);
}
```

在插入数据时，在相应的位置对当前 `bucket` 进行上锁。

```
static struct entry*
get(int key)
{
  int i = key % NBUCKET;
pthread_mutex_lock(&locks[i]);

  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key) break;
  }
  pthread_mutex_unlock(&locks[i]);
  return e;
}
```

同理，查询时也需要对当前 `bucket` 上锁。

主要注意的是，只需要确保两个线程不会同时操作同一个 `bucket` 即可，不能对整个哈希表进行上锁，因为这样意味着每一时刻只能有一个线程在操作哈希表，这里实际上等同于将哈希表的操作变回单线程了，又由于锁操作是有开销的，所以性能甚至不如单线程版本。

Barrier

在本部分中，需要利用UNIX接口实现 `barrier` 屏障功能，其功能为只有当参与屏障的所有线程均调用 `barrier` 函数时，任意线程才可以从 `barrier` 中返回，否则就阻塞在 `barrier` 中。

barrier.c

```
static void
barrier()
{
  // YOUR CODE HERE
  // 你需要在这里添加你的代码
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  // 阻塞，直到所有线程都调用了 barrier()，然后增加 bstate.round。
  // 获得 barrier_mutex 锁，用于互斥访问屏障状态
  pthread_mutex_lock(&bstate.barrier_mutex);
  // 如果还有线程没有达到屏障
  if (++bstate.nthread < nthread) {
    // 当前线程进入等待状态，释放 barrier_mutex 互斥锁
    // 在其他线程唤醒时重新获取锁
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  } else {
    // 如果所有线程都已经达到屏障
    bstate.nthread = 0; // 重置 nthread 计数为 0
    bstate.round++;     // 增加轮次 round
    // 唤醒所有休眠在 barrier_cond 条件变量上的线程
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
  // 释放 barrier_mutex 互斥锁
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

线程进入同步屏障 `barrier` 时，将已进入屏障的线程数量增加 1，然后再判断是否已经达到总线程数。
如果未达到，则进入睡眠，等待其他线程。
如果已经达到，则唤醒所有在 `barrier` 中等待的线程，所有线程继续执行；屏障轮数 +1；

「将已进屏障的线程数量增加 1，然后再判断是否已经达到总线程数」这一步并不是原子操作，并且这一步和后面的两种情况中的操作「睡眠」和「唤醒」之间也不是原子的。如果在这里发生 `race-condition`，则会导致出现「`lost wake-up` 问题」。解决方法是使用一个互斥锁 `barrier_mutex` 来保护这一部分代码。`pthread_cond_wait` 会在进入睡眠的时候原子性的释放 `barrier_mutex`，从而允许后续线程进入 `barrier`，防止死锁。

测试结果

```
$ make qemu-gdb
uthread: OK (5.3s)
    (Old xv6.out.uthread failure log removed)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/ysl/daima/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/ysl/daima/xv6-labs-2021'
ph_safe: OK (24.5s)
== Test ph_fast == make[1]: Entering directory '/home/ysl/daima/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/ysl/daima/xv6-labs-2021'
ph_fast: OK (53.7s)
== Test barrier == make[1]: Entering directory '/home/ysl/daima/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/ysl/daima/xv6-labs-2021'
barrier: OK (4.3s)
== Test time ==
time: OK
Score: 60/60
```

Lab: networking

Your Job

在本实验中，需要补全xv6的网络驱动程序，使其可以基于E1000设备进行网络通信。

E1000的接受与传输机制

为了完成本实验，我们必须要了解E1000硬件的数据包接受与传输机制。在软件层面上，E1000的接受和传输主要都是基于一种名为 **描述符** 的数据结构完成的。描述符担任了硬件与软件数据传输中介的功能，并为E1000设备提供了DMA（直接内存访问）机制。在每个描述符中都存放了一个位于RAM内存的地址，其可被硬件和软件共同访问和操作。并且，为了提升E1000设备的大规模数据承载能力，E1000驱动程序维护了 **描述符环** 数据结构作为数据包传输和接受的缓冲机制。

接受描述符环

接受描述符的数据结构如下所示：

```
struct rx_desc
{
    uint64 addr;       /* Address of the descriptor's data buffer */
    uint16 length;     /* Length of data DMAed into data buffer */
    uint16 csum;       /* Packet checksum */
    uint8 status;      /* Descriptor status */
    uint8 errors;      /* Descriptor Errors */
    uint16 special;
};
```

其中，主要的条目为 `addr` 、 `length` 和 `status` 。其中 `addr` 为该描述符所映射的地址，其内存由驱动程序软件预先分配，并当E1000设备接受到数据包时被硬件填充。 `length` 为硬件填充的数据长度。 `status` 中存放了当前描述符的状态，其中最重要的状态位为 `E1000_RXD_STAT_DD` ，当硬件完成了对当前描述符的填充时，该位被置为1，以通知驱动程序软件将其提取。其数据结构为一个循环队列。

传输描述符环

传输描述符的数据结构如下所示：

```
struct tx_desc
{
    uint64 addr;
    uint16 length;
    uint8 cso;
    uint8 cmd;
    uint8 status;
    uint8 css;
    uint16 special;
};
```

其中，主要的条目为 `addr` 、 `length` 、 `cmd` 和 `status` 。 `addr` 为该描述符所映射的内存，其内容被软件填充，并被硬件所提取； `length` 为软件填充的数据长度； `cmd` 控制了该描述符的行为，当 `E1000_TXD_CMD_EOP` 被设置时，表示当前描述符为用于存储一块数据包的最后一个描述符（当数据包较大时可由多个描述符共同存储），当 `E1000_TXD_CMD_RS` 被设置时，硬件在提取当前描述符的内容时，会自动将该描述符的 `status` 置为 `E1000_TXD_STAT_DD` 。

e1000.c

```
//"buffer" 通常指的是用于存储网络数据包的内存区域或数据结构。这些数据包缓冲区被用于暂时存储从网络接口卡（NIC）接收到的数据包或要发送到网络的数据包。
int
e1000_transmit(struct mbuf *m)//描述符是一种元数据，它不存储实际的数据内容，而只是提供了数据的相关信息。
{
  acquire(&e1000_lock); // 获取 E1000 的锁，防止多进程同时发送数据出现 race
  // 尝试获取该锁，如果已经被其他进程占用，则会阻塞等待，直到获取到锁为止。这样可以保证同一时间只有一个进程可以访问 E1000 网卡的资源，避免数据混乱或丢失。

  uint32 ind = regs[E1000_TDT]; // 下一个可用的 buffer 的下标
  //regs 数组是一个映射了 E1000 网卡寄存器的内存区域，E1000_TDT 是一个常量，表示 Transmit Descriptor Tail Register 的下标
  //，该寄存器存储了下一个可用的发送 buffer 的下标!

  //传输描述符结构体      tx_ring是存储传输描述符的数组
  struct tx_desc *desc = &tx_ring[ind]; // 获取 buffer 的描述符，其中存储了关于该 buffer 的各种信息
  // tx_ring 数组是一个存储了发送描述符的内存区域，每个描述符包含其物理地址、长度、命令、状态等信息。

  // 如果该 buffer 中的数据还未传输完，则代表我们已经将环形 buffer 列表全部用完，缓冲区不足，返回错误
  if(!(desc->status & E1000_TXD_STAT_DD)) {
    release(&e1000_lock);//当硬件完成了对当前描述符的填充时，E1000_RXD_STAT_DD 被置为 1，以通知驱动程序软件将其提取。
    return -1;
  }// 检查 desc 指向的描述符中的 status 字段是否包含 E1000_TXD_STAT_DD 这个位，该位表示该
  // buffer 中的数据是否已经被网卡发送完毕。如果没有包含，则说明该 buffer 还在使用中，不能被覆盖。由于我们使用了环形
  // buffer 列表，如果遇到这种情况，则说明我们已经没有空闲的 buffer 可以使用了，因此需要释放锁并返回 -1 表示错误。

  // 如果该下标仍有之前发送完毕但未释放的 mbuf，则释放
  if(tx_mbufs[ind]) {// 这几行的作用是检查 tx_mbufs 数组中 ind 下标对应的元素是否为空指针，
    mbuffree(tx_mbufs[ind]);//该数组是一个存储了与发送 buffer 对应的 mbuf 指针的内存区域。
    tx_mbufs[ind] = 0;//mbuf 是一个网络数据包的结构体，包含了数据内容和长度等信息。如果不为空指针，则说明该下标之前已经发送过一个 mbuf，
  }//并且已经被网卡发送完毕，但是还没有被释放。因此需要调用 mbuffree 函数来释放该 mbuf，并将 tx_mbufs 数组中 ind 下标对应的元素置为零。
```

```
  // 将要发送的 mbuf 的内存地址与长度填写到发送描述符中
  desc->addr = (uint64)m->head;
  desc->length = m->len;
  //将传入函数的参数 m 指向的 mbuf 结构体中的 head 和 len 字段分别赋值给 desc 指向的描述符中的 addr 和 length 字段。
  //head 字段是一个指向 mbuf 中数据内容的指针，len 字段是一个表示 mbuf 中数据长度的整数。这样就将要发送的 mbuf 的内存地址和长度告诉了网卡。

  // 设置参数，EOP 表示该 buffer 含有一个完整的 packet
  // RS 告诉网卡在发送完成后，设置 status 中的 E1000_TXD_STAT_DD 位，表示发送完成。
  desc->cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;
  //cmd 字段是一个表示发送 buffer 的命令参数的整数，E1000_TXD_CMD_EOP表示当前描述符是数据包的最后一个描述符。当数据包跨越多个描述符时，只有最后一个描述符应该设置这个标志。它告诉硬件，这是数据包的结束，可以进行传输。
  //E1000_TXD_CMD_RS 是一个常量，表示要求网卡在发送完成后，设置 status 字段中的 E1000_TXD_STAT_DD 位，以便我们可以检查该 buffer 是否已经发送完毕。
//通常，E1000_TXD_CMD_RS 会与 E1000_TXD_CMD_EOP 一起使用，后者表示数据包的最后一个描述符。当数据包跨越多个描述符时，只有最后一个描述符需要同时设置 E1000_TXD_CMD_RS 和 E1000_TXD_CMD_EOP 标志，以指示传输完成并报告状态。

  // 保留新 mbuf 的指针，方便后续再次用到同一下标时释放。
  tx_mbufs[ind] = m;

  // 环形缓冲区内下标增加一。
  regs[E1000_TDT] = (regs[E1000_TDT] + 1) % TX_RING_SIZE;// Transmit Descriptor Tail (TDT) 寄存器


  release(&e1000_lock);
  return 0;
}// 调用 release 函数，传入一个指向 e1000_lock 的指针，该函数会释放该锁，让其他进程可以获取。然后返回 0 表示成功。


static void
e1000_recv(void)
{
  while(1) { // 每次 recv 可能接收多个包

    uint32 ind = (regs[E1000_RDT] + 1) % RX_RING_SIZE;//  ind 表示下一个要被网卡写入数据的接收 buffer 的下标。

    struct rx_desc *desc = &rx_ring[ind];//rx_ring 数组是一个存储了接收 buffer 描述符的内存区域，每个描述符包含了 buffer 的物理地址、长度、状态等信息。
```

```
    // 如果需要接收的包都已经接收完毕，则退出
    if(!(desc->status & E1000_RXD_STAT_DD)) {
      return;
    }//E1000_RXD_STAT_DD 这个位，表示该 buffer 中的数据是否已经被网卡写入完毕。如果没有包含，则说明该 buffer 还没有接收到数据，或者数据还没有完整地写入。
    //由于我们使用了环形 buffer 列表，如果遇到这种情况，则说明我们已经没有新的数据包可以接收了，因此需要退出循环并返回。

    rx_mbufs[ind]->len = desc->length;
    // 这一行的作用是将 desc 指向的描述符中的 length 字段赋值给 rx_mbufs 数组中 ind 下标对应的元素指向的 mbuf 结构体中的 len 字段。rx_mbufs 数组是一个存储了与接收 buffer 对应的 mbuf 指针的内存区域。length 字段是一个表示接收 buffer 中数据长度的整数，len 字段是一个表示 mbuf 中数据长度的整数。这样就将接收到的数据包的长度告诉了 mbuf。

    net_rx(rx_mbufs[ind]); // 传递给上层网络栈。上层负责释放 mbuf

    // 分配并设置新的 mbuf，供给下一次轮到该下标时使用
    rx_mbufs[ind] = mbufalloc(0);
    desc->addr = (uint64)rx_mbufs[ind]->head;//head 字段是一个指向 mbuf 中数据内容的指针，addr 字段是一个表示接收 buffer 的物理地址的整数。这样就将新分配的 mbuf 的内存地址告诉了网卡。
    desc->status = 0;     //最后将 desc 指向的描述符中的 status 字段赋值为 0，表示该 buffer 可以被网卡写入数据。


    regs[E1000_RDT] = ind;
    // 这一行的作用是将 ind 变量赋值给 regs 数组中 E1000_RDT 下标对应的元素。这样就更新了最后一个被网卡写入数据的接收 buffer 的下标。
  }
}
```

对于 `e1000_transmit()` 函数，我们的任务是：

- 获取 `E1000` 期望的下一个传输数据包的 `TX ring` 索引。
- 检查 `TX ring` 是否溢出。如果前一个传输请求尚未完成，返回错误。
- 使用 `mbuffree()` 释放上一个从该描述符传输的 `mbuf` 。
- 填充描述符，将 `mbuf` 的数据指针和长度放入描述符中。
- 更新 `TX ring` 。
- 根据操作的成功与否返回0或−1。

对于 `e1000_recv()` 函数，我们的任务是：

- 获取 `E1000` 期望的下一个接收数据包的 `RX ring` 索引。
- 检查是否有新数据包可用。

- 如果有，更新 `mbuf` 的长度，将 `mbuf` 传递给网络堆栈，并分配一个新的 `mbuf` 以供下次使用。
- 更新 `E1000` 的 `RDT` 寄存器。

按要求完成即可。

测试结果

```
== Test running nettests ==
$ make qemu-gdb
(4.4s)
== Test   nettest: ping ==
  nettest: ping: OK
== Test   nettest: single process ==
  nettest: single process: OK
== Test   nettest: multi-process ==
  nettest: multi-process: OK
== Test   nettest: DNS ==
  nettest: DNS: OK
== Test time ==
time: OK
Score: 100/100
```

Lab: locks

在本实验中，需要通过降低xv6中内存分配器及磁盘块缓存中锁的粒度，以提升这两个程序的并行性。在 `Thread` 实验中我们已经尝试了锁的基本使用，而在本实验中则需要对并发条件下的临界区保护具备更深的理解，以在降低锁粒度的条件下不破坏程序的并发安全。

锁竞争优化一般有几个思路：

- 只在必须共享的时候共享（对应为将资源从 CPU 共享拆分为每个 CPU 独立）
- 必须共享时，尽量减少在关键区中停留的时间（对应“大锁化小锁”，降低锁的粒度）

这两个思路将贯穿实验始终。

Memory allocator

在本部分中，我们需要将内存分配器的锁粒度从全局锁降低为每CPU一个的小粒度锁，并使得各CPU分别拥有并使用它们的空闲列表，使得各CPU的内存分配是相互独立的，以提升并行性。当一个CPU的空闲列表为空时，该CPU需要在其他CPU的空闲列表中试图寻找空闲内存块，这一点为本部分的主要复杂度来源。

kalloc.c

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem[NCPU]; // 为每个 CPU 分配独立的 freelist，并用独立的锁保护它。

char *kmem_lock_names[] = {
  "kmem_cpu_0",
  "kmem_cpu_1",
  "kmem_cpu_2",
  "kmem_cpu_3",
  "kmem_cpu_4",
  "kmem_cpu_5",
  "kmem_cpu_6",
  "kmem_cpu_7",
};


void
kinit()
{
  for(int i=0;i<NCPU;i++) { // 初始化所有锁
    initlock(&kmem[i].lock, kmem_lock_names[i]);
  }
  freerange(end, (void*)PHYSTOP);
}
```

可以看到，我们为每个 CPU 的核分配独立的 freelist，这样多核 CPU 并发分配物理页就不再会互相排斥了，提高了并行性。

```
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa ≥ PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;
```

```
  push_off();// 禁用中断
//在 xv6 中，关闭中断并不能完全保证原子性，因为还有可能存在多核并发的情况。如果有两个 CPU 同时执行同一段代码，即使它们都关闭了中断，也可能会发生竞争条件。因此，需要使用锁
  int cpu = cpuid();//当前是哪个CPU

  acquire(&kmem[cpu].lock);//上锁
  r->next = kmem[cpu].freelist;// 将当前页面插入到当前 CPU 的空闲页面列表的头部
  kmem[cpu].freelist = r;
  release(&kmem[cpu].lock);

  pop_off();
}
```

释放页表，只需把当前页表插入当前CPU核的空闲页表头即可。

注意仅靠`push_off()`关中断不一定能保证原子性，还需要在下方进行`acquire()`上锁，因为xv6是一个教育用的操作系统，最初设计为单核操作系统。因此，`push_off()`函数只能关闭当前核的中断，不能关闭其他核的中断。

```
void *
kalloc(void)
{
  struct run *r;

  push_off();

  int cpu = cpuid();

  acquire(&kmem[cpu].lock);

  if(!kmem[cpu].freelist) { // 如果当前 CPU 的 free list 为空，没有可用内存页
    int steal_left = 64; //  尝试从其他 CPU 偷取 64 个页面
    for(int i=0;i<NCPU;i++) {
      if(i == cpu) continue; // no self-robbery
      acquire(&kmem[i].lock);
      struct run *rr = kmem[i].freelist;
      while(rr && steal_left) {
        kmem[i].freelist = rr->next;
        rr->next = kmem[cpu].freelist;//将别的CPU的页移到自己后面
        kmem[cpu].freelist = rr;
        rr = kmem[i].freelist;
        steal_left--;
      }
      release(&kmem[i].lock);
      if(steal_left == 0) break; // done stealing
```

```
    }
  }

  r = kmem[cpu].freelist;//取出链表头即可
  if(r)
    kmem[cpu].freelist = r->next;
  release(&kmem[cpu].lock);

  pop_off();

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

注意在分配页表时，在当前内存页不足的时候，从其他的 CPU 核“偷” 64 个页，所以一个 CPU 的 `freelist` 并不是只会被其对应 CPU 访问，还可能在“偷”内存页的时候被其他 CPU 访问，故仍然需要使用单独的锁来保护每个 CPU 的 `freelist` 。但一个 CPU `freelist` 中空闲页不足的情况相对来说是比较稀有的，所以总体性能依然比单独 `kmem` 大锁要快。这里64是随意取的，在现实场景中，最好进行测量后选取合适的数值，尽量使得综合效率更高。

Buffer cache

在本部分中，需要降低磁盘缓冲区的锁粒度，以提升其并行性。

因为不像 `kalloc` 中一个物理页分配后就只归单个进程所管， `bcache` 中的区块缓存是会被多个进程（进一步地，被多个 CPU核心）共享的（由于多个进程可以同时访问同一个区块）。所以 `kmem` 中为每个 CPU 预先分割一部分专属的页的方法在这里是行不通的。

原本磁盘缓冲区通过单一空闲链表以管理缓存区，因此在 `bget` 和 `brelse` 必须获取全局锁以保护缓冲区。因此，我们可以通过哈希表的方式来管理缓冲区，使得一组 `dev` 及 `blockno` 与一个哈希桶单一映射，通过此方法我们仅需在 `bget` 和 `brelse` 中保持哈希桶的锁，既可以保证对应缓冲区保持并发安全。

之前起到过，锁竞争优化一般有两个思路，在这里， `bcache` 属于“必须共享”的情况，所以需要用到第二个思路，降低锁的粒度，用更精细的锁 `scheme` 来降低出现竞争的概率。

buf.h

```
struct buf {
  int valid;   // has data been read from disk?
  int disk;    // does disk "own" buf?
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
   uint lastuse; // *newly added, used to keep track of the least-recently-
used buf
  struct buf *prev; // LRU cache list
  struct buf *next;
  uchar data[BSIZE];
};
```

我们在分配页时使用最近最少使用 `(LRU)` 算法，因此在 `buf` 中加入 `lastuse` 。

```
#define NBUFMAP_BUCKET 13 //几个哈希表
#define BUFMAP_HASH(dev, blockno) ((((dev)<<27)|(blockno))%NBUFMAP_BUCKET) //
计算哈希函数

struct {
  struct buf buf[NBUF];
  struct spinlock eviction_locks[NBUFMAP_BUCKET];//驱逐锁

  // Hash map: dev and blockno to buf
  struct buf bufmap[NBUFMAP_BUCKET];
  struct spinlock bufmap_locks[NBUFMAP_BUCKET];
} bcache;

void
binit(void)
{
  // Initialize bufmap
  for(int i=0;i<NBUFMAP_BUCKET;i++) {
    initlock(&bcache.eviction_locks[i], "bcache_eviction");
    initlock(&bcache.bufmap_locks[i], "bcache_bufmap");
    bcache.bufmap[i].next = 0;
  }

  // Initialize buffers
  for(int i=0;i<NBUF;i++){
    struct buf *b = &bcache.buf[i];
    initsleeplock(&b->lock, "buffer");
    b->lastuse = 0;
```

```
      b->refcnt = 0;
      // put all the buffers into bufmap[0]
      b->next = bcache.bufmap[0].next;
      bcache.bufmap[0].next = b;
    }
  }
```

我们可以建立一个从 `blockno` 到 `buf` 的哈希表，并为每个桶单独加锁。

下面是我们本次实验的核心函数：

```
// 在设备 dev 上查找缓存块。
// 如果未找到，则分配一个缓存块。
// 无论哪种情况，都会返回一个已锁定的缓存块。
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  uint key = BUFMAP_HASH(dev, blockno);

  // printf("dev: %d, blockno: %d, locked: %d\n", dev, blockno,
bcache.bufmap_locks[key].locked);

  acquire(&bcache.bufmap_locks[key]);

  // 这个块是否已经在缓存中?
  for(b = bcache.bufmap[key].next; b; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.bufmap_locks[key]);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // 未在缓存中找到。

  // 为了获得一个适合重用的块，我们需要在所有的桶中搜索一个块，
  // 这意味着需要获取它们的桶锁。
  // 但是在持有一个锁的同时尝试获取每个单独的桶锁是不安全的。
  // 这可能很容易导致循环等待，从而产生死锁。

  release(&bcache.bufmap_locks[key]);
  // 我们需要释放桶锁，以便在迭代所有桶时不会产生循环等待和死锁。
```

```
  // 但是，释放桶锁的副作用是，其他 CPU 可能会同时请求相同的块号，
  // 在最坏的情况下，块号的缓存块可能被多次创建。
  // 因此，在获取 eviction_locks[key] 之后，我们再次检查“块是否已缓存”，
  // 以确保我们不会创建重复的缓存块。


  // 阻止其他线程启动并发的逐出操作（防止相同块号的重复缓冲区）
  acquire(&bcache.eviction_locks[key]);

  // 再次检查，块是否已缓存?
  // 我们在持有 eviction_locks[key] 期间不会发生针对该桶的其他分配，
  // 这意味着此桶的链接列表结构不会改变。
  // 因此，在不持有相应桶锁的情况下，通过 `bcache.bufmap[key]` 进行迭代是可以的，
  // 因为我们持有更强的 eviction_locks[key]。
  for(b = bcache.bufmap[key].next; b; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      acquire(&bcache.bufmap_locks[key]); // 必须，用于 `refcnt++`
      b->refcnt++;
      release(&bcache.bufmap_locks[key]);
      release(&bcache.eviction_locks[key]);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // 仍未缓存。
  // 现在我们只持有逐出锁，没有任何桶锁由我们持有。
  // 因此，现在可以安全地获取任何桶的锁，而不会出现循环等待和死锁。

  // 查找所有桶中最近最不常使用的缓冲区。
  // 完成后，它将持有相应桶的锁。
  struct buf *before_least = 0;
  uint holding_bucket = -1;
  for(int i = 0; i < NBUFMAP_BUCKET; i++){
    // 在获取之前，要么没有持有锁，要么只有左侧的桶的锁。
    // 所以这里永远不会发生循环等待。（免于死锁）
    acquire(&bcache.bufmap_locks[i]);
    int newfound = 0; // 在此桶中找到的新的最不常使用的缓冲区
    for(b = &bcache.bufmap[i]; b->next; b = b->next) {
      if(b->next->refcnt == 0 && (!before_least || b->next->lastuse <
before_least->next->lastuse)) {
        before_least = b;
        newfound = 1;
      }
    }
    if(!newfound) {
```

```
        release(&bcache.bufmap_locks[i]);
      } else {
        if(holding_bucket != -1) release(&bcache.bufmap_locks[holding_bucket]);
        holding_bucket = i;
        // 保持此桶的锁...
      }
    }
    if(!before_least) {
      panic("bget:no buffers");
    }
    b = before_least->next;

    if(holding_bucket != key) {
      // 从原始桶中删除缓冲区
      before_least->next = b->next;
      release(&bcache.bufmap_locks[holding_bucket]);
      // 重新哈希并将其添加到正确的桶中
      acquire(&bcache.bufmap_locks[key]);
      b->next = bcache.bufmap[key].next;
      bcache.bufmap[key].next = b;
    }

    b->dev = dev;
    b->blockno = blockno;
    b->refcnt = 1;
    b->valid = 0;
    release(&bcache.bufmap_locks[key]);
    release(&bcache.eviction_locks[key]);
    acquiresleep(&b->lock);
    return b;
  }
```

这里我们将详细介绍这个函数

`bget(dev, blockno)` 函数通常用于操作系统的文件系统，特别是在文件系统中实现缓存管理时。它的作用是获取一个特定设备 `(dev)` 上的一个特定块 `(blockno)` 的缓存块。这个函数的主要目的是从缓存池中获取一个块，如果该块已经在缓存中，则返回它，否则，它会从磁盘中读取相应的块，并将其放入缓存中，以供后续的读取和写入操作使用。

我们的策略比较复杂，具体而言：

- 根据 `dev` 和 `blockno` 获得哈希值 `key`
- 查看 `key` 对应的哈希表中是否已有对其分配的页，若之前已分配过，则直接返回这个页即可，注意查完之后就释放锁
- 之前没有分配过，则我们需要使用最近最少使用策略为其分配页表，上 `eviction_locks[key]` 锁

- 再次查找 `key` 对应的哈希表中是否已有对其分配的页，若之前已分配过，则直接返回这个页即可
- 若没有，则枚举每个哈希表，访问第 `i` 个表时获取锁 `bufmap_locks[i]`
- 若在本哈希表中找到更优的最近最少使用的页，则本页不释放锁，否则释放本页锁
- 枚举完成后，若没有找到合法页，则报错，否则这个页就是目标页
- 将这个页从之前的哈希表中移出，并解锁
- 获得 `key` 对应的哈希表的锁，并将那个页移入表中
- 释放所有锁，并返回答案

我们的策略相当复杂，是为了应对各种各样的锁问题：

Q:为什么在一开始查找到这个页不存在后就要释放 `key` 哈希表的锁

A：防止出现死锁，具体而言，如果有两个线程分别占用了 `i` 和 `j` 的锁，那么第一个函数会去第 `j` 个哈希表申请页，第二个函数会去第 `i` 个哈希表申页，这样就形成了死锁。

Q：为什么将目标页从之前的哈希表中移出后才开锁?

A：因为如果不这样，其它线程也可以修改这个页。

Q： `eviction_locks[key]` 的作用是什么?

A：防止同时有两个进程同时调用同一个 `bget(dev,blockno)` （二者的 `dev` 和 `blockno` 的值相等），如果没有 `eviction_locks[key]` 的话这能会出现二者都发现没有这个页然后都去申请页表，这样就申请了两个页表，这是我们不希望看到的，因此要加入这个驱逐锁。

事实上，引入 `eviction_locks[key]` 相当于把乐观锁变成了悲观锁。乐观锁每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据。悲观锁认为对于同一个数据的并发操作一定是会发生修改的，采取加锁的形式，悲观地认为，不加锁的并发操作一定会出问题。乐观锁适用于读多写少且对错误的容忍度较高的场景，例如博客文章、社交媒体帖子等。悲观锁适用于写多且对数据的准确性要求较高的场景，例如银行转账、库存管理等。

```
// Release a locked buffer.
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  uint key = BUFMAP_HASH(b->dev, b->blockno);

  acquire(&bcache.bufmap_locks[key]);
  b->refcnt--;
```

```
  if (b->refcnt == 0) {
    b->lastuse = ticks;//当前系统的时间戳
  }//当引用计数 b->refcnt 仍为正数时，没有必要更新 lastuse 字段，因为最近最少使用只看空页
  release(&bcache.bufmap_locks[key]);
}

void
bpin(struct buf *b) {
  uint key = BUFMAP_HASH(b->dev, b->blockno);

  acquire(&bcache.bufmap_locks[key]);
  b->refcnt++;//非原子操作，需要上锁
  release(&bcache.bufmap_locks[key]);
}

void
bunpin(struct buf *b) {
  uint key = BUFMAP_HASH(b->dev, b->blockno);

  acquire(&bcache.bufmap_locks[key]);
  b->refcnt--;
  release(&bcache.bufmap_locks[key]);
}
```

剩下的代码就比较简单了，直接修改引用计数即可，不过记得更新 `lastuse` 用于最近最少使用的判断

需要注意的是 `b->refcnt++;` 不是原子操作，包含取数，增加，赋值三个部分，所以也需要上锁。

测试结果

```
== Test running kalloctest ==
$ make qemu-gdb
(80.4s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (8.7s)
== Test running bcachetest ==
$ make qemu-gdb
```

```
(8.8s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (118.0s)
== Test time ==
time: OK
Score: 70/70
```

Lab: file system

本实验的实验要求是为 xv6 的文件系统添加大文件以及符号链接支持。

Large files

xv6 文件系统中的每一个 `inode` 结构体中，采用了混合索引的方式记录数据的所在具体盘块号。每个文件所占用的前 12 个盘块的盘块号是直接记录在 `inode` 中的（每个盘块 1024 字节），所以对于任何文件的前 12 KB 数据，都可以通过访问 `inode` 直接得到盘块号。这一部分称为直接记录盘块。

对于大于12 个盘块的文件，大于 12 个盘块的部分，会分配一个额外的一级索引表，用于存储这部分数据的所在盘块号。由于一级索引表[]可以包含 `BSIZE(1024) / 4 = 256` 个盘块号，加上 inode 中的 12 个盘块号，一个文件最多可以使用 `12+256 = 268` 个盘块，也就是 268KB。

本 lab 的目标是通过为混合索引机制添加二级索引页，具体而言就是将 NDIRECT 直接索引的盘块号减少 1，腾出 inode 中的空间来存储二级索引的索引表盘块号。

file.h

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
```

```
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+2]; // NDIRECT+1 → NDIRECT+2
};
```

inode是文件系统用来存储文件属性的一种数据结构，它包含了文件的大小，权限，所有者，创建时间，修改时间等信息。

fs.h

```
#define NDIRECT 11 // 12 → 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT * NINDIRECT)
//MAXFILE的值计算为直接块的数量 NDIRECT，加上单间接块的数量 NINDIRECT，再加上双间接块的数量 NINDIRECT * NINDIRECT。
//这个值表示了一个文件在xv6文件系统中可以拥有的最大数据块数量。

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses (NDIRECT+1 → NDIRECT+2) 增加二级索引
};
```

不同于内存中的inode，dinode是在硬盘中的，同样也需要修改。

fs.c

```
// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
static uint
bmap(struct inode *ip, uint bn)//文件的逻辑块号为 bn
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){
```

```
    if((addr = ip->addrs[bn]) == 0)//如果对应直接块的磁盘块地址 addr 为0，说明该块还未分配磁盘空间，于是通过 balloc 函数进行分配。
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){ // singly-indirect
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)//如果对应单间接块的磁盘块地址 addr 为0，说明该块还未分配磁盘空间，于是通过 balloc 函数进行分配。
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){//如果 a[bn] 的块号为0，说明需要分配一个新的数据块并将其块号存储在单间接块中。
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
  bn -= NINDIRECT;

  if(bn < NINDIRECT * NINDIRECT) { // doubly-indirect双间接块
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT+1]) == 0)//二级
      ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);//通过 bread 函数读取双间接块的内容
    a = (uint*)bp->data;
    if((addr = a[bn/NINDIRECT]) == 0){//除数确定上级
      a[bn/NINDIRECT] = addr = balloc(ip->dev);
      log_write(bp);//log_write(bp) 是一个用于写入日志的函数调用。在文件系统中，日志（log）用于记录正在执行的事务，
    }//以便在系统崩溃或断电等情况下，能够回滚或重做这些事务，从而保持文件系统的一致性和完整性。
    brelse(bp);
    bn %= NINDIRECT;//余数确定下级
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
```

```
    }

    panic("bmap: out of range");
  }
```

`bmap` 负责将逻辑文件块号(logical block number，即文件中的块号）映射到物理磁盘块号（disk block number，即磁盘上的块号）。在 `bmap` 中，将块映射分三部分执行：

对于块号小于 `NDIRECT + NINDIRECT` 的块，查找方式与之前相同。

如当前块号大于间接块和直接块的表示范围，则在双重间接块中进行查找。首先根据 `bn` 减去前两部分中的 `NDIRECT+NINDIRECT`，表示其在双间接块中的相对块号，进入这个块后需要确定内部的两个块号，由于一级索引表能存储 `256` 个盘块，因此我们将 `bn/256` 作为一级索引号，再将 `bn%256` 作为内部的直接索引号。这样通过两次映射，即可从双重间接块中提取子间接块，再从子间接块中提取所需的磁盘快。

需要注意的是，在子间接块或所需磁盘块未被分配时，我们要对其进行分配。

```
// Truncate inode (discard contents).
// Caller must hold ip->lock.
void
itrunc(struct inode *ip)
{//遍历并释放文件的块
  int i, j;
  struct buf *bp;
  uint *a;

  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

  if(ip->addrs[NDIRECT]){
    bp = bread(ip->dev, ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT]);
    ip->addrs[NDIRECT] = 0;
  }
```

```
    // 如果存在一个双间接块
    if(ip->addrs[NDIRECT+1]){
      // 读取双间接块
      bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
      a = (uint*)bp->data;
      // 遍历双间接块中的块地址
      for(j = 0; j < NINDIRECT; j++){
        if(a[j]) {
          // 读取间接块
          struct buf *bp2 = bread(ip->dev, a[j]);
          uint *a2 = (uint*)bp2->data;
          // 遍历间接块中的块地址
          for(int k = 0; k < NINDIRECT; k++){
            if(a2[k])
              // 释放间接块中的块
              bfree(ip->dev, a2[k]);
          }
          brelse(bp2); // 释放缓冲区
          // 释放间接块本身
          bfree(ip->dev, a[j]);
        }
      }
      brelse(bp); // 释放缓冲区 (buffer)
      // 释放双间接块本身
      bfree(ip->dev, ip->addrs[NDIRECT+1]);
      ip->addrs[NDIRECT+1] = 0; // 将inode中的块地址清零
    }

    ip->size = 0;
    iupdate(ip);
  }
```

在 `btrunc` 中，我们需要搜索双重间接块中的所有子间接块，以及子间接块中的所有直接块，并释放其中被分配的块，其实现逻辑比较简单，遍历直接块以及间接块内指向的盘块然后清空即可。

Symbolic links

在本部分中，需要为xv6提供符号链接功能。在UNIX操作系统中，文件链接可以被分为硬链接和符号链接（软连接）两种。其中硬链接即为原xv6中的 `sys_link` 功能，其为链接得到的新文件分配与被链接的旧文件相同的 `inode` ，因 `inode` 中存放了构成文件的各磁盘块的块号，因此硬链接得到了原有文件的物理副本，且仅当硬链接文件和原文件均被删除时，构成文件的磁盘块

才会被释放；符号连接则仅在新文件中存放了被链接文件的路径名，其链接方式更加轻量且可跨文件系统、跨主机链接，但当被链接文件被删除时符号链接文件也随之失效。

需要注意的是，本实验要求实现了系统调用 `symlink` ，其中 `user/usys.pl, user/user.h,kernel/syscall.h,kernel/syscall.c` 的修改在上述实验中已经提及，这里不再赘述。

stat.h

```
#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device
#define T_SYMLINK 4   // Symbolic link

struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

首先定义 `T_SYMLINK` 类型的文件，即符号链接类型的文件。

fcntl.h

```
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_TRUNC   0x400
#define O_NOFOLLOW 0x800//用于指示打开文件时不要跟随符号链接。
```

需要注意的是如果 `O_NOFOLLOW` 为是 0，表示不设置 `O_NOFOLLOW` ，允许跟随符号链接。如果结果不等于 0，表示设置了 `O_NOFOLLOW` ，不允许跟随符号链接，应该直接处理符号链接本身而不是跟随它。

sysfile.c

```
uint64
sys_symlink(void)//系统调用，用于创建符号链接文件
{
  struct inode *ip;
```

```
  char target[MAXPATH], path[MAXPATH];
  // 从用户程序获取符号链接的目标路径和链接路径
  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
    return -1;
  begin_op(); // 尝试获取文件系统锁，以确保该操作是原子的
  ip = create(path, T_SYMLINK, 0, 0);  // 在path的位置创建符号链接文件，其inode是ip
  if(ip == 0){
    end_op();
    return -1;
  }
  if(writei(ip, 0, (uint64)target, 0, strlen(target)) < 0) {//将target写入ip的头
    end_op();
    return -1;
  }
  // 在 create 函数中，会调用 ilock 来锁定这新创建的 inode
  iunlockput(ip); // 解锁 inode 并释放对 inode 的引用(引用计数-1)
  end_op();
  return 0;
}
```

`sys_symlink` 是用于创建符号链接文件的系统调用。

注意到需要使用 `begin_op` 和 `end_op` 包围对文件系统进行操作的代码段，防止冲突。

以及此外在调用 `create` 时会自动获取 `inode` 锁,因此在最后需要 `iunlockput` 解锁。

```
uint64
sys_open(void)
{
  char path[MAXPATH];
  int fd, omode;
  struct file *f;
  struct inode *ip;
  int n;

  if((n = argstr(0, path, MAXPATH)) < 0 || argint(1, &omode) < 0)
    return -1;

  begin_op();

  if(omode & O_CREATE){
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
```

```
  } else {//递归地跟踪符号链接，以找到最终的目标文件或目录，并进行一些检查，如检查是否存在循环引用以及目标是否是目录
    int symlink_depth = 0; // 初始化一个变量用于跟踪符号链接的层数
    while(1) { // 无限循环，用于递归地跟踪符号链接
      if((ip = namei(path)) == 0){ // 使用 namei 函数查找文件路径对应的 inode，如果找不到，返回0
        end_op();
        return -1;
      }
      ilock(ip); // 锁住 inode 以确保其他进程不会同时修改它
      if(ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0)//允许打开符号链接并跟随它，以打开符号链接指向的文件。
       {
        if(++symlink_depth > 10) { // 检查符号链接的层数是否超过10，以避免可能的循环
          // 层数太多，可能是循环引用（一个符号链接引用了自身或者引用了其他文件，最终形成一个环路。这种情况下，文件系统或程序可能会陷入无限循环）
          iunlockput(ip);
          end_op();
          return -1;
        }
        if(readi(ip, 0, (uint64)path, 0, MAXPATH) < 0) // 读取符号链接的目标路径
        {//从过程上看是从文件 ip 的开头读取数据，并将这些数据复制到 path 所指向的内存位置中，最多不超过 MAXPATH 字节
          iunlockput(ip);
          end_op();
          return -1;
        }
        iunlockput(ip); // 解锁并释放 inode
      } else {
        break; // 如果不是符号链接，退出循环
      }
    }
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip); // 如果 inode 类型是目录且打开模式不是只读，解锁并释放 inode
      end_op(); // 结束操作
      return -1; // 返回错误代码表示失败
    }
  }
  ..........
}
```

在寻找或创建 `inode` 后，分配 `file` 和文件描述符前，需要根据 `O_NOFOLLOW` 判断是否对符号链接进行跟踪。在这里，首先对当前 `inode` 类型进行判断，如未指定 `O_NOFOLLOW` 、当前文件类型为 `T_SYMLINK` 、且跟随次数小于10，则对该符号链接进行跟随。跟随方式即为从符号链接中读取其指向的路径名，并将当前 `inode` 指定为路径名所对应的文件即可。如无法找到该

文件或跟随次数超过10，则认为可能是出现了循环引用的情况，返回 -1。

测试结果

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (168.0s)
== Test running symlinktest ==
$ make qemu-gdb
(0.9s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (265.8s)
== Test time ==
time: OK
Score: 100/100
```

Lab: mmap

实验目的：支持将文件映射到一片用户虚拟内存区域内，并且支持将对其的修改写回磁盘。

memlayout.h

```
#define TRAPFRAME (TRAMPOLINE - PGSIZE)
// MMAP 所能使用的最后一个页+1
#define MMAPEND TRAPFRAME
```

xv6 对用户的地址空间的分配中，heap 的范围一直从 stack 到 trapframe，其中，stack从低地址向高地值生长，frapframe处在高地址。为了尽量使得 map 的文件使用的地址空间不要和进程所使用的地址空间产生冲突，我们选择将 mmap 映射进来的文件 map 到尽可能高的位置，也就是刚好在 trapframe 下面。并且若有多个 mmap 的文件，则向下生长。因此我们将首mmap首地址就定义为 `TRAPFRAME` 。

proc.h

```
struct vma {
  int valid;//是否有效
  uint64 vastart;//起始地址
  uint64 sz;//大小
  struct file *f;//关联的文件
  int prot;//权限
  int flags;//指定一些特殊属性，例如共享映射或私有映射
  uint64 offset;//指定在文件中从哪个位置开始映射
};

#define NVMA 16

// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  struct vma vmas[NVMA];       // virtual memory areas
};
```

定义 vma 结构体，其中包含了 mmap 映射的内存区域的各种必要信息，并在proc中定义vma数组，表示本进程最多同时允许 `NVMA` 个mmap 映射。

sysfile.c

```
#include "memlayout.h"
uint64
sys_mmap(void)
{
  uint64 addr, sz, offset;
  int prot, flags, fd; struct file *f;

  if(argaddr(0, &addr) < 0 || argaddr(1, &sz) < 0 || argint(2, &prot) < 0
    || argint(3, &flags) < 0 || argfd(4, &fd, &f) < 0 || argaddr(5, &offset) <
0 || sz == 0)
    return -1;

  if((!f->readable && (prot & (PROT_READ)))//不可读但请求了读权限
     || (!f->writable && (prot & PROT_WRITE) && !(flags & MAP_PRIVATE)))//请求
了写权限，但文件不可写，并且不是私有映射
    return -1;//私有映射：将同一个文件映射到它们各自的地址空间，但每个进程对这个映射的修改
不会影响其他进程的映射

  sz = PGROUNDUP(sz);
  // 将请求的大小向上对齐为页的整数倍（sz对4096上取整）

  struct proc *p = myproc();
  struct vma *v = 0;
  uint64 vaend = MMAPEND;

  // 我们的实现将文件从高地址到低地址映射到trapframe下方。
  for(int i=0;i<NVMA;i++) {
    struct vma *vv = &p->vmas[i];
    if(vv->valid == 0) {//是否被使用
      if(v == 0) {// found free vma;
        v = &p->vmas[i];
        v->valid = 1;
      }
    } else if(vv->vastart < vaend) {
      vaend = PGROUNDDOWN(vv->vastart);//找最下面的
    }
  }
  // 查找一个未使用的 VMA（虚拟内存区域）结构，然后计算文件映射的位置

  if(v == 0){
    panic("mmap: no free vma");
  }
  // 如果找不到可用的 VMA 结构，则触发 panic（内核恐慌）
```

```
  v->vastart = vaend - sz;
  v->sz = sz;
  v->prot = prot;
  v->flags = flags;
  v->f = f; // assume f->type == FD_INODE
  v->offset = offset;
  // 设置 VMA 结构的各个字段，包括虚拟地址起始地址、大小、保护标志、标志、文件指针和偏移量

  filedup(v->f);
  // 增加文件引用计数，以确保文件不会在关闭时被删除

  return v->vastart;
}
```

这里来到了我们的第一个系统调用 `sys_mmap`，它在进程的 16 个 vma 槽中，找到可用的空槽，并且顺便计算所有 vma 中使用到的最低的虚拟地址，然后将当前文件映射到该最低地址下面的位置。注意这里的仅仅是虚拟地址，空间比较富余，因此我们并不需要垃圾回收机制。而因为我们使用懒分配方式，所以立即产生一个新的问题，就是何时实分配，因此我们暂时把目光转到 `trap.c` 中。

trap.c

```
void
usertrap(void)
{
  ..........

  if(r_scause() == 8){
    ..........
    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok  哪个设备中断正在处理将存储在 which_dev 变量中
  } else {//既不是系统调用，也没有设备中断
    uint64 va = r_stval();//13在尝试访问尚未分配物理内存的虚拟地址时
    if((r_scause() == 13 || r_scause() == 15)){ // 15 超过可访问物理内存的地址
      if(!vmatrylazytouch(va)) {//懒分配
        goto unexpected_scause;
      }
    } else {
    unexpected_scause:
    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;
      }
    }

    ..........
  }
```

可以发现，当我们尝试访问尚未分配物理内存的虚拟地址或超过可访问物理内存的地址，可能就是需要实分配了，对于实分配的具体过程，我们将目光转回 `sysfile.c` 中。

sysfile.c

```
// 通过虚拟地址查找包含该虚拟地址的虚拟内存区域（VMA）结构
struct vma *findvma(struct proc *p, uint64 va) {
  for(int i=0;i<NVMA;i++) {
    struct vma *vv = &p->vmas[i];
    if(vv->valid == 1 && va >= vv->vastart && va < vv->vastart + vv->sz) {
      return vv; // 如果VMA有效（已分配），并且虚拟地址va位于该VMA范围内
    }
  }
  return 0; // 如果未找到匹配的VMA，返回0表示未找到
}

// 检查是否需要懒惰分配（lazy allocation）并尝试触摸（touch）虚拟地址的页面
int vmatrylazytouch(uint64 va) {
  struct proc *p = myproc();
  struct vma *v = findvma(p, va);
  if(v == 0) {
    return 0;
  }

  // 分配物理页面
  void *pa = kalloc();
  if(pa == 0) {
    panic("vmalazytouch: kalloc");
  }
  memset(pa, 0, PGSIZE);

  // 从磁盘读取数据
  begin_op();
  ilock(v->f->ip); // 锁
  readi(v->f->ip, 0, (uint64)pa, v->offset + PGROUNDDOWN(va - v->vastart),
PGSIZE);
  iunlock(v->f->ip); // 解锁
```

```
  end_op();

  // 设置适当的权限，然后进行页面映射
  int perm = PTE_U;
  if(v->prot & PROT_READ)
    perm |= PTE_R;
  if(v->prot & PROT_WRITE)
    perm |= PTE_W;
  if(v->prot & PROT_EXEC)
    perm |= PTE_X;

  if(mappages(p->pagetable, va, PGSIZE, (uint64)pa, PTE_R | PTE_W | PTE_U) <
0) {
    panic("vmalazytouch: mappages");
  }

  return 1;
}
```

`findvma` 找到虚拟地址所在的区域比较容易实现，只需要遍历vmas数组查找即可。之后只需进行物理页分配，并从磁盘中写入数据，最后给予适当地权限即可。

```
uint64
sys_munmap(void)//解除内存映射
{
  uint64 addr, sz;

  if(argaddr(0, &addr) < 0 || argaddr(1, &sz) < 0 || sz == 0)
    return -1;

  struct proc *p = myproc();

  struct vma *v = findvma(p, addr);
  if(v == 0) {
    return -1;
  }

  if(addr > v->vastart && addr + sz < v->vastart + v->sz) {
    // trying to "dig a hole" inside the memory range. 不让有中空
    return -1;
  }

  uint64 addr_aligned = addr;
  if(addr > v->vastart) {//地址对齐到页面边界
    addr_aligned = PGROUNDUP(addr);
```

```
  }

  int nunmap = sz - (addr_aligned-addr); // nbytes to unmap
  if(nunmap < 0)
    nunmap = 0;

  vmaunmap(p->pagetable, addr_aligned, nunmap, v); // 取消映射

  if(addr <= v->vastart && addr + sz > v->vastart) { // 如果开头没了，找新的起点
    v->offset += addr + sz - v->vastart;
    v->vastart = addr + sz;
  }
  v->sz -= sz;

  if(v->sz <= 0) {
    fileclose(v->f);// 减小引用计数
    v->valid = 0;
  }

  return 0;
}
```

最后还需要实现一个释放映射的系统调用。

这里首先通过传入的地址找到对应的 vma 结构体，然后检测了一下在 vma 区域中间“挖洞”释放的错误情况，计算出应该开始释放的内存地址以及应该释放的内存字节数量。

计算出来释放内存页的开始地址以及释放的个数后，调用我们在 `vm.c` 中实现的 vmaunmap 方法对物理内存页进行释放，并在需要的时候将数据写回磁盘。之后对 v 作相应的修改，并关闭对文件的引用即可。下面我们将注意力放到 vmaunmap 函数上。

riscv.h

```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_G (1L << 5) // global mapping
#define PTE_A (1L << 6) // accessed
#define PTE_D (1L << 7) // dirty
```

在讨论 vmaunmap 之前，我们先定义好xv6中没有定义的标志位。

vm.c

```
// 从虚拟地址 va 开始，移除 n 字节（而不是页面）的 VMA 映射。va 必须是页对齐的。
// 这些映射不一定存在。
// 如果需要，还会释放物理内存并将 VMA 数据写回磁盘。
void
vmaunmap(pagetable_t pagetable, uint64 va, uint64 nbytes, struct vma *v)
{
  uint64 a;
  pte_t *pte;

  // 借用了“uvmunmap”的代码
  for(a = va; a < va + nbytes; a += PGSIZE){ //从va开始，按页大小循环解除映射
    if((pte = walk(pagetable, a, 0)) == 0) //如果找不到页表项，报错
      panic("sys_munmap: walk");
    if(PTE_FLAGS(*pte) == PTE_V)//如果页表项不是叶子节点，报错
      panic("sys_munmap: not a leaf");
    if(*pte & PTE_V){ //如果页表项有效
      uint64 pa = PTE2PA(*pte); //获取物理地址
      if((*pte & PTE_D) && (v->flags & MAP_SHARED)) { //如果页表项是脏的，并且VMA
是共享的，需要写回到磁盘
        begin_op(); //开始文件操作
        ilock(v->f->ip); //锁定文件对应的inode
        uint64 aoff = a - v->vastart; //计算相对于VMA起始地址的偏移量
        if(aoff < 0) { //如果第一页不是完整的4k页面，从偏移量开始写入文件
          writei(v->f->ip, 0, pa + (-aoff), v->offset, PGSIZE + aoff);
        } else if(aoff + PGSIZE > v->sz){  //如果最后一页不是完整的4k页面，只写入VMA
大小的数据到文件
          writei(v->f->ip, 0, pa, v->offset + aoff, v->sz - aoff);
        } else { //如果是完整的4k页面，直接写入文件
          writei(v->f->ip, 0, pa, v->offset + aoff, PGSIZE);
        }
        iunlock(v->f->ip); //解锁inode
        end_op(); //结束文件操作
      }
      kfree((void*)pa); //释放物理内存
      *pte = 0; //清空页表项
    }
  }
}
```

仿照 uvmunmap 的实现，查找范围内的每一个页，检测其脏位 `dirty bit (D)` 是否被设置，如果被设置，则代表该页被修改过，需要将其写回磁盘。

需要注意的是不是每一个页都需要完整的写回，这里需要处理开头页不完整、结尾页不完整以及中间完整页的情况。

proc.c

```
static struct proc*
allocproc(void)
{
  ..........

  // Clear VMAs
  for(int i=0;i<NVMA;i++) {
    p->vmas[i].valid = 0;
  }

  return p;
}

// free a proc structure and the data hanging from it,
// including user pages.
// p->lock must be held.
static void
freeproc(struct proc *p)
{
  ..........
  for(int i = 0; i < NVMA; i++) {
    struct vma *v = &p->vmas[i];
    vmaunmap(p->pagetable, v->vastart, v->sz, v);
  }
  ..........
}

// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  ..........

  // copy vmas created by mmap.
  // actual memory page as well as pte will not be copied over.
  for(i = 0; i < NVMA; i++) {
    struct vma *v = &p->vmas[i];
    if(v->valid) {
      np->vmas[i] = *v;
```

```
      filedup(v->f);
    }
  }

  ..........
}
```

我们最后还需要在 `proc.c` 中添加处理进程 vma 的各部分代码。

1. 初始化进程的时候，清空vma 槽
2. 释放进程时，将所有 vma 都释放，并写回磁盘
3. fork 时，拷贝父进程的所有 vma，并增加引用计数

测试结果

```
== Test running mmaptest ==
$ make qemu-gdb
(3.9s)
== Test   mmaptest: mmap f ==
  mmaptest: mmap f: OK
== Test   mmaptest: mmap private ==
  mmaptest: mmap private: OK
== Test   mmaptest: mmap read-only ==
  mmaptest: mmap read-only: OK
== Test   mmaptest: mmap read/write ==
  mmaptest: mmap read/write: OK
== Test   mmaptest: mmap dirty ==
  mmaptest: mmap dirty: OK
== Test   mmaptest: not-mapped unmap ==
  mmaptest: not-mapped unmap: OK
== Test   mmaptest: two files ==
  mmaptest: two files: OK
== Test   mmaptest: fork_test ==
  mmaptest: fork_test: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (288.1s)
== Test time ==
time: OK
Score: 140/140
```

结语

通过这次实验，不仅使我对操作系统的知识有了更深一步的理解，明晰了一些之前一些含糊不清的问题，如系统调用的实现、页表的管理、内存与页表的分配，多线程的机制、文件管理、锁与冲突等等；还让我操作系统的代码架构有了一定的认知。除此之外还增加了我对git的熟练度。从总体上来说是收获非常大的一次实验。