操作系统课程设计项目说明文档

MIT 操作系统实验 2022

|  |  |
| --- | --- |
| 作者姓名： | 吴浩泽 |
| 学 号： | 1953729 |
| 指导教师： | 王冬青 |
| 学院专业： | 软件学院 软件工程 |

同济大学

Tongji University

目录

[实验一 Xv6 and Unix utilities 5](#_bookmark0)

* 1. [实验目的 5](#_bookmark1)
  2. [实验内容 5](#_bookmark2)
  3. [实验步骤 5](#_bookmark3)
     1. [运行 xv6 5](#_bookmark4)
     2. [实现 sleep 6](#_bookmark5)
     3. [实现 pingpong 7](#_bookmark6)
     4. [实现 primes 8](#_bookmark7)
     5. [实现 find 10](#_bookmark8)
     6. [实现 xargs 12](#_bookmark9)
     7. [综合测试 14](#_bookmark10)
  4. [实验小结 15](#_bookmark11)

[实验二 System calls 16](#_bookmark12)

* 1. [实验目的 16](#_bookmark13)
  2. [实验内容 16](#_bookmark14)
  3. [实验步骤 16](#_bookmark15)
     1. [实现系统调用跟踪 16](#_bookmark16)
     2. [实现 sysinfo 19](#_bookmark17)
     3. [综合测试 21](#_bookmark18)
  4. [实验小结 21](#_bookmark19)

[实验三 Page Tables 22](#_bookmark20)

* 1. [实验目的 22](#_bookmark21)
  2. [实验内容 22](#_bookmark22)
  3. [实验步骤 22](#_bookmark23)
     1. [打印页表结构 22](#_bookmark24)
     2. [每个进程一个内核页表 23](#_bookmark25)
     3. [简化 copyin/copyinstr 28](#_bookmark26)
     4. [综合测试 30](#_bookmark27)
  4. [实验小结 30](#_bookmark28)

[实验四 traps 31](#_bookmark29)

* 1. [实验目的 31](#_bookmark30)
  2. [实验内容 31](#_bookmark31)
  3. [实验步骤 31](#_bookmark32)
     1. [RISC-V 程序集 31](#_bookmark33)
     2. [实现 Backtrace 33](#_bookmark34)
     3. [实现 alarm 34](#_bookmark35)
     4. [综合测试 38](#_bookmark36)
  4. [实验小结 38](#_bookmark37)

[实验五 Lazy Allocation 39](#_bookmark38)

* 1. [实验目的 39](#_bookmark39)
  2. [实验内容 39](#_bookmark40)
  3. [实验步骤 39](#_bookmark41)
     1. [删除 sbrk 中的分配 39](#_bookmark42)
     2. [实现 lazy allocation 40](#_bookmark43)
     3. [通过 lazytests 和 usertests 42](#_bookmark44)
  4. [实验小结 43](#_bookmark45)

[实验六 Copy on write 44](#_bookmark46)

* 1. [实验目的 44](#_bookmark47)
  2. [实验内容 44](#_bookmark48)
  3. [实验步骤 44](#_bookmark49)
     1. [理解 fork() 系统调用 44](#_bookmark50)
     2. [阅读 xv6 源码 45](#_bookmark51)
     3. [编码实现 COW 46](#_bookmark52)
  4. [实验小结 50](#_bookmark53)

[实验七 Multithreading 51](#_bookmark54)

* 1. [实验目的 51](#_bookmark55)
  2. [实验内容 51](#_bookmark56)
  3. [实验步骤 51](#_bookmark57)
     1. [实现一个用户级进程的创建和切换 51](#_bookmark58)
     2. [使用 UNIX pthread 线程库实现一个线程安全的哈希表 54](#_bookmark59)
     3. [实现 barrier 函数 55](#_bookmark60)
     4. [测试成绩 56](#_bookmark61)
  4. [实验小结 57](#_bookmark62)

[实验八 Locks 58](#_bookmark63)

* 1. [实验目的 58](#_bookmark64)
  2. [实验内容 58](#_bookmark65)
  3. [实验步骤 58](#_bookmark66)
     1. [重新设计系统管理内存的方式 58](#_bookmark67)
     2. [修改系统的 IO 缓冲区 61](#_bookmark68)
     3. [测试成绩 63](#_bookmark69)
  4. [实验小结 63](#_bookmark70)

[实验九 File System 64](#_bookmark71)

* 1. [实验目的 64](#_bookmark72)
  2. [实验内容 64](#_bookmark73)
  3. [实验步骤 64](#_bookmark74)
     1. [修改文件系统使其支持更大的文件存储 64](#_bookmark75)
     2. [增加一个通过符号链接的系统调用 68](#_bookmark76)
     3. [测试成绩 71](#_bookmark77)
  4. [实验小结 71](#_bookmark78)

[实验十 Mmap 73](#_bookmark79)

* 1. [实验目的 73](#_bookmark80)
  2. [实验内容 73](#_bookmark81)
  3. [实验步骤 73](#_bookmark82)
     1. [理解虚拟地址与 mmap 73](#_bookmark83)
     2. [实现 mmap 系统调用 74](#_bookmark84)
     3. [实现 munmap 系统调用 79](#_bookmark85)
     4. [修改其他相关函数 80](#_bookmark86)

[10.3.4 运行测试 81](#_bookmark87)

* 1. [实验小结 82](#_bookmark88)

[实验总结 83](#_bookmark89)

# 实验一 Xv6 and Unix utilities

## 实验目的

1. 初步掌握 xv6 这一操作系统内核和用户。
2. 了解系统调用。
3. 了解父子进程，以及 fork()函数创建父子进程的方法。
4. 掌握素数筛，以及利用管道编写素数筛的方法。
5. 完成 find 和 xargs 程序的编写。

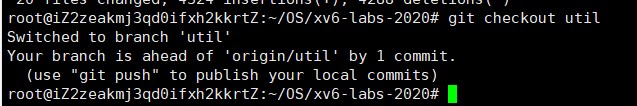
## 实验内容

1. 运行 xv6。
2. 为 xv6 实现 UNIX 程序 sleep：sleep 应该暂停一段用户所指定的时间。
3. 编写一个程序，使用 UNIX 系统调用在两个进程之间通过一对管道“乒乓” 一个字节，每个管道一个。
4. 使用管道编写并发版本的主筛。
5. 编写一个简单版本的 UNIX 查找程序：查找具有特定名称的目录树中的所有文件。
6. 编写一个简单版本的 UNIX xargs 程序：从标准输入中读取行并为每一行运行一个命令，将该行作为命令的参数提供。

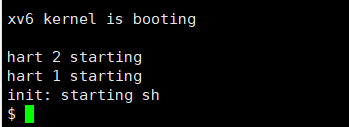
## 实验步骤

### 运行 xv6

1. 切换到 util 分支：



1. 利用 make qemu 指令运行 xv6：



1. 使用 ls 命令查看文件：



### 实现 sleep

1. 查看 sleep 系统调用函数：

**uint64 sys\_sleep**(void)

{

int n;

**uint** ticks0;

if(**argint**(0, &n) < 0) return -1;

**acquire**(&tickslock); ticks0 = ticks; while(ticks - ticks0 < n){

if(**myproc**()->killed){ **release**(&tickslock); return -1;

}

**sleep**(&ticks, &tickslock);

}

**release**(&tickslock); return 0;

}

1. 根据要求，.0.需要从所给的参数中获取下标为 1 的，然后将其转为整数类型。接下来，调用系统函数即可：

int

**main**(int argc, char \*argv[])

{

if(argc !=2){

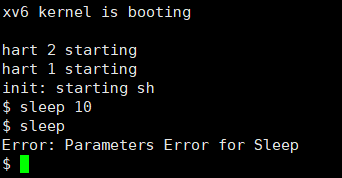
**fprintf**(2, "Error: Parameters Error for Sleep\n");

**exit**(1);

}

**sleep**(**atoi**(argv[1])); **exit**(0);

}

1. 最后，在 Makefile 文件中加上该方法即可。
2. 完成上述步骤后，我便可以在 xv6 中实现 sleep 方法：

### 实现 pingpong

1. 根据题目所述，我应当先创建两个管道：

int fd1[2]; int fd2[2]; **pipe**(fd1); **pipe**(fd2);

1. 创建缓冲区，用于存储所需要传送的消息。同时，我借助 fork()创建子进程。需要注意到，我可以通过返回值来判断当前是父进程还是子进程：

char buffer[16];

if(**fork**()){

*//parent do this* **write**(fd1[1],"ping",**strlen**("ping")); **read**(fd2[0],buffer,4);

**printf**("%d: received %s\n",**getpid**(),buffer);

}

else{

*//child do this*

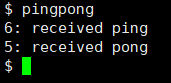
**read**(fd1[0],buffer,4);

**printf**("%d: received %s\n",**getpid**(),buffer);

**write**(fd2[1],"pong",**strlen**("pong"));

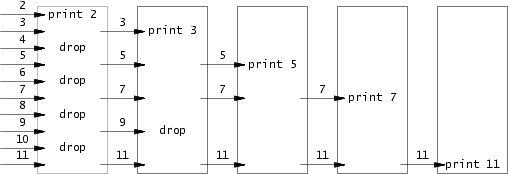
}

1. 在 MakeFile 文件中添加该调用。完成后在 xv6 中测试：



至此，该部分完成。

### 实现 prime

1. 为了利用管道编写并发版本的素数筛，我先了解它的原理，如下图所示：

和上图阐述的原理一样，我只需要借助父子进程的并发性，同时利用上一小问中所学习到的管道通信机制来传输消息即可。

不过需要注意，当文件描述符不再被需要的时候需要及时关闭，否则将会资源不足。

1. 为了实现该功能，我在 main 函数中完成下述功能：

int **main**(int argc, char \*argv[]) { int pd[2];

**pipe**(pd);

if (**fork**()) { **redirect**(0, pd); **sink**();

} else { **redirect**(1, pd); **source**();

}

**exit**(0);

}

1. 同时，各个函数代码如下所示：

void **source**() { int i;

for (i = 2; i < 36; i++) {

**write**(1, &i, sizeof(i));

}

}

void **cull**(int p) { int n;

while (**read**(0, &n, sizeof(n))) { if (n % p != 0) {

**write**(1, &n, sizeof(n));

}

}

}

void **redirect**(int k, int pd[]) {

**close**(k); **dup**(pd[k]);

**close**(pd[0]);

**close**(pd[1]);

}

void **sink**() { int pd[2]; int p;

if (**read**(0, &p, sizeof(p))) { **printf**("prime %d\n", p); **pipe**(pd);

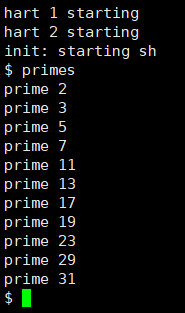
if (**fork**()) { **redirect**(0, pd); **sink**();

} else { **redirect**(1, pd); **cull**(p);

}

}

}

1. 在 MakeFile 中添加 primes 之后，我便可以在 xv6 中运行 primes 命令，结果如下：

### 实现 find

1. 参考 ls 方法，我可以发现 find 实现方法的整体思路应当为深度优先搜索。因此，在 main 函数中我只需要检查参数。如果参数合法，只需要直接调用深度优先搜索函数 findDfs 即可：

int

**main**(int argc, char \*argv[])

{

if(argc !=3){

**fprintf**(2, "Usage: Find Parameters Error...\n");

**exit**(1);

}

*//Find from argv[1]* **findDfs**(argv[1],argv[2]); **exit**(0);

}

1. 针对于 findDfs 函数，它的作用是在每一级中检索是否有满足要求的文件：

void **findDfs**(char \*path,char \*target){ char buf[512],\*p;

int fd;

struct **dirent** de; struct **stat** st;

if((fd = **open**(path, 0)) < 0){

**fprintf**(2, "find: cannot open %s\n", path); return;

}

if(**fstat**(fd, &st) < 0){

**fprintf**(2, "find1: cannot stat %s\n", path);

**close**(fd); return;

}

switch (st.type)

{

case **T\_FILE**:

*//check whether the name is the same*

if(**strcmp**(target,**fmtname**(path))==0){

**printf**("%s\n",path);

}

break; case **T\_DIR**:

if(**strlen**(path) + 1 + **DIRSIZ** + 1 > sizeof buf){

**printf**("find: path too long\n"); break;

}

**strcpy**(buf, path);

p = buf+**strlen**(buf);

\*p++ = '/';

while (**read**(fd, &de, sizeof(de)) == sizeof(de)){

if (de.inum == 0) continue;

**memmove**(p, de.name, **DIRSIZ**); p[**DIRSIZ**] = 0;

if (**stat**(buf, &st) < 0){

**printf**("Here is:%s\n",buf); **printf**("find2: cannot stat %s\n", buf); continue;

}

if (**strlen**(de.name) == 1 && de.name[0] == '.') continue;

if (**strlen**(de.name) == 2 && de.name[0] == '.' && d e.name[1] == '.')

continue;

*//dfs*

**findDfs**(buf,target);

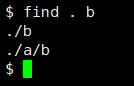
}

break;

}

**close**(fd);

}

1. 完成上述代码后，在 MakeFile 文件中添加 find 方法。之后，便可以在 xv6 中使用 find 方法：

### 实现 xargs

1. 通过阅读本题题目的要求，可以发现需要解决的主要问题是对字符串进行处理。其中，| 之前的结果会在缓冲流中。因此，我需要通过读取缓冲流并且分解字符串（类似于 python 中的 split 函数）。
2. 为了实现等待效果，我利用 wait()方法，代码如下：

int

**main**(int argc, char \*argv[])

{

if(argc<2){

**fprintf**(2,"Error: too less parameters for xargs\n");

**exit**(1);

}

int i;

int pNumber=0; char\* p[**MAXARG**];

for(i=1;i<argc;++i) { p[pNumber++]=argv[i];

}

int place=pNumber; char ch;

char \*line; char s[512]; line=s;

int index=0; while(**read**(0,&ch,1)>0){

if(ch=='\n'){

line[index]='\0'; index=0;

p[pNumber++]=line; p[pNumber]=0;

if(**fork**()){

**wait**(0); pNumber=place;

}

else{

**exec**(argv[1],p);

}

}

else if (ch==' '){ line[index]='\0'; index=0; p[pNumber++]=line; char s[512]; line=s;

}

else{

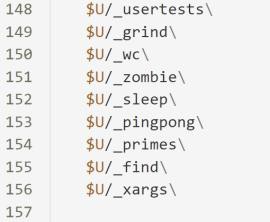
line[index++]=ch;

}

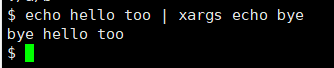
}

**exit**(0);

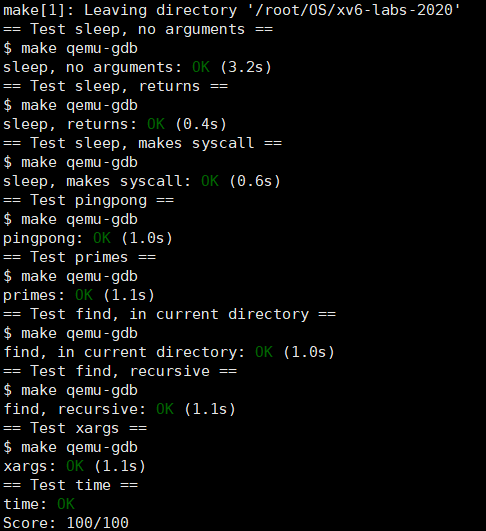
}

1. 添加到 MakeFile 文件中：

成功添加后，即可实现 xargs 功能：



### 综合测试

我通过 make grade 进行本实验综合测试：

至此，本实验全部完成。

## 实验小结

通过本实验，我初步了解了 xv6 这一操作系统内核，同时也了解了 qemu 模拟器的使用方法。

在完成实验的过程中，我遇到了比较多的困难，包括不知道管道如何创建、创建完成后如何关闭、素数筛到底是什么等等。但是通过和同学交流以及利用网络资源，我成功的解决了这些问题。

此外，我还初步了解到了系统调用接口，并且直到了操作系统内核和用户的一个区别。

# 实验二 System calls

## 实验目的

1. 进一步了解系统调用。
2. 掌握添加系统调用的方法。
3. 理解系统调用的工作原理和 xv6 内核的工作过程。

## 实验内容

1. 为 xv6 添加系统调用跟踪功能。
2. 添加系统调用 sysinfo，用于收集正在运行的系统的信息。

## 实验步骤

### 实现系统调用跟踪

1. 为了实现对系统调用的跟踪，我应该首先为进程 proc 结构体增加成员变量

mask，用来表示每一个进程的掩码：

struct **proc** {

struct **spinlock** lock;

......

int mask; *// 增加 mask*

};

1. 在添加了成员变量 mask 之后，我发现 syscall.c 中的 syscall 函数有以下语句：

p->trapframe->a0 = syscalls[num]();

这一行就是在调用系统调用，同时将系统调用的返回值存储在 a0 寄存器中。因此，我应当修改 syscalls 数组，添加：

[**SYS\_trace**] **sys\_trace**,

为了能够获得所调用函数的名称，需要注意到 p->name 所存储的是线程的名称，因此我需要额外定义一个数组：

char \*sysname[] = { [**SYS\_fork**] "fork", [**SYS\_exit**] "exit", [**SYS\_wait**] "wait", [**SYS\_pipe**] "pipe", [**SYS\_read**] "read", [**SYS\_kill**] "kill", [**SYS\_exec**] "exec", [**SYS\_fstat**] "stat", [**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", [**SYS\_sysinfo**] "sysinfo",

};

1. 修改 syscall 函数，增加两行，通过掩码和编号做与运算进行输出：

if((1<<num)&p->mask){

**printf**("%d: syscall %s -> %d\n", p->pid, sysname[num], p->t rapframe->a0);

}

1. 针对于 trace 的用户函数，我需要将 mask 等参数传递进来，因此需要修改

trace.c 文件。其中，main 函数如下：

int

**main**(int argc, char \*argv[])

{

int i;

char \*nargv[**MAXARG**];

if(argc < 3 || (argv[1][0] < '0' || argv[1][0] > '9')){ **fprintf**(2, "Usage: %s mask command\n", argv[0]); **exit**(1);

}

if (**trace**(**atoi**(argv[1])) < 0) {

**fprintf**(2, "%s: trace failed\n", argv[0]);

**exit**(1);

}

for(i = 2; i < argc && i < **MAXARG**; i++){ nargv[i-2] = argv[i];

}

**exec**(nargv[0], nargv);

**exit**(0);

}

1. 最后，修改 user.h、user.pl，同时在 syscall.c 中添加函数引用：

extern **uint64 sys\_trace**(void);

1. 在 sysproc.c 中添加 sys\_trace 实现，将参数传递给 mask 即可：

**uint64 sys\_trace**(void)

{

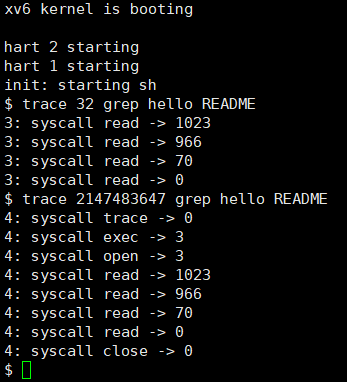
int mask;

if(**argint**(0, &mask) < 0) return -1;

**myproc**()->mask = mask; return 0;

}

1. 在 MakeFile 文件中添加 trace 后，即可成功运行：



### 实现 sysinfo

1. 通过阅读源代码，我发现 proc.c 文件中定义了一个数组：

struct **proc** proc[**NPROC**];

该数组中保存着所有的进程，同时进程的状态存储在 state 结构体中：

enum **procstate** {UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

1. 通过上述分析我可以看出，如果需要统计未被使用的进程，只需要遍历当前所有进程，记录状态为 UNUNSED 的进程即可。需要注意的是，在进行该操作之前需要进行上锁操作：

**uint64 unusedproc**(void){

struct **proc** \*p;

**uint64** unused=0;

for(p=proc;p<&proc[**NPROC**];++p){

**acquire**(&p->lock); if(p->state!=UNUSED){

++unused;

}

**release**(&p->lock);

}

return unused;

}

1. 针对于当前可以使用的空间，我可以参考 kalloc.c 文件。其中定义了一个链表 kemem，每个链表都指向了一个可用空间，而 kmem 则保存了最后一个链表。

struct {

struct **spinlock** lock; struct **run** \*freelist;

} kmem;

事实上，我参考 kfree 函数可以完成我的 freemem 函数。这里需要注意， 该函数计算的单位是 PGSIZE，因此我还需要乘上一页的大小（直接左移 12 位即可）：

**uint64 freemem**(void){

struct **run** \*r; **uint64** freepage=0; **acquire**(&kmem.lock); r=kmem.freelist; while(r){

++freepage; r=r->next;

}

**release**(&kmem.lock); return freepage<<12;

}

1. 完成上述步骤后，我即可在 sysproc.c 中实现 sysinfo 函数。我首先通过argaddr 函数传递参数，之后调用之前所定义的 freemem()和 unusedproc()即可。需要注意的地方在于，我最后调用 copyout()函数以完成将内容复制回用户空间的要求：

**uint64 sys\_sysinfo**(void){

**uint64** addr;

struct **sysinfo** info; struct **proc** \*p=**myproc**(); if(**argaddr**(0, &addr) < 0)

return -1; info.freemem=**freemem**(); info.nproc=**unusedproc**();

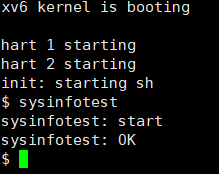
if(**copyout**(p->pagetable, addr, (char \*)&info, sizeof(info))

< 0)

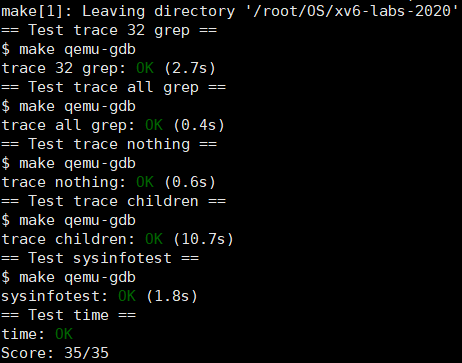
return -1;

return 0;

}

1. 在头文件中完成声明后，即完成了全部要求。测试如下：

### 综合测试

我通过 make grade 进行本实验综合测试：

## 实验小结

本次实验实际上是对系统调用的深入理解，在做这次实验的过程中我遇到的问题相对比较多。在第一个 trace 的过程中，需要阅读比较多的内核 kernel 源代码，包括 syscall()等函数，只有熟悉了这些函数才能够比较好的完成实验内容。

在完成第二部分 sysinfo 的时候，由于对进程的定义仅仅停留在课本上，因此刚开始的时候难以下手。但是通过查阅相关资料，我了解到 xv6 中对进程的定义， 以及存储所有进程的变量和进程状态变量，仿照着已经存在的内核函数，终于完成了该部分。

总体来说，这次实验让我在实验一的基础上更加了解系统调用，同时也为未来的实验打下了比较扎实的基础。此外，也让我更加熟悉了写系统调用的方法， 对内核和用户也有了比较好的区分。

# 实验三 Page Tables

## 实验目的

1. 了解 RISC-V 中的页表机制。
2. 深入理解逻辑地址到物理地址的映射关系，掌握如何实现一个页表映射。
3. 为每个进程实现一个页表，以更加深刻的认识到页表的重要性。
4. 加深对操作系统内存管理的理解。

## 实验内容

1. 定义函数 vmprint()，接受一个 pagetable\_t 参数，功能是打印页表结构。
2. 修改内核，使得每个进程在内核中执行的时候都使用自己的内核页表副本。
3. 内核中的 copyin()读取用户指针指向的内存，它会将其翻译成物理地址。给每个进程的内核页表都添加用户映射，使得 copyin 可以直接使用用户指针。

## 实验步骤

### 打印页表结构

1. 我在 vm.c 中定义 vmprint()函数，参数为 pagetable\_s，如下：

void

**vmprint**(**pagetable\_t** s){

**printf**("page table %p\n",s);

**vmprintdfs**(s,1);

}

1. 针对于 vmprintdfs 函数，它需要递归的调用页表中的每一项，并加以输出。需要注意的是，递归最大层数为 3 层。因此，代码如下：

void

**vmprintdfs**(**pagetable\_t** pagetable,int level){

*// there are 2^9 = 512 PTEs in a page table.*

for(int i = 0; i < 512; i++){

**pte\_t** pte = pagetable[i];

if(pte & **PTE\_V**){

*//print level*

for(int j=0;j<level;++j){

**printf**("..");

}

*// this PTE points to a lower-level page table.*

**uint64** child = **PTE2PA**(pte);

**printf**("%d: pte %p pa %p\n",i,pte,child); if(level==3){

continue;

}

else{

**vmprintdfs**((**pagetable\_t**) child,level+1);

}

}

}

}

1. 在 exec.c 中添加对于 vmprint()的调用，当进程的 pid 等于 1 的时候即调用该函数：

if(p->pid==1)

**vmprint**(p->pagetable);

1. 完成后，我在 qemu 模拟器中测试如下：



### 每个进程一个内核页表

1. 由于每个进程都有一个内核页表，因此首先我应该给 proc 结构体添加一个成员变量：

struct **proc** {

......

*//kernel pageTable*

**pagetable\_t** kernelPagetable;

};

1. 在 vm.c 中新增加一个 vmmake()方法，用来创建内核页表，其中不需要包含对CLINT 的映射：

**pagetable\_t vmmake**(){

**pagetable\_t** kernel\_pagetable = (**pagetable\_t**) **kalloc**(); **memset**(kernel\_pagetable, 0, **PGSIZE**);

*// uart registers*

**mappages**(kernel\_pagetable,**UART0**, **UART0**, **PGSIZE**, **PTE\_R** | **PTE\_ W**);

*// virtio mmio disk interface*

**mappages**(kernel\_pagetable,**VIRTIO0**, **VIRTIO0**, **PGSIZE**, **PTE\_R** |

**PTE\_W**);

*// CLINT*

*//mappages(CLINT, CLINT, 0x10000, PTE\_R | PTE\_W);*

*// PLIC*

**mappages**(kernel\_pagetable,**PLIC**, **PLIC**, 0x400000, **PTE\_R** | **PTE\_ W**);

*// map kernel text executable and read-only.*

**mappages**(kernel\_pagetable,**KERNBASE**, **KERNBASE**, (**uint64**)etext-

**KERNBASE**, **PTE\_R** | **PTE\_X**);

*// map kernel data and the physical RAM we'll make use of.*

**mappages**(kernel\_pagetable,(**uint64**)etext, (**uint64**)etext, **PHYS TOP**-(**uint64**)etext, **PTE\_R** | **PTE\_W**);

*// map the trampoline for trap entry/exit to*

*// the highest virtual address in the kernel.*

**mappages**(kernel\_pagetable,**TRAMPOLINE**, (**uint64**)trampoline, **PG SIZE**, **PTE\_R** | **PTE\_X**);

return kernel\_pagetable;

}

1. 定义了 vmmake()后，我修改 vminit()方法，将其内部的实现通过之前所定义的 vmmake()来完成，同时需要 CLINT 映射：

void

**kvminit**()

{

*/\**

*kernel\_pagetable = (pagetable\_t) kalloc(); memset(kernel\_pagetable, 0, PGSIZE);*

*// uart registers*

*kvmmap(UART0, UART0, PGSIZE, PTE\_R | PTE\_W);*

*// virtio mmio disk interface*

*kvmmap(VIRTIO0, VIRTIO0, PGSIZE, PTE\_R | PTE\_W);*

*// CLINT*

*kvmmap(CLINT, CLINT, 0x10000, PTE\_R | PTE\_W);*

*// PLIC*

*kvmmap(PLIC, PLIC, 0x400000, PTE\_R | PTE\_W);*

*// map kernel text executable and read-only. kvmmap(KERNBASE, KERNBASE, (uint64)etext-*

*KERNBASE, PTE\_R | PTE\_X);*

*// map kernel data and the physical RAM we'll make use of. kvmmap((uint64)etext, (uint64)etext, PHYSTOP-*

*(uint64)etext, PTE\_R | PTE\_W);*

*// map the trampoline for trap entry/exit to*

*// the highest virtual address in the kernel. kvmmap(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE\_R | PTE\_X*

*);*

*\*/*

kernel\_pagetable=**vmmake**();

**mappages**(kernel\_pagetable,**CLINT**,0x1000,**CLINT**,**PTE\_R**|**PTE\_W**);

}

1. 我在 proc.c 的 procinit()方法中，修改代码，不在该处为每个进程分配内核：

void

**procinit**(void)

{

struct **proc** \*p;

**initlock**(&pid\_lock, "nextpid");

for(p = proc; p < &proc[**NPROC**]; p++) {

**initlock**(&p->lock, "proc");

*// Allocate a page for the process's kernel stack.*

*// Map it high in memory, followed by an invalid*

*// guard page.*

*// char \*pa = kalloc();*

*// if(pa == 0)*

*// panic("kalloc");*

*// uint64 va = KSTACK((int) (p - proc));*

*// kvmmap(va, (uint64)pa, PGSIZE, PTE\_R | PTE\_W);*

*// p->kstack = va;*

}

*//kvminithart();*

}

1. 由于不再在 procinit()方法中分配内核，我需要在 allocproc()中创建内核页表， 并且在内核页表上分配一个内核栈。如下所示：

static struct **proc**\* **allocproc**(void)

{

......

*// Allocate a trapframe page.*

if((p->trapframe = (struct **trapframe** \*)**kalloc**()) == 0){

**release**(&p->lock); return 0;

}

*//create a kernel pagetable* p->kernelPagetable=**vmmake**(); char \*pa=**kalloc**(); if(pa==0){

**panic**("kalloc");

}

**uint64** va=**TRAMPOLINE**-2\***PGSIZE**;

*//mappages(p->kernelPagetable,va,PGSIZE,(uint64)pa,PTE\_R|PTE*

*\_W);*

p->kstack=va;

......

}

1. 在 proc.c 中，修改 scheduler()，在 swtch()切换进程前修改 satp，保证进程执行期间用的是进程内核页表，切换完后再修改 satp 为全局内核页表。该部分代码如下所示：

**w\_satp**(**MAKE\_SATP**(p->kernelPagetable)); **sfence\_vma**();

**swtch**(&c->context, &p->context);

**kvminithart**();

1. 我需要注意到，kvmpa()方法会在进程执行的过程中被调用。根据题目所告诉我的，此时不应该再调用全局内核页表，而是应该调用进程内核页表。因此， 我应当加上下面这句：

pte = **walk**(**myproc**()->kernelPagetable, va, 0);

同时需要注意，由于该函数中调用了 myproc()函数，我还需要修改 proc.c 的头文件引用。

1. 我还需要在 proc.c 中新增 proc\_free\_kernel\_pagetable()方法，用于释放该进程中的内核页表所指向的内存，以及内核页表。该部分代码如下所示：

void

**proc\_free\_kernel\_pagetable**(**uint64** kstack,**pagetable\_t** pagetable

){

**uvmunmap**(pagetable,**UART0**,1,0); **uvmunmap**(pagetable,**VIRTIO0**,1,0); **uvmunmap**(pagetable,**CLINT**,0x10000/**PGSIZE**,0); **uvmunmap**(pagetable,**PLIC**,0x40000/**PGSIZE**,0); **uvmunmap**(pagetable,**KERNBASE**,((**uint64**)etext-

**KERNBASE**)/**PGSIZE**,0);

**uvmunmap**(pagetable,(**uint64**)etext,(**PHYSTOP**- (**uint64**)etext)/**PGSIZE**,0);

**uvmunmap**(pagetable,**TRAMPOLINE**,1,0);

**uvmfree2**(pagetable,kstack,1);

}

1. 最后，我在 vm.c 中新增 uvmfree2()方法，用于释放内核页表中的内核栈， 代码如下：

void

**uvmfree2**(**pagetable\_t** pagetable, **uint64** va,**uint** npages)

{

if(npages > 0)

**uvmunmap**(pagetable, va,npages,1);

**freewalk**(pagetable);

}

### 简化 copyin/copyinstr

1. 首先，根据 hints 中的提示所述，我需要修改 fork()、exec()和 sbrk()方法：

int

**fork**(void)

{

int i, pid; struct **proc** \*np;

struct **proc** \*p = **myproc**();

*// Allocate process.*

if((np = **allocproc**()) == 0){ return -1;

}

*// Copy user memory from parent to child.*

if(**uvmcopy**(p->pagetable, np->pagetable, p->sz) < 0){

**freeproc**(np); **release**(&np->lock); return -1;

}

np->sz = p->sz;

np->parent = p;

*// copy saved user registers.*

\*(np->trapframe) = \*(p->trapframe);

*// Cause fork to return 0 in the child.*

np->trapframe->a0 = 0;

*// increment reference counts on open file descriptors.*

for(i = 0; i < **NOFILE**; i++)

if(p->ofile[i])

np->ofile[i] = **filedup**(p->ofile[i]); np->cwd = **idup**(p->cwd);

**safestrcpy**(np->name, p->name, sizeof(p->name)); pid = np->pid;

np->state = RUNNABLE;

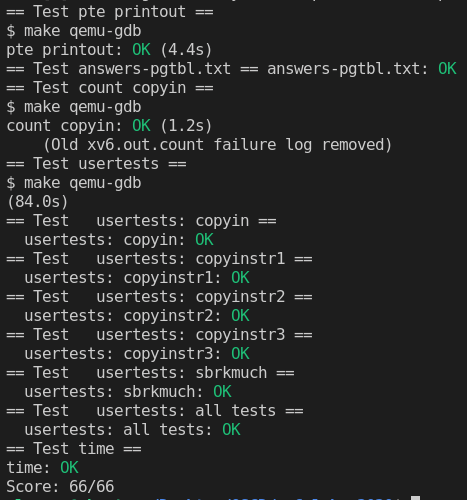
**release**(&np->lock); return pid;

}

在 exec()方法中，我需要释放进程旧内核页面映射，同时建立新页表的的映射。需要注意，我需要添加 PLIC 限制。而针对于 sbrk()，则是在内存大小发生变化的时候，需要相应更改进程内核页表。

1. 根据提示所述，我还需要修改 proc\_free\_kernel\_pagetable()方法，取消进程页表地址映射。完成上述步骤后，替换相应函数即可。

### 综合测试



## 实验小结

这个实验可以说是所有实验中花费时间最多的一个实验了。因为这次实验涉及到了操作系统中一个重要的概念，就是页表结构。同时，还涉及到了逻辑地址到物理地址之间的转换，只有把这些概念牢牢掌握才能够比较好的完成这一次的实验。

在本次实验的过程中，还涉及到了操作系统中地址的定位，在完成的过程中还要注意越界判断等。所以可以说，这次实验的完成是对能力的一个很好的提升。

# 实验四 traps

## 实验目的

1. 进一步了解系统调用所发挥的重要作用。
2. 掌握通过中断进行系统调用的过程及其所发挥的作用。
3. 了解简单的汇编语言，知道汇编语言是如何发挥作用的。
4. 理解 xv6 中的堆栈，同时试着实现一个用户级中断处理。

## 实验内容

1. 了解 RISC-V 程序集，阅读 call.asm 中的函数 g、f 和 main 的代码，并回答问题。
2. 完成 Backtrace 函数，当发生错误的时候查看当前堆栈中的系统调用。
3. 添加系统调用 sigalarm，当用户程序运行了 n 个 ticks 后，触发回调函数。

## 实验步骤

### RISC-V 程序集

1. 我先阅读 call.asm 的源代码，可以发现有函数 g、函数 f 和 main 函数，其中：main 函数调用如下：

exit(0);

a0,0x0

a0,a0,1968 # 7d8 <malloc+0xea> ra,0x0

1536(ra) # 630 <printf>

auipc

addi auipc jalr

li a2,13

li a1,12

4635

45b1

00000517

7b050513

00000097

600080e7

24:

26:

28:

2c:

30:

34:

printf("%d %d\n", f(8)+1, 13);

s0,sp,16

addi

addi sp,sp,-16

sd ra,8(sp) sd s0,0(sp)

1141

e406 e022

0800

1c:

1e:

20:

22:

void main(void) {

|  |  |  |
| --- | --- | --- |
| 38: | 4501 | li a0,0 |
| 3a: | 00000097 | auipc ra,0x0 |
| 3e: | 27e080e7 | jalr 638(ra) # 2b8 <exit> |

1. 哪一个寄存器存储了传递给函数的参数？例如，哪一个寄存器存储了 main 函数中调用 printf 函数的参数 13？

通过阅读源代码，我可以发现 RISC-V 中一共有 a0~a7 一共 8 个寄存器。在 main 函数调用 printf 函数的过程中，参数 13 被存储在了寄存器 a2 中。

1. 在汇编代码中，main 函数对函数 f 的调用在哪里？对函数 g 的调用在哪里？ 需要注意，编译器可能会进行函数内联。

通过阅读 main 函数，我会发现对于函数 f 调用是直接计算出了结果 12， 对于函数 g 的调用则是内联在了函数 f 中。

1. 函数 printf 位于什么位置？

通过阅读源代码，我发现了这一句：

34: 600080e7 jalr 1536(ra) # 630 <printf>

也即跳转到 ra+1536 的位置，此时 ra 的值即为 pc 的值，也即 printf 的地址为

0x30+1536=0x630。

1. 在 main 函数通过 jalr 跳转到 printf 函数后，此时 ra 寄存器中的值为多少？ 查询 jalr 指令，我发现：

|  |
| --- |
| 跳转并链接指令（jal）具有双重功能。若将下一条指令 PC + 4 的地址保存到目标寄存器 |
| 中，通常是返回地址寄存器 ra，便可以用它来实现过程调用。如果使用零寄存器（x0）替 |
| 换 ra 作为目标寄存器，则可以实现无条件跳转，因为 x0 不能更改。像分支一样，jal 将 |
| 其 20 位分支地址乘以 2，进行符号扩展后再添加到 PC 上，便得到了跳转地址。  跳转和链接指令的寄存器版本（jalr）同样是多用途的。它可以调用地址是动态计算出来的函数，或者也可以实现调用返回（只需 ra 作为源寄存器，零寄存器（x0）作为目的寄  存器）。Switch 和 case 语句的地址跳转，也可以使用 jalr 指令，目的寄存器设为 x0。 |

因此，ra 此时存储的应当是 printf 函数的返回地址，也即 0x38。

1. 我直接运行下面的代码：

unsigned int i = 0x00646c72;

**printf**("H%x Wo%s", 57616, &i);

输出结果为：He110 World。我不需要修改，因为编译器会自己进行转换。

1. 在下面的代码中，‘y=’后面所输出的值是什么？为什么会发生这一情况？

**printf**("x=%d y=%d", 3);

它所输出的是寄存器 a2 的值，这是因为 printf 会从 a2 寄存器读取参数作为

y 的值。

### 实现 Backtrace

1. 首先，我应该在 defs.h 中声明 backtrace()：

void **backtrace**(void);

1. 通过阅读题目，我获得了一个定义在 ricsv.h 中用来获取当前栈帧指针的函数：

static inline **uint64 r\_fp**()

{

**uint64** x;

asm volatile("mv %0, s0" : "=r" (x) ); return x;

}

1. 需要注意到，下一个栈帧指针地址为 fp-8，返回值地址为 fp-16。基于此，利用循环就可以完成：

void

**backtrace**(void){ **printf**("backtrace:\n");

*//get the start location*

**uint64** curLoc=**r\_fp**();

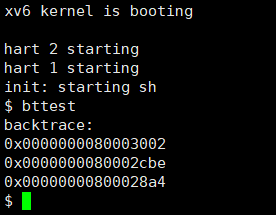
**uint64** upLine=**PGROUNDUP**(curLoc);

while(curLoc<upLine){ **printf**("%p\n",\*((**uint64**\*)(curLoc-8))); curLoc=\*((**uint64**\*)(curLoc-16));

}

}

1. 完成之后，我在 qemu 模拟器中测试如下：



### 实现 alarm

1. 这一题难度相对比较大，涉及到了比较多的内核和用户程序。首先，我需要参照实验二将 sigalarm()和 sigreturn()声明为系统调用：

int **sigalarm**(int ticks, void (\*handler)()); int **sigreturn**(void);

1. 我首先试着实现 sigalarm()函数。通过阅读题设资料，我可以发现这个函数的作用是设定给定的时间间隔以及函数。因此，我需要给进程增加两个属性， 用于保存这两个参数。

struct **proc** {

struct **spinlock** lock;

......

*//the interval of the alarm*

int alarmInterval;

*//the function that deal with the alarm*

**uint64** handler;

};

添加了这两个参数后，我还需在进程初始化的时候给这两个参数初始化， 也即在 allocproc()中给这几个参数加以初始化：

p->alarmInterval=0; p->handler=0;

1. 在完成了上述参数的定义后，就可以开始试着实现 sigalarm()。和需求一致， 该函数内部只需要将所给参数赋值给进程中的成员变量即可。因此，定义如下所示：

**uint64 sys\_sigalarm**(void){

int interval;

**uint64** handler;

*//check the parameter*

if(**argint**(0,&interval)<0){ return -1;

}

if(**argaddr**(0,&handler)<0){ return -1;

}

**myproc**()->alarmInterval=interval; **myproc**()->handler=handler; **myproc**()->tickNow=0;

**myproc**()->flag=1;

return 0;

}

1. 通过阅读 hints，我了解到：每次 tick 产生，硬件都会产生一次中断。因此， 我在 trap.c 的 usertrap()中对该中断进行处理即可：

if(which\_dev == 2){ p->tickNow++; **yield**();

}

1. 在完成了 sigalarm()函数后，我便需要实现 sigreturn()函数，也即到达一定时间 ticks 后如何调用我预先设定的自定义函数了。我注意到进入 sigalarm 时寄存器内容已经被改变，但是中断结束之后返回用户程序的位置应该和调用中断之前相同（包括寄存器内容），因此我需要在调用函数之前将所有寄存器中的内容存储下来。

基于这一思想，我首先需要在进程中增加以下成员变量：

**uint** flag;

**uint64** kernel\_satp; *// kernel page table*

**uint64** kernel\_sp; *// top of process's kernel stack*

**uint64** kernel\_trap; *// usertrap()*

**uint64** kernel\_hartid;

*/\* 40 \*/* **uint64** ra;

*/\* 48 \*/* **uint64** sp;

*/\* 56 \*/* **uint64** gp;

*/\* 64 \*/* **uint64** tp;

*/\* 72 \*/* **uint64** t0;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| */\** | *80* | *\*/* | **uint64** | t1; |
| */\** | *88* | *\*/* | **uint64** | t2; |
| */\** | *96* | *\*/* | **uint64** | s0; |
| */\** | *104* | *\*/* | **uint64** | s1; |
| */\** | *112* | *\*/* | **uint64** | a0; |
| */\** | *120* | *\*/* | **uint64** | a1; |
| */\** | *128* | *\*/* | **uint64** | a2; |
| */\** | *136* | *\*/* | **uint64** | a3; |
| */\** | *144* | *\*/* | **uint64** | a4; |
| */\** | *152* | *\*/* | **uint64** | a5; |
| */\** | *160* | *\*/* | **uint64** | a6; |
| */\** | *168* | *\*/* | **uint64** | a7; |
| */\** | *176* | *\*/* | **uint64** | s2; |
| */\** | *184* | *\*/* | **uint64** | s3; |
| */\** | *192* | *\*/* | **uint64** | s4; |
| */\** | *200* | *\*/* | **uint64** | s5; |
| */\** | *208* | *\*/* | **uint64** | s6; |
| */\** | *216* | *\*/* | **uint64** | s7; |
| */\** | *224* | *\*/* | **uint64** | s8; |
| */\** | *232* | *\*/* | **uint64** | s9; |
| **uint64** | | s10; | | |
| **uint64** | | s11; | | |
| **uint64** | | t3; | | |
| **uint64** | | t4; | | |
| **uint64** | | t5; | | |
| **uint64** | | t6; | | |
| **uint64** | | epc; | | |

1. 定义完上面的成员变量后，我就可以开始着手实现 sigreturn()函数。通过定义我知道，在这个函数里面只需要将之前存储下来的进程状态赋值回去就可以 了。同时需要注意到，为了防止对进程状态写冲突，我还定义了一个状态为 flag：

**uint64 sys\_sigreturn**(void){

struct **proc** \*p = **myproc**(); p->trapframe->ra = p->ra; p->trapframe->sp = p->sp; p->trapframe->gp = p->gp; p->trapframe->tp = p->tp; p->trapframe->t0 = p->t0; p->trapframe->t1 = p->t1; p->trapframe->t2 = p->t2;

p->trapframe->s0 = p->s0; p->trapframe->s1 = p->s1; p->trapframe->a0 = p->a0; p->trapframe->a1 = p->a1; p->trapframe->a2 = p->a2; p->trapframe->a3 = p->a3; p->trapframe->a4 = p->a4; p->trapframe->a5 = p->a5; p->trapframe->a6 = p->a6; p->trapframe->a7 = p->a7; p->trapframe->s2 = p->s2; p->trapframe->s3 = p->s3; p->trapframe->s4 = p->s4; p->trapframe->s5 = p->s5; p->trapframe->s6 = p->s6; p->trapframe->s7 = p->s7; p->trapframe->s8 = p->s8; p->trapframe->s9 = p->s9;

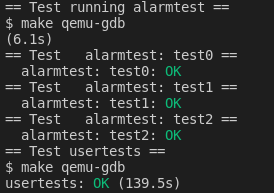
p->trapframe->s10 = p->s10; p->trapframe->s11 = p->s11; p->trapframe->t3 = p->t3;

p->trapframe->t4 = p->t4; p->trapframe->t5 = p->t5; p->trapframe->t6 = p->t6; p->trapframe->epc=p->epc;

**myproc**()->flag=1; return 0;

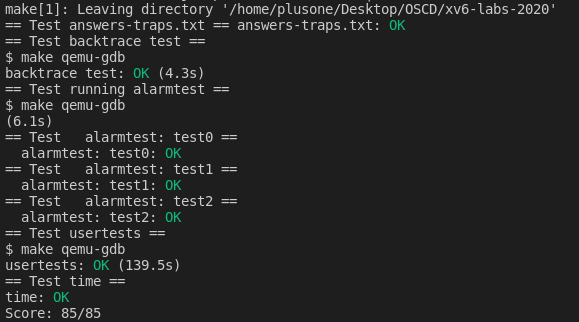
}

1. 最后，在上面的进入该函数之前利用寄存器存储下来当前进程状态，就完成了本次实验。测试如下：



### 综合测试

我通过 make grade 进行本实验综合测试：



## 实验小结

这次实验主要是针对于操作系统中一个非常重要的中断机制而展开的。中断是指计算机运行过程中，出现某些意外情况需主机干预时，机器能自动停止正在运行的程序并转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行。

在本次实验的 alarm 中，我实际上就是通过实现中断现场的保存和处理结束后现场的恢复，实现了一个中断的过程。在这个过程中，其实我遇到的问题挺多的，包括不知道哪些状态需要保存下来、保存到什么地方去、保存之后怎么办等等。但是真正把这一个实验做下来之后，我对中断也有了更加深刻的理解，对操作系统中断机制也有了更好的认识。

# 实验五 Lazy Allocation

## 实验目的

1. 通过该实验了解操作系统中常用的策略惰性空间分配。
2. 体会到惰性空间分配对提高操作系统性能所发挥的巨大作用。
3. 尝试完成惰性空间分配，并体会它所带来的性能上的提升。

## 实验内容

1. 从 sbrk()系统调用实现中删除页面分配，即 sysproc.c 中的函数 sys\_sbrk()。
2. 修改 trap.c 中的代码，通过在错误地址映射新分配的物理内存页来响应来自用户空间的页面错误，然后返回到用户空间让进程继续执行。
3. lazytests 用于测试一些可能会给惰性内存分配器带来压力的特定情况，修改内核代码以通过全部测试。

## 实验步骤

### 删除 sbrk 中的分配

这个问题相对而言比较简单，只需要直接修改 sbrk()函数中分配内存空间的部分就可以了。

因此，我删除 growproc(n)如下所示：

**uint64 sys\_sbrk**(void)

{

int addr; int n;

if(**argint**(0, &n) < 0) return -1;

addr = **myproc**()->sz;

*//return new size*

**myproc**()->sz=addr+n;

return addr;

*//if(growproc(n) < 0)*

*// return -1;*

return addr;

}

### 实现 lazy allocation

1. 我首先需要修改 vm.c 中的 mappages()函数，去掉它的 static 关键词，从而能够在 trap 中调用该函数。
2. 在 RISCV 中，缺页异常号分别是 13 和 15（对应于读和写），该异常号会被保存在 scause 寄存器中，同时导致缺页的地址会被保存在 stval 寄存器中。因此， 我只需要在 trap.c 中对于 r\_scause()的返回值为 13 和 15 的情况进行处理即可。

if(**r\_scause**() == 8){

*// system call*

......

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

**uint64** va = **r\_stval**();

if (va >= p->sz || va < p->trapframe->sp) { p->killed = 1;

} else {

**uint64** ka = (**uint64**) **kalloc**(); if (ka == 0){

p->killed = 1;

} else {

**memset**((void \*)ka, 0, **PGSIZE**); va = **PGROUNDDOWN**(va);

if(**mappages**(p->pagetable, va, **PGSIZE**, ka, **PTE\_W**|**PTE\_R**| **PTE\_U**) != 0) {

**kfree**((void \*)ka); p->killed = 1;

}

}

}

}

1. 最后，我要注意到如果一个进程需要释放自己申请了但是没有访问过的空间，由于 lazy allocation 的机制会导致该空间并没有真正被分配，因此会导致 panic 报错。我需要修改 uvmunmap()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");*

continue;

}

if((\*pte & **PTE\_V**) == 0){

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

*//it is ok for pagefault*

continue;

}

if(**PTE\_FLAGS**(\*pte) == **PTE\_V**) **panic**("uvmunmap: not a leaf");

if(do\_free){

**uint64** pa = **PTE2PA**(\*pte); **kfree**((void\*)pa);

}

\*pte = 0;

}

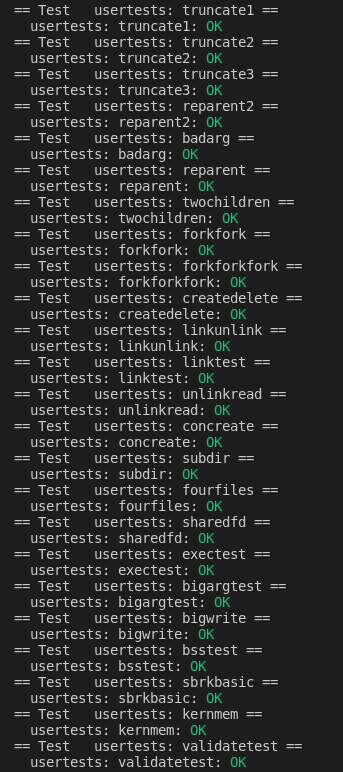
}

1. 图片至此，本部分完成。测试如下：

### 通过 lazytests 和 usertests

根据题目所说，这两个测试会检查我所完成的 lazy allocation 在日常情景中能否正常完成工作。实际上，我要解决的是 write 和 read 系统调用访问了申请但是尚未分配地址空间这一情景。通过阅读相关材料，我了解到：write 和read 系统调用最终会调用内核中的 copyin 和 copyout 函数，因此修改这两个函数就可以了。

### 综合测试

我通过 make grade 进行本实验的综合测试：

## 实验小结

在这个实验中，我真正实现了 UNIX 操作系统中为了提高效率而采用的一个策略，即 Lazy Allocation 惰性分配。这个实验实际上是实验三的拓展，只有掌握了实验三的页表分配机制才能够比较好的完成本次实验。

此外，本次实验中也更加深刻的体会到了用户空间和内核空间之间的区别。例如第二部分修改了 usertrap 还不够，还需要在第三部分中修改 kernel 中的异常处理。只有两部分结合起来，才是更好的操作系统。

# 实验六 Copy on write

## 实验目的

1. 理解什么是 Copy On Write（COW），其思想和作用是什么。
2. 熟悉简单 fork() 系统调用的工作机制，提出一种 COW 的实现手段。
3. 理解 xv6 中的 fork 函数，并修改内核程序，实现 COW。

## 实验内容

修改 xv6 内核代码，使系统支持 Copy on Write 功能。虽然只有一项，但是直接完成该实验是十分困难的，因此我将实验内容拆分为了三个步骤。前两个步骤帮助我理解这个实验的内容，第三个步骤才着手修改 xv6 的源代码。

完善后的实验内容：

* + 1. 了解 Unix 中的 fork() 系统调用的功能以及实现机制，重点了解调用 fork()

过程中的内存分配流程。

* + 1. 阅读 xv6 源码，理解 fork 函数以及父子进程拷贝内容的机制。
    2. 根据提示中给出的方案啊，修改 xv6 内核使系统支持 COW。

## 实验步骤

### 理解 fork() 系统调用

首先需要理解系统中的 fork 函数原型——也就是 Unix 系统中 fork 所完成的工作，为此我查阅了一些资料。

资料中提到，在一个 Unix-like 的操作系统中，一个进程创建新进程的唯一方法是通过系统调用 fork。若是一个进程想要（并发地）执行一个新的程序，它应该先调用 fork 创建一个子进程，之后在子进程中使用 exec 系统调用来覆盖之后的行为。

fork 系统调用会拷贝当前进程的所有内存内容，创建一个只有 PID 和 PPID 与原来进程不一样的新进程。原进程被称作“父进程”，而新的进程被称作“子进程”。

在一个有效的 fork() 实现中，并不是所有父进程的内存空间/页面（在分页式内存管理中）都需要被拷贝。有以下两点原因：

* + - 1. 多数情况下，子进程都会通过 exec 覆盖父进程之后的程序体，父进程所拥有的程序、数据等占用内存空间的地方使用率不高。
      2. 只有在父进程或者子进程修改了原有的数据后，才有必要对这一块内存进行拷贝。如果不做写入（write）而只是读取（read），并没有必要复制，父子进程间只要共享这个区域的内存就可以。

上面的描述也就是 Copy on Write 技术的核心思想：通过 fork 系统调用产生的进程，只在对内存进行写入时，才复制对应的内存空间。

### 阅读 xv6 源码

理解了 fork 系统调用的工作原理并且了解了 COW 技术的思路，我需要通过阅读 xv6 源码来印证 fork 函数的工作原理，并且寻找我需要修改的部分， 帮助我实现 COW 功能。

在 xv6 操作系统 fork 函数的实现代码有一个片段：

*// Copy user memory from parent to child.*

if(**uvmcopy**(p->pagetable, np->pagetable, p->sz) < 0){

**freeproc**(np); **release**(&np->lock); return -1;

}

通过函数 **uvmcopy** 复制了用户空间的所有页表。如果 **uvmcopy** 并不复制页面，而是将父子进程的页面关联（共享），并标记成只读状态，在写该页时就会发生一个缺页中断。像实验 5 一样，针对缺页中断在 **usertrap** 函数中处理中断，为进程分配页面就能实现 COW 的主体功能。

在这样的解决方案之下存在另一个需要解决的问题：页面状态的记录与释放。Fork 调用产生的进程可能拥有自己修改过的页面（有 PTE\_W 标记的页面），或是与父进程共享一个页面（只有 PTE\_R 标记的页面）。在实现 COW 功能时需要正确地分配、记录和释放子进程产生的页面。

另外，xv6 提供了一个 **copyout** 函数以便用户将内存空间从内核拷贝到用户空间，这个函数也需要处理遇到 COW 页面的情况。

### 编码实现 COW

根据以上方案，我可以开始实现 COW 功能。

* + - 1. 修改 uvmcopy 函数，使其不会立即分配空间，而是把页面标记为 COW 页面。

**acquire**(&ref\_cnt\_lock);

reference\_count[pa / PGSIZE] += 1; *// 递增 rc*

**release**(&ref\_cnt\_lock);

\*pte &= ~PTE\_W; *// 去掉写标记*

\*pte |= PTE\_COW; *// 标记为 COW 页*

flags = **PTE\_FLAGS**(\*pte);

if(**mappages**(new, i, PGSIZE, (uint64)pa, flags) != 0){ goto err;

}

代码中出现的 reference\_count 和 ref\_cnt\_lock 定义在 kalloc.c 中，

PTE\_COW 定义在 riscv.h 中。

kalloc.c

int reference\_count[PHYSTOP / PGSIZE]; struct spinlock ref\_cnt\_lock;

在 **kinit** 函数中，将这两个变量初始化：

**initlock**(&rc\_lock, "refCount");

**acquire**(&kmem.lock);

for (int i = 0; i < (PHYSTOP / PGSIZE); i++)

{

reference\_count[i] = 0;

}

**release**(&kmem.lock);

riscv.h

#define **PTE\_COW** (1L << 5)

* + - 1. 修改 **usertrap** 函数，使其处理缺页中断时为 COW 页面分配内存空间。

trap.c

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

*// ok*

} else if (**r\_scause**() == 15 || **r\_scause**() == 13) {*// 处理缺页*

pte\_t \*pte;

uint64 va = **r\_stval**(); *// 缺页地址*

struct proc \*p = **myproc**();

if((pte = **walk**(p->pagetable, va, 0)) == 0) p->killed = 1; *// pte 页面不存在*

if ((\*pte & PTE\_V) == 0) p->killed = 1;

if ((\*pte & PTE\_COW) == 0)

p->killed = 1; *// 页面不是 COW*

*// 分配新空间*

uint64 pa = **PTE2PA**(\*pte); *// 共享页面物理地址*

uint64 ka = (uint64) **kalloc**(); *// 分配新空间*

if (ka == 0) {

p->killed = 1; *// 内存不足*

} else {

**memmove**((char\*)ka, (char\*)pa, PGSIZE); *// 拷贝页面*

va = **PGROUNDDOWN**(va); *// 虚拟地址*

uint flags = **PTE\_FLAGS**(\*pte);

**uvmunmap**(p->pagetable, va, 1, 1); *// 与原页面解绑*

\*pte = **PA2PTE**(ka) | flags | PTE\_W;

\*pte &= ~PTE\_COW;

}

}

同时，在 **kalloc** 函数中应该把对应的 reference\_count 项设为 1。

kalloc.c

if(r) {

**memset**((char\*)r, 5, PGSIZE); *// fill with junk* **acquire**(&ref\_cnt\_lock); reference\_count[(uint64)r>>12] = 1; *// 初始化 rc* **release**(&ref\_cnt\_lock);

}

在函数中需要把对应页面的项减一，以实现解绑。

vm.c

uint64 pa = **PTE2PA**(\*pte);

**acquire**(&ref\_cnt\_lock);

reference\_count[pa>>12] -= 1;

**release**(&ref\_cnt\_lock);

if(do\_free){ … }

* + - 1. 修改 **kfree** 函数，使其在一个页面的引用计数（reference\_count）清零时才释放该页面的物理空间。

kalloc.c

void

**kfree**(void \*pa)

{

struct run \*r;

**acquire**(&ref\_cnt\_lock);

if (reference\_count[(uint64)pa >> 12] == 0) {

**release**(&ref\_cnt\_lock);

if(((uint64)pa % PGSIZE) != 0 || (char\*)pa < end || (uint64)pa >= PHYSTOP) {

**panic**("kfree");

}

**memset**(pa, 1, PGSIZE); r = (struct run\*)pa; **acquire**(&kmem.lock);

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

} else {

*// 什么也不做*

**release**(&ref\_cnt\_lock);

}

}

* + - 1. 最后修改函数 **copyout**，与修改 **usertrap** 的方式一样。这个函数提供了将内核内存拷贝到用户内存空间的功能。

vm.c

int

**copyout**(pagetable\_t pagetable, uint64 dstva, char \*src, uint64 len)

{

uint64 n, va0, pa0; pte\_t \*pte;

while(len > 0){

va0 = **PGROUNDDOWN**(dstva);

pa0 = **walkaddr**(pagetable, va0); if(pa0 == 0)

return -1;

pte = **walk**(pagetable, va0, 0); if (\*pte & PTE\_COW) {

uint64 ka = (uint64) **kalloc**(); *// 分配新空间*

if (ka == 0){

struct proc \*p = **myproc**();

p->killed = 1; *// 杀死出错进程*

} else {

**memmove**((char\*)ka, (char\*)pa0, PGSIZE); *// 拷贝内容*

uint flags = **PTE\_FLAGS**(\*pte); **uvmunmap**(pagetable, va0, 1, 1);

\*pte = **PA2PTE**(ka) | flags | PTE\_W;

\*pte &= ~PTE\_COW; pa0 = ka;

}

}

n = PGSIZE - (dstva - va0); if(n > len)

n = len;

**memmove**((void \*)(pa0 + (dstva - va0)), src, n);

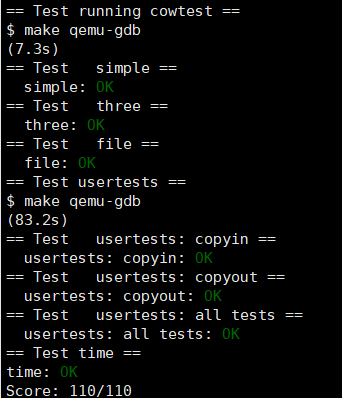
len -= n; src += n;

dstva = va0 + PGSIZE;

}

return 0;

}

* + - 1. 完成之后，我在 qemu 模拟器中测试如下：

## 实验小结

实验 5 和本次实验都是关于内存分配策略的设计。本次实验实现了按需复制的 fork 策略，这也是实现了 fork 系统调用的操作系统中一般都会实现的优化方式。

此外，虽然在实验任务中并未提到，但是我观察到 xv6 系统实现了分页式的内存管理，这使得父子进程间的内存可以按照页面为单位进行拷贝，这也提高 了一定的内存有效数据比例、降低了一定的拷贝开销（相较于直接复制整个进程）。通过这次实验，我对操作系统中内存分配的方式和机制有了更深刻的体会，完成 COW 功能的过程也帮助我回顾了一块内存的分配、使用、销毁回收这个完整的生命周期。

# 实验七 Multithreading

## 实验目的

1. 熟悉多线程编程，理解进程切换方式和执行顺序。
2. 了解多核多线程对程序执行效率的影响。
3. 理解锁的含义，能够判断何时、何处应该加锁。
4. 回顾进程同步/互斥的模式并在程序中应用。

## 实验内容

1. 实现一个用户级进程的创建和切换。
2. 使用 UNIX pthread 线程库实现一个线程安全的哈希表。
3. 实现 barrier 函数。其作用是：当进程到达 barrier 函数调用时，会开始等待其他进程，当所有进程都到达 barrier 时，才停止等待。

## 实验步骤

### 实现一个用户级进程的创建和切换

该实验要求修改 user/uthread.c 和 user/uthread\_switch.S 程序，从而实现进程间的切换。该步骤有两个要点：

* + - 1. 保存被替换进程的上下文。
      2. 恢复进入运行态进程的上下文。

因此需要有一个结构体来存储需要保存的内容，即进程运行时的寄存器：

user/uthread.c

struct uthread\_context { uint64 ra;

uint64 sp;

*// callee-saved registers*

uint64 s0; uint64 s1; uint64 s2;

uint64 s3; uint64 s4; uint64 s5; uint64 s6; uint64 s7; uint64 s8; uint64 s9; uint64 s10; uint64 s11;

};

这个结构体应该由进程本身保存和维护，因此要添加到进程结构体中。

struct uthread\_context context;*/\* thread context \*/*

};

*/\* FREE, RUNNING, RUNNABLE \*/*

state;

stack[STACK\_SIZE]; */\* the thread's stack \*/*

char

int

struct thread {

在进程切换时，只需要交换调用方保存的进程上下文。ra 寄存器代表的是返回地址（return address），sp 寄存器代表的是栈底指针（stack pointer），这两个寄存器并不需要在切换进程时交换，而是由系统更新。

thread\_switch: 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)

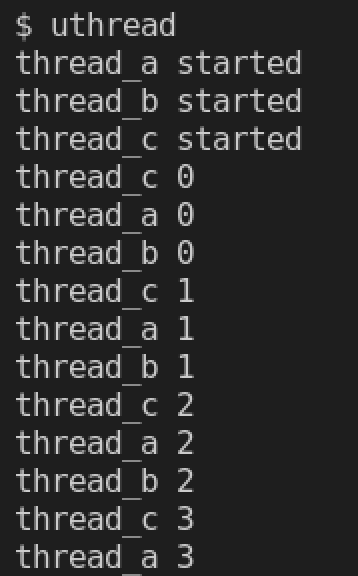
sd s10, 96(a0)

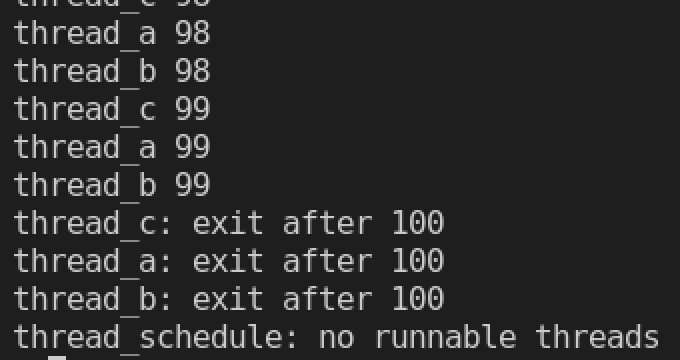
sd s11, 104(a0)

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 \*/

按照说明运行 uthread 测试，得到如下结果：



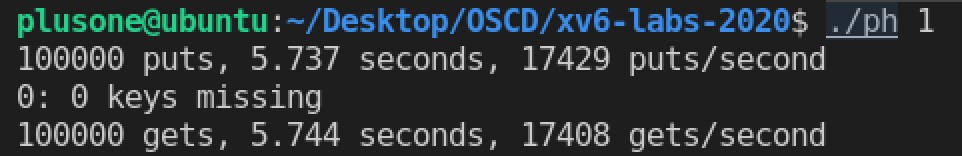


这与我的预期相符。

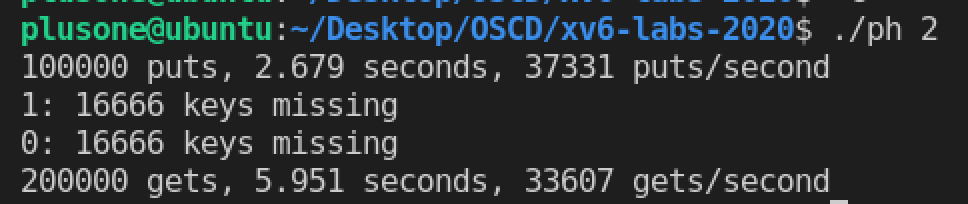
### 使用 UNIX pthread 线程库实现一个线程安全的哈希表

本实验与 xv6 系统代码无关，是一个使用 Unix 环境 pthread 库来学习多线程编程的练习。

按照说明，先运行指令 make ph，之后运行./ph 1，结果如下：



接下来运行./ph 2，发现有丢失的键，说明实现有误。



这个实验的目的是通过该测试且没有丢失的键。同时该实验应该保证一定的效率，即在双线程运行时可以达到 2 倍的效率（假设处理器核数大于二）。

解决该问题需要考虑为何丢失了一些键。因为同时有多个线程工作，所以当一个进程往哈希表中填入一个键时，可能有其他进程也希望填入一个键。如果两个键的哈希值不相等，那么插入是没有问题的，两个键都可以成功在不同的哈希筒中出现。但是如果两个键哈希值相同，那么其插入的位置一样，就会导致其中一个（相对后写入的）将另一个新插入的键覆盖，从而导致了键的丢失。

解决键丢失的问题，需要采用上锁的策略。一个简单的思想是每次只允许一个进程对于整个哈希表操作，但是这个想法存在明显缺陷：多进程应该效率更高， 如果一次只允许一个进程，那么多进程和单进程的并行性相等，且多进程因为进程切换等操作会浪费更多时间。正确的做法是分别加锁：对于每个哈希筒，每次只允许一个进程访问。以下是实现步骤：

1. 声明锁数组，并在 main 函数中初始化。

notxv6/ph.c

pthread\_mutex\_t locks[NBUCKET];

for (int i = 0; i < NBUCKET; i++) {

**pthread\_mutex\_init**(&locks[i], NULL);

}

1. 在 **put** 函数中使用锁

if(e){

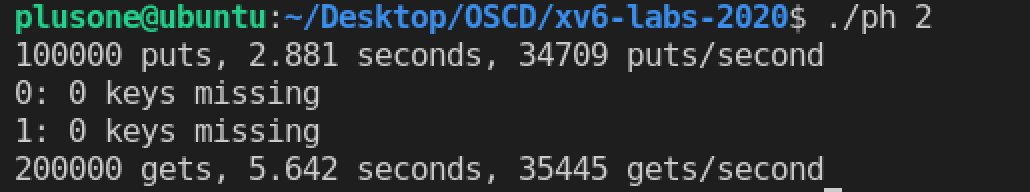
*// update the existing key.*

e->value = value;

} else {

*// the key is new.* **pthread\_mutex\_lock**(&locks[i]); **insert**(key, value, &table[i], table[i]); **pthread\_mutex\_unlock**(&locks[i]);

}

完成修改后运行./ph 2，没有丢失的键，结果如下：

### 实现 barrier 函数

**barrier** 函数会在所有进程都调用 **barrier** 时才解锁所有进程。在实现开始前，发现已有定义的数据结构 bstate：

*round of the barrier*

int round; *// Barrier round*

} bstate;

*// Number of threads that have reached this*

int nthread;

struct barrier {

pthread\_mutex\_t barrier\_mutex; pthread\_cond\_t barrier\_cond;

通过比较 bstate.nthread 与全局变量 nthread，可以分别判断是否使进程等待或者唤醒其他进程。

库函数 **pthread\_cond\_wait**(&cond, &mutex); 使进程释放mutex 锁并进入睡眠，等待 cond 将其唤醒。**pthread\_cond\_broadcast**(&cond); 将所有等待 cond 的进程唤醒。

因此，**barrier** 函数的实现如下：

static void

**barrier**()

{

*// YOUR CODE HERE*

*//*

*// Block until all threads have called barrier() and*

*// then increment bstate.round.* **pthread\_mutex\_lock**(&bstate.barrier\_mutex); bstate.nthread += 1;

if (bstate.nthread == nthread) { bstate.round += 1;

bstate.nthread = 0;

**pthread\_cond\_broadcast**(&bstate.barrier\_cond);

} else {

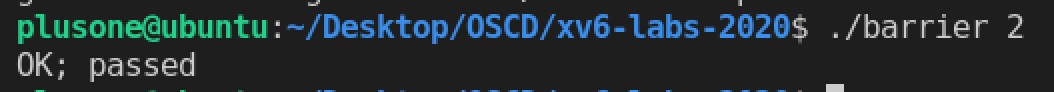
*// 还有进程没有同步，进入等待*

**pthread\_cond\_wait**(&bstate.barrier\_cond, &bstate.barrier\_mutex);

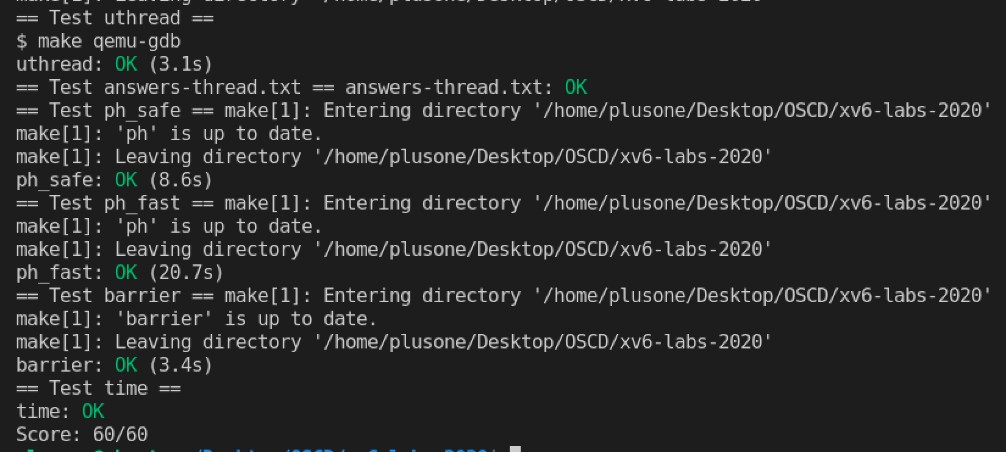
}

**pthread\_mutex\_unlock**(&bstate.barrier\_mutex);

}

编译运行 **barrier** 并测试，测试通过：

### 测试成绩

通过 make grade 进行本实验综合测试：

至此，本实验全部完成。

## 实验小结

本实验是关于多线程编程的。之前并没有编写过多线程的程序，只知道多线程程序可以运用多个处理器，可以达到提高速度的效果。在操作系统课程中介绍了锁的概念和几个模型，并讨论了多线程存在的死锁问题。在实验中，实现了进程切换、运用锁来完成线程间的互斥（多线程哈希表）和运用锁和条件信号量来完成多线程之间的同步（barrier），三个小实验加深了我对于多线程的理解，提供了练习多线程编程的机会。

# 实验八 Locks

## 实验目的

1. 理解锁竞争，包括锁竞争产生的原因以及解决方法。
2. 理解通过增加锁的数量来降低锁竞争的原理，并应用到程序中。
3. 加深对于使用锁来达到进程互斥方法的掌握程度。

## 实验内容

1. 重新设计系统管理内存的方式，实现每个 CPU 管理一个空闲链表和一个锁。2. 修改系统的 IO 缓冲区，通过散列的方式完成对缓冲区和锁的分割。

## 实验步骤

### 重新设计系统管理内存的方式

本实验的目的是对每个 CPU 拆分一个空闲空间链表并用单独的锁管理，从而降低锁冲突带来的额外开销，增加并行性。

实验的要点是增加锁和空闲链表的数量并正确地初始化，然后在函数中对不同的 CPU 使用不同的锁-链表组合，完成资源和锁的分割。按照要求，一个 CPU 的空闲链表为空时，需要向别的 CPU“借”一些空间以保证所有空间都能正常分配。

以下是实现步骤：

1.声明符合 CPU 个数的锁-链表结构，并在函数 **kinit** 中正确初始化。

kalloc.c

struct {

struct spinlock lock; struct run \*freelist;

} kmem[NCPU];

其中 NCPU 在 param.h 中声明，值为 8。

void

**kinit**()

{

char buf[10];

for (int i = 0; i < NCPU; i++)

{

**snprintf**(buf, 10, "kmem\_CPU%d", i);

**initlock**(&kmem[i].lock, buf);

}

**freerange**(end, (void\*)PHYSTOP);

}

1. 在函数中，需要完成对 CPU 维护的锁-链表结构申请空间。在 CPU 空间不足时，需要从其他 CPU 维护的结构中“借”一些空间来完成这次的分配。

首先要获取 CPU 的 ID，在这一个过程中不能允许进程被其他进程抢占，所以要调用 **push\_off** 和 **pop\_off** 函数来关闭/打开中断：

**push\_off**();

int cpu = **cpuid**(); **pop\_off**();

之后是分配过程的实现：

**acquire**(&kmem[cpu].lock); r = kmem[cpu].freelist; if(r)

kmem[cpu].freelist = r->next; else *// 从其他 CPU 借用空间*

{

struct run\* tmp;

for (int i = 0; i < NCPU; ++i)

{

if (i == cpu) continue; **acquire**(&kmem[i].lock); tmp = kmem[i].freelist; if (tmp == 0) {

**release**(&kmem[i].lock); continue;

} else {

for (int j = 0; j < 1024; j++) {

*// 借 用 1024 页*

if (tmp->next) tmp = tmp->next;

else

break;

}

kmem[cpu].freelist = kmem[i].freelist; kmem[i].freelist = tmp->next;

tmp->next = 0; **release**(&kmem[i].lock); break;

}

}

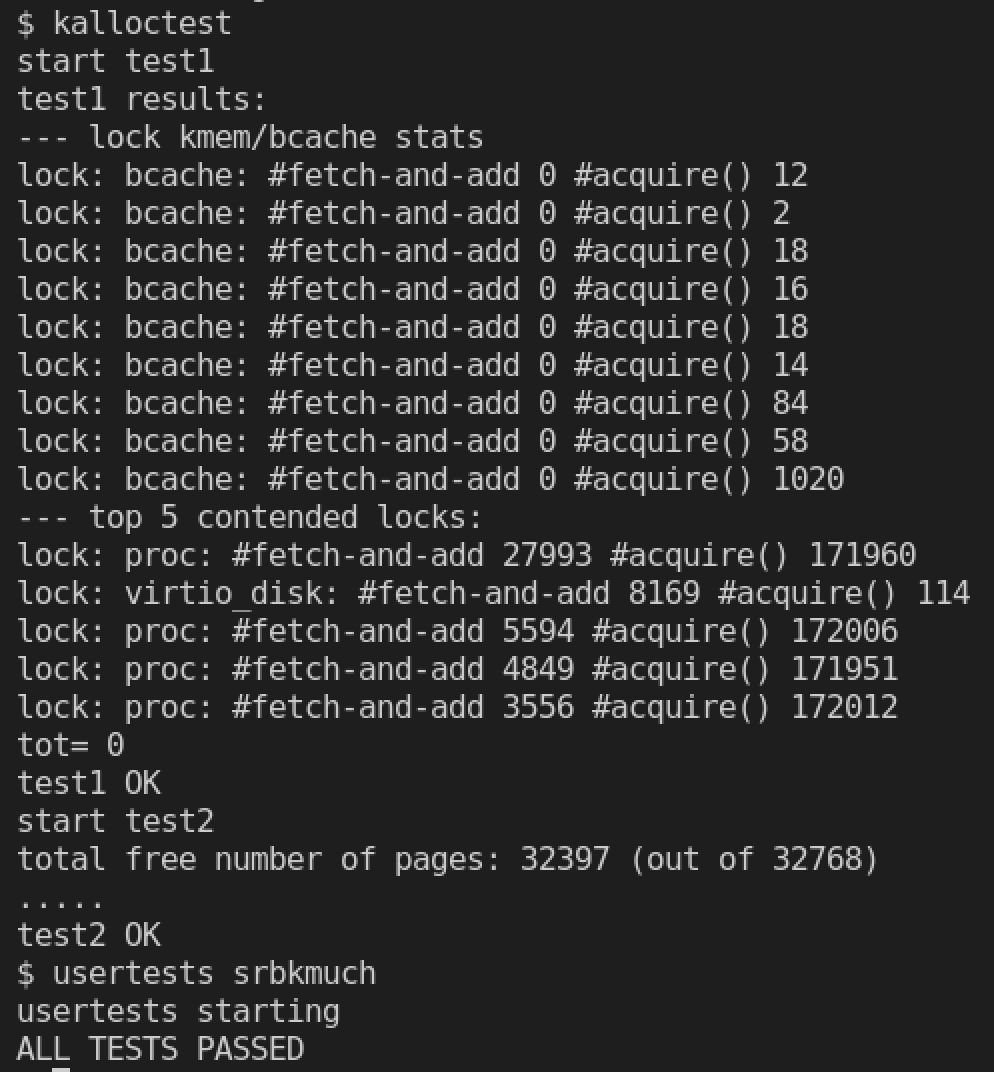
r = kmem[cpu].freelist; if (r)

kmem[cpu].freelist = r->next;

}

**release**(&kmem[cpu].lock);

完成以上步骤后，运行测试显示通过：



### 修改系统的 IO 缓冲区

该实验也是要对资源进行分割，因为 IO 缓冲区是所有 CPU、进程共享的资源且在每次使用时都会刷新。但是根据局部性原理，某个进程载入缓冲区的数据有可能经常复用，直接覆盖的方式并不合理。因此实验堆间的修改方式是针对磁盘空间中的**块号**进行散列处理，并用哈希表来维护缓冲区。

以下是实验的步骤：

1.首先要决定哈希筒的个数，一个合适大小的质数是一个理想的选择，因此我依照提示选择了 13。

param.h

#define **NBUCKET** 13

1. 接下来需要修改原先 buffer cache 的定义，以适应使用散列表的方式。我用数组装填了缓冲区的头部、筒对应的磁盘块以及锁。

bio.c

struct {

struct spinlock lock; struct buf head[NBUCKET];

struct buf hash[NBUCKET][NBUF];

struct spinlock hashlock[NBUCKET]; *// lock per bucket*

} bcache;

此外，这些值需要被正确地初始化：

void

**binit**(void)

{

struct buf \*b;

**initlock**(&bcache.lock, "bcache"); for (int i = 0; i < NBUCKET; i++) {

**initlock**(&bcache.hashlock[i], "bcache");

*// Create linked list of buffers* bcache.head[i].prev = &bcache.head[i]; bcache.head[i].next = &bcache.head[i];

for(b = bcache.hash[i]; b < bcache.hash[i]+NBUF; b++){

b->next = bcache.head[i].next; b->prev = &bcache.head[i];

**initsleeplock**(&b->lock, "buffer"); bcache.head[i].next->prev = b; bcache.head[i].next = b;

}

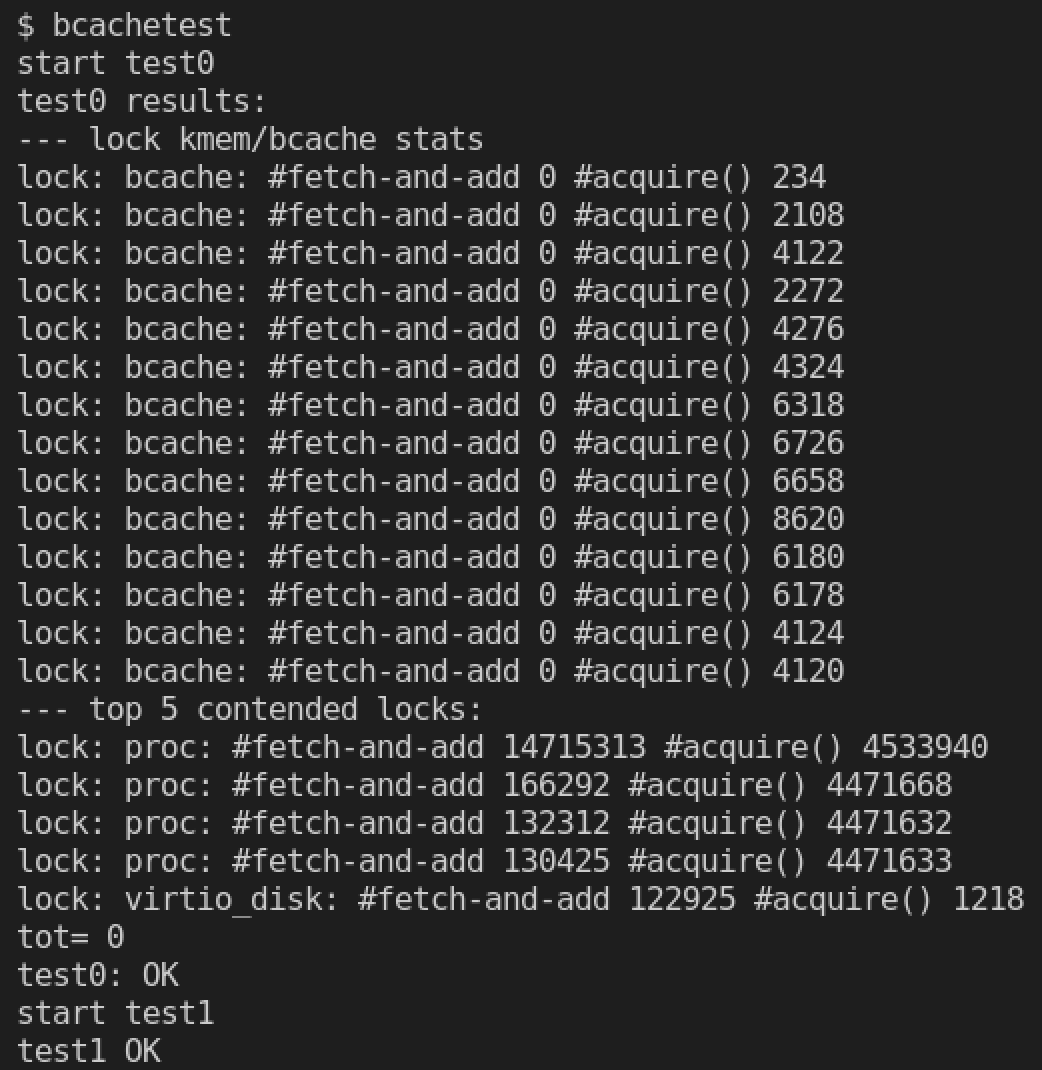
}

}

3.每次访问的缓冲区可以用 blockno 算出：

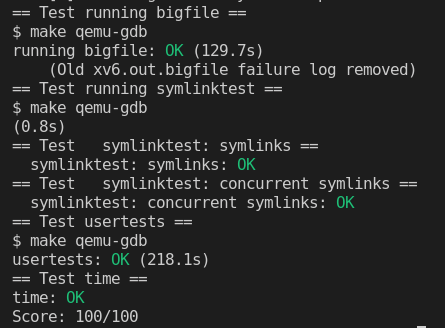
uint hashcode = b->blockno % NBUCKET;

在 **bget** 和 **brelse** 函数中，只需要将所有 bcache.lock 改为bcache.hashlock[hashcode]，把 bcache.head 改为bcache.head[hashcode]，即可完成该实验。

1. 运行的实验结果如下：

### 测试成绩

通过 make grade 进行本实验综合测试：



## 实验小结

实验 7 与本实验都是关于并行编程中问题的性能优化。用户可以实现用户级的多线程程序来更好地使用处理器资源，而分时系统通过多个进程的并发运行可以让多个程序看起来像同时在执行。在处理多线程或多进程中资源抢占而导致的锁竞争现象，常用的做法是通过分割资源并且分别加锁。锁的数量越多，单个锁上的冲突就越少（但是也要考虑锁的数量增多的负面影响）。

我在两个实验中分别体会了控制进程的同步/互斥（实验 7）和分割资源来使锁竞争得到缓解、提高资源并行利用率。实验同时还涉及到了关中断、散列、资源分配等其他方面的细节，关联了我之前课程中学习的知识。从功能性角度来说，一个操作系统只需要拥有运行程序的能力，与用户交互的系统也最多需要增加并发运行这一个功能限制。而在实验中通过对资源的划分实现了提高系统效率和资源利用率这些非功能性需求，在一个面向现实世界的操作系统中，非功能性需求一样有重要地位。

# 实验九 File System

## 实验目的

1. 理解 UNIX 系统以及 xv6 系统组织文件的方式，包括 inode 数据结构以及地址的映射关系。
2. 掌握通过增加索引级数的文索引扩展方式。
3. 理解文件映射和打开的方式，了解 xv6 系统的文件系统层级架构。
4. 理解硬链接和软链接的实现方式。

## 实验内容

1. 修改文件系统使其支持更大的文件存储。具体做法是实现二级间接的索引节点（inode）。
2. 增加一个通过符号链接的系统调用。将目标符号与系统路径链接，使对符号文件的操作都同步到对相应路径文件的操作。

## 实验步骤

### 修改文件系统使其支持更大的文件存储

实验开始之前，需要按照指示阅读 xv6 系统中文件系统相关的描述以及 fs.c

的源代码。

在 xv6 中，文件系统是层级结构的，总共有 7 个层级：

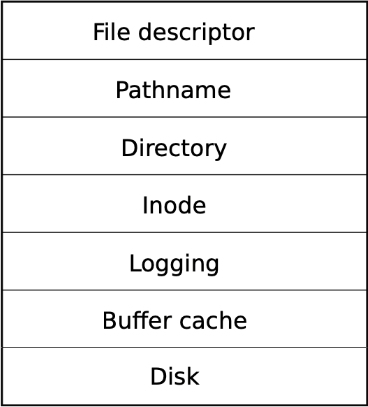


图 0-1 摘自xv6-book 的文件系统层级图

其中，最底层 Disk 是物理磁盘区域，剩下的层将磁盘封装成对应操作系统中的数据结构。对于磁盘文件的读写发生在 Buffer Cache 层，更高层则通过接口操作内存中的文件缓存。

我需要关心的是 Inode 层中的内存哪文件组织。这是一个 inode 数据结构的示意图：

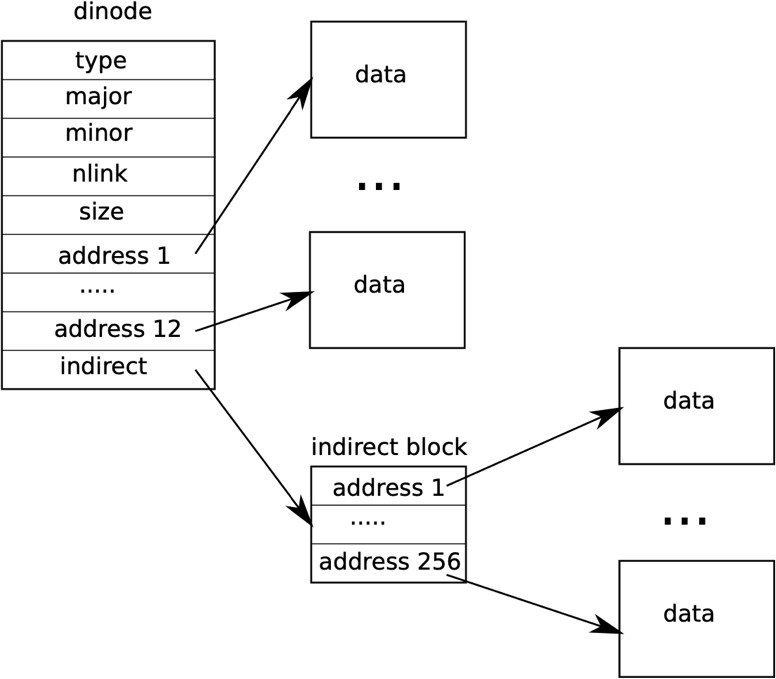


图 0-2 摘自xv6-book 的 inode 结构示意图

在这个示意图中，inode 支持了一级间接的块数据，inode 本身储存 12 个块地址，因此可以索引 268（=12+256）个文件块。本实验的目的是将一个直接索引节点改为二级间接索引（它可以索引的文件块数为 256\*256 个），使一个文件的总大小可以达到 65803（=256\*256+256+11）个。

与该实验相关的代码为 fs.c 中的 inode 定义以及用块号查找对应内存地址的函数。Inode 数据结构的定义如下：

uint addrs[NDIRECT+1]; *// Data block addresses*

};

*// Size of file (bytes)*

*// File type*

*// Major device number (T\_DEVICE only)*

*// Minor device number (T\_DEVICE only)*

*// Number of links to inode in file*

short type;

short major; short minor; short nlink;

*system*

uint size;

*// On-disk inode structure*

struct **dinode** {

需要注意到，dinode 是磁盘内inode 的数据结构，内存内inode 的定义比dinode

了一些操作系统维护的数据。

在 fs.h 中规定了直接索引地址的数量与非直接索引节点对应的地址个数：

#define **NDIRECT** 12

#define **NINDIRECT** (BSIZE / sizeof(uint))

将 inode 中某个块对应的内存地址返回的函数是 bmap，它检查输入的的块号，若小于直接索引的块号便直接从 addrs 数组中取数据，否则用 addrs 数组的最后一个地址索引的缓冲区中的 data 字段当作 256 个间接地址字段，返回对应的地址。

static uint

**bmap**(struct inode \*ip, uint bn)

{

uint addr, \*a; struct buf \*bp; if(bn < NDIRECT){

if((addr = ip->addrs[bn]) == 0)

ip->addrs[bn] = addr = **balloc**(ip->dev); return addr;

}

bn -= NDIRECT; if(bn < NINDIRECT){

*// Load indirect block, allocating if necessary.*

if((addr = ip->addrs[NDIRECT]) == 0)

ip->addrs[NDIRECT] = addr = **balloc**(ip->dev); 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");

}

该实验需要改动的地方是 inode 的定义以及增加 bmap 对二级索引的支持。具体步骤如下：

* + - 1. 减少直接索引数量和增加二级索引相关宏定义。

fs.h

#define **NDIRECT** 11

#define **NINDIRECT** (BSIZE / sizeof(uint)) #define **NDOUBLYINDIRECT** (NINDIRECT \* NINDIRECT)

#define **MAXFILE** (NDIRECT + NINDIRECT + NDOUBLYINDIRECT)

* + - 1. 修改 inode 和 dinode 中的字段 addrs 的定义。

fs.h, file.h

uint addrs[NDIRECT+2]; *// Data block addresses*

* + - 1. 在 bmap 函数中添加一段处理映射到二级索引的代码。

bn -= NDIRECT;

if(bn < NINDIRECT){

*// Load indirect block, allocating if necessary.*

if((addr = ip->addrs[NDIRECT]) == 0)

ip->addrs[NDIRECT] = addr = **balloc**(ip->dev); 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;

}

bn -= NINDIRECT;

uint indirect\_index, offset; if (bn < NDOUBLYINDIRECT) {

if ((addr = ip->addrs[NDIRECT+1]) == 0)

ip->addrs[NDIRECT+1] = addr = **balloc**(ip->dev); indirect\_index = bn / NINDIRECT;

offset = bn % NINDIRECT;

bp = **bread**(ip->dev, addr); a = (uint\*)bp->data;

if ((addr = a[indirect\_index]) == 0) { a[indirect\_index] = addr = **balloc**(ip->dev); **log\_write**(bp);

}

**brelse**(bp);

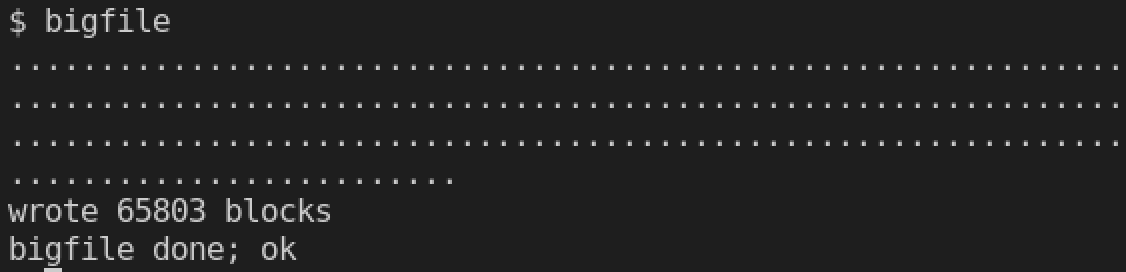
bp2 = **bread**(ip->dev, addr); a = (uint\*)bp2->data;

if ((addr = a[offset]) == 0) { a[offset] = addr = **balloc**(ip->dev); **log\_write**(bp2);

}

**brelse**(bp2); return addr;

}

* + - 1. 完成实验，运行测试。

### 增加一个通过符号链接的系统调用

Xv6 系统本身提供了通过 link 系统调用实现的硬链接：在硬链接文件 B 到文件 A 时，系统将文件 B 的 inode 中 inum 字段设置成与 A 一样，从而产生一种两个文件映射到同一块磁盘空间的效果。而符号链接是一种软链接，它的作用是产生一个特殊的链接文件类型并且在打开文件时索引到它所链接的文件。软链接类似于常用操作系统中的制作快捷打开方式（make alias）功能。

要增加符号链接系统调用，首先需要增加一种链接文件类型，然后在创建符号链接时，系统创建类型为链接文件的链接文件，并且将其的字段填写进链接目标路径。

对于打开该链接文件时的操作应该要考虑递归打开链接文件的情况（因为允许链接到符号链接文件），当递归打开超过一定深度时需要直接返回并且报错。最后系统打开的文件应该是一个非链接文件（最终的目标文件）。

实验的具体步骤如下：

* + - 1. 创建一个新的系统调用。

kernel/syscall.h

#define **SYS\_symlink** 26

kernel/syscall.c

extern uint64 **sys\_symlink**(void);

static **uint64** (\*syscalls[])(void) = {

…

[SYS\_symlink] sys\_symlink,

};

user/user.h

int **symlink**(char\*, char\*);

user/usys.pl

entry("symlink");

* + - 1. 添加对应的宏，支持符号链接文件类型和不跟随打开模式。

Stat.h

#define **T\_SYMLINK** 4

fcntl.h

#define **O\_NOFOLLOW** 0x800

* + - 1. 添加并实现系统调用 sys\_symlink 函数。其逻辑是：读取两个字符串作为参数，一个是链接到的目标文件路径 target，另一个是新建链接的路径 path。在路径下创建一个新文件并且将目标文件名填入新文件中。

sysfile.c

uint64 **sys\_symlink**(void) {

char target[MAXPATH], path[MAXPATH]; struct inode \*ip;

int n;

if ((n = **argstr**(0, target, MAXPATH))< 0 || **argstr**(1, path, MAXPATH) < 0)

return -1;

**begin\_op**();

ip = **create**(path, T\_SYMLINK, 0, 0); if (ip == 0) {

**end\_op**(); return -1;

}

if(**writei**(ip, 0, (uint64)target, 0, n) != n) {

**end\_op**(); return -1;

}

**iunlockput**(ip); **end\_op**(); return 0;

}

* + - 1. 修改 open 系统调用，以处理打开符号链接文件。打开符号链接文件时递归打开其指向的最终目标文件。以下代码位于 sys\_open 函数中。

sysfile.c

if (ip->type == T\_SYMLINK && !(omode & O\_NOFOLLOW)) { int end\_recursion = 10;

while (ip->type == T\_SYMLINK && end\_recursion > 0) {

if (**readi**(ip, 0, (uint64)path, 0, ip->size) != ip->size) {

**iunlockput**(ip); **end\_op**(); return -1;

}

**iunlockput**(ip);

if ((ip = **namei**(path)) == 0) {

**end\_op**(); return -1;

}

**ilock**(ip); end\_recursion--;

}

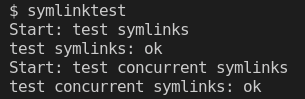
if (end\_recursion == 0) { **iunlockput**(ip); **end\_op**();

return -1;

}

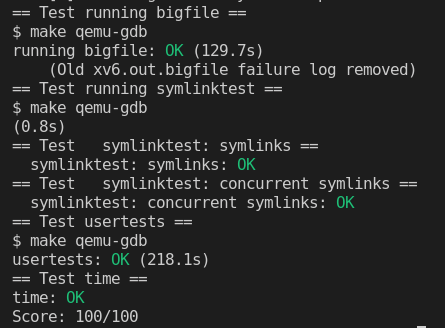
}

* + - 1. 完成修改，运行测试。



### 测试成绩

运行 make grade 查看实验成绩，全部通过。



至此，实验九全部完成。

## 实验小结

本次试验是关于 xv6 系统文件系统的，拓展了可以索引的文件大小，也创建了符号链接这种软链接方式。

第一个部分中的拓展文件大小是一个典型的程序数据结构设计思想，可以看作是对于一个高度为 1 的树，将其某一些节点拓展成指向相似的节点的非叶子结点，从而扩展这个数的容量（使其高度增加）。这种思想被运用在可拓展地址数的操作数设计（计算机组成原理课程中），以及多级页表的设计（操作系统课程中）。手动实现这一功能的过程十分有趣。

第二个部分是实现系统文件软链接相关的内容。这个部分在课程内要求不高， 在实现的过程中最大的困难在于理解“软链接也是一种文件”这一概念。这个实 验让我加深了对电脑上那些左下角有箭头符号的文件本质是什么这一问题理解。

# 实验十 Mmap

## 实验目的

1. 了解使用虚拟内存的原因和好处。
2. 理解创建和释放文件映射的方式。
3. 梳理文件内存映射的生命周期。

## 实验内容

添加 mmap 与 mumap 两个系统调用。前者将一个文件内存映射到进程的地址空间，后者取消已有地址空间的映射。本实验的实验内容只有一条，但是难度被标为困难（hard），因此我还是将实验步骤细分，这样更利于实验的理解和报告的组织。细分后的实验分为三步：

* + 1. 理解虚拟地址映射和访问的方式，了解 mmap 系统调用的实际工作流程。
    2. 实现 mmap 系统调用。3.实现 munmap 系统调用。4.修改其他相关的函数。

## 实验步骤

### 理解虚拟地址与 mmap

虚拟地址是虚拟内存的地址，虚拟内存的提出是为了解决多进程系统中同时运行多个程序的内存空间问题。直接使用物理地址的系统进程之间的地址空间不能隔离，并且可以同时运行的进程数量收到内存空间的限制。为了解决这个问题， 通过对进程地址空间的划分（分段或是分页），人们提出了一种在不使用某一部分进程数据时，就将其调到磁盘的内存管理策略，虚拟内存的技术也由此诞生。使用虚拟内存技术的操作系统中，进程地址空间都很大，并且允许不同进程使用相同的虚拟地址。在寻址时再对这些虚拟地址进行翻译转化。

mmap 允许用户将文件映射到进程的地址空间内，实现了文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文

件磁盘上。通过 mmap 的文件操作会更加高效，因为常规文件操作需要从磁盘到页缓存再到用户主存的两次数据拷贝；而 mmap 操控文件只需要从磁盘到用户主存的一次数据拷贝过程。

mmap 会记录映射的文件，并将文件信息填充到对应进程中。它并不会直接打开映射文件或是创建物理内存映射文件，而是在访问内存产生缺页中断时，再将文件内容懒加载到内存中。系统调用会在进程的地址空间中寻找一个并未使用的区域来映射文件，并将其指针填入进程记录的文件虚拟地址区域，供下次使用。

### 实现 mmap 系统调用

* + - 1. 添加系统调用。

kernel/syscall.h

#define **SYS\_mmap** 27

#define **SYS\_munmap** 28

kernel/syscall.c

extern uint64 **sys\_mmap**(void);

extern uint64 **sys\_munmap**(void);

static **uint64** (\*syscalls[])(void) = {

...

[SYS\_mmap] sys\_mmap, [SYS\_munmap] sys\_munmap,

};

user/user.h

void \***mmap**(void \*addr, uint length, int prot, int flags, int fd, uint offset);

int **munmap**(void \*addr, int length);

user/usys.pl

entry("mmap");

entry("munmap");

2.添加 vma 数据结构的定义，该结构记录地址，长度，权限，文件等由 mmap 创建的虚拟内存范围。定义后，给进程的数据结构添加一个 vmas 数组，并在allocproc 中初始化。

kernel/param.h

#define **NVMA** 16

kernel/proc.c

*// if address is valid*

*// the address of the vma*

*// length of mapped memory*

*// access guard of memory*

*// shared or private*

struct **vma**

{

int valid; uint64 addr; int length; int prot; int flags;

struct **file** \*mapfile;

};

*// process virtual memory area*

struct **proc** {

struct **vma** vmas[NVMA];

};

static struct proc\*

**allocproc**(void)

{

...

found:

*// init vmas array*

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

}

...

}

1. 添加 mmap 系统调用的函数体。首先需要读取参数，之后在当前进程的 vmas

字段中寻找一个空位，填充其对应的字段，就完成了记录该映射文件的工作。

kernel/sysfile.c

uint64

**sys\_mmap**(void) { uint64 addr = 0; uint length;

int prot; int flags; int fd;

uint offset;

if (**argaddr**(0, &addr) < 0 || **argint**(1, (int \*)&length) < 0

|| **argint**(2, &prot) < 0 || **argint**(3, &flags) < 0

|| **argint**(4, &fd) < 0 || **argint**(5, (int \*)&offset)) { return -1;

}

*// doing lazy things here.*

struct proc\* p = **myproc**();

struct file\* mapfile = p->ofile[fd];

if ((!mapfile->writable) && (prot & PROT\_WRITE) && (!(flags & MAP\_PRIVATE)))

return -1;

*// traverse process's VMAs to find empty ones*

int i;

for (i = 0; i < NVMA; i++) { if (!p->vmas[i].valid) {

p->vmas[i].valid = 1;

p->vmas[i].prot = prot;

p->vmas[i].addr = addr = p->sz; p->vmas[i].length = length;

p->vmas[i].flags = flags;

p->vmas[i].mapfile = mapfile;

**filedup**(p->ofile[fd]); break;

}

}

if (i == NVMA)

return -1;

p->sz += length; return addr;

}

1. 由于使用了懒加载方式，在访问未加载界面时，会产生一个缺页中断。我需要在 usertrap 函数中处理该中断来正确地加载页面。首先，缺页中断的地址会作为 r\_scause 的返回值；获取地址后，查找进程 vmas 字段中包含该地址的虚拟地址块，并对该地址所在的页面分配一个页大小的空间。

kernel/trap.c

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

*// ok*

} else if (**r\_scause**() == 13 || **r\_scause**() == 15) { uint64 va = **r\_stval**();

if (va >= p->sz || va < p->trapframe->sp) { p->killed = 1;

} else { int i;

for (i = 0; i < NVMA; i++) { if (p->vmas[i].valid) {

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

break;

}

}

if (i == NVMA) {

p->killed = 1;

} else {

uint64 ka = (uint64) **kalloc**(); if (ka == 0){

p->killed = 1;

} else {

**memset**((void \*)ka, 0, PGSIZE); va = **PGROUNDDOWN**(va);

**ilock**(p->vmas[i].mapfile->ip);

**readi**(p->vmas[i].mapfile->ip,

p->vmas[i].addr, PGSIZE);

**iunlock**(p->vmas[i].mapfile->ip);

0,

ka,

va

-

uint64 pm = PTE\_U;

if (p->vmas[i].prot & PROT\_READ) pm |= PTE\_R;

if (p->vmas[i].prot & PROT\_WRITE) pm |= PTE\_W;

if (**mappages**(p->pagetable, va, PGSIZE, ka, pm) != 0) {

**kfree**((void \*)ka); p->killed = 1;

}

}

}

}

} else {

**printf**("usertrap(): **r\_scause**(), p->pid);

unexpected

scause

%p

pid=%d\n",

**printf**(" sepc=%p stval=%p\n", **r\_sepc**(), **r\_stval**());

p->killed = 1;

}

5. 中断处理的懒加载方式允许了缺页错误的产生，因此在 uvmcopy 和

uvmunmap 中需要删去对没有映射页面的报错。

kernel/vm.c

if((\*pte & PTE\_V) == 0)

*// panic("uvmcopy: page not present");*

continue;

if((\*pte & PTE\_V) == 0)

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

continue;

### 实现 munmap 系统调用

1.添加系统调用，该步骤已经在 10.3.2 中完成。

2.实现 munmap 系统调用。首先要获取系统调用的参数：取消映射的地址及长度。然后检查当前进程的 vmas 中有没有包含该地址的记录。对于找到的 vmas 记录，若标记为 MAP\_SHARED，则需要将文件块直接写回到磁盘。之后对于整个映射释放和释放一部分映射的操作分别完成释放。

kernel/sysfile.c

uint64

**sys\_munmap**(void) { uint64 addr;

int length;

if (**argaddr**(0, &addr) < 0 || **argint**(1, &length) < 0) return -1;

struct proc\*p = **myproc**(); int i;

for (i = 0; i < NVMA; i++) { if (p->vmas[i].valid) {

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

break;

}

}

if (i == NVMA)

return -1;

struct vma \*vmap = &p->vmas[i]; if (vmap->flags & MAP\_SHARED) {

**filewrite**(vmap->mapfile, addr, length);

}

if (vmap->addr == addr && vmap->length == length) {

**uvmunmap**(p->pagetable, addr, length / PGSIZE, 1);

**fileclose**(vmap->mapfile); vmap->length -= length;

} else if (vmap->addr + vmap->length == addr + length) {

**uvmunmap**(p->pagetable, addr, length / PGSIZE, 1); vmap->length -= length;

}

return 0;

}

### 修改其他相关函数

* + - 1. 修改 fork 函数，使子进程复制父进程的文件映射。

kernel/proc.c

int

**fork**(void)

{

...

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

np->vmas[i].valid = 1;

np->vmas[i].addr = p->vmas[i].addr;

np->vmas[i].length = p->vmas[i].length; np->vmas[i].prot = p->vmas[i].prot;

np->vmas[i].flags = p->vmas[i].flags;

np->vmas[i].mapfile = p->vmas[i].mapfile;

}

}

...

}

2.修改 exit 函数，当退出进程时写回并释放相应的文件映射。

void

**exit**(int status)

{

...

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

if (p->vmas[i].flags & MAP\_SHARED)

**filewrite**(p->vmas[i].mapfile,p->vmas[i].addr, p->vmas[i].length);

**uvmunmap**(p->pagetable, p->vmas[i].addr, p->vmas[i].length/PGSIZE, 1);

**filedup**(p->vmas->mapfile);

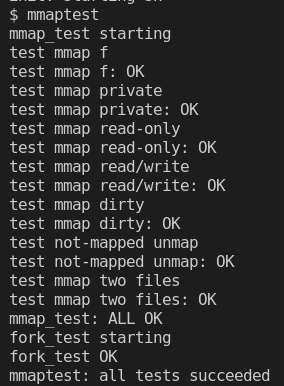
}}

...

}

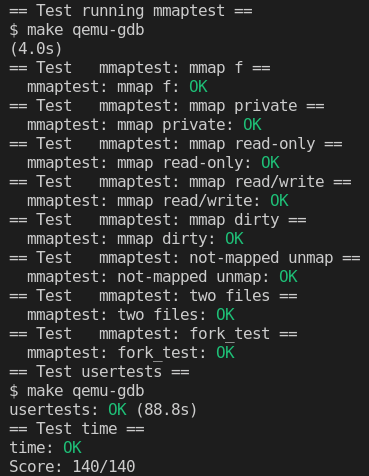
### 10.3.4 运行测试

1. 运行 mmaptest，得到结果如下：



测试通过。同样运行 usertests 也全部通过。

1. 运行 make grade，实验通过。



至此，该实验全部完成。

## 实验小结

本次实验也是关于系统与文件交互的内容，通过实验，实现了一个磁盘文件到虚拟内存地址的映射方式。这次实验的内容较多，覆盖了 xv6 系统中虚拟内存、文件地址、懒加载等知识点。在实验的提示中详细地列举了需要完成的对源代码的修改和添加，但是数目很多，稍有遗漏就会产生难以排查的 bug。

实验中需要调用的系统函数众多，这也是在实现两个系统调用时比较难以理解的部分。例如在映射一个页面时需要调用 mappages，而在取消映射关系时则会调用 uvmunmap 函数。此外，该实验涉及到了许多关于文件的操作，都让阅读了很多相关的源码和 xv6-book 相应的章节。

这个实验让我印象最深刻的地方在于对于页面懒加载的实现以及vma 数据结构的设计。懒加载这种方式在实验 5（Lazy allocation）和实验 6（Copy on write）中都有涉及，中心思想是不能一次性加载、拷贝所有内存，因为大多时候用不到。这种方式的泛用性印证了懒加载思想的重要地位。而在设计 vma 数据结构时，除了记录地址、长度、权限、文件等描述中提醒的字段之外，还记录了该内存映射地址是否正在使用、文件的共享标记这两个必要的字段。提出这两个字段是开始编写 mmap 系统调用时发现实现中有缺漏或是困难，才发现有字段缺失。

本次试验做起来阻力较大，但是收获也很多。我认为这次实验让我对操作系统中的内存设计和组织理解更加深刻。