同濟大學
TONGJI UNIVERSITY

Xv6 实验报告

院　　系：软件学院

专　　业：软件工程

姓　　名：赵鹏振

学　　号：2151516

2023 年 8 月 6 日

目录

1 实验一 Xv6 and Unix utilities

1.1 实验目的

1. 熟悉 xv6 以及其系统调用
2. 了解 git 的基本用法
3. 通过阅读 xv6-book 了解进程和内存
4. 掌握I/O、文件描述符、管道和文件系统的相关知识

1.2 实验内容

1.2.1 Boot xv6（easy）

掌握如何运行程序并且判断其是否正确，了解 xv6 的运行指令。

1.2.2 sleep（easy）

为 xv6 实现 UNIX 程序 sleep；您的睡眠应该暂停用户指定的节拍数。tick 是 xv6 内核定义的 时间概念，即计时器芯片两次中断之间的时间。解决方案应该在 user/sleep.c 文件中。

1.2.3 pingpong (easy)

编写一个程序，使用 UNIX 系统调用在一对管道上的两个进程之间“乒乓”一个字节，每个方向一个。父级应向子级发送一个字节；子进程应打印“<pid>：received ping”，其中<pid>是其进程 ID，将管道上的字节写入父进程，然后退出；父级应从子级读取字节，打印“<pid>：received pong”，然后退出。您的解决方案应该在 user/pingpong.c 文件中。

1.2.4 primes (moderate)/(hard)

使用管道编写 prime sieve 的并发版本。解决方案应该在 user/primes.c 文件中。目标是使用管道和叉来设置管道。第一个进程将数字 2 到 35 输入到管道中。对于每个素数，安排创建一个进程，该进程通过一个管道从其左邻居读取数据，并通过另一个管道向其右邻居写入数据。由于 xv6 的文件描述符和进程数量有限，因此第一个进程可以在 35 处停止。

1.2.5 find（moderate）

编写一个简单版本的 UNIX 查找程序：查找目录树中具有特定名称的所有文件。解决方案应该在 user/find.c 文件中。

1.2.6 xargs（moderate）

编写 UNIX xargs 程序的简单版本：从标准输入中读取行，并为每行运行一个命令，将该行作为参数提供给该命令。解决方案应该在 user/xargs.c 文件中。

1.2.7 submit

提交实验。

1.3 实验步骤

1.3.1 启动 xv6

（1）获取 xv6 资源并且切换到 util 分支
通过运行如下指令进行 clone 和切换分支

```
$ git clone git://g.csail.mit.edu/xv6-labs-2020
$ cd xv6-labs-2020
$ git checkout util
```

（2）查看git日志

```
$ git log
```

（3）尝试第一次提交

```
$ git commit -am 'my solution for util lab exercise 1'
```

（4）运行xv6

```
$ make qemu
```

（5）查看目录

```
$ ls
```

（6）退出xv6

```
Ctrl-a x
```

1.3.2 sleep

（1）在 kernel/sysproc.c 中查看 sleep 这一系统调用
（2）在所写的 sleep.c 中调用系统函数，当缺少参数或参数过多时报错，并且调用系统函数sleep，并在程序结尾添加 exit() 函数

```
int main(int argc,char **argv)
{
if(argc < 2)
{
    fprintf(2, "usage: system sleep...\n");
    exit(1);
}
if(argc>2)
{
    fprintf(2,"too many arguments!\n");
    exit(1);
}
sleep(atoi(argv[1]));
exit(0);
}
```

（3）将 sleep 程序添加到 Makefile 中的 UPROGS 中；完成后，make qemu 将编译程序，并能够从 xv6 shell 运行它
（4）运行 sleep

1.3.3 pingpong

（1）阅读xv6-book了解文件描述符所代表含义。其中，0表示标准输入（standard input），1表示标准输出（standard output），2表示标准错误（standard error）。

（2）调用read(fd,buf,n)从文件描述符fd中读取最多n个字节，将它们复制到buf中，并返回读取的字节数；调用write(fd,buf,n)将n个字节从buf写入文件描述符fd，并返回写入的字节数，只有在发生错误时，才会写入少于n个字节。

（3）管道是一个小型内核缓冲区，作为一对文件描述符公开给进程，一个用于读取，一个用于写入。将数据写入管道的一端，可以从管道的另一端读取该数据。管道为进程提供了一种通信方式。

（4）fork()函数的返回值为0时代表当前处于子进程中。

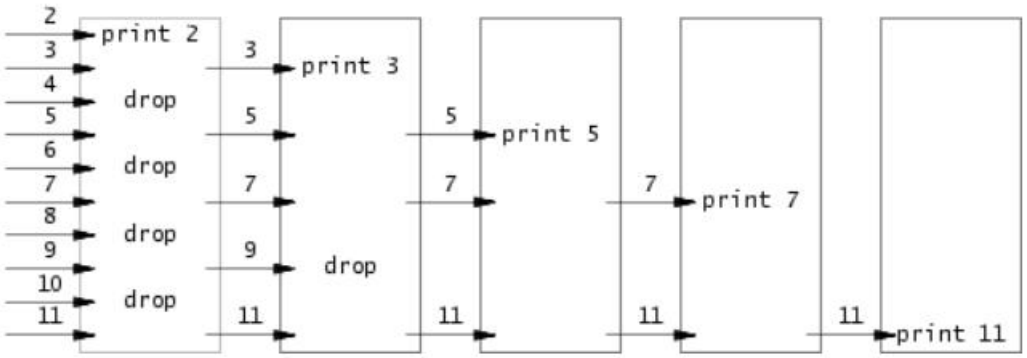
（5）当处于父进程时，将（任意）数据写入管道中并关闭p[1]，在数据写入后子进程从管道中读入并关闭p[0]，输出<pid>:received ping并将数据写入管道，作为子进程运行完的判断并关闭p[1]，当父进程等待写入完成后进行数据的读取，读取到后证明子进程已经运行完成，关闭p[0]并输出<pid>:received pong。

```
//parent process
if(fork())
{
    close(parent_fd[0]);
    write(parent_fd[1],"ping",4);
    close(child_fd[1]);
    read(child_fd[0],buff,4);
    printf("%d: received %s\n",getpid(),buff);
    exit(0);
}
//child process
else
{
    close(parent_fd[1]);
    read(parent_fd[0],buff,4);
    printf("%d: received %s\n",getpid(),buff);
    close(child_fd[0]);
    write(child_fd[1],"pong",4);
    exit(0);
}
```

（6）运行pingpong

1.3.4 Primes

（1）本次编写的素数筛原理如下图所示，从2开始进行遍历，并消去所有当前数字的倍数，直到完成筛选。



（2）通过递归实现素数筛，并及时关闭不需要的文件描述符，防止资源耗尽。

（3）运行primes

1.3.5 Find

（1）查看ls.c阅读如何进行程序的编写。

（2）参考ls.c的编写方法对find.c进行编写，不用对.和..进行判断，并将所有包含查询文件的路径输出

```
switch(st.type){
case T_FILE:
    //printf("File re: %s, fmtpath: %s\n", re, fmtname(path));
    if(match(re, fmtname(path)))
        printf("%s\n", path);
    break;
        //printf("%s %d %d %l\n", fmtname(path), st.type, st.ino, st.size);

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;
    //递归路径存储到buf中
        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0;
        if(stat(buf, &st) < 0)
    {
            printf("find: cannot stat %s\n", buf);
            continue;
        }
        char* lstname = fmtname(buf);
        if(strcmp(".", lstname) == 0 || strcmp("..", lstname) == 0)
    {
          //printf("%s %d %d %d\n", buf, st.type, st.ino, st.size);
          continue;
        }
        else
    {
          find(buf, re);
        }
    }
    break;
}
```

（3）运行find

1.3.6 xargs

（1）为实现xargs指令需要对字符串进行处理，从缓冲流中读入字符并进行分割处理

```
/* 以‘\n’分割的子串 */
void substring(char s[], char *sub, int pos, int len)
{
   int c = 0;
   while (c < len)
   {
      *(sub + c) = s[pos+c];
      c++;
   }
   *(sub + c) = '\0';
}

/* 截断 '\n' */
char* cutoffinput(char *buf)
{
    /* 为char *新分配一片地址空间，否则编译器默认指向同一片地址 */
    if(strlen(buf) > 1 && buf[strlen(buf) - 1] == '\n')
    {
        char *subbuff = (char*)malloc(sizeof(char) * (strlen(buf) - 1));
        substring(buf, subbuff, 0, strlen(buf) - 1);
        return subbuff;
    }
    else
    {
        char *subbuff = (char*)malloc(sizeof(char) * strlen(buf));
        strcpy(subbuff, buf);
        return subbuff;
    }
}
```

（2）运行xargs

1.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示。

```
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (2.4s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.6s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.0s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.0s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.0s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.1s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.1s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.0s)
```

1.5 实验困难和心得

本次实验对xv6系统进行了简单的介绍，并且通过各个实验对read、write等系统调用更加熟悉，了解了管道的基本用法等，逐渐从最基本的操作系统接口对操作系统进行了解。

在第二个实验里我对exit的理解不到位，必须了解到：父进程必须在所有子进程exit以后才可以exit，这是实验的要点。

在完成实验的过程中也遇到了一些困难，比如怎么进行素数筛的编写、如何通过read和write进行管道接收消息的判断等，我通过阅读xv6-book以及询问同学等方法对遇到的困难进行解决，最终完成了实验。

2 实验二 System Calls

2.1 实验目的

1. 进一步了解系统调用
2. 通过阅读xv6-book了解用户态、核心态相关知识并理解为什么这么设计

3. 尝试修改kernel中的系统调用并理解其运行过程

2.2 实验内容

2.2.1 System call tracing（moderate）

在本作业将添加一个系统调用跟踪功能，该功能可能在以后调试实验室时有所帮助。创建一个新的跟踪系统调用来控制跟踪。它应该有一个参数，一个整数“mask”，其位指定要跟踪的系统调用。例如，为了跟踪fork系统调用，程序调用trace（1<<SYS_fork），其中SYS_fork是kernel/syscall.h中的syscall编号。如果在mask中设置了系统调用的编号，则必须修改xv6内核，以便在每个系统调用即将返回时打印出一行。该行应包含进程id、系统调用的名称和返回值；不需要打印系统调用参数。跟踪系统调用应启用对调用它的进程及其随后派生的任何子进程的跟踪，但不应影响其他进程。

2.2.2 Sysinfo（moderate）

在此作业中，需要添加一个系统调用sysinfo，该调用收集有关正在运行的系统的信息。系统调用接受一个参数：指向结构sysinfo的指针（请参阅kernel/sysinfo.h）。内核应填写此结构的字段：freemem字段应设置为可用内存的字节数，nproc字段应设置为状态未使用的进程数。我们提供了一个测试程序sysinfotest；如果它打印“sysinfotest:OK”，则通过此项目。

2.2.3 submit

提交实验。

2.3 实验步骤

2.3.1 system call tracing

（1）为了实现trace功能需要修改proc.h中的porc结构体，添加mask变量表示每个进程的mask

（2）编写trace函数，参照sysproc.c中其他函数进行编写，给进程对应的mask变量赋值。

```
sys_trace(void)
{
  int trace_mask;
  // 取 a0 寄存器中的值返回给 mask
  if(argint(0, &trace_mask) < 0)
    return -1;

  // 把 mask 传给现有进程的 mask
  myproc()->trace_mask = trace_mask;
  return 0;
}
```

（3）在如下行中调用了syscalls这一数组，所以在此数组中对应地加入trace相关的内容。

```
132   [SYS_trace]   sys_trace,
133   [SYS_sysinfo] sys_sysinfo,
```

（4）为输出系统调用的名称，我们使用一个新的字符串数组来进行输出，完成后的syscall函数如下所示。

```
static char *syscall_names[] = {
  "", "fork", "exit", "wait", "pipe",
  "read", "kill", "exec", "fstat", "chdir",
  "dup", "getpid", "sbrk", "sleep", "uptime",
  "open", "write", "mknod", "unlink", "link",
  "mkdir", "close", "trace", "sysinfo",
};
```

```
  //start  从a7读取系统调用的编号，将1<<num与进程的tracemask比较，相等则打印
 if((1 << num) & p->trace_mask)
 {
    printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num], p->trapframe->a0);
 }
  //end
```

（5）在user.h以及usys.pl中进行修改。

（6）运行trace

2.3.2 sysinfo

（1）sysinfo为获得当前系统的可用进程数和可用内存。

（2）为获得空闲的物理内存即在kalloc.c中添加freemem函数统计可用内存。因为物理内存是采用链表存储当前空闲内存块，所以统计内存时遍历链表并且统计长度就可以得到空闲的内存。（注：需要乘以页面大小）

```
//返回空闲空间的byte数
uint64
free_mem(void)
{
  struct run *r;
  //nun用于存储空闲页数
  uint64 num = 0;
  //统计时对内存加锁
  acquire(&kmem.lock);
  //r指针指向空闲表
  r = kmem.freelist;
  //遍历空闲表
  while (r)
  {
    num++;
    r = r->next;
  }
  //统计完毕解锁
  release(&kmem.lock);
  return num * PGSIZE;
}
```

（3）为获得可用进程数，只需要遍历所有进程，即proc数组，并查看其state是否为UNUSED即可统计完成，代码写于proc.c

（4）在有以上两个函数后可以在sysproc.c中进行sys_sysinfo的实现，用freemem和unusedproc统计未使用的进程和内存，最后使用copyout方法进行复制，将内容复制回用户空间。

```
uint64
sys_sysinfo(void)
{
//addr是临时申请的一个空间，用于copyout
  uint64 addr;
  struct sysinfo info;
  struct proc *p = myproc();

  if (argaddr(0, &addr) < 0)
    return -1;
  info.freemem = free_mem();
  info.nproc = nproc();

  if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
    return -1;

  return 0;
}
```

（5）在两个统计函数实现后需要在defs.h中进行声明才能在sysproc.c中使用，同时需要按照如trace一样的方式对其他函数进行声明。

（6）运行sysinfo

2.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示。

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (2.0s)
    (Old xv6.out.trace_32_grep failure log removed)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.8s)
    (Old xv6.out.trace_all_grep failure log removed)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (12.1s)
    (Old xv6.out.trace_children failure log removed)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (1.5s)
    (Old xv6.out.sysinfotest failure log removed)
== Test time ==
time: OK
Score: 35/35
```

2.5 实验困难和心得

本次实验更深入的对系统调用进行了解和学习，在编写代码时不仅仅只需要修改一个文件，而是在多个文件中协同进行修改实现最终代码的运行。同时还需要阅读更多kernel中的代码并且了解其中的函数来学习如何编写实验的内容。

本次实验让我更加了解xv6的工作原理并且通过实验对操作系统的结构和概念掌握的更加深刻了。我对于实验提示中的一些参考的代码阅读起来较为困难，需要花时间把整个流程走一遍，边走边理解会好很多，需要从全局把握代码，这样理解才会更加深入。

3 实验三 Page Tables

3.1 实验目的

1. 了解页表机制及其结构
2. 了解内核虚拟地址和物理地址的映射关系
3. 通过实验自己尝试将用户映射添加到每个进程的内核页表上

3.2 实验内容

3.2.1Speed up system calls (easy)

有些操作系统（例如 Linux）通过在用户空间和内核之间共享数据的只读区域来加速某些系统调用。这样可以消除执行这些系统调用时的内核转换的需要。为了帮助学习如何向页表中插入映射，第一个任务是为 xv6 实现这种优化，以加速 getpid() 系统调用。当每个进程被创建时，在地址 USYSCALL 处（在 memlayout.h 中定义）映射一个只读页面。在这个页面的开头，存储一个 struct usyscall（也在 memlayout.h 中定义），并将其初始化以存储当前进程的 PID。对于此实验，用户空间已经提供了 ugetpid()，它将自动使用 USYSCALL映射。

3.2.2 Print a page table (easy)

定义一个名为vmprint（）的函数。它应该采用pagetable_t参数，并以下面描述的格式打印该pagetable。在返回argc之前的exec.c中插入if（p->pid==1）vmprint（p->pagetable），以打印第一个进程的页面表。当启动xv6时，它应该像这样打印输出，描述第一个进程刚刚完成exec（）ing init时的页面表:

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

3.2.3 Detecting which pages have been accessed (hard)

某些垃圾收集器（一种自动内存管理形式）可以受益于有关哪些页面已被访问（读取或写入）的信息。在本实验的这一部分中，你将向 xv6 添加一个新特性，通过检查 RISC-V 页表中的访问位，检测和报告此信息给用户空间。当 RISC-V 硬件页行程器解决 TLB miss 时，它会在页表条目 (PTE) 中标记这些位。

你的任务是实现 pgaccess() 系统调用，它用于报告哪些页面已被访问。此系统调用接受三个参数。首先，它接受要检查的第一个用户页面的起始虚拟地址。其次，它接受要检查的页面数。最后，它接受一个用户空间地址，指向一个缓冲区，用于将结果存储为位掩码（位掩码是一种数据结构，每个页面使用一个位，并且第一个页面对应最低有效位）

3.2.4 submit

提交实验。

3.3 实验步骤

3.3.1 Speed up system calls

（1）在 kernel/proc.c 中的 proc_pagetable() 函数中执行映射。

（2）选择权限位，允许用户空间仅对页面进行读取。

```
if(mappages(pagetable, USYSCALL, PGSIZE,
            (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
 uvmunmap(pagetable, TRAMPOLINE, 1, 0);
 uvmunmap(pagetable, TRAPFRAME, 1, 0);
 uvmfree(pagetable, 0);
 return 0;
}
```

（3）在 allocproc() 中分配和初始化页面。

（4）确保在 freeproc() 中释放页面。

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

（5）运行 pgtbltest 时 ugetpid 测试用例通过

3.3.2 print a page table

（1）为实现vmprint()函数我们需要实现travelsal_pt()来递归调用页表中的每一项并输出，因为xv6采用了三级页表机制，因此只需要递归到第三层。

```
void travelsal_pt(pagetable_t pagetable,int level)
{
  for(int i=0;i<512;i++)
  {
    pte_t pte=pagetable[i];
    if(pte & PTE_V)
    {
      uint64 child=PTE2PA(pte);
      if(level==0)
      {
        printf("..%d: pte %p pa %p\n",i,pte,child);
        travelsal_pt((pagetable_t)child,level+1);
      }
      else if(level==1)
      {
        printf(".. ..%d: pte %p pa %p\n",i,pte,child);
        travelsal_pt((pagetable_t)child,level+1);
      }
      else
        printf(".. .. ..%d: pte %p pa %p\n",i,pte,child);
    }
  }

}
```

（2）编写vmprint()函数调用travelsal_pt()并进行输出。

（3）为运行vmprint()函数，我们需要在defs.h中对该函数进行声明并在exec.c中添加对于vmprint()的调用

（4）运行vmprint

3.3.3 Detecting which pages have been accessed (hard)

（1）首先，在 kernel/sysproc.c 中实现 sys_pgaccess() 函数。需要使用 argaddr() 和 argint() 来解析参数。对于输出位掩码，可以在内核中存储一个临时缓冲区，并在填充正确的位后，通过 copyout() 将其复制到用户空间。

```
// 从起始地址开始，逐页判断PTE_A是否被置位
// 如果被置位，则设置对应BitMask的位，并将PTE_A清空
for(i = 0 ; i < NumberOfPages ; StartVA += PGSIZE, ++i){
  if((pte = walk(myproc()->pagetable, StartVA, 0)) == 0)
    panic("pgaccess : walk failed");
  if(*pte & PTE_A){
    BitMask |= 1 << i;  // 设置BitMask对应位
    *pte &= ~PTE_A;   // 将PTE_A清空
  }
}
```

（2）需要在 kernel/riscv.h 中定义 PTE_A，即访问位。请参考 RISC-V 手册以确定其值。在检查是否设置了 PTE_A 之后，确保清除它。否则，就无法确定自上次调用 pgaccess() 以来是否访问过该页面

3.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示。

```
== Test pgtbltest ==
$ make qemu-gdb
(1.7s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.8s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(116.6s)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

3.5 实验困难和心得

本次实验的完成较为困难，它通过修改内核完成proc维护每个进程的内核页表并在切换进程时切换内核页表，并且实现了用户页表到内核页表的映射。页表作为操作系统中内存管理的重要部分，实现了将虚拟地址转换为物理地址的重要功能，这次实验提高了我对页表结构及其应用的认识和熟悉程度。
在system calls请求的加速中，有较多头文件和声明需要配置，最初没能掌握其中的逻辑和关系，所以漏掉许多声明导致报错，在熟悉逻辑和函数关系后成功解决了该问题。

4 实验四 Traps

4.1 实验目的

1. 探索如何使用陷阱
2. 了解一定RISC-V汇编知识
3. 理解xv6中的堆栈
4. 进行用户级陷阱处理的编写

4.2 实验内容

4.2.1 RISC-V assembly（easy）

了解RISC-V组件，阅读call.asm中函数g、f和main的代码并回答相应问题。

4.2.2 Backtrace（moderate）

在kernel/printf.c中实现backtrace()函数。在sys_sleep中插入对该函数的调用，然后运行bttest，它调用sys_sleep。您的输出应如下所示：

backtrace:
0x0000000080002cda
0x0000000080002bb6
0x0000000080002898

测试结束后退出qemu。在终端中：地址可能略有不同，但如果运行addr2line -e kernel/kernel（或riscv64-unknown-elf-addr2line -e kernel/kernel）并按如下方式剪切和粘贴上述地址：

```
$ addr2line -e kernel/kernel
0x0000000080002de2
0x0000000080002f4a
0x0000000080002bfc
Ctrl-D
```

您应该看到如下内容:

kernel/sysproc.c:74
kernel/syscall.c:224
kernel/trap.c:85

4.2.3 Alarm（hard）

在本练习中，将向xv6添加一个特性，该特性在进程使用CPU时间时定期向其发出警报。这对于希望限制占用多少CPU时间的受计算限制的进程，或者对于希望进行计算但又希望执行某些周期性操作的进程，可能很有用。更一般地说，将实现用户级中断/故障处理程序的原始形式；例如，可以使用类似的方法来处理应用程序中的页面错误。如果解决方案通过alarmtest和usertests，那么它就是正确的。

4.2.4 submit

提交实验。

4.3 实验步骤

4.3.1 RISC-V assembly

（1）首先执行

$ **make fs.img**

用于生成call.asm文件。

（2）Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf?

答:

```
void main(void) {
  1c: 1141                addi  sp,sp,-16
  1e: e406                sd ra,8(sp)
  20: e022                sd s0,0(sp)
  22: 0800                addi  s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24: 4635                li a2,13
  26: 45b1                li a1,12
  28: 00000517            auipc a0,0x0
  2c: 7a050513            addi  a0,a0,1952 # 7c8 <malloc+0xe8>
  30: 00000097            auipc ra,0x0
  34: 5f8080e7            jalr  1528(ra) # 628 <printf>
  exit(0);
  38: 4501                li a0,0
  3a: 00000097            auipc ra,0x0
  3e: 274080e7            jalr  628(ra) # 2ae <exit>
```

a0~a7储存函数参数，例如`printf`的13寄存在a2寄存器中。

（3）Where is the call to function f in the assembly code for main? Where is the call to g? (Hint: the compiler may inline functions.)

答:

同样从上段代码中可以看出printf()函数对于f函数的调用直接获得了12这一结果，所以在这里对f函数进行了内联优化处理。

```
int f(int x) {
   e: 1141                  addi  sp,sp,-16
  10: e422                  sd s0,8(sp)
  12: 0800                  addi  s0,sp,16
  return g(x);
}
  14: 250d                  addiw a0,a0,3
  16: 6422                  ld s0,8(sp)
  18: 0141                  addi  sp,sp,16
  1a: 8082                  ret
```

在函数f中调用了函数g，同样可以看出这里的g函数也进行了内联优化处理。

（4）At what address is the function printf located?

答：printf位于0x628的位置。

（5）What value is in the register ra just after the jalr to printf in main ?

答：ra存储的是printf函数的返回地址，ra=pc+4=0x34+4=0x38

（6）Run the following code.

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

What is the output? Here's an ASCII table that maps bytes to characters.
The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

答：output为Hello World，57616使用十六进制标识，无符号整型i使用字符串的形式输出，由于RISC-V是小端，所以对应的字符为'0x72','0x6c','0x64'
如果RISC-V是大端，上述代码需要输出同样的结果，那么需要把i设为0x726c6400，不需要改变57616的值

（7）In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

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

答：在printf函数中未写明y之后%d所对应的值，所以输出的答案不确定，取决于寄存器中存储的值。

4.3.2 backtrace

（1）在defs.h中添加函数声明以便之后调用
（2）根据题目要求在riscv.h中添加r_fp()函数。
（3）在printf.c中对backtrace函数进行编写。

```
void
backtrace()
{
  printf("backtrace:\n");
  uint64 fp=r_fp();
  uint64 sd=PGROUNDDOWN(fp);//sd is used to mark the bottom of stack
  uint64 ra=0;
  while(PGROUNDDOWN(fp)==sd)
  {
    ra=*(uint64*)(fp-8);//the address of function
    fp=*(uint64*)(fp-16);//the address of next frame pointer
    printf("%p\n",ra);
  }
}
```

即遍历一遍栈帧并打印返回地址，在xv6中只会给每个用户进程分配一个页（4KB）作为用户栈，所以可以通过检查fp是否到达页的开头来判断是否到达栈头。

（4）运行backtrace

4.3.3 alarm

（1）在user.h、usys.pl、syscall.h以及syscall.c中添加相应内容来调用sigalarm以及sigreturn

（2）首先在struct proc中添加相应的变量。

```
  int alarmticks; // ticks interval
  uint64 alarmhandler; // function to handler alarm
  int tickspass;
  struct trapframe * alarmtrapframe;
  int accessable;
```

（3）实现sigalarm这个系统调用，该函数用于初始化新加的proc属性。

（4）实现sys_sigreturn函数，用于还原寄存器组。

```
sys_sigalarm(void)
{
    int ticks;
    if(argint(0,&ticks)<0)
      return -1;
    if(ticks==0)
      return 0;
    uint64 addr;
    if(argaddr(1,&addr)<0)
        return -1;
    struct proc *p=myproc();
    p->handler=(void(*)())addr;
    p->ticks=ticks;
    p->ticksused=0;
    return 0;
}
uint64
sys_sigreturn(void)
{
    struct proc* p=myproc();
    *p->trapframe=p->trapframe_bak;
    p->alarm_running=0;
    return 0;
}
```

（5）由于题目中提示每次tick产生都会产生一次中断，并且可以在trap.c中的usertrap里对中断进行处理，所以对trap.c中的usertrap函数进行修改。

```
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
   {
    if(p->ticks>=0&&p->alarm_running!=1)
    {
      p->ticksused++;
      if(p->ticksused>=p->ticks)
      {
        printf("alarm!\n");
        p->trapframe_bak=*(p->trapframe);
        p->alarm_running=1;
        p->trapframe->epc=(uint64)p->handler;
        p->ticksused=0;
      }
    }

    yield();
  }
```

（6）运行alarm

4.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示。

```
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (2.5s)
== Test running alarmtest ==
$ make qemu-gdb
(3.4s)
== 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 (116.5s)
    (Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 85/85
```

4.5 实验困难和心得

本次实验引入了堆栈结构以及中断机制。

对于每一个进程的都有自己的堆栈对自身数据进行存储和使用，对于栈而言，存在两个重要的寄存器来确定栈的范围，一个是寄存栈底部指针的SP，另一个是寄存当前栈帧顶部的FP来确保不会产生读写错误。本次实现的backtrace就对进程的栈进行了遍历，让我更直观的看到栈是怎样的。

而本次实现的alarm就是中断现场的保存以及完成操作后现场的恢复，这让我对中断机制的执行过程更加了解了。
在usertests中出现了一个运行错误，在alarm测试中没进行内存块的申请初始化和释放，经过调试之后成功解决了该问题。

5 实验五 Copy-on-Write Fork for xv6

5.1 实验目的

1. 了解COW是什么
2. 了解COW有什么优势以及他是怎么对操作系统运行进行优化的
3. 了解COW的局限性

5.2 实验内容

问题：在 xv6 中，fork() 系统调用会完全复制父进程的用户空间内存到子进程中，这可能会花费很长时间，并且在一些情况下是浪费的。只有在父进程和子进程都使用并对某个页面进行写入操作时，才真正需要进行复制。

解决办法：Copy-on-write (COW) fork() 的目标是推迟分配和复制子进程的物理内存页，直到实际需要复制的时候。它通过共享父进程的物理页，并标记 PTE 为不可写，以实现复制的延迟和节省内存。在需要写入时，触发页错误，进行复制，实现只有在必要时才复制和分配内存。释放物理页时，需要处理多个进程共享的情况，只有当最后一个引用消失时，才释放物理页。

5.2.1 Implement copy-on write（hard）
在xv6内核中实现写时拷贝fork。如果修改后的内核成功地执行了cowtest和usertests程序，那么就完成了。

5.2.2 submit
提交实验。

5.3 实验步骤

5.3.1 implement copy-on write

有两个场景需要处理 cow 的写入内存页场景：

一个是用户进程写入内存，此时会触发 page fault 中断（15号中断是写入中断，只有这个时候会触发
cow，而13号中断是读页面，不会触发 cow）；
另一个是直接在内核状态下写入对应的内存，此时不会触发 usertrap 函数，需要另做处理。

（1）kernel/kalloc.c 中定义一个用于计数的数组，在 kernel/riscv.h 中定义 COW 标记位和计算物理内存页下标的宏函数。
（2）引入COW机制后，创建子进程不是创建父进程的拷贝，而是只创建指向父进程物理页面的页表，所以我们改写uvmcopy()函数，不再给子进程分配页面，而是将父进程的物理页映射进子进程的页表，并将两个进程的PTE_W都清零。

```
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);
    *pte = (*pte & ~PTE_W) | PTE_COW;
    flags = PTE_FLAGS(*pte);
    // if((mem = kalloc()) == 0)
    //  goto err;
    // memmove(mem, (char*)pa, PGSIZE);
    if (mappages(new, i, PGSIZE, pa, flags) != 0)
    {
      // kfree(mem);
      goto err;
    }
    // acquire(&ref_lock);
    page_ref[COW_INDEX(pa)]++;
    // release(&ref_lock);
  }
  return 0;
err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

（3）当某个进程试图写某个页会触发page fault(scause=15)，这是因为此时父子进程对所有的COW页都没有写权限，因此需要在trap.c中处理异常。

```
else if (r_scause() == 15)
{
  uint64 va = r_stval();
  if (va >= p->sz)
    p->killed = 1;
  else if (cow_alloc(p->pagetable, va) != 0)
    p->killed = 1;
}
```

其中cow_alloc函数代码如下，其功能是检查权限位并分配新的物理页，将它映射到产生缺页异常的进程的页表中，同时设置写权限位。

```c
int cow_alloc(pagetable_t pagetable, uint64 va)
{
  va = PGROUNDDOWN(va);
  if (va >= MAXVA)
    return -1;
  pte_t *pte = walk(pagetable, va, 0);
  if (pte == 0)
    return -1;
  uint64 pa = PTE2PA(*pte);
  if (pa == 0)
    return -1;
  uint64 flags = PTE_FLAGS(*pte);
  if (flags & PTE_COW)
  {
    uint64 mem = (uint64)kalloc();
    if (mem == 0)
      return -1;
    memmove((char *)mem, (char *)pa, PGSIZE);
    uvmunmap(pagetable, va, 1, 1);
    flags = (flags | PTE_W) & ~PTE_COW;
    //*pte = PA2PTE(mem) | flags;
    if (mappages(pagetable, va, PGSIZE, mem, flags) != 0)
    {
      kfree((void *)mem);
      return -1;
    }
  }
  return 0;
}
```

（4）设置全局数组记录每个物理页被几个进程所拥有。

（5）对分配物理页函数kalloc进行修改

（6）在进程fork时调用uvmcopy函数使COW页对应的计数器加一，free时将计数器减一。

（7）如果内核调用copyout函数修改一个进程的COW页也需要进行cow_alloc类似的操作。.

```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);
    if (cow_alloc(pagetable, va0) != 0)
      return -1;
    pa0 = walkaddr(pagetable, va0);
    if (pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if (n > len)
      n = len;
    pte = walk(pagetable, va0, 0);
    if (pte == 0)
      return -1;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

5.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示。

```
== Test running cowtest ==
$ make qemu-gdb
(5.7s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(109.4s)
== 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
```

5.5 实验困难和心得

本次实验介绍了COW机制，即写时拷贝技术，这项技术会在各段内容要发生变化的时候才会将父进程的内容复制给子进程。这减少了因为无意义的复制而导致的效率下降，让操作系统更加高效。

本次实验也让我更进一步地了解到一块内存的分配、使用、销毁、回收的一系列过程，即整个生命周期是如何变化的，让我对内存分配策略有了更进一步的认识。

这次实验涉及的物理内存和虚拟地址的操作难度更大。在处理中断 usertrap 时需要判断很多异常情况，测试用例会测试到。因此进行了多次修改。期间比较难以想到的就是实现 ref_count 减少 1，在 kfree 中添加这一功能，所有的物理页面取消映射时，最终都会调用 kfree，因为要将其释放掉。

6 实验六 Multithreading

6.1 实验目的

1. 理解多线程的概念
2. 通过实验完成如何实现线程的切换
3. 学习如何完成多线程之间的同步

6.2 实验内容

6.2.1 Uthread:switiching between threads（moderate）

您的工作是提出一个创建线程和保存/恢复寄存器以在线程之间切换的计划，并实现该计划。完成后，make grade应该表明您的解决方案通过了uthread测试。

6.2.2 Using threads（moderate）

请在notxv6/ph.c中的put和get中插入lock和unlock语句，以便在两个线程中丢失的密钥数始终为0。修改代码，使一些put操作在保持正确性的同时并行运行。ph_fast测试要求两个线程产生的put/s至少 是一个线程的 1.25 倍。当make grade说您的代码通过ph_安全测试时，您就完成了，该测试需要两个线程的零缺失键。

6.2.3 Barrier（moderate）

屏障：应用程序中的一个点，所有参与的线程都必须等待，直到所有其他参与的线程也到达该点。您将
使用 pthread 条件变量，这是一种类似于 xv6 的睡眠和唤醒的序列协调技术。
实现所需的屏障行为
pthread_cond_wait(&cond, &mutex); // 在 cond 上休眠，释放互斥锁，在唤醒时获取
pthread_cond_broadcast(&cond); // 唤醒所有在 cond 上休眠的线程

6.2.4 submit

提交实验。

6.3 实验步骤

6.3.1 uthread: switching between threads

（1）定义上下文字段，从 proc.h 中复制一下 context 结构体内容，用于保存 ra、sp 以及 callee-saved registers
（2）参考kernel/swtch.S，在uthread_switch.S中保存当前线程的寄存器，恢复即将要切换到线程的寄存器。
（3）在uthread.c中的thread_create()函数中添加保存新线程返回地址和栈指针

```
    t->contex.ra = (uint64)func;
  t->contex.sp = (uint64)&t->stack + (STACK_SIZE - 1);
  }
```

（4）根据提示在uthread.c中的thread_schedule()添加调用，调用thread_switch()

```
  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((uint64)&t->contex, (uint64)&current_thread->contex);
  } else
    next_thread = 0;
}
```

（5）运行uthread

6.3.2 using threads

（1）根据提示运行make ph，./ph 1以及./ph 2，其中./ph 1无丢失的键，./ph 2由丢失的键。

（2）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

答：因为由多个线程同时工作，所以当其中一个进程给哈希表填入一个键的时候其他进程也可能在添加键，当两个哈希值不相等时插入成功，但两个哈希值相同时会导致后写的覆盖另一个，导致键的丢失。

（3）为提高多线程的效率设计锁数组，如果只加一个锁，锁的粒度很大，会导致丢失性能，结果还不如不加锁的单线程。因此需要将锁的粒度减小，为每个槽位（bucket）加一个锁。并且在 main 函数中初始化锁

```
39  pthread_mutex_t lock[NBUCKET];
40  static
41  void put(int key, int value)
```

（4）根据代码可知数组table的元素table[i]为链表，其中存放元素entry，并行访问不同的table不会导致数据的丢失，所以修改put和get函数。

```
static void put(int key, int value)
{
  int i = key % NBUCKET;

  pthread_mutex_lock(&lock[i]);
  // is the key already present?
  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(&lock[i]);
}
```

```
static struct entry *
get(int key)
{
  int i = key % NBUCKET;

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

（5）运行./ph 2

6.3.3 barrier

（1）通过round调用barrier的线程进行技术，并用pthread_cond_wait和pthread_cond_broadcast进行线程在cond上的等待和唤醒。一个线程只有在其他线程调用pthread_cond_wait进入等待之后才能获得mutex。生产者消费者模式，如果还有线程没到达，就加入到队列中，等待唤起；如果最后一个线程到达了，就将轮数加一，然后唤醒所有等待这个条件变量的线程。

（2）barrier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);
  if (++bstate.nthread == nthread)
  {
    bstate.nthread = 0;
    bstate.round++;
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
  else
  {
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

6.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示。

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (2.7s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/pzone/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/pzone/xv6-labs-2021'
ph_safe: OK (11.8s)
== Test ph_fast == make[1]: Entering directory '/home/pzone/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/pzone/xv6-labs-2021'
ph_fast: OK (24.1s)
== Test barrier == make[1]: Entering directory '/home/pzone/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/pzone/xv6-labs-2021'
barrier: OK (3.2s)
== Test time ==
time: OK
Score: 60/60
```

6.5 实验困难和心得

条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。
本次实验是有关多线程的编写以及锁的运用。我们在本次实验中实现了线程的切换，来保存当前线程以及恢复将要切换到的线程，并且掌握如何通过锁和条件信号量来完成多线程之间的同步，对线程这一概念掌握的更加深刻。

7 实验七 networking

7.1 实验目的

本实验中，使用一个名为E1000的网络设备来处理网络通信。任务是完成kernel/e1000.c中的e1000_transmit()和e1000_recv()函数，以使驱动程序能够进行数据包的发送和接收。

7.2 实验内容

7.2.1 networking

任务是需要完成E1000网卡的驱动程序，使得xv6操作系统能够使用E1000处理网络通信。完成kernel/e1000.c中的e1000_transmit()和e1000_recv()函数，以使驱动程序能够进行数据包的发送和接收。当make grade命令显示您的解决方案通过所有测试时，您的工作就完成了。

7.2.2 submit

提交实验。

7.3 实验步骤

（1）完善e1000_transmit 函数

```
  acquire(&e1000_lock);
  uint index = regs[E1000_TDT];
  if ((tx_ring[index].status & E1000_TXD_STAT_DD) == 0)
  {
    release(&e1000_lock);
    return -1;
  }
  if (tx_mbufs[index])
    mbuffree(tx_mbufs[index]);
  tx_mbufs[index] = m;
  tx_ring[index].length = m->len;
  tx_ring[index].addr = (uint64)m->head;
  tx_ring[index].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
  regs[E1000_TDT] = (index + 1) % TX_RING_SIZE;
  release(&e1000_lock);
  return 0;
}
```

（2）完善e1000_recv 函数

```
  uint index = regs[E1000_RDT];
  index = (index + 1) % RX_RING_SIZE;
  while (rx_ring[index].status & E1000_RXD_STAT_DD)
  {
    rx_mbufs[index]->len = rx_ring[index].length;
    net_rx(rx_mbufs[index]);
    if ((rx_mbufs[index] = mbufalloc(0)) == 0)
      panic("e1000");
    rx_ring[index].addr = (uint64)rx_mbufs[index]->head;
    rx_ring[index].status = 0;
    index = (index + 1) % RX_RING_SIZE;
  }
  if (index == 0)
    index = RX_RING_SIZE;
  regs[E1000_RDT] = (index - 1) % RX_RING_SIZE;
}
```

7.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示。

```
== Test running nettests ==
$ make qemu-gdb
(3.9s)
== 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
```

7.5 实验困难和心得

本次实验参考的是 e1000 的硬件手册,根据手册内容书写发送和接收函数，主要是利用一个循环列表不断读取和写入。主要是基于已有的资料进行学习和参考，在引导和摸索中一步步实现了网络通信这一任务，该实验提高了我学习已有知识并加以转化和利用的能力，真正做到了从学习到实践这一过程。

8 实验八 Locks

8.1 实验目的

1. 提高对于并行性的认识
2. 理解多核机器上并行性差的常见症状为high lock contention
3. 学习如何通过更改数据结构和锁定策略减少争用

8.2 实验内容

8.2.1 Memory allocator（moderate）

您的工作是实现每个CPU的空闲列表，并在CPU的空闲列表为空时进行窃取。您必须给出所有以“kmem”开头的锁的名称。也就是说，您应该为每个锁调用initlock，并传递一个以“kmem”开头的名称。运行kalloctest以查看您的实现是否减少了锁争用。要检查它是否仍然可以分配所有内存，请运行usertests sbrkmuch。您的输出将与下面所示的类似，在kmem锁上的争用总数将大大减少，尽管具体的数字会有所不同。确保usertests中的所有测试都通过。

8.2.2 Buffer cache（hard）

修改块缓存，以便在运行bcachetest时，bcache中所有锁的获取循环迭代次数接近于零。理想情况下，块缓存中涉及的所有锁的计数总和应为零，但如果总和小于500则可以。修改bget和brelse，以便bcache中不同块的并发查找和释放不太可能在锁上发生冲突（例如，不必全部等待bcache.lock）。必须保持不变，即每个块最多缓存一个副本。

8.2.3 submit

提交实验。

8.3 实验步骤

8.3.1 memory allocator

（1）本次实验目的为对多CPU为每个CPU拆分一个空闲空间链表并采用单独的锁管理来降低锁冲突带来的额外开销，增加并行性。

（2）在param.h中定义了xv6支持最大并行的CPU数量NCPU，因此将kalloc.c中kmem定义为NCPU个。

（3）对kmem初始化进行修改

（4）修改kfree，对于释放页面放入freelist的机制进行修改。

```
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();
  int id = cpuid();
  pop_off();
  acquire(&kmem[id].lock);
  r->next = kmem[id].freelist;
  kmem[id].freelist = r;
  release(&kmem[id].lock);
}
```

（5）修改kalloc函数，当在当前核心上申请失败时尝试从其他核心上获取页面。

（6）运行memory allocator

8.3.2 buffer cache

（1）在多个进程密集的使用文件系统时，他们可能会争用bcache.lock，所以针对磁盘空间中的块号进行散列处理，并用哈希表维护缓冲区。

（2）根据提示首先修改结构，先修改数据结构，将 buf 分成 13 份（实验指导书上建议使用的质数），同时获取trap.c 中的 ticks 变量。需要所有 bucket 的小锁，需要一个大的锁防止死锁。不能像任务一那样直接使用 bcache 数组，这会改变 buf 的大小。

```
extern uint ticks;
#define NBUCKET 13
```

（3）为 kernel/buf.h 中添加 lastuse 字段，便于使用 LRU 机制。

（4）接下来，初始化，使用双向链表实现。修改brels，bpin和bunpin，直接加锁即可。brelse中，由于不使用之前的方式实现 LRU，因此当遇到空闲块的时候，直接设置它的使用时间即可。

（5）当需要查找一个块并将其替换的时候找到timestamp最小的空闲块，如果没有这个空闲块就需要在哈希表中寻找，修改bget函数。

```c
static struct buf *
bget(uint dev, uint blockno)
{
  struct buf *b, *b2 = 0;
  int i = hash(blockno), min_ticks = 0;
  acquire(&bcache.lock[i]);
  // 1. Is the block already cached?
  for (b = bcache.head[i].next; b != &bcache.head[i]; b = b->next)
  {
    if (b->dev == dev && b->blockno == blockno)
    {
      b->refcnt++;
      release(&bcache.lock[i]);
      acquiresleep(&b->lock);
      return b;
    }
  }
  release(&bcache.lock[i]);
  // 2. Not cached.
  acquire(&bcache.biglock);
  acquire(&bcache.lock[i]);
  // 2.1 find from current bucket.
  for (b = bcache.head[i].next; b != &bcache.head[i]; b = b->next)
  {
    if (b->dev == dev && b->blockno == blockno)
    {
      b->refcnt++;
      release(&bcache.lock[i]);
      release(&bcache.biglock);
      acquiresleep(&b->lock);
      return b;
    }
  }
```

```c
  // 2.3 find block from the other buckets.
  for (int j = hash(i + 1); j != i; j = hash(j + 1))
  {
    acquire(&bcache.lock[j]);
    for (b = bcache.head[j].next; b != &bcache.head[j]; b = b->next)
    {
      if (b->refcnt == 0 && (b2 == 0 || b->lastuse < min_ticks))
      {
        min_ticks = b->lastuse;
        b2 = b;
      }
    }
    if (b2)
    {
      b2->dev = dev;
      b2->refcnt++;
      b2->valid = 0;
      b2->blockno = blockno;
      // remove block from its original bucket.
      b2->next->prev = b2->prev;
      b2->prev->next = b2->next;
      release(&bcache.lock[j]);
      // add block
      b2->next = bcache.head[i].next;
      b2->prev = &bcache.head[i];
      bcache.head[i].next->prev = b2;
      bcache.head[i].next = b2;
      release(&bcache.lock[i]);
      release(&bcache.biglock);
      acquiresleep(&b2->lock);
      return b2;
    }
    release(&bcache.lock[j]);
  }
  release(&bcache.lock[i]);
  release(&bcache.biglock);
  panic("bget: no buffers");
```

步骤：

1 首先还是判断是否命中，如果已经缓存好，直接返回；

2 如果没找到，释放锁，按顺序先获取大锁，再获取小锁，避免死锁；这时由于可能释放锁后，又可能会有缓存，因此再遍历一遍；

3 如果还没命中，就去寻找当前 bucket 对应的 LRU 的空闲块，使用 ticks 的方式寻找，如果找到了，就返回；

4 如果还没找到，需要向其他 bucket 中拿内存块。

（6）运行bcachetest

8.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示

```
== Test running kalloctest ==
$ make qemu-gdb
(74.0s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (11.6s)
== Test running bcachetest ==
$ make qemu-gdb
(11.5s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (149.4s)
== Test time ==
time: OK
Score: 70/70
```

8.5 实验困难和心得

本次实验考察的是对于并行系统的优化，通过分割资源来使高锁竞争得到缓解，提高其并行利用率，在文件较大或者其他情况下这对于操作系统的优化较大，可以提高资源的利用率。

在处理多线程或多进程中资源抢占导致的锁竞争现象可以通过分割资源并分别加锁，锁的数量越多单个锁上冲突就越少。而这种并行性正是操作系统提高资源利用率所要追求的。

整体思路就是降低锁的粒度，将一个大锁更换为一些粒度小的锁，这样可以大幅度降低锁的竞争。利用 LRU 机制，而且不使用原来的方式，而是用 ticks 的方式来替换，因此设计起来复杂了一些。这个实验过程复杂，但是跟着思路走的话比较流畅，调试时间也不长。

9 实验九 File System

9.1 实验目的

1. 了解UNIX系统以及xv6系统组织文件的方式
2. 理解直接块号和间接块号
3. 学习硬链接和软链接的实现方式

9.2 实验内容

9.2.1 Large files（moderate）

修改bmap（），以便除了直接块和单间接块之外，它还实现双间接块。你只需要有11个直接块，而不是12个，为你的新的双间接块腾出空间；不允许更改磁盘inode的大小。ip->addrs[]的前11个元素应该是直接块；第12个应该是一个单独的间接块（与当前块一样）；13号应该是你的新双间接块。当bigfile写入65803个块并成功运行usertests时，完成此练习

9.2.2 Symbolic links（moderate）

实现symlink（char*target，char*path）系统调用，该调用在引用由target命名的文件的路径处创建一个新的符号链接。有关更多信息，请参阅手册页符号链接。要进行测试，请将symlinktest添加到Makefile并运行它。当测试产生以下输出（包括usertests）时，解决方案就完成了。

```
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
```

```
$ usertests
...
ALL TESTS PASSED
$
```

9.2.3 submit

提交实验。

9.3 实验步骤

9.3.1 large files

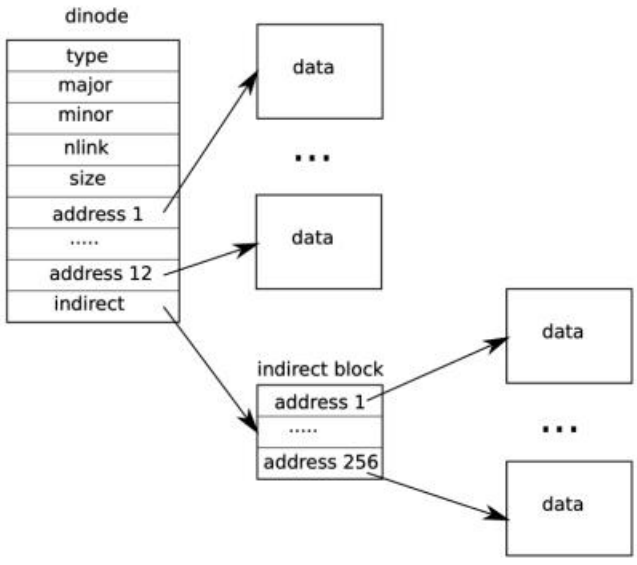
（1）阅读xv6-book文件系统相关部分并阅读fs.c中的bmap()。

（2）修改fs.h的宏定义和结构体

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDINDIRECT NINDIRECT * NINDIRECT
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)

// 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
};
```

（3）原本bmap()支持以及块表，如下图所示



现修改bmap()使其支持二级块表

```
// 二级索引
bn -= NINDIRECT;
if (bn < NDINDIRECT)
{
  if ((addr = ip->addrs[NDIRECT + 1]) == 0)
    ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
  // 通过一级索引，找到下一级索引
  bp = bread(ip->dev, addr);
  a = (uint *)bp->data;
  if ((addr = a[bn / NINDIRECT]) == 0)
  {
    a[bn / NINDIRECT] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);
  // 重复上面的代码，实现二级索引
  bp = bread(ip->dev, addr);
  a = (uint *)bp->data;
  if ((addr = a[bn % NINDIRECT]) == 0)
  {
    a[bn % NINDIRECT] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);
  return addr;
}
```

```
  if (ip->addrs[NDIRECT + 1])
  {
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint *)bp->data;
    for (j = 0; j < NINDIRECT; j++)
    {
      if (a[j])
      {
        bp2 = bread(ip->dev, a[j]);
        a2 = (uint *)bp2->data;
        for (i = 0; i < NINDIRECT; i++)
        {
          if (a2[i])
            bfree(ip->dev, a2[i]);
        }
        brelse(bp2);
        bfree(ip->dev, a[j]);
        a[j] = 0;
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT + 1]);
    ip->addrs[NDIRECT] = 0;
  }

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

（2）运行largefile

9.3.2 symbolic links

（1）硬连接指通过索引节点来进行连接。在Linux的文件系统中，保存在磁盘分区中的文件不管是什么类型都给它分配一个编号，称为索引节点号(Inode Index)。在Linux中，多个文件名指向同一索引节点是存在的。一般这种连接就是硬连接。硬连接的作用是允许一个文件拥有多个有效路径名，这样用户就可以建立硬连接到重要文件，以防止“误删”的功能。其原因如上所述，因为对应该目录的索引节点有一个以上的连接。只删除一个连接并不影响索引节点本身和其它的连接，只有当最后一个连接被删除后，文件的数据块及目录的连接才会被释放。也就是说，文件真正删除的条件是与之相关的所有硬连接文件均被删除。

（2）另外一种连接称之为符号连接（Symbolic Link），也叫软连接。软链接文件有类似于Windows的快捷方式。它实际上是一个特殊的文件。在符号连接中，文件实际上是一个文本文件，其中包含的有另一文件的位置信息。

（3）sys_link的作用就是 假设原本有目录 /a/b 现在调用path(/a/b , /c/d) 那么目录a指向b的同时还指向c目录下的d ， 但是二者必须在同一磁盘之上，软连接sys_symlink就是取消了二者必须在同一磁盘上的限制实现sys_symlink,在dp->data添加MAXPATH个字节 保存symlink指向的target path。sys_symlink实现如下。

```
uint64
sys_symlink(void)
{
  char path[MAXPATH], target[MAXPATH];
  struct inode *ip;
  // 读取参数
  if (argstr(0, target, MAXPATH) < 0)
    return -1;
  if (argstr(1, path, MAXPATH) < 0)
    return -1;
  // 开启事务
  begin_op();
  // 为这个符号链接新建一个 inode
  if ((ip = create(path, T_SYMLINK, 0, 0)) == 0)
  {
    end_op();
    return -1;
  }
  // 在符号链接的 data 中写入被链接的文件
  if (writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH)
  {
    iunlockput(ip);
    end_op();
    return -1;
  }
  // 提交事务
  iunlockput(ip);
  end_op();
  return 0;
}
```

（4）最后修改sys_open，在打开文件时，如果遇到符号链接，直接打开对应的文件。这里为了避免符号链接彼此之间互相链接，
导致死循环，设置了一个访问深度（我设成了 20），如果到达该访问次数，则说明打开文件失败。每次先读取对应的 inode，根据其中的文件名称找到对应的 inode，然后继续判断该 inode 是否为符号链接。
（5）运行sysinfo

9.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示。

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (120.7s)
== Test running symlinktest ==
$ make qemu-gdb
(0.6s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (223.8s)
== Test time ==
time: OK
Score: 100/100
```

9.5 实验困难和心得

本次实验考察了unix系统的文件组织结构，并且让我加深了对一级间接文件结构和二级间接文件结构的理解和认识。同时，在编写软链接的时候我进一步区分了硬链接和软链接的区别，也理解了为什么说“软链接也是一种文件”的概念。这种思路将复杂的系统简化为了几种类型让整个逻辑结构更加直观了。

iput 的作用是将 ip->ref 减一，如果 ip->ref == 1 就直接将其从 cache 中释放掉

writebig: panic freeing free block，如果完成了 symlinktest，在执行 usertests 之前，要先 执行一次 bigfile，否则会导致出错，因为 wirtebif 要写和 MAXFILE 一样大的文件，而只 有执行 bigfile 之后才有修改后的 MAXFILE

10 实验十 Mmaps

10.1 实验目的

1. 了解使用虚拟内存的原因
2. 理解创建和释放文件映射的方式

10.2 实验内容

10.2.1 mmap（hard）

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

10.2.2 submit

提交实验。

10.3 实验步骤

10.3.1 mmap

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

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

```c
uint64
sys_mmap(void)
{
  uint64 addr;
  int length, prot, flags, fd, offset;
  struct file *file;
  struct proc *p = myproc();
  if (argaddr(0, &addr) || argint(1, &length) || argint(2, &prot) ||
      argint(3, &flags) || argfd(4, &fd, &file) || argint(5, &offset))
  {
    return -1;
  }
  if (!file->writable && (prot & PROT_WRITE) && flags == MAP_SHARED)
    return -1;
  length = PGROUNDUP(length);
  if (p->sz > MAXVA - length)
    return -1;
  for (int i = 0; i < VMASIZE; i++)
  {
    if (p->vma[i].used == 0)
    {
      p->vma[i].used = 1;
      p->vma[i].addr = p->sz;
      p->vma[i].length = length;
      p->vma[i].prot = prot;
      p->vma[i].flags = flags;
      p->vma[i].fd = fd;
      p->vma[i].file = file;
      p->vma[i].offset = offset;
      filedup(file);
      p->sz += length;
      return p->vma[i].addr;
    }
  }
  return -1;
}
```

（3）由于使用了懒加载方式，所以在访问未加载界面时会产生缺页中断。在usertrap中对缺页中断进行处理。由于是懒加载，在读取或写入相应的虚拟地址时，会存在地址未映射的情况。这时需要将物理地址上的数据读到虚拟地址中，然后重新进行读取或写入操作。由于测试时会测试地址在栈空间之外等不合法的地方，因此产生读写中断时，需要首先判断地址是否合法。然后判断地址是否在某个文件映射的虚拟地址范围内，如果找到该文件，则读取磁盘，并将地址映射到产生中断的虚拟地址上，由于一些地址并没有进行映射，因此在 walk 的时候，遇到这些地址直接跳过即可

```c
  else if (r_scause() == 13 || r_scause() == 15)
  {
    // 缺页异常
    uint64 va = r_stval(); // 获取缺页地址
    if (va > MAXVA || va >= p->sz)
    {
      // 越界
      p->killed = 1;
    }
    else
    {
      struct VMA *pvma = 0;
      for (int i = 0; i < VMASIZE; i++)
      {
        // 找到对应的VMA
        if (p->vma[i].active == 0)
          continue;
        if (va >= p->vma[i].addr && va < p->vma[i].addr + p->vma[i].length)
        {
          pvma = &p->vma[i];
        }
      }

      if (pvma)
      {
        // 若找到了对应的VMA，则要把对应的虚拟地址分配到物理地址，同时把文件内容读到物理地址
        va = PGROUNDDOWN(va);
        uint64 pa = (uint64)kalloc();
```

```
      if (pa == 0)
      {
        // 分配失败
        p->killed = 1;
      }
      else
      {
        memset((void *)pa, 0, PGSIZE);
        ilock(pvma->fp->ip);                                  // 加锁
        readi(pvma->fp->ip, 0, pa, va - pvma->addr, PGSIZE); // 读取文件内容
        iunlock(pvma->fp->ip);                                // 解锁

        // 根据flag参数设置PTE
        int PTE_flags = PTE_U;
        if (pvma->prot & PROT_READ)
          PTE_flags |= PTE_R;
        if (pvma->prot & PROT_WRITE)
          PTE_flags |= PTE_W;
        if (pvma->prot & PROT_EXEC)
          PTE_flags |= PTE_X;

        // 映射
        printf("map start\n");
        if (mappages(p->pagetable, va, PGSIZE, pa, PTE_flags) != 0)
        {
          // 映射失败
          kfree((void *)pa);
          p->killed = 1;
        }
      }
    }
  }
}
```

（4）munmap即释放mmap空间，当mmap是MAP_SHARED时需要进行写回，除此之外不用写回。同时需要在数组中找到对应的vma结构体，且需要设定偏移量。

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

```
for (int i = 0; i < MAXVMA; i++)
{
  struct VMA *v = &p->vma[i];
  struct VMA *nv = &np->vma[i];
  // only unmap at start,end or the whole region
  if (v->used)
  {
    memmove(nv, v, sizeof(struct VMA));
    filedup(nv->f);
  }
}
```

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

```
if ((*pte & PTE_V) == 0)
  // panic("uvmcopy: page not present");
  continue;
```

（7）运行mmap

10.4 实验结果

通过make grade对所有实验内容进行测试，结果如下图所示。

```
== Test running mmaptest ==
$ make qemu-gdb
(3.6s)
== 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 (118.7s)
    (Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 140/140
```

10.5实验困难和心得

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

在实现 trap 中断处理时，要注意由于测试时会测试地址在栈空间之外等不合法的地方，因此产生读写中断时，需要首先判断地址是否合法。然后判断地址是否在某个文件映射的虚 拟地址范围内，如果找到该文件，则读取磁盘，并将地址映射到产生中断的虚拟地址。此外，还需要注意，由于一些地址并没有进行映射，因此在 walk 的时候，遇到这些地址直接跳过即可。