Xv6-2021 实验报告

2152602 王星琳

目录

Github 仓库地址：

https://github.com/xinglinwang1/xv6-lab-2021.git

Xv6-2021 实验内容概览

Xv6 简介

xv6 是一个教学用的操作系统，设计得相对简单，它基于早期的 UNIX 版本。这个操作系统主要用于教授操作系统的基本概念和原理，尤其是在 MIT 的操作系统课程中。

基于早期的 UNIX:xv6 的设计灵感来源于 Version 6 Unix (也被称为 V6)，这是 Unix 的一个早期版本。

xv6 的设计是基于教学目的。xv6 不是为了实际生产或日常使用而设计的，而是作为一个学习工具，帮助学生理解操作系统的基本概念，如进程管理、内存管理、文件系统等。xv6 的代码具有简洁性的特点。xv6 的代码相对简单和清晰，使其成为学习操作系统内部工作原理的理想材料。平台:尽管早期版本是为 x86 架构设计的，但随着时间的推移，xv6 也为其他架构(如 RISC-V)提供了支持。xv6 有众多的可用资源:除了源代码之外，xv6 还配有一本详细的教科书，该教科书解释了其设计和实现的细节。这使得独立学习者也可以理解和学习 xv6。xv6 有着活跃的社区：由于其在教育中的广泛使用，xv6 拥有一个活跃的社区，提供了大量的补丁、扩展和教学资源。xv6 设计了许多实验和挑战，在许多使用 xv6 的课程中，学生通常被分配特定的实验或项目，要求他们修改和扩展 xv6，从而深入了解操作系统的某些特定概念。总的来说，xv6 是一个强大的教学工具，为那些希望深入理解操作系统工作原理的学生和自学者提供了极大的帮助。

Xv6 实验内容

本次暑假小学期我选取的 xv6 实验为 2021 年版本的 xv6 Labs ，除去实验指导和环境配 置部分，共分为以下 10 个实验：

1. Utilities
- 目标:熟悉 xv6 的环境和基本工具。
- 任务:为 xv6 添加新的系统调用，并创建一个用户级程序来测试这个新的系统调用。

2.System calls
- 目标:理解系统调用的工作原理。 - 任务:为 xv6 添加几个新的系统调用。

3. Page tables
- 目标:深入了解虚拟内存和页表。
- 任务:修改 xv6 的内存管理来支持更复杂的页表结构。

4. Traps
- 目标:理解中断和异常的处理。
- 任务:修改 xv6 的陷阱处理代码，实现在用户空间的陷阱处理函数。

5. Copy on-write
- 目标:写时复制实验。
- 任务:修改 xv6 以实现惰性页分配。

6. Multithreading
- 目标:理解并发和多线程。
- 任务:为 xv6 添加对内核和用户级线程的支持。

7. Networking
• 目标:掌握网络通信的基本概念。
• 任务:为 xv6 添加网络支持，包括实现一个简单的网络堆栈和相关的系统调用。
8. Lock
• 目标:理解并发控制和锁。
• 任务:使用锁来保护 xv6 的数据结构，确保并发访问时的正确性。
9. File system
• 目标:理解文件系统的基本概念。
• 任务:增加新的文件类型并修改 xv6 的文件系统来支持日志。
10. Mmap
• 目标:内存映射实验。
• 任务:实现内存映射关系。

这些实验中，Lab Utilities 和 Lab System calls 主要涉及操作系统的使用与修改的基本方法；Lab Page tables 、Lab Traps 、Lab Multithreading 和 Lab File system 主要涉及操作系统中内存管理、进程管理和文件管理的重要概念；而 Lab Copy on-write 、Lab network driver 、Lab Lock 和 Lab mmap 主要是前面介绍的概念的综合应用，且这些应用在真实的现代操作系统中也较为常见。

由于时间关系和一些个人原因，我仅完成了前九个实验，最后一个实验尚未完成。

实验环境配置

安装 Ubuntu20.04

我选择的是 VMWare WorkStation 虚拟机平台，到官网下载相应的安装包，访问 https://www.vmware.com/cn/products/workstation-pro/workstation-pro-evaluation.html 后进行下载，然后使用默认设置进行安装，并重启计算机。

安装完成后，下载我们需要的 Ubuntu 20.04 镜像。考虑到国内特殊的网络环境，不建议直接在官网上下载，而是从国内一些高校的镜像站中进行下载。。下载完成后，在 VMWare 中新建虚拟机，选择系统为 Ubuntu ，然后使用默认设置一路向下，完成虚拟机的创建。然后在存储设置中，将光驱置入我们刚刚下载的镜像，然后启动虚拟机，按 Ubuntu 的提示进行安装，设置时区、账号、密码等。

配置 RISC-V 相关工具链

如果前文的镜像配置成功，那就可以直接在 Ubuntu 的终端中使用下面两条命令安装相关工具链：

```
sudo apt-get update && sudo apt-get upgrade -y
sudo apt-get install -y git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

输入管理员密码后，耐心等待即可安装成功，若中途出现问题，则可能需要从安装 Ubuntu 20.04 开始重试。

获取 xv6 原码

首先，确认上述的工具链已经配置完成，各环境变量也都设置好。然后对 git 进行配置， 主要在终端中使用下面的命令设置 git 的用户名和邮箱：

```
git config --global user.name "jwy"
git config --global user.email "1951510@tongji.edu.cn"
```

设置完成后，使用 git 将原始的 xv6 代码库克隆下来：

```
git clone git://g.csail.mit.edu/xv6-labs-2021
```

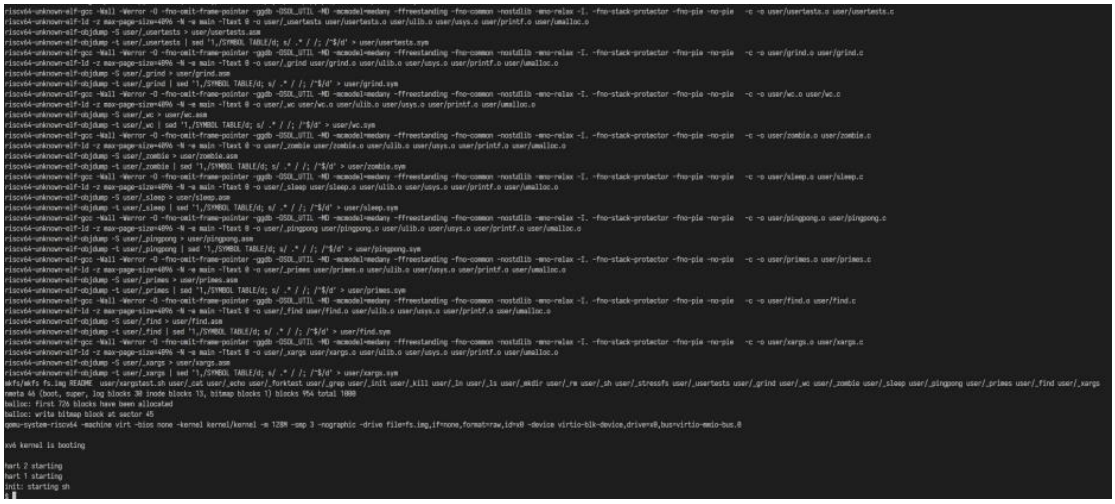
Lab1 Xv6 and Unix utilities

实验一 Boot xv6

在第一个实验中，我们需要编译代码库中的源码并在 qemu 中启动 xv6 系统。首先，我们需要将实验的代码库切换到 Lab Utilities 的分支，在终端中使用下面的指令：

```
$ cd xv6-labs-2021
$ git checkout util
```

之后直接使用 make qemu 即可编译并在 qemu 中运行 xv6，make 将执行一系列编译和链接操作。若运行成功，结果如下图所示：



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
```

此时即可在 qemu 中运行相关指令。

实验二 Sleep

在该实验中，我们需要实现 Unix 中经典的实用工具 sleep 。sleep 工具的作用是等待给定的 tick 数并退出（tick 是指两次次时钟中断的间隔时间）。根据要求，我们需要将实现的源码 放置在 user/sleep.c 。

根据提示，我们仿照 user/echo.c 的写法建立文件 user/sleep.c ，并应用 user/user.h 中定义的系统调用 int sleep(int)来实现我们的程序。

注意处理没有传入参数时的情况，输出错误提示并退出。

```c
int main(int argc, char *argv[])
{
    int sec;
    if (argc <= 1)
    {
        printf("usage: sleep [seconds]\n");
```

```
        exit(0);
    }
    sec = atoi(argv[1]);
    sleep(sec);
    exit(0);
}
```

程序编写完成后，打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面加入一行： $U/_sleep\

在终端执行 make qemu，输入 sleep 1000，行为符合预期。

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util sleep ，即可进行自动评测：

```
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$  ./grade-lab-util
sleep
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (0.9s)
== Test sleep, returns == sleep, returns: OK (0.8s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.8s)
```

实验三 Pingpong

在该实验中，我们需要实现一个名为 pingpong 的实用工具，用以验证 xv6 的进程通信的 一些机制。该程序创建一个子进程，并使用管道与子进程通信：父进程首先发送一字节的数 据给子进程，子进程接收到该数据后，在 shell 中打印”：received ping” ，其中 pid 为子进 程的进程号；子进程接收到数据后，向父进程通过管道发送一字节数据，父进程收到数据后 在 shell 中打印”：received pong” ，其中 pid 为父进程的进程号。

在编写代码之前，我们首先要学习一些所需的系统调用，如 fork，read，write，getpid，pipe 等。同时，user/user.h 提供了 void printf(const char*, ...) 函数，用于 shell 中的格式化输 出。

首先，创建双向管道用于父子进程间通信。随后创建子进程，根据题目要求实现代码逻辑。父进程首先向管道发送一字节的数据，子进程从管道中读取该数据后，打印<pid> received ping,并将管道上的一个字节写入父进程，父进程接收到这个字节后，打印<pid> received pong。

```
char buf[128];

int main(int argc, char *argv[])
{
    int p1[2], p2[2], pid;
    pipe(p1);
    pipe(p2);

    if (fork() == 0)
    {
        close(p2[1]);
        close(p1[0]);
        if (read(p2[0], buf, 1) == 1)
```

```
        {
            printf("%d: received ping\n", getpid());
            write(p1[1], buf, 1);
        }
    }
    else
    {
        close(p1[1]);
        close(p2[0]);
        write(p2[1], buf, 1);
        if (read(p1[0], buf, 1) == 1)
        {
            printf("%d: received pong\n", getpid());
        }
    }

    exit(0);
}
```

程序编写完成后，打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面加入一行： $U/_pingpong\

在终端执行 make qemu，输入 pingpong，行为符合预期。

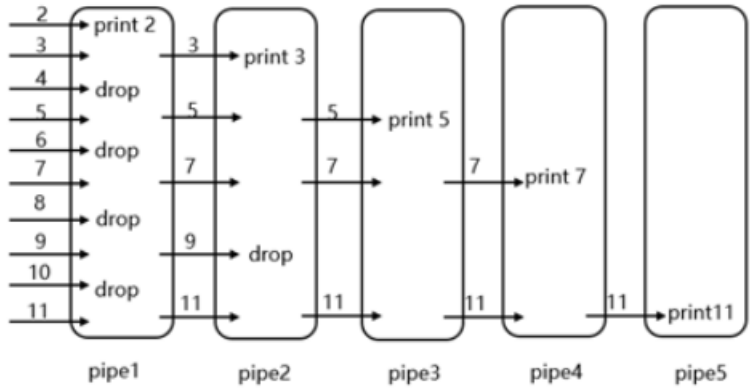
使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util pingpong，即可进行自动评测：

```
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$ ./grade-lab-util p
ingpong
make: “kernel/kernel”已是最新。
== Test pingpong == pingpong: OK (0.7s)
```

实验四 Prime

该实验要求实现一个基于管道的多进程质数筛。该质数筛的基本思路是每一个进程负责检验一个质数是否为输入的因子，在每次发现一个新质数后，就新建一个进程并令其负责检验该质因子，进程间通过管道传递数据。示例流程如下图所示：

第一个进程以质数 2 位因子，剔除 2 的倍数，剩余数字进入管道，如果管道不为空，则创建一个新的进程；如果为空则停止递归。下一个进程以第一个输入该进程的质数 3 为因子，剔除 3 的倍数，以此递归



具体代码实现如下：

```
void prime(int fd){
    int factor;
    //读出第一个数设为 factor
    if(read(fd, &factor, sizeof(int)) == 0){
        //管道为空，直接 exit
        exit(0);
    }

    printf("prime %d\n", factor);
    //管道不为空，创建新的管道
    int p[2];
    pipe(p);
    //子进程递归调用 prime
    if(fork() == 0){
        //读端作为参数给新的子进程
        close(p[1]);
        prime(p[0]);
    }
    //父进程
    else{
        close(p[0]);
        int divideNum;
        //读取管道中的数，若不能被 factor 整除，则发送给子进程
        while(read(fd, &divideNum, sizeof(int))){
            if(divideNum % factor != 0){
                write(p[1], &divideNum, sizeof(int));
            }
        }
        close(p[1]);
    }

    wait(0);
```

```
    exit(0);
}

int main(int argc, char const *argv[])
{
    int p[2];
    pipe(p);
    if(fork() == 0){
        close(p[1]);
        prime(p[0]);
    }
    else{
        close(p[0]);
        for(int i = 2; i <=35; i++){
            write(p[1], &i, sizeof(int));
        }
        close(p[1]);
    }
    wait(0);
    exit(0);
}
```

程序编写完成后，打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面加入一行： $U/_prime\

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util prime，即可进行自动评测：

```
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$  ./grade-lab-util
primes
make: “kernel/kernel”已是最新。
== Test primes == primes: OK (1.9s)
```

实验五 Find

在 xv6 上实现用户程序 find，即在目录树中查找名称与字符串匹配的所有文件，输出文件的相对路径。该程序的命令格式为“findpath file_name”。

这个实验的实现原理类似 ls，搜索本文件夹下所有文件，若名字相同则输出，若为文件夹则递归地进行搜索。

首先打开输入的目录，判断路径长度是否合法，如果不合法则打印错误信息并退出，如果合法则遍历目录下的所有文件。先跳过上级目录和当前目录，获取到完整路径之后，如果遍历到的是目录则递归，是文件则进行比较，如果与查找一致则输出即可。

下面是具体实现：

```
void find(char *dir, char *file) {
    char buf[512] = {0}, *p;
    int fd;
    struct dirent de;
    struct stat st;
```

```
    // 打开目录
    if ((fd = open(dir, 0)) < 0) {
        fprintf(2, "find: cannot open %s\n", dir);
        return;
    }
    // 判断路径长度
    if (strlen(dir) + 1 + DIRSIZ + 1 > sizeof(buf)) {
        fprintf(2, "find: path too long\n");
        close(fd);
        return;
    }

    strcpy(buf, dir);
    p = buf + strlen(buf);
    *p++ = '/';
    // 遍历目录下文档
    while (read(fd, &de, sizeof(de)) == sizeof(de)) {
        // 跳过当前目录和上级目录
        if (de.inum == 0 || strcmp(de.name, ".") == 0 || strcmp(de.name,
"..") == 0) {
            continue;
        }
        // 得到完整路径
        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0;
        // 获取当前文档的状态
        if (stat(buf, &st) < 0) {
            fprintf(2, "find: cannot stat %s\n", buf);
            continue;
        }
        // 是目录则递归遍历
        if (st.type == T_DIR) {
            find(buf, file);
        } else if (strcmp(de.name, file) == 0) {    //是文件则进行比较，若
与查找一致则输出
            printf("%s\n", buf);
        }
    }
    close(fd);
    return;
}

int main(int argc, char *argv[]) {
    struct stat st;
```

```
    if (argc != 3) {
        fprintf(2, "Usage: find dir file\n");
        exit(1);
    }
    // 获取查找目录的状态并判断是否为目录
    if (stat(argv[1], &st) < 0) {
        fprintf(2, "find: cannot stat %s\n", argv[1]);
        exit(2);
    }
    if (st.type != T_DIR) {
        fprintf(2, "find: '%s' is not a directory\n", argv[1]);
        exit(3);
    }
    find(argv[1], argv[2]);
    exit(0);
}
```

程序编写完成后，打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面加入一行： $U/_find\

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util find，即可进行自动评测：

```
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$ ./grade-lab-util f
ind
make: “kernel/kernel”已是最新。
== Test find, in current directory == find, in current directory: OK (0.9s)
== Test find, recursive == find, recursive: OK (1.0s)
```

实验六 Xargs

该部分实验重点之一在于 xargs 指令的用法,其中读取输入是通过标准输入进行的,在代码上体现为 read()函数。另一方面是参数的解析，下面代码考虑了 xargs 后面无参数时默认以 echo 作为指令。此外,另一个比较困难的地方在于"空格" "回车"以及""输入结束"三种情况的处理.网上有些代码没有考虑输入结束,也就是 read()返回 0 的情况.

本实验中我对"空格""回车".以及"输入结束"三种情况都进行处理.也就是在读取输入时会根据空格将参数进行划分

具体实现如下：

```
#define MAXLEN 32

int main(int argc, char *argv[]) {
    // 执行命令
    char *path = "echo";
    // 存放参数的数组和指针
    char buf[MAXLEN * MAXARG] = {0},*p;
    // 参数的指针数组
    char *params[MAXARG];
```

```
    // xargs 后命令所带的参数个数
    int oriParamCnt = 0;
    // 参数序号
    int paramIdx;
    // 参数长度
    int paramLen;
    int i;
    // 临时记录读取字符
    char ch;
    // read 读取长度，用于判断是否输入结束
    int res;

    // 参数数量大于 1
    if (argc > 1) {
        // 提取指令
        path = argv[1];
        // 设置参数，注意也需要带上指令的参数
        for (i = 1; i < argc; ++i) {
            params[oriParamCnt++] = argv[i];
        }
    } else {    //参数唯一,即只有 xargs
        // 即指令为 echo
        params[oriParamCnt++] = path;
    }

    // 后续参数起始序号
    paramIdx = oriParamCnt;
    p = buf;
    paramLen = 0;
    while (1) {
        res = read(0, p, 1);
        ch = *p;

        // 若读取的为一般字符
        if (res != 0 && ch != ' ' && ch != '\n') {
            ++paramLen;
            ++p;
            continue;
        }
        // 未读取成功，或者读取的是空格或回车
        // 计算参数起始位置
        params[paramIdx++] = p - paramLen;
        // 参数长度置 0
        paramLen = 0;
```

```
        // 设置字符结束符
        *p = 0;
        ++p;
        // 若读取的参数超过上限
        if (paramIdx == MAXARG && ch == ' ') {
            // 一直读到回车即下个命令为止
            while ((res = read(0, &ch, 1)) != 0) {
                if (ch == '\n') {
                    break;
                }
            }
        }
        // 若读取的不为空格，即 res==0||ch=='\n'
        if (ch != ' ') {
            // 创建子进程执行命令
            if (fork() == 0) {
                exec(path, params);
                exit(0);
            } else {
                // 父进程等待子进程
                wait((int *) 0);
                // 重置参数序号和指针
                paramIdx = oriParamCnt;
                p = buf;
            }
        }
        // 若输入读取完毕则退出
        if (res == 0) {
            break;
        }
    }
    exit(0);
}
```

程序编写完成后，打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面加入一行： $U/_xargs\

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util xargs，即可进行自动评测：

```
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$ ./grade-lab-util x
args
make: “kernel/kernel”已是最新。
== Test xargs == xargs: OK (1.2s)
```

在整个 Lab 的最后，运行 make grade，结果如下：

```
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (2.9s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.5s)
== 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 (1.2s)
== Test time ==
time: OK
Score: 100/100
```

Lab2 System Call

在上一个实验中，我们使用系统调用编写了一些实用程序。在本实验中，我们将向 xv6 添加一些新的系统调用，这将帮助我们理解它们是如何工作的，并将向我们展示 xv6 内核的一些内部结构。我们将在以后的实验中添加更多的系统调用。

实验一 System call tracing

为了方便对系统调用进行 debug ，我们引入一个新的系统调用 trace ，用以追踪（打印）用 户程序使用系统调用的情况。该系统调用接受一个参数，被称为 mask（掩码），用以设置被追踪 的系统调用：例如，若我们想要追踪 fork 系统调用，则需要调用 trace(1 << SYS_fork) ，其中 SYS_fork 是 fork 系统调用的编号。该进程调用过 trace(1 << SYS_fork) 后，如果该进程后续 调用了 fork 系统调用，调用 fork 时内核则会打印形如 : syscall fork -> 的信息（但无需打印传入系统调用的参数）。

这个实验需要我们着手修改 xv6 的内核。在修改内核之前，我们需要先将 $U/_trace 加入到 Makefile 中的 UPROGS 环境变量中。首先按照下面的步骤将 trace 系统调用加入到户态库和内核中：

1. 在 user/user.h 中的系统调用部分加入一行 int trace(int); 作为 trace 系统调用在用 户态的入口。
2. 在 user/usys.pl 中的系统调用部分加入一行 entry("trace"); 用以生成调用 trace(,→int) 入口时在用户态执行的汇编代码（这段代码被称为 stubs ）。
3. 在 kernel/syscall.h 中#define SYS_trace 22 为 trace 分配一个系统调用的编号。

接下来，我们找到文件 kernel/proc.h，找到进程的数据结构 struct proc，在其中加入一行 int tracemask。

为了将 trace(int) 的参数保存到 struct proc 中，我们需要在 kernel/sysproc.c 中实 现 sys_trace(void) 函数，argint(0, &mask)，用于获取用户传入系统调用的第一个参数，而 myproc() 则返回当前进程控制块的指针，将其 tracemask 赋值为获取到的参数即可代码如下：

```
uint64
sys_trace(void)
{
  int mask;
  argint(0, &mask);
  myproc()->tracemask = mask;
  return 0;
}
```

然后我们需要修改 kernel/syscall.c 中的 syscall(void) ，使其能够根据 tracemask 打印所需要的信息。首先定义一个字符串常量数组，保存各系统调用的名称：

```
const char *syscallnames[SYS_CALL_NUM + 1] = {
    "",
```

```
    "fork",
    "exit",
    "wait",
    "pipe",
    "read",
    "kill",
    "exec",
    "fstat",
    "chdir",
    "dup",
    "getpid",
    "sbrk",
    "sleep",
    "uptime",
    "open",
    "write",
    "mknod",
    "unlink",
    "link",
    "mkdir",
    "close",
    "trace",
    "sys_sysinfo",
};
```

然后修改 syscall(void) ，在调用系统调用后，增加如下 if 语句的内容：

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

    if (p->tracemask >> num & 1)
    {
      printf("%d: syscall %s -> %d\n",
             p->pid, syscallnames[num], p->trapframe->a0);
    }
```

最后，为了让 fork 后的子进程能够继承 trace 的 tracemask ，需要在 kernel/proc.c 中 的 fork(void) 中加入复制 tracemask 的语句

```
// copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // copy trace mask
  np->tracemask = p->tracemask;

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;
```

trace 系统调用实现完成，编译并启动 xv6 后，在 shell 中运行 trace 32 grep hello README ，即可看到预期的输出。

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-syscall trace ，即可进行自动评测，结果如下：

```
== Test trace 32 grep == trace 32 grep: OK (2.0s)
== Test trace all grep == trace all grep: OK (0.7s)
== Test trace nothing == trace nothing: OK (1.1s)
== Test trace children == trace children: OK (12.4s)
```

实验二 Sysinfo

这个实验要求我们实现一个名为 sysinfo 系统调用，用于获取 xv6 运行时的信息。具体来说，xv6 已经为我们定义好了一个结构体 struct sysinfo（在源码 kernel/sysinfo.h ）中，我们的 sysinfo 系统调用接受一个参数，即指向该结构体的指针，然后将对应的内容填入该结构体（freemem 字段填写空闲的内存空间字节数，nproc 字段填写状态为未使用（UNUSED ）的进程数）

首先和上面类似，修改 Makefile 中的 UPROGS 环境变量，将 $U/_sysinfotest 加入到其 中。然后按照上面的流程添加 sysinfo 系统调用的入口，接着在 user/user.h 中添加 struct sysinfo 的声明:

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

为了获得空闲内存大小，我们需要在 kernel/kalloc.c 实现一个函数。由于 xv6 管理内存空闲空间使用的是空闲链表，参照链表的结构后，只需遍历链表并计算数量，然后乘以页面大小即可。

```
// Return the amount of free memory.
int getfreemem(void)
{
  int count = 0;
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  while (r)
  {
    count++;
    r = r->next;
  }
  release(&kmem.lock);
  return count * PGSIZE;
}
```

统计空闲进程控制块的数量的函数的实现也类似，只是进程控制块是用静态数组管理的， 故而只需要用一个循环遍历该数组即可:

```
// Return the number of non-UNUSED procs in the process table.
int getnproc(void)
{
```

```
  struct proc *p;
  int count = 0;
  for (p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);
    if (p->state != UNUSED)
    {
      count++;
      release(&p->lock);
    }
    else
    {
      release(&p->lock);
    }
  }
  return count;
}
```

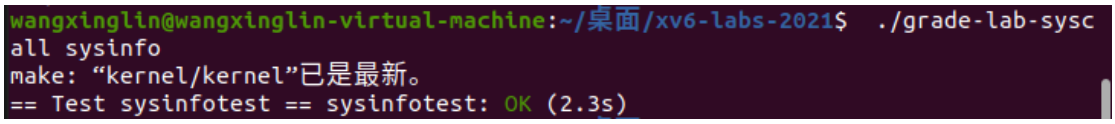
下面将 struct sysinfo 拷贝到用户态进程中，为此我们实现一个函数 sys_sysinfo(void),其中，copyout 函数接受进程的页表，并将内核态中的一段数据拷贝到进程内存空间中的地址处:

```
extern int getnproc(void);
extern int getfreemem(void);
uint64
sys_sysinfo(void)
{
  struct proc *p = myproc();
  struct sysinfo st;
  uint64 addr; // user pointer to struct stat
  st.freemem = getfreemem();
  st.nproc = getnproc();
  if (argaddr(0, &addr) < 0)
    return -1;
  if (copyout(p->pagetable, addr, (char *)&st, sizeof(st)) < 0)
    return -1;
  return 0;
}
```

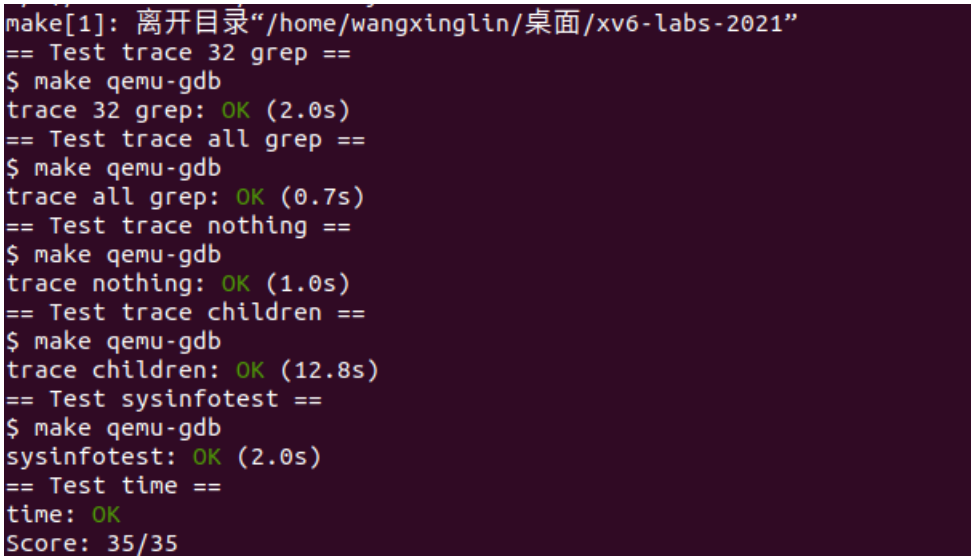
到此 sysinfo 系统调用实现完成，编译并启动 xv6 后，在 shell 中运行 sysinfotest ，结果如下（符合预期）:

```
hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
```

最后使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-syscall sysinfo ，即可进行自动评测，结果如下：



在整个 Lab 的最后，运行 make grade，结果如下：



Lab3 Page table

在本实验中，我们将探索页表并修改它们，以加快某些系统调用并检测已访问的页面。

实验一 Speed up system calls

一些操作系统(例如 Linux)通过在用户空间和内核之间共享只读区域中的数据来加速某些系统调用。这就消除了内核交叉的需要执行这些系统调用。为了帮助我们了解如何将映射插入页表，第一个任务是为 xv6 中的 getpid（）系统调用实现这种优化。

首先修改 kernel/proc.c 中的 proc_ ,→pagetable(struct proc *p) ，即用于为新创建的进程分配页面的函数：

```
pagetable_t
proc_pagetable(struct proc *p)
{
  pagetable_t pagetable;

  // An empty page table.
  pagetable = uvmcreate();
  if(pagetable == 0)
    return 0;

  //map a user read only page at USYSCALL, for optimization for the
getpid()
  if(mappages(pagetable, USYSCALL, PGSIZE,
              (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }

  // map the trampoline code (for system call return)
  // at the highest user virtual address.
  // only the supervisor uses it, on the way
  // to/from user space, so not PTE_U.
  if(mappages(pagetable, TRAMPOLINE, PGSIZE,
              (uint64)trampoline, PTE_R | PTE_X) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }

  // map the trapframe just below TRAMPOLINE, for trampoline.S.
  if(mappages(pagetable, TRAPFRAME, PGSIZE,
              (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
```

```
    uvmfree(pagetable, 0);
    return 0;
  }

  return pagetable;
}
```

页面映射时，需要设定其权限为只读，故权限位为 PTE_R | PTE_U ，此后在 allocproc() 中添加初始化该页面，向其中写入数据结构的代码：

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

在完成页面的初始化后，进程应当得以正常运行，但需要记得在进程结束后释放分配的页面，该动作在 freeproc() 中进行，添加的代码如下：

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

最后，运行 pgtbltest，结果如下：

```
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
```

实验二 Print a pagetable

为了方便将页表的内容可视化，我们需要实现一个名为 vmprint() 的函数，其接受一个 pagetable_t 类型的参数，并按固定的格式打印页表。

我们仿照 freewalk 函数的递归方式，对该函数略做修改，可以构造遍历页表并按层数格式打印的函数 vmprintwalk(pagetable_t pagetable ,int depth)：

```
void
vmprintwalk(pagetable_t pagetable, int depth)
{
  // 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) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      // this PTE points to a lower-level page table.
      for (int n = 0; n < depth; n++)
        printf(" ..");
      printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
      uint64 child = PTE2PA(pte);
      vmprintwalk((pagetable_t)child, depth+1);
```

```
    } else if(pte & PTE_V){
      for (int n = 0; n < depth; n++)
        printf(" ..");
      printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
    }
  }
}
```

然后实现 vmprint 打印页表：

```
void
vmprint(pagetable_t pagetable)
{
  printf("page table %p\n",pagetable);
  vmprintwalk(pagetable,1);
}
```

最后按照要求在 kernel/exec.c 中的 exec() 函数的 return argc 前加入 if(p->pid ==1)vmprint(p->pagetable) ：

```
if(p->pid==1) vmprint(p->pagetable);
return argc; // this ends up in a0, the first argument to main(argc, argv)
```

运行 make qemu，输出如下：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
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
 .. .. ..509: pte 0x0000000021fddc13 pa 0x0000000087f77000
 .. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
 .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

实验三 Detecting which pages have been accessed

现代很多编程语言都具备了内存垃圾回收的功能，利用页表的硬件机制（访问位）和操作系统相结合，在 xv6 中添加一 个系统调用 pgaccess() 用以读取页表的访问位并传递给用户态程序，从而告知他们一些内存 页面自上次检查以来的访问情况，是一个较好的方法。

首先，按照前文提及的方法添加系统调用以及其在 kernel/sysproc.c 中的实现 sys_ pgaccess() ，并使用 argint() 和 argaddr() 获取传入的参数：

```
#ifdef LAB_PGTBL
extern pte_t *walk(pagetable_t, uint64, int);
int sys_pgaccess(void)
```

```
{
  // lab pgtbl: your code here.
  uint64 srcva, st;
  int len;
  uint64 buf = 0;
  struct proc *p = myproc();

  acquire(&p->lock);

  argaddr(0, &srcva);
  argint(1, &len);
  argaddr(2, &st);
  if ((len > 64) || (len < 1))
    return -1;
  pte_t *pte;
  for (int i = 0; i < len; i++)
  {
    pte = walk(p->pagetable, srcva + i * PGSIZE, 0);
    if(pte == 0){
      return -1;
    }
    if((*pte & PTE_V) == 0){
      return -1;
    }
    if((*pte & PTE_U) == 0){
      return -1;
    }
    if(*pte & PTE_A){
      *pte = *pte & ~PTE_A;
      buf |= (1 << i);
    }
  }
  release(&p->lock);
  copyout(p->pagetable, st, (char *)&buf, ((len -1) / 8) + 1);
  return 0;
}
#endif
```

之后，对获取的参数进行预处理后，对每个需要检查的页面，使用 kernel/vm.c 中提供的 walk(pagetable_t, uint64, int) 来获取其页表项，重置该页表的 PTE_A 访问位，并使 用位运算将其访问状态置入临时位向量中；最后将临时位向量拷贝至用户内存空间指定的地址中。

```
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
```

```
  // printf("walk: walking at va %p\n",va);
  if(va >= MAXVA)
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

```
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
```

整个实验的最后，运行 make grade，结果如下：

```
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
== Test pgtbltest ==
$ make qemu-gdb
(2.7s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.6s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(142.0s)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

Lab4 Traps

本实验探讨如何使用陷阱实现系统调用。我们将首先使用堆栈做一个热身练习，然后实现一个用户级陷阱处理的示例。

实验一 RISC-V assembly

理解一点 RISC-V 汇编是很重要的，这是在 6.004 中接触到的。存在一个文件用户/call.C 在的 x6 的 repo.make fs.img 对其进行编译，并在 user/call.asm 中生成程序的可读汇编版本。读取调用中的代码，asm 表示函数 g， t 和 main。RISC-V 的使用手册在参考页面。这里有一些我们需要回答的问题(将答案存储在文件 answers-traps.txt 中)。

我们首先需要使用 make 将 user/call.c 源文件编译为对应的目标代码，在这个过程中， xv6 的 Makefile 会自动生成反汇编后的代码。编译完成后，打开新生成的 user/call.asm，回 答 xv6 实验手册中提出的问题。具体的答案在 answers-traps.txt 中。

实验二 Backtrace

为 gdb 等调试工具很难进行内核态的调试，为了便于调试内核，在内核中实现 backtrace 函数是大有裨益的。我们需要在 kernel/printf.c 实现 backtrace() ，并且在 sys_sleep 调 用中插入一个 backtrace() 函数，用以打印当时的栈。

我们的 backtrace() 函数只需打印每次保存的 ra 即可，然后根据 s0 的值寻找上一个栈 帧，然后继续打印保存的 ra ，如此往复直到到达栈底。根据 xv6 实验指导的提醒，我们首先 在 kernel/riscv.h 中加入以下内联汇编函数用于读取当前的 s0 寄存器：

```
static inline uint64
r_fp()
{
    uint64 x;
    asm volatile("mv %0, s0" : "=r" (x) );
    return x;
}
```

利用上面的思想，在 kernel/printf.c 实现 backtrace()，

```
void
backtrace(void)
{
  printf("backtrace:\n");
  for (uint64 *fp = (uint64 *)r_fp(); (uint64)fp <
PGROUNDUP((uint64)fp); fp = (uint64 *)(*(fp-2)))
  {
    printf("%p\n",*(fp-1));
  }
}
```

实现完成后，在 sys_sleep 调用中插入一个 backtrace() 函数即可。

实验三 Alarm

做本实验的目的是为了让一个进程能够在消耗一定的 CPU 时间后被告知。因此我们需要实现 sigalarm(n, fn) 系统调用。用户程序执行 sigalarm(n, fn) 系统调用后，将会在其每消耗 n 个 tick 的 CPU 时 间后被中断并且运行其给定的函数 fn 。这个系统调用对于不希望占用过多 CPU 时间，或希望 进行周期性操作的程序来说是及其有用的。此外，实现 sigalarm(n, fn) 系统调用有利于我 们学会如何构建用户态的中断和中断处理程序，这样就可以使用类似的方法使用户态程序可 以处理缺页等常见的异常。

正确实现 sigalarm(n, fn) 系统调用面临以下几个问题： 1.如何保存每个进程的 sigalarm 参数； 2.如何并更新每个已经消耗的 tick 数； 3.如何在已经消耗的 tick 数符合要求时使进程调用中断处理过程 fn ； 4.中断处理过程 fn 执行完毕后如何返回。

首先，我们 kernel/syscall.h， kernel/syscall.c， user/usys.pl 和 user/user.h 文件中添加下面两个函数的声明：

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

下面解决第一个问题，我们需要在每个进程的控 制块的数据结构中加入存储 sigalarm(n, fn) 中参数的项，即在 kernel/proc.h 的 struct proc 中加入如下的项：

```
int alarminterval;          // sys_sigalarm() alarm interval in ticks
  int alarmticks;             // sys_sigalarm() alarm interval in ticks
  void (*alarmhandler)();      // sys_sigalarm() pointer to the alarm
handler function
  struct trapframe alarmtrapframe;// for saving registers
  int sigreturned;
```

然后在 kernel/proc.c 的 allocproc(void) 中对变量进行初始化：

```
// Initialize alarmticks
  p->alarmticks = 0;
  p->alarminterval = 0;
  p->sigreturned = 1;
```

在调用 sigalarm(n, fn) 系统调用时，执行的 kernel/sysproc.c 中的 sys_sigalarm() 需根据传入的参数设置 struct proc 的对应项：

```
uint64
sys_sigalarm(void)
{
  int ticks;
  uint64 handler;
  struct proc *p = myproc();
  if(argint(0, &ticks) < 0 || argaddr(1, &handler) < 0)
    return -1;
  p->alarminterval = ticks;
  p->alarmhandler = (void (*)())handler;
```

```
  p->alarmticks = 0;
  return 0;
}
```

下面来实现 usertrap 函数，这也是本实验中最为核心的函数：

```
//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else {
```

```
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(),
p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
  {
    p->alarmticks += 1;
    if ((p->alarmticks >= p->alarminterval) && (p->alarminterval > 0))
    {
      p->alarmticks = 0;
      if (p->sigreturned == 1)
      {
        p->alarmtrapframe = *(p->trapframe);
        p->trapframe->epc = (uint64)p->alarmhandler;
        p->sigreturned = 0;
        usertrapret();
      }
    }
    yield();
  }

  usertrapret();
}
```

这样整个触发 sigalarm 的过程便成功完成了。

最后，解决下面这个问题：当传入的 fn 执行完毕后，如何返回到进程正常的执行过程中。由于之前我们保存了进程执行的上下文，我们需要在内核态对应的 sys_sigreturn() 中将备份的上下文恢复，然后返回用户态，该实现放在 kernel/sysproc.c 中：

```
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  p->sigreturned = 1;
  *(p->trapframe) = p->alarmtrapframe;
  usertrapret();
  return 0;
}
```

整个实验的最后，运行 make grade，结果如下：

```
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (2.6s)
== Test running alarmtest ==
$ make qemu-gdb
(3.6s)
== 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 (141.6s)
== Test time ==
time: OK
Score: 85/85
```

Lab5 Copy-on-Write

实验 Implement copy-on write

虚拟内存提供了一种间接性:内核可以通过将 pte 标记为无效或只读来拦截内存引用，从而导致页面错误，并且可以通过修改 pte 来改变地址的含义。在计算机系统中有一种说法，任何系统问题都可以通过某种程度的间接解决。延迟分配实验室提供了一个示例。本实验探讨另一个例子:copy-on 写 fork。

xv6 中的 fork 系统调用将父进程的所有用户空间内存复制到子进程中。如果父节点很大，复制可能需要很长时间。更糟糕的是，这些工作通常都被浪费了;例如，子进程中的 fork()后跟 exec)将导致子进程放弃复制的内存，可能根本不会使用大部分内存。另一方面，如果父节点和子节点都使用一个页面，并且其中一个或两个都写入该页面，则确实需要副本。

对此，我们的解决方案如下：写时复制(copy-on-write, COW)分支的目标是推迟为子进程分配和复制物理内存页，直到实际需要这些副本时再进行。COW fork()只为子进程创建一个页表，而用户内存的 pte 则指向父进程的物理页面。COW fork()将父级和子级中的所有用户 pte 标记为不可写。当任一进程试图写这些 COW 页时，CPU 将强制出现页错误。内核页面故障处理程序检测到这种情况，为故障进程分配一个物理内存页面，将原始页面复制到新页面中，并修改故障进程中的相关 PTE 以引用新页面，这次将 PTE 标记为可写。当页面错误处理程序返回时，用户进程将能够写入该页的副本。Cow fork()使得释放实现用户内存的物理页面变得有点棘手。一个给定的物理页可能被多个进程的页表引用，并且应该只被释放当最后一个引用消失时。

在这个实验中，我们的任务是在 xv6 内核中实现写时复制分叉。如果修改后的内核成功地执行了 cowtest 和 usertest 程序，那么就完成了。

由于这个实验涉及到将进程、分页和中断综合起来，故整个实现较为困难。我们首先修 改物理内存分配器，由于 COW 中一个物理页面不仅被映射给一个页表，故而我们需要一个数 据结构维护其引用的数量，故而在 kalloc.c 中添加以下变量：

```
  struct spinlock lock;
```

然后在初始化内存管理器的 kinit 中添加初始化引用计数锁的语句：

```
void
kinit()
{
  initlock(&kmem.lock, "kmem");
  freerange(end, (void*)PHYSTOP);
}
```

之后更改释放物理页面的代码，使得每次释放仅将引用计数减一，直到引用计数为 0 后才真正释放页面：

```
void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
```

```
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}
```

到此，我们已经完成了修改物理内存分配器的准备工作。

下面，我们来将父进程的页面映射给子进程，并给父进程和子进程的页表设为只读。

我们需要对 uvmcopy 进行修改，使用 mappages 只映射页面、增加引用计数并修改权限为只读，而不分配页面：

```
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);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;

 err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

在删除页面映射的过程 uvmunmap 中也要加入对应的操作：

```
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
  uint64 a;
  pte_t *pte;
```

```
  if((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");

  for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    if((*pte & PTE_V) == 0)
      panic("uvmunmap: not mapped");
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

到此，父进程的页面映射给子进程，并给父进程和子进程的页表设为只读的工作完成。如果父进程和子进程都在 fork 后没有写入内存，则一切相安无事；而若其中一个进程尝试写入内存，则会引发页面权限错误，从而产生一个中断最终调用 usertrap()。

最后来实现剩余的部分。我们通过 riscv.h 中提供的读寄存器函数获取这两个寄存器的内容后，就可以稍作判断，若违例访问确为 COW 引起的，就可以分配页面、复制数据并进行释放页面、修改页面权限和标志等后续工作，否则杀死进程并进行善后工作。需要将 trap.c 中的 usertrap 函数做如下修改：

```
else if((which_dev = devintr()) != 0){
    // ok
  } else if (r_scause() == 12 || r_scause() == 15){
    // deal with cow pages
    pte_t *pte;
    uint64 pa, va;
    // uint64 va;
    uint flags;
    char *mem;

    va = r_stval();
    if(va >= MAXVA)
    {
      p->killed = 1;
      exit(-1);
    }

    if((pte = walk(p->pagetable, va, 0)) == 0)
    {
```

```
      p->killed = 1;
      exit(-1);
    }
    if((*pte & PTE_V) == 0)
    {
      p->killed = 1;
      exit(-1);
    }
    if((*pte & PTE_COW) == 0)
    {
      p->killed = 1;
      exit(-1);
    }

    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte) | PTE_W;
    flags &= ~(PTE_COW);
    if((mem = kalloc()) == 0)
    {
      p->killed = 1;
      exit(-1);
    }
    memmove(mem, (char*)pa, PGSIZE);
    uvmunmap(p->pagetable, PGROUNDDOWN(va), 1, 1);
    if(mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)mem,
flags) != 0){
      kfree(mem);
      panic("cowhandler: mappages failed");
    }
  }
  else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(),
p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();
```

```
  usertrapret();
```

到此 COW fork 的机制基本完成，但 xv6 的实验手册还提醒我们，需要修改 copyout() 使得其能够适应 COW 机制。我们判断一个页面是否为 COW 页面使用的是上面提到的 #define PTE_COW (1L << 8) 标志位，然后使用与处理违例访问相似的方法，进行分配页面、复制数据并进行释放页面、修改页面权限和标志等后续工作。

```
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(dstva >= MAXVA)
      return -1;
    pte = walk(pagetable, va0, 0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    if (*pte & PTE_COW)
    {
      uint flags;
      char *mem;
      flags = PTE_FLAGS(*pte) | PTE_W;
      flags &= ~(PTE_COW);
      if((mem = kalloc()) == 0)
      {
        // printf("copyout: kalloc failed, killed the process\n");
        return -1;
      }
      memmove(mem, (char*)pa0, PGSIZE);
      uvmunmap(pagetable, va0, 1, 1);
      if(mappages(pagetable, va0, PGSIZE, (uint64)mem, flags) != 0)
      {
        kfree(mem);
        panic("copyout: mappages failed");
      }
    }
    pa0 = walkaddr(pagetable, va0);
    memmove((void *)(pa0 + (dstva - va0)), src, n);
```

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

到此，整个实验宣告完成。

整个 Lab 的最后，运行 make grade，结果如下：

```
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
== Test running cowtest ==
$ make qemu-gdb
(8.0s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(133.9s)
== 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
```

Lab6 Multithreading

本实验将使你熟悉多线程。我们将在用户级线程包中实现线程之间的切换，使用多个线程来加速程序，并实现屏障。

实验一 Uthread: switching between threads

在本练习中，我们将为用户级线程系统设计上下文切换机制，然后实现它。首先，xv6 有两个文件 user/uthread.c 和 user/uthread_switch.s，并在 Makefile 中添加一条规则来构建一个线程程序。uthread.c 包含了大部分用户级线程包，以及三个简单测试线程的代码。线程包缺少一些创建线程和在线程之间切换的代码。

首先我们查看 user/uthread.c 中关于线程的一些数据结构：

```
struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct context context;       // uthread_switch() here to switch the
thread
};
struct thread all_thread[MAX_THREAD];
struct thread *current_thread;
extern void thread_switch(uint64, uint64);
```

我们需要将其修改为：

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

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

```
struct thread {
  char       stack[STACK_SIZE]; //线程的栈
  int        state;             //线程的状态
  struct context context;       //保存每个线程的上下文
};
struct thread all_thread[MAX_THREAD];
struct thread *current_thread;
extern void thread_switch(uint64, uint64);
```

其中，字节数组用作线程的栈，整数用于表示线程的状态，同时新定义一个数据结构用于保存每个线程的上下文。参考 xv6 实验手册的提示，除了 sp 、s0 和 ra 寄存器，我们只需要保存 callee-saved 寄存器， 因此构造了上面的 struct context 结构体。

随后，我们仿照 kernel/trampoline.S 的结构，按照 struct context 各项在内存中的位置，在 user/uthread_switch.S 中加入相应的的代码，上下文切换功能完成。

接下来我们来实现创建线程和调度线程的功能。首先是创建线程，我们实现如下：

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

  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  t->context.ra = (uint64)func;
  t->context.sp = (uint64)&t->stack[STACK_SIZE];
}
```

调度线程，我们实现如下：

```
void thread_schedule(void)
{
  struct thread *t, *next_thread;

  /* Find another runnable thread. */
  next_thread = 0;
  t = current_thread + 1;
  for(int i = 0; i < MAX_THREAD; i++){
    if(t >= all_thread + MAX_THREAD)
      t = all_thread;
    if(t->state == RUNNABLE) {
      next_thread = t;
      break;
```

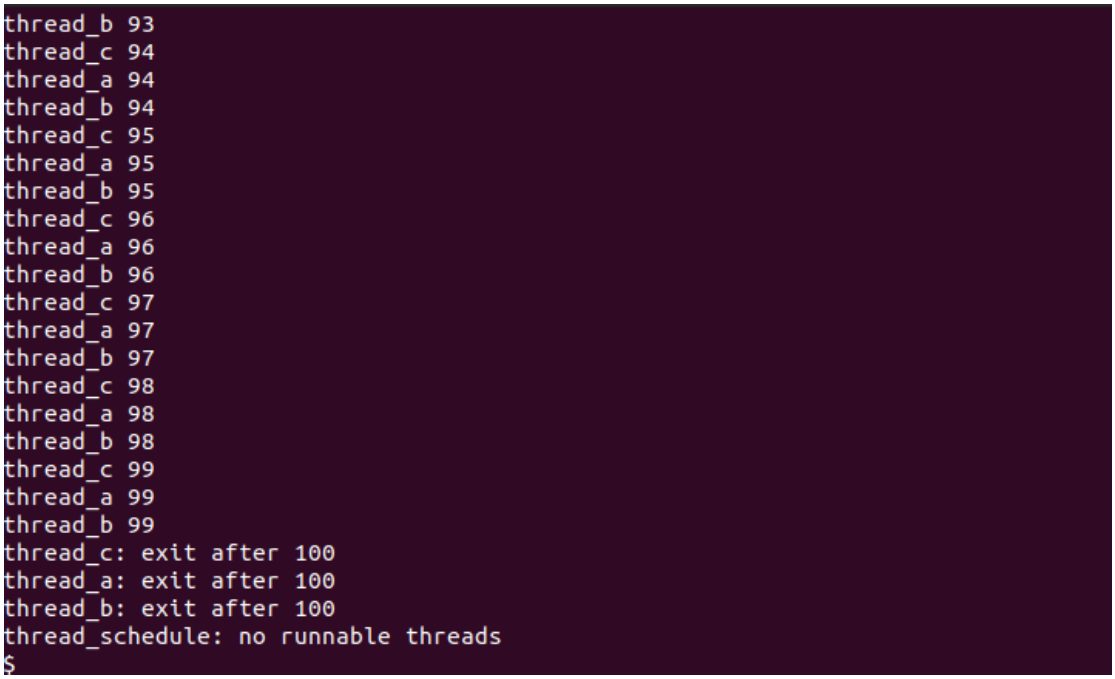
```
    }
    t = t + 1;
  }

  if (next_thread == 0) {
    printf("thread_schedule: no runnable threads\n");
    exit(-1);
  }

  if (current_thread != next_thread) {
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    thread_switch((uint64)&t->context,
(uint64)&current_thread->context);

  } else
    next_thread = 0;
}
```

最后编译运行 xv6 ，然后执行 uthread ，便可看到各线程有序地执行，如下图所示：



实验二 Using thread

在本作业中，我们将探索使用哈希表进行线程和锁并行编程。我们应该在具有多个内核的真正的 Linux 或 MacOS 计算机(不是 xv6，也不是 qemu)上执行此任务。大多数最新的笔记本电脑都有多核处理器。

这个赋值使用 UNIX pthread 线程库。我们可以通过 man pthreads 从手册页找到有关它的信息，也可以在 web 上查找。

本实验中，我们需要修改 notxv6/ph.c，使之在多线程读写一个哈希表的情况下能够产生正确的结果。

首先定义一个保护哈希表的互斥锁，然后在开始线程前对其进行初始化：

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

最后在写入操作哈希表时加上获取锁的操作，并在写入完成后释放锁：

```
static
void put(int key, int value)
{

  int i = key % NBUCKET;

  // 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.
    pthread_mutex_lock(&lock);
    insert(key, value, &table[i], table[i]);
    pthread_mutex_unlock(&lock);
  }

}
```

使用 make ph 编译 notxv6/ph.c ，运行 ./ph 2 ，则发现没有读出的数据缺失，如下图所示：

```
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$ ./ph2
bash: ./ph2: 没有那个文件或目录
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$ ./ph 2
100000 puts, 3.143 seconds, 31814 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 6.615 seconds, 30235 gets/second
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$
```

实验三 Barrier

在本作业中，我们将实现一个屏障:应用程序中的一个点，所有参与的线程必须等待，直到所有其他参与的线程也到达该点。我们将使用 pthread 条件变量，这是一种序列协调技术，类似于 xv6 的睡眠和唤醒。

我们应该在一台真正的计算机(不是 xv6，也不是 qemu)上完成这个任务。

为实现线程屏障，需要维护一个互斥锁、一个条件变量、用以记录到达线程屏障的线程数 的整数和记录线程屏障轮数的整数。在初始化用 barrier_init() 中，互斥锁、条件变量及 nthread 被初始化。此后在某个线程到达 barrier() 时，需要获取互斥锁进而修改 nthread。 当 nthread 与预定的值相等时，nthread 清零，轮数加一，同时唤醒所有等待中的线程。最后要释放互斥锁。

```
static void
barrier()
{
  pthread_mutex_lock(&bstate.barrier_mutex);
  bstate.nthread ++;
  //当 nthread 与预定的值相等时，nthread 清零，轮数加一
  if (bstate.nthread == nthread)
  {
    bstate.round++;
    bstate.nthread = 0;
    pthread_cond_broadcast(&bstate.barrier_cond);
  } else {
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }
  //释放互斥锁
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

最后运行结果如下：

```
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
wangxinglin@wangxinglin-virtual-machine:~/桌面/xv6-labs-2021$ ./barrier 2
OK; passed
```

整个实验的最后，make grade，结果如下：

```
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
== Test uthread ==
$ make qemu-gdb
uthread: OK (2.6s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录“/home/wangxinglin/桌面/xv6-labs-2021”
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
ph_safe: OK (9.6s)
== Test ph_fast == make[1]: 进入目录“/home/wangxinglin/桌面/xv6-labs-2021”
make[1]: “ph”已是最新。
```


make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
ph_fast: OK (22.4s)
== Test barrier == make[1]: 进入目录“/home/wangxinglin/桌面/xv6-labs-2021”
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
barrier: OK (12.7s)
== Test time ==
time: OK
Score: 60/60

Lab7 Networking

在本实验中，我们将为网络接口卡(NIC)编写一个 xv6 设备驱动程序。获取实验室的 xv6 源代码并检查 net 分支。

背景 Background

我们将使用一个名为 E1000 的网络设备来处理网络通信。对于 xv6(以及我们编写的驱动程序)，E1000 看起来就像连接到真正的以太网局域网(LAN)的真正硬件。实际上，驱动程序将与之对话的 E1000 是由 qemu 提供的仿真，它连接到一个也由 qemu 仿真的局域网。在这个模拟的 LAN 上，xv6 的 IP 地址为 10.0.2.15。Qemu 还安排运行 Qemu 的计算机出现在 IP 地址为 10.0.2.2 的局域网中。当 xv6 使用 E1000 向 10.0.2.2 发送数据包时，qemu 将数据包发送到运行 qemu 的计算机上的适当应用程序。我们将使用 QEMU 的“用户模式网络堆栈”。我们已经更新了 Makefile 以启用 QEMU 的用户模式网络堆栈和 E1000 网卡。Makefile 配置 QEMU 将所有传入和传出的数据包记录到文件数据包中。Pcap 在实验室目录。检查这些记录以确认 xv6 正在发送和接收我们期望的数据包可能会有所帮助。为了这个实验，我们向 xv6 存储库添加了一些文件。文件 kernel/e1000.c 包含 E1000 的初始化代码，以及发送和接收数据包的空函数，我们将填写。kernel/e1000_dev.h 包含由 E1000 定义的寄存器和标志位的定义，并在 Intel E1000 软件开发人员手册中描述。kernel/net.h 包含一个简单的网络栈，实现 IP UDP 和 ARP 协议。这些文件还包含用于保存数据包的灵活数据结构(称为 nbuf)的代码。最后，kernel/ PCI .c 包含在 xv6 启动时在 PCI 总线上搜索 E1000 卡的代码。

实验一 Your Job 实现 Intel E1000 网卡驱动

在开始编写驱动程序前，我们需要获取 Intel 提供的关于 E1000 网卡的驱动的开发者文档 Intel E1000 Software Developer’s Manual 1 ，其中包含了关于该网卡硬件特性和工作机制的说明。

我们主要需要实现 kernel/e1000.c 中的两个函数：用于发送 数据包的 e1000_transmit() 和用于接收数据包的 e1000_recv() 。

在设计的数据结构中，有两个环形缓冲区 tx_ring 和 rx_ring 。我们需要将需要发送的数据包放入环形缓冲区中，设置好对应的参数并更新管理缓冲区的寄存器，即可视为完成了数据包的发送。此后网卡的硬件会自动在合适的时间将我们放入的数据包按照配置发送出去。

我们首先来实现第一个函数 e1000_transmit()：

```
Int e1000_transmit(struct mbuf *m){
  acquire(&e1000_lock);
  uint32 idx = regs[E1000_TDT];
  // ask the E1000 for the TX ring index
  // at which it's expecting the next packet,
  // by reading the E1000_TDT control register.
  if (tx_ring[idx].status != E1000_TXD_STAT_DD)
  {
```

```
    // 检查 tx_ring 是否溢出
    // 如果 E1000_TDT 索引的描述符中没有设置 E1000_TXD_STAT_DD，则 E1000 没有
完成相应的先前传输请求，返回错误。
    printf("e1000_transmit: tx queue full\n");
    __sync_synchronize();
    release(&e1000_lock);

    return -1;
  }
  else {
    // 否则，调用 mbuffree 释放最后一个 mbuf
    // that was transmitted from that descriptor (if there was one).
    if (tx_mbufs[idx] != 0){
      mbuffree(tx_mbufs[idx]);
    }
    // m->head 指向内存中 packet 的内容
    tx_ring[idx].addr = (uint64) m->head;
    // m->len 是 packet 的长度
    tx_ring[idx].length = (uint16) m->len;
    tx_ring[idx].cso = 0;
    tx_ring[idx].css = 0;
    tx_ring[idx].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
    tx_mbufs[idx] = m;

    regs[E1000_TDT] = (regs[E1000_TDT] + 1) % TX_RING_SIZE;
  }
  __sync_synchronize();

  release(&e1000_lock);

  return 0;
}
```

下面再来实现第二个函数 e1000_recv()：

```
extern void net_rx(struct mbuf *);
static void
e1000_recv(void)
{
  uint32 idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
  struct rx_desc* dest = &rx_ring[idx];
  // 通过检查描述符状态部分中的 E1000_RXD_STAT_DD 位来检查新数据包是否可用
  while (rx_ring[idx].status & E1000_RXD_STAT_DD){
    acquire(&e1000_lock);

    struct mbuf *buf = rx_mbufs[idx];
```

```
    mbufput(buf, dest->length);
    if (!(rx_mbufs[idx] = mbufalloc(0)))
      panic("mbuf alloc failed");
    dest->addr = (uint64)rx_mbufs[idx]->head;
    dest->status = 0;
    regs[E1000_RDT] = idx;
    __sync_synchronize();

    release(&e1000_lock);

    net_rx(buf);
    idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    dest = &rx_ring[idx];
  }

}
```

这里同样需要加锁保证互斥，并使用 __sync_synchronize 保证内存访问顺序，否则会使 接收数据包的顺序产生变化。

最后，make grade，结果如下：

```
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
== Test running nettests ==
$ make qemu-gdb
(3.2s)
== 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: FAIL
    ...
        testing single-process pings: OK
        testing multi-process pings: OK
        testing DNS
        Didn't receive an arecord
        $ qemu-system-riscv64: terminating on signal 15 from pid 11952 (make)
    MISSING '^DNS OK$'
== Test time ==
time: OK
Score: 81/100
```

DNS 的 test 没有通过，其余通过。

Lab8 Lock

在本实验中，您将获得重新设计代码以增加并行性的经验。多核机器上并行性差的一个常见症状是高锁争用。提高并行性通常需要改变数据结构和锁定策略，以减少争用。您将为 xv6 内存分配器和块缓存执行此操作。

实验一 Memory allocator

程序 user/kalloctest 强调 xv6 的内存分配器:三个进程增加和缩小它们的地址空间，导致对 kalloc 和 kfree 的多次调用。Kalloc 和 kfree 获取 kmem.lock。对于 kmem 锁和其他一些锁，Kalloctest 打印(作为“#fetch-and-add”)由于试图获取另一个核心已经持有的锁而导致的获取中的循环迭代次数。acquire 中的循环迭代次数是锁争用的粗略度量。而 acquire() 时被阻塞消耗的循环的计数十分巨大，说明内存分配时等待获取锁会浪 费大量时间，从而使得内存分配的效率较低。

我们要降低加锁引起的开销，可以为每一个 CPU 核心设置一个单独的空闲链表和对应的锁，这样运行在某 CPU 上的进程在试图分配内存时，会获取自己所在的 CPU 上空闲链表的锁然后尝试分配内存，只有当无法分配内存时，内存分配器才会到其它 CPU 的空闲链表中借取空闲的页面。

首先，修改 kalloc.c 中的数据结构，使单一的空闲链表变为多个空闲链表的数组：

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem[NCPU];
```

之后要对 kalloc() 进行修改：

```
void *kalloc(void){
  struct run *r;
  push_off();
/*利用 cpuid() 读取特殊寄存器以获得当前运行的 CPU 的编号，该编号对应着一个空闲链表，在获取该编号时需要关闭中断*/
  int c = cpuid();
  pop_off();
  /*成功获取 CPU 编号后，获取对应的空闲链表的锁，然后试图分配页面*/
  acquire(&kmem[c].lock);
  r = kmem[c].freelist;
  if(r){
    kmem[c].freelist = r->next;
    release(&kmem[c].lock);
  }
  /*若该 CPU 的空闲链表中没有空闲页面，则到其它 CPU 中借用一个*/
  else {
    release(&kmem[c].lock);
    for (int i = 0; i<NCPU; i++){
      acquire(&kmem[i].lock);
```

```
      r = kmem[i].freelist;
      if(r){
        kmem[i].freelist = r->next;
        release(&kmem[i].lock);
        break;
      }
      else {
        release(&kmem[i].lock);
      }
    }
  }
  /*最后若分配成功，则返回获取的页面*/
  if(r)
    memset((char*)r, 5, PGSIZE);
  return (void*)r;
}
```

下面来修改对应的用于释放内存的 kfree()， 直接将页面加入到当前的 CPU 空闲链表中，实现如下：

```
Void kfree(void *pa){
  struct run *r;
  push_off();
  int c = cpuid();
  pop_off();

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

  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  acquire(&kmem[c].lock);
  r->next = kmem[c].freelist;
  kmem[c].freelist = r;
  release(&kmem[c].lock);
}
```

最后初始化内存分配器的 kinit()，更改为初始化每个 CPU 的空闲链表和锁：

```
Void kinit(){
  for (int i = 0; i<NCPU; i++) {
    initlock(&kmem[i].lock, "kmem");
  }
```

```
  freerange(end, (void*)PHYSTOP);
}
```

修改完成后，运行 make qemu，输入 kalloctest 并运行，结果如下：

```
hart 1 starting
hart 2 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 182564
lock: kmem: #test-and-set 0 #acquire() 172907
lock: kmem: #test-and-set 0 #acquire() 77585
lock: bcache: #test-and-set 0 #acquire() 1248
--- top 5 contended locks:
lock: proc: #test-and-set 123692 #acquire() 226465
lock: proc: #test-and-set 38989 #acquire() 226518
lock: proc: #test-and-set 28185 #acquire() 226465
lock: proc: #test-and-set 27254 #acquire() 226465
lock: proc: #test-and-set 23122 #acquire() 226536
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

可以发现加锁开销远小于改进前的值。

实验二 Buffer Cache

作业的这一半是独立于前一半的;不管是否完成了前半部分，都可以完成后半部分(并通过测试)。如果多个进程密集地使用文件系统，它们可能会争用 bcache。保护内核/ bios .c 中的磁盘块缓存。为了在 bcache 上产生争用，Bcachetest 创建了几个重复读取不同文件的进程。

在原始的 xv6 中，对于缓存的读写是由单一的锁 bcache.lock 来保护的，这就导致了如 果系统中有多个进程在进行 IO 操作，则等待获取 bcache.lock 的开销就会较大。为了减少加 锁的开销，xv6 的实验手册提示我们可以将缓存分为几个桶，为每个同单独设置一个锁，这样 69 10.2 实现 IO 缓存 如果两个进程访问的缓存块在不同的桶中，则可以同时获得两个锁从而进行操作，而无需等 待加锁。我们的目标是将 bcachetest 中统计值 tot 降到规定值以下。

首先，改造一下需要的数据结构：

```
struct {
  struct spinlock lock[NBUCKET];
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // Sorted by how recently the buffer was used.
  // head.next is most recent, head.prev is least.
  // 将单个锁改成多个锁，并将缓存块分组
  struct buf head[NBUCKET];
```

```
} bcache;
```

然后，修改一下对应的头文件：

defs.h 中，加入：

```
int             can_lock(int, int);
```

在 buf.h 中，修改结构体如下：

```
struct buf {
  int valid;   // has data been read from disk?
  int disk;    // does disk "own" buf?
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  uchar data[BSIZE];
  uint time;
};
```

之后修改对应的 bcache 初始化函数 binit()，使之与修改后的数据结构相适应：

```
void
binit(void)
{
  struct buf *b;

  for (int i=0;i<NBUCKET;i++)
  {
    initlock(&bcache.lock[i], "bcache");
  }
  // Create linked list of buffers
  // bcache.head[0].prev = &bcache.head[0];
  bcache.head[0].next = &bcache.buf[0];
  for(b = bcache.buf; b < bcache.buf+NBUF-1; b++){
    b->next = b+1;
    initsleeplock(&b->lock, "buffer");
  }
  initsleeplock(&b->lock, "buffer");
}
```

修改 bget()，将原来的集中管理改为分桶进行管理：

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b, *last;
  struct buf *take_buf = 0;
  int id = HASH(blockno);
```

```
  acquire(&(bcache.lock[id]));

  // 在本池子中寻找是否已缓存，同时寻找空闲块，并记录链表最后一个节点便于待会插入新节点使用
  b = bcache.head[id].next;
  last = &(bcache.head[id]);
  for(; b; b = b->next, last = last->next)
  {

    if(b->dev == dev && b->blockno == blockno)
    {
      b->time = ticks;
      b->refcnt++;
      release(&(bcache.lock[id]));
      acquiresleep(&b->lock);
      return b;
    }
    if(b->refcnt == 0)
    {
      take_buf = b;
    }
  }

  //如果没缓存并且在本池子有空闲块，则使用它
  if(take_buf)
  {
    write_cache(take_buf, dev, blockno);
    release(&(bcache.lock[id]));
    acquiresleep(&(take_buf->lock));
    return take_buf;
  }

  // 到其他池子寻找最久未使用的空闲块
  int lock_num = -1;

  uint64 time = __UINT64_MAX__;
  struct buf *tmp;
  struct buf *last_take = 0;
  for(int i = 0; i < NBUCKET; ++i)
  {

    if(i == id) continue;
    //获取寻找池子的锁
    acquire(&(bcache.lock[i]));
```

```
    for(b = bcache.head[i].next, tmp = &(bcache.head[i]); b; b =
b->next,tmp = tmp->next)
    {
      if(b->refcnt == 0)
      {
        //找到符合要求的块
        if(b->time < time)
        {

          time = b->time;
          last_take = tmp;
          take_buf = b;
          //如果上一个空闲块不在本轮池子中，则释放那个空闲块的锁
          if(lock_num != -1 && lock_num != i &&
holding(&(bcache.lock[lock_num])))
            release(&(bcache.lock[lock_num]));
          lock_num = i;
        }
      }
    }
    //没有用到本轮池子的块，则释放锁
    if(lock_num != i)
      release(&(bcache.lock[i]));
  }

  if (!take_buf)
    panic("bget: no buffers");

  //将选中块从其他池子中拿出
  last_take->next = take_buf->next;
  take_buf->next = 0;
  release(&(bcache.lock[lock_num]));
  //将选中块放入本池子中，并写 cache
  b = last;
  b->next = take_buf;
  write_cache(take_buf, dev, blockno);


  release(&(bcache.lock[id]));
  acquiresleep(&(take_buf->lock));

  return take_buf;
}
```

把涉及到的其余三个函数也修改一下：

```
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  int h = HASH(b->blockno);
  acquire(&bcache.lock[h]);
  b->refcnt--;
  release(&bcache.lock[h]);
}

void
bpin(struct buf *b) {
  acquire(&bcache.lock[HASH(b->blockno)]);
  b->refcnt++;
  release(&bcache.lock[HASH(b->blockno)]);
}

void
bunpin(struct buf *b) {
  acquire(&bcache.lock[HASH(b->blockno)]);
  b->refcnt--;
  release(&bcache.lock[HASH(b->blockno)]);
}
```

编译并运行 xv6 ，然后运行 bcachetest ，会得到 类似下图的结果：

```
hart 2 starting
hart 1 starting
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 33009
lock: kmem: #test-and-set 0 #acquire() 16
lock: kmem: #test-and-set 0 #acquire() 44
lock: bcache: #test-and-set 0 #acquire() 4138
lock: bcache: #test-and-set 0 #acquire() 2137
lock: bcache: #test-and-set 0 #acquire() 4292
lock: bcache: #test-and-set 0 #acquire() 2274
lock: bcache: #test-and-set 0 #acquire() 4282
lock: bcache: #test-and-set 0 #acquire() 4320
lock: bcache: #test-and-set 0 #acquire() 6685
lock: bcache: #test-and-set 0 #acquire() 8221
lock: bcache: #test-and-set 0 #acquire() 6193
```

```
lock: bcache: #test-and-set 0 #acquire() 6193
lock: bcache: #test-and-set 0 #acquire() 6201
lock: bcache: #test-and-set 0 #acquire() 6195
lock: bcache: #test-and-set 0 #acquire() 4137
--- top 5 contended locks:
lock: proc: #test-and-set 437684 #acquire() 488959
lock: proc: #test-and-set 332179 #acquire() 488957
lock: proc: #test-and-set 328238 #acquire() 488957
lock: proc: #test-and-set 320965 #acquire() 488958
lock: proc: #test-and-set 319653 #acquire() 488957
tot= 0
test0: OK
start test1
test1 OK
```

最后 bcachetest 输出 test0 OK 和 test1 OK，且加锁开销远小于改进前的值，符合我们的预期。

整个 lab 的最后，make grade 一下：

```
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
== Test running kalloctest ==
$ make qemu-gdb
(89.4s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (11.5s)
== Test running bcachetest ==
$ make qemu-gdb
(10.2s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
Timeout! usertests: FAIL (300.2s)
    ...
        OK
        test opentest: OK
        test writetest: OK
        test writebig: panic: balloc: out of blocks
        qemu-system-riscv64: terminating on signal 15 from pid 9316 (make)
    MISSING '^ALL TESTS PASSED$'
    QEMU output saved to xv6.out.usertests
== Test time ==
time: OK
Score: 51/70
make: *** [Makefile:336: grade] 错误 1
```

得到 51/70 分，不太尽如人意。

Lab9 File system

在本实验中，将学习文件系统并向 xv6 文件系统添加大文件和符号链接。

实验一 Large Files

在原始的 xv6 的实现中，其文件系统的部分与原版的 Unix v6 类似，均使用基于 inode 和 目录的文件管理方式，但其 inode 仅为两级索引，共有 12 个直接索引块和 1 个间接索引块，间 接索引块可以指向 256 个数据块，故而一个文件最多拥有 268 个数据块。我们需要将 xv6 的 文件系统进行扩充，使之可以支持更大的文件。 xv6 的实验手册中推荐的方案是：使用三级索引，共有 11 个直接索引，1 个间接索引块 和 1 个二级间接索引块，故总共支持文件大小为 65803 块。

首先我们修改 fs.h 中的一些定义，使之符合三级索引的要求：

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

由于文件的读写需要使用 bmap() 来找到需要操作的文件块，故而我们修改 fs.c 中用于找到文件块的 bmap() ，使之能够支持三级索引。一个简单的思路是通过访问文件的位置来判断该位置是位于几级索引中，这样可以复用原先的大部分代码，而只需要实现二级间接索引的部分：

```
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;
  }
  bn -= NINDIRECT;

  if(bn < NDOUBLEINDIRECT){
    // load doubly-indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT+1]) == 0){
      ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
    }
    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);
    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;
  }

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

最后，在 xv6 中运行 bigfile，结果如下：

```
xv6 kernel is booting

init: starting sh
$ bigfile
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
...................
wrote 65803 blocks
bigfile done; ok
```

实验二 Symbolic links

在本练习中，您将向 xv6 添加符号链接。符号链接(或软链接)指的是通过路径名链接的文件;当一个符号链接被打开时，内核沿着这个链接指向被引用的文件。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。尽管 xv6 不支持多设备，但是实现这个系统调用是理解路径名查找如何工作的一个很好的练习。

首先要分别修改以下文件，添入关于 symlink 的相关声明：

kernel/syscall.h

```
#define SYS_symlink 22
```

,kernel/syscall.c

```
extern uint64 sys_symlink(void);
[SYS_symlink] sys_symlink,
```

user/usys.pl

```
entry("symlink");
```

user/user.h

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

添加新的文件类型 T_SYMLINK 到 kernel/stat.h 中
添加新的文件标志位 O_NOFOLLOW 到 kernel/fcntl.h 中

下面，在 sysfile.c 中实现 sys_symlink 函数：

```
uint64
sys_symlink(void){
  char target[MAXPATH], path[MAXPATH];
  struct inode *ip;

  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0){
    return -1;
  }

  begin_op();
  if((ip = namei(path)) == 0){
    ip = create(path, T_SYMLINK, 0, 0);
    iunlock(ip);
  }

  ilock(ip);
  if(writei(ip, 0, (uint64)target, ip->size, MAXPATH) != MAXPATH){
    return -1;
  }
  iunlockput(ip);
  end_op();
  return 0;
}
```

然后修改 sys_open 函数：

```
uint64
sys_open(void)
```

```
{
  char path[MAXPATH];
  int fd, omode;
  struct file *f;
  struct inode *ip;
  int n;

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

  begin_op();

  if(omode & O_CREATE){
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  } else {
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
    if(ip->type == T_SYMLINK) {
      if((omode & O_NOFOLLOW) == 0){
        char target[MAXPATH];
        int recursive_depth = 0;
        while(1){
          if(recursive_depth >= 10){
      iunlockput(ip);
      end_op();
      return -1;
    }
          if(readi(ip, 0, (uint64)target, ip->size-MAXPATH, MAXPATH) !=
MAXPATH){
      return -1;
    }
          iunlockput(ip);
```

```
    if((ip = namei(target)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);
    if(ip->type != T_SYMLINK){
      break;
    }
    recursive_depth++;
        }
      }
    }
  }
  if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
    iunlockput(ip);
    end_op();
    return -1;
  }

  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }

  if(ip->type == T_DEVICE){
    f->type = FD_DEVICE;
    f->major = ip->major;
  } else {
    f->type = FD_INODE;
    f->off = 0;
  }
  f->ip = ip;
  f->readable = !(omode & O_WRONLY);
  f->writable = (omode & O_WRONLY) || (omode & O_RDWR);

  if((omode & O_TRUNC) && ip->type == T_FILE){
    itrunc(ip);
  }

  iunlock(ip);
  end_op();
```

```
  return fd;
}
```

将 symlinktest 加入 Makefile 中后，编译运行 xv6，运行 symlinktest，结果如下：

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

最后 make grade 一下，结果如下：

```
make[1]: 离开目录“/home/wangxinglin/桌面/xv6-labs-2021”
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (166.2s)
== Test running symlinktest ==
$ make qemu-gdb
(0.8s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (272.7s)
    (Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 100/100
```