

OS 课设 Xv6

实验报告

姓	名:	李 奥
学	号:	2152667

二〇二三年 八 月

目录

代码位置	5
Lab1 Utilities	5
1. Boot xv6(easy)	5
1.1 实验目的	5
1.2 实验步骤	5
1.3 实验结果	5
1.4 实验心得	6
2. Sleep(easy)	6
2.1 实验目的	6
2.2 实验步骤	6
2.3 实验结果	7
3. Pingpong(easy)	7
3.1 实验目的	7
3.2 实验步骤	7
3.3 实验结果	9
3.4 遇到的问题	9
3.5 实验心得	9
4. Primes(moderate)	9
4.1 实验目的	9
4.2 实验步骤	9
4.3 实验结果	12
4.4 实验心得	12
5. Find(moderate)	12
5.1 实验目的	12
5.2 实验步骤	13
5.3 实验结果	14
5.4 遇到的问题	15
6. Xargs(moderate)	15
6.1 实验目的	15
6.2 实验步骤	15
6.3 实验结果	17
6.4 遇到的问题	17
6.5 实验心得	17
7. Make grade	17
Lab2 System calls	18
1. System call tracing(moderate)	18
1.1 实验目的	18
1.2 实验步骤	18
1.3 实验结果	20
1.4 遇到的问题	21
1.5 实验心得	21
2. Sysinfo(moderate)	21

2.1 实验目的	21
2.2 实验步骤	21
2.3 实验结果	23
2.4 遇到的问题	23
2.5 实验心得	24
3. Make grade	24
Lab3 Page tables	24
1. Speed up system calls(easy)	24
1.1 实验目的	24
1.2 实验步骤	24
1.3 实验结果	26
1.4 遇到的问题	26
1.5 实验心得	26
2. Print a page table(easy)	26
2.1 实验目的	27
2.2 实验步骤	27
2.3 实验结果	28
3. Deceting which pages have been accessed(hard)	28
3.1 实验目的	28
3.2 实验步骤	28
3.3 实验结果	29
3.4 遇到的问题	29
4. Make grade	29
Lab4 Traps	30
1. RISC-V assembly(easy)	30
1.1 实验目的	30
1.2 实验步骤	30
1.3 实验结果	30
1.4 实验心得	31
2. Backtrace(moderate)	31
2.1 实验目的	31
2.2 实验步骤	31
2.3 实验结果	33
3. Alarm(hard)	33
3.1 实验目的	33
3.2 实验步骤	33
3.3 实验结果	35
3.4 遇到的问题	35
3.5 实验心得	35
4. Make grade	35
Lab5 Copy on-write	36
1. Implement copy-on write	36
1.1 实验目的	36
1.2 实验步骤	36

1.3 实验结果	41
1.4 遇到的问题	42
1.5 实验心得	42
Lab6 Multithreading	42
1. switching between threads (moderate)	43
1.1 实验目的	43
1.2 实验步骤	43
1.3 实验结果	44
1.4 遇到的问题	44
2. Using threads (moderate)	44
2.1 实验目的	44
2.2 实验步骤	45
2.3 实验结果	46
2.4 遇到的问题	46
2.5 实验心得	46
3. Barrier(moderate)	46
3.1 实验目的	46
3.2 实验步骤	46
3.3 实验结果	47
4. Make grade	47
Lab7 networking	48
1. Your job	48
1.1 实验目的	48
1.2 实验步骤	48
1.3 实验结果	50
1.4 遇到的问题	50
2. Make grade	50
Lab8 Lock	51
1. Memory allocator (moderate)	51
1.1 实验目的	51
1.2 实验步骤	51
1.3 实验结果	53
2. Buffer cache (hard)	54
2.1 实验目的	55
2.2 实验步骤	55
2.3 实验结果	59
2.4 遇到的问题	59
2.5 实验心得	59
3. Make grade	59
Lab9 File system	60
1. Large files(moderate)	60
1.1 实验目标	60
1.2 实验步骤	60
1.3 实验结果	63

1.4 实验心得	63
2. Symbolic links (moderate)	63
2.1 实验目的	63
2.2 实验步骤	64
2.3 实验结果	65
3. Make grade	65
Lab10 mmap	65
1. Mmap(hard)	65
1.1 实验目的	65
1.2 实验步骤	66
1.3 实验结果	71
2. Make grade	72

代码位置

<https://github.com/LiAo111111/20223-os-xv6.git>

Lab1 Utilities

1. Boot xv6 (easy)

1.1 实验目的

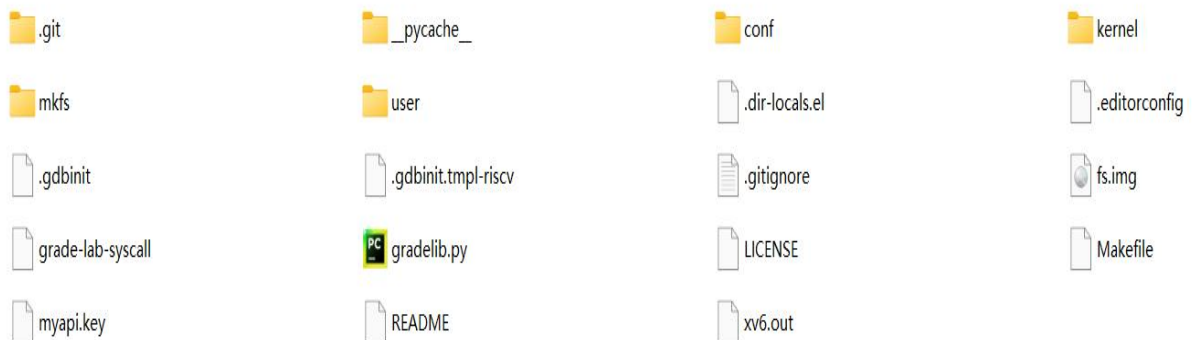
这一个实验的任务是构建实验环境。

1.2 实验步骤

```
$ git clone git://g.csail.mit.edu/xv6-labs-2021
$ cd xv6-labs-2021
新建 util 分支。
$ git checkout util
构建并运行 xv6
$ make qemu
```

1.3 实验结果

在 xv6-labs-2021 文件夹中可以看到如下的文件：



1.4 实验心得

成功搭建出实验环境是实验的前提,根据第一个实验,总结出了做 xv6 实验的一般流程。

首先打开 cmd 终端,输入 ubuntu。

切换到实验目录: `cd xv6-labs-2021`。

使用 `git checkout+分支名`, 切换到所做实验的分支。

使用 `make qemu` 等, 进一步检验实验结果,

使用 `ctrl -a +x`, 退出 `make qemu` 环境。

使用 `make grade`,检查整个实验的得分。

`git add`.添加上所有文件, `git add time.txt` 等, 添加一个文件,

`git commit -am 'aaaaa'`,进行提交。

最后: `make handin`。

2. Sleep(easy)

2.1 实验目的

为 xv6 实现 UNIX 程序 `sleep`; `sleep` 应该暂停对于用户指定的即时报价数 (tick 是由 xv6 内核定义的时间概念, 即计时器芯片两次中断的时间间隔)。

2.2 实验步骤

程序主要通过对系统调用 `sys_sleep()`的简单包装来实现, 具体步骤如下:

(1) 创建 `user/sleep.c`

(2) 编写 `sleep.c`

`Sleep.c` 中的代码如下:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
void main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(2, "Usage: %s <num> \n", argv[0]);
        exit(1);
    }
    if (sleep(atoi(argv[1])) < 0)
    {
        fprintf(2, "Sleep error\n");
    }
    exit(0);
}
```

(3) 在 `Makefile` 中的 `UPROGS` 下添加 `$U/_sleep\`

2.3 实验结果

当不传入任何参数时：

```
$ sleep
Usage: sleep <num>
```

当正确传入参数时，程序等待一段时间后继续：

```
$ sleep 10
$ |
```

3. Pingpong(easy)

3.1 实验目的

通过使用 UNIX 系统调用来实现在两个进程之间通过一组管道来“ping-pong”一个字节。（每个方向有一个管道）。

Parent 向 child 发送一个字节，child 接收到后打印”<pid>: received ping”，再将一个字节为 parent 写到管道上，然后退出。Parent 从 child 读取字节，打印”<pid>:receivedpong”，最后退出。

3.2 实验步骤

在该程序中，需要创建两条管道，一个是父进程到子进程的，一个是子进程到父进程的。并实现父子进程之间的通信。具体步骤如下：

- (1) 创建 user/pingpong.c
- (2) 在 Makefile 中的 UPROGS 下添加\$U/_pingpong\
- (3) 编写 pingpong.c

首先创建管道

```
int fds_p2c[2]; //父进程到子进程
int fds_c2p[2]; //子进程到父进程
//创建两个管道
pipe(fds_c2p);
pipe(fds_p2c);
```

其中，用下标为 0 表示读端，1 表示写端

然后 fork 出子进程

```
int pid;
```

```
pid = fork(); //fork 出子进程
```

父子进程通过关闭相对应的 fds 进行读写

具体的代码如下：


```

int main(int argc, char *argv[])
{
    int fds_p2c[2]; //父进程到子进程
    int fds_c2p[2]; //子进程到父进程
    //创建两个管道
    pipe(fds_c2p);
    pipe(fds_p2c);
    int pid;
    pid = fork(); //fork 出子进程
    //子进程
    if (pid == 0) //fork()返回值为 0，返回到新创建的子进程
    {
        //通信方向：子进程到父进程，子进程不需要读
        close(fds_c2p[INDEX_READ]);
        //通信方向：父进程到子进程，子进程不需要写
        close(fds_p2c[INDEX_WRITE]);
        //step2: 子进程收到 ping，打印
        char buf[BUFFER_SIZE];
        if (read(fds_p2c[INDEX_READ], buf, 1) == 1)
        {
            printf("%d: received ping\n", getpid());
        }
        //step3: 子进程继续向父进程发出信号
        write(fds_c2p[INDEX_WRITE], "f", 1);
        exit(0);
    }
    //父进程
    else
    {
        //通信方向：子进程到父进程，父进程不需要写
        close(fds_c2p[INDEX_WRITE]);
        //通信方向：父进程到子进程，父进程不需要读
        close(fds_p2c[INDEX_READ]);
        //step1: 父进程给子进程发 ping
        write(fds_p2c[INDEX_WRITE], "f", 1);
        //step4: 父进程读取子进程发出的 ping
        char buf[BUFFER_SIZE];
        if (read(fds_c2p[INDEX_READ], buf, 1) == 1)
        {
            printf("%d: received pong\n", getpid());
        }
    }
    //父进程结束
    exit(0);
}

```

```
}
```

3.3 实验结果

```
$ pingpong  
6: received ping  
5: received pong
```

3.4 遇到的问题

知道通道的理论知识，但是如何运用通道并不简单，需要考虑得更加细节，子进程到父进程，子进程不需要读；父进程到子进程，子进程不需要写；子进程到父进程，父进程不需要写；父进程到子进程，父进程不需要读。

3.5 实验心得

理论联系实际，将学过的知识以代码的形式书写出来，才能更好地理解知识。

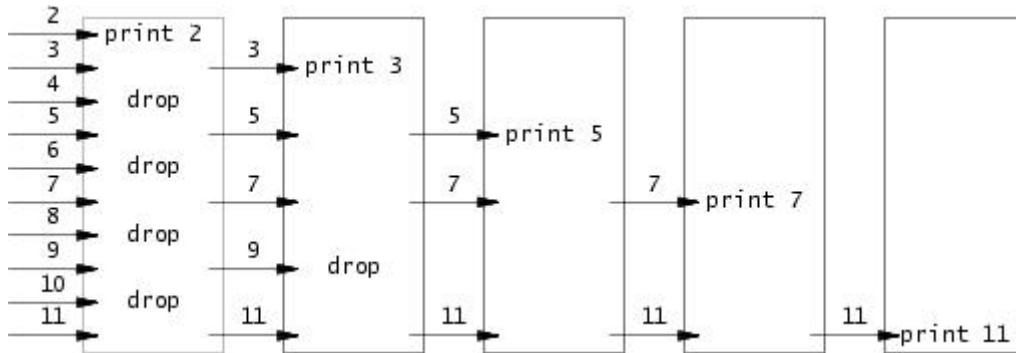
4. Primes(moderate)

4.1 实验目的

使用管道编写素数筛的并发版本。使用 `pipe` 和 `fork` 来创建管道，第一个进程将数字 2 到 35 送入管道中。最后输出区间范围内的所有素数。

4.2 实验步骤

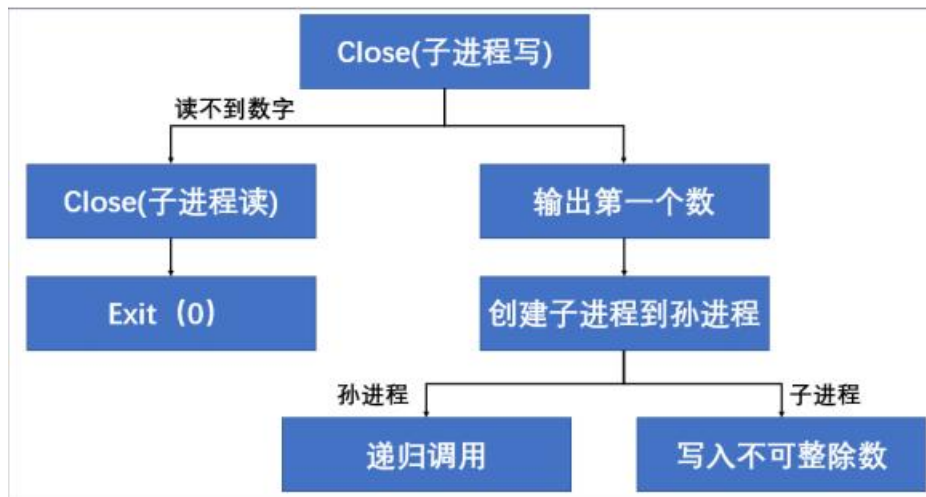
如下图所示，首先将所有的数字输入到最左边的管道，然后打印出输入管道的第一个数 2，并将管道中所有 2 的倍数剔除。接着将剔除后的所有数字输入到右边的管道，打印第一个数字 3 并剔除 3 的倍数，以此类推最终打印出所有素数。



具体的实验步骤如下：

- (1) 创建 user/primes.c
- (2) 在 Makefile 中的 UPROGS 下添加 \$U/_primes\
- (3) 编写 primes.c

代码的流程图如图所示：



代码实现如下：

首先创建 child() 函数，方便后面的调用。

```
void child(int fds_p2c[])
```

```
{
```

```
    //通讯方向：父进程到子进程，子进程不需要写
```

```
    close(fds_p2c[INDEX_WRITE]);
```

```
    int i;
```

```
    if (read(fds_p2c[INDEX_READ], &i, sizeof(i)) == 0)
```

```
    {
```

```
        //如果已经读不到数字了，说明已经全部素数都输出完毕了。这个时候子进程可以关闭管道，直接退出。
```

```
        close(fds_p2c[INDEX_READ]);
```

```
        exit(0);
```

```
    }
```

```
    printf("prime %d\n", i);
```

```
    int num = 0;
```

```

//子进程到孙子进程的管道
int fds_c2gc[2];
pipe(fds_c2gc);
int pid;
//孙子进程,递归调用本函数
if ((pid = fork()) == 0)
{
    child(fds_c2gc);
}
//子进程
else
{
    //通讯方向：子进程到孙子进程，子进程不需要读
    close(fds_c2gc[INDEX_READ]);
    while (read(fds_p2c[INDEX_READ], &num, sizeof(num)) > 0)
    {
        //如果不整除才发出去
        if (num % i != 0)
        {
            write(fds_c2gc[INDEX_WRITE], &num, sizeof(num));
        }
    }
    close(fds_c2gc[INDEX_WRITE]);
    //一样要等待所有的子进程结束
    wait(0);
}
//子进程结束
exit(0);
}
在主函数中进行控制。
int main(int argc, char *argv[])
{
    int fds_p2c[2]; //父进程到子进程
    pipe(fds_p2c);
    int pid;
    //子进程
    if ((pid = fork()) == 0)
    {
        child(fds_p2c);
    }
    //父进程
    else
    {
        //通讯方向：父进程到子进程，子进程不需要读

```

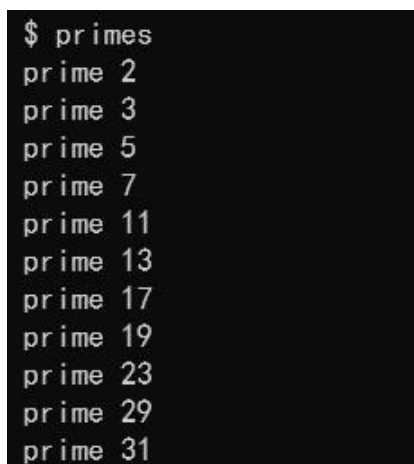
```

        close(fds_p2c[INDEX_READ]);
        for (int i = 2; i <= 35; i++)
        {
            write(fds_p2c[INDEX_WRITE], &i, sizeof(i));
        }
        //必须关闭管道，否则子进程在 read 函数阻塞
        close(fds_p2c[INDEX_WRITE]);

        //等待子进程结束
        wait(0);
    }
    //父进程结束
    exit(0);
}

```

4.3 实验结果



```

$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31

```

4.4 实验心得

有时候问题看起来比较简单，但是实现起来比较复杂，这时候通过一些形象化的图形和书写流程图可以很好地获得思路。

5. Find(moderate)

5.1 实验目的

编写一个简单版本的 UNIX 寻找程序。找到在目录树中具有特定名称的所有文件。

5.2 实验步骤

- (1) 创建 user/find.c
- (2) 在 Makefile 中的 UPROGS 下添加\$U/_find\
- (3) 编写 find.c

首先将路径格式化为文件名，去除路径中的左斜杠/，这是通过 `fmtname()` 函数实现的。

定义函数 `equal_print()` 来在确定查找到目标文件时进行输出。

在 `find()` 函数中，首先声明了文件信息的结构体 `st`，其次试探是否能够进入给定的路径，然后系统调用获得一个已存在文件的模式并判断其类型。如果该路径不是目录类型则报错。接着拷贝绝对路径，循环获取路径下的文件名并与要查找的文件名进行比较，若相同则输出路径，如果是目录类型则递归调用 `find()` 继续查找。

代码如下：

```
char *fmtname(char *path)
{
    char *p;
    for (p = path + strlen(path); p >= path && *p != '/'; p--);
    p++;
    return p;
}

void equal_print(char *path, char *findname)
{
    if (strcmp(fmtname(path), findname) == 0)
        printf("%s\n", path);
}

void find(char *dir_name, char *file_name)
{
    int fd;
    if ((fd = open(dir_name, 0)) < 0)
    {
        fprintf(2, "ls: cannot open %s\n", dir_name);
        return;
    }
    struct stat st;
    if (fstat(fd, &st) < 0)
    {
        fprintf(2, "ls: cannot stat %s\n", dir_name);
        close(fd);
        return;
    }
    //循环判断
    struct dirent de;
```

```

//buf 是用来记录文件前缀的，这样才会打印出之前的目录
char buf[512], *p;
switch (st.type)
{
case T_FILE:
    equal_print(dir_name, file_name);
    break;
case T_DIR:
    if (strlen(dir_name) + 1 + DIRSIZ + 1 > sizeof buf)
    {
        printf("find: path too long\n");
        break;
    }
    //将 path 复制到 buf 里
    strcpy(buf, dir_name);
    //p 为一个指针，指向 buf(path)的末尾
    p = buf + strlen(buf);
    //在末尾添加/ 比如 path 为 a/b/c 经过这步后变为 a/b/c/<-p
    *p++ = '/';
    // 如果是文件夹，则循环读这个文件夹里面的文件
    while (read(fd, &de, sizeof(de)) == sizeof(de))
    {
        if (de.inum == 0 || (strcmp(de.name, ".") == 0) || (strcmp(de.name, "..") ==
0))
            continue;
        //拼接出形如 a/b/c/de.name 的新路径(buf)
        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0;
        //递归查找
        find(buf, file_name);
    }
    break;
}
close(fd);
}

```

5.3 实验结果

程序正常执行时：

```
init: starting sh
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b
./b
./a/b
$ |
```

5.4 遇到的问题

当进行实验检验时，出现了以下问题：

```
$ echo >b
$ mkdir a
mkdir: a failed to create
```

经过各种尝试，得到了解决这个问题的方法：先 `make clean`，然后再 `make qemu`，成功解决了这个问题。

6. Xargs (moderate)

6.1 实验目的

编写一个简单版本的 UNIX `xargs` 程序。从标准输入中读取并为每一行运行一个命令，将该行作为命令的参数提供。

6.2 实验步骤

- (1) 创建 `user/xargs.c`
- (2) 在 `Makefile` 中的 `UPROGS` 下添加 `$U/_xargs`
- (3) 编写 `xargs.c`

代码的实现思路如下：

调用 `fork()` 创建子进程并调用 `exec()` 执行命令。从标准输入中读取行并为每行运行一个命令，且将该行作为命令的参数。根据空格符和换行符分割参数，调用子进程执行命令。

定义字符数组作为子进程的参数列表，用于存放子进程要执行的参数。然后建立一个索引便于后续追加参数，并循环拷贝一份命令行参数。创建缓冲区，用于存放从管道读出的数据。

循环读取管道中的数据，放入缓冲区，建立一个新的临时缓冲区存放追加的参数。把临时缓冲区追加到子进程参数列表后面。并循环获取缓冲区字符，当该字符不是换行时，直接

给临时缓冲区；否则创建子进程，把执行的命令和参数列表传入 `exec()` 函数中，执行命令。

代码如下：

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

int
main(int argc, char *argv[])
{
    char buf[512];
    char *full_argv[MAXARG];
    int i;
    int len;
    if(argc < 2){
        fprintf(2, "Usage: xargs command\n");
        exit(1);
    }
    if(argc + 1 > MAXARG){
        fprintf(2, "Usage: too many args\n");
        exit(1);
    }

    for(i=1 ; i < argc ; i++){
        full_argv[i-1] = argv[i];
    }
    full_argv[argc] = 0;
    while (1) {
        i = 0;
        while (1) {
            len = read(0, &buf[i], 1);
            if(len <= 0 || buf[i] == '\n') break;
            i++;
        }
        if(i == 0) break;
        buf[i] = 0;
        full_argv[argc-1]=buf;
        if(fork() == 0){
            exec(full_argv[0], full_argv);
            exit(0);
        } else {
            wait(0);
        }
    }
}
```

```
    exit(0);  
}
```

6.3 实验结果

```
$ echo hello too | xargs echo bye  
bye hello too
```

```
hart 2 starting  
hart 1 starting  
init: starting sh  
$ sh < xargstest.sh  
$ $ $ $ $ $ hello  
hello  
hello  
$ $ |
```

6.4 遇到的问题

在运行 `$ sh < xargstest.sh` 时，同样需要先 `make clean`，否则会出现错误，无法得到想要的结果。

6.5 实验心得

每一个实验的有其特殊性，不能完全照搬之前实验的流程，这个实验要想要得出正确的结果，需要先进行 `make clean`。

7. Make grade

```

== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (2.8s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.4s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (0.9s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.1s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.0s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.0s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.1s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.0s)
== Test time ==
time: OK
Score: 100/100

```

Lab2 System calls

1. System call tracing(moderate)

1.1 实验目的

在本次实验中添加一个名为 trace 的系统调用。

trace 接受一个整型参数 mask，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 fork 的系统调用号。

如果向 mask 传递了一个系统调用的编号，则必须修改 xv6 的内核，使得每个系统调用即将返回时打印出一行内容。该行内容应包含进程 id、系统调用的名称和返回值，不需要打印系统调用参数。

trace 系统调用应该对调用它的进程以及由它派生的任何子进程开启跟踪，但不应影响其他进程。

1.2 实验步骤

(1) 首先将\$U/_trace 添加到 Makefile 中的 UPROGS 里

(2) 将 trace 的声明添加到 user.h 中去

```
int trace(int);
```

(3) 在 user/usys.pl 中添加一行生成实际桩代码(user/usys.S)的脚本: entry("trace");

(4) 在 kernel/syscall.h 中添加 trace 系统调用序号的宏定义 SYS_trace

(5) 在 kernel/proc.h 中修改 proc 结构体(记录当前进程信息)。给 proc 结构体添加一个成员变量 mask, 表示当前进程需要跟踪的系统调用。

(6) 在 kernel/syscall.c 中, 新增的 trace 系统调用添加到函数指针数组*syscalls[]上, 并给内核态的系统调用 trace 加上声明。

```
extern uint64 sys_trace(void);
```

```
[SYS_trace] sys_trace,
```

(7) 在 kernel/sysproc.c 中给出 sys_trace 函数的具体实现

实现的代码如下:

```
uint64
```

```
sys_trace(void)
```

```
{
```

```
int mask;
```

```
// 取 a0 寄存器中的值返回给 mask
```

```
if(argint(0, &mask) < 0)
```

```
return -1;
```

```
// 把 mask 传给现有进程的 mask
```

```
myproc()->mask = mask;
```

```
return 0;
```

```
}
```

(8) 在 kernel/proc.c 中 fork 函数调用时, 添加子进程复制父进程的 mask 的代码

(9) 修改 syscall 函数打印跟踪每个系统调用的输出结果

(10) 在 kernel/syscall.c 中添加 syscall_name 数组。

数组代码如下:

```
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",};
```

1.3 实验结果

```
hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 968
4: syscall read -> 235
4: syscall read -> 0
4: syscall close -> 0
$ |
```

```
$ grep hello README
$ trace 2 usertests forkforkfork
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
9: syscall fork -> 11
10: syscall fork -> 12
9: syscall fork -> 13
10: syscall fork -> 14
11: syscall fork -> 15
9: syscall fork -> 16
12: syscall fork -> 17
11: syscall fork -> 18
10: syscall fork -> 19
12: syscall fork -> 20
11: syscall fork -> 21
10: syscall fork -> 22
12: syscall fork -> 23
11: syscall fork -> 24
```

```
9: syscall fork -> 68
12: syscall fork -> -1
10: syscall fork -> -1
11: syscall fork -> -1
9: syscall fork -> -1
OK
6: syscall fork -> 69
ALL TESTS PASSED
```

1.4 遇到的问题

在进行 `trace 32 grep hello README` 时，`syscall` 的显示错误，无法正确地显示出 `read/trace/exec/open` 等，经过多次尝试以及修改，最终通过添加 `static char *syscall_names[]` 成功解决了这个问题。

1.5 实验心得

当输出和自己的设想不一致时，可以想想程序运行的过程，一步步地从代码中找到 `bug`，该增加的函数就添加上。

2. Sysinfo(moderate)

2.1 实验目的

在本次实验中，添加一个系统调用 `sysinfo`，他收集有关正在运行的系统信息。系统调用接受一个参数：指向 `struct sysinfo` 的指针。

内核应填写此结构的字段：

- (1) `freemem` 字段应设置为空闲内存的字节数
- (2) `nproc` 字段应设置为状态不是 `UNUSED` 的进程数

2.2 实验步骤

- (1) 修改 `Makefile $U/_sysinfotest\`
- (2) 在 `user/user.h` 中添加 `sysinfo` 结构体以及 `sysinfo` 函数的声明

```
struct sysinfo;
int sysinfo(struct sysinfo*);
```
- (3) 在 `user/usys.pl` 中添加 `sysinfo` 的用户态接口 `entry("sysinfo");`
- (4) 将 `sysinfo` 系统调用序号定义在 `kernel/syscall.h` 中

```
#define SYS_sysinfo 23
```
- (5) 在 `kernel/syscall.c` 中，在函数指针数组中新增 `sys_info` 的函数指针、在函数名称数组中新增 `sys_sysinfo` 的函数调用名称

```
Extern uint64 sys_sysinfo(void);
[SYS_sysinfo] sys_sysinfo,
```

- (7) 在 kernel/kalloc.c 中添加 free_mem 函数统计空余内存量
函数代码如下：

```
uint64
free_mem(void)
{
    struct run *r;
    // counting the number of free page
    uint64 num = 0;
    // add lock
    acquire(&kmem.lock);
    // r points to freelist
    r = kmem.freelist;
    // while r not null
    while (r)
    {
        // the num add one
        num++;
        // r points to the next
        r = r->next;
    }
    // release lock
    release(&kmem.lock);
    // page multiplicated 4096-byte page
    return num * PGSIZE;
}
```

- (8) 在 kernel/proc.c 中新增函数 nproc 以获取可用进程数目
函数代码如下：

```
uint64
nproc(void)
{
    struct proc *p;
    // counting the number of processes
    uint64 num = 0;
    // traverse all processes
    for (p = proc; p < &proc[NPROC]; p++)
    {
        // add lock
        acquire(&p->lock);
        // if the processes's state is not UNUSED
        if (p->state != UNUSED)
        {
            // the num add one

```

```

        num++;
    }
    // release lock
    release(&p->lock);
}
return num;
}

```

(9) 在 kernel/defs.h 中添加上述两个新增函数的声明

(10) 在 kernel/sysproc.c 文件中添加 sys_sysinfo 函数的具体实现
函数代码如下：

```

uint64
sys_sysinfo(void)
{
    // addr is a user virtual address, pointing to a struct sysinfo
    uint64 addr;
    struct sysinfo info;
    struct proc *p = myproc();

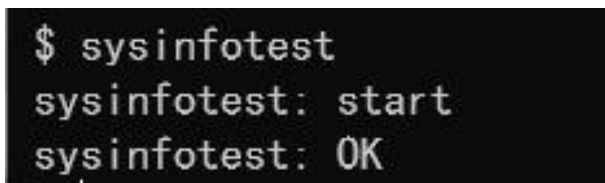
    if (argaddr(0, &addr) < 0)
        return -1;
    // get the number of bytes of free memory
    info.freemem = free_mem();
    // get the number of processes whose state is not UNUSED
    info.nproc = nproc();

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

    return 0;
}

```

2.3 实验结果



```

$ sysinfotest
sysinfotest: start
sysinfotest: OK

```

2.4 遇到的问题

在进行 make qemu 时，出现了错误，经过反复地比对和排除，发现是忘记在 Makefile 文件

中忘记加上\$U/_sysinfotest\。

2.5 实验心得

实验的流程一定要严谨，每一步要清楚地记录下来，不能忘记最基本的简单步骤。

3. Make grade

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (2.3s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.4s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (10.6s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (1.9s)
== Test time ==
time: OK
Score: 35/35
```

Lab3 Page tables

1. Speed up system calls(easy)

1.1 实验目的

某些操作系统（例如 Linux）通过共享用户空间和内核之间的只读区域中的数据来加速某些系统调用。这消除了执行这些系统调用时需要内核交叉。将映射插入页表中，针对 xv6 中的 getpid（）系统调用进行了优化。

1.2 实验步骤

（1）在 kernel/proc.c 的 allocproc 函数中为进程分配一个物理内存页面，专门用来存放共享信息。

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

```
struct usyscall *usyscall; // <加入一个指向加速页面的指针>
```

(2) 在 kernel/proc.c 中，在 allocproc 函数中我们为这枚指针分配了一个物理页，并将 pid 信息放入了这个页面。

对函数 static struct proc* allocproc(void)进行修改。

修改的代码如下：

```
if((p->usyscall) == (struct usyscall *)kalloc() == 0)
{
    freeproc(p);
    release(&p->lock);
    return 0;
}
p->usyscall->pid = p->pid;
p->pagetable = proc_pagetable(p);
if(p->pagetable == 0)
{
    freeproc(p);
    release(&p->lock);
    return 0;
}
memset(&p->context, 0, sizeof(p->context));
p->context.ra = (uint64)forkret;
p->context.sp = p->kstack + PGSIZE;
```

(3) 在 kernel/proc.c 中，在 freeproc 函数中编写释放内存的代码，这是用于可能出现的错误处理。

修改后的函数如下：

```
static void
freeproc(struct proc *p)
{
    if(p->trapframe)
        kfree((void*)p->trapframe);
    p->trapframe = 0;
    //add by me
    if(p->usyscall)
        kfree((void*)p->usyscall);
    p->usyscall=0;

    if(p->pagetable)
        proc_freepagetable(p->pagetable, p->sz);

    p->pagetable = 0;
    p->sz = 0;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->chan = 0;
```

```

    p->killed = 0;
    p->xstate = 0;
    p->state = UNUSED;
}

```

(4) 在 `proc_freepagetable` 函数中解除一下页面的映射关系。
修改后的 `proc_freepagetable` 函数如下：

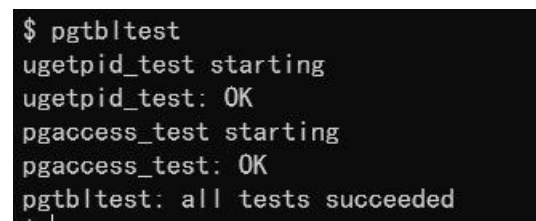
```

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

```

(5) 在 `kernel/proc.c` 的 `proc_pagetable` 函数中完成这个页面的映射，并设置页面访问权限为用户态只读(PTE_U | PTE_R)。

1.3 实验结果



```

$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded

```

1.4 遇到的问题

实验做起来比较复杂，尤其是错误处理部分。`proc_pagetable` 函数中映射页面失败时，会返回一个空指针给 `allocproc`，这里如果返回空指针给 `allocproc` 的话，后面会直接进入 `freeproc`，且此时因为 `p->pagetable` 指针为空，所以 `proc_freepagetable` 函数不会执行，所以原本属于它的释放逻辑全部得写到 `proc_pagetable` 中作为补偿，所以其实 `proc_pagetable` 中的错误处理逻辑和 `proc_freepagetable` 的逻辑从形式上是十分相似的。

1.5 实验心得

在实验的网站上，有很多实验指导，跟着指导的顺序完成实验，一般就可以解决所有的问题。

2. Print a page table(easy)

2.1 实验目的

编写一个函数 打印页表的内容。

2.2 实验步骤

(1) 在 kernel/vm.c 中定义了 raw_vmprint 函数，这个函数的作用在于按照页表的层次打印指定数量的缩进符。

函数的代码如下：

```
void
raw_vmprint(pagetable_t pagetable, int Layer)
{
    for(int i = 0 ; i < 512 ; ++i){
        pte_t pte = pagetable[i];
        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0)
        {
            uint64 phaddr = (pte >> 10) << 12;
            for( ; Layer != 0 ; --Layer)
                printf(".. ");
            printf("..%d: pte %p pa %p\n", i, pte, phaddr);
            uint64 child = PTE2PA(pte);
            raw_vmprint((pagetable_t)child, Layer + 1);
        }
        else if (pte & PTE_V){
            uint64 phaddr = (pte >> 10) << 12;
            printf(".. ...%d: pte %p pa %p\n", i, pte, phaddr);
        }
    }
}
```

(2) 在此函数的基础上封装一层，构成最终的 vmprint 函数。

代码如下：

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

(3) 在 def.h 函数中加上 vmprint 的函数签名

```
Void vmprint(pagetable_t);
```

2.3 实验结果

```
xv6 kernel is booting

hart 2 starting
hart 1 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 0x0000000021fdd813 pa 0x0000000087f76000
.. .. .510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. .511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

3. Decoding which pages have been accessed (hard)

3.1 实验目的

在 xv6 中实现一个系统调用来检测某个页面是否被访问。

3.2 实验步骤

(1) 按照实验指导书中的指示，首先在头文件 `kernel/riscv.h` 中将 `PTE_A` 加入头文件，查询一下 RISC-V 的指令手册，发现 `PTE_A` 这个标志在第 6 位。

```
#define PTE_A (1L << 6)
```

(2) 规定下一次最多可以查询的页面数量 `MAXSCAN`，它同样也定义在 `kernel/riscv.h` 文件中。

```
#define MAXSCAN 32
```

(3) 实现一下 `sys_pgaccess` 函数。

函数代码如下：

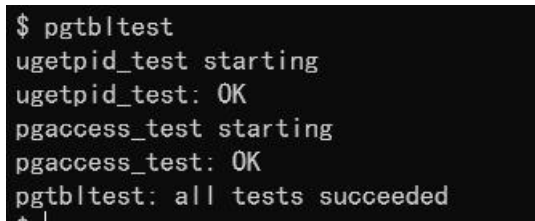
```
#ifndef LAB_PGTBL
extern pte_t * walk(pagetable_t, uint64, int);
int sys_pgaccess(void)
{
    uint64 BitMask = 0;
    int NumberOfPages;
    uint64 BitMaskVA;
    if(argint(1, &NumberOfPages) < 0)
```

```

        return -1;
    if(argaddr(0, &StartVA) < 0)
        return -1;
    if(argaddr(2, &BitMaskVA) < 0)
        return -1;
    int i;
    pte_t* pte;
    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
        }
    }
    copyout(myproc()->pagetable, BitMaskVA, (char*)&BitMask, sizeof(BitMask));
    return 0;
} #endif

```

3.3 实验结果



```

$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded

```

3.4 遇到的问题

PTE_A 标志所在的位置需要查询 RISC-V 的指令手册得知为 6，一次最多可以查询的页面数量 MAXSCAN，这个值需要通过阅读 user/pgtbltest.c 中的 pgaccess_test 函数确认。

4. Make grade

```

== Test pgtbltest ==
$ make qemu-gdb
(2.8s)
== Test   pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.4s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(112.5s)
== Test   usertests: all tests ==
usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46

```

Lab4 Traps

1. RISC-V assembly(easy)

1.1 实验目的

了解部分重要 RISC-V 程序集。

1.2 实验步骤

(1) xv6 存储库中有一个文件 `user/call.c`。编译它 并在 `User/call.asm` 中生成程序的可读汇编版本。

(2) 阅读 `call.asm` 中函数 `g`、`f` 和 `main` 的代码。

(3) 回答问题，将答案存储在一个 `answers-traps.txt` 文件中。

1.3 实验结果

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

`a0-a7` 包含函数参数；`a2` 存储 13。如果参数超过 8 个，则保存在内存中，函数调用的默认第一个参数为函数和本身名字。

(2) 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.)

在 main 的汇编代码中没有显示对函数进行调用。对 f 函数有内联调用。

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

从 call.asm 第 50 行看出 printf 的地址在 0x630。

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

Ra 寄存器用来保存函数执行以后的下一个执行指令的地址，为 0x38。

(5) 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?

如果 risc-v 是大端序，i 需要设置为 0x726c64，57616 不需要变。

(6) 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);
```

当前 a2 寄存器的值，所以不确定。

1.4 实验心得

对于单独添加的文件，需要使用 git add answers-traps.txt，然后 commit 才可以提交所有文件。

2. Backtrace(moderate)

2.1 实验目的

对于调试来说，反向跟踪通常很有用:在错误发生点以上的堆栈上的函数调用列表。本次实验的目的是实现 backtrace()函数并进行调用。

2.2 实验步骤

(1) 在 kernel/defs.h 添加定义

(2) 修改 kernel/riscv.h 中增加 r_fp()的实现，用来读取寄存器 s0

(3) 在 kernel/printf.c 中增加 backtrace()的实现

代码如下：

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



```

// 读取当前帧指针
uint64 fp = r_fp();
while (fp != PGROUNDUP(fp)) {
    // xv6 中，用一个页来存储栈，如果 fp 已经达到了栈页的上届，说明已经达到栈底
    // 地址扩张是向低地址扩展，所以当 fp 到达最高地址时说明到达栈底
    uint64 ra = *(uint64*)(fp - 8); // return address
    printf("%p\n", ra);
    fp = *(uint64*)(fp - 16); // previous fp
}
}

```

(4) 在 kernel/sysproc.c 的 sys_sleep() 函数中调用 backtrace()

修改后的 sys_sleep() 函数如下：

```

uint64
sys_sleep(void)
{
    int n;
    uint ticks0;
    if(argint(0, &n) < 0)
        return -1;
    acquire(&tickslock);
    ticks0 = ticks;
    while(ticks - ticks0 < n){
        if(myproc()->killed){
            release(&tickslock);
            return -1;
        }
        sleep(&ticks, &tickslock);
    }
    release(&tickslock);
    backtrace();
    return 0;
}

```

2.3 实验结果

```
hart 2 starting
hart 1 starting
init: starting sh
$ bttest
backtrace:
0x000000008000218a
0x0000000080001fee
0x0000000080001c9e
$ QEMU: Terminated
liao@LIAO:~/xv6-labs-2021$ addr2line -e kernel/kernel
0x000000008000218a
/home/liao/xv6-labs-2021/kernel/sysproc.c:75
0x0000000080001fee
/home/liao/xv6-labs-2021/kernel/syscall.c:144
0x0000000080001c9e
/home/liao/xv6-labs-2021/kernel/trap.c:76
^C
```

3. Alarm(hard)

3.1 实验目的

在本练习中，您将向 xv6 添加一个特性，该特性在进程使用 CPU 时间时周期性地向进程发出警报。这对于希望限制它们消耗多少 CPU 时间的计算绑定进程，或者希望进行计算但也希望采取一些周期性操作的进程来说可能很有用。更一般地，您将实现用户级中断/故障处理程序的基本形式；例如，您可以使用类似的方法来处理应用程序中的页面错误。如果您的解决方案通过了警报测试和用户测试，那么它就是正确的。

3.2 实验步骤

首先是 test0: invoke handler

(1) 在 user/user.h 中添加两个系统调用的函数原型。

(2) 在 user/usys.pl 脚本中添加两个系统调用的相应 entry，在 kernel/syscall.h 和 kernel/syscall.c 添加相应声明。

(3) 当前编写 sys_sigreturn() 只需要返回 0。该函数置于了 kernel/sysproc.c 文件中。代码如下：

```
uint64 sys_sigreturn(void) {
    struct proc* p = myproc();
    // trapframecopy must have the copy of trapframe
    if(p->trapframecopy != p->trapframe + 512) {
```

```

    return -1;
}
memmove(p->trapframe, p->trapframecopy, sizeof(struct trapframe));
p->passedticks = 0; // prevent re-entrant
p->trapframecopy = 0; // 置零
return 0;
}

```

(4) 在 kernel/proc.h 中的 struct proc 结构体中添加记录时间间隔, 调用函数地址, 以及经过时钟数的字段。

(5) 编写 sys_sigalarm() 函数, 将 interval 和 handler 的值存到当前进程的 struct proc 结构体的相应字段中。

函数代码如下:

```

uint64 sys_sigalarm(void) {
    int interval;
    uint64 handler;
    struct proc *p;
    if (argint(0, &interval) < 0 || argaddr(1, &handler) < 0 || interval < 0) {
        return -1;
    }
    p = myproc();
    p->interval = interval;
    p->handler = handler;
    p->passedticks = 0;
    return 0;
}

```

(6) kernel/proc.c 的 allocproc() 函数负责分配并初始化进程, 此处对上述 struct proc 新增的三个字段进行初始化赋值。

然后是 test1/test2 : resume interruptes code

(7) 修改 struct proc 结构体, 添加 trapframe 的副本字段。

(8) 在 kernel/trap.c 的 usertrap() 中覆盖 p->trapframe->epc 前做 trapframe 的副本。

(9) 在 sys_sigreturn() 中将副本恢复到原 trapframe。

(10) 为了保证 trapframecopy 的一致性, 在初始进程 kernel/proc.c 的 allocproc() 中, 初始化 p->trapframecopy 为 0, 表明初始时无副本。

(11) 将 alarmtest.c 添加到 makefile 文件中。

3.3 实验结果

```
hart 2 starting
hart 1 starting
init: starting sh
$ alarmtest
test0 start
..... alarm!
test0 passed
test1 start
... alarm!
..... alarm!
... alarm!
.. alarm!
... alarm!
... alarm!
..... alarm!
... alarm!
... alarm!
... alarm!
test1 passed
test2 start
..... alarm!
test2 passed
```

3.4 遇到的问题

实验步骤中的第 11 步是经过反复检查发现的，由于一开始忘记在 Makefile 文件中添加相应的文件，make qemu 时一直存在错误。

3.5 实验心得

在 makefile 文件中添加的文件不一定是程序代码文件，有时候可能是用来测试的文件。

4. Make grade

```

== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (3.2s)
== Test running alarmtest ==
$ make qemu-gdb
(3.3s)
== 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 (110.5s)
== Test time ==
time: OK
Score: 85/85

```

Lab5 Copy on-write

1. Implement copy-on write

1.1 实验目的

xv6 中的 `fork()`调用需要拷贝所有父进程中的用户空间内存给子进程，如果用户空间内存中的空间很大，那么这个拷贝的过程需要消耗非常多的时间。而且这个拷贝的动作很多时候都是浪费的，因为在 `fork()`后往往还会调用 `exec()`，之前所拷贝的用户空间内存需要被丢弃掉。

为了解决以上问题，需要实现 `copy-on-write fork`。在 `COW fork` 中将会推迟对物理内存的分配和拷贝操作，一开始只是创建一个 `pagetable` 给子进程，该 `pagetable` 指向父进程的物理页，`COW fork` 会将父进程和子进程中的 `pte` 标记为不可写的，当其中的一个进程试图对物理页进行写操作时，CPU 就会发生 `page fault`，kernel 中的 `page fault handler` 会对 `page fault` 进行处理，`handler` 会为发生 `page fault` 的 `process` 分配一个物理页，并将原来的物理页中的内容拷贝到新的物理页，然后修改发生 `page fault` 的 `process` 中的 `pte` 指向新分配的物理页，并将 `pte` 标记为可写入的。物理页的删除需要通过一个引用计数来记录有多少个 `process` 正在共享当前的物理页，当引用数量为 0 时，才真正地释放该物理页。

1.2 实验步骤

(1) 修改 `uvmcopy`，在 `xv6` 里面的 `fork` 里面调用 `uvmcopy` 来让子进程复制父进程的物理空间，把原来的页表项拿出来，改一下标志位，具体来说就是去掉 `write` 标志位，这可以引发

异常便于我们后续分配物理页，加上 COW 标记，可以在 COW 分配物理页时检查是否合法。

代码如下：

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;
    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");
        // 清空页表项中的 PTE_W
        *pte &= ~PTE_W;
        *pte |= PTE_COW;
        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte);
        // 这里没有申请新的物理页 没有必要释放 pa
        // 将原来的物理页直接映射到子进程的页表中 标志位设置不可写 COW
        if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
            goto err;
        }
        addref(pa);
    }
    return 0;
err:
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
}
```

(2) 修改 trap.c，在下次写一个 COW 页的时候会触发 15 号异常，我们在 trap.c 里的 usertrap 里边修改一个处理的代码：

```
else if(r_scause() == 15){// Store/AMO page fault
    // 取出无法翻译的地址
    // printf("cow\n");
    uint64 va=r_stval();
    if(handler_cow_pagefault(p->pagetable, va)<0){
        //杀死进程
        p->killed=1;
    }
}
```

(3) 在 vm.c 里面实现 handler_cow_pagefault 函数的定义，这样对于访问类似于 walk 函数啥的会更加方便，而且对外提供一个封装接口。

代码如下：

```

int
handler_cow_pagefault(pagetable_t pagetable, uint64 va)
{
    // 取出原来无法翻译的 va 地址
    if (va >= MAXVA)
        return -1;

    pte_t *pte=walk(pagetable, va, 0);

    if (pte == 0 || (*pte & PTE_COW)==0)
        return -1;

    uint64 pa=PTE2PA(*pte);

    // 分配一个新的物理页 将原来物理页中内容拷贝至新物理页
    pagetable_t new_page=(pte_t*)kalloc();
    if (new_page==0)
        return -1;

    memmove((char*)new_page, (char*)pa, PGSIZE);
    // 减少原来物理页引用计数
    kfree((void*)pa);
    // 将新的物理页映射至页表中 去除 COW 位 加上 write 位
    pte_t flags=PTE_FLAGS(*pte);
    flags &= ~PTE_COW;
    flags |= PTE_W;
    // printf("In cow set:   cow:%d   w:%d\n",flags&PTE_COW, flags&PTE_W);
    *pte=PA2PTE(new_page) | flags;
    return 0;
}

```

(4) 引入计数机制

首先定义数据结构。

```

#define INDEX(pa) ((pa - KERNBASE)>>12)
#define INDEXSIZE 32768//由 KERNBASE 到 PHYSTOP 共 32768 个页面
struct {
    struct spinlock lock;
    int count[INDEXSIZE];
} memref;

```

然后在初始化的时候，我们得把引用计数变成 1，变成 1 的原因是，在 init 的时候调用 freerange，里面用的 kfree，会把页面释放然后放到空闲链表里，然后页表引用计数就变成 0 了。

```

void
kinit()
{

```

```

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

```

```

void
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);
    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    {
        // printf("%d\n", INDEX((uint64)p));
        acquire(&memref.lock);
        memref.count[INDEX((uint64)p)]=1;
        release(&memref.lock);
        kfree(p);
    }
    // kfree(p);
}

```

接下来是 kalloc，在成功分配一个页的时候只要把引用计数加一

```

void addref(uint64 pa)
{
    acquire(&memref.lock);
    memref.count[INDEX(pa)]++;
    release(&memref.lock);
}

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

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
    {
        kmem.freelist = r->next;
        addref((uint64)r);
    }
    release(&kmem.lock);

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

```



```
}
```

处理 kfree 函数，检查这个页面的引用计数是否大于 1，如果是，直接调整引用计数然后返回。检查这个页面的物理地址范围。对引用计数是 1 和 0 分别处理。

```
void
```

```
kfree(void *pa)
```

```
{
```

```
    struct run *r;
```

```
    // printf("In kfree:1  %d ref:%d\n",INDEX((uint64)pa),memref.count[INDEX((uint64)pa)]);
```

```
    acquire(&memref.lock);
```

```
    if (memref.count[INDEX((uint64)pa)]>1)
```

```
    {
```

```
        --memref.count[INDEX((uint64)pa)];
```

```
        release(&memref.lock);
```

```
        return;
```

```
    }
```

```
    release(&memref.lock);
```

```
    // printf("In kfree:2\n");
```

```
    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;
```

```
    acquire(&memref.lock);
```

```
    if (memref.count[INDEX((uint64)pa)]==0)
```

```
    {
```

```
        release(&memref.lock);
```

```
        panic("kfree");
```

```
    }
```

```
    else if (memref.count[INDEX((uint64)pa)]==1)
```

```
    {
```

```
        --memref.count[INDEX((uint64)pa)];
```

```
        release(&memref.lock);
```

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

```
        r->next = kmem.freelist;
```

```
        kmem.freelist = r;
```

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

```
    }
```

```
    else
```

```
    {
```

```
        release(&memref.lock);
```

```
        panic("kfree");
```

```
    }
```

```
}
```

(5) 修改 copyout

在把内核数据复制到用户空间的时候，也就相当于在写用户的物理地址，如果这个时候对应的页表项是写时复制的，我们应该也做一次新的物理页的分配，也就是之前的 handler_cow_pagefault 函数。

```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
    uint64 n, va0, pa0;
    while(len > 0){
        va0 = PGROUNDDOWN(dstva);
        pte_t *pte;
        if (va0 >= MAXVA)
        {
            return -1;
        }
        if ((pte=walk(pagetable,va0,0))==0)
        {
            return -1;
        }
        if (*pte & PTE_COW)
        {
            if(handler_cow_pagefault(pagetable, va0)<0)
            {
                return -1;
            }
        }
        pa0 = walkaddr(pagetable, va0);
        if(pa0 == 0)
            return -1;
        n = PGSIZE - (dstva - va0);
        if(n > len)
            n = len;
        memmove((void *)(pa0 + (dstva - va0)), src, n);
        len -= n;
        src += n;
        dstva = va0 + PGSIZE;
    }
    return 0;
}
```

1.3 实验结果

```
hart 2 starting
hart 1 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
```

```
test killstatus: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

1.4 遇到的问题

这个实验比较复杂,有许多的细节在实验过程中不能一次考虑全面,在定义数据结构时,一个数组是不够的,必须给数组加上全局锁,避免数据竞争。

在释放一个页的时候,由于是 **COW**,可能有多个进程可能共享一个一样的映射,那么每个进程释放一个页的时候都把这个页放到空闲链表里是行不通的,解决这个问题最直接的方法是,给每个页一个引用计数,当这个计数从 **1** 变成 **0** 的时候,我们就把这个页放到空闲页链表里边,如果是大于 **1** 的,我们只需要减小引用计数就行,如果等于 **0** 或者小于 **0**,直接 **panic**。

1.5 实验心得

“在下次写一个 **COW** 页的时候会触发 **15** 号异常,我们在 **trap.c** 里的 **usertrap** 里边修改一个处理的代码”,通过对异常情况进行特殊处理可以更快速地解决问题,这需要对系统基础知识有着更深的了解。

Lab6 Multithreading

1. switching between threads (moderate)

1.1 实验目的

为用户级线程系统设计上下文切换机制，然后实现它。

制定一个计划，创建线程并保存/恢复寄存器以在线程之间切换，并实现该计划。

1.2 实验步骤

(1) 修改 user/uthread.c，新增头文件引用。

添加头文件：

```
#include "kernel/riscv.h"
#include "kernel/spinlock.h"
#include "kernel/param.h"
#include "kernel/proc.h"
```

(2) 修改 user/uthread.c，struct thread 增加成员 struct context，用于线程切换时保存/恢复寄存器信息。此处 struct context 即 kernel/proc.h 中定义的 struct context。

(3) 修改 user/uthread.c，修改 thread_switch 函数定义

修改为：extern void thread_switch(struct context*, struct context*);

(4) 修改 user/uthread.c，修改 thread_create 函数，修改 ra 保证初次切换到线程时从何处执行，修改 sp 保证栈指针位于栈顶（地址最高）。

(5) 修改 user/uthread.c，修改 thread_schedule，调用 thread_switch 调用，保存当前线程上下文，恢复新线程上下文。

(6) 修改 user/uthread_switch.S，实现保存/恢复寄存器。

代码如下：

thread_switch:

/* YOUR CODE HERE */

sd ra, 0(a0)

sd sp, 8(a0)

sd s0, 16(a0)

sd s1, 24(a0)

sd s2, 32(a0)

sd s3, 40(a0)

sd s4, 48(a0)

sd s5, 56(a0)

sd s6, 64(a0)

sd s7, 72(a0)

sd s8, 80(a0)

sd s9, 88(a0)

sd s10, 96(a0)

```

sd s11, 104(a0)

ld ra, 0(a1)
ld sp, 8(a1)
ld s0, 16(a1)
ld s1, 24(a1)
ld s2, 32(a1)
ld s3, 40(a1)
ld s4, 48(a1)
ld s5, 56(a1)
ld s6, 64(a1)
ld s7, 72(a1)
ld s8, 80(a1)
ld s9, 88(a1)
ld s10, 96(a1)
ld s11, 104(a1)
ret    /* return to ra */

```

1.3 实验结果

```

thread_c 98
thread_a 98
thread_b 98
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads

```

1.4 遇到的问题

修改 user/uthread.c，修改 thread_switch 函数定义这一步骤比较隐晦，容易遗漏。

2. Using threads (moderate)

2.1 实验目的

使用哈希表探索并行程序（带有线程和锁）。

2.2 实验步骤

(1) 修改 notxv6/ph.c, 定义互斥锁, 每个 bucket 一个锁。

```
pthread_mutex_t lock[NBUCKET];
```

(2) 修改 notxv6/ph.c, 初始化互斥锁。

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

```
{
    pthread_mutex_init(&lock[i],NULL);
}
```

(3) 修改 notxv6/ph.c, 在 put () 中对数组某个 bucket 加锁。

修改后的 put 函数如下:

```
static
void put(int key, int value)
{
    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;
    }
    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]);
}
```

2.3 实验结果

```
liao@LIAO:~/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
liao@LIAO:~/xv6-labs-2021$ ./ph 1
100000 puts, 5.642 seconds, 17726 puts/second
0: 0 keys missing
100000 gets, 5.618 seconds, 17799 gets/second
liao@LIAO:~/xv6-labs-2021$ ./ph 2
100000 puts, 3.517 seconds, 28437 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 5.431 seconds, 36827 gets/second
```

2.4 遇到的问题

相比于之前熟悉的 `make qemu`，这次实验中为 `make ph`。

`./ph 1` 和 `./ph 2`，“ph”和数字之间需要有空格。

在第一次实验时，定义了 `pthread_mutex_t lock`，虽然可以得到结果，但是只可以通过 `ph_safe`，无法通过 `ph_fast`。之后定义 `pthread_mutex_t lock[NBUCKET]`，做出相应的修改，通过了 `ph_fast`。

2.5 实验心得

在多线程作业时，设置互斥锁具有非常重要的意义，加锁对于系统正常运行至关重要。

3. Barrier(moderate)

3.1 实验目的

实现 `barrier`（应用中的一个点）：所有参与线程必须等待，直到所有其他参与线程也到达该点。

3.2 实验步骤

- (1) 修改 `notxv6/barrier.c`，实现 `barrier()` 方法。

代码如下：

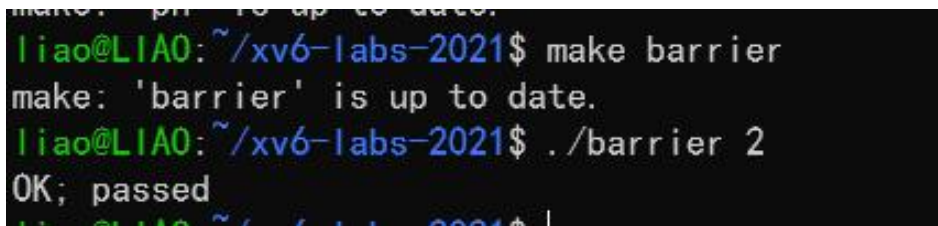
```
static void
barrier()
{
    pthread_mutex_lock(&(bstate.barrier_mutex));
```

```

    bstate.nthread++;
    if(nthread==bstate.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));
}

```

3.3 实验结果



```

liao@LIAO:~/xv6-labs-2021$ make barrier
make: 'barrier' is up to date.
liao@LIAO:~/xv6-labs-2021$ ./barrier 2
OK; passed

```

3.4 实验心得

在进行实验检验时，一共出现了三种形式：

make qemu

make ph

make barrier

4. Make grade


```

== Test uthread ==
$ make qemu-gdb
uthread: OK (3.1s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/liao/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/liao/xv6-labs-2021'
ph_safe: OK (8.8s)
== Test ph_fast == make[1]: Entering directory '/home/liao/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/liao/xv6-labs-2021'
ph_fast: OK (20.3s)
== Test barrier == make[1]: Entering directory '/home/liao/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/liao/xv6-labs-2021'
barrier: OK (3.3s)
== Test time ==
time: OK
Score: 60/60

```

Lab7 networking

1. Your job

1.1 实验目的

为网络接口卡（network interface card）写一个 xv6 设备驱动。

完成 `e1000_transmit()` 和 `e1000_recv()`，都在 `kernel/e1000.c`，以便于驱动可以传递、接收包。

1.2 实验步骤

（1）实现 `e1000_transmit`

先通过读取 `E1000_TDT` 控制寄存器，来向 `E1000` 了解应该在 `TX` 环的哪个索引处放置下一个数据包。

然后检查是否环溢出了。如果 `E1000_TXD_STAT_DD` 没有在被 `E1000_TDT` 指定的描述符中被设置，那么 `E1000` 还没有完成之前的传输请求的发送，返回一个 `error`。否则，使用 `mbuffree()` 来释放上一个存在这个描述符的 `mbuf`（如果有的话）。

然后填写这个描述符。`m->head` 指向内存中这个包的内容，`m->len` 是包的长度。设置必要的 `cmd flags`（看 `E1000` 手册的 3.3 节），并保存一个指向 `mbuf` 的指针，以便稍后释放。

最后通过给 `E1000_TDT+1` 模 `TX_RING_SIZE` 来更新环的位置。

如果 `e1000_transmit()` 成功添加了 `mbuf` 到环上，那么返回 0。如果失败（比如没有可以获得的描述符），返回 -1，所以调用者直到去释放掉这个 `mbuf`。

代码如下：

```
int
```

```

e1000_transmit(struct mbuf *m)
{
    acquire(&e1000_lock);
    uint reg_tdt = regs[E1000_TDT];
    if((tx_ring[reg_tdt].status & E1000_TXD_STAT_DD) == 0){
        return -1;
    }
    if(tx_mbufs[reg_tdt] != 0)
        mbuf_free(tx_mbufs[reg_tdt]);
    tx_mbufs[reg_tdt] = m;
    tx_ring[reg_tdt].length = m->len;
    tx_ring[reg_tdt].addr = (uint64)(m->head);
    tx_ring[reg_tdt].cmd = 9;
    regs[E1000_TDT] = (regs[E1000_TDT] + 1) % TX_RING_SIZE;
    release(&e1000_lock);
    return 0;
}

```

(2) 实现 e1000_recv

先通过 E1000_RDT 控制寄存器，并且加 1 模 RX_RING_SIZE，来了解 E1000 把下一个接收到的数据包放在哪。

然后通过检查描述符中的 status 部分的 E1000_RXD_STAT_DD 位，检查是否有一个新的包可以获得。如果没有，就停止。否则，更新 mbuf 的 m->len 位描述符中的长度。用 net_rx() 把这个 mbuf 传输到网络栈。

用一个 mbuf_alloc() 来分配一个新的 mbuf 来替代刚刚传入 net_rx() 的那个。把它的数据指针 m->head 放到描述符中。把描述符的状态位清 0。

最后，更新 E1000_RDT 寄存器为下一个需要处理的描述符。

代码如下：

```

static void
e1000_recv(void)
{
    uint reg_rdt = regs[E1000_RDT];
    int i = (reg_rdt + 1) % RX_RING_SIZE;
    while(rx_ring[i].status & E1000_RXD_STAT_DD){
        rx_mbufs[i]->len = rx_ring[i].length;
        net_rx(rx_mbufs[i]);
        if((rx_mbufs[i] = mbuf_alloc(0)) == 0)
            panic("e1000");
        rx_ring[i].addr = (uint64)rx_mbufs[i]->head;
        rx_ring[i].status = 0;
        i = (i + 1) % RX_RING_SIZE;
    }
    if(i == 0)
        i = RX_RING_SIZE;
    regs[E1000_RDT] = (i - 1) % RX_RING_SIZE;
}

```

```
}
```

1.3 实验结果

运行 tcpdump -XXnr packets.pcap

```
liao@LIAO:~/xv6-labs-2021$ tcpdump -XXnr packets.pcap
reading from file packets.pcap, link-type EN10MB (Ethernet), snapshot length 65536
18:27:00.418805 IP 10.0.2.15.2000 > 10.0.2.2.26099: UDP, length 19
    0x0000:  ffff ffff ffff 5254 0012 3456 0800 4500  ....RT..4V..E.
    0x0010:  002f 0000 0000 6411 3eae 0a00 020f 0a00  ./....d.>.....
    0x0020:  0202 07d0 65f3 001b 0000 6120 6d65 7373  ....e.....a.mess
    0x0030:  6167 6520 6672 6f6d 2078 7636 21      age.from.xv6!
18:27:00.419129 ARP, Request who-has 10.0.2.15 tell 10.0.2.2, length 46
    0x0000:  ffff ffff ffff 5255 0a00 0202 0806 0001  ....RU.....
    0x0010:  0800 0604 0001 5255 0a00 0202 0a00 0202  ....RU.....
    0x0020:  0000 0000 0000 0a00 020f 0000 0000 0000  ....
    0x0030:  0000 0000 0000 0000 0000 0000      ....
18:27:01.374575 ARP, Reply 10.0.2.15 is-at 52:54:00:12:34:56, length 28
    0x0000:  ffff ffff ffff 5254 0012 3456 0806 0001  ....RT..4V....
    0x0010:  0800 0604 0002 5254 0012 3456 0a00 020f  ....RT..4V....
    0x0020:  5255 0a00 0202 0a00 0202      RU.....
18:27:01.374575 IP 10.0.2.2.26099 > 10.0.2.15.2000: UDP, length 17
```

一个终端运行 make server, 另外一个终端运行 make qemu + nettests。

```
hart 2 starting
hart 1 starting
init: starting sh
$ nettests
nettests running on port 26099
testing ping: OK
testing single-process pings: OK
testing multi-process pings: OK
testing DNS
DNS arecord for pdos.csail.mit.edu. is 128.52.129.126
DNS OK
all tests passed.
```

1.4 遇到的问题

在进行 nettests 检验时, 需要在两个终端进行同时操作, 一个终端运行 make server, 另外一个终端运行 make qemu + nettests, 这样才能正常检验。

2. Make grade

```

== Test running nettests ==
$ make qemu-gdb
(4.0s)
== 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

```

Lab8 Lock

多核计算机上较差的并行性的一个常见症状是高锁争用。提高并行性通常需要同时改变数据结构和锁定策略，以减少争用。为 xv6 内存分配器和块缓存执行操作，以减少锁的竞争从而提高运行效率。

1. Memory allocator (moderate)

1.1 实验目的

一开始是有个双向链表用来存储空闲的内存块，如果很多个进程要竞争这一个链表，就会把效率降低很多。所以把链表拆成每个 CPU 一个，在申请内存的时候就直接在本 CPU 的链表上找就行了。没找到的话，就直接从别的链表里边获取。

1.2 实验步骤

(1) 定义多个内存链表数据结构。

```

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

```

(2) 在初始化时初始化锁。

代码如下：

```

void
kinit()
{

```

```

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

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

```

(3) 在释放内存时修改单个链表的，获取 CPUid 直到更新对应 CPU 链表要关中断，否则切了 CPU 回来可能把内存放错位置。

代码如下：

```

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=cpuuid();
    if (id<0 || id>=NCPU)
    {
        panic("kfree:wrong cpuid");
    }
    acquire(&kmem[id].lock);
    r->next = kmem[id].freelist;
    kmem[id].freelist = r;
    release(&kmem[id].lock);
    pop_off();
}

```

(4) 分配内存时先在本 CPU 链表中寻找，如果没有空闲的，从其他链表中获取。

代码如下：

```

void *
kalloc(void)
{
    struct run *r;
    push_off();
    int id=cpuuid();
    acquire(&kmem[id].lock);
    r = kmem[id].freelist;
    if(r){

```

```

        kmem[id].freelist = r->next;
    }else{
        for (int i = 0; i < NCPU; i++)
        {
            if (i==id)
                continue;
            acquire(&kmem[i].lock);
            r = kmem[i].freelist;
            if (r){ //链表中还存在下一个节点 需要更新节点
                kmem[i].freelist=r->next;
                release(&kmem[i].lock);
                break;
            }
            release(&kmem[i].lock);
        }
    }
    release(&kmem[id].lock);
    pop_off();
    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
}

```

1.3 实验结果

```

$ kallocat
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 39533
lock: kmem: #test-and-set 0 #acquire() 193177
lock: kmem: #test-and-set 0 #acquire() 200334
lock: bcache: #test-and-set 0 #acquire() 18
lock: bcache.bucket: #test-and-set 0 #acquire() 20
lock: bcache.bucket: #test-and-set 0 #acquire() 4
lock: bcache.bucket: #test-and-set 0 #acquire() 3
lock: bcache.bucket: #test-and-set 0 #acquire() 4
lock: bcache.bucket: #test-and-set 0 #acquire() 6
lock: bcache.bucket: #test-and-set 0 #acquire() 8
lock: bcache.bucket: #test-and-set 0 #acquire() 7
lock: bcache.bucket: #test-and-set 0 #acquire() 140
lock: bcache.bucket: #test-and-set 0 #acquire() 3
lock: bcache.bucket: #test-and-set 0 #acquire() 2
lock: bcache.bucket: #test-and-set 0 #acquire() 4
lock: bcache.bucket: #test-and-set 0 #acquire() 2
--- top 5 contended locks:
lock: proc: #test-and-set 82031 #acquire() 762437
lock: proc: #test-and-set 59152 #acquire() 762382
lock: proc: #test-and-set 47591 #acquire() 762441
lock: proc: #test-and-set 45613 #acquire() 762381
lock: proc: #test-and-set 45329 #acquire() 762382
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED

```

执行 usertests:

```

test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test killstatus: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED

```

2. Buffer cache (hard)

2.1 实验目的

问题是一堆用来和外设交换数据的 **buffer**，也有一堆进程想要用，那么想用的时候就会竞争，于是将这些 **buffer** 按照块号哈希成多个队列，之后要读了，或者要释放了，就利用块号到对应的哈希表中去找。

2.2 实验步骤

(1) 定义结构体

定义 **bcache** 结构体：

```
struct
{
    struct spinlock lock;
    struct buf buf[NBUF];
    struct buf head[NBUCKETS];           // 哈希桶头
    struct spinlock hash_lock[NBUCKETS]; // 哈希桶锁
} bcache;
```

(2) 初始化

代码如下：

```
void binit(void)
{
    struct buf *b;
    initlock(&bcache.lock, "bcache");
    // 哈希桶和哈希桶锁初始化
    for (int i = 0; i < NBUCKETS; i++)
    {
        initlock(&bcache.hash_lock[i], "bcache.bucket");
        bcache.head[i].prev = &bcache.head[i];
        bcache.head[i].next = &bcache.head[i];
    }
    int hash_num;
    for (int i = 0; i < NBUF; i++)
    {
        b = &bcache.buf[i];
        hash_num = HASHNUM(b->blockno);
        b->next = bcache.head[hash_num].next;
        b->prev = &bcache.head[hash_num];
        initsleeplock(&b->lock, "buffer");
        bcache.head[hash_num].next->prev = b;
        bcache.head[hash_num].next = b;
    }
}
```

(3) 更改 **bget**

在 `usertests` 中, `manywrites` 要用到 `balloc` 和 `bfree`, 现在很多进程想读一个块进来, 每个都发现不存在, 所以都会去分配一块内存块, 如果分配了多个 `buffer` 也就是存在多个副本, 那么在 `bfree` 的时候就会 `panic` (多次释放一块磁盘)。这里我们需要一种机制来保证只分配一个块。那就是, 保证检查引用计数和分配一个块的操作捆绑在第一次引用计数更新之后。这里用了 `LRU` 的链表, 如下实现方法是采用“将刚用完的插入链表头, 取的时候从链表尾部向前取”的方式实现。

代码如下:

```
static struct buf *
bget(uint dev, uint blockno)
{
    struct buf *b;

    int hash_num = HASHNUM(blockno);

    // acquire(&bcache.lock);

    // todo 获取哈希桶锁
    acquire(&bcache.hash_lock[hash_num]);

    // Is the block already cached?
    // 如果块已经映射在缓冲中
    // 增加块引用数 维护 LRU 队列 释放 bcache 锁 获取块的睡眠锁
    for (b = bcache.head[hash_num].next; b != &bcache.head[hash_num]; b = b->next)
    {
        if (b->dev == dev && b->blockno == blockno)
        {
            b->refcnt++;
            // todo 释放哈希锁
            release(&bcache.hash_lock[hash_num]);
            // release(&bcache.lock);
            acquiresleep(&b->lock);
            return b;
        }
    }
    release(&bcache.hash_lock[hash_num]);

    // Not cached.
    // Recycle the least recently used (LRU) unused buffer.
    // 遍历所有哈希桶找到一个引用为空的块

    acquire(&bcache.lock);
    acquire(&bcache.hash_lock[hash_num]);
    for (b = bcache.head[hash_num].next; b != &bcache.head[hash_num]; b = b->next)
    {
```

```

if (b->dev == dev && b->blockno == blockno)
{
    b->refcnt++;
    // todo 释放哈希锁
    release(&bcache.lock);
    release(&bcache.hash_lock[hash_num]);
    // release(&bcache.lock);
    acquiresleep(&b->lock);
    return b;
}
}
release(&bcache.hash_lock[hash_num]);

for (int i = 0; i < NBUCKETS; i++)
{
    acquire(&bcache.hash_lock[i]);
    for (b = bcache.head[i].prev; b != &bcache.head[i]; b = b->prev)
    {
        if (b->refcnt==0)
        {
            b->dev=dev;
            b->blockno=blockno;
            b->valid=0;
            b->refcnt=1;

            b->prev->next=b->next;
            b->next->prev=b->prev;

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

            release(&bcache.hash_lock[i]);
            release(&bcache.lock);
            // release(&bcache.hash_lock[hash_num]);
            // release(&bcache.lock);
            acquiresleep(&b->lock);
            return b;
        }
    }
}
release(&bcache.hash_lock[i]);
}

```

```

        panic("bget: no buffers");
    }
(4) 实现 brelse 函数
代码如下:
void brelse(struct buf *b)
{
    if (!holdingsleep(&b->lock))
        panic("brelse");
    releasesleep(&b->lock);
    int hash_num = HASHNUM(b->blockno);
    if (b->refcnt > 0)
    {
        b->refcnt--;
    }
    if (b->refcnt == 0)
    {
        // no one is waiting for it.
        b->next->prev = b->prev;
        b->prev->next = b->next;
        b->next = bcache.head[hash_num].next;
        b->prev = &bcache.head[hash_num];
        bcache.head[hash_num].next->prev = b;
        bcache.head[hash_num].next = b;
    }
}

```

(5) 修改 bpin 函数和 bunpin 函数

```

void bpin(struct buf *b)
{
    int hash_num=HASHNUM(b->blockno);
    acquire(&bcache.hash_lock[hash_num]);
    b->refcnt++;
    release(&bcache.hash_lock[hash_num]);
}
void bunpin(struct buf *b)
{
    int hash_num=HASHNUM(b->blockno);
    acquire(&bcache.hash_lock[hash_num]);
    b->refcnt--;
    release(&bcache.hash_lock[hash_num]);
}

```

(6) 将 param.h 里边的 FSSIZE 参数变为原来的十倍，主要用于通过 bigwrite 测试。

2.3 实验结果

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 1112621
lock: kmem: #test-and-set 5 #acquire() 2925536
lock: kmem: #test-and-set 0 #acquire() 2887713
lock: kmem: #test-and-set 0 #acquire() 92
lock: kmem: #test-and-set 0 #acquire() 92
lock: kmem: #test-and-set 0 #acquire() 92
lock: kmem: #test-and-set 0 #acquire() 92
lock: kmem: #test-and-set 0 #acquire() 92
lock: bcache: #test-and-set 11479 #acquire() 48706
lock: bcache.bucket: #test-and-set 0 #acquire() 181752
lock: bcache.bucket: #test-and-set 0 #acquire() 53813
lock: bcache.bucket: #test-and-set 0 #acquire() 62857
lock: bcache.bucket: #test-and-set 7 #acquire() 54955
lock: bcache.bucket: #test-and-set 0 #acquire() 48762
lock: bcache.bucket: #test-and-set 0 #acquire() 252271
lock: bcache.bucket: #test-and-set 504 #acquire() 51925
lock: bcache.bucket: #test-and-set 0 #acquire() 203131
lock: bcache.bucket: #test-and-set 139 #acquire() 504574
lock: bcache.bucket: #test-and-set 0 #acquire() 173418
lock: bcache.bucket: #test-and-set 18 #acquire() 180557
lock: bcache.bucket: #test-and-set 0 #acquire() 150283
lock: bcache.bucket: #test-and-set 0 #acquire() 142994
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 88279732 #acquire() 233974
lock: proc: #test-and-set 86521576 #acquire() 67547049
lock: proc: #test-and-set 56365971 #acquire() 67557083
lock: proc: #test-and-set 55792163 #acquire() 66230262
lock: proc: #test-and-set 50583812 #acquire() 67546941
tot= 12152
test0: OK
start test1
test1 OK
```

2.4 遇到的问题

当很多进程想读一个块进来时，每个都发现不存在，所以都会去分配一块内存块，如果分配了多个 buffer 也就是存在多个副本，那么在 bfree 的时候就会多次释放一块磁盘。最终使用 LRU 的链表，保证检查引用计数和分配一个块的操作捆绑在第一次引用计数更新之后，实现方法为“将刚用完的插入链表头，取的时候从链表尾部向前取”。

2.5 实验心得

这个实验的任务完成起来很复杂，对哈希表了解更多，接触到了哈希锁的设置。

3. Make grade

```

== Test running kalloc test ==
$ make qemu-gdb
(82.1s)
== Test kalloc test: test1 ==
kalloc test: test1: OK
== Test kalloc test: test2 ==
kalloc test: test2: OK
== Test kalloc test: sbrkmuch ==
$ make qemu-gdb
kalloc test: sbrkmuch: OK (8.6s)
== Test running bcachetest ==
$ make qemu-gdb
(10.3s)
== Test bcachetest: test0 ==
bcachetest: test0: OK
== Test bcachetest: test1 ==
bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (129.3s)
== Test time ==
time: OK
Score: 70/70

```

Lab9 File system

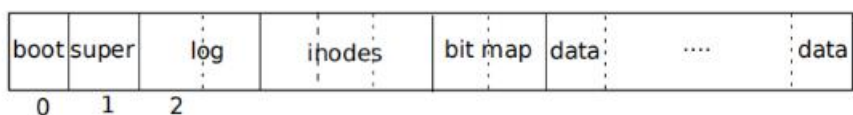
1. Large files(moderate)

1.1 实验目标

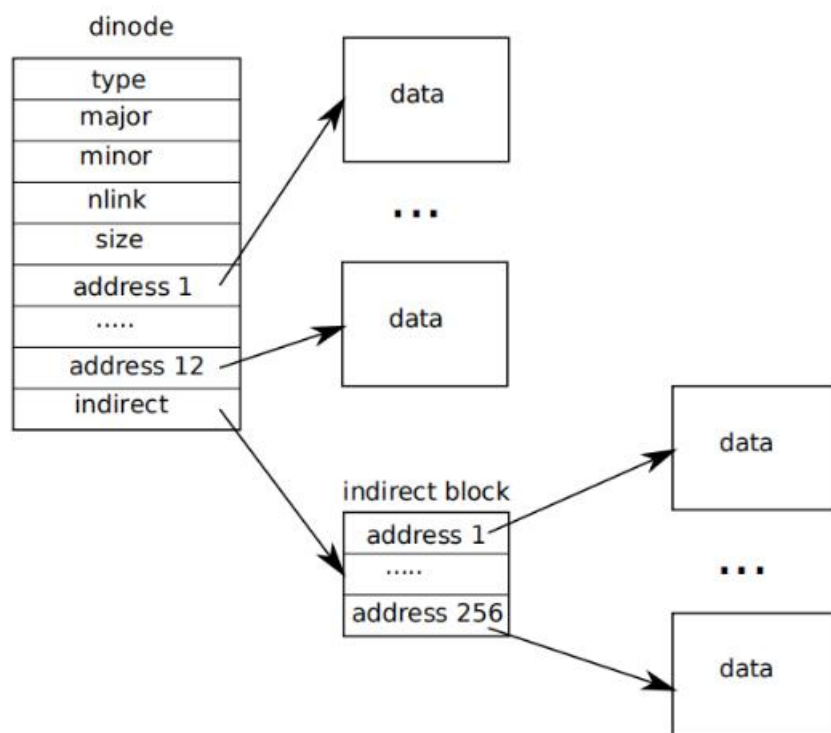
增加 xv6 文件的最大大小, 现在 xv6 文件的大小限制在 268 个块, 此限制来自以下事实: XV6 inode 包含 12 个“直接”块号和一个“单间接”块号 块号, 指最多可容纳 256 个块的块 数字, 总共 $12+256=268$ 个区块。

1.2 实验步骤

主要思想是: 将一个直接查找的位置变成二级索引。
这是 xv6 文件系统的结构图:



硬盘的数据布局如下图所示：



(1) 修改 dinode 结构体与 inode 结构体

```

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

```

```

short type;          // copy of disk inode
short major;
short minor;
short nlink;
uint size;
uint addrs[NDIRECT+2];
};
(2) 修改 bmap 函数，仿照一级索引块处理流程，二级索引块->一级索引块->数据块
static uint bmap(struct inode *ip, uint bn) {
    uint addr, *a, *b;
    struct buf *bp, *buffer;
    uint blk_no, blk_index;  // 一级索引块块号以及索引块下标

    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 < NDOUBLYINDIRECT) {
        // 载入二级索引块
        if ((addr = ip->addrs[NDIRECT + 1]) == 0) ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
        bp = bread(ip->dev, addr);
        a = (uint *)bp->data;
        // 计算块号与下标
        blk_no = bn / NINDIRECT;
        blk_index = bn % NINDIRECT;
        // 载入一级索引块
        if ((addr = a[blk_no]) == 0) {
            a[blk_no] = addr = balloc(ip->dev);
            log_write(bp);
        }
    }
}

```

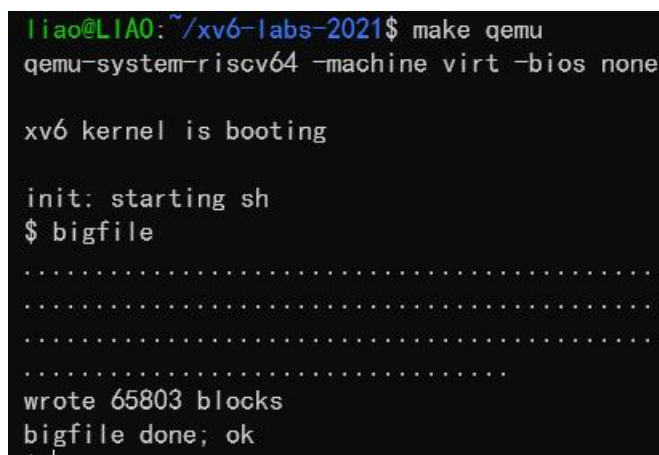
```

    buffer = bread(ip->dev, addr);
    b = (uint *)buffer->data;
    // 必要时分配数据块
    if ((addr = b[blk_index]) == 0) {
        b[blk_index] = addr = balloc(ip->dev);
        log_write(buffer);
    }
    // 释放 buffer, 返回数据块号
    brelse(buffer);
    brelse(bp);
    return addr;
}
panic("bmap: out of range");
}

```

- (3) 修改 itrunc 函数, 对索引块先释放 buffer 再 free, 数据块则直接 free。
- (4) 在 param.h 里边的 FSSIZE 参数改成 200000。

1.3 实验结果



```

liao@LIAO:~/xv6-labs-2021$ make qemu
qemu-system-riscv64 -machine virt -bios none

xv6 kernel is booting

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

```

1.4 实验心得

在进行文件系统的实验之前, 要对 xv6 的文件系统有一个大致的了解, 才有利于更顺利地地完成实验。

2. Symbolic links (moderate)

2.1 实验目的

在这部分的测试中, 你需要在 xv6 中添加符号链接。符号链接（软连接）会通过路径

名指向一个被连接的文件；当一个符号链接打开时，内核需要随着连接抵达被指向的文件。符号链接类似于硬链接，但是硬链接更加严格的指向在同一磁盘位置的文件，而且符号链接可以跨设备。虽然 xv6 不支持多设备，实现总共系统调用是一个理解路径名查找如何工作的好的机会。你将要实现 `symlink(char * target, char * path)` 系统调用，这会在 `path` 处创建一个新的符号引用，指向 `target`。

2.2 实验步骤

- (1) 首先需要为 `symlink` 创建一个新的系统调用号（`syscall.h` 中定义），向 `user/usuy.pl`、`User/user.h` 添加一个条目，并在 `kernel/sysfile.c` 中实现一个空的 `sys_symlink`。
- (2) 在 `kernel/stat.h` 中添加一个新的文件类型(`T_SYMLINK`)来表示符号链接。
- (3) 修改 `kernel/fcntl.h`，定义 `O_NOFOLLOW`。
- (4) 修改 `kernel/sysfile.c`。实现 `sys_symlink` 函数。

```
uint64 sys_symlink(void){
    char target[MAXPATH],path[MAXPATH];
    struct inode *dp,*ip;
    if(argstr(0,target,MAXPATH)<0 || argstr(1,path,MAXPATH)<0)
        return -1;
    begin_op();
    if((ip=namei(target))!=0){
        if(ip->type==T_DIR){
            goto bad;
        }
    }
    if((dp=create(path,T_SYMLINK,0,0))==0){
        goto bad;
    }
    if(writei(dp,0,(uint64)target,0,MAXPATH)!=MAXPATH){
        panic("symlink: writei");
    }
    iunlockput(dp);
    end_op();
    return 0;
bad:
    end_op();
    return -1;
}
```

- (5) 修改 `kernel/sysfile.c`。修改 `sys_open` 函数。
- (6) 修改 `kernel/Makefile`。

2.3 实验结果

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

3. Make grade

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (308.1s)
== Test running symlinktest ==
$ make qemu-gdb
(1.4s)
== Test  symlinktest: symlinks ==
symlinktest: symlinks: OK
== Test  symlinktest: concurrent symlinks ==
symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (450.1s)
== Test time ==
time: OK
Score: 100/100
```

Lab10 mmap

1. Mmap(hard)

1.1 实验目的

实现两个函数：mmap 和 munmap。

实验要求如下：

- (1) 映射相同 MAP_SHARED 文件的进程没有共享物理内存是可以的。
- (2) munmap(addr, length)应该移除已知地址范围的 mmap 映射。
- (3) 如果进程已经更改了内存，且映射为 MAP_SHARED，更改应该首先被写到文件。

- (4) `munmap` 调用可能只覆盖 `mmap-ed` 区域的一部分，但可以假设：要么 `unmap` 起始，要么 `unmap` 结束、要么整个区域（而不是在 `region` 中打孔）。
- (5) 你可以实现足够的 `map` 和 `munmap` 功能来通过 `mmaptest` 测试程序。
- (6) 如果是 `mmaptest` 不用的 `mmap` 特性，你无需实现。

1.2 实验步骤

`mmap` 是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对应关系。

`mmap` 函数的原型为：

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

成功执行时，`mmap()` 返回被映射区的指针。失败时，`mmap()` 返回 `MAP_FAILED` [其值为 `(void *)-1`]，`error` 被设为以下的某个值

- 1 EACCESS: 访问出错
- 2 EAGAIN: 文件已被锁定，或者太多的内存已被锁定
- 3 EBADF: `fd` 不是有效的文件描述词
- 4 EINVAL: 一个或者多个参数无效
- 5 ENFILE: 已达到系统对打开文件的限制
- 6 ENODEV: 指定文件所在的文件系统不支持内存映射
- 7 ENOMEM: 内存不足，或者进程已超出最大内存映射数量
- 8 EPERM: 权能不足，操作不允许
- 9 ETXTBSY: 已写的方式打开文件，同时指定 `MAP_DENYWRITE` 标志
- 10 SIGSEGV: 试着向只读区写入
- 11 SIGBUS: 试着访问不属于进程的内存区

参数中，`start` 是映射区的开始地址，`length` 是映射区的长度，`prot` 是期望的内存保护标志，不能与文件的打开模式冲突。`fd` 是有效的文件描述词，如果 `MAP_ANONYMOUS` 被设定，为了兼容问题，其值应该为 -1。`offset` 是被映射对象内容的起点。

实验的具体步骤如下：

- (1) 修改 `user/user.h`，新增 `mmap` 和 `munmap` 函数声明。
- (2) 修改 `kernel/syscall.h`，新增 `SYS_mmap` 和 `SYS_munmap` 定义。
- (3) 修改 `user/usys.pl`，新增 `mmap` 和 `munmap` 函数入口。
- (4) 修改 `kernel/syscall.c`。
- (5) 修改 `kernel/proc.c`，新增 `lazy_grow_proc` 函数，用于增长进程空间。

```
int lazy_grow_proc(int n)
{
    struct proc *p=myproc();
    p->sz=p->sz+n;
    return 0;
}
```

- (6) 修改 `kernel/defs.h`，新增 `lazy_grow_proc` 函数声明。
- (7) 新增 `kernel/vma.h`，新增一个结构体 `vm_area_struct`，用于表示 VMA。

```
struct vm_area_struct{
    char* addr;
```

```

        uint64 length;
        char prot;
        char flags;
        struct file *file;
    };

```

(8) 新增 kernel/vma.c, 用于初始化、分配、释放 VMA。

```

#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "param.h"
#include "fs.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "file.h"
#include "vma.h"

struct {
    struct spinlock lock;
    struct vm_area_struct areas[NOFILE];
}vma_table;

void
vma_init(void)
{
    initlock(&vma_table.lock, "vma_table");
}

struct vm_area_struct*
vma_alloc(void)
{
    struct vm_area_struct *vmap;
    acquire(&vma_table.lock);
    for(vmap=vma_table.areas;vmap<vma_table.areas+NOFILE;vmap++){
        if(vmap->file==0){
            release(&vma_table.lock);
            return vmap;
        }
    }
    release(&vma_table.lock);
    return 0;
}

void vma_free(struct vm_area_struct *vmap)
{

```

- ```

 vmap->file=0;
 }

```
- (9) 修改 kernel/defs.h, 新增 vma\_init、vma\_alloc、vma\_free 函数声明。
  - (10) 修改 kernel/main.c, 用于初始化 VMA。
  - (11) 修改 kernel/proc.h, 在结构体 proc 中, 新增结构体 vm\_area\_struct 的指针数组 areaps。
  - (12) 修改 kernel/sysfile.c, 新增函数 sys\_mmap。

```

uint64
sys_mmap(void)
{
 struct vm_area_struct *vmap;
 struct proc *pr;
 int length, prot, flags, fd, i;
 uint64 sz;

 if(argint(1, &length) < 0 || argint(2, &prot) < 0 ||
 argint(3, &flags) < 0 || argint(4, &fd) < 0){
 return 0xffffffffffff;
 }
 pr = myproc();
 if(!pr->ofile[fd]->readable){
 if(prot & PROT_READ)
 return 0xffffffffffff;
 }
 if(!pr->ofile[fd]->writable){
 if(prot & PROT_WRITE && flags == MAP_SHARED)
 return 0xffffffffffff;
 }

 if((vmap = vma_alloc()) == 0){
 return 0xffffffffffff;
 }

 acquire(&pr->lock);
 for(i = 0; i < NOFILE; i++){
 if(pr->areaps[i] == 0){
 pr->areaps[i] = vmap;
 release(&pr->lock);
 break;
 }
 }
 if(i == NOFILE){
 return 0xffffffffffff;
 }
 sz = pr->sz;

```

```

 if(lazy_grow_proc(length)<0){
 return 0xffffffffffff;
 }

 vmap->addr=(char *)sz;
 vmap->length=length;
 vmap->prot=(prot & PROT_READ) | (prot & PROT_WRITE);
 vmap->flags=flags;
 vmap->file=pr->ofile[fd];
 filedup(pr->ofile[fd]);
 return sz;
}

```

(13) 修改 kernel/sysfile.c, 新增函数 sys\_munmap。

```

uint64
sys_munmap(void)
{
 struct proc *pr=myproc();
 int startAddr,length;

 if(argint(0,&startAddr)<0 || argint(1,&length)<0){
 return -1;
 }

 for(int i=0;i<NOFILE;i++){
 if(pr->areaps[i]==0)
 continue;
 if((uint64)pr->areaps[i]->addr==startAddr){
 if(length>=pr->areaps[i]->length){
 length = pr->areaps[i]->length;
 }
 if(pr->areaps[i]->prot & PROT_WRITE &&
 pr->areaps[i]->flags==MAP_SHARED){
 begin_op();
 ilock(pr->areaps[i]->file->ip);
 writei(pr->areaps[i]->file->ip,1,(uint64)startAddr,0,length);
 iunlock(pr->areaps[i]->file->ip);
 end_op();
 }
 uvmunmap(pr->pagetable,(uint64)startAddr,length/PGSIZE,1);
 if(length==pr->areaps[i]->length){
 fileclose(pr->areaps[i]->file);
 vma_free(pr->areaps[i]);
 }
 }
 }
}

```

```

 pr->areaps[i]=0;
 return 0;
 }else{
 pr->areaps[i]->addr+=length;
 pr->areaps[i]->length-=length;
 return 0;
 }
}
}
return -1;
}

```

(14) 修改 kernel/trap.c, 修改函数 usertrap, 实现 page lazy allocation。

```

else if(r_scause() == 13 || r_scause() == 15){
 uint64 stval=r_stval();
 if(stval >= p->sz){
 p->killed=1;
 } else {
 uint64 protectTop =PGROUNDDOWN(p->trapframe->sp);
 uint64 stvalTop =PGROUNDUP(stval);
 if(protectTop != stvalTop){
 struct vm_area_struct *vmap;
 int i;
 uint64 addr;
 for(i=0;i<NOFILE;i++){
 if(p->areaps[i]==0){
 continue;
 }
 addr = (uint64) (p->areaps[i]->addr);
 if(addr <= stval && stval < addr +p->areaps[i]->length){
 vmap =p->areaps[i];
 break;
 }
 }
 if(i!=NOFILE){
 char *mem =kalloc();
 int prot =PTE_U;
 if(mem==0){
 p->killed=1;
 } else {
 memset(mem,0,PGSIZE);
 ilock(vmap->file->ip);
 readi(vmap->file->ip,0,(uint64)mem,PGROUNDDOWN(stval-addr),PGSIZE);
 iunlock(vmap->file->ip);
 if(vmap->prot & PROT_READ){

```

```

 prot |= PTE_R;
 }
 if(vmap->prot & PROT_WRITE){
 prot |= PTE_W;
 }
 if(mappages(p->pagetable,PGROUNDDOWN(stval),PGSIZE,(uint64)mem,prot)!=0){
 kfree(mem);
 p->killed =1;
 }
}
}else{
 p->killed=1;
}
}else{
 p->killed=1;
}
}
}

```

(15) 修改 kernel/vm.c, 修改 uvmunmap 函数, 实现 page lazy allocation。

(16) 修改 kernel/proc.c, 修改函数 exit。

(17) 修改 kernel/proc.c, 修改函数 fork。

(18) 修改 Makefile。

## 1.3 实验结果



```
hart 1 starting
hart 2 starting
init: starting sh
$ mmaptest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
fork_test OK
mmaptest: all tests succeeded
```

```
test pipe1: OK
test killstatus: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

## 2. Make grade

```
== Test running mmaptest ==
$ make qemu-gdb
(3.0s)
== 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 (107.8s)
== Test time ==
time: OK
Score: 140/140
```