

写在前面

代码地址：[xv6-lab](#)

所有实验都需要提交一个耗时文件 `time.txt`，由于没有提交，所以在评分时最后一项 `failed`，减1分

Lab 1 Utilities

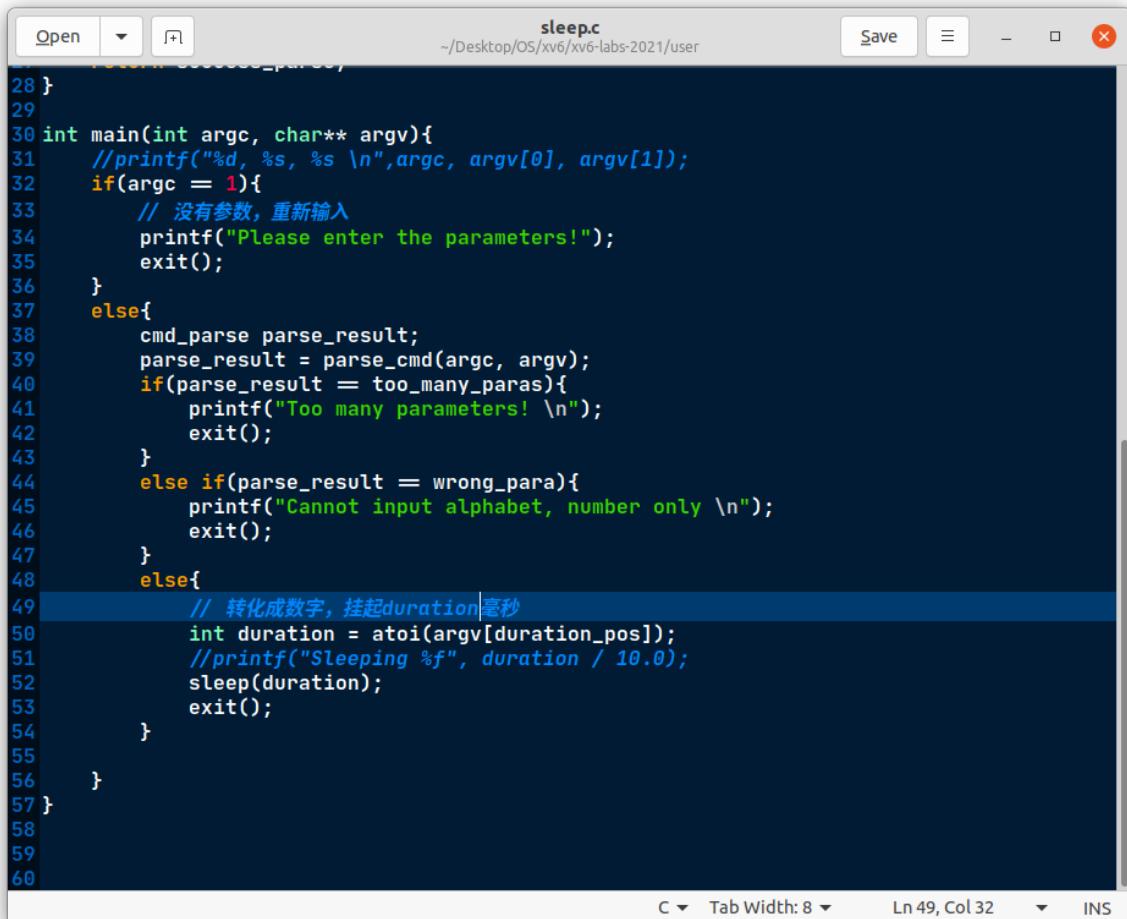
Assignment 1 —— sleep

这个任务要求利用调用**系统调用函数sleep**完成sleep n。我们调用的是syscall的sleep，这时xv6提供的，而不是Linux环境中的`<sys>`中的sleep。

步骤：

- 在user文件夹下新建sleep.c，判断一下输入格式，调用一下sleep即可。
- 导包，user.h为xv6提供的系统函数，types.h为其提供的变量类型。

sleep.c文件：



The screenshot shows a code editor window with the file `sleep.c` open. The code is written in C and implements a simple sleep command. It checks for parameters, parses them, and then calls the `sleep` syscall. The code is color-coded for readability.

```
28 }
29
30 int main(int argc, char** argv){
31     //printf("%d, %s, %s \n", argc, argv[0], argv[1]);
32     if(argc == 1){
33         // 没有参数，重新输入
34         printf("Please enter the parameters!");
35         exit();
36     }
37     else{
38         cmd_parse parse_result;
39         parse_result = parse_cmd(argc, argv);
40         if(parse_result == too_many_params){
41             printf("Too many parameters! \n");
42             exit();
43         }
44         else if(parse_result == wrong_para){
45             printf("Cannot input alphabet, number only \n");
46             exit();
47         }
48         else{
49             // 转化成数字，挂起duration毫秒
50             int duration = atoi(argv[duration_pos]);
51             //printf("Sleeping %f", duration / 10.0);
52             sleep(duration);
53             exit();
54         }
55     }
56 }
57 }
```

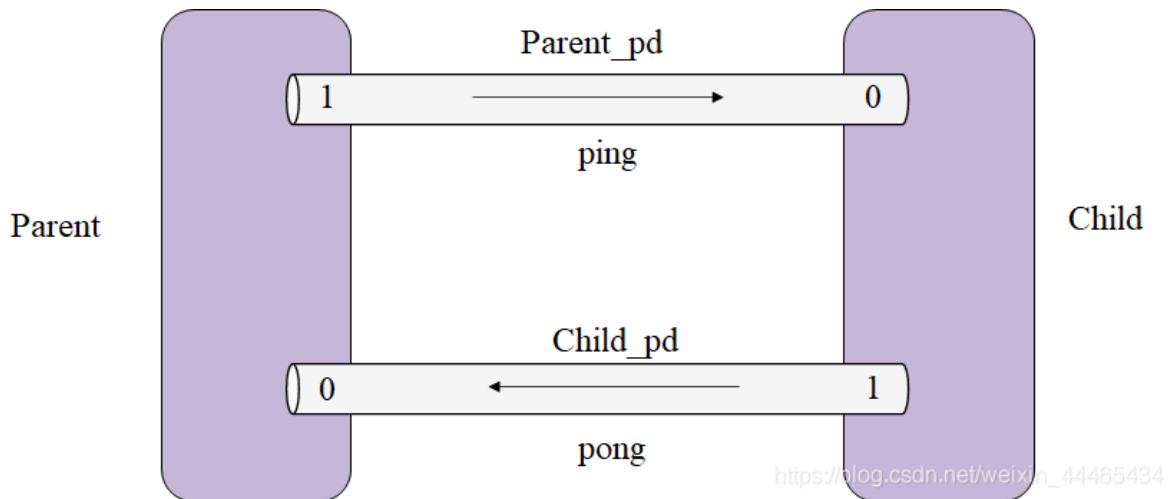
cmd_parse是枚举类型，作为parse_cmd函数的返回类型.enum {wrong_para, success_parse, too_many_paras} cmd_parse三个值分别表示输入的sleep参数错误、输入成功、太多参数。

sleep运行结果及自动评价：

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ make qemu  
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0  
  
xv6 kernel is booting  
  
hart 2 starting  
hart 1 starting  
init: starting sh  
$ sleep  
Please enter the parameters!$  
$ sleep 15s5  
Cannot input alphabet, number only  
$ sleep 10  
$ QEMU: Terminated  
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ ./grade-lab-util sleep  
make: 'kernel/kernel' is up to date.  
== Test sleep, no arguments == sleep, no arguments: OK (1.2s)  
== Test sleep, returns == sleep, returns: OK (1.0s)  
== Test sleep, makes syscall == sleep, makes syscall: OK (0.9s)
```

Assignment 2 —— pingpong

本任务要求实现利用管道实现进程间的通信：父进程发送ping，子进程收到后发送pong，父进程收到后将其打印出来，下图方便理解



Hints 1：利用pipe()函数创建管道，pipe()函数接收一个长度为2的数组，数组下标0为读端、1为写端；

Hints 2：利用fork()函数创建新的进程；

Hints 3：利用read、write函数读写管道；

fd是文件描述符file description flag。每个进程都有一张表，而xv6 内核就以文件描述符作为这张表的索引，所以每个进程都有一个从0开始的文件描述符空间。按照惯例，进程从文件描述符0读入（标准输入），从文件描述符1输出（标准输出），从文件描述符2输出错误（标准错误输出）。管道两端正是文件描述符，1表示输出，0表示读入，在其中任意一端操作都要关闭另一端(通过close系统调用)

```
pingpong.c
~/Desktop/OS/xv6/xv6-labs-2021/user

5 * @brief fd是文件描述符file description flag。每个进程都有一张表,
6 * 而xv6 内核就以文件描述符作为这张表的索引，所以每个进程都有一个从0开
7 * 始的文件描述符空间。按照惯例，进程从文件描述符0读入（标准输入），从
8 * 文件描述符1输出（标准输出），从文件描述符2输出错误（标准错误输出）。
9 * 我们会看到shell 正是利用了这种惯例来实现I/O 重定向。shell 保证在任
10 * 何时候都有3个打开的文件描述符 (8007) ，他们是控制台 (console) 的默
11 * 认文件描述符。
12 *
13 */
14
15 int main(int argc, char **argv)
16 {
17     int pid;
18     int parent_fd[2];
19     int child_fd[2];
20     char buf[20];
21     //为父子进程建立管道
22     pipe(child_fd);
23     pipe(parent_fd);
24
25     // Child Progress
26     if ((pid = fork()) == 0)
27     {
28         // 关闭写
29         close(parent_fd[1]);
30         read(parent_fd[0], buf, 4);
31         // getpid() 得到当前进程的pid
32         printf("%d: received %s\n", getpid(), buf);
33         close(child_fd[0]);
34         write(child_fd[1], "pong", sizeof(buf));
35         exit(0);
36     }
37 }
```

C Tab Width: 8 Ln 14, Col 1 INS

main函数中展示了子中的进程操作，先从管道中读出 buffer中四个字节，读之前先关闭管道的写端，读完后输出子进程接收到'ping'，然后向另一管道（xv6中的管道是单向的）中写端，同理，写之前应该关闭该管道的读一端。

pingpong运行结果及自动评价：

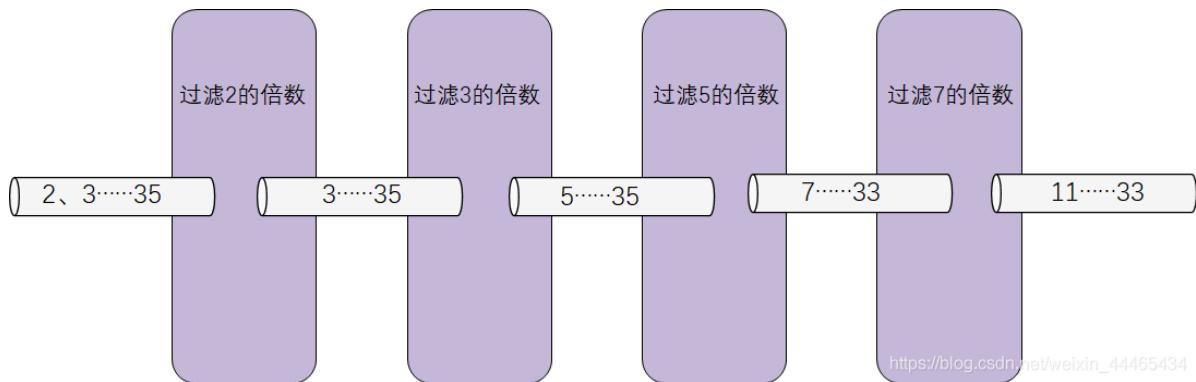
```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ make qemu
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -DLAB_UTIL -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie
-c -o user/primes.o user/primes.c
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_primes user/primes.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
riscv64-unknown-elf-objdump -S user/_primes > user/primes.asm
riscv64-unknown-elf-objdump -t user/_primes | sed '1,/SYMBOL TABLE/d; s/.* //; /^$/d' > user/primes.sym
mkfs/mkfs fs.img README user/xargstest.sh user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind user/_wc user/_zombie user/_sleep user/_pingpong user/_primes user/_find user/_xargs
nmeta 4G (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000
balloc: first 724 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ pingpong
4: received ping
3: received pong
$ QEMU: Terminated
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (0.8s)
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$
```

Assignment 3 —— primes

本任务要求完成质数筛选器，它要求用fork和pipe实现：输入为2 ~ 35，输出为2 ~ 35间的所有质数，例如：2、3、5、7等。第一次我们将2 ~ 35给到一个进程，这个进程发现给到的第一个数为2，则输出2，然后将不能被2除尽的数（3、5、7、9.....）发送给下一个进程，下一个进程发现给到的第一个数为3，则输出3，然后将不能被3除尽的数（5、7.....）发送给下一个进程.....以此类推。下图辅助理解：



`read(fd, buf, n)` 从 `fd` 读最多 `n` 个字节（`fd` 可能没有 `n` 个字节），将它们拷贝到 `buf` 中，然后返回读出的字节数。每一个指向文件的文件描述符都和一个偏移关联（文件内指针）。`read` 从当前文件偏移处读取数据，然后把偏移增加读出字节数。紧随其后的 `read` 会从新的起点开始读数据。当没有数据可读时，`read` 就会返回0，这就表示文件结束了。

`write(fd, buf, n)` 写 `buf` 中的 `n` 个字节到 `fd` 并且返回实际写出的字节数。如果返回值小于 `n` 那么只可能是发生了错误。就像 `read` 一样，`write` 也从当前文件的偏移处开始写，在写的过程中增加这个偏移。

A screenshot of a terminal window titled "primes.c" with the path "/Desktop/OS/xv6/xv6-labs-2021/user". The window contains the source code for a C program named "primes.c". The code includes headers for kernel/types.h, kernel/stat.h, and user/user.h. It defines constants RD (0) and WR (1). The prime function takes an array p1[2] and reads from its read end (p1[RD]). If the read fails, it prints "prime %d\n" and exits. Otherwise, it creates a pipe p2, forks, and writes to p2[WR]. The parent then closes p1[RD]. The code is numbered from 1 to 34.

```
1 #include "kernel/types.h"
2 #include "kernel/stat.h"
3 #include "user/user.h"
4
5 #define RD 0
6 #define WR 1
7
8 void prime(int p1[2])
9 {
10     close(p1[WR]);
11     int p;
12     if (read(p1[RD], &p, sizeof(int)) != 0)
13     {
14         fprintf(1, "prime %d\n", p);
15     }
16     else
17     {
18         exit(0);
19     }
20     int n;
21     int p2[2];
22     pipe(p2);
23     int pid = fork();
24     if (pid > 0)
25     {
26         close(p2[RD]);
27         while (read(p1[RD], &n, sizeof(int)) != 0)
28         {
29             if (n % p != 0)
30             {
31                 write(p2[WR], &n, sizeof(int));
32             }
33         }
34     }
35     close(p1[RD]);
```

主函数如下：

```
1 int main()
2 {
3     // 创建管道
4     int p1[2];
5     pipe(p1);
6     // 创建下一子进程
7     int pid = fork();
8     if (pid < 0)
9     {
10         fprintf(2, "fork error!\n");
11         close(p1[RD]);
12         close(p1[WR]);
13         exit(1);
14     }
15     // child
16     else if (pid == 0)
17     {
18         prime(p1);
19     }
20     // parent
21     else
22     { // 关闭读
23         close(p1[RD]);
```

```
24     for (int i = 2; i <= 35; i++)
25     {
26         write(p1[WR], &i, sizeof(int));
27     }
28     close(p1[WR]);
29     wait((int *)0);
30 }
31
32 exit(0);
33 }
```

主函数中创建初始管道，初始化管道中写入2~35数字，prime函数中是递归的子进程和管道，筛选每一个数的倍数，第一次递归输出2，然后筛选掉2的倍数；第二次输出3，递归筛选掉3的倍数；第三次输出5(4被筛掉了)，筛选掉5的倍数.....依次类推。

primes运行结果及自动评价：

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$ QEMU: Terminated
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (0.9s)
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$
```

Assignment 4 —— find

本任务要求写一个find函数，其基本用法为 find arg1 arg2，即在目录arg1下找到arg2。这和file system有关。任务的一个Hint：Look at user/ls.c to see how to read directories。参考ls.c的写法。再复制grep.c的正则匹配代码，匹配目录或文件

```
find.c
~/Desktop/OS/xv6/xv6-labs-2021/user

5
6 void find(char *path, char *file)
7 {
8     int fd;
9     struct stat st;
10    struct dirent de;
11    char buf[512];
12    char *p;
13    if ((fd = open(path, 0)) < 0)
14    {
15        fprintf(2, "can not open%d\n", path);
16        exit(1);
17    }
18
19    if ((fstat(fd, &st)) < 0)
20    {
21        fprintf(2, "can not stat%d\n", path);
22        close(fd);
23        exit(1);
24    }
25    switch (st.type)
26    {
27        case T_FILE:
28            fprintf(2, "please find file in dir");
29            close(fd);
30            break;
31        case T_DIR:
32            strcpy(buf, path);
33            p = buf + strlen(buf);
34            *p++ = '/';
35            while (read(fd, &de, sizeof(de)) == sizeof(de))
36            {
37                if (de.inum == 0)
38                    continue;
```

其中用到了两个函数：`fstat` 可以获取一个文件描述符指向的文件的信息。它填充一个名为 `stat` 的结构体（程序中为实例化的`st`），它在 `stat.h` 中定义为：

```
1 struct stat {
2     int dev;      // File system's disk device
3     uint ino;     // Inode number
4     short type;   // Type of file
5     short nlink;  // Number of links to file
6     uint64 size;  // Size of file in bytes
7 };
```

`open` 函数返回指定路径的文件描述符，用以传递给 `fstat` 函数

find运行结果及自动评价：

```

$ echo > b
$ mkdir ./a
$ echo > a/b
$ ls
.
..
README          1 1 1024
xargstest.sh    2 2 2226
cat             2 3 93
echo            2 4 23984
forktest        2 5 22816
grep            2 6 13192
init            2 7 27344
kill             2 8 23920
ln               2 9 22784
ls               2 10 22744
mkdir            2 11 26224
rm               2 12 22896
sh               2 13 22880
stressfs        2 14 41768
usertests       2 15 23888
grind            2 16 156104
wc               2 17 38056
zombie           2 18 25128
sleep            2 19 22288
pingpong         2 20 24488
primes           2 21 23456
find             2 22 26608
xargs            2 23 25512
console          2 24 24128
3 25 0

```

```

b          2 26 0
a          1 27 48
$ find . b
./b
./a/b
$ QEMU: Terminated
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (0.8s)
  (Old xv6.out.find_curdir failure log removed)
== Test find, recursive == find, recursive: OK (1.2s)
  (Old xv6.out.find_recursive failure log removed)

```

Assignment 5 —— xargs★

```

5 #include"kernel/fs.h"
6 #include"kernel/param.h"
7 int main(char argc,char * argv[]){
8     char buf;
9     char command[DIRSIZ];
10    char *args[MAXARG]={0};
11    int argnum=0;
12    if (argc>1){
13        strcpy(command,argv[1]);
14        while (argnum<(argc-1))
15        {
16            args[argnum]=argv[argnum+1];
17            argnum++;
18        }
19        char line[128];
20        int linenum=0;
21        while (read(0,&buf,1)>0)
22        {
23            if (buf=='\n'){
24                line[linenum]=0;
25                args[argnum]=line;
26                int pid=fork();
27                if(pid<0{
28                    fprintf(2,"xargs: fork error");
29                }else if (pid==0){
30                    exec(command,args);
31                    exit(0);
32                }else{
33                    linenum=0;
34                    wait((int *)0);
35                }
36            }else{
37                line[linenum++]=buf;
38            }
39        }
40    }
41 }

```

多参数实现，当用户输入"ctrl+d"时停止参数的输入。这里要说明的是：对于输入的命令，我们要用exec执行，其中exec接收两个参数，第一个参数为命令cmd，第二个参数为一个数组，该数组的格式必须为{cmd, "arg1", "arg2", ..., 0}，以下是xargstest.sh中的内容

```
1 $ $ cat xargstest.sh
2 mkdir a
3 echo hello > a/b
4 mkdir c
5 echo hello > c/b
6 echo hello > b
7 find . b | xargs grep hello
8
```

xargs运行结果及自动评价：

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ cat xargstest.sh
mkdir a
echo hello > a/b
mkdir c
echo hello > c/b
echo hello > b
find . b | xargs grep hello
$ sh < xargstest.sh
$ mkdir: a failed to create
$ $ mkdir: c failed to create
$ $ $ hello
hello
hello
$ $ QEMU: Terminated
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (1.3s)
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$
```

由于第一次实验时已经创建了a和b文件夹，所以再一次创建会失败。

Lab 2 Syscall

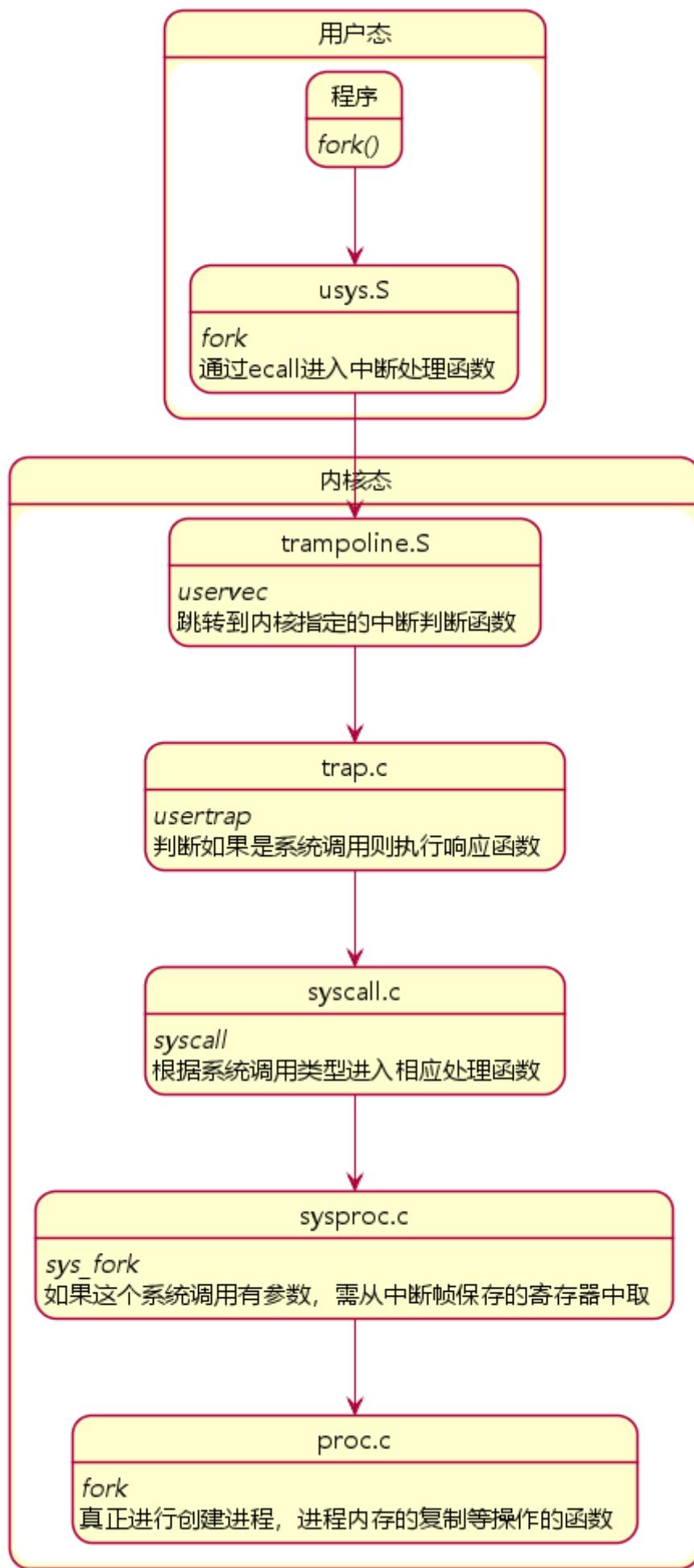
该实验添加两个系统调用trace和sysinfo

Assignment 1 —— System call tracing

本任务中要求添加一个系统调用跟踪功能，该功能可能会在以后调试实验时帮助您。您将创建一个新的trace系统调用来控制跟踪。它应该采用一个参数mask，int整数类型，其位指定要跟踪的系统调用。例如，为了跟踪fork系统调用，程序调用trace(1 << SYS_fork)，其中SYS_fork是来自kernel/syscall.h的系统调用号。如果掩码中设置了系统调用号，则必须修改xv6内核以在每个系统调用即将返回时打印一行。该行应包含

进程 id、系统调用的名称和返回值；不需要打印系统调用参数。`trace` 系统调用应该启用对调用它的进程以及它随后产生的任何子进程的跟踪，但不应影响其他进程。

系统调用原理（以 fork 为例）：



该实验需要打印其他系统调用的信息。根据上面的分析和文档说明，首先需要给 user.h、usys.pl（用来生成usys.S的辅助脚本）和syscall.h添加对应的函数的系统调用号，然后给syscall.c的系统调用数组添加对应的函数指针和函数头，在sysproc.c添加对应的函数实现，sysproc.c里主要是接收参数并给proc结构体复制，具体代码如下：

```
1 # kernel/sysproc.c
2 uint64 sys_trace(void) {
3     int mask;
4     if (argint(0, &mask) < 0) return -1;
5     myproc()→mask = mask; return 0;
6 }
7
8 # user/user.h
9 // system calls
10 ...
11 int trace(int); //添加trace原型
12
13 # user/usys.pl
14 ...
15 entry("trace");
16
17 # kernel/syscall.h
18 // System call numbers
19 ...
20 #define SYS_trace 22
21
22 # kernel/proc.h
23 // Per-process state
24 struct proc {
25     ...
26     char name[16];           // Process name (debugging)
27     int mask;                // Tracing parameter
28 };
29
30 # kernel/syscall.c
31 ...
32 extern uint64 sys_trace(void);
33
34 static uint64 (*syscalls[])(void) = {
35     ...
36     [SYS_trace]    sys_trace,
37 };
38
39 char* syscalls_name[] = {
40     [SYS_fork]      "fork",
41     [SYS_exit]      "exit",
42     [SYS_wait]      "wait",
```

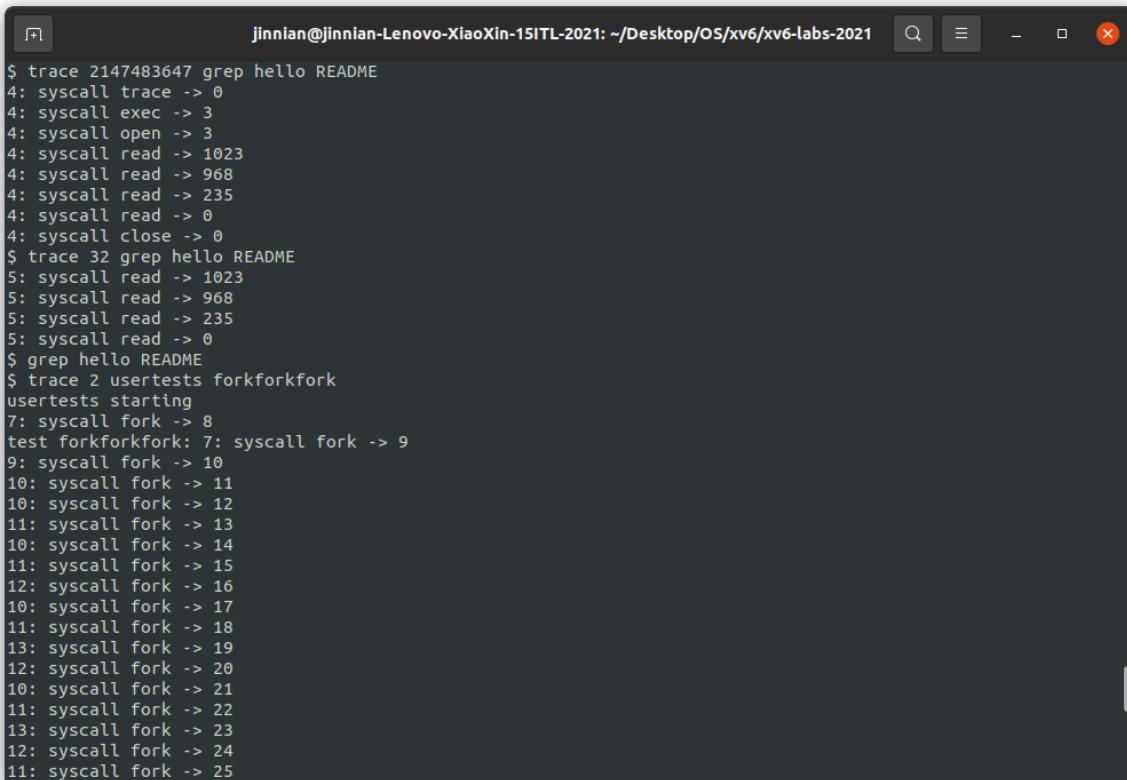
```

43 [SYS_pipe]      "pipe",
44 [SYS_read]       "read",
45 [SYS_kill]       "kill",
46 [SYS_exec]       "exec",
47 [SYS_fstat]     "fstat",
48 [SYS_chdir]     "chdir",
49 [SYS_dup]        "dup",
50 [SYS_getpid]    "getpid",
51 [SYS_sbrk]       "sbrk",
52 [SYS_sleep]     "sleep",
53 [SYS_uptime]    "uptime",
54 [SYS_open]       "open",
55 [SYS_write]     "write",
56 [SYS_mknod]     "mknod",
57 [SYS_unlink]    "unlink",
58 [SYS_link]       "link",
59 [SYS_mkdir]     "mkdir",
60 [SYS_close]     "close",
61 [SYS_trace]     "trace",
62 };
63
64 void
65 syscall(void)
66 {
67     ...
68     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
69         p->trapframe->a0 = syscalls[num]();
70         if((1 << num) & p->mask){//mask的二进制下为1的数位表示需要跟踪的进程
71             printf("%d: syscall %s → %d\n",
72                   p->pid, syscalls_name[num], p->trapframe->a0);
73         }
74     }
75     ...
76 }
77
78 # kernel/proc.c
79 int
80 fork(void)
81 {
82     ...
83     safestrcpy(np->name, p->name, sizeof(p->name));
84     np->mask = p->mask; //添加mask给子进程
85     pid = np->pid;
86     ...
87 }
88
89 static struct proc*
90 allocproc(void)

```

```
91 {  
92     ...  
93     p->context.sp = p->kstack + PGSIZE;  
94     p->mask = 0; // 给每个进程mask赋个初值即可。  
95  
96     return p;  
97 }
```

trace 运行结果及自动评测：结构是 pid: syscall '调用类型' -> 返回值



A terminal window titled 'jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6/xv6-labs-2021' displays the output of a trace command. The output shows a sequence of system calls and their arguments. The first part of the output shows reads from a file named 'hello'. The second part shows a series of fork operations, starting with a test fork, followed by user tests, and then a sequence of 14 fork operations. The output is as follows:

```
$ 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  
$ trace 32 grep hello README  
5: syscall read -> 1023  
5: syscall read -> 968  
5: syscall read -> 235  
5: syscall read -> 0  
$ grep hello README  
$ trace 2 usertests forkforkfork  
usertests starting  
7: syscall fork -> 8  
test forkforkfork: 7: syscall fork -> 9  
9: syscall fork -> 10  
10: syscall fork -> 11  
10: syscall fork -> 12  
11: syscall fork -> 13  
10: syscall fork -> 14  
11: syscall fork -> 15  
12: syscall fork -> 16  
10: syscall fork -> 17  
11: syscall fork -> 18  
13: syscall fork -> 19  
12: syscall fork -> 20  
10: syscall fork -> 21  
11: syscall fork -> 22  
13: syscall fork -> 23  
12: syscall fork -> 24  
11: syscall fork -> 25
```

```
10: syscall fork -> 42
12: syscall fork -> 43
11: syscall fork -> 44
13: syscall fork -> 45
12: syscall fork -> 46
10: syscall fork -> 47
13: syscall fork -> 48
12: syscall fork -> 49
10: syscall fork -> 50
11: syscall fork -> 51
13: syscall fork -> 52
12: syscall fork -> 53
11: syscall fork -> 54
13: syscall fork -> 55
12: syscall fork -> 56
10: syscall fork -> 57
13: syscall fork -> 58
12: syscall fork -> 59
11: syscall fork -> 60
10: syscall fork -> 61
13: syscall fork -> 62
12: syscall fork -> 63
10: syscall fork -> 64
11: syscall fork -> 65
13: syscall fork -> 66
12: syscall fork -> 67
10: syscall fork -> 68
13: syscall fork -> 69
12: syscall fork -> -1
10: syscall fork -> -1
13: syscall fork -> -1
11: syscall fork -> -1
OK
7: syscall fork -> 70
ALL TESTS PASSED
```

Assignment 2 —— Sysinfo

在这个任务中，添加一个系统调用 `sysinfo`，它收集有关正在运行的系统的信息。系统调用有一个参数：指向 `struct sysinfo` 的指针（参见 `kernel/sysinfo.h`）。内核应填写此结构的字段：`freemem` 字段应设置为空闲内存的字节数，`nproc` 字段应设置为状态不是 `UNUSED` 即正在进行的进程数。官方提供了一个测试程序 `sysinfotest`；如果它打印“`sysinfotest: OK`”，表示通过了这个任务。如下：

```
1 struct sysinfo {
2     uint64 freemem;      // amount of free memory (bytes)
3     uint64 nproc;        // number of process
4 };
```

同上面的 `syscall` 一样，首先需要给 `user.h`、`usys.pl`（用来生成 `usys.S` 的辅助脚本）和 `syscall.h` 添加对应的函数和系统调用号。

2.1 获取空闲内存和空闲进程

接下来，完善 `getfreemem()` 和 `getnproc()` 这两个函数。

getfreemem()

根据提示查看 `kernel/kalloc.c` 文件，从中可以看出，`xv6` 系统维护着一个 `kmem.freelist` 链表来存储空闲页，遍历 `kmem.freelist` 即可计算出空闲内存。因为要遍历该链表，为防止冲突，需要先获取 `lock`，计算完后再 `release`。

```
1 # kernel/kalloc.c
2 uint64 getfreemem(void)
3 {
4     uint64 freemem = 0;
5     struct run *r;
6     // 循环等待锁
7     acquire(&kmem.lock);
8     r = kmem.freelist;
9     while(r){
10         freemem += PGSIZE; //若r存在，加上页的内存大小
11         r = r->next;
12     }
13     // 释放锁
14     release(&kmem.lock);
15     return freemem;
16 }
17
```

getnproc()

根据提示查看 `kernel/proc.c`，其内部定义了 `struct proc proc[NPROC]`，所以我们只要遍历这个数组，统计其中 `state != UNUSED` 的个数即可。

```
1 uint64 getnproc(void)
2 {
3     uint64 res = 0;
4     struct proc *p;
5     for(p = proc; p < &proc[NPROC]; p++){
6         if(p->state != UNUSED) res++;
7     }
8     return res;
9 }
```

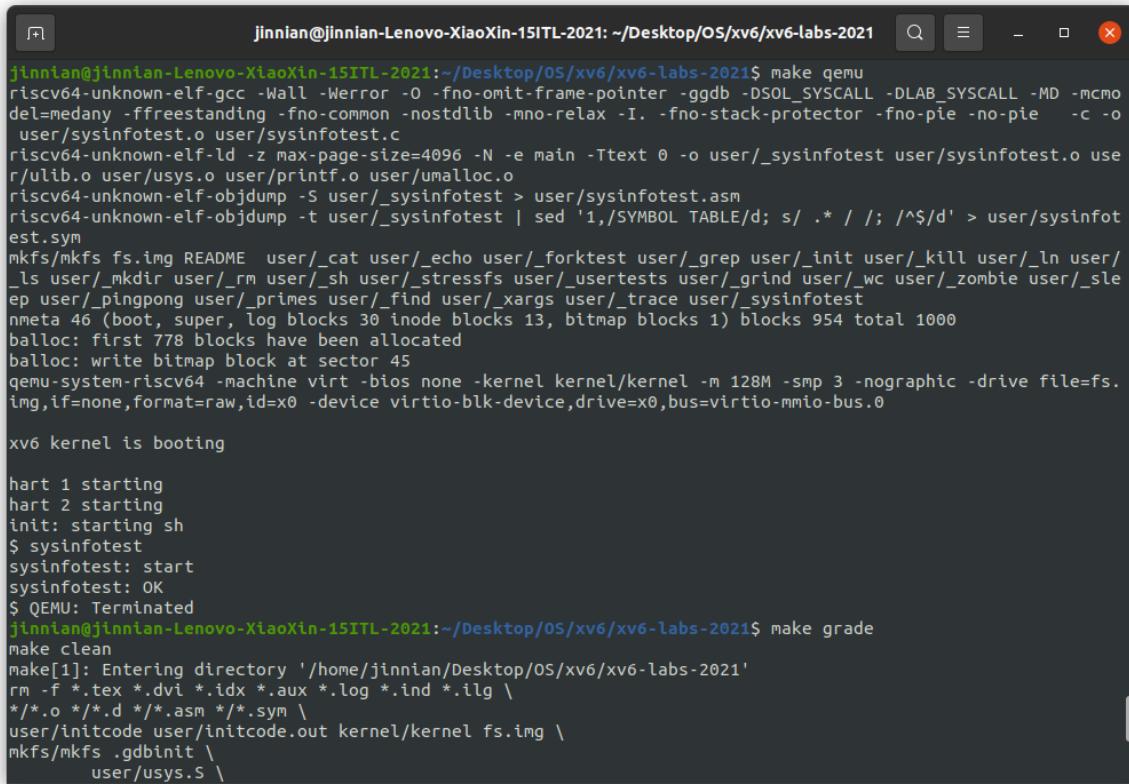
2.2 sysinfo函数

仿照 `sys_fstat()` (`kernel/sysfile.c`) 和 `filestat()` (`kernel/file.c`) 即可实现 `sys_sysinfo()` 的核心功能，把内核中的结构体 `sysinfo` 拷贝到用户区的指针中。

```
1 # kernel/sysproc.c
2 uint64 sys_sysinfo(void)
3 {
4     uint64 addr;
5     struct sysinfo sinfo;
6     // 获取用户态传入的sysinfo结构体
7     if(argaddr(0, &addr) < 0)
8         return -1;
```

```
9     sinfo.freemem = getfreemem();
10    sinfo.nproc = getnproc();
11    // 将内核态中的info复制到用户态
12    if(copyout(myproc()→pagetable, addr, (char *)&sinfo,
13      sizeof(sinfo)) < 0)
14        return -1;
15    return 0;
16 }
```

运行结果及自动评分：



The screenshot shows a terminal window with the following output:

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ make qemu
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_SYSCALL -DLAB_SYSCALL -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/sysinfotest.o user/sysinfotest.c
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_sysinfotest user/sysinfotest.o user/r/lib.o user/usys.o user/printf.o user/umalloc.o
riscv64-unknown-elf-objdump -S user/_sysinfotest > user/sysinfotest.asm
riscv64-unknown-elf-objdump -t user/_sysinfotest | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/sysinfotest.sym
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind user/_wc user/_zombie user/_sleep user/_pingpong user/_primes user/_fnd user/_xargs user/_trace user/_sysinfotest
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000
balloc: first 778 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$ QEMU: Terminated
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6/xv6-labs-2021$ make grade
make clean
make[1]: Entering directory '/home/jinnian/Desktop/OS/xv6/xv6-labs-2021'
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
/*/*.o /*/*.d /*/*.asm /*/*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
```

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6/xv6-labs-2021
del=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -march=rv64g -nostdinc -I. -Ikernel -c user/initcode.S -o user/initcode.o
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e start -Ttext 0 -o user/initcode.out user/initcode.o
riscv64-unknown-elf-objcopy -S -O binary user/initcode.out user/initcode
riscv64-unknown-elf-objdump -S user/initcode.o > user/initcode.asm
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/kall
oc.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.
o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o k
ernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/sta
rt.o kernel/console.o kernel/printf.o kernel/uart.o kernel/spinlock.o
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^\$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/jinnian/Desktop/OS/xv6/xv6-labs-2021'
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (2.2s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.3s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (10.4s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (1.5s)
(Old xv6.out.sysinfotest failure log removed)
== Test time ==
time: FAIL
    Cannot read time.txt
Score: 34/35
make: *** [Makefile:343: grade] Error 1
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6/xv6-labs-2021$
```

由于实验最后有一项提交 `time.txt` 文件的要求，没有提交故 `FAIL`

Lab 3 Page Table

Assignment 1 —— `vmprint`

定义一个名为 `vmprint()` 的函数。它应该接受一个 `pagetable_t` 参数，并以下面描述的格式打印该页表。在 `exec.c` 中的 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)`，以打印第一个进程的页表，通过 `make Grade` 的 `pte` 打印输出测试。

Hints:

- 可以将 `vmprint()` 放在 `kernel/vm.c` 中。
- 使用文件 `kernel/riscv.h` 末尾的宏。
- `freewalk` 的功能可能有用。
- 在 `kernel/defs.h` 中定义 `vmprint` 的原型，以便可以从 `exec.c` 中调用它。
- 在的 `printf` 调用中使用 `%p` 打印出完整的 64 位十六进制 PTE 和地址，如示例中所示。

示例：

```
1 xv6 kernel is booting
2
3 hart 2 starting
4 hart 1 starting
5 init: starting sh
```

```
6 page table 0x0000000087f6e000
7 ..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
8 .. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
9 .. ...0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
10 .. ...1: pte 0x0000000021fda00f pa 0x0000000087f68000
11 .. ...2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
12 ..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
13 .. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
14 .. ...510: pte 0x0000000021fdd807 pa 0x0000000087f76000
15 .. ...511: pte 0x0000000020001c0b pa 0x0000000080007000
16
```

vmprint 函数：

```
1 # kernel/excu.c:119 return argc; 前插入
2 if(p->pid==1) vmprint(p->pagetable);
3
4 # kernel/vm.c
5 void _vmprint(pagetable_t pagetable, int level)
6 {
7     for(int i = 0; i < 512; i++){
8         pte_t pte = pagetable[i];
9         if((pte & PTE_V)){
10             // this PTE points to a lower-level page table.
11             for (int j = 0; j < level; j++)
12             {
13                 if (j == 0) printf(".. ");
14                 else printf(" .. ");
15             }
16             uint64 child = PTE2PA(pte);
17             printf("%d: pte %p pa %p\n", i, pte, child);
18             // 查看flag位是否被设置，若被设置则为最低一层，
19             // 见vm.c161行，可以看到只有最底层被设置了符号位
20             if ((pte & (PTE_R|PTE_W|PTE_X)) == 0)
21                 _vmprint((pagetable_t)child, level + 1);
22         }
23     }
24 }
25
26 void vmprint(pagetable_t pagetable)
27 {
28     printf("page table %p\n", pagetable);
29     _vmprint(pagetable, 1);
30 }
```

vmprint 运行结果：

```

jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6-labs-2021$ make qemu
riscv64-unknown-elf-gcc -Wall -O0 -fno-omit-frame-pointer -ggdb -DSOL_PGTBL -DLAB_PGTBL -MD -mcmode=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/exec.o kernel/exec.c
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.lds -o kernel/kernel kernel/kernel/entry.o kernel/kalloc.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleep.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/spinlock.o
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6f000
..0: pte 0x0000000021fdac01 pa 0x0000000087f6b000
... ..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
... ... ..0: pte 0x0000000021fdb01f pa 0x0000000087f6c000
... ... ..1: pte 0x0000000021fda40f pa 0x0000000087f69000
... ... ..2: pte 0x0000000021fdb01f pa 0x0000000087f68000
..255: pte 0x0000000021fdb801 pa 0x0000000087f6e000
... ..511: pte 0x0000000021fdb401 pa 0x0000000087f6d000
... ... .510: pte 0x0000000021fddc07 pa 0x0000000087f77000
... ... ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$ 

```

Assignment 2 —— A kernel page table per process

xv6 拥有一个单独的内核页表供所有进入内核的进程使用，该页表与实际物理内存直接映射，不需要转换地址。但如果想使用用户态的一个地址时，当内核态需要使用一个用户态指针时，需要翻译转换虚拟地址为物理地址。因此，本实验的目标是给每个进程创建一个内核态页表。第一项工作是修改内核，以便每个进程在内核中执行时都使用自己的内核页表副本。修改 `struct proc` 为每个进程维护一个内核页表，并修改调度器在切换进程时切换内核页表。对于这一步，每个进程的内核页表应该与现有的全局内核页表相同。如果 `usertests` 运行正确，就通过了这部分的实验。

Hints:

- 为进程的内核页表添加一个字段到 `struct proc`。

```

1 # kernel/proc.h
2 struct proc {
3     struct spinlock lock;
4     .....
5     pagetable_t kernelpt;           // 内核映射表
6 };

```

- 为新进程生成内核页表的合理方法是实现 `kvminit` 的修改版本，该版本生成新页表而不是修改 `kernel_pagetable`。需要从 `allocproc` 调用此函数。

```

1 # kernel/vm.c
2 // 模仿vm.c中kvminit的方式构建每个进程自己的内核映射表 TODO:删除
3 pagetable_t proc_kpt_init()
4 {
5     pagetable_t kpt;
6     kpt = uvmcreate();
7     if (kpt == 0) return 0;
8     uvmmmap(kpt, UART0, UART0, PGSIZE, PTE_R | PTE_W);
9     uvmmmap(kpt, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
10    uvmmmap(kpt, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
11    uvmmmap(kpt, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
12    uvmmmap(kpt, KERNBASE, KERNBASE, (uint64)etext-KERNBASE,
13              PTE_R | PTE_X);
14    uvmmmap(kpt, (uint64)etext, (uint64)etext, PHYSTOP-
15              (uint64)etext, PTE_R | PTE_W);
16    uvmmmap(kpt, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R |
17              PTE_X);
18    return kpt;
19 }
20
21 // 添加映射到用户进程的kernel pagetable
22 void uvmmmap(pagetable_t pagetable, uint64 va, uint64 pa,
23               uint64 sz, int perm)
24 {
25     if(mappages(pagetable, va, sz, pa, perm) != 0)
26         panic("kvmmmap");
27 }
28
29 # kernel/proc.c/allocproc allocproc调用, 仿照创建空表
30 // 添加kernel pagetable
31 p->kernelpt = proc_kpt_init();
32 if (p->kernelpt == 0){
33     freeproc(p);
34     release(&p->lock);
35     return 0;
36 }
```

- 确保每个进程的内核页表都有该进程的内核堆栈的映射。在未修改的 xv6 中，所有内核堆栈都设置在 `procinit` 中。需要将部分或全部功能移至 `allocproc`。

```

1 # kernel/proc.c/allocproc
2 // 把内核映射放到到进程的内核栈里
3 // Allocate a page for the process's kernel stack.
4 // Map it high in memory, followed by an invalid
5 // guard page.
6 char *pa = kalloc();
7 if(pa == 0)
8     panic("kalloc");
9 uint64 va = KSTACK((int) (p - proc));
10 // 添加kernel stack的映射到用户的kernel pagetable里
11 uvmmmap(p->kernelpt, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
12 p->kstack = va;

```

- 修改 `scheduler()` 以将进程的内核页表加载到内核的 `satp` 寄存器中（请参阅 `kvminithart` 以获得灵感）。不要忘记在调用 `w_satp()` 之后调用 `sfence_vma()`。
- `scheduler()` 应该在没有进程运行时 使用 `kernel_pagetable` 。

```

1 # kernel/proc.c
2 void
3 scheduler(void)
4 {
5     struct proc *p;
6     struct cpu *c = mycpu();
7
8     c->proc = 0;
9     for(;;){
10         // Avoid deadlock by ensuring that devices can interrupt.
11         intr_on();
12
13         int found = 0;
14         for(p = proc; p < &proc[NPROC]; p++){
15             acquire(&p->lock);
16             if(p->state == RUNNABLE) {
17                 // Switch to chosen process. It is the process's job
18                 // to release its lock and then reacquire it
19                 // before jumping back to us.
20                 p->state = RUNNING;
21                 c->proc = p;
22                 // 将当前进程的kernel page存入satp寄存器中
23                 w_satp(MAKE_SATP(p->kernelpt));
24                 sfence_vma();
25                 swtch(&c->context, &p->context);
26
27                 // Process is done running for now.
28                 // It should have changed its p->state before coming
back.

```

```

29         c->proc = 0;
30         found = 1;
31     }
32     release(&p->lock);
33 }
34 #if !defined (LAB_FS)
35 if(found == 0) {
36     intr_on();
37     // 没有进程在运行则使用内核原来的kernel pagetable
38     w_satp(MAKE_SATP(kernel_pagetable));
39     sfence_vma();
40     asm volatile("wfi");
41 }
42 #else
43 ;
44 #endif
45 }
46 }

```

- 在freeproc中释放进程的内核页表。

```

1 # kernel/proc.c
2 static void
3 freeproc(struct proc *p)
4 {
5     if(p->trapframe)
6         kfree((void*)p->trapframe);
7     p->trapframe = 0;
8     // 删除kernel stack
9     if (p->kstack)
10    {
11        pte_t* pte = walk(p->kernelpt, p->kstack, 0);
12        if (pte == 0)
13            panic("freeproc: kstack");
14        kfree((void*)PTE2PA(*pte));
15    }
16    p->kstack = 0;
17    .....
18 }

```

- vmprint 在调试页表时可能会派上用场。
- 修改xv6功能或添加新功能都可以；可能至少需要在 kernel/vm.c 和 kernel/proc.c 中执行此操作。（但是，不要修改 kernel/vmcopyin.c、kernel/stats.c、user/usertests.c 和 user/stats.c。）

- 缺少页表映射可能会导致内核遇到页面错误。它将打印一个包含 `sepc=0x00000000XXXXXXXX` 的错误。可以通过在 `kernel/kernel.asm` 中搜索 ```XXXXXXXX` 来找出故障发生的位置。

Assignment 3 —— Simplify copyin/copyinstr

将 `kernel/vm.c` 中的 `copyin` 主体替换为对 `copyin_new` 的调用（在 `kernel/vmcopyin.c` 中定义）；对 `copyinstr` 和 `copyinstr_new` 执行相同的操作。将用户地址的映射添加到每个进程的内核页表，以便 `copyin_new` 和 `copyinstr_new` 工作。如果 `usertests` 运行正确并且所有的 `make Grade` 测试都通过了，那么你就通过了这个作业。

Hints:

- 先将 `copyin()` 替换为对 `copyin_new` 的调用，并使其工作，然后再转到 `copyinstr`。
- 在内核更改进程的用户映射的每一点，都以相同的方式更改进程的内核页表。这些点包括 `fork()`、`exec()` 和 `sbrk()`。
- 不要忘记在 `userinit` 的内核页表中包含第一个进程的用户页表。
- 用户地址的 PTE 在进程的内核页表中需要什么权限？（在内核模式下无法访问设置了 `PTE_U` 的页面。）
- 不要忘记上述 PLIC 限制。

```
1 # kernel/vm.c
2 //先仿照uvmcopy创建一个从用户态页表复制到内核态页表的函数
3 void
4 uvm_user2ker_copy(pagetable_t u, pagetable_t k, uint64 start,
5 uint64 end)
6 {
7     pte_t *user;
8     pte_t *kernel;
9     for(uint64 i = start; i < end; i += PGSIZE)
10    {
11        user = walk(u, i, 0);
12        kernel = walk(k, i, 1);
13    /*
14        根据内核态页表的特点--直接映射到物理内存
15        我们无需使用mappage建立映射
16        记得消除PTE_U标志位
17    */
18        *kernel = (*user) & (~PTE_U);
19    }
20}
21}
```

```
22
23
24
25 # kernel/proc.c
26 void
27 userinit(void)
28 {
29     struct proc *p;
30     .....
31     uvminit(p→pagetable, initcode, sizeof(initcode));
32     p→sz = PGSIZE;
33
34     uvm_user2ker_copy(p→pagetable, p→kernel_pagetable, 0, p→sz);
35     .....
36 }
37
38
39 int
40 fork(void)
41 {
42     int i, pid;
43     .....
44     np→sz = p→sz;
45     uvm_user2ker_copy(np→pagetable, np→kernel_pagetable, 0, np-
46 >sz);
47     .....
48 }
49
50
51 # kernel/exec.c
52 int
53 exec(char *path, char **argv)
54 {
55     char *s, *last;
56     .....
57     // Load program into memory.
58     for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
59         .....
60         uint64 sz1;
61         if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
62             goto bad;
63     /*
64      判断是否越界
65     */
66     if(sz1 > PLIC)
67         goto bad;
68     .....
```

```

69 }
70 .....
71 uvm_user2ker_copy(p→pagetable, p→kernel_pagetable, 0, sz);
72
73 if(p→pid == 1)
74     vmprint(p→pagetable);
75
76 return argc; // this ends up in a0, the first argument to
77 main(argc, argv)
78 .....
79
80
81 # kernel/syproc.c
82 uint64
83 sys_sbrk(void)
84 {
85 .....
86 if(growproc(n) < 0)
87     return -1;
88 if(n > 0)
89 {
90     uvm_user2ker_copy(myproc()→pagetable, myproc()-
91 >kernel_pagetable, addr, addr + n);
92 }
93 //考虑到内核页表内容是根据用户页表改变，所以只增加/覆盖内容，不删除内容（我
94 猜可行
95
96 return addr;
}

```

Lab 4 Traps

Assignment 2 —— RISC-V assembly

1 Q: 哪些寄存器存储了函数调用的参数？举个例子，main 调用 printf 的时候，13 被存在了哪个寄存器中？

2 A: a0-a7; a2;

3

4 Q: main 中调用函数 f 对应的汇编代码在哪？对 g 的调用呢？（提示：编译器有可能会内联(inline)一些函数）

5 A: 没有这样的代码。g(x) 被内联到 f(x) 中，然后 f(x) 又被进一步内联到 main() 中

6

```
7 Q: printf 函数所在的地址是?  
8 A: 0x00000000000000630, main 中使用 pc 相对寻址来计算得到这个地址。  
9  
10 Q: 在 main 中 jalr 跳转到 printf 之后, ra 的值是什么?  
11 A: 0x0000000000000040, jalr 指令的下一条汇编指令的地址。  
12  
13 Q: 运行下面的代码  
14  
15     unsigned int i = 0x00646c72;  
16     printf("H%x Wo%s", 57616, &i);  
17  
18 输出是什么?  
19 如果 RISC-V 是大端序的, 要实现同样的效果, 需要将 i 设置为什么? 需要将 57616  
修改为别的值吗?  
20 A: "He110 WorlD"; 0x726c6400; 不需要, 57616 的十六进制是 110, 无论端序  
(十六进制和内存中的表示不是同个概念)  
21  
22 Q: 在下面的代码中, 'y=' 之后会答应什么? (note: 答案不是一个具体的值) 为  
什么?  
23  
24     printf("x=%d y=%d", 3);  
25  
26 A: 输出的是一个受调用前的代码影响的“随机”的值。因为 printf 尝试读的参数数量  
比提供的参数数量多。  
27 第二个参数 `3` 通过 a1 传递, 而第三个参数对应的寄存器 a2 在调用前不会被设置  
为任何具体的值, 而是会  
28 包含调用发生前的任何已经在里面的值。
```

Assignment 2 —— Back trace

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

```
1 backtrace:  
2 0x0000000080002cda  
3 0x0000000080002bb6  
4 0x0000000080002898
```

bttest 后退出 qemu。在终端中: 地址可能略有不同, 但如果运行 addr2line -e kernel/kernel (或 riscv64-unknown-elf-addr2line -e kernel/kernel) 并剪切并粘贴上述地址, 如下所示:

```
1 $ addr2line -e kernel/kernel
2 0x0000000080002de2
3 0x0000000080002f4a
4 0x0000000080002bfc
5 Ctrl-D
```

应该可以看到如下内容：

```
1 kernel/sysproc.c:74
2 kernel/syscall.c:224
3 kernel/trap.c:85
```

Hints:

- 将回溯的原型添加到 `kernel/defs.h` 以便您可以在 `sys_sleep` 中调用回溯。

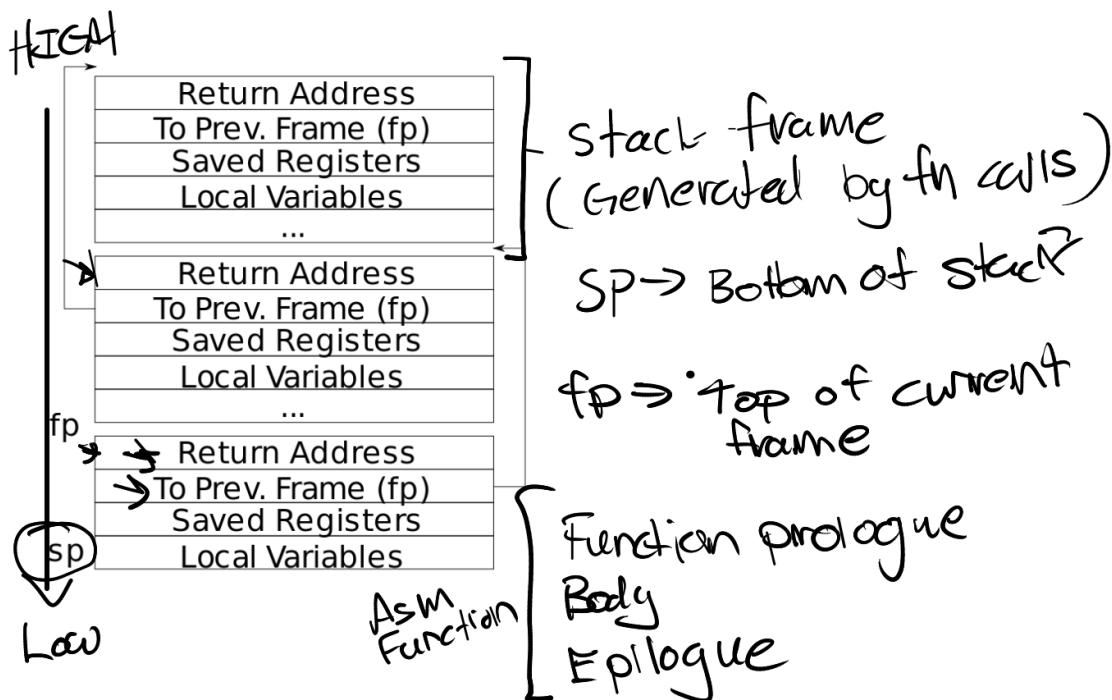
```
1 // kernel/printf.c
2 ...
3 void         printfinit(void);
4 void         backtrace(void);
5
6 // kernel/sysproc.c
7
8 uint64 sys_sleep(void)
9 {
10    int n;
11    uint ticks0;
12
13    backtrace(); // print stack backtrace.
14    ...
15 }
```

- GCC 编译器将当前执行函数的帧指针存储在寄存器 `s0` 中。将以下函数添加到 `kernel/riscv.h`：

```
1 // 获取上一级栈帧地址
2 static inline uint64
3 r_fp()
4 {
5    uint64 x;
6    asm volatile("mv %0, s0" : "=r" (x) );
7    return x;
8 }
```

并在 `backtrace` 中调用此函数以读取当前帧指针。此函数使用内联汇编来读取 `s0`。

- 堆栈帧布局，请注意，返回地址位于距堆栈帧的帧指针的固定偏移量 (-8) 处，而保存的帧指针位于距帧指针的固定偏移量 (-16) 处。



fp 指向当前栈帧的开始地址，sp 指向当前栈帧的结束地址。（栈从高地址往低地址生长，所以 fp 虽然是帧开始地址，但是地址比 sp 高）。栈帧中从高到低第一个 8 字节 fp-8 是 return address，也就是当前调用层应该返回到的地址。栈帧中从高到低第二个 8 字节 fp-16 是 previous address，指向上一层栈帧的 fp 开始地址。剩下的为保存的寄存器、局部变量等。一个栈帧的大小不固定，但是至少 16 字节。在 xv6 中，使用一个页来存储栈，如果 fp 已经到达栈页的上界，则说明已经到达栈底。

- xv6 为 xv6 内核中的每个堆栈在 PAGE 对齐地址处分配一个页面。可以使用 PGROUNDDOWN(fp) 和 PGROUNDUP(fp) 计算堆栈页面的顶部和底部地址（请参阅 kernel/riscv.h）。这些数字有助于回溯终止其循环。

backtrace 函数：

```

1 // kernel/printf.c
2 void backtrace() {
3     uint64 fp = r_fp();
4     while(fp != PGROUNDUP(fp)) { // 如果已经到达栈底
5         uint64 ra = *(uint64*)(fp - 8); // return address
6         printf("%p\n", ra);
7         fp = *(uint64*)(fp - 16); // previous fp
8     }
9 }
```

实验运行结果：

```
$ bttest
0x00000000800020cc
0x0000000080001fa6
0x0000000080001c90
$ QEMU: Terminated
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-labs-2021$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ bttest
0x00000000800020cc
0x0000000080001fa6
0x0000000080001c90
$ QEMU: Terminated
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-labs-2021$ addr2line -e kernel/kernel
0x00000000800020cc
0x0000000080001fa6
0x0000000080001c90
/home/jinnian/Desktop/OS/xv6-labs-2021/kernel/sysproc.c:63
/home/jinnian/Desktop/OS/xv6-labs-2021/kernel/syscall.c:140
/home/jinnian/Desktop/OS/xv6-labs-2021/kernel/trap.c:76
```

Assignment 3 —— Alarm

在本任务中，向 xv6 添加一个功能，该功能会在进程使用 CPU 时间时定期提醒它。这对于想要限制它们占用多少 CPU 时间的计算绑定进程，或者对于想要计算但又想要采取一些定期操作的进程可能很有用。更一般地说，您将实现一种原始形式的用户级中断/故障处理程序；例如，您可以使用类似的东西来处理应用程序中的页面错误。如果它通过了警报测试和用户测试，则您的解决方案是正确的。

添加一个新的 `sigalarm(interval, handler)` 系统调用。如果应用程序调用 `sigalarm(n, fn)`，那么在程序消耗的 每 n 个CPU 时间之后，内核应该调用应用程序函数 `fn`。当 `fn` 返回时，应用程序应该从中断的地方继续。在 xv6 中，`tick` 是一个相当随意的时间单位，由硬件定时器产生中断的频率决定。如果应用程序调用 `sigalarm(0, 0)`，内核应该停止生成定期警报调用。在 xv6 存储库中找到文件 `user/alarmtest.c`，将其添加到 Makefile。在添加 `sigalarm` 和 `sigreturn` 系统调用（见下文）之前，它不会正确编译。`alarmtest` 在 `test0` 中调用 `sigalarm(2,periodic)` 以要求内核每 2 个滴答声强制调用一次 `periodic()`，然后旋转一段时间。可以在 `user/alarmtest.asm` 中看到 `alarmtest` 的汇编代码，这对于调试可能很方便。当 `alarmtest`` 产生这样的输出并且 `usertests` 也正确运行时，解决方案是正确的。

test 0 : 调用中断程序：

- 添加调用：在 `user.h` 声明调用，在 `syscall.h` 添加系统调用号等等

```

1 # kernel/user.h
2 int sigalarm(int ticks, void (*handler)());
3 int sigreturn(void);
4
5 # kernel/syscall.h
6 #define SYS_sigalarm 22
7 #define SYS_sigreturn 23

```

- 将警报间隔和指向处理函数的指针存储在 `proc` 结构的新字段中（在 `kernel/proc.h` 中）

```

1 # kernel/proc.h
2 struct proc {
3     // .....
4     int alarm_interval;           // 时钟周期, 0 为禁用
5     void(*alarm_handler)();       // 时钟回调处理函数
6     int alarm_ticks;             // 下一次时钟响起前还剩下的 ticks 数
7     struct trapframe *alarm_trapframe; // 时钟中断时刻的
8     trapframe, 用于中断处理完成后恢复原程序的正常执行
9     int alarm_goingoff;          // 是否已经有一个时钟回调正在执行且还未
10    return (用于防止在 alarm_handler 中途闹钟到期再次调用 alarm_handler,
11    导致 alarm_trapframe 被覆盖)
12 };

```

- `sigalarm` 和 `sigreturn` 实现：🌟不要忘记 `defs.h` 添加函数声明

```

1 # kernel/sysproc.c
2 uint64 sys_sigalarm(void) {
3     int n;
4     uint64 fn;
5     if(argint(0, &n) < 0)
6         return -1;
7     if(argaddr(1, &fn) < 0)
8         return -1;
9
10    return sigalarm(n, (void(*)())(fn));
11 }
12
13 uint64 sys_sigreturn(void) {
14     return sigreturn();
15 }
16
17 // kernel/trap.c
18 int sigalarm(int ticks, void(*handler)()) {
19     // 设置 myproc 中的相关属性
20     struct proc *p = myproc();
21     p->alarm_interval = ticks;

```

```

22     p->alarm_handler = handler;
23     p->alarm_ticks = ticks;
24     return 0;
25 }
26
27 int sigreturn() {
28     // 将 trapframe 恢复到时钟中断之前的状态，恢复原本正在执行的程序流
29     struct proc *p = myproc();
30     *p->trapframe = *p->alarm_trapframe;
31     p->alarm_goingoff = 0;
32     return 0;
33 }
```

- 仿照 allocproc 中分配中断帧的方式分配 alarm 中断帧和初始化

```

1 # kernel/proc.c/allocproc:130
2 // Allocate a trapframe page for alarm_trapframe.
3 if((p->alarm_trapframe = (struct trapframe *)kalloc()) == 0)
{
4     release(&p->lock);
5     return 0;
6 }
7
8 p->alarm_interval = 0;
9 p->alarm_handler = 0;
10 p->alarm_ticks = 0;
11 p->alarm_goingoff = 0;
```

- 释放代码：

```

1 # kernel/proc.c/freeproc:169
2 if(p->alarm_trapframe)
3     kfree((void*)p->alarm_trapframe);
4 p->alarm_trapframe = 0;
5
6 // .....
7
8 p->alarm_interval = 0;
9 p->alarm_handler = 0;
10 p->alarm_ticks = 0;
11 p->alarm_goingoff = 0;
12
```

- 在 usertrap 函数中，实现时钟机制具体代码：

```

1 # kernel/trap.c/usertrap:80
2 ...
```

```

3  if(p→killed)
4      exit(-1);
5
6 → begin:80
7 // give up the CPU if this is a timer interrupt.
8 // if(which_dev == 2)
9 //     yield();
10
11 // give up the CPU if this is a timer interrupt.
12 if(which_dev == 2) {
13     if(p→alarm_interval != 0) { // 如果设定了时钟事件
14         if(--p→alarm_ticks <= 0) { // 时钟倒计时 -1 tick, 如果已经
到达或超过设定的 tick 数
15             if(!p→alarm_goingoff) { // 确保没有时钟正在运行
16                 p→alarm_ticks = p→alarm_interval;
17                 // jump to execute alarm_handler
18                 *p→alarm_trapframe = *p→trapframe; // backup
trapframe
19                 p→trapframe→epc = (uint64)p→alarm_handler;
20                 p→alarm_goingoff = 1;
21             }
22             // 如果一个时钟到期的时候已经有一个时钟处理函数正在运行，则会推
迟到原处理函数运行完成后的下一个 tick 才触发这次时钟
23         }
24     }
25     yield();
26 }
27
28 usertrapret();

```

在每次时钟中断的时候，如果进程有已经设置的时钟 (`alarm_interval != 0`)，则进行 `alarm_ticks` 倒数。当 `alarm_ticks` 倒数到小于等于 0 的时候，如果没有正在处理的时钟，则尝试触发时钟，将原本的程序流保存起来 (`*alarm_trapframe = *trapframe`)，然后通过修改 `pc` 寄存器的值，将程序流转跳到 `alarm_handler` 中，`alarm_handler` 执行完毕后再恢复原本的执行流 (`*trapframe = *alarm_trapframe`)。这样从原本程序执行流的视角，就是不可感知的中断了。

实验运行结果：

测试结果：

```
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.lds -o kernel/kernel kernel/kernel/entry.o kernel/ka  
lloc.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/r  
ap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/fi  
le.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o ke  
rnel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/spinlock.o  
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm  
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /\$/d' > kernel/kernel.s  
ym  
make[1]: Leaving directory '/home/jinnian/Desktop/OS/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  
(3.0s)  
== 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 (111.9s)  
== Test time ==  
time: FAIL  
    Cannot read time.txt  
Score: 84/85  
make: *** [Makefile:337: grade] Error 1  
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-labs-2021$
```

Lab 5 xv6 lazy page allocation

操作系统可以使用页表硬件的技巧之一是延迟分配用户空间堆内存（lazy allocation of user-space heap memory）。Xv6应用程序使用 `sbrk()` 系统调用向内核请求堆内存。在内核中，`sbrk()` 分配物理内存并将其映射到进程的虚拟地址空间。内核为一个大请求分配和映射内存可能需要很长时间。例如，考虑由 262144 个 4096 字节的页组成的千兆字节；即使单独一个页面的分配开销很低，但合起来如此大的分配数量将不可忽视。此外，有些程序申请分配的内存比实际

使用的要多（例如，实现稀疏数组），或者为了以后的不时之需而分配内存。为了让 `sbrk()` 在这些情况下更快地完成，复杂的内核会延迟分配用户内存。也就是说，`sbrk()` 不分配物理内存，只是记住分配了哪些用户地址，并在用户页表中将这些地址标记为无效。当进程第一次尝试使用延迟分配中给定的页面时，CPU生成一个页面错误（page fault），内核通过分配物理内存、置零并添加映射来处理该错误。

Assignment 1 —— Eliminate allocation from `sbrk()`

该任务要求删除 `sbrk(n)` 系统调用中的页面分配代码（位于 `sysproc.c` 中的函数 `sys_sbrk()`）。`sbrk(n)` 系统调用将进程的内存大小增加 `n` 个字节，然后返回新分配区域的开始部分（即旧的大小）。新的 `sbrk(n)` 应该只将进程的大小 (`myproc()->sz`) 增加 `n`，然后返回旧的大小。它不应该分配内存——因此应该删除对 `growproc()` 的调用（但是仍然需要增加进程的大小！）。

在 `sbrk()` 时只增长进程的 `myproc()->sz` 而不实际分配内存：

```
1 # kernel/sysproc.c
2 uint64 sys_sbrk(void)
3 {
4     int addr;
5     int n;
6     if(argint(0, &n) < 0)
7         return -1;
8     addr = myproc()→sz;
9     myproc()→sz += n;
10    // if(growproc(n) < 0)
11    //     return -1;
12    return addr;
13 }
```

1 | 运行结果：

```
1 init: starting sh
2 $ echo hi
3 usertrap(): unexpected scause 0x000000000000000f pid=3
4             sepc=0x0000000000001258 stval=0x0000000000004008
5 va=0x0000000000004000 pte=0x0000000000000000
6 panic: uvmunmap: not mapped
```

Assignment 2 —— Lazy allocation

进程第一次尝试使用延迟分配中给定的页面时，CPU生成一个页面错误 page fault

该任务修改 trap.c 中的代码以响应来自用户空间的页面错误，方法是新分配一个物理页面并映射到发生错误的地址，然后返回到用户空间，让进程继续执行。在生成 “usertrap(): ...” 消息的 printf 调用之前添加代码，修改任何其他 xv6 内核代码，以使 echo hi 正常工作。13 为 page load fault，15 为 page write fault

Hints:

- 在 usertrap() 中查看 r_scause() 的返回值是否为 13 或 15 来判断该错误是否为页面错误
 - 13 为 page load fault，15 为 page write fault
- stval 寄存器中保存了造成页面错误的虚拟地址，可以通过 r_stval() 读取
- 参考 vm.c 中的 uvmalloc() 中的代码，那是一个 sbrk() 通过 growproc() 调用的函数。你将需要对 kalloc() 和 mappages() 进行调用
- 使用 PGROUNDDOWN(va) 将出错的虚拟地址向下舍入到页面边界
- 当前 uvmunmap() 会导致系统 panic 崩溃；请修改程序保证正常运行

```
1 # kernel/vm.c/uvmunmap
2 if((pte = walk(pagetable, a, 0)) == 0)
3     continue;
4 // panic("uvmunmap: walk");
5 if(*pte & PTE_V) == 0
6     continue;
7 // panic("uvmunmap: not mapped");
8 if(PTE_FLAGS(*pte) == PTE_V)
9     panic("uvmunmap: not a leaf");
10 if(do_free){
```

- 如果内核崩溃，请在 kernel/kernel.asm 中查看 sepc
- 使用 pgtbl lab 的 vmprint 函数打印页表的内容
- 如果看到错误“incomplete type proc”，请 include“spinlock.h”然后是“proc.h”。

```
1 # kernel/trap.c/usertrap:
2 } else if((which_dev = devintr()) != 0){
3     // ok
4 } else {
5     if (r_scause() == 13 || r_scause() == 15) {
6         // page fault
7         uint64 va = r_stval();
8         char *mem;
```

```

9     va = PGROUNDDOWN(va);
10    if ((mem = kalloc()) == 0) {
11        panic("cannot allocate for lazy alloc\n");
12        exit(-1);
13    }
14    if (mappages(p→pagetable, va, PGSIZE, (uint64)mem,
15      PTE_W|PTE_X|PTE_R|PTE_U) != 0) {
16        kfree(mem);
17        panic("cannot map for lazy alloc\n");
18        exit(-1);
19    }
20  } else {
21      printf("usertrap(): unexpected scause %p pid=%d\n",
22      r_scause(), p→pid);
23      printf("          sepc=%p stval=%p\n", r_sepc(),
24      r_stval());
25      p→killed = 1;
26  }
27 }
```

Assignment 3 —— Lazytests and Usertests (moderate)

Hints:

- 处理 `sbrk()` 参数为负的情况。

```

1  if (n < 0) {
2      // deallocate the memory
3      if ((myproc()→sz + n) < 0) {
4          return -1;
5      } else {
6          if (uvmdealloc(myproc()→pagetable, addr, addr+n)
7              != (addr+n)) {
8              return -1;
9          }
10 }
```

- 如果某个进程在高于 `sbrk()` 分配的任何虚拟内存地址上出现页错误，则终止该进程 & 处理用户栈下面的无效页面上发生的错误。

当造成的page fault在进程的user stack以下（栈底）或者在 `p->sz` 以上（堆顶）时，kill这个进程。在kernel/trap.c的 `usertrap` 中增加以下判断条件

```
1 | if ((va < p→sz) && (va > PGROUNDDOWN(p→trapframe→sp)))
```

- 在 fork() 中正确处理父到子内存拷贝。

fork() 中将父进程的内存复制给子进程的过程中用到了 uvmcopy，uvmcopy 原本在发现缺失相应的PTE等情况下会panic，这里也要 continue 掉。在 kernel/proc.c 的 uvmcopy 中

```
1 | # kernel/proc.c/uvmcopy
2 | if((pte = walk(old, i, 0)) == 0)
3 |     continue;
4 | // panic("uvmcopy: pte should exist");
5 | if(*pte & PTE_V) == 0)
6 |     continue;
7 | // panic("uvmcopy: page not present");
8 |
```

- 处理这种情形：进程从 sbrk() 向系统调用（如 read 或 write）传递有效地址，但尚未分配该地址的内存。
- 正确处理内存不足：如果在页面错误处理程序中执行 kalloc() 失败，则终止当前进程。

```
1 | # kernel/vm.c/excu:
2 | if((pte == 0) || ((*pte & PTE_V) == 0)) {
3 |     if (va > myproc()→sz || va < PGROUNDDOWN(myproc()-
4 | >trapframe→sp)) {
5 |         return 0;
6 |     }
7 |     if ((mem = (uint64)kalloc()) == 0) return 0;
8 |     va = PGROUNDDOWN(va);
9 |     if (mappages(myproc()→pagetable, va, PGSIZE, mem,
10 | PTE_W|PTE_X|PTE_R|PTE_U) != 0) {
11 |         kfree((void*)mem);
12 |         return 0;
13 |     }
14 |     return mem;
15 | }
```

运行结果及评分：

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6/MIT_6S081
```

```
== Test lazy: unmap ==
lazy: unmap: OK
== Test usertests ==
$ make qemu-gdb
(80.7s)
== Test usertests: pbug ==
usertests: pbug: OK
== Test usertests: sbrkbugs ==
usertests: sbrkbugs: OK
== Test usertests: argptest ==
usertests: argptest: OK
== Test usertests: sbrkmuch ==
usertests: sbrkmuch: OK
== Test usertests: sbrkfail ==
usertests: sbrkfail: OK
== Test usertests: sbrkarg ==
usertests: sbrkarg: OK
== Test usertests: stacktest ==
usertests: stacktest: OK
== Test usertests: execout ==
usertests: execout: OK
== Test usertests: copyin ==
usertests: copyin: OK
== Test usertests: copyout ==
usertests: copyout: OK
== Test usertests: copyinstr1 ==
usertests: copyinstr1: OK
== Test usertests: copyinstr2 ==
usertests: copyinstr2: OK
== Test usertests: copyinstr3 ==
usertests: copyinstr3: OK
== Test usertests: rwsbrk ==
usertests: rwsbrk: OK
```

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6/MIT_6S081
```

```
== Test lazy: unmap ==
lazy: unmap: OK
== Test usertests ==
$ make qemu-gdb
(80.7s)
== Test usertests: pbug ==
usertests: pbug: OK
== Test usertests: sbrkbugs ==
usertests: sbrkbugs: OK
== Test usertests: argptest ==
usertests: argptest: OK
== Test usertests: sbrkmuch ==
usertests: sbrkmuch: OK
== Test usertests: sbrkfail ==
usertests: sbrkfail: OK
== Test usertests: sbrkarg ==
usertests: sbrkarg: OK
== Test usertests: stacktest ==
usertests: stacktest: OK
== Test usertests: execout ==
usertests: execout: OK
== Test usertests: copyin ==
usertests: copyin: OK
== Test usertests: copyout ==
usertests: copyout: OK
== Test usertests: copyinstr1 ==
usertests: copyinstr1: OK
== Test usertests: copyinstr2 ==
usertests: copyinstr2: OK
== Test usertests: copyinstr3 ==
usertests: copyinstr3: OK
== Test usertests: rwsbrk ==
usertests: rwsbrk: OK
```

Lab 5 xv6 lazy page allocation

操作系统可以使用页表硬件的技巧之一是延迟分配用户空间堆内存（lazy allocation of user-space heap memory）。Xv6应用程序使用 `sbrk()` 系统调用向内核请求堆内存。在内核中，`sbrk()` 分配物理内存并将其映射到进程的虚拟地址空间。内核为一个大请求分配和映射内存可能需要很长时间。例如，考虑由262144个4096字节的页组成的千兆字节；即使单独一个页面的分配开销很低，但合起来如此大的分配数量

将不可忽视。此外，有些程序申请分配的内存比实际使用的要多（例如，实现稀疏数组），或者为了以后的不时之需而分配内存。为了让 `sbrk()` 在这些情况下更快地完成，复杂的内核会延迟分配用户内存。也就是说，`sbrk()` 不分配物理内存，只是记住分配了哪些用户地址，并在用户页表中将这些地址标记为无效。当进程第一次尝试使用延迟分配中给定的页面时，CPU生成一个页面错误（page fault），内核通过分配物理内存、置零并添加映射来处理该错误。

Assignment 1 —— Eliminate allocation from `sbrk()`

该任务要求删除 `sbrk(n)` 系统调用中的页面分配代码（位于 `sysproc.c` 中的函数 `sys_sbrk()`）。`sbrk(n)` 系统调用将进程的内存大小增加 `n` 个字节，然后返回新分配区域的开始部分（即旧的大小）。新的 `sbrk(n)` 应该只将进程的大小（`myproc()>sz`）增加 `n`，然后返回旧的大小。它不应该分配内存——因此应该删除对 `growproc()` 的调用（但是仍然需要增加进程的大小！）。

在 `sbrk()` 时只增长进程的 `myproc()>sz` 而不实际分配内存：

```
1 # kernel/sysproc.c
2 uint64 sys_sbrk(void)
3 {
4     int addr;
5     int n;
6     if(argint(0, &n) < 0)
7         return -1;
8     addr = myproc()>sz;
9     myproc()>sz += n;
10    // if(growproc(n) < 0)
11    //     return -1;
12    return addr;
13 }
```

1 | 运行结果：

```
1 init: starting sh
2 $ echo hi
3 usertrap(): unexpected scause 0x000000000000000f pid=3
4             sepc=0x0000000000001258 stval=0x0000000000004008
5 va=0x0000000000004000 pte=0x0000000000000000
6 panic: uvmunmap: not mapped
```

Assignment 2 —— Lazy allocation

进程第一次尝试使用延迟分配中给定的页面时，CPU生成一个页面错误 page fault

该任务修改**trap.c**中的代码以响应来自用户空间的页面错误，方法是新分配一个物理页面并映射到发生错误的地址，然后返回到用户空间，让进程继续执行。在生成“`usertrap(): ...`”消息的 `printf` 调用之前添加代码，修改任何其他xv6内核代码，以使 `echo hi` 正常工作。13为page load fault，15为page write fault

Hints:

- 在 `usertrap()` 中查看 `r_scause()` 的返回值是否为13或15来判断该错误是否为页面错误

13为 `page load fault`，15为 `page write fault`

- `stval` 寄存器中保存了造成页面错误的虚拟地址，可以通过 `r_stval()` 读取
- 参考 **vm.c** 中的 `uvmalloc()` 中的代码，那是一个 `sbrk()` 通过 `growproc()` 调用的函数。你将需要对 `kalloc()` 和 `mappages()` 进行调用
- 使用 `PGRUNDDOWN(va)` 将出错的虚拟地址向下舍入到页面边界
- 当前 `uvmunmap()` 会导致系统 panic 崩溃；请修改程序保证正常运行

```
1 # kernel/vm.c/uvmunmap
2 if((pte = walk(pagetable, a, 0)) == 0)
3     continue;
4 // panic("uvmunmap: walk");
5 if(*pte & PTE_V) == 0)
6     continue;
7 // panic("uvmunmap: not mapped");
8 if(PTE_FLAGS(*pte) == PTE_V)
9     panic("uvmunmap: not a leaf");
10 if(do_free){
```

- 如果内核崩溃，请在 **kernel/kernel.asm** 中查看 `sepc`
- 使用 `pgtbl lab` 的 `vmprint` 函数打印页表的内容
- 如果看到错误“incomplete type proc”，请 `include "spinlock.h"` 然后是“`proc.h`”。

```
1 # kernel/trap.c/usertrap:
2 } else if((which_dev = devintr()) != 0){
3     // ok
4 } else {
5     if (r_scause() == 13 || r_scause() == 15) {
6         // page fault
7         uint64 va = r_stval();
8         char *mem;
9         va = PGRUNDDOWN(va);
```

```

10     if ((mem = kalloc()) == 0) {
11         panic("cannot allocate for lazy alloc\n");
12         exit(-1);
13     }
14     if (mappages(p→pagetable, va, PGSIZE, (uint64)mem,
15 PTE_W|PTE_X|PTE_R|PTE_U) ≠ 0) {
16         kfree(mem);
17         panic("cannot map for lazy alloc\n");
18         exit(-1);
19     }
20     else {
21         printf("usertrap(): unexpected scause %p pid=%d\n",
r_scause(), p→pid);
22         printf("          sepc=%p stval=%p\n", r_sepc(),
r_stval());
23         p→killed = 1;
24     }
25 }
```

Assignment 3 —— Lazytests and Usertests (moderate)

Hints:

- 处理 `sbrk()` 参数为负的情况。

```

1  if (n < 0) {
2      // deallocate the memory
3      if ((myproc()→sz + n) < 0) {
4          return -1;
5      } else {
6          if (uvmdealloc(myproc()→pagetable, addr, addr+n) ≠
(addr+n)) {
7              return -1;
8          }
9      }
10 }
```

- 如果某个进程在高于 `sbrk()` 分配的任何虚拟内存地址上出现页错误，则终止该进程 & 处理用户栈下面的无效页面上发生的错误。

当造成的page fault在进程的user stack以下（栈底）或者在 `p->sz` 以上（堆顶）时，kill这个进程。在kernel/trap.c的 `usertrap` 中增加以下判断条件

```
1 | if ((va < p→sz) && (va > PGROUNDDOWN(p→trapframe→sp)))
```

- 在 `fork()` 中正确处理父到子内存拷贝。

`fork()` 中将父进程的内存复制给子进程的过程中用到了 `uvmcopy`，`uvmcopy` 原本在发现缺失相应的PTE等情况下会panic，这里也要 `continue` 掉。在 `kernel/proc.c` 的 `uvmcopy` 中

```
1 # kernel/proc.c/uvmcopy
2 if((pte = walk(old, i, 0)) == 0)
3     continue;
4 // panic("uvmcopy: pte should exist");
5 if((*pte & PTE_V) == 0)
6     continue;
7 // panic("uvmcopy: page not present");
8
```

- 处理这种情形：进程从 `sbrk()` 向系统调用（如 `read` 或 `write`）传递有效地址，但尚未分配该地址的内存。
- 正确处理内存不足：如果在页面错误处理程序中执行 `kalloc()` 失败，则终止当前进程。

```
1 # kernel/vm.c/excu:
2 if((pte == 0) || ((*pte & PTE_V) == 0)) {
3     if (va > myproc()→sz || va < PGROUNDDOWN(myproc()→trapframe→sp)) {
4         return 0;
5     }
6     if ((mem = (uint64)kalloc()) == 0) return 0;
7     va = PGROUNDDOWN(va);
8     if (mappages(myproc()→pagetable, va, PGSIZE, mem,
9       PTE_W|PTE_X|PTE_R|PTE_U) ≠ 0) {
10        kfree((void*)mem);
11        return 0;
12    }
13    return mem;
14 }
```

运行结果及评分：

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6/MIT_6S081
```

```
== Test lazy: unmap ==
lazy: unmap: OK
== Test usertests ==
$ make qemu-gdb
(80.7s)
== Test usertests: pbug ==
usertests: pbug: OK
== Test usertests: sbrkbugs ==
usertests: sbrkbugs: OK
== Test usertests: argptest ==
usertests: argptest: OK
== Test usertests: sbrkmuch ==
usertests: sbrkmuch: OK
== Test usertests: sbrkfail ==
usertests: sbrkfail: OK
== Test usertests: sbrkarg ==
usertests: sbrkarg: OK
== Test usertests: stacktest ==
usertests: stacktest: OK
== Test usertests: execout ==
usertests: execout: OK
== Test usertests: copyin ==
usertests: copyin: OK
== Test usertests: copyout ==
usertests: copyout: OK
== Test usertests: copyinstr1 ==
usertests: copyinstr1: OK
== Test usertests: copyinstr2 ==
usertests: copyinstr2: OK
== Test usertests: copyinstr3 ==
usertests: copyinstr3: OK
== Test usertests: rwsbrk ==
usertests: rwsbrk: OK
```

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6/MIT_6S081
```

```
== Test lazy: unmap ==
lazy: unmap: OK
== Test usertests ==
$ make qemu-gdb
(80.7s)
== Test usertests: pbug ==
usertests: pbug: OK
== Test usertests: sbrkbugs ==
usertests: sbrkbugs: OK
== Test usertests: argptest ==
usertests: argptest: OK
== Test usertests: sbrkmuch ==
usertests: sbrkmuch: OK
== Test usertests: sbrkfail ==
usertests: sbrkfail: OK
== Test usertests: sbrkarg ==
usertests: sbrkarg: OK
== Test usertests: stacktest ==
usertests: stacktest: OK
== Test usertests: execout ==
usertests: execout: OK
== Test usertests: copyin ==
usertests: copyin: OK
== Test usertests: copyout ==
usertests: copyout: OK
== Test usertests: copyinstr1 ==
usertests: copyinstr1: OK
== Test usertests: copyinstr2 ==
usertests: copyinstr2: OK
== Test usertests: copyinstr3 ==
usertests: copyinstr3: OK
== Test usertests: rwsbrk ==
usertests: rwsbrk: OK
```

Lab 6 Copy-on-Write

Assignment 1 Implement copy-on write

- Problem:

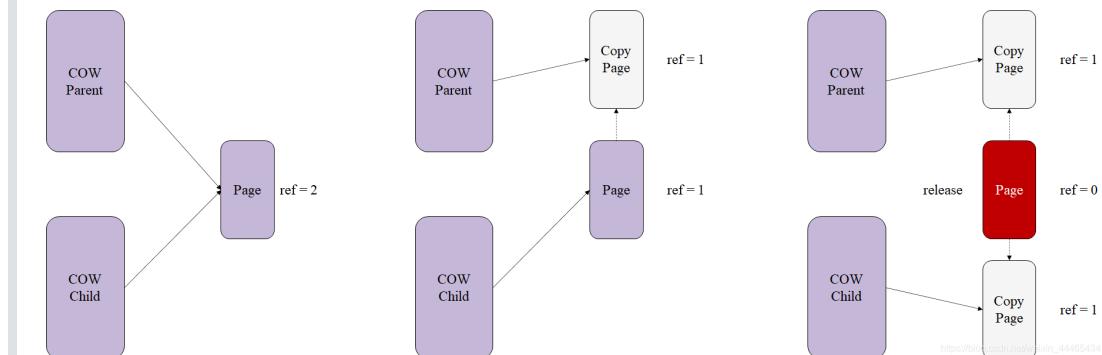
xv6 中的 `fork()` 系统调用将所有父进程的用户空间内存复制到子进程中。如果父级很大，则复制可能需要很长时间。更糟糕的是，这项工作常常被大量浪费。例如，子进程中的 `fork()` 后跟 `exec()` 将导致子进程丢弃复制的内存，可能不会使用大部分内存。另一方面，如果父母和孩子都使用一个页面，并且一个或两个都写它，那么确实需要一个副本。

- Solution:

写时复制 (cow) `fork()` 的目标是推迟为子进程分配和复制物理内存页面，直到实际需要副本（如果有的话）。`COW fork()` 只为子级创建一个页表，用户内存的 PTE 指向父级的物理页面。`COW fork()` 将 `parent` 和 `child` 中的所有用户 PTE 标记为不可写。当任一进程尝试写入这些 COW 页之一时，CPU 将强制发生页错误。内核页面错误处理程序检测到这种情况，为出错进程分配物理内存页面，将原始页面复制到新页面中，并修改出错进程中的相关 PTE 以引用新页面，这次使用 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它的页面副本。

`COW fork()` 使实现用户内存的物理页面的释放变得有点麻烦。一个给定的物理页可以被多个进程的页表引用，并且只有在最后一个引用消失时才应该被释放。

基于cow的`fork()`函数只在子进程中创建指向父进程物理页面的页表，而不创建真实的物理页面；在调用`fork()`函数后，子进程和父进程的PTE（page table entry）均被置为不可写，并且予以一个COW标记（每个 PTE，有一种方法来记录它是否是 COW 映射可能很有用。为此，可以使用 RISC-V PTE 中的 RSW（为软件保留）位），表示该PTE是属于cow的，这样，当其中一个进程要写的时候，就会在trap.c中捕捉到写错误同时发现va对应的PTE是被COW标记的，就会对原物理页进行复制操作，并修改该PTE映射的物理页为被复制的物理页。此外还需要注意一个细节：我们应该为每一块物理页面添加一个引用指针，用于记录它被进程引用的次数。当其引用次数为0的时候，我们就应该将其释放。这种情况对应着两个进程都复制了原物理页，那么原物理页就没有存在的必要了，调用`kfree`释放即可。如下：



Hints:

1. 修改 `uvmcopy()` 以将父级的物理页面映射到子级，而不是分配新页面。在子级和父级的 PTE 中使 `PTE_W`（写权限）无效。

```

1 int
2 uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
3 {

```

```

4  pte_t *pte;
5  uint64 pa, i;
6  uint flags;
7 //  char *mem;
8
9  for(i = 0; i < sz; i += PGSIZE){
10     if((pte = walk(old, i, 0)) == 0)
11         panic("uvmcopy: pte should exist");
12     if((*pte & PTE_V) == 0)
13         panic("uvmcopy: page not present");
14     pa = PTE2PA(*pte);
15     flags = PTE_FLAGS(*pte);
16     if (flags & PTE_W) {
17         flags = (flags | PTE_COW) & (~PTE_W);
18         *pte = PA2PTE(pa) | flags;
19     }
20     refcinc((void*)pa); // increase the reference count of the
21 // physical page to 1
22 //     if((mem = kalloc()) == 0)
23 //         goto err;
24 //     memmove(mem, (char*)pa, PGSIZE);
25     if(mappages(new, i, PGSIZE, pa, flags) != 0){
26 //         kfree(mem);
27     goto err;
28     }
29     return 0;
30 }

```

2. 修改 `usertrap()` 以识别页面错误。当 COW 页面发生缺页时，使用 `kalloc()` 分配新页面，将旧页面复制到新页面，并将新页面安装到 PTE 中并设置 `PTE_W`。

在 `usertrap()` 中用 `scause() == 13 || scause() == 15` 来判断是否为 `page fault`，当发现是 `page fault` 并且 `r_stval()` 的物理页是 `COW` 页时，说明需要分配物理页，并重新映射到这个页表相应的虚拟地址上，当无法分配时，需要 `kill` 这个进程。注意：需要判断虚拟地址是否是有效的，其中包括需要判断这个虚拟地址是不是处在 `stack` 的 `guard page` 上，通过 `va <= PGROUNDDOWN(p->trapframe->sp) && va >= PGROUNDDOWN(p->trapframe->sp) - PGSIZE` 进行判断。

```

1 # kernel/trap.c/usertrap:70
2 } else if((which_dev = devintr()) != 0){
3     // ok
4 } else if (r_scause() == 13 || r_scause() == 15) {
5     uint64 va = r_stval();
6

```

```

7     if (va >= MAXVA || (va <= PGROUNDDOWN(p→trapframe→sp) &&
8         va >= PGROUNDDOWN(p→trapframe→sp) - PGSIZE)) {
9         p→killed = 1;
10    } else if (cow_alloc(p→pagetable, va) ≠ 0) {
11        p→killed = 1;
12    }
13 } else {
14     p→killed = 1;
15 }

```

3. 确保在对它的最后一个 PTE 引用消失时释放每个物理页面。为每个物理页保留一个“引用计数”，该“引用计数”是指引用该页的用户页表的数量。当 `kalloc()` 分配页面时，将页面的引用计数设置为 1。当 `fork` 导致子共享页面时增加页面的引用计数，并在每次任何进程从其页表中删除页面时减少页面的计数。`kfree()` 如果其引用计数为零，则应仅将页面放回空闲列表中。可以将这些计数保存在固定大小的整数数组中。可以使用页的物理地址除以 4096 来索引数组，并为数组赋予等于 `kalloc.c` 中 `kinit()` 放置在空闲列表中的任何页的最高物理地址的元素数。经过计算，`kalloc` 最多可分配**32723**的物理页面，因此，直接开辟了一个32723大小的 `ref` 数组，用以记录。

```

1 # kernel/kalloc.c
2 // allocate a physical address for virtual address va in
3 // for copy on write lab
4 int cow_alloc(pagetable_t pagetable, uint64 va) {
5     uint64 pa;
6     pte_t *pte;
7     uint flags;
8
9     if (va >= MAXVA) return -1;
10    va = PGROUNDDOWN(va);
11    pte = walk(pagetable, va, 0);
12    if (pte == 0) return -1;
13    if ((*pte & PTE_V) == 0) return -1;
14    pa = PTE2PA(*pte);
15    if (pa == 0) return -1;
16    flags = PTE_FLAGS(*pte);
17
18    if (flags & PTE_COW) {
19        char *mem = kalloc();
20        if (mem == 0) return -1;
21        memmove(mem, (char*)pa, PGSIZE);
22        flags = (flags & ~PTE_COW) | PTE_W;
23        *pte = PA2PTE((uint64)mem) | flags;
24        kfree((void*)pa);
25        return 0;
26    }

```

```
27     return 0;
28 }
29
30 # kernel/kalloc.c
31 //为了记录每个物理页被多少进程的页表引用，需要在kalloc.c中定义一个结构体
32 //refc,
33 //其中有一个大小为PGROUNDUP(PHYSTOP)/PGSIZE的int array来存放每个物
34 //理页的引用数
35 // struct to maintain the ref counts
36 struct {
37     struct spinlock lock;
38     int count[PGROUNDUP(PHYSTOP) / PGSIZE];
39 } refc;
40
41 # kernel/kalloc.c
42 void
43 refcinit()
44 {
45     initlock(&refc.lock, "refc");
46     for (int i = 0; i < PGROUNDUP(PHYSTOP) / PGSIZE; i++) {
47         refc.count[i] = 0;
48     }
49 }
50 void
51 refcinc(void *pa)
52 {
53     acquire(&refc.lock);
54     refc.count[PA2IDX(pa)]++;
55     release(&refc.lock);
56 }
57 void
58 refcdec(void *pa)
59 {
60     acquire(&refc.lock);
61     refc.count[PA2IDX(pa)]--;
62     release(&refc.lock);
63 }
64 // 获得索引
65 int
66 getrefc(void *pa)
67 {
68     return refc.count[PA2IDX(pa)];
69 }
70 // 初始化函数
71 void
72 kinit()
```

```

73 {
74     refcinit();
75     initlock(&kmem.lock, "kmem");
76     freerange(end, (void*)PHYSTOP);
77 }
78
79 # kernel/kalloc.c 修改kfree
80 void
81 kfree(void *pa)
82 {
83     struct run *r;
84
85     if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
86         panic("kfree");
87     refcdec(pa);
88     if (getrefc(pa) > 0) return;
89     // Fill with junk to catch dangling refs.
90     memset(pa, 1, PGSIZE);
91
92     r = (struct run*)pa;
93
94     acquire(&kmem.lock);
95     r->next = kmem.freelist;
96     kmem.freelist = r;
97     release(&kmem.lock);
98 }
```

4. 修改 `copyout()` 以在遇到 COW 页面时使用与页面错误相同的方案。

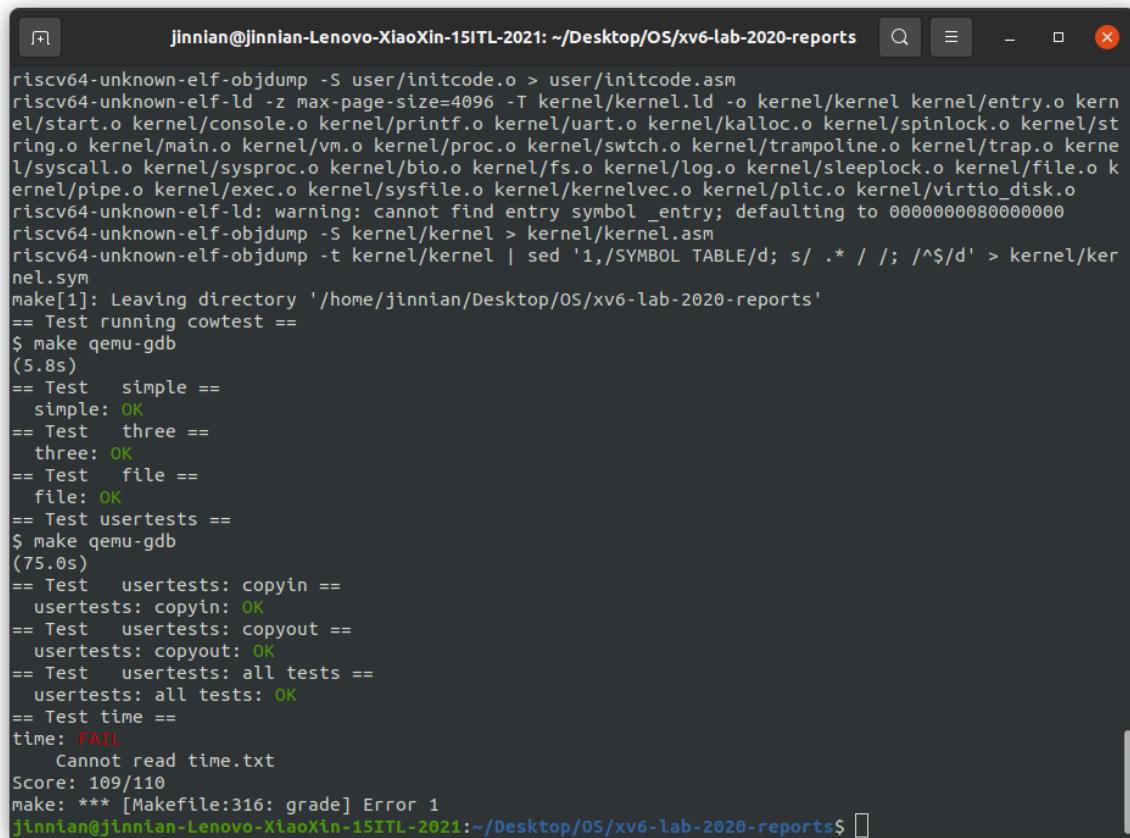
```

1 # kernel/vm.c
2 int
3 copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
4 {
5     uint64 n, va0, pa0;
6     pte_t *pte;
7     while(len > 0){
8         va0 = PGROUNDDOWN(dstva);
9         if(va0 >= MAXVA){
10             //printf("copyout(): va is greater than MAXVA\n");
11             return -1;
12         }
13         pte = walk(pagetable, va0, 0);
14         if(*pte & PTE_COW){
15             //printf("copyout(): got page COW faults at %p\n", va0);
16             char *mem;
17             if((mem = kalloc()) == 0)
```

```

18  {
19      printf("copyout(): memory alloc fault\n");
20      return -1;
21  }
22  memset(mem, 0, sizeof(mem));
23  uint64 pa = walkaddr(pagetable, va0);
24  if(pa){
25      memmove(mem, (char*)pa, PGSIZE);
26      int perm = PTE_FLAGS(*pte);
27      perm |= PTE_W;
28      perm &= ~PTE_COW;
29      if(mappages(pagetable, va0, PGSIZE, (uint64)mem, perm)
30      != 0){
31          printf("copyout(): can not map page\n");
32          kfree(mem);
33          return -1;
34      }
35      kfree((void*) pa);
36  }
37  pa0 = walkaddr(pagetable, va0);
38  if(pa0 == 0)
39      return -1;
40  n = PGSIZE - (dstva - va0);
41  if(n > len)
42      n = len;
43  memmove((void*)(pa0 + (dstva - va0)), src, n);
44
45  len -= n;
46  src += n;
47  dstva = va0 + PGSIZE;
48 }
49 return 0;
50 }
```

运行结果及评分：



```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6-lab-2020-reports
riscv64-unknown-elf-objdump -S user/initcode.o > user/initcode.asm
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/kalloc.o kernel/spinlock.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o
riscv64-unknown-elf-ld: warning: cannot find entry symbol _entry; defaulting to 0000000080000000
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/jinnian/Desktop/OS/xv6-lab-2020-reports'
== Test running cowtest ==
$ make qemu-gdb
(5.8s)
== Test simple ==
simple: OK
== Test three ==
three: OK
== Test file ==
file: OK
== Test usertests ==
$ make qemu-gdb
(75.0s)
== Test usertests: copyin ==
usertests: copyin: OK
== Test usertests: copyout ==
usertests: copyout: OK
== Test usertests: all tests ==
usertests: all tests: OK
== Test time ==
time: FAIL
    Cannot read time.txt
Score: 109/110
make: *** [Makefile:316: grade] Error 1
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-lab-2020-reports$
```

Lab 7 Multithreading

Assignment 1 —— switching between threads

本任务要求为用户级线程系统设计上下文切换机制，然后实现它。xv6有两个文件：`user/uthread.c`和`*user/uthread_switch.S`，以及一个规则：运行在Makefile中以构建`uthread`程序。`uthread.c`包含大多数用户级线程包，以及三个简单测试线程的代码。线程包缺少一些用于创建线程和在线程之间切换的代码。最终目标是提出一个创建线程和保存/恢复寄存器以在线程之间切换的计划，并实现该计划。完成后，`make grade`应该表明解决方案通过了`uthread`测试。

Hints:

- 在`thread_create()`、`thread_switch` 和 `thread_schedule()` 中添加你的代码

第一次创建进程时需要初始化`ra`和`sp`寄存器。`ra`寄存器需要存放传入的函数地址，`sp`寄存器传入当前线程的栈底(最开始的位置)

```
1 # user/uthread.c
2 void thread_create(void (*func)())
3 {
4     struct thread *t;
```

```

5
6   for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
7     if (t->state == FREE) break;
8     t->state = RUNNABLE;
9     // YOUR CODE HERE
10    t->context.ra = (uint64)func;
11    t->context.sp = (uint64)t->stack + STACK_SIZE;
12  }
13 }
14
15 # user/uthread.c/thread_schedule
16 //thread_schedule()中,直接调用thread_switch,仿照swtch的格式进行上下文切换
17
18 /* YOUR CODE HERE
19 * Invoke thread_switch to switch from t to next_thread:
20 * thread_switch(??, ??);
21 */
22 thread_switch((uint64)&t->context,
23 (uint64)&current_thread->context);

```

- 修改 `struct thread` 来保存上下文

```

1 struct thread {
2   char      stack[STACK_SIZE]; /* the thread's stack */
3   int       state;           /* FREE, RUNNING, RUNNABLE */
4   struct context context;
5 };
6 struct thread all_thread[MAX_THREAD];
7 struct thread *current_thread;
8 extern void thread_switch(uint64, uint64);

```

- 在 `thread_schedule()` 中调用 `thread_switch()` --该函数应该在 `uthread_switch.S` 中实现

The screenshot shows a code editor window with the file name "uthread_switch.S" open. The assembly code is as follows:

```
8     .globl thread_switch
9 thread_switch:
10    /* 每个uint64 大小为8个字节, 按照偏移量储存然后再载入就行*/
11    /* YOUR CODE HERE */
12    sd ra, 0(a0)
13    sd sp, 8(a0)
14    sd s0, 16(a0)
15    sd s1, 24(a0)
16    sd s2, 32(a0)
17    sd s3, 40(a0)
18    sd s4, 48(a0)
19    sd s5, 56(a0)
20    sd s6, 64(a0)
21    sd s7, 72(a0)
22    sd s8, 80(a0)
23    sd s9, 88(a0)
24    sd s10, 96(a0)
25    sd s11, 104(a0)
26
27    ld ra, 0(a1)
28    ld sp, 8(a1)
29    ld s0, 16(a1)
30    ld s1, 24(a1)
31    ld s2, 32(a1)
32    ld s3, 40(a1)
33    ld s4, 48(a1)
34    ld s5, 56(a1)
35    ld s6, 64(a1)
36    ld s7, 72(a1)
37    ld s8, 80(a1)
38    ld s9, 88(a1)
39    ld s10, 96(a1)
40    ld s11, 104(a1)
41    ret    /* return to ra */
```

At the bottom of the editor, there are status bars for "Plain Text", "Tab Width: 8", "Ln 10, Col 30", and "INS".

- `thread_switch()` 只需要保存和还原唤醒线程和被唤醒线程的上下文

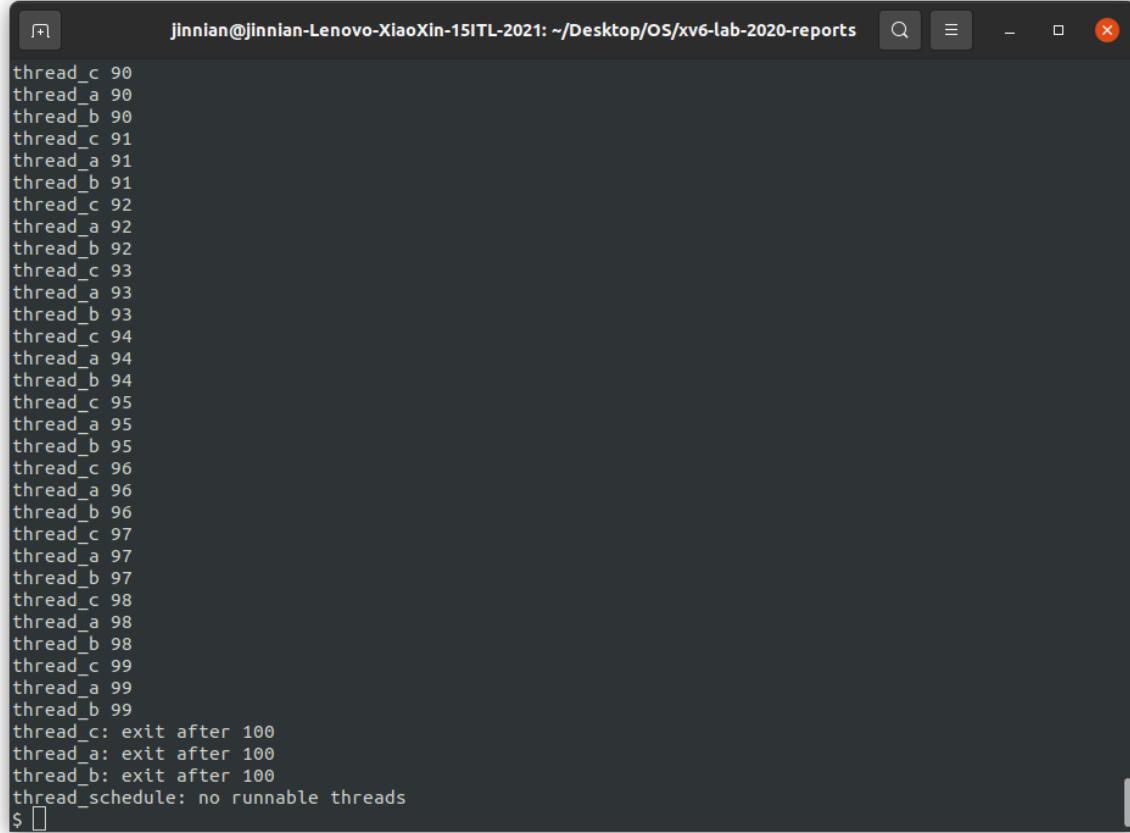
运行结果：

The screenshot shows a terminal window with the command "make qemu" run on the xv6 kernel. The output shows the kernel booting and three threads (a, b, c) starting and printing their IDs.

```
make: *** [Makefile:317: grade] Error 1
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-lab-2020-reports$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_b 2
thread_c 3
thread_a 3
thread_b 3
thread_c 4
thread_a 4
thread_b 4
thread_c 5
thread_a 5
thread_b 5
thread_c 6
thread_a 6
thread_b 6
```



jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6-lab-2020-reports

```
thread_c 90
thread_a 90
thread_b 90
thread_c 91
thread_a 91
thread_b 91
thread_c 92
thread_a 92
thread_b 92
thread_c 93
thread_a 93
thread_b 93
thread_c 94
thread_a 94
thread_b 94
thread_c 95
thread_a 95
thread_b 95
thread_c 96
thread_a 96
thread_b 96
thread_c 97
thread_a 97
thread_b 97
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
$ 
```

Assignment 2 —— Using threads

在本任务中，使用哈希表的线程和锁的并行编程。在具有多个内核的真实Linux或MacOS计算机（不是xv6，不是qemu）上执行此任务。这个任务使用UNIX的pthread线程库。使用`man pthreads`在手册页面上找到关于它的信息。

文件**notxv6/ph.c**包含一个简单的哈希表，如果单个线程使用，该哈希表是正确的，但是多个线程使用时，该哈希表是不正确的。类似于：

```
1 $ ./ph 1
2 100000 puts, 3.991 seconds, 25056 puts/second
3 0: 0 keys missing
4 100000 gets, 3.981 seconds, 25118 gets/second
```

`ph`运行两个基准程序。首先，它通过调用`put()`将许多键添加到哈希表中，并以每秒为单位打印`puts`的接收速率。之后它使用`get()`从哈希表中获取键。它打印由于`puts`而应该在哈希表中但丢失的键的数量（在本例中为0），并以每秒为单位打印`gets`的接收数量。

当增加使用哈希表的线程：

```
1 $ ./ph 2
2 100000 puts, 1.885 seconds, 53044 puts/second
3 1: 16579 keys missing
4 0: 16579 keys missing
5 200000 gets, 4.322 seconds, 46274 gets/second
```

考虑以下情况：

有两个键 k1 和 k2，他们属于散列表中的同一链表，并且链表中都还不存在这两个键值对。现在有两个线程 t1 和 t2，它们分别尝试在该链表中插入这两个键值。

那么有如下的可能情况：

t1 先检查了链表中不存在 k1，于是准备调用 `insert()` 在链表前插入键值对。这个时候，线程调度器切换到了 t2（也可能是在多核环境下，两个线程并行执行，但是 t2 比 t1 快）。然后 t2 也发现了链表中不存在 k2，所以调用 `insert()` 插入。插入之后，k2 成为了链表的第一个元素。随后 t1 也真正的插入了 k1。但是，因为 t1 并不知道 t2 已经把 k2 插入到了开头，于是在其认为的链表开头 (k2 所处位置) 插入了 k1，k2 就被覆盖掉了，于是造成了键值对丢失。

为了避免这种事件序列，在`notxv6/ph.c`中的 `put` 和 `get` 中插入 `lock` 和 `unlock` 语句，以便在两个线程中丢失的键数始终为0。相关的pthread调用包括：

- `pthread_mutex_t lock; // declare a lock`
- `pthread_mutex_init(&lock, NULL); // initialize the lock`
- `pthread_mutex_lock(&lock); // acquire lock`
- `pthread_mutex_unlock(&lock); // release lock`

```
1 # notxv6/ph.c
2 // 定义一个全局变量pthread_mutex_t lock[NBUCKET]; 其中NBUCKET是同时可以
3 // 进行操作的线程最大数量
4 #define NBUCKET 5
5 pthread_mutex_t lock[NBUCKET];
6
7 // 在main函数中初始化
8 for (int i=0; i<NBUCKET; i++) pthread_mutex_init(&lock[i], NULL);
9
10 // 在put()和get()函数中上锁和解锁以保护对哈希表键值对的读写操作.
11 static
12 void put(int key, int value)
13 {
14     int i = key % NBUCKET;
15     pthread_mutex_lock(&lock[i]);
16     // is the key already present?
17     struct entry *e = 0;
18     for (e = table[i]; e != 0; e = e->next) {
```

```

19     break;
20 }
21 if(e){
22     // update the existing key.
23     e->value = value;
24 } else {
25     // the new is new.
26     insert(key, value, &table[i], table[i]);
27 }
28 pthread_mutex_unlock(&lock[i]);
29 }
30
31 static struct entry*
32 get(int key)
33 {
34     int i = key % NBUCKET;
35
36     pthread_mutex_lock(&lock[i]);
37     struct entry *e = 0;
38     for (e = table[i]; e != 0; e = e->next) {
39         if (e->key == key) break;
40     }
41     pthread_mutex_unlock(&lock[i]);
42     return e;
43 }
44

```

Assignment 3 —— Barrier

在本任务中，完成一个**屏障**：应用程序中的一个点，所有参与的线程都必须等待，直到所有其他参与的线程也到达该点。使用 pthread 条件变量，这是一种类似于 xv6 的睡眠和唤醒的序列协调技术。

In [parallel computing](#), a **barrier** is a type of [synchronization](#) method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

T：在并行计算中，**屏障**是一种同步方法。源代码中一组线程或进程的屏障意味着任何线程/进程必须在此时停止，并且在所有其他线程/进程到达此屏障之前无法继续。

文件**notxv6/barrier.c**包含一个残缺的屏障实现：

```

1 $ ./barrier 2
2 barrier: notxv6/barrier.c:42: thread: Assertion `i == t' failed.

```

2指定在屏障上同步的线程数 (**barrier.c**中的**nthread**)。每个线程执行一个循环。在每次循环迭代中，线程都会调用**barrier()**，然后以随机微秒数休眠。如果一个线程在另一个线程到达屏障之前离开屏障将触发断言 (assert)。期望的行为是每个线程在**barrier()**中阻塞，直到**nthreads**的所有线程都调用了**barrier()**。

目标是实现期望的屏障行为。除了在ph作业中看到的lock原语外，还需要以下新的pthread原语：

- `pthread_cond_wait(&cond, &mutex);` // 在cond上进入睡眠，释放锁mutex，在醒来时重新获取
- `pthread_cond_broadcast(&cond);` // 唤醒睡在cond的所有线程

Hints:

- 必须处理一系列的**barrier**调用，称每一连串的调用为一轮 (round)。
`bstate.round`记录当前轮数。每次当所有线程都到达屏障时，都应增加`bstate.round`。
- 必须处理这样的情况：一个线程在其他线程退出**barrier**之前进入了下一轮循环。特别是，在前后两轮中重复使用`bstate.nthread`变量。确保在前一轮仍在使用`bstate.nthread`时，离开**barrier**并循环运行的线程不会增加`bstate.nthread`。

Solution:

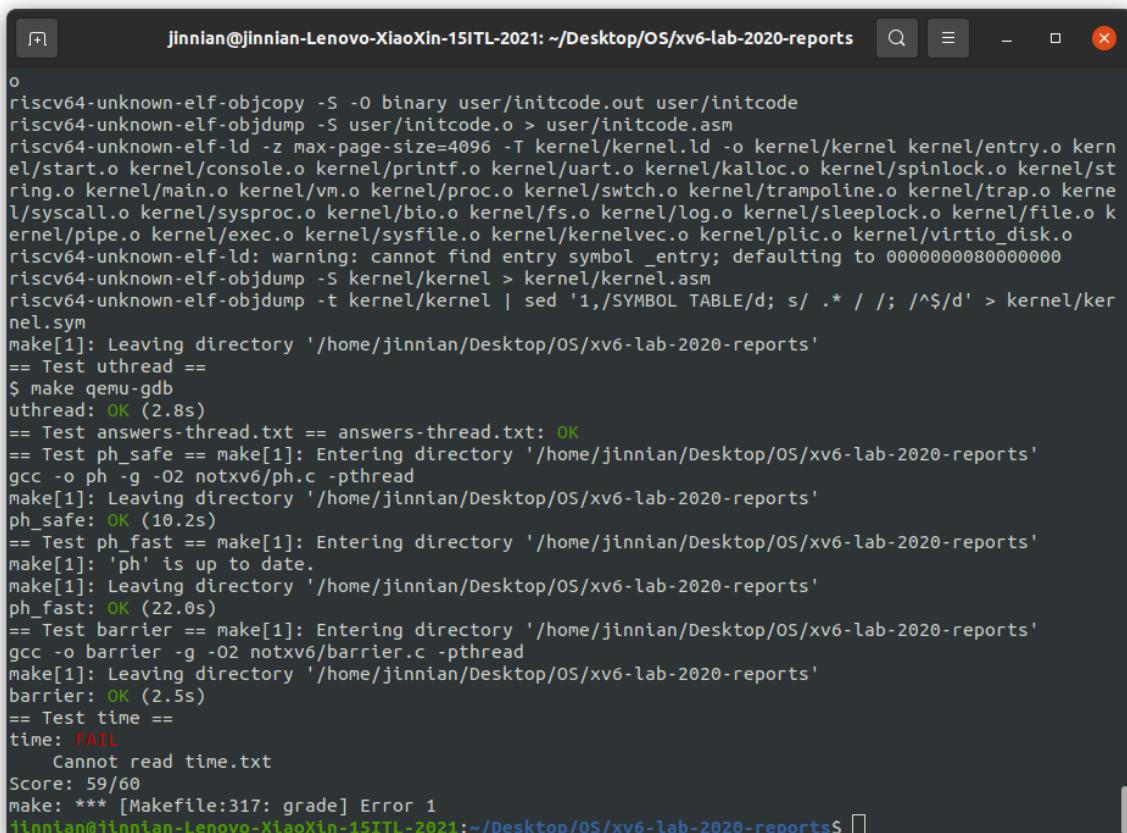
当每个线程调用了**barrier()**之后，需要增加`bstate.nthread`以表明到达当前**round**的线程数量增加了1，但是由于`bstate`这个数据结构是线程之间共享的，因此需要用`pthread_mutex_lock`对这个数据结构进行保护。当`bstate.nthread`的数量达到线程总数`nthread`之后，将`bstate.round`加1。注意，一定要等到所有的线程都达到了这个**round**，将`bstate.nthread`清零之后才能将所有正在睡眠的线程唤醒，否则如果先唤醒线程的话其他线程如果跑得很快，在之前的线程将`bstate.nthread`清零之前就调用了`bstate.nthread++`，会出现问题(**round**之间是共用`bstate.nthread`的)。

```
1 # notxv6/barrier.c
2 static void
3 barrier()
4 {
5     // YOUR CODE HERE
6     //
7     // Block until all threads have called barrier() and
8     // then increment bstate.round.
9     //
10    pthread_mutex_lock(&bstate.barrier_mutex);
11    bstate.nthread++;
12    if (bstate.nthread == nthread) {
13        bstate.round++;
14        bstate.nthread = 0;
15        pthread_cond_broadcast(&bstate.barrier_cond);
```

```

16     pthread_mutex_unlock(&bstate.barrier_mutex);
17 } else {
18     pthread_cond_wait(&bstate.barrier_cond,
19         &bstate.barrier_mutex);
20 }
21 }
```

运行结果及评分：



```

o
riscv64-unknown-elf-objcopy -S -O binary user/initcode.out user/initcode
riscv64-unknown-elf-objdump -S user/initcode.o > user/initcode.asm
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/kalloc.o kernel/spinlock.o kernel/strring.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o
riscv64-unknown-elf-ld: warning: cannot find entry symbol _entry; defaulting to 0000000080000000
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/jinnian/Desktop/OS/xv6-lab-2020-reports'
== Test uthread ==
$ make qemu-gdb
uthread: OK (2.8s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/jinnian/Desktop/OS/xv6-lab-2020-reports'
gcc -o ph -g -O2 notxv6/ph.c -pthread
make[1]: Leaving directory '/home/jinnian/Desktop/OS/xv6-lab-2020-reports'
ph_safe: OK (10.2s)
== Test ph_fast == make[1]: Entering directory '/home/jinnian/Desktop/OS/xv6-lab-2020-reports'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/jinnian/Desktop/OS/xv6-lab-2020-reports'
ph_fast: OK (22.0s)
== Test barrier == make[1]: Entering directory '/home/jinnian/Desktop/OS/xv6-lab-2020-reports'
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/jinnian/Desktop/OS/xv6-lab-2020-reports'
barrier: OK (2.5s)
== Test time ==
time: FAIL
    Cannot read time.txt
Score: 59/60
make: *** [Makefile:31: grade] Error 1
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-lab-2020-reports$
```

Lab 8 Locks

多核机器上并行性差的一个常见症状是频繁的锁争用。提高并行性通常涉及更改数据结构和锁定策略以减少争用。本任务将对xv6内存分配器和块缓存执行此操作。

Assignment 1 —— Memory allocator

程序 `user/kalloctest.c` 强调了 xv6 的内存分配器：三个进程增长和缩小地址空间，导致对 `kalloc` 和 `kfree` 的多次调用。`kalloc` 和 `kfree` 获得 `kmem.lock`。`kalloc` 打印（作为“#fetch-and-add”）在 `acquire` 中由于尝试获取另一个内核已经持有的锁而进行的循环迭代次数，如 `kmem` 锁和一些其他锁。`acquire` 中的循环迭代次数是锁争用的粗略度量。完成实验前，`kalloc` 的输出与此类似：

```

1 $ kalloc test \
2 start test1
3 test1 results:
4 --- lock kmem/bcache stats
5 lock: kmem: #fetch-and-add 83375 #acquire() 433015
6 lock: bcache: #fetch-and-add 0 #acquire() 1260
7 --- top 5 contended locks:
8 lock: kmem: #fetch-and-add 83375 #acquire() 433015
9 lock: proc: #fetch-and-add 23737 #acquire() 130718
10 lock: virtio_disk: #fetch-and-add 11159 #acquire() 114
11 lock: proc: #fetch-and-add 5937 #acquire() 130786
12 lock: proc: #fetch-and-add 4080 #acquire() 130786
13 tot= 83375
14 test1 FAIL

```

`acquire` 为每个锁维护要获取该锁的 `acquire` 调用计数，以及 `acquire` 中循环尝试但未能设置锁的次数。`kalloc test` 调用一个系统调用，使内核打印 `kmem` 和 `bcache` 锁（这是本实验的重点）以及5个最有竞争的锁的计数。如果存在锁争用，则 `acquire` 循环迭代的次数将很大。系统调用返回 `kmem` 和 `bcache` 锁的循环迭代次数之和。

`kalloc test` 中锁争用的根本原因是 `kalloc()` 有一个空闲列表，由一个锁保护。要消除锁争用，必须重新设计内存分配器，以避免使用单个锁和列表。基本思想是为每个CPU维护一个空闲列表，每个列表都有自己的锁。因为每个CPU将在不同的列表上运行，不同CPU上的分配和释放可以并行运行。主要的挑战将是处理一个CPU的空闲列表为空，而另一个CPU的列表有空闲内存的情况；在这种情况下，一个CPU必须“窃取”另一个CPU空闲列表的一部分。也就是从其他核心“偷”空间。

Question:

目标是实现每个CPU的空闲列表，并在CPU的空闲列表为空时进行窃取。所有锁的命名必须以“`kmem`”开头。也就是说，为每个锁调用 `initlock`，并传递一个以“`kmem`”开头的名称。运行 `kalloc test` 以查看是否减少了锁争用。要检查它是否仍然可以分配所有内存，请运行 `usertests sbrkmuch`。输出将与下面所示的类似，在 `kmem` 锁上的争用总数将大大减少，尽管具体的数字会有所不同。确保 `usertests` 中的所有测试都通过。评分应该表明考试通过。

```

1 $ kalloc test
2 start test1
3 test1 results:
4 --- lock kmem/bcache stats
5 lock: kmem: #fetch-and-add 0 #acquire() 42843
6 lock: kmem: #fetch-and-add 0 #acquire() 198674
7 lock: kmem: #fetch-and-add 0 #acquire() 191534
8 lock: bcache: #fetch-and-add 0 #acquire() 1242
9 --- top 5 contended locks:
10 lock: proc: #fetch-and-add 43861 #acquire() 117281

```

```

11 lock: virtio_disk: #fetch-and-add 5347 #acquire() 114
12 lock: proc: #fetch-and-add 4856 #acquire() 117312
13 lock: proc: #fetch-and-add 4168 #acquire() 117316
14 lock: proc: #fetch-and-add 2797 #acquire() 117266
15 tot= 0
16 test1 OK
17 start test2
18 total free number of pages: 32499 (out of 32768)
19 ....
20 test2 OK
21 $ usertests sbrkmuch
22 usertests starting
23 test sbrkmuch: OK
24 ALL TESTS PASSED
25 $ usertests
26 ...
27 ALL TESTS PASSED

```

Hints:

- 可以使用 kernel/param.h 中的常量 `NCPU` (即CPU核心的数量)
- 让 `freerange` 将所有空闲内存分配给运行 `freerange` 的 CPU。
- 函数 `cpuid` 返回当前的核心编号，但只有在中断关闭时调用它并使用它的结果才是安全的。使用 `push_off()` 和 `pop_off()` 来关闭和打开中断。
- 查看kernel/sprintf.c 中的 `snprintf` 函数以了解字符串格式化的想法。尽管可以将所有锁命名为“`kmem`”。

Solution:

在 `kernel/kalloc.c` 中，修改 `kmem` 结构体为数组形式

`kinit()` 要循环初始化每一个 `kmem` 的锁

```

1 # kernel/kalloc.c
2 struct {
3     struct spinlock lock;
4     struct run *freelist;
5 } kmem[NCPU];
6
7 void kinit()
8 {
9     for (int i = 0; i < NCPU; i++) {
10         initlock(&kmem[i].lock, "kmem");
11     }
12     freerange(end, (void*)PHYSTOP);
13 }

```

`kfree` 将释放出来的 `freelist` 节点返回给调用 `kfree` 的CPU

```

1 # kernel/kalloc.c
2 void
3 kfree(void *pa)
4 {
5     struct run *r;
6
7     if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa
8     ≥ PHYSTOP)
9         panic("kfree");
10
11     // Fill with junk to catch dangling refs.
12     memset(pa, 1, PGSIZE);
13
14     r = (struct run*)pa;
15
16     push_off();
17     int ncpu = cpuid();
18
19     acquire(&kmem[ncpu].lock);
20     r→next = kmem[ncpu].freelist;
21     kmem[ncpu].freelist = r;
22     release(&kmem[ncpu].lock);
23     pop_off();
24 }
```

`kalloc` 中，当发现freelist已经用完后，需要向其他CPU的freelist借用节点

```

1 # kernel/kalloc.c
2 void *
3 kalloc(void)
4 {
5     struct run *r;
6
7     push_off();
8     int ncpu = cpuid();
9
10    acquire(&kmem[ncpu].lock);
11    r = kmem[ncpu].freelist;
12    if(r) {
13        kmem[ncpu].freelist = r→next;
14    }
15    release(&kmem[ncpu].lock);
16    if (!r) {
17        // steal other cpu's freelist
18        for (int i = 0; i < NCPU; i++) {
19            if (i == ncpu) continue;
20            acquire(&kmem[i].lock);
```

```

21     r = kmem[i].freelist;
22     if (r) {
23         kmem[i].freelist = r->next;
24         release(&kmem[i].lock);
25         break;
26     }
27     release(&kmem[i].lock);
28 }
29 }
30 pop_off();
31
32 if(r)
33     memset((char*)r, 5, PGSIZE); // fill with junk
34 return (void*)r;
35 }

```

运行结果：

```

jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6-lab-2020-reports
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 81578
lock: kmem: #fetch-and-add 0 #acquire() 194013
lock: kmem: #fetch-and-add 0 #acquire() 157410
lock: bcache: #fetch-and-add 0 #acquire() 14
lock: bcache: #fetch-and-add 0 #acquire() 18
lock: bcache: #fetch-and-add 0 #acquire() 25
lock: bcache: #fetch-and-add 0 #acquire() 21
lock: bcache: #fetch-and-add 0 #acquire() 20
lock: bcache: #fetch-and-add 0 #acquire() 13
lock: bcache: #fetch-and-add 0 #acquire() 16
lock: bcache: #fetch-and-add 0 #acquire() 13
lock: bcache: #fetch-and-add 0 #acquire() 275
lock: bcache: #fetch-and-add 0 #acquire() 3
lock: bcache: #fetch-and-add 0 #acquire() 5
lock: bcache: #fetch-and-add 0 #acquire() 8
lock: bcache: #fetch-and-add 0 #acquire() 11
--- top 5 contended locks:
lock: proc: #fetch-and-add 24189 #acquire() 162742
lock: virtio_disk: #fetch-and-add 8062 #acquire() 57
lock: proc: #fetch-and-add 6425 #acquire() 162780
lock: proc: #fetch-and-add 5049 #acquire() 162784
lock: proc: #fetch-and-add 4420 #acquire() 162748
tot= 0
test1 OK
start test2
total free number of pages: 32471 (out of 32768)
.....
test2 OK
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$ 

```

Assignment 2 —— Buffer cache

xv6文件系统的buffer cache采用了一个全局的锁 `bcache.lock` 来负责对buffer cache进行读写保护，当xv6执行读写文件强度较大的任务时会产生较大的锁竞争压力，因此需要一个哈希表，将buf entry以 `buf.blockno` 为键哈希映射到这个哈希表的不同的BUCKET中，给每个BUCKET一个锁，`NBUCKET`最好选择素数，这里选择13。注意：这个实验不能

像上一个一样给每个CPU一个`bcache`，因为文件系统在多个CPU之间是真正实现共享的，否则将会造成一个CPU只能访问某些文件的问题。

如果多个进程密集地使用文件系统，它们可能会争夺`bcache.lock`，它保护`kernel/bio.c`中的磁盘块缓存。`bcachetest`创建多个进程，这些进程重复读取不同的文件，以便在`bcache.lock`上生成争用；如果查看`kernel/bio.c`中的代码，将看到`bcache.lock`保护已缓存的块缓冲区的列表、每个块缓冲区中的引用计数（`b->refcnt`）以及缓存块的标识（`b->dev`和`b->blockno`）。（在完成本实验之前）其输出如下所示：

```
1 $ bcachetest
2 start test0
3 test0 results:
4 --- lock kmem/bcache stats
5 lock: kmem: #fetch-and-add 0 #acquire() 33035
6 lock: bcache: #fetch-and-add 16142 #acquire() 65978
7 --- top 5 contended locks:
8 lock: virtio_disk: #fetch-and-add 162870 #acquire() 1188
9 lock: proc: #fetch-and-add 51936 #acquire() 73732
10 lock: bcache: #fetch-and-add 16142 #acquire() 65978
11 lock: uart: #fetch-and-add 7505 #acquire() 117
12 lock: proc: #fetch-and-add 6937 #acquire() 73420
13 tot= 16142
14 test0: FAIL
15 start test1
16 test1 OK
```

Question:

修改块缓存，以便在运行`bcachetest`时，`bcache`（buffer cache的缩写）中所有锁的`acquire`循环迭代次数接近于零。理想情况下，块缓存中涉及的所有锁的计数总和应为零，但只要总和小于500就可以。修改`bget`和`brelse`，以便`bcache`中不同块的并发查找和释放不太可能在锁上发生冲突（例如，不必全部等待`bcache.lock`）。必须保护每个块最多缓存一个副本的不变量。

Hints:

- 将所有的锁以“`bcache`”开头进行命名。也就是说，为每个锁调用`initlock`，并传递一个以“`bcache`”开头的名称。
- 可以使用固定数量的散列桶，而不动态调整哈希表的大小。使用素数个存储桶（例如13）来降低散列冲突的可能性。
- 在哈希表中搜索缓冲区并在找不到缓冲区时为该缓冲区分配条目必须是原子的，即不可分割的。
- 删除保存了所有缓冲区的列表（`bcache.head`等），改为标记上次使用时间的时间戳缓冲区（即使用`kernel/trap.c`中的`ticks`）。通过此更改，`brelse`不需要获取`bcache`锁，并且`bget`可以根据时间戳选择最近使用最少的块。★

- 可以在 `bget` 中串行化回收（即 `bget` 中的一部分：当缓存中的查找未命中时，它选择要复用的缓冲区）。
- 在某些情况下，可能需要持有两个锁；例如，在回收过程中，可能需要持有 `bcache` 锁和每个 `bucket`（散列桶）一个锁。确保避免死锁。
- 替换块时，可能会将 `struct buf` 从一个 `bucket` 移动到另一个 `bucket`，因为新块散列到不同的 `bucket`。可能会遇到一个棘手的情况：新块可能会散列到与旧块相同的 `bucket` 中。在这种情况下，请确保避免死锁。
- 一些调试技巧：实现 `bucket` 锁，但将全局 `bcache.lock` 的 `acquire / release` 保留在 `bget` 的开头/结尾，以串行化代码。一旦确定它在没有竞争条件的情况下是正确的，请移除全局锁并处理并发性问题。还可以运行 `make CPUS=1 qemu` 以使用一个内核进行测试。

Solution:

在 `kernel/bio.c` 中，首先设置 `NBUCKET` 宏定义为 13，声明外部变量 `ticks`，修改 `bcache` 以为每个 `BUCKET` 设置一个链表头，并设置一个锁

```

1 # kernel/bio.c
2 #define NBUCKET 13
3
4 uint extern ticks;
5
6 struct {
7     struct spinlock lock[NBUCKET];
8     struct buf buf[NBUF];
9
10    // Linked list of all buffers, through prev/next.
11    // Sorted by how recently the buffer was used.
12    // head.next is most recent, head.prev is least.
13    struct buf head[NBUCKET];
14 } bcache;
```

修改 `kernel/buf.h` 中的 `buf` 结构体，删除 `struct buf *prev`，即将双向链表变为单向链表，添加 `uint time` 这个成员变量作为时间戳。

```

1 # kernel/buf.h
2 int hash (int n) {
3     int result = n % NBUCKET;
4     return result;
5 }
```

修改 `binit` 函数，为每个 `bcache.lock` 以及 `b->lock` 进行初始化，并将所有 `buf` 先添加到 `bucket 0` 哈希表中

```

1 void
2 binit(void)
```

```

3 {
4     struct buf *b;
5
6     for (int i = 0; i < NBUCKET; i++) {
7         initlock(&bcache.lock[i], "bcache");
8     }
9
10    bcache.head[0].next = &bcache.buf[0];
11    // for initialization, append all bufs to bucket 0
12    for (b = bcache.buf; b < bcache.buf+NBUF-1; b++) {
13        b->next = b+1;
14        initsleeplock(&b->lock, "buffer");
15    }
16 }

```

修改 `bget`，先查找当前哈希表中有没有和传入参数 `dev`、`blockno` 相同的 `buf`。首先要将 `blockno` 哈希到一个 `id` 值，然后获得对应 `id` 值的 `bcache.lock[id]` 锁，然后在这个 `bucket id` 哈希链表中查找符合对应条件的 `buf`，如果找到则返回，返回前释放掉 `bcache.lock[id]`，并对 `buf` 加 `sleeplock`。

```

1 # kernel/bio.c/bget
2 int id = hash(blockno);
3 acquire(&bcache.lock[id]);
4 b = bcache.head[id].next;
5 while (b) {
6     if (b->dev == dev && b->blockno == blockno) {
7         b->refcnt++;
8         if (holding(&bcache.lock[id]))
9             release(&bcache.lock[id]);
10        acquiresleep(&b->lock);
11        return b;
12    }
13    b = b->next;
14 }

```

如果没有找到对应的 `buf`，需要在整个哈希表中查找 LRU(least recently used) `buf`，将其替换掉。这里由于总共有 `NBUCKET` 个哈希表，而此时一定是持有 `bcache.lock[id]` 这个哈希表的锁的，因此当查找其他哈希表时，需要获取其他哈希表的锁，这时就会有产生死锁的风险。

风险1：查找的哈希表正是自己本身这个哈希表，在已经持有自己哈希表锁的情况下，不能再尝试 `acquire` 一遍自己的锁。

风险2：假设有2个进程同时要进行此步骤，进程1已经持有了哈希表A的锁，尝试获取哈希表B的锁，进程2已经持有了哈希表B的锁，尝试获取哈希表A的锁，同样会造成死锁，因此要规定一个规则，当持有哈希表A的情况下如果能够获取哈希表B的锁，则当持有哈希表B锁的情况下不能够持有哈希表A的锁。该规则在 `can_lock` 函数中实现。

```
1 # kernel/bio.c
2 int can_lock(int id, int j) {
3     int num = NBUCKET/2;
4     if (id <= num) {
5         if (j > id && j <= (id+num))
6             return 0;
7     } else {
8         if ((id < j && j < NBUCKET) || (j <= (id+num)%NBUCKET)) {
9             return 0;
10        }
11    }
12    return 1;
13 }
```

其中 `id` 是已经持有的锁，`j` 是判断是否能获取该索引哈希表的锁。这个规则实际上规定了在持有某一个锁的情况下，只能再尝试获取 `NBUCKET/2` 个哈希表锁，另一半哈希表锁是不能获取的。

确定了这个规则之后，尝试遍历所有的哈希表，通过 `b->time` 查找LRU `buf`。先判断当前的哈希表索引是否为 `id`，如果是，则不获取这个锁（已经获取过它了），但是还是要遍历这个哈希表的；同时也要判断当前哈希表索引是否满足 `can_lock` 规则，如果不满足，则不遍历这个哈希表，直接 `continue`。如果哈希表索引 `j` 既不是 `id`，也满足 `can_lock`，则获取这个锁，并进行遍历。当找到了一个当前情况下的 `b->time` 最小值时，如果这个最小值和上一个最小值不在同一个哈希表中，则释放上一个哈希表锁，一直持有拥有当前情况下 LRU `buf` 这个哈希表的锁，直到找到新的 LRU `buf` 且不是同一个哈希表为止。找到 LRU `buf` 后，由于此时还拥有这个哈希表的锁，因此可以直接将这个 `buf` 从该哈希链表中剔除，并将其 append 到 bucket `id` 哈希表中，修改这个锁的 `dev`、`blockno`、`valid`、`refcnt` 等属性。最后释放所有的锁。

```
1 # kernel/bio.c
2 int index = -1;
3 uint smallest_tick = __UINT32_MAX__;
4 // find the LRU unused buffer
5 for (int j = 0; j < NBUCKET; ++j) {
6     if (j != id && can_lock(id, j)) {
7         // if j == id, then lock is already acquired
8         // can_lock is to maintain an invariant of lock
acquisition order
```

```

9         // to avoid dead lock
10        acquire(&bcache.lock[j]);
11    } else if (!can_lock(id, j)) {
12        continue;
13    }
14    b = bcache.head[j].next;
15    while (b) {
16        if (b->refcnt == 0) {
17            if (b->time < smallest_tick) {
18                smallest_tick = b->time;
19                if (index != -1 && index != j &&
holding(&bcache.lock[index])) release(&bcache.lock[index]);
20                    index = j;
21                }
22            }
23            b = b->next;
24        }
25        if (j!=id && j!=index && holding(&bcache.lock[j]))
release(&bcache.lock[j]);
26    }
27    if (index == -1) panic("bget: no buffers");
28    b = &bcache.head[index];
29
30    while (b) {
31        if ((b->next)->refcnt == 0 && (b->next)->time ==
smallest_tick) {
32            selected = b->next;
33            b->next = b->next->next;
34            break;
35        }
36        b = b->next;
37    }
38    if (index != id && holding(&bcache.lock[index]))
release(&bcache.lock[index]);
39    b = &bcache.head[id];
40    while (b->next) {
41        b = b->next;
42    }
43    b->next = selected;
44    selected->next = 0;
45    selected->dev = dev;
46    selected->blockno = blockno;
47    selected->valid = 0;
48    selected->refcnt = 1;
49    if (holding(&bcache.lock[id]))
50        release(&bcache.lock[id]);
51    acquiresleep(&selected->lock);
52    return selected;

```

修改 `brelse`。当 `b->refcnt==0` 时，说明这个 `buf` 已经被使用完了，可以进行释放，为其加上时间戳

```

1 # kernel/bio.c
2 void
3 brelse(struct buf *b)
4 {
5     if(!holdingsleep(&b->lock))
6         panic("brelse");
7
8     releasesleep(&b->lock);
9
10    int id = hash(b->blockno);
11    acquire(&bcache.lock[id]);
12    b->refcnt--;
13    if (b->refcnt == 0) {
14        b->time = ticks;
15    }
16
17    release(&bcache.lock[id]);
18 }
```

修改 `bpin` 和 `bunpin`，将 `bcache.lock` 修改为 `bcache.lock[id]`

最后在 `param.h` 中修改 `NBUF` 的值，由于 `can_lock` 规则，`NBUF` 实际上变成了原来的一半，可能会出现 `buffer run out` 的 `panic`，无法通过 `writebig` 测试，因此适当增大 `NBUF`，这里将其修改为了 `(MAXOPBLOCKS*12)`

运行测试：

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6-lab-2020-reports
```

```
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 431953
lock: kmem: #fetch-and-add 0 #acquire() 1975797
lock: kmem: #fetch-and-add 0 #acquire() 1941300
lock: kmem: #fetch-and-add 0 #acquire() 53
lock: bcache: #fetch-and-add 0 #acquire() 2166
lock: bcache: #fetch-and-add 0 #acquire() 4183
lock: bcache: #fetch-and-add 0 #acquire() 2328
lock: bcache: #fetch-and-add 0 #acquire() 4330
lock: bcache: #fetch-and-add 0 #acquire() 4380
lock: bcache: #fetch-and-add 0 #acquire() 6368
lock: bcache: #fetch-and-add 0 #acquire() 6778
lock: bcache: #fetch-and-add 0 #acquire() 6729
lock: bcache: #fetch-and-add 0 #acquire() 8052
lock: bcache: #fetch-and-add 0 #acquire() 6224
lock: bcache: #fetch-and-add 0 #acquire() 6222
lock: bcache: #fetch-and-add 0 #acquire() 4169
lock: bcache: #fetch-and-add 0 #acquire() 4169
--- top 5 contended locks:
lock: proc: #fetch-and-add 9260796 #acquire() 4076147
lock: proc: #fetch-and-add 266742 #acquire() 4072297
lock: virtio_disk: #fetch-and-add 140459 #acquire() 1194
lock: proc: #fetch-and-add 67954 #acquire() 4072397
lock: proc: #fetch-and-add 49967 #acquire() 4072289
tot= 0
test0: OK
start test1
test1 OK
$ 
```

评分：

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6-lab-2020-reports
```

```
ring.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/stats.o kernel/sprintf.o
riscv64-unknown-elf-ld: warning: cannot find entry symbol _entry; defaulting to 0000000080000000
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/jinnian/Desktop/OS/xv6-lab-2020-reports'
== Test running kalloc test ==
$ make qemu-gdb
(73.9s)
== 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 (7.3s)
== Test running bcachetest ==
$ make qemu-gdb
(3.3s)
== Test bcachetest: test0 ==
bcachetest: test0: OK
== Test bcachetest: test1 ==
bcachetest: test1: OK
== Test user tests ==
$ make qemu-gdb
user tests: OK (83.2s)
== Test time ==
time: FAIL
    Cannot read time.txt
Score: 69/70
make: *** [Makefile:317: grade] Error 1
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-lab-2020-reports$ 
```

Lab 9 File System

写在前面：

i 节点这个术语可以有两个意思。它可以指的是磁盘上的记录文件大小、数据块扇区号的数据结构。也可以指内存中的一个 *i* 节点，它包含了一个磁盘上 *i* 节点的拷贝，以及一些内核需要的附加信息。

所有的磁盘上的 *i* 节点都被打包在一个称为 *i* 节点块的连续区域中。每一个 *i* 节点的大小都是一样的，所以对于一个给定的数字n，很容易找到磁盘上对应的 *i* 节点。事实上这个给定的数字就是操作系统中 *i* 节点的编号。

磁盘上的 *i* 节点由结构体 `dinode` 定义

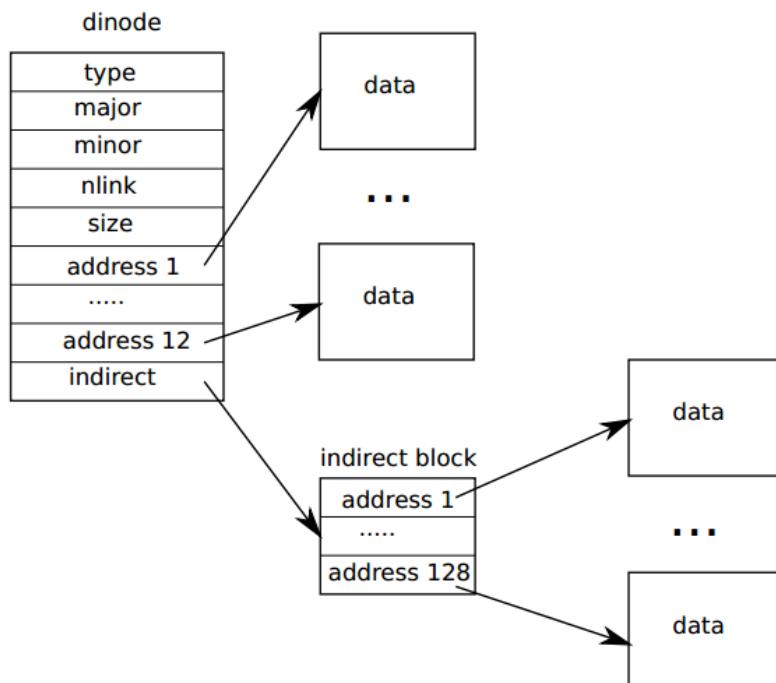


Figure 6-4. The representation of a file on disk.

- `type` 域用来区分文件、目录和特殊文件的 *i* 节点。如果 `type` 是0的话就意味着这是一个空闲的 *i* 节点。
- `nlink` 域用来记录指向了这一个 *i* 节点的目录项，这是用于判断一个 *i* 节点是否应该被释放的。
- `size` 域记录了文件的字节数。
- `addrs` 数组用于这个文件的数据块的块号。

内核在内存中维护活动的 *i* 节点。结构体 `inode` 是磁盘中的结构体 `dinode` 在内存中的拷贝。内核只会在有 C 指针指向一个 *i* 节点的时候才会把这个 *i* 节点保存在内存中。`ref` 域用于统计有多少个 C 指针指向它。如果 `ref` 变为0，内核就会丢掉这个 *i* 节点。`iget` 和 `iput` 两个函数申请和释放 *i* 节点指针，修改引用计数。*i* 节点指针可能从文件描述符产生，从当前工作目录产生，也有可能从一些内核代码如 `exec` 中产生。

Assignment 1 —— Large files

在本任务中，增加 xv6 文件的大小上限。目前，xv6 文件限制为268个块或 `268*BSIZE` 字节（在xv6中 `BSIZE` 为1024）。此限制来自：一个xv6 inode包含12个“直接”块号和一个“间接”块号，“一级间接”块指一个最多可容纳256个块号的块，总共 $12+256=268$ 个块。

`bigfile`命令可以创建最长的文件，并报告其大小：

```
1 $ bigfile
2 ..
3 wrote 268 blocks
4 bigfile: file is too small
```

测试失败，因为 `bigfile` 希望能够创建一个包含65803个块的文件，但未修改的xv6 将文件限制为268个块。

更改xv6文件系统代码，以支持每个inode中可包含256个一级间接块地址的“二级间接”块，每个一级间接块最多可以包含256个数据块地址。结果将是一个文件将能够包含多达65803个块，或 $256*256+256+11=65803$ 个块（11而不是12，因为我们将为二级间接块牺牲一个直接块号）。

Preliminaries:

`mkfs` 程序创建xv6文件系统磁盘映像，并确定文件系统的总块数；此大小由 *kernel/param.h* 中的 `FSSIZE` 控制。

```
1 #define FSSIZE      2000000 // size of file system in blocks
```

该实验室存储库中的 `FSSIZE` 设置为200000个块。在 `make` 输出中看到来自 `mkfs/mkfs` 的以下输出：

```
1 nmeta 70 (boot, super, log blocks 30 inode blocks 13, bitmap blocks
25) blocks 199930 total 200000
```

这一行描述了 `mkfs/mkfs` 构建的文件系统：它有70个元数据块（用于描述文件系统的块）和199930个数据块，总计200000个块。

如果在实验期间的任何时候，发现必须从头开始重建文件系统，可以运行 `make clean`，强制 `make` 重建 `fs.img`。

磁盘索引节点的格式由 `fs.h` 中的 `struct dinode` 定义，在磁盘上查找文件数据的代码位于 `fs.c` 的 `bmap()` 中。在读取和写入文件时都会调用 `bmap()`。写入时，`bmap()` 会根据需要分配新块以保存文件内容，如果需要，还会分配间接块以保存块地址。`bmap()` 处理两种类型的块编号。`bn` 参数是一个“逻辑块号”——文件中相对于文件开头的块号。`ip-addrs[]` 中的块号和 `bread()` 的参数都是磁盘块号。`bmap()` 将文件的逻辑块号映射到磁盘块号。

Question:

修改 `bmap()`，以便除了直接块和一级间接块之外，它还实现二级间接块。只需要有11个直接块，而不是12个，为新的二级间接块腾出空间；不允许更改磁盘 `inode` 的大小。`ip->addrs[]` 的前11个元素应该是直接块；第12个应该是一个一级间接块（与当前的一样）；13号应该是你的新二级间接块。当 `bigfile` 写入65803个块并成功运行 `usertests` 时，此练习完成。

Solution:

修改 `kernel/fs.h` 中的宏定义

```
1 #define NDIRECT 11
2 #define NINDIRECT (BSIZE / sizeof(uint))
3 #define NDBINDIRECT ((BSIZE / sizeof(uint)) * (BSIZE /
4 #define MAXFILE (NDIRECT + NINDIRECT + NDBINDIRECT)
```

并修改 `fs.h` 和 `file.h` 中的 `struct inode` 中的 `addrs`

```
1 uint addrs[NDIRECT+1+1];
```

修改 `kernel/fs.c` 中的 `bmap`，增加 `bn < NDBINDIRECT` 情况下的两级映射。先要让 `bn` 减掉上一级的 `NINDIRECT`，DOUBLE INDIRECT BLOCK 映射的第一级 index 应为 `bn/NINDIRECT`，第二级 index 为 `bn%NINDIRECT`。依次读取每一级 block 中的 `addr` (`blockno`)，当不存在时进行 `malloc`。

```
1 # kernel/fs.c/bmap
2 bn -= NINDIRECT;
3
4 if (bn < NDBINDIRECT) {
5     uint index_1 = bn/NINDIRECT;
6     uint index_2 = bn%NINDIRECT;
7     if ((addr = ip->addrs[NDIRECT+1]) == 0)
8         ip->addrs[NDIRECT+1] = addr = malloc(ip->dev);
9     bp = bread(ip->dev, addr);
10    a = (uint*)bp->data;
11    if ((addr = a[index_1]) == 0) {
12        a[index_1] = addr = malloc(ip->dev);
13        log_write(bp);
14    }
15    brelse(bp);
16    bp = bread(ip->dev, addr);
17    a = (uint*)bp->data;
18    if ((addr = a[index_2]) == 0) {
19        a[index_2] = addr = malloc(ip->dev);
20        log_write(bp);
```

```
21     }
22     brelse(bp);
23     return addr;
24 }
```

| 在 `itrunc` 中，也要释放掉DOUBLE INDIRECT BLOCK中每一级的block

```
1 # kernel/fs.c/itrunc
2 if (ip->addrs[NDIRECT+1]) {
3     bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
4     a = (uint*)bp->data;
5     for (j = 0; j < NINDIRECT; j++) {
6         if (a[j]) {
7             bp2 = bread(ip->dev, a[j]);
8             b = (uint*)bp2->data;
9             for (i = 0; i < NINDIRECT; i++) {
10                 if (b[i]) {
11                     bfree(ip->dev, b[i]);
12                 }
13             }
14             brelse(bp2);
15             bfree(ip->dev, a[j]);
16         }
17     }
18     brelse(bp);
19     bfree(ip->dev, ip->addrs[NDIRECT+1]);
20     ip->addrs[NDIRECT+1] = 0;
21 }
```

运行结果：

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6-lab-2020-reports
```

riscv64-unknown-elf-objdump -t user/_zombie | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^\$/d' > user/zombie.sym
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_FS -DLAB_FS -MD -mcmmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/symlinktest.o user/symlinktest.c
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_symlinktest user/symlinktest.o user/ulib.o user/usys.o user/printfo.o user/umalloc.o
riscv64-unknown-elf-objdump -S user/_symlinktest > user/symlinktest.asm
riscv64-unknown-elf-objdump -t user/_symlinktest | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^\$/d' > user/symlinktest.sym
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_FS -DLAB_FS -MD -mcmmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/bigfile.o user/bigfile.c
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_bigfile user/bigfile.o user/ulib.o user/usys.o user/printfo.o user/umalloc.o
riscv64-unknown-elf-objdump -S user/_bigfile > user/bigfile.asm
riscv64-unknown-elf-objdump -t user/_bigfile | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^\$/d' > user/bigfile.sym
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_l test user/_bigfile
nmeta 70 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 25) blocks 199930 total 200000
balloc: first 677 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
\$ bigfile

wrote 65803 blocks
bigfile done; ok
\$

Assignment 2 — Symbolic links

本任务将向xv6添加符号链接。符号链接（或软链接）是指按路径名链接的文件；当一个符号链接打开时，内核跟随该链接指向引用的文件。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不支持多个设备。

Question:

实现 `symlink(char *target, char *path)` 系统调用，该调用在引用由 `target` 命名的文件的路径处创建一个新的符号链接。有关更多信息，请参阅 `symlink` 手册页（注：执行 `man symlink`）。

Hints:

- 首先，为 `symlink` 创建一个新的系统调用号，在 `user/usys.pl`，`user/user.h` 中添加一个入口，在 `kernel/sysfile.c` 中实现一个空的 `sys_symlink`。
 - 将新文件类型 (`T_SYMLINK`) 添加到 `kernel/stat.h` 以表示符号链接。
 - 向 `kernel/fcntl.h` 添加一个新标志 (`O_NOFOLLOW`)，可与 `open` 系统调用一起使用。请注意，传递给 `open` 的标志是使用按位 OR 运算符组合的，因此新标志不应与任何现有标志重叠。
 - 实现 `symlink(target, path)` 系统调用以在指向目标的路径上创建一个新的符号链接。请注意，系统调用成功时不需要存在目标。你需要选择某个位置来存储符号链接的目标路径，例如，在 `inode` 的数据块中。`symlink` 应该返回一个表示成功 (0) 或失败 (-1) 的整数，类似于 `link` 和 `unlink`。

- 修改 `open` 系统调用以处理路径引用符号链接的情况。如果文件不存在，则打开必须失败。当进程在 `open` 标志中指定 `O_NOFOLLOW` 时，`open` 应该打开符号链接（而不是跟随符号链接）。
- 如果链接文件也是符号链接，则必须递归地跟随它，直到到达非链接文件。如果链接形成循环，则必须返回错误代码。如果链接的深度达到某个阈值（例如，10），可以通过返回错误代码来近似此值。
- 其他系统调用（例如，链接和取消链接）不得遵循符号链接；这些系统调用对符号链接本身进行操作。

Solution：

添加系统调用的方法同Lab 2，这里不再赘述。

在 `kernel/fcntl.h` 中添加 `open` 的 flag `O_NOFOLLOW`，该 flag 不能和其他 flag 的位重叠

```
1 #define O_NOFOLLOW 0x010
```

在 `kernel/stat.h` 中添加 inode 类型 `T_SYMLINK`

```
1 #define T_SYMLINK 4 // Symlink
```

在 `kernel/sysfile.c` 中，添加 `symlink` 的实现：首先要判断是否存在 `path` 所代表的 inode，如果不存在就用 `create` 添加一个 `T_SYMLINK` 类型的 inode。在 inode 的最后添加需要软链接到的 `target` 的路径名称

```
1 # kernel/sysfile.c
2 uint64 sys_symlink(void) {
3     char target[MAXPATH];
4     char path[MAXPATH];
5     struct inode *ip;
6     // char test[MAXPATH];
7
8     if (argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0) {
9         return -1;
10    }
11    begin_op();
12    if ((ip = namei(path)) == 0) {
13        // the path inode does not exist
14        ip = create(path, T_SYMLINK, 0, 0);
15        iunlock(ip);
16    }
17    ilock(ip);
18    // write the target path name into the end of inode
19    if (writei(ip, 0, (uint64)target, ip->size, MAXPATH) != MAXPATH)
20    {
```

```
20     panic("symlink");
21 }
22 iunlockput(ip);
23 end_op();
24 return 0;
25 }
```

修改 open，添加对 T_SYMLINK 类型文件的处理方法:要判断打开 open 的flag是否为 NOFOLLOW，如果是的话，就不打开软链接所指向的文件，否则需要将inode递归地替换为软链接指向的文件，直到最终的inode类型不是 T_SYMLINK 为止。整个递归的深度不能超过10，否则报错。

```
1 if (ip->type == T_SYMLINK) {
2     if ((omode & O_NOFOLLOW) == 0) {
3         // recursively follow symlink
4         int count = 0;
5         char sympath[MAXPATH];
6         while (1) {
7             if (count >= 10) {
8                 iunlockput(ip);
9                 end_op();
10                return -1;
11            }
12            // read the path name from inode
13            if (readi(ip, 0, (uint64)sympath, ip->size-MAXPATH,
14 MAXPATH) != MAXPATH) {
15                panic("open symlink");
16            }
17            iunlockput(ip);
18            if ((ip = namei(sympath)) == 0) {
19                // could not find this file
20                end_op();
21                return -1;
22            }
23            ilock(ip);
24            if (ip->type != T_SYMLINK) {
25                break;
26            }
27            count++;
28        }
29 }
```

运行结果：

评分：

```
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6-lab-2020-reports
```

kernel/plic.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_FS -DLAB_FS -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/virtio_disk.o kernel/virtio_disk.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_FS -DLAB_FS -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -march=rv64g -nostdin c -I . -Ikernel -c user/initcode.S -o user/initcode.o
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e start -Ttext 0 -o user/initcode.out user/user/initcode.o
riscv64-unknown-elf-objcopy -S -O binary user/initcode.out user/user/initcode
riscv64-unknown-elf-objdump -S user/initcode.o > user/initcode.asm
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.lds -o kernel/kernel kernel/entry.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/kalloc.o kernel/spinlock.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o
riscv64-unknown-elf-ld: warning: cannot find entry symbol _entry; defaulting to 0000000000000000
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /\\$/d' > kernel/kernel.sym
make[1]: 离开目录“/home/jinnian/Desktop/OS/xv6-lab-2020-reports”
== Test running bigfile ==
\$ make qemu-gdb
running bigfile: **OK** (65.8s)
== Test running symlinktest ==
\$ make qemu-gdb
(0.5s)
== Test symlinktest: symlinks ==
symlinktest: symlinks: **OK**
== Test symlinktest: concurrent symlinks ==
symlinktest: concurrent symlinks: **OK**
== Test usertests ==
\$ make qemu-gdb
usertests: **OK** (137.4s)
== Test time ==
time: **FAIL**
 Cannot read time.txt
Score: 99/100
make: *** [Makefile:318: grade] 错误 1
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021: ~/Desktop/OS/xv6-lab-2020-reports\$

Lab 10 Mmap

写在前面

mmap：将一个文件或者其他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。

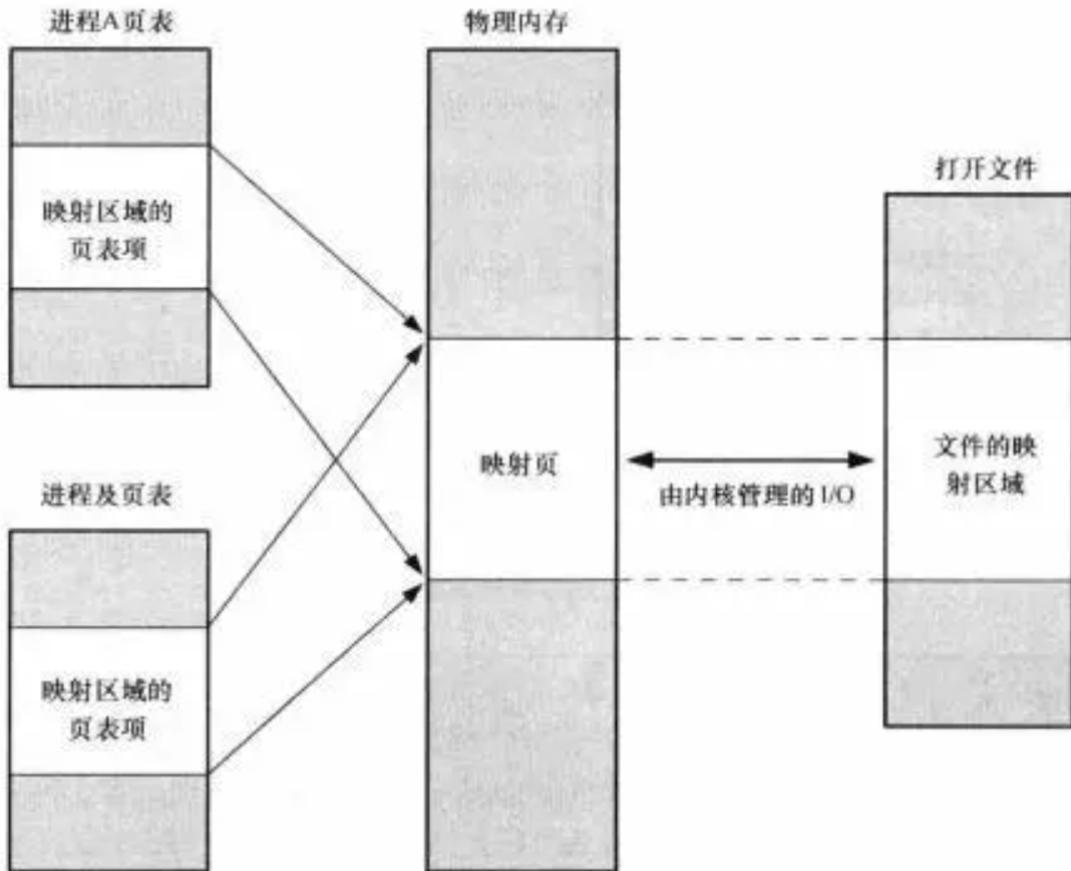


图 49-2：两个进程和一个文件的同一区域的共享映射

@1

为什么要建立文件磁盘地址和进程虚拟地址空间的映射？因为常规的文件系统操作是用户态发起 `read` syscall，然后在 buffer cache 中查找是否有相应的数据，如果没有就从磁盘中拷贝数据到 buffer cache 中，因为 buffer cache 处在内核态，因此需要将 buffer cache `copyout` 到用户进程的虚拟内存中，这就需要 2 次拷贝操作，而在 mmap 中只需要直接将文件数据拷贝到对应的用户空间虚拟内存即可。

Assignment 1 —— mmap

`mmap` 和 `munmap` 系统调用允许 UNIX 程序对其地址空间进行详细控制。它们可用于在进程之间共享内存，将文件映射到进程地址空间，并作为用户级页面错误方案的一部分，如本课程中讨论的垃圾收集算法。本实验将把 `mmap` 和 `munmap` 添加到 xv6 中，重点关注内存映射文件 (memory-mapped files)。

```
1 $ man 2 mmap  
2 void *mmap(void *addr, size_t length, int prot, int flags, int fd,  
off_t offset);
```

可以通过多种方式调用 `mmap`，但本实验只需要与内存映射文件相关的功能子集。假设 `addr` 始终为零，这意味着内核应该决定映射文件的虚拟地址。`mmap` 返回该地址，如果失败则返回 `0xffffffffffffffffffff`。`length` 是要映射的字节，它可能与文件的长度不同。`prot` 指示内存是否应映射为可读、可写，以及/或者可执行的；可以认为 `prot` 是 `PROT_READ` 或 `PROT_WRITE` 或两者兼有。`flags` 要么是 `MAP_SHARED`（映射内存的修改应写回文件），要么是 `MAP_PRIVATE`（映射内存的修改不应写回文件）。不必在 `flags` 中实现任何其他位。`fd` 是要映射的文件的打开文件描述符。可以假定 `offset` 为零（它是要映射的文件的起点）。允许进程映射同一个 `MAP_SHARED` 文件而不共享物理页面。

`munmap(addr, length)` 应删除指定地址范围内的 `mmap` 映射。如果进程修改了内存并将其映射为 `MAP_SHARED`，则应首先将修改写入文件。`munmap` 调用可能只覆盖 `mmap` 区域的一部分，但您可以认为它取消映射的位置要么在区域起始位置，要么在区域结束位置，要么就是整个区域(但不会在区域中间“打洞”)。

Question:

实现 `mmap` 和 `munmap` 的功能以通过 `mmaptest` 的测试。

Hints:

- 首先，向 `UPROGS` 添加 `_mmaptest`，以及 `mmap` 和 `munmap` 系统调用，以便让 `user/mmaptest.c` 进行编译。现在，只需从 `mmap` 和 `munmap` 返回错误。
- 惰性地填写页表，以响应页错误。也就是说，`mmap` 不应该分配物理内存或读取文件。相反，在 `usertrap` 中（或由 `usertrap` 调用）的页面错误处理代码中执行此操作，就像在 `lazy page allocation` 实验中一样。惰性分配的原因是确保大文件的 `mmap` 是快速的，并且比物理内存大的文件的 `mmap` 是可能的。
- 跟踪 `mmap` 为每个进程映射的内容。定义 `VMA`（虚拟内存区域）结构体，记录 `mmap` 创建的虚拟内存范围的地址、长度、权限、文件等。由于 `xv6` 内核中没有内存分配器，因此可以声明一个固定大小的 `VMA` 数组，并根据需要从该数组进行分配。大小为 16 应该就足够了。
- 实现 `mmap`：在进程的地址空间中找到一个未使用的区域来映射文件，并将 `VMA` 添加到进程的映射区域表中。`VMA` 应该包含指向映射文件对应 `struct file` 的指针；`mmap` 应该增加文件的引用计数，以便在文件关闭时结构体不会消失（提示：请参阅 `filedup`）。运行 `mmaptest`：第一次 `mmap` 应该成功，但是第一次访问被 `mmap` 的内存将导致页面错误并终止 `mmaptest`。
- 添加代码以导致在 `mmap` 的区域中产生页面错误，从而分配一页物理内存，将 4096 字节的相关文件读入该页面，并将其映射到用户地址空间。使用 `readi` 读取文件，它接受一个偏移量参数，在该偏移处读取文件（但必须 `lock/unlock` 传递给 `readi` 的索引结点）。不要忘记在页面上正确设置权限。运行 `mmaptest`；它应该到达第一个 `munmap`。
- 实现 `munmap`：找到地址范围的 `VMA` 并取消映射指定页面（提示：使用 `uvmunmap`）。如果 `munmap` 删除了先前 `mmap` 的所有页面，它应该减少相应 `struct file` 的引用计

数。如果未映射的页面已被修改，并且文件已映射到 `MAP_SHARED`，请将页面写回该文件。查看 `filewrite` 以获得灵感。

- 理想情况下，您的实现将只写回程序实际修改的 `MAP_SHARED` 页面。RISC-V PTE中的脏位（D）表示是否已写入页面。但是，`mmaptest` 不检查非脏页是否没有回写；因此，您可以不用看 D 位就写回页面。
- 修改 `exit` 将进程的已映射区域取消映射，就像调用了 `munmap` 一样。运行 `mmaptest`；`mmap_test` 应该通过，但可能不会通过 `fork_test`。
- 修改 `fork` 以确保子对象具有与父对象相同的映射区域。不要忘记增加VMA的 `struct file` 的引用计数。在子进程的页面错误处理程序中，可以分配新的物理页面，而不是与父级共享页面。后者会更酷，但需要更多的实施工作。运行 `mmaptest`；它应该通过 `mmap_test` 和 `fork_test`。

Solution：

在 `kernel/proc.h` 中添加对 `vma` 结构体的定义，每个进程中最多可以有16个VMA。

```
1 #define MAXVMA 16
2 struct vma {
3     int valid;                      // whether the vma is valid
4     uint64 addr;                    // starting virtual address of vma
5     int len;                        // length of vma, unit: bytes
6     int prot;                       // permission
7     int flags;                      // flag
8     struct file *f;                // pointer to mapped file
9     int off;                        // offset of the valid mapped address
10    int valid_len;                 // length of the valid mapped address
11 };
12 struct vma procvma[MAXVMA]; // process vma
```

`kernel/sysfile.c` 中添加注册 `sys_mmap`，在 `mmap` 中先不要分配物理内存，只是完成对 `vma` 结构体的写入。找到新的空闲内存地址 `p->sz` 作为 `vma` 返回的地址并 `p->sz+=len`，让后面需要使用这个返回的内存地址时陷入缺页异常，在 `trap` 中再分配物理内存完成lazy allocation

```
1 uint64
2 sys_mmap(void)
3 {
4     int length, prot, flags, fd;
5     struct file *f;
6     if (argint(1, &length)<0 || argint(2, &prot)<0 || argint(3,
7         &flags)<0 || argfd(4, &fd, &f)<0) {
8         return -1;
9     }
10    if (!f->writable && (prot & PROT_WRITE) && (flags & MAP_SHARED))
11        return -1; // to assert that readonly file could not be opened
12        with PROT_WRITE && MAP_SHARED flags
```

```

10 struct proc *p = myproc();
11 struct vma *pvma = p->procvma;
12 for (int i = 0; i < MAXVMA; i++) {
13     if(pvma[i].valid == 0) {
14         pvma[i].addr = p->sz;
15         pvma[i].f = filedup(f); // increment the refcount for f
16         pvma[i].flags = flags;
17         pvma[i].len = PGROUNDDOWN(length);
18         pvma[i].prot = prot;
19         pvma[i].valid = 1;
20         pvma[i].off = 0;
21         pvma[i].valid_len = pvma[i].len;
22         p->sz += pvma[i].len;
23         return pvma[i].addr;
24     }
25 }
26 return -1;
27 }
```

在 `kernel/trap.c` 中添加 lazy allocation: 需要判断 page fault 的地址是合理的, 如果 fault 的地址低于了当前进程的栈底(`p->trapframe->sp`)或者高于等于当前进程的堆顶(`p->sz`)就说明是不合理的, 需要进入 exception。然后判断当前 fault 的地址是在哪一个 VMA 的合法范围内, 找到这个 VMA 后分配一页物理页, 并用 `mappages` 将这一页物理页映射到 fault 的用户内存中, 然后用 `readi` 打开需要映射的文件, 将对应的文件内容用 `readi` 放入这一页内存中去。

```

1 } else if((which_dev = devintr()) != 0){
2     // ok
3 } else if (r_scause() == 13 || r_scause() == 15) {
4     uint64 va = r_stval();
5     if (va < p->trapframe->sp || va >= p->sz) {
6         goto exception;
7     }
8     struct vma *pvma = p->procvma;
9     va = PGROUNDDOWN(va);
10
11    for (int i = 0; i < MAXVMA; i++) {
12        if (pvma[i].valid && va >= pvma[i].addr + pvma[i].off && va
13 < pvma[i].addr + pvma[i].off + pvma[i].valid_len) {
14            // allocate one page in the physical memory
15            char *mem = kalloc();
16            if (mem == 0) goto exception;
17            memset(mem, 0, PGSIZE);
18            int flag = (pvma[i].prot << 1) | PTE_U; // PTE_R = 2 and
PROT_READ = 1
19            if (mappages(p->pagetable, va, PGSIZE, (uint64)mem, flag)
20 != 0) {
```

```

19         kfree(mem);
20         goto exception;
21     }
22     int off = va - pvma[i].addr;
23     ilock(pvma[i].f->ip);
24     readi(pvma[i].f->ip, 1, va, off, PGSIZE);
25     iunlock(pvma[i].f->ip);
26     break;
27 }
28 }
29 } else {
30     exception:
31     printf("usertrap(): unexpected scause %p pid=%d\n",
32             r_scause(), p->pid);
33     printf("          sepc=%p stval=%p\n", r_sepc(), r_stval());
34     p->killed = 1;
35 }
```

实现 `munmap`：首先需要找到对应的vma，然后根据unmap的大小和起点的不同进行讨论。如果是从vma有效部分的起点开始，当整个vma都被unmap掉时，需要标记这个打开的文件被关闭（但是现在还不能关闭，因为后面可能需要写回硬盘中的文件），将当前的vma设置为invalid，减小 `p->sz`（这一部分可能有点问题，因为如果是unmap中间的vma的话不需要减小 `p->sz`，但是本实验中是可以通过测试的）。如果只是部分vma被unmap，则修改vma的 `off` 和 `valid_len`，如果是从中间部分开始被unmap一直到结尾，则不需要修改 `off`，只需要修改 `valid_len` 和需要被 `uv munmap` 的 `length`。然后判断是否是 `MAP_SHARED`，如果是就用 `_filewrite` 写回原文件，这里是对 `filewrite` 函数进行了修改，使其能够从某个offset开始写。

```

1 int sys_munmap(void)
2 {
3     uint64 addr;
4     int length;
5     if (argaddr(0, &addr) < 0 || argint(1, &length) < 0) {
6         return -1;
7     }
8     struct proc *p = myproc();
9     struct vma *pvma = p->procvma;
10    int close = 0;
11    // find the corresponding vma
12    for (int i = 0; i < MAXVA; i++) {
13        if (pvma[i].valid && addr ≥ pvma[i].addr && addr <
14            pvma[i].addr + pvma[i].len) {
15            addr = PGROUNDDOWN(addr);
16            if (addr == pvma[i].addr + pvma[i].off) {
17                // starting at begin of the valid address of vma
18                if (length ≥ pvma[i].valid_len) {
```

```

18         // whole vma is unmapped
19         pvma[i].valid = 0;
20         length = pvma[i].valid_len;
21         close = 1;
22         p→sz -= pvma[i].len;
23     } else {
24         pvma[i].off += length;
25         pvma[i].valid_len -= length;
26     }
27 } else {
28     // starting at middle, should unmap until the end
29     length = pvma[i].addr + pvma[i].off + pvma[i].valid_len -
30     addr;
31     pvma[i].valid_len -= length;
32 }
33 if (pvma[i].flags & MAP_SHARED) {
34     // write the page back to the file
35     if (_filewrite(pvma[i].f, addr, length, addr -
36     pvma[i].addr) == -1) return -1;
37     uvmunmap(p→pagetable, addr, PGROUNDUP(length)/PGSIZE, 0);
38     if (close) fileclose(pvma[i].f);
39     return 0;
40 }
41 return -1;
42 }

43
44 int
45 _filewrite(struct file *f, uint64 addr, int n, uint off) {
46     int r, ret = 0;
47
48     if(f→writable == 0)
49         return -1;
50
51     if(f→type == FD_PIPE){
52         ret = pipewrite(f→pipe, addr, n);
53     } else if(f→type == FD_DEVICE){
54         if(f→major < 0 || f→major ≥ NDEV || !devsw[f→major].write)
55             return -1;
56         ret = devsw[f→major].write(1, addr, n);
57     } else if(f→type == FD_INODE){
58         // write a few blocks at a time to avoid exceeding
59         // the maximum log transaction size, including
60         // i-node, indirect block, allocation blocks,
61         // and 2 blocks of slop for non-aligned writes.
62         // this really belongs lower down, since writei()
63         // might be writing a device like the console.

```

```

64     int max = ((MAXOPBLOCKS-1-1-2) / 2) * BSIZE;
65     int i = 0;
66     while(i < n){
67         int n1 = n - i;
68         if(n1 > max)
69             n1 = max;
70
71         begin_op();
72         ilock(f->ip);
73         if ((r = writei(f->ip, 1, addr + i, off, n1)) > 0)
74             off += r;
75         iunlock(f->ip);
76         end_op();
77
78         if(r != n1){
79             // error from writei
80             break;
81         }
82         i += r;
83     }
84     // ret = (i == n ? n : -1);
85 } else {
86     panic("filewrite");
87 }
88 return ret;
89 }
```

最后 `uvmunmap` 掉这个vma中对应的虚拟内存（这里释不释放物理内存都可以通过测试），如果需要关闭文件就调用 `fclose`。

最后，由于 `p->sz` 以内的内存不是都有对应的映射，因此可能会造成 `uvmunmap` 和 `uvmcopy` 出现panic。直接将对应的panic注释掉并continue就可以

```

1 // uvmunmap
2 // if(*pte & PTE_V) == 0)
3     // panic("uvmunmap: not mapped");
4
5 // uvmcopy
6 if(*pte & PTE_V) == 0)
7     panic("uvmcopy: page not present");
8     continue;
9 // panic("uvmcopy: page not present");
```

测试过程中发现 `kfree` 中似乎会试图杀掉0这个物理内存，没有搞明白是在哪里杀的，因此直接修改 `kfree`，当 `pa==0` 时直接return避免panic

运行结果：

```
Score: 99/100
make: *** [Makefile:318: grade] 错误 1
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-lab-2020-reports$ git status
位于分支 fs
您的分支与上游分支 'origin/fs' 一致。

无文件要提交，干净的工作区
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-lab-2020-reports$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ mmptest
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
mmptest: ALL OK
fork_test starting
fork_test OK
mmptest: all tests succeeded
$ QEMU: Terminated
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-lab-2020-reports$
```

评分：

```
riscv64-unknown-elf-objdump -S user/initcode.o > user/initcode.asm
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.lds -o kernel/kernel kernel/kernel/entry.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/kalloc.o kernel/spinlock.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o
riscv64-unknown-elf-ld: warning: cannot find entry symbol _entry; defaulting to 0000000080000000
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
make[1]: 离开目录“/home/jinnian/Desktop/OS/xv6-lab-2020-reports”
== Test running mmptest ==
$ make qemu-gdb
(3.6s)
== Test mmptest: mmap f ==
mmptest: mmap f: OK
== Test mmptest: mmap private ==
mmptest: mmap private: OK
== Test mmptest: mmap read-only ==
mmptest: mmap read-only: OK
== Test mmptest: mmap read/write ==
mmptest: mmap read/write: OK
== Test mmptest: mmap dirty ==
mmptest: mmap dirty: OK
== Test mmptest: not-mapped unmap ==
mmptest: not-mapped unmap: OK
== Test mmptest: two files ==
mmptest: two files: OK
== Test mmptest: fork_test ==
mmptest: fork_test: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (78.5s)
== Test time ==
time: FAIL
    Cannot read time.txt
Score: 139/140
make: *** [Makefile:318: grade] 错误 1
jinnian@jinnian-Lenovo-XiaoXin-15ITL-2021:~/Desktop/OS/xv6-lab-2020-reports$
```

