

给出实现方案，提交作品源代码和开发文档。阶段性技术报告和开发状态演示

构思并实现一个与操作系统相关的系统或模块，知识：编译技术、操作系统、计算机体系结构

OS功能、性能、OS与硬件结合的软硬协同优化系统、OS调试工具、支持OS的模拟器、OB Bug/漏洞分析工具等

全国赛成绩100=第一阶段0.2+第二阶段0.8

第一阶段项目/代码展示0.1，文档0.1；第二阶段项目/代码执行0.15，文档0.25，演示答辩0.4

赛题：<https://github.com/oscomp>，具有“projXXX-YYY”名称的Public类型的仓库

gitlab仓库：[https://gitlab.eduxiji.net/users/sign\\_in](https://gitlab.eduxiji.net/users/sign_in)，账号：T202410574992849

项目仓库: project2210132-237569.tgz

大赛评测用的镜像：os\_contest\_v6.1.tar.gz文件

gitlab仓库默认是private，按大赛要求需要改成public：设置-通用-可用性

选择 mit6.S081 项目，有人把官方教材翻译成了中文版，见：<https://github.com/duguosheng/6.S081-All-in-one>

## 环境部署

[环境部署官方参考](#)

[安装wsl参考](#)

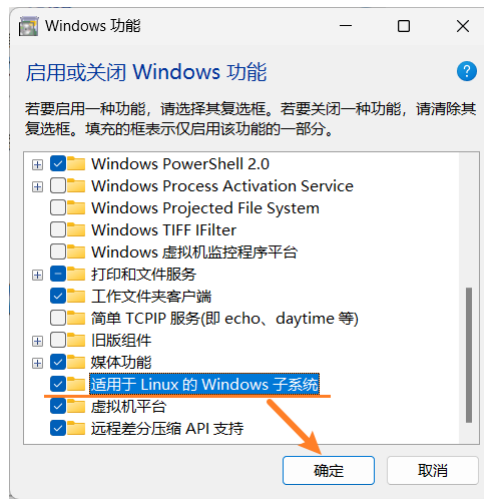
在cmd中使用管理员模式，执行

```
1 wsl --install
```

```
C:\Windows\System32>wsl --install
正在安装：虚拟机平台
已安装 虚拟机平台。
正在安装：适用于 Linux 的 Windows 子系统
已安装 适用于 Linux 的 Windows 子系统。
正在安装：Ubuntu
已安装 Ubuntu。
请求的操作成功。直到重新启动系统前更改将不会生效。
```

从Microsoft Store中获取Ubuntu 20.04.6 LTS





重启生效，创建账户和密码，即成功安装

```
Ubuntu 20.04.6 LTS
Installing, this may take a few minutes...
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: starsinhands
New password:
Retype new password:
passwd: password updated successfully
操作成功完成。
```

在管理员 power shell 启动 wsl。

```
1 | wsl -u root # 不然非管理员用户登录
```

安装必要依赖：

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

访问 wsl 根目录，在文件资源管理器输入路径：\\wsl\$\\Ubuntu-20.04 (具体看自己的 wsl 版本，可以通过 `wsl --list` 查看)

随便找一个目录 clone 并 cd，抓取 xv6 代码：[参考](#)

```
1 | git clone git://g.csail.mit.edu/xv6-labs-2021
2 | cd xv6-labs-2021
```

编译：

```
1 | git checkout util #切换分支
2 | mingw32-make qemu #官方是make qemu，但我使用make会报错
```

在执行mingw32-make qemu的时候报错，先直接sudo apt install gcc-riscv64-unknown-elf，确保有riscv64-unknown-elf-gcc --version qemu-system-riscv64 --version这两个组件，再执行make

之后进入一个 xv6 CLI，可以执行：[参考官方](#)

```
1 | ls
```

退出方法：先按 `ctrl+a`，然后直接按 `x` (不是一边按 `ctrl+a` 一边 `ctrl+x` 或 `ctrl+a+x`)。

vscode 配置，安装插件 `remote-wsl` (或名为 `wsl`，特征为蓝色背景白色企鹅)。参考

在 wsl terminal 内在项目根目录执行：`code .`，弹出 vsc 窗口。

更改文件权限，以自己的为例，这样才能让 vsc 和 windows 文件资源管理器可以写：

```
1 | sudo chmod -R 777 /home/starsinhands/xv6-labs-2021
```

因为实际上 wsl 内的东西不在一个逻辑机，所以 git 比较麻烦，必须用 wsl terminal 来 add, commit(本机的 git bash 的话— `git add .` 就把东西全删了，理由未知)，然后因为用户要用到本机的，所以在 git bash 里搞 `git push`。

对 vsc，注意到在 wsl 和在本机插件独立，所以给 wsl 装一个 `C/C++` 插件。安装完成的话，对要调用的函数，进行 ctrl 单机能定位到出处。

部署完毕，即通过 要求的 boot xv6 任务。

注意到 wsl 内需要配置 git 用户名和邮箱，不然 git log 看到的不是自己，github 也没有自己的 commits 记录。将本机公钥私钥覆盖虚拟机的，如管理员就把 `id_rsa`, `id_rsa.pub` 丢到 `/root/.ssh` 里，可以先通过 vscode WSL 插件丢到项目目录，然后 `mv` 过去即可。

## 开发任务

### lab1 util

参考 实现 util

#### sleep

##### 要求

官方要求：Implement the UNIX program `sleep` for xv6; your `sleep` should pause for a user-specified number of ticks. A tick is a notion of time defined by the xv6 kernel, namely the time between two interrupts from the timer chip. Your solution should be in the file `user/sleep.c`.

**1.进程和内存：**每个进程都拥有自己的用户空间内存以及内核空间状态，当进程不再执行时，xv6会存储和这些进程有关的CPU寄存器到下一次运行这些进程。kernel中一个进程有唯一的PID。常用的syscall：

- `fork`：原型是 `int fork()`。作用是让一个进程生成另一个和这个进程的内存内容相同的子进程。在父进程中，`fork` 的返回子进程的PID，在子进程中，返回值是0。
- `exit`：原型 `int exit(int status)`。作用是让调用它的进程停止执行并且将内存等占用的资源全部释放。`status` 是状态参数，0代表正常退出，1代表非正常退出。
- `wait`：原型 `int wait(int *status)`。等待子进程退出，返回子进程PID，子进程的退出状态存储到\*status地址中。如果没有调用子进程，wait返回-1。
- `exec`：原型 `int exec(char *file, char *argv[])`。作用是加载 一个文件，获取执行它的参数，执行。执行错误返回-1，执行成功则不会返回，而开始从文件入口位置开始执行命令，文件格式必须是ELF格式。

#### 2.IO和文件描述符

- file descriptor: 文件描述符, 一个被内核管理的、可以被进程读、写的对象的一个整数, 通过打开文件、目录、设备等方式获得。一个文件被打开的越早, 文件描述符越小。每个进程都有自己独立的文件描述符列表, 0是标准输入, 1是标准输出, 2是标准错误。shell保证总是3个文件描述符是可用的, 在给的源码中的 `sh.c` 中有这样一段代码

```

1  int fd;
2
3  // Ensure that three file descriptors are open.
4  while((fd = open("console", O_RDWR)) >= 0){
5      if(fd >= 3){
6          close(fd);
7          break;
8      }
9  }
10

```

- `read` 和 `write`: 原型 `int write(int fd, char *buf, int n)` 和 `int read(int fd, char *buf, int n)`。实现从/向文件描述符 `fd` 中写`n`字节 `buf` 内容, 返回值时读取/写入的字节数。每个文件描述符有一个offset, `read` 会从这个offset开始读取内容, 读完`n`个字节后将offset后移`n`个字节, 下一个`read`从新的offset开始读取字节。`write` 类似。
- `close`: 原型 `int close(int fd)`, 作用是将打开的文件 `fd` 释放, 使该文件描述符可以被后面的系统调用使用。父进程的fd table不会被子进程的变化硬性, 但文件中的offset共享。
- `dup`: 原型 `int dup(int fd)`, 复制一个新的 `fd` 指向的I/O对象, 返回这个新的fd值, 两个I/O对象的offset相同。

**3.管道Pipes:** 管道是暴露给进程的一对文件描述符, 一个文件描述符用来读, 另一个文件描述符用来写, 将数据从管道的一端写入, 将使其能够被从管道的另一端读出。`pipe` 也是一个系统调用, 原型是 `int pipe(int p[])`, `p[0]` 为读取的文件描述符, `p[1]` 为写入的文件描述符。

**4.文件系统:** xv6文件系统包含了文件(byte arrays)和目录(对其他文件和目录的引用)。目录生成了一个树, 树从根目录 `/` 开始。对于不以 `/` 开头的路径, 认为是相对路径。相关系统调用:

- `mknod`: 创建设备文件, 一个设备文件有一个major device #和一个minor device #用来唯一确定这个设备。当一个进程打开了这个设备文件是, 内核会将 `read` 和 `write` 系统调用重新定向到设备上。
- 一个文件的名称和文件本身是不一样的, 文件本身, 也叫inode, 可以有多个名字, 也叫link, 每个link包括了一个文件名和一个对inode的引用。一个inode存储了文件的元数据, 包括该文件的类型(file, directory or device)、大小、文件在硬盘中的存储位置以及指向这个inode的link的个数
- `fstat`: 原型为 `int fstat(int fd, struct stat *st)`, 作用是将inode中的相关信息存储到 `st` 中。
- `link`: 创建一个指向同一个inode的文件名, `unlink` 则是将一个文件名从文件系统中移除, 只有当指向这个inode的文件名的数量为0时这个inode以及其存储的文件内容才会被从硬盘上移除

要阅读 [文档](#) 第一章, 或参考 `user/echo.c` 写好的方法, 查看编写格式, 即如何进行输入输出。

注意到 `main` 函数参数 `argc` 是传进去的参数数目, `argv` 是这些参数。注意到 `argv[1]` 开始才是真的第一个参数, 即传入的第二个参数。从 `echo` 也可印证这一点。

根据文档第 13 页，可以看到 `write` 函数的参数描述，它可以通过 `#include "user/user.h"` 引用，第一个参数是 file descriptor 文件描述符，整数；第二个参数是字符串指针(空指针)，第三个参数是字符数(在 `user/ulib.c` 手写了 `strlen` 函数)，在文档第 13 页头，描述了常用的几个文件描述符：①0 标准输入；②1 标准输出；③2 标准错误输出(`stderr`)

传入一个参数，可以使用 `user/ulib.c` 的 `atoi` 函数转 `char` 为 `int`。在 `kernel/sysproc.c` 有 `sys_sleep` 的手写函数，代表了 `sleep` 的核心逻辑。在 `user/user.h` 提供了调用的 `sleep` 函数。  
`user/usys.S` 汇编语言，部分代码表示进入内核态执行 `sleep`。

具体而言，检查传入的参数，如果不够 1 个或其他原因报错，正常情况 `exit`。在根目录看到 `Makefile`，需要添加一些东西。

## 实现

一个参考实现 [here 汇总](#)

因为对项目架构不太懂，所以暂且参考了一下现成代码。

具体而言，新建一个 `user/sleep.c`，代码如下：

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4  // the above 3 headers must be includes in such exact orders
5  int main(int argc, char **argv)
6  {
7      if (argc < 2)
8      { //一个 ticks 目测大约 100ms
9          printf("usage: sleep <ticks>\n");
10     }
11     int ticks = atoi(argv[1]);
12     sleep(ticks);
13     exit(0);
14 }
```

在 `makefile` `UPROGS=` 那一堆追加：

```
1  $U/_sleep\
```

验证正确性：

```
1  make qemu
2  sleep 10
3  # 关闭
```

单元测试：

```
1  ./grade-lab-util sleep
```

# pingpong

## 要求

Write a program that uses UNIX system calls to “ping-pong” a byte between two processes over a pair of pipes, one for each direction. The parent should send a byte to the child; the child should print “: received ping”, where is its process ID, write the byte on the pipe to the parent, and exit; the parent should read the byte from the child, print “: received pong”, and exit. Your solution should be in the file `user/pingpong.c`.

使用管道，实现一对进程的测试通信。父进程给子进程发送一个字节(byte)，子进程输出 `<pid>: received ping`，然后将收到的字节发回去，父进程输出 `<pid>: received pong`。在 `user/pingpong.c` 实现。

需要用到的函数：

- `int pipe(int*)`；传入一个长为 2 的 int 数组，下标 0 是从管道读取数据的文件描述符，下标 1 是向管道写入数据的文件描述符。参考文档第 15 页。一个管道是半双工的，一方读一方写。
- `int fork(void)`；参考文档第 10 页，创建一个当前进程的子进程，返回新建进程的 pid。新进程的内存内容与原进程一样。此时，进入并发，被新建的进程的 `fork` 返回 0，所以如果调用了 `fork`，根据返回值判断当前程序代码是被父进程还是子进程走，需要 `if else`。
- `int read(int, void*, int)`；第一个参数是管道的第一个下标 0 代表的描述符，代表从这个管道读，第二个参数是 `char`(按地址传入，或指针)，读一个字节，第三个参数代表第二个参数的字符长度。如果读不到，会阻塞等待，直到读到了再往下走。参考文档第 15 页。如果管道写侧关闭，会返回 0。

同理有 `int write(int, const void*, int)`，第一个传的是管道下标 1 的值

- `int getpid()`；参考文档第 11 页。直接返回当前进程 pid。
- `int exit(int status)` 参考文档第 11 页。结束运行。参考 11 页尾，0 是成功，1 是失败。
- `int wait(int*)` 参考文档第 11 页。等待直接子进程结束，结束后继续执行，否则阻塞，以传入的状态 `int * status` 结束，返回子进程 pid。更多参见 11 页尾。如果有很多个子，可以 `while(wait(0) != -1)`；自旋等。

## 实现

按照题意实现即可。

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int main(int argc, char **argv)
6  {
7      int pp2c[2]; // pipe parent to child
8      pipe(pp2c);
9      int pc2p[2]; // pipe child to parent
10     pipe(pc2p);
11     if (fork() != 0) // parent process
12     {
13         write(pp2c[1], "+", 1); // any char with length 1
14         char c;
15         read(pc2p[0], &c, 1); // read char of length 1
16         printf("%d: received pong\n", getpid());
```

```

17     wait(0);
18 } else {
19     char c;
20     read(pp2c[0], &c, 1);
21     printf("%d: received ping\n", getpid());
22     write(pc2p[1], &c, 1);
23 }
24 exit(0);
25 }

```

记得添加 makefile。

测试：

```

1 make qemu
2 pingpong

```

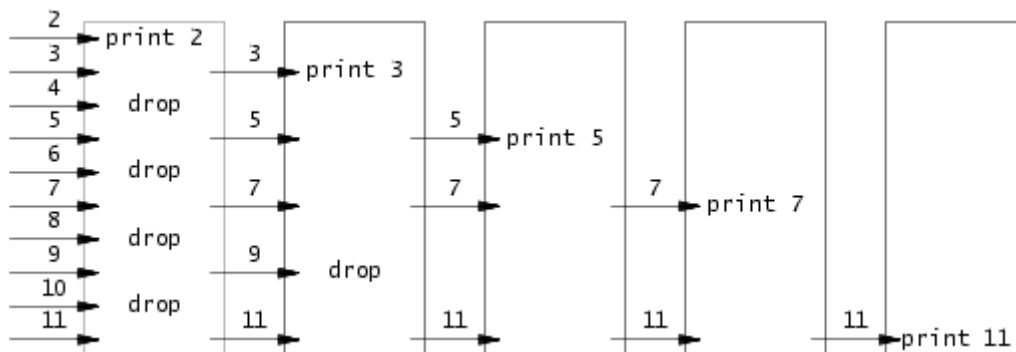
## primes

### 要求

Write a concurrent version of prime sieve using pipes. This idea is due to Doug McIlroy, inventor of Unix pipes. The picture halfway down this page and the surrounding text explain how to do it.

Your solution should be in the file `user/primes.c`.

使用管道并发地获取质数，写在 `user/primes.c`，输出 [2,35] 的全部质数。对每个质数，创建一个进程用管道读取它的左邻居，写给它的右邻居。（采用递归实现，每个进程中读取管道中的第一个数，就是一个素数，然后创建一个新的管道，将筛后的素数传到管道里，传给子进程。）



提示：

- 关闭进程不需要的文件需要小心，否则可能会资源枯竭。
- 首个进程到达 35 时，需要等待管道终止，即它的全部子孙，即主进程应在所有输出都完毕后再关闭
- 注意写侧管道关闭会 read 返回 0
- 最简单是直接写进管道，而不是格式化

参考实现思路文章 [这里](#)

参考函数：

- `int close(int)` 关闭文件描述符

## 实现

- 主进程生成 [2,35] 的全部质数
- 子进程1 输出2, 筛掉所有 2 的倍数
- 子进程2 输出3, 筛掉所有 3 的倍数
- 子进程3 输出5, 筛掉所有 5 的倍数
- .....

具体而言, 每个进程有一个父, 数据交换为 pleft, 子从父读全部数据, 输出第一个数据, 然后剩下的数据把倍数筛了, 再丢给下一个进程处理, 如此递归, 然后每个线程都等待自己的儿子结束自己再结束。

要注意关闭不需要用到的文件描述符, 否则跑到  $n = 13$  的时候就会爆掉, 出现读到全是 0 的情况, 因为 xv6 每个进程能打开的文件描述符总数只有 16 个, 参考 defs.h 中的 NOFILE 和 proc.h 中的

```
struct file *ofile[NOFILE]; // Open files.
```

由于一个管道会同时打开一个输入文件和一个输出文件, 所以一个管道就占用了 2 个文件描述符, 并且复制的子进程还会复制父进程的描述符, 于是跑到第六七层后, 就会由于最末端的子进程出现 16 个文件描述符都被占满的情况, 导致新管道创建失败。

所以:

- 关闭管道的两个方向中不需要用到的方向的文件描述符 (在具体进程中将管道变成只读/只写)
- 子进程创建后, 关闭父进程与祖父进程之间的文件描述符 (因为子进程并不需要用到之前 stage 的管道)

参考:

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  void filt(int[2]);
6  int main(int argc, char **argv)
7  {
8      int input_pipe[2];
9      pipe(input_pipe);
10     if (fork() != 0) // parent process
11     {
12         close(input_pipe[0]); // no need to input
13         int i;
14         for (i = 2; i <= 35; ++i)
15         {
16             write(input_pipe[1], &i, sizeof i);
17         }
18         i = -1; // denotes completed
19         write(input_pipe[1], &i, sizeof i);
20     }
21     else
22     {
23         close(input_pipe[1]); // no need to output
24         filt(input_pipe);
25         exit(0);
26     }
27     wait(0);
```

```

28     exit(0);
29 }
30 // pipe left
31 // read one prime, and delete all its multiples
32 void filt(int pleft[2])
33 {
34     int p; // prime
35     read(pleft[0], &p, sizeof p);
36     if (p == -1)
37     {
38         exit(0);
39     }
40     printf("prime %d\n", p);
41
42     int pright[2]; // pipe right
43     pipe(pright);
44     if (fork() != 0) // parent process
45     {
46         close(pright[0]);
47         int val;
48         while (read(pleft[0], &val, sizeof val) && val != -1)
49         {
50             if (val % p)
51             {
52                 write(pright[1], &val, sizeof val);
53             }
54         }
55         val = -1;
56         write(pright[1], &val, sizeof val);
57         wait(0);
58         exit(0);
59     }
60     else
61     {
62         close(pright[1]);
63         close(pleft[0]);
64         filt(pright);
65     }
66 }

```

## find

### 要求

Write a simple version of the UNIX find program: find all the files in a directory tree with a specific name. Your solution should be in the file `user/find.c`.

就是实现在给定路径下出发，查找所有路径中给定文件名的路径，输出所有文件的路径，可以根据 `ls.c` 改造。

编写 `user/find.c`，找到目录下所有特定文件名的文件

参考 `user/ls.c` 查看如何读目录，参考函数：

- `int open(char *file, int flags)` 返回文件描述符, 0 是最小的文件描述符, 不存在返回负数。参考第 14 页, 打开方式在 `kernel/fcntl.h` 描述, 分别有 `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREATE`, `O_TRUNC`。不存在就新建, 不论读写。`trunc` 是清空。使用示例参考第 17-18 页。可以用 `|` 来如新建+只写。
- `int fstat(int fd, struct stat *st)` 或 `int stat(char *file, struct stat *st)`  
其中 `stat` 结构体在 `kernel/stat.h` 定义, 描述了文件大小、文件是目录还是文件等信息。参考第 18 页。

使用递归来找子目录, 不要重复找 `.` 和 `..`。注意对文件系统的改变是持久化的, 清空文件系统使用 `make clean`, 然后重新 `make qemu`。

`kernel/fs.h` 有 `struct dirent`, 对目录用 `int read(int, void*, int)` 时第二个参数是 `fs.h` 里结构体 `dirent` 的地址, 第三个参数是其长度, 返回成功读取的长度, 每次读取该目录的下一个路径, 直到读完, 可以当迭代器使用。返回成功读取的长度。其中 `dirent` 的成员 `inum` 仿照 `ls.c` 可知为 0 要跳过, `char name[14]` 是该目录下的一个文件名。

参考测试样例:

```
1 make qemu
2 echo > b
3 mkdir a
4 echo > a/b
5 echo > a/a #我加的
6 find . b
7 # 文件系统包含 ./b 和 ./a/b
```

## 实现

参考要求, 仿照 `ls.c` 格式, 不难得出具体实现:

```
1 #include "kernel/types.h"
2 #include "kernel/stat.h"
3 #include "user/user.h"
4 #include "kernel/fs.h"
5 #include "kernel/fcntl.h"
6
7 #define CHAR_LEN 512
8 void find(char *, char *);
9 int main(int argc, char *argv[])
10 {
11     if (argc < 3)
12     {
13         printf("find: too few args");
14         exit(0);
15     }
16
17     // target = "/" + argv[2] for convenience for cmp
18     char target[CHAR_LEN];
19     target[0] = '/';
20     strcpy(target + 1, argv[2]);
21
22     find(argv[1], target);
23     exit(0);
```

```

24 }
25 // curr path: path, relative name to find: target
26 void find(char *path, char *target)
27 {
28     int fd;
29     fd = open(path, O_RDONLY);
30     if (fd < 0)
31     {
32         fprintf(2, "find: cannot open %s\n", path);
33         return;
34     }
35
36     struct stat st;
37     if (fstat(fd, &st) < 0)
38     {
39         fprintf(2, "find: cannot stat %s\n", path);
40         close(fd);
41         return;
42     }
43
44     switch (st.type)
45     {
46     case T_FILE:
47         // suffix is target
48         if (strcmp(path + strlen(path) - strlen(target), target) == 0)
49         {
50             printf("%s\n", path);
51         }
52         break;
53     case T_DIR:
54         //make a {} field to contains char buf
55         char buf[CHAR_LEN], *p;
56         // similar to user/ls.c line 50
57         if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf)
58         {
59             printf("ls: path too long\n");
60             break;
61         }
62
63         strcpy(buf, path); // path prefix
64         p = buf + strlen(buf); // pointer move
65         *p++ = '/'; // cur pointer
66
67         struct dirent de;
68         while (read(fd, &de, sizeof(de)) == sizeof(de))
69         {
70             // similar to user/ls.c
71             if (de.inum == 0)
72             {
73                 continue;
74             }
75             // copy filename and add into buf, do not change position of p
76             memmove(p, de.name, DIRSIZ);
77             p[DIRSIZ] = '\0';
78
79             if (stat(buf, &st) < 0)

```

```

80         {
81             printf("find: cannot stat %s\n", buf);
82             continue;
83         }
84
85         // skip . and ..
86         if (strcmp(buf + strlen(buf) - 2, "/.") != 0 &&
87             strcmp(buf + strlen(buf) - 3, "/..") != 0)
88         {
89             find(buf, target);
90         }
91     }
92     break;
93 }
94 }
95 close(fd);
96 }

```

注意为了方便比较，可以让比较目标设定为 `/ + target`，这样刚好对应，`cmp` 方便，免得假查。这个任务基本上就是对字符串匹配和 DFS 的小模拟，熟悉相关库函数和项目结构即可。

特别注意在 `switch` 内定义局部变量需要有作用域，所以 `case` 要加多一层大括号。

## xargs

### 要求

Write a simple version of the UNIX `xargs` program: read lines from the standard input and run a command for each line, supplying the line as arguments to the command. Your solution should be in the file `user/xargs.c`.

前置知识：

**命令行参数与标准化输入：**命令行参数是在 shell 中输入命令时跟在命令后边的参数，例如 `mkdir a b c`，`a`，`b`，`c` 就是 `mkdir` 接收的命令行参数。标准化输入是程序执行时，在 shell 中输入的东西，程序所等待的就是标准化输入。

**标准化输出：**命令返回结果就是一个标准化输出

**管道符 |：**管道符的作用是前一个命令的输出会作为后一个命令的输入

```
1 | cmdA | cmdB
```

cmdA 的输出会作为 cmdB 的输入

**xargs：**`xargs` 与管道符搭配使用，前一个命令的输出会作为后一个命令的命令行参数。 [参考](#)

通过管道，将输入传给 `xargs`，将输入逐行传给 `xargs` 执行其他命令，如：

```

1 | echo hello too | xargs echo bye # 等于 xargs echo bye hello too 即 echo 输出拼上去
2 | echo "1\n2" | xargs -n 1 echo line # 输出：
3 | #line 1
4 | #line 2

```

对每个 `b` 文件查询 `hello`:

```
1 | find . b | xargs grep hello
```

通过测试:

```
1 | sh < xargstest.sh
```

具体而言, 是初始目录下的如下指令组成的脚本:

```
1 | mkdir a
2 | echo hello > a/b
3 | mkdir c
4 | echo hello > c/b
5 | echo hello > b
6 | find . b | xargs grep hello
```

预期输出: (每一个 `$` 对应执行一行然后没输出)

```
1 | $ $ $ $ $ $ hello
2 | hello
3 | hello
4 | $ $
```

参考需要的命令:

- `int exec(char*, char**)` 第一个参数是代码文件名, 加载该代码文件并用第二个参数代表的参数执行, 执行出错时返回

## 实现

从标准输入读取管道内容, 每次读到空白字符时存一下当前单词, 将它追加到 `argv` 去, 每次读到换行时, 对当前维护的 `argv`, 开一个 `fork` 去执行 `xargs` 指令。注意尾处理。整体而言还是一个字符串小模拟。

参考实现:

```
1 | #include "kernel/types.h"
2 | #include "kernel/stat.h"
3 | #include "user/user.h"
4 | #include "kernel/fs.h"
5 |
6 | void run(char *program, char **args)
7 | {
8 |     if (fork() == 0)
9 |     {
10 |         exec(program, args);
11 |         exit(0);
12 |     }
13 | }
14 | int main(int argc, char *argv[])
15 | {
16 |     // the below purpose is to delete the argv[0]
17 |     // 128 pointers pointing argv
```

```

18     char *pargs[128];
19     // the first position have no argv
20     char **args = pargs;
21     for (int i = 1; i < argc; ++i)
22     {
23         *args = argv[i];
24         ++args;
25     }
26
27     // stored the chars read from stdin
28     char buf[2048];
29     // p is latest position, last_p is last word's first char's position
30     char *p = buf, *last_p = buf;
31     // the prefix (cmd to exec xargs)
32     char **pa = args;
33     // read from stdin
34     for (; read(0, p, 1) != 0; ++p)
35     {
36         if (*p == ' ' || *p == '\n') // get a word
37         {
38             *p = '\0';
39             *pa = last_p; // add a word to pargs
40             pa++;
41             last_p = p + 1; // next word in future
42             if (*p == '\n')
43             {
44                 *pa = 0; // nullptr
45                 run(argv[1], pargs);
46                 pa = args; // revoke what read in the line
47             }
48         }
49     }
50     // the last line have no \n
51     if (pa != args)
52     {
53         *p = '\0';
54         *(pa++) = last_p;
55         *pa = 0;
56         run(argv[1], pargs);
57     }
58     while (wait(0) != -1)
59         ;
60     exit(0);
61 }

```

至此，Lab1:Xv6 and Unix utilities必做实验部分结束，感觉好难啊。

以下内容可选做任务。

## uptime

### 要求

选做任务。

调用系统函数 `uptime()`，返回一个整数，输出从系统运行到现在的时间刻数量(tick)。一秒大约20左右 tick，没细测。

### 实现

直接调库。

```
1 #include "kernel/types.h"
2 #include "kernel/stat.h"
3 #include "user/user.h"
4
5 int main() {
6     printf("%d\n", uptime());
7     exit(0);
8 }
```

## find.grep

### 要求

选做任务。

修改 `find.c`，参照 `grep.c` 的 `match` 函数，将 `find` 改成支持基础通配符如 `.*^$`。

### 实现

直接照抄 `grep.c` 的 `match` 函数和它依赖的两个函数，因为不确定能不能直接 `include grep`，所以直接把这几个函数 CV 了，放到 `find` 里，然后对 `case T_FILE` 的 if，直接改成 `match(target,path)`。

注意到 `match(char *re, char *text)` 是查看 re 是否是 text 的前缀的意思，为了准确查找，而不只是前缀，不妨修改 `target` 为 `/target/`，目标串后面也加一个 `/`。考虑到 `*` 是贪心，所以有 `*` 就不加，否则加。

经过测试，怀疑他标准实现的 `match` 有误，所以忽略该任务，这不是操作系统的重点，所以暂且不管了。放弃对通配符的支持。

具体而言，`match` 参数分别为 `/p*g /grind/`，能匹配出来，但 `/p*g /pingpong/` 匹配不出来，这是不符合 `match` 规则的。

感觉跟操作系统关系不大，这里不做了。

## sh

### 要求

选做任务。

对 `user/sh.c`，改进：

- 处理文件时不输出 `$`
- 支持 `wait`

- 支持 `;` 分隔一行多个指令
- 支持子命令 `()`。在括号内的命令将在一个新的 shell 进程中执行
- 支持 tab 补全
- 支持 shell 命令历史

## 实现

感觉跟操作系统关系不大，这里不做了。

没做，一个实现的版本 [参考](#)

## lab2 system calls

**前置知识（对Xv6的了解）：**操作系统必须满足三个要求：多路复用、隔离和交互，Xv6 book第二章就是介绍相关知识的。

### 1. 用户态，核心态，以及系统调用：

RISC-V有三种CPU可以执行的模式：机器模式、用户模式和管理模式，机器模式是cpu启动时的状态，主要用于配置计算机，然后更改为管理模式。在管理模式下，CPU被允许执行特权指令。

想要调用内核函数的应用程序必须过渡到内核，CPU提供一个特殊的指令，将CPU从用户模式切换到管理模式，并在内核指定的入口点进入内核（RISC-V为此提供 `ecall` 指令）。一旦CPU切换到管理模式，内核就可以验证系统调用的参数，决定是否允许应用程序执行请求的操作，然后拒绝它或执行它。

### 2. 内核组织

**宏内核：**整个操作系统都驻留在内核中，所有的系统调用的实现都以管理模式运行。

**微内核：**操作系统设计者可以最大限度地减少在管理模式下运行的操作系统代码量，并在用户模式下执行大部分操作系统。这种内核组织被称为**微内核（microkernel）**

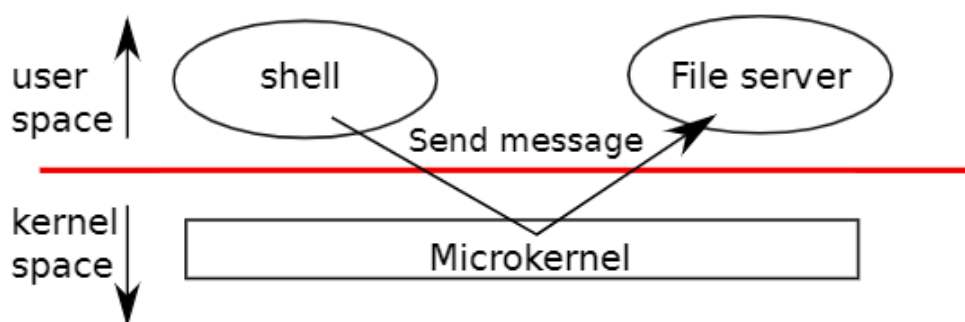


Figure 2.1: A microkernel with a file-system server

CSDN @up-to-star

上图说明了这种微内核设计。在图中，文件系统作为用户级进程运行。作为进程运行的操作系统服务被称为服务器。为了允许应用程序与文件服务器交互，内核提供了允许从一个用户态进程向另一个用户态进程发送消息的进程间通信机制。例如，如果像shell这样的应用程序想要读取或写入文件，它会向文件服务器发送消息并等待响应。

### 3. Xv6架构

XV6的源代码位于`kernel/`子目录中，源代码按照模块化的概念划分为多个文件，图2.2列出了这些文件，模块间的接口都被定义在了`def.h`（`kernel/defs.h`）。

文件	描述
bio.c	文件系统的磁盘块缓存
console.c	连接到用户的键盘和屏幕
entry.S	首次启动指令
exec.c	exec()系统调用
file.c	文件描述符支持
fs.c	文件系统
kalloc.c	物理页面分配器
kernelvec.S	处理来自内核的陷入指令以及计时器中断
log.c	文件系统日志记录以及崩溃修复
main.c	在启动过程中控制其他模块初始化
pipe.c	管道
plic.c	RISC-V中断控制器
printf.c	格式化输出到控制台
proc.c	进程和调度
sleeplock.c	Locks that yield the CPU
spinlock.c	Locks that don't yield the CPU.
start.c	早期机器模式启动代码
string.c	字符串和字节数组库
swtch.c	线程切换
syscall.c	Dispatch system calls to handling function.
sysfile.c	文件相关的系统调用
sysproc.c	进程相关的系统调用
trampoline.S	用于在用户和内核之间切换的汇编代码
trap.c	对陷入指令和中断进行处理并返回的C代码
uart.c	串口控制台设备驱动程序
virtio_disk.c	磁盘设备驱动程序
vm.c	管理页表和地址空间

**4. 进程概述：**Xv6（和其他Unix操作系统一样）中的隔离单位是一个进程。进程抽象防止一个进程破坏或监视另一个进程的内存、CPU、文件描述符等。它还防止一个进程破坏内核本身，这样一个进程就不能破坏内核的隔离机制。内核用来实现进程的机制包括用户/管理模式标志、地址空间和线程的时间切片。

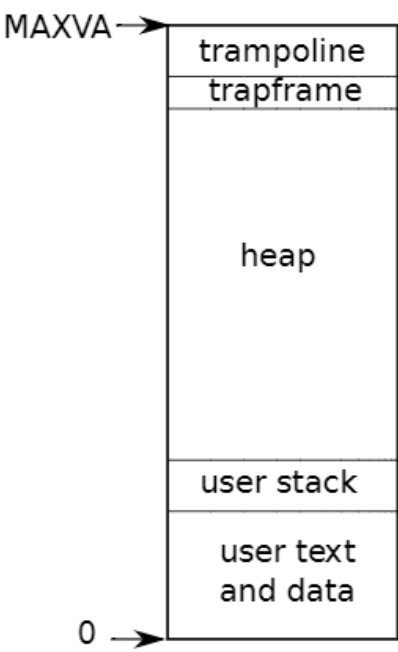


Figure 2.3: Layout of a process's virtual address space CSDN @up-to-star

Xv6为每个进程维护一个单独的页表，定义了该进程的地址空间。如图2.3所示，以虚拟内存地址0开始的进程的用户内存地址空间。首先是指令，然后是全局变量，然后是栈区，最后是一个堆区域（用于 malloc）以供进程根据需要进行扩展。

每个进程都有一个执行线程（或简称线程）来执行进程的指令。一个线程可以挂起并且稍后再恢复。为了透明地在进程之间切换，内核挂起当前运行的线程，并恢复另一个进程的线程。线程的大部分状态（本地变量、函数调用返回地址）存储在线程的栈区上。每个进程有两个栈区：一个用户栈区和一个内核栈区（p->kstack）。当进程执行用户指令时，只有它的用户栈在使用，它的内核栈是空的。当进程进入内核（由于系统调用或中断）时，内核代码在进程的内核堆栈上执行；当一个进程在内核中时，它的用户堆栈仍然包含保存的数据，只是不处于活动状态。进程的线程在主动使用它的用户栈和内核栈之间交替。内核栈是独立的（并且不受用户代码的保护），因此即使一个进程破坏了它的用户栈，内核依然可以正常运行。

一个进程可以通过执行RISC-V的 `ecall` 指令进行系统调用，该指令提升硬件特权级别，并将程序计数器（PC）更改为内核定义的入口点，入口点的代码切换到内核栈，执行实现系统调用的内核指令，当系统调用完成时，内核切换回用户栈，并通过调用 `sret` 指令返回用户空间，该指令降低了硬件特权级别，并在系统调用指令刚结束时恢复执行用户指令。

**5. 启动XV6和第一个进程**

当RISC-V计算机上电时，它会初始化自己并运行一个存储在只读内存中的引导加载程序。引导加载程序将xv6内核加载到内存中。然后，在机器模式下，中央处理器从 `_entry` (`kernel/entry.S:6`)开始运行xv6。Xv6启动时页式硬件（paging hardware）处于禁用模式：也就是说虚拟地址将直接映射到物理地址。

加载程序将xv6内核加载到物理地址为 0x80000000 的内存中。它将内核放在 0x80000000 而不是 0x0 的原因是地址范围 0x0:0x80000000 包含I/O设备。

`_entry` 的指令设置了一个栈区，这样xv6就可以运行C代码。Xv6在 `start.c` (**`kernel/start.c:11`**)文件中为初始栈`stack0`声明了空间。由于RISC-V上的栈是向下扩展的，所以`_entry`的代码将栈顶地址`stack0+4096`加载到栈顶指针寄存器 `sp` 中。现在内核有了栈区，`_entry` 便调用C代码 `start` (**`kernel/start.c:21`**)。

函数 `start` 执行一些仅在机器模式下允许的配置，然后切换到管理模式。RISC-V提供指令 `mret` 以进入管理模式，该指令最常用于将管理模式切换到机器模式的调用中返回。而 `start` 并非从这样的调用返回，而是执行以下操作：它在寄存器 `mstatus` 中将先前的运行模式改为管理模式，它通过将 `main` 函数的地址写入寄存器 `mepc` 将返回地址设为 `main`，它通过向页表寄存器 `satp` 写入0来在管理模式禁用虚拟地址转换，并将所有的中断和异常委托给管理模式。

在进入管理模式之前，`start` 还要执行另一项任务：对时钟芯片进行编程以产生计时器中断。清理完这些“家务”后，`start` 通过调用 `mret` “返回”到管理模式。这将导致程序计数器（PC）的值更改为 `main` (`kernel/main.c:11`)函数地址。

在 `main` (`kernel/main.c:11`)初始化几个设备和子系统后，便通过调用 `userinit` (**`kernel/proc.c:212`**)创建第一个进程，第一个进程执行一个用RISC-V程序集写的小型程序：`initcode.S` (**`user/initcode.S:1`**)，它通过调用 `exec` 系统调用重新进入内核。正如我们在第1章中看到的，`exec` 用一个新程序（本例中为 `/init`）替换当前进程的内存和寄存器。一旦内核完成 `exec`，它就返回 `/init` 进程中的用户空间。如果需要，`init` (**`user/init.c:15`**)将创建一个新的控制台设备文件，然后以文件描述符0、1和2打开它。然后它在控制台上启动一个shell。系统就这样启动了。

## 任务要求

做增量更新，进行：

```
1 git remote add origin2 git://g.csail.mit.edu/xv6-labs-2021
2 git fetch remote2 # 如果没挂或挂了多个remote灵活处理
3 git merge origin2/syscall # 合并冲突的话自行处理
4 make clean # 清空文件系统
```

预计能正常跑。

我遇到的冲突是 `Makefile` 和 `conf/lab.mk` 和 `user/usys.pl`，我 `Makefile` 用自己的，后两个用他新的。

## trace

### 要求

设计一个 `trace` 系统调用来追踪系统调用。传入一个整数参数 `mask`，位表示要追踪的系统调用，如追踪 `fork` 则调用 `trace(1 << SYS_fork)`，其中该常量可以在 `kernel/syscall.h` 查看。

输出一行表示对应系统调用的进程 id，系统调用的名字和返回值。追踪的进程包含该进程所 `fork` 的。

提供了 `user/trace.c` 提供用户态对 `trace` 的调用。

参考正常的测试用例和结果：

```
1 $ trace 32 grep hello README
2 3: syscall read -> 1023 #箭头右边的可能有所不同
3 3: syscall read -> 966
4 3: syscall read -> 70
5 3: syscall read -> 0
```

```

6 $
7 $ trace 2147483647 grep hello README
8 4: syscall trace -> 0
9 4: syscall exec -> 3
10 4: syscall open -> 3
11 4: syscall read -> 1023
12 4: syscall read -> 966
13 4: syscall read -> 70
14 4: syscall read -> 0
15 4: syscall close -> 0
16 $
17 $ grep hello README
18 $
19 $ trace 2 usertests forkforkfork
20 usertests starting
21 test forkforkfork: 407: syscall fork -> 408
22 408: syscall fork -> 409
23 409: syscall fork -> 410
24 410: syscall fork -> 411
25 409: syscall fork -> 412
26 410: syscall fork -> 413
27 409: syscall fork -> 414
28 411: syscall fork -> 415
29 ... #最后输出ALL TESTS PASSED
30 $

```

测试用例解释：

- 32 是 SYS\_read
- 2147483647 包含全部系统调用
- 第三个命令没有追踪
- 第四个命令，usertests 对应的源文件几千行代码。

应该做的事情：

- 把 user/trace.c 加到 makefile。
- 运行 make qemu 无法编译 user/trace.c，因为系统调用还没存在。需要在 user/user.h 添加对 trace 系统调用的定义，这是与 makefile 调用的 perl 脚本 user/usys.pl 相关的，该脚本生成 user/usys.S。系统调用使用 RISC-V ecall 指令去转入内核。如果这些做好了，调用 trace 32 grep hello README 还是会 fail，因为还没实现内核系统调用
- 因此，需要增加 sys\_trace() 函数在 kernel/sysproc.c，把 trace 函数卸载 kernel/proc.h 里，然后在 kernel/syscall.c 添加内容暴露给用户空间。可以查看已有代码模仿着写。
- 修改 fork() 函数(在 kernel/proc.c)，让子进程复制 trace mask。
- 修改 kernel/syscall.c 的 syscall() 函数，来打印 trace 输出。增加一个数组表示系统调用的名字和下标。

## 实现

在 `kernel/sysproc.c` 添加代码如下：

```
1  uint64
2  sys_trace(void)
3  {
4      int mask;
5      if(argint(0,&mask)<0) {
6          return -1;
7      }
8      myproc()->syscall_trace = mask;
9      return 0;
10 }
```

在 `kernel/syscall.h` 添加一行：

```
1  #define SYS_trace 22
```

在 `kernel/syscall.c` 添加新的函数声明：

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

在同文件的 `static uint64 (*syscalls[])(void) =` 添加：

```
1  [SYS_trace]    sys_trace,
```

具体而言，这是一个函数指针数组，`static` 表示私有，只有定义它的源文件可以访问，`uint64` 是函数返回值，`(*syscalls[])` 是不定数目的函数指针数组，`(void)` 是函数传入的参数。`[SYS_trace]` 是下标索引，`sys_trace` 是函数名字，具体指代 `kernel/sysproc.c` 定义的那个函数。除了这些下标 (`SYS_trace` 在刚刚 `kernel/syscall.h` 定义了)外，其他元素填充 0，达成数组。如：`int arr[] = {[3] 2333, [6] 6666}`。

在 `user/usys.pl`，加入用户态到内核态跳板函数：

```
1  entry("trace");
```

该脚本运行后，生成汇编文件 `usys.s`，定义了跳板函数如：

```
1  .global trace
2  trace:      # 定义用户态跳板函数
3      li a7, SYS_trace    # 将系统调用 id 存入 a7 寄存器
4      ecall            # ecall, 调用 system call，跳到内核态的统一系统调用处理函数
5      syscall() (syscall.c)
6      ret
```

在 `user/user.h` 添加定义：

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

梳理一下，各代码文件功能：

1	<code>user/user.h:</code>	用户态程序调用跳板函数 <code>trace()</code>
2	<code>user/usys.s:</code>	跳板函数 <code>trace()</code> 使用 CPU 提供的 <code>ecall</code> 指令，调用到内核态
3	<code>kernel/syscall.c</code>	到达内核态统一系统调用处理函数 <code>syscall()</code> ，所有系统调用都会跳到这里来处理。
4	<code>kernel/syscall.c</code>	<code>syscall()</code> 根据跳板传进来的系统调用编号，查询 <code>syscalls[]</code> 表，找到对应的内核函数并调用。
5	<code>kernel/sysproc.c</code>	到达 <code>sys_trace()</code> 函数，执行具体内核操作

主要目的是实现用户态和内核态的良好隔离。

由于内核与用户进程的页表不同，寄存器也不互通，所以参数无法直接通过 C 语言参数的形式传过来，而是需要使用 `argaddr`、`argint`、`argstr` 等系列函数，从进程的 `trapframe` 中读取用户进程寄存器中的参数。

同时由于页表不同，指针也不能直接互通访问（也就是内核不能直接对用户态传进来的指针进行解引用），而是需要使用 `copyin`、`copyout` 方法结合进程的页表，才能顺利找到用户态指针（逻辑地址）对应的物理内存地址。

修改 `kernel/proc.h` 的 `struct proc`，添加成员 `uint64 syscall_trace`；。在 `kernel/proc.c` 创建新进程时，给该成员赋初始值 0，具体而言，在 `allocproc` 函数 `return` 前添加：

```
1 p->syscall_trace = 0;
```

如此便能看懂最开始添加的那一段 `sys_trace` 代码。

修改 `kernel/proc.c` 的 `fork` 函数，找到复制进程代码，在其之后添加复制 `syscall_trace` 成员的代码，具体而言：

```
1 safestrcpy(np->name, p->name, sizeof(p->name));
2 np->syscall_trace = p->syscall_trace; //新加的
3 pid = np->pid;
```

根据上方提到的系统调用的全流程，可以知道，所有的系统调用到达内核态后，都会进入到 `syscall()` 这个函数进行处理，所以要跟踪所有的内核函数，只需要在 `syscall()` 函数里埋点就行了。

在 `kernel/syscall.c` 找到 `syscall` 函数，`if` 分支尾部追加：

```
1 if((p->syscall_trace >> num) & 1) {
2     printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num], p->trapframe->a0); // syscall_names[num]: 从 syscall 编号到 syscall 名的映射表
3 }
```

其中，`syscall_names` 未定义，故同文件内定义：

```
1 const char *syscall_names[] = {
2     [SYS_fork]    "fork",
3     [SYS_exit]    "exit",
4     [SYS_wait]    "wait",
5     [SYS_pipe]    "pipe",
6     [SYS_read]    "read",
7     [SYS_kill]    "kill",
8     [SYS_exec]    "exec",
9     [SYS_fstat]   "fstat",
```

```

10     [SYS_chdir]    "chdir",
11     [SYS_dup]     "dup",
12     [SYS_getpid]  "getpid",
13     [SYS_sbrk]    "sbrk",
14     [SYS_sleep]   "sleep",
15     [SYS_uptime]  "uptime",
16     [SYS_open]    "open",
17     [SYS_write]   "write",
18     [SYS_mknod]   "mknod",
19     [SYS_unlink]  "unlink",
20     [SYS_link]    "link",
21     [SYS_mkdir]   "mkdir",
22     [SYS_close]   "close",
23     [SYS_trace]   "trace",
24 };

```

最后别忘了 `makefile` 在 `UPROGS=` 添加:

```
1 $U/_trace\
```

至此, 可以 `make qemu` 了。执行上面的测试, 预期可以通过。

## sysinfo

### 要求

In this assignment you will add a system call, `sysinfo`, that collects information about the running system. The system call takes one argument: a pointer to a `struct sysinfo` (see `kernel/sysinfo.h`). The kernel should fill out the fields of this struct: the `freemem` field should be set to the number of bytes of free memory, and the `nproc` field should be set to the number of processes whose `state` is not `UNUSED`. We provide a test program `sysinfotest`; you pass this assignment if it prints "sysinfotest: OK".

添加系统调用 `sysinfo`, 收集正在运行的系统的信息。占用一个参数, 对 `struct sysinfo` 的指针(查看 `kernel/sysinfo.h`), 结构体含空余字节内存数(`freemem`)和运行进程数(`nproc`)两个 `uint64` 成员。有测试程序 `user/sysinfotest.c`, 帮助测试实现是否正确。具体 `make qemu` 然后执行 `sysinfotest` 即可。

这个实验的过程和上个实验的流程基本是一致的, 首先要根据上个实验的提示, 在相关文件中添加内容, 实现创建系统调用的功能。然后根据提示进行推进实验。

需要

- 把 `sysinfotest` 添加到 `makefile`
- 添加 `sysinfo` 系统调用, 预定义已存在的 `struct sysinfo`。将其复制到用户态, 参考 `kernel/sysfile.c` 的 `sys_fstat()` 和 `kernel/file.c` 的 `filestat()` 参考如何使用 `copyout()`。
- 在 `kernel/kalloc.c` 添加统计内存的函数, 在 `kernel/proc.c` 添加统计进程的函数。

## 实现

常见的记录空闲页的方法有：空闲表法、空闲链表法、位示图法（位图法）、成组链接法。这里 xv6 采用的是空闲链表法。

可以看到 `kalloc.c` 里有一个结构体变量名为 `kmem`，包含单向链表 `freelist` 和 `lock` 自旋锁。所以正确的做法是先获取自旋锁，然后遍历 `kmem` 的单向链表，它有几个节点，每个节点代表一页，就乘以每页的大小单位(在 `kernel/riscv.h` 的常量 `PGSIZE`)，计算完后释放锁。注意不要维护一个什么变量动态在每次页发生变化时改变空闲内存数，要查时暴力查询即可。

模仿 `kalloc.c` 的 `kalloc` 函数，知道如何操作锁、遍历链表。可以看到每次分配新空间的办法是把单向链表第一个节点丢出去，根节点后移一位。同理，回收办法是把新空闲挂到单向链表头部。

因为是读，不用写，所以不加锁也行。

参考：

```
1  uint64
2  count_free_mem(void)
3  {
4      //acquire(&kmem.lock);
5      uint64 cnt = 0; //bytes of free memory
6      for(struct run *p = kmem.freelist; p; p = p->next) {
7          ++cnt;
8      }
9      cnt *= PGSIZE;
10     //release(&kmem.lock);
11     return cnt;
12 }
```

`kernel/defs.h` 定义了每个 `kernel` 的 `.c` 的函数。所以要统计内存，先定义一个函数，在 `//kalloc.c` 里追加定义：

```
1  uint64 count_free_mem(void);
```

运行进程数，同理，去 `proc.c` 定义一个函数。在 `proc.h` 定义了 `struct proc`，有枚举成员 `state`，若枚举值不为 `UNUSED`，则空闲(虽然不懂为什么逻辑反着来的，但是文档就这么说的)。其中，`proc.c` 的空闲表 `struct proc proc[NPROC]` 维护了所有运行中的进程，所以遍历这个数组即可(也可以用指针遍历)。可以不用锁。

参考：

```

1  uint64
2  count_process(void)
3  {
4      uint64 cnt = 0;
5      for(int i=0;i<NPROC;++i) {
6          struct proc *p = &proc[i]; //pointer avoid copy
7          if(p->state != UNUSED) {
8              ++cnt;
9          }
10     }
11     return cnt;
12 }
13 //更优遍历: for(struct proc *p = proc; p < &proc[NPROC]; p++) {

```

同理追加定义:

```

1  uint64 count_process(void);

```

模仿 kernel/file.c 的 filestat() 对 copyout 的格式, 可以知道需要 myproc() 的页表, 地址, 结构体和结构体大小。模仿 sysfile.c 的 sys\_fstat() 可知通过 argaddr 获取地址。

实现一个 sys\_sysinfo, 在 kernel/sysproc.c 添加:

```

1  uint64
2  sys_sysinfo(void)
3  {
4      uint64 addr; //memory to store sysinfo
5      if(argaddr(0,&addr)<0){
6          return -1;
7      }
8      struct sysinfo info; //should include "sysinfo.h"
9      info.freemem = count_free_mem();
10     info.nproc = count_process();
11     if(copyout(myproc()->pagetable, addr, (char *)&info, sizeof info) < 0) {
12         return -1;
13     }
14     return 0;
15 }

```

相应工作:

- kernel/syscall.h 添加:

```

1  #define SYS_sysinfo 23

```

- kernel/syscall.c 同 trace 理, 类似位置添加三处:

```

1  extern uint64 sys_sysinfo(void);
2  [SYS_sysinfo] sys_sysinfo,
3  [SYS_sysinfo] "sysinfo",

```

- Makefile 的 UPROGS= 添加

```
1 | $U/_sysinfotest\
```

- user/user.h 两个不同位置添加: (仿照原有 user.h 结构可知)

```
1 | struct sysinfo;
2 | int sysinfo(struct sysinfo *);
```

- user/usys.pl 添加:

```
1 | entry("sysinfo");
```

可选任务:

- 输出 trace 跟踪的系统调用的参数
- 计算 load average 并输出

## lab3 page tables

**基础知识:** 页表是操作系统为每个进程提供私有地址空间和内存的机制。基于页表, XV6 隔离不同进程的地址空间, 并将它们复用到单个物理内存上。页表还提供了一层抽象 (a level of indirection), 这允许xv6执行一些特殊操作: 映射相同的内存到不同的地址空间中 (a trampoline page), 并用一个未映射的页面保护内核和用户栈区。

### 1. 页式硬件

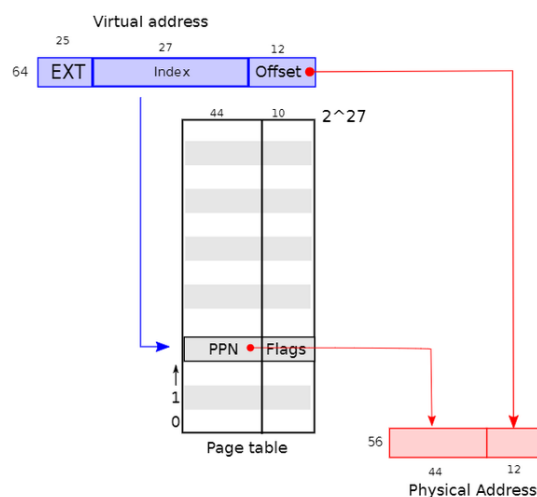


Figure 3.1: RISC-V virtual and physical addresses, with a simplified logical page table. ©2021 up-to-star

XV6基于Sv39RISC-V运行, 这意味着它只使用64位虚拟地址的低39位, 而高25位不使用。

页表在逻辑上由一个  $2^{27}$  个页表条目 (Page Table Entries/PTE) 组成的数组, 每个PTE包含一个44位的物理页码 (Physical Page Number/PPN) 和一些标志。虚拟地址的前27位作为虚页号, 后12位作为页内偏移, 也就是说XV6页大小为4096B。

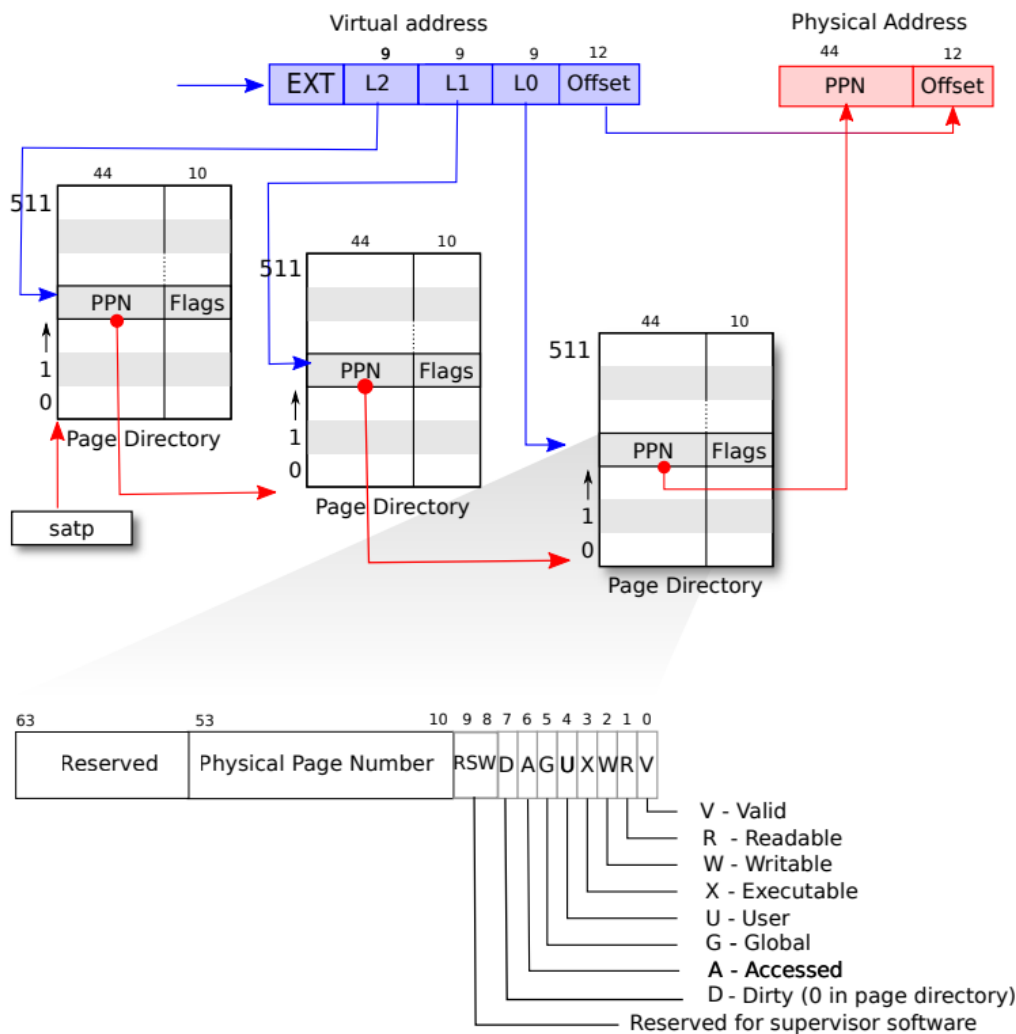


Figure 3.2: RISC-V address translation details. [CSDN @up-to-star](#)

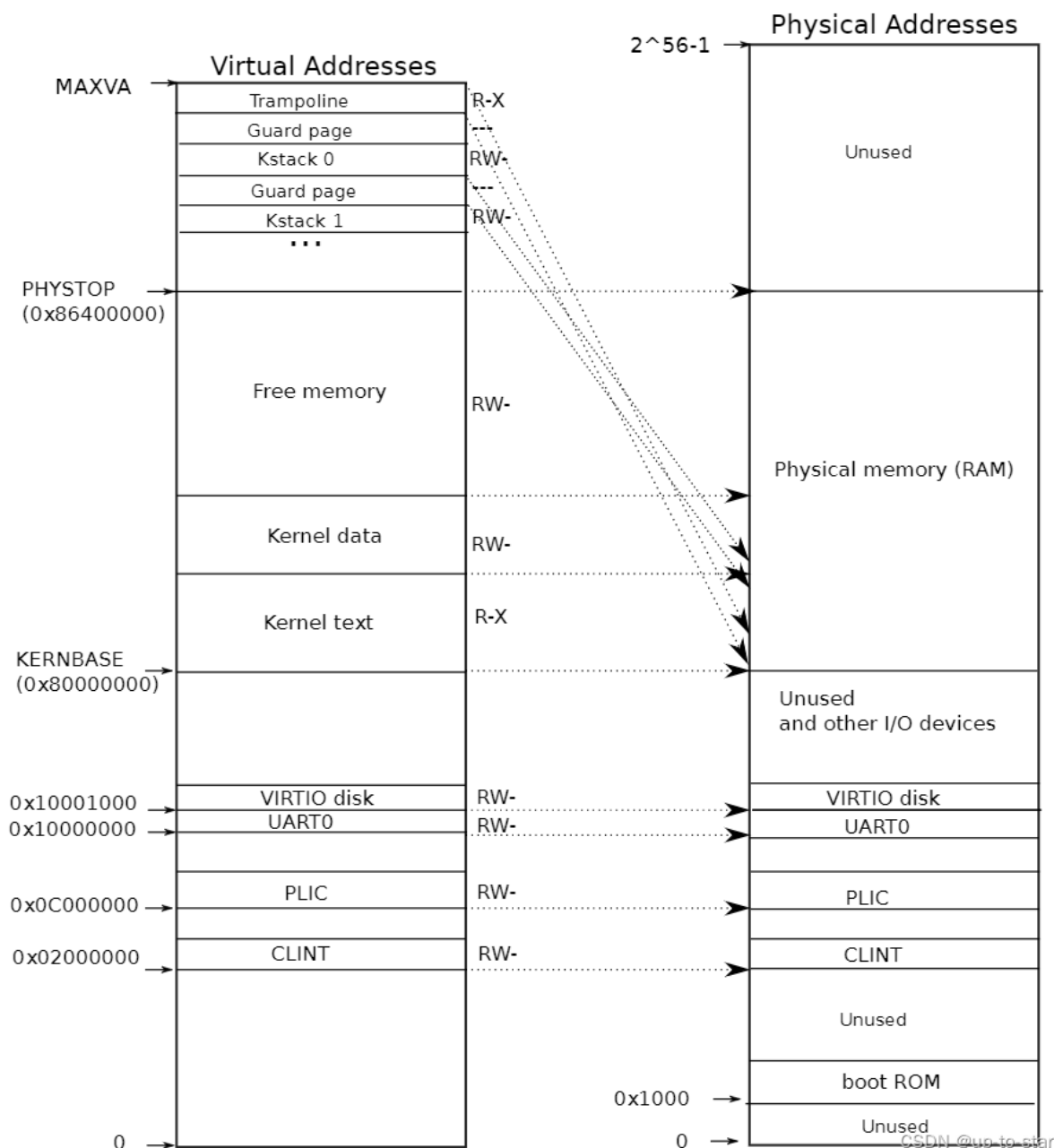
上图是一个三级页表，地址转换是类似的。

因为 CPU 在执行转换时会在硬件中遍历三级结构，所以缺点是 CPU 必须从内存中加载三个 PTE 以将虚拟地址转换为物理地址。为了减少从物理内存加载 PTE 的开销，RISC-V CPU 将页表条目缓存在 Translation Look-aside Buffer (TLB) 中。

每个PTE包含标志位，这些标志位告诉分页硬件允许如何使用关联的虚拟地址。PTE\_V 指示PTE是否存在：如果它没有被设置，对页面的引用会导致异常（即不允许）。PTE\_R 控制是否允许指令读取到页面。PTE\_W 控制是否允许指令写入到页面。PTE\_X 控制CPU是否可以将页面内容解释为指令并执行它们。PTE\_U 控制用户模式下的指令是否被允许访问页面；如果没有设置 PTE\_U，PTE只能在管理模式上使用。

为了告诉硬件使用页表，内核必须将根页表页的物理地址写入到 satp 寄存器中（satp 的作用是存放根页表页在物理内存中的地址）。每个CPU都有自己的 satp，一个CPU将使用自己的 satp 指向的页表转换后续指令生成的所有地址。

**2. 内核地址空间：**Xv6为每个进程维护了一个页表，用以描述每个进程的用户地址空间，外加一个单独描述内核地址空间的页表。内核配置其地址空间的布局，以允许自己可以预测的虚拟地址访问内存和各种硬件资源。



这个图显示了如何将内核虚拟地址映射到物理地址。内核使用“直接映射”获取内存和内存映射设备寄存器；也就是说，将资源映射到等于物理地址的虚拟地址。

### 3. 代码

用于操作地址空间和页表的代码大部分在 `vm.c` (**`kernel/vm.c:1`**) 中。其核心数据结构是 `pagetable_t`，它实际上是指向RISC-V根页表页的指针；一个 `pagetable_t` 可以是内核页表，也可以是一个进程页表。最核心的函数是 `walk` 和 `mappages`，前者为虚拟地址找到PTE，后者为新映射装载PTE。`copyout` 和 `copyin` 复制数据到用户虚拟地址或从用户虚拟地址复制数据，这些虚拟地址作为系统调用参数提供。

启动时，`main` 调用 `kvminit` 使用 `kvmake` 创建内核的页表，这个页表的地址直接引用物理内存。`kvmake` 首先分配一个物理内存页来保存根页表页。然后它调用 `kvmmap` 来装载内核需要的转换。转换包括内核的指令和数据、物理内存的上限到 `PHYSTOP`，并包括实际上是设备的内存 `Proc_mapstacks` (**`kernel/proc.c:33`**) 为每个进程分配一个内核堆栈。它调用 `kvmmap` 将每个堆栈映射到由 `KSTACK` 生成的虚拟地址，从而为无效的堆栈保护页面留出空间。

**5. 进程的地址空间：**每个进程都有一个单独的页表，在进程切换时，也会更改页表。一个进程的用户内存从虚拟地址零开始，可以增长到 `MAXVA`，最大为256G

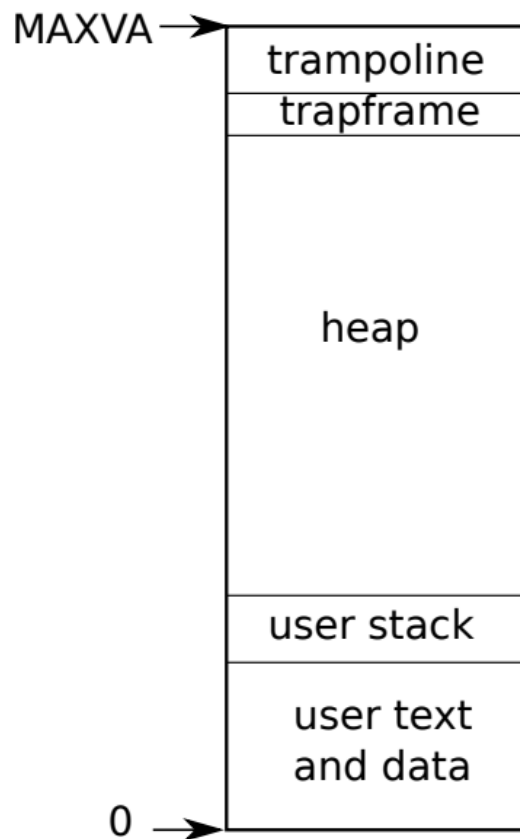


Figure 2.3: Layout of a process's virtual address space

当进程向xv6请求更多的用户内存时，xv6首先使用 `kalloc` 来分配物理页面。然后，它将PTE添加到进程的页表中，指向新的物理页面。Xv6在这些PTE中设置 `PTE_W`、`PTE_X`、`PTE_R`、`PTE_U` 和 `PTE_V` 标志。大多数进程不使用整个用户地址空间；xv6在未使用的PTE中留空`PTE_V`。

首先，不同进程的页表将用户地址转换为物理内存的不同页面，这样每个进程都拥有私有内存。第二，每个进程看到的自己的内存空间都是以0地址起始的连续虚拟地址，而进程的物理内存可以是非连续的。第三，内核在用户地址空间的顶部映射一个带有蹦床（trampoline）代码的页面，这样在所有地址空间都可以看到一个单独的物理内存页面。

在xv6中，栈是一个单独的页面，显示由 `exec` 创建后的初始内容。包含命令行参数的字符串以及指向它们的指针数组位于栈的最顶部。再往下是允许程序在 `main` 处开始启动的值（即 `main` 的地址、`argc`、`argv`），这些值产生的效果就像刚刚调用了 `main(argc, argv)` 一样。

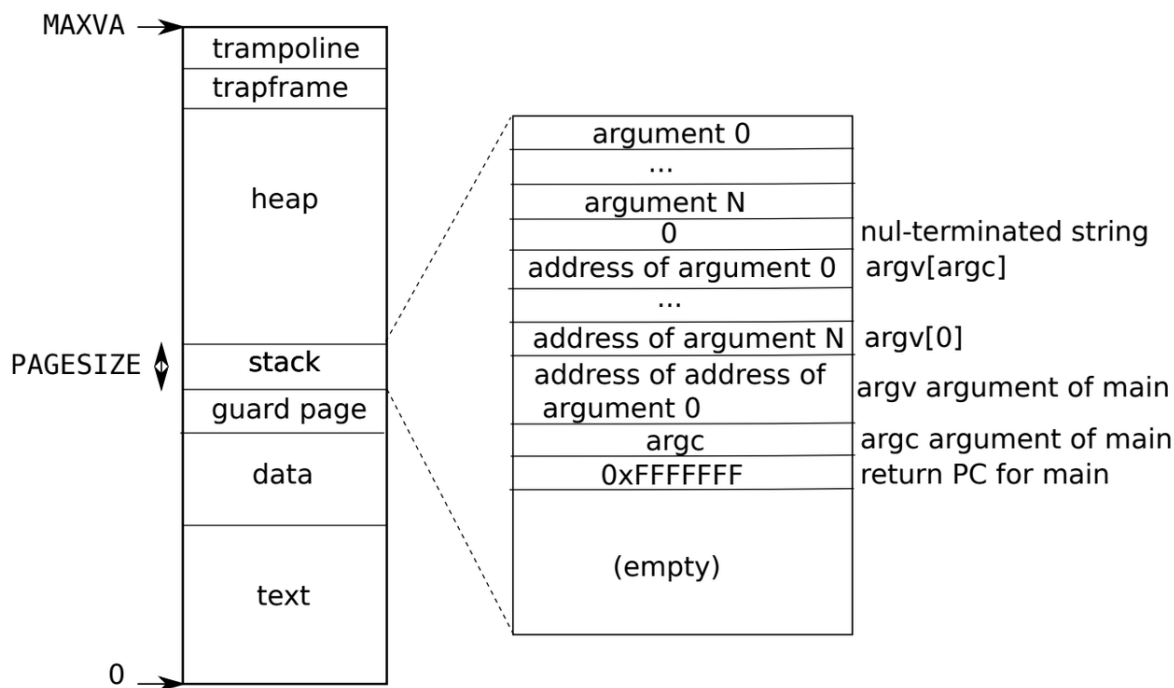


Figure 3.4: A process's user address space, with its initial stack. CSDN @up-to-star

在栈的下方有一个无效的保护页(guard page)。如果用户栈溢出并且进程试图使用栈下方的地址，那么由于映射无效（`PTE_V` 为 0）硬件将生成一个页面故障异常。实际操作时可能会分配更多的物理空间。

`sbrk` 是一个用于进程减少或增长其内存的系统调用。这个系统调用由函数 `growproc` 实现 (`kernel/proc.c`)。 `growproc` 根据 `n` 是正的还是负的调用 `uvmalloc` 或 `uvmdalloc`。  
`uvmalloc` (`kernel/vm.c:229`) 用 `kalloc` 分配物理内存，并用 `mappages` 将 PTE 添加到用户页表中。  
`uvmdalloc` 调用 `uvmunmap` (`kernel/vm.c:174`)， `uvmunmap` 使用 `walk` 来查找对应的 PTE，并使用 `kfree` 来释放 PTE 引用的物理内存。

## 要求

页表。

废弃的合并步骤：

```
1 | git merge origin2/pgtbl #不是1是l
2 | sudo chmod -R 777 /home/starsinhands/xv6-labs-2021 #按照自己的路径,方便vscode有权限操作
3 | # usys.pl, usys.h, syscall.c, Makefile 保留双方
4 | # syscall.h, lab.mk 用新的
5 | git add 上面那几个文件
6 | git commit -m "... " #git push
```

直接切换分支，将笔记 `ctrl cv` 一下，或者迁移笔记所在分支。这里选择后者。

具体关键操作大致为：

```
1 | git push -u origin master:notes #只存笔记的本地目录下
```

github 上修改主要分支为 `notes`。此时原 `master` 分支为本来的 `lab1+lab2`，`pgtbl` 分支为 `lab3+notes(update to lab2)`。

因为 lab2 合并到 lab3 不便，所以放弃了一直更新，直接 lab3 重开，丢弃 lab1+lab2+lab3 合在一起的想法

所以最后直接按照提示暴力合并即可，即：

```
1 git fetch origin2
2 git checkout origin2/pgtbl
3 make clean #清除lab2遗留，不然编译挂
4 make qemu #检验可以编译
```

## speed up system calls

### 要求

Linux 等操作系统通过在用户态和内核态间共享只读区域来加速系统调用。因为这样可以避免用户态内核态切换，所以能提高速度。

将 mappings 插入到页表(page table)，实现 `getpid()` 系统调用的优化。

进程创建时，映射一个只读页，使用定义在 `kernel/memlayout.h` 的 `USYSCALL` 虚拟地址(VA)常量。同文件内定义了结构体 `struct usyscall`，有 `int pid` 一个成员。用户态的 `ugetpid()` 已经提供了，需要通过 `pgtbltest` 调用，预期输出 `ugetpid_test: OK`。

提示：

- 通过 `kernel/proc.c` 执行映射，使用 `proc_pagetable()` 函数。
- 选择权限位，使得用户态只能读该页。
- 使用 `mappages()` 功能。
- 在 `allocproc()` 分配和初始化页。
- 释放页，通过 `freeproc()`

阅读 `proc_pagetable()` 函数，发现调用了 `kernel/vm.c` 的 `mappages()` 函数，创建从 `va` 开始的虚拟地址的页表项(Page Table Entry, PTE)，指向从 `pa` 开始的物理地址。页大小是 `kernel/riscv.h` 定义的 `PGSIZE` 为 4096 bytes。成功返回 0。

一个页表 `pagetable_t` 是一个 `uint64`，存储 512 个页表项。因为  $512 \times 8 = 4096$ ，易得。该函数先新建一个页表，然后尝试将 `TRAMPOLINE` (意为蹦床)映射，若失败删除新建的页表；然后将 `TRAPFRAME` 映射，若失败，取消上一个映射，清空页表。

传入的参数是 `struct proc* p`，表示进程，在 `kernel/proc.h` 定义。需要用到的有 `trapframe` 成员属性(已经被第二个映射用了)，`pid` 成员属性等。

搜索手册 33 页，可以查到权限位的含义：

- `PTE_V` 是否该 PTE 存在
- `PTE_R` 允许读
- `PTE_W` 允许写
- `PTE_X` 允许 CPU 翻译该页内容并执行
- `PTE_U` 用户态是否可以用这个页

## 实现

[错误参考](#) [参考](#) [同义参考](#)

`memlayout.h` 无需就地添加 `#define LAB_PGTBL`，虽然本地项目看到 `#ifdef` 没亮，但是如果 `#define` 一下编译就说重定义了。

失败品：

在 `proc_pagetable()` 函数，在两个映射中间，定义一个 `usyscall` 结构体，存储传入的 `p->pid`，具体而言：

```
1 struct usyscall* usc = (struct usyscall *)kalloc();
2 usc->pid = p->pid;
```

根据提示，仿照上下文，知道需要将 `USYSCALL` 映射。仿照上下文，映射过去的用户态物理地址是 `usc` 的地址，需要的权限是 `PTE_R` 和 `PTE_U`，所以使用：

```
1 if(mappages(pagetable, USYSCALL, PGSIZE, (uint64)usc, PTE_R | PTE_U) < 0)
{
```

模仿下面映射，知道至少要释放掉第一个 `if` 的内容，即把原本那三行内容照搬。除此之外，还要把刚刚定义的 `usc` 释放掉，使用 `kfree` 函数，因此具体有下面代码在 `if` 内：

```
1 kfree(usc);
2 uvmunmap(pagetable, TRAMPOLINE, 1, 0);
3 uvmfree(pagetable, 0);
4 return 0;
```

因为申请了 `usc` 和 `USYSCALL`，所以下面那个映射失败的话也要把这两个万一清理掉，即增设：

```
1 kfree(usc);
2 uvmunmap(pagetable, USYSCALL, 1, 0);
```

在同代码文件的 `proc_freepagetable` 函数，添加：

```
1 kfree((char *)walkaddr(pagetable, USYSCALL));
2 uvmunmap(pagetable, USYSCALL, 1, 0);
```

其中，`walkaddr` 是 `kernel/vm.c` 定义的函数，传入页表和 `va`，返回 `pa`；若返回 `0` 表示没有这样的 `va` 到 `pa` 的映射。

`kernel/proc.h` 对 `proc` 结构体定义追加：

```
1 struct usyscall* usyscall;
```

对 `kernel/proc.c` 的 `allocproc` 函数，模仿同函数内 `p->trapframe` 的初始化，其初始化后添加：

```

1  if((p->usyscall = (struct usyscall *)kalloc()) == 0){
2      freeproc(p);
3      release(&p->lock);
4      return 0;
5  }
6  p->usyscall->pid = p->pid;

```

同文件的 `freeproc()`，模仿 `trapframe`，添加：

```

1  if(p->usyscall)
2      kfree((void*)p->usyscall);
3  p->usyscall = 0;

```

对 `kernel/proc.c` 的 `proc_pagetable` 函数，模仿，`return` 前追加：

```

1  if(mappages(pagetable, USYSCALL, PGSIZE,
2      (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
3      uvmunmap(pagetable, TRAMPOLINE, 1, 0);
4      uvmunmap(pagetable, TRAPFRAME, 1, 0);
5      uvmfree(pagetable, 0);
6      return 0;
7  }

```

在同文件的 `proc_freepagetable()` 取消映射：

```

1  uvmunmap(pagetable, USYSCALL, 1, 0);

```

至此，通过测试。

## Print a page table

### 要求

Define a function called `vmprint()`. It should take a `pagetable_t` argument, and print that pagetable in the format described below. Insert `if(p->pid==1) vmprint(p->pagetable)` in `exec.c` just before the `return argc`, to print the first process's page table. You receive full credit for this assignment if you pass the `pte printout` test of `make grade`.

打印一个页表的内容。定义一个函数 `vmprint()`，将 `pagetable_t` 作为参数。执行函数为 `exec.c` 的方法如下，在 `return argc` 前插入：

```

1  if(p->pid==1) vmprint(p->pagetable)

```

将输出第一个进程的页表。需要通过 `pte printout` 测试。具体而言执行：

```

1  ./grade-lab-pgtbl pte printout

```

按如下格式输出：

```

1 | page table 0x0000000087f6e000
2 | ..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
3 | .. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
4 | .. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
5 | .. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
6 | .. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
7 | ..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
8 | .. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
9 | .. .. ..509: pte 0x0000000021fdd813 pa 0x0000000087f76000
10 | .. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
11 | .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000

```

第一行是 `vmprint` 的参数。对每个 PTE(页表项)按树级输出，输出信息是：页表项的下标、PTE 的位(虚拟地址大概)、物理地址。不要输出无效的 PTE 项。第一级树从 0 到 255；0 的儿子只有 0，其子为 0,1,2。

提示：

- 将 `vmprint()` 放在 `kernel/vm.c`
- 使用 `kernel/riscv.h` 尾部定义的宏
- 参考 `freewalk` 函数
- 在 `kernel/defs.h` 定义 `vmprint` 的原型，以让 `exec.c` 可以调用
- 使用 `%p` 输出虚拟地址。

实现

### 参考

在 `kernel/defs.h` 添加：

```

1 | // vm.c
2 | int vmprint(pagetable_t);

```

按要求在 `kernel/exec.c` 的 `exec` 函数的 `return argc;` 前：

```

1 | vmprint(p->pagetable);

```

对 `kernel/vm.c` 添加：

```

1 | int pgtblprint(pagetable_t pagetable, int depth) {
2 |     // there are 2^9 = 512 PTEs in a page table.
3 |     for(int i = 0; i < 512; i++){
4 |         pte_t pte = pagetable[i];
5 |         if(pte & PTE_V) { // valid
6 |             printf("..  
7 |             for(int j=0;j<depth;j++) {
8 |                 printf("..  
9 |             }
10 |             printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
11 |
12 |             // not leaf
13 |             if((pte & (PTE_R|PTE_W|PTE_X)) == 0){

```

```

14         // this PTE points to a lower-level page table.
15         uint64 child = PTE2PA(pte);
16         pgtblprint((pagetable_t)child, depth+1);
17     }
18 }
19 }
20 return 0;
21 }
22
23 int vmprint(pagetable_t pagetable) {
24     printf("page table %p\n", pagetable);
25     return pgtblprint(pagetable, 0);
26 }

```

代码解释: `pte_t` 是 `uint64`。是否有效 `PTE_V`。 `PTE2PA` 是定义在 `riscv.h` 的宏函数。如果不是叶子节点, `pte_t` 本身代表一个页表, 如果是叶子节点, 才代表实际有作用的一个页。所以递归直接把 `pte_t` 作为新的 `pagetable_t` DFS 即可。如果不存在读写执行这样的权限位, 代表不是叶子节点。

执行测试, 结果通过。

## Detecting which pages have been accessed

### 要求

垃圾回收器需要使用哪个页面被访问(读/写)的信息。PTE 会标记位, 当解决 TLB(Translation Lookaside Buffer, 快表, 存储最近使用的虚拟地址到物理地址的映射)缺失时。

实现系统调用 `pgaccess()`, 报告哪些页被访问, 使用三个参数:

- 第一个用户页的起始虚拟地址
- 要检查的页数
- 存储结果的用户页地址(使用位掩码, 一个位一个页, LSB Least significant bit 是第一页。如, 1101, 最右的 1 是最低有效位, 相反是 MSB most significant bit, 如最左是 1。大端存储 big endian LSB 存在最低内存地址, MSB 在最高; 小端 LSB 最低, MSB 最高, 如 0x1234 这个值, 大端存储 0x0 地址存 12、0x1 存 34, 符合阅读顺序; 小端存储 little endian)

提示:

- 实现 `kernel/sysproc.c` 的 `sys_pgaccess()`
- 使用 `argaddr()`, `argint()` 获取参数
- 输出先存在核态临时缓存, 然后用 `copyout()` 复制知道用户态
- 可以设置可以扫描的页数的上限
- `kernel/vm.c` 的 `walk()` 可以找到正确的 PTE
- 定义 `PTE_A` 权限位, 在 `kernel/riscv.h`, 查手册看具体定义
- 检查后清理 `PTE_A` 位
- `vmprint()` 可以方便 debug

在第 33 页, 说明 A 位是 accessed, 表明自从上次该位被清除以来, 该页面是否被访问过。([参考](#))

## 实现

在 xv6 设计里，用户进程在用户态使用各自的用户态页表，但是一旦进入内核态（例如使用了系统调用），则切换到内核页表，已经定义了全局共享的页表，在 `vm.c` 可见已有如下：

```
1 | pagetable_t kernel_pagetable;
```

仿照 `riscv.h` 里对 `PTE_?` 的定义，阅读 33 页图表，可知 `PTE_A` 是第六位，即：

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

在 `kernel/defs.h` 添加对 `walk` 的定义：

```
1 | pte_t * walk(pagetable_t, uint64, int);
```

在 `kernel/sysproc.c`，找到 `int sys_pgaccess(void)`，函数体实现：

```
1 | int pgaccess(pagetable_t pagetable, uint64 start_va, int page_num, uint64
2 | result_va)
3 | {
4 |     if (page_num > 64)
5 |     {
6 |         panic("pgaccess: too much pages");
7 |         return -1;
8 |     }
9 |     unsigned int bitmask = 0;
10 |    int cur_bitmask = 1;
11 |    int count = 0;
12 |    uint64 va = start_va;
13 |    pte_t *pte;
14 |    for (; count < page_num; count++, va += PGSIZE)
15 |    {
16 |        if ((pte = walk(pagetable, va, 0)) == 0)
17 |            panic("pgaccess: pte should exist");
18 |        if ((*pte & PTE_A))
19 |        {
20 |            bitmask |= (cur_bitmask << count);
21 |            *pte &= ~PTE_A;
22 |        }
23 |    }
24 |    copyout(pagetable, result_va, (char*)&bitmask, sizeof(bitmask));
25 |    return 0;
26 | }
27 | #ifdef LAB_PGTBL
28 | int
29 | sys_pgaccess(void)
30 | {
31 |     // lab pgtbl: your code here.
32 |     uint64 va;
33 |     if(argaddr(0, &va) < 0) { // 0th param is va
34 |         return -1;
```

```

35     }
36     int num;
37     if(argint(1,&num)<0){//1st param is num
38         return -1;
39     }
40     uint64 res;
41     if(argaddr(2,&res)<0){
42         return -1;
43     }
44     struct proc* p = myproc();
45     if(pgaccess(p->pagetable, va, num, res) < 0) {
46         return -1;
47     }
48     return 0;
49 }
50 #endif

```

代码解释：

- 参考同文件其他函数获取参数的方式，知道 `argint()` 取 `int`，`argaddr()` 取 `uint64`。
- 之后使用 `myproc()` 获取当前运行进程，取它的页表，进入核心程序。
- 遍历从 `va` 开始的每一页，页大小是 `PGSIZE`，可以指针加法。进行一个 `for`。
- `walk()` 函数根据页表项虚拟地址返回页表项指针代表页表内容(一个 `uint64` 对应一些信息)。根据手册 36 页，可以从 111 页后查表找到，从 PTE 找地址。如果设置了 `alloc` 参数，找不到就分配新的页表。否则找不到就返回 0。
- 每次遍历 `va`，遍历页表找到页表项，如果找到了且有 `PTE_A` 位，即页表项内容的 `1<<6` 位，即 `*pte & PTE_A` 是 1，那么这一位访问过，记录结果，结果的第 `i` 低位标 1。  
并且，有 `PTE_A` 就清空这个位。这个位的意思是上次询问以来是否访问过。因为询问更新了，现在就是下一个上次询问，所以这个位要清零。
- 阅读 `vm.c` 的定义，参考 `proc.c` 对其的调用，观察可知 `copyout` 的用法，传入当前进程页表，传入调用参数给的存结果的虚拟地址，将数据转成字符串，给定长度，转到对应地方去。

测试，预期全部成功：

```

1 make qemu
2 pgtbltest

```

更多选做任务：

- 使用超级页来减少页表的 PTE
- 从地址 `0x0` 到 `0xFFF` 是物理第一页。要求从第二个物理页开始用，对应第一个虚拟页。这是因为 `0x0` 可以代表空指针，这么做后指向这一页可以意味着空指针越界。
  - **Unmap the first page of a user process:** 在虚拟内存系统中，内存被分为大小固定的“页”（通常是 4KB 或其他大小）。“取消映射第一页”意味着使用户进程的第一页不与任何实际的物理内存关联。这样，如果用户进程试图访问这一页的任何地址，它将无法找到对应的物理内存。

- **dereferencing a null pointer will result in a fault:** 在大多数编程语言中，空指针（或NULL指针）通常表示为地址 `0x0`。如果程序错误地尝试访问（或解引用）一个空指针，由于第一页已被取消映射，该访问将导致一个“错误”或“异常”（通常称为段错误或访问违规）。
- **You will have to start the user text segment at, for example, 4096, instead of 0:** 由于第一页（从地址 `0x0` 到 `0xFFFF`，总共4KB）已被取消映射，用户进程的代码（也称为“text segment”）必须从第二页开始，即地址 `0x1000`（十进制为4096）。这意味着程序的实际执行代码不再从地址 `0x0` 开始，而是从 `0x1000` 开始。
- 添加系统调用来报告脏页，使用 `PTE_D`

## lab4 traps

[参考](#) 部署：

```
1 git checkout origin2/traps
2 make clean
3 make qemu
4 sudo chmod -R 777 /home/starsinhands/xv6-labs-2021
5 #之后push用这个
6 git push origin HEAD:refs/heads/traps
```

### 基础知识

**系统调用、异常和中断：**有三种事件会导致CPU搁置普通指令的执行，强制将控制权转移给处理该事件的特殊代码。一种情况是**系统调用**，当用户程序执行`ecall`指令要求内核为其做某事时。另一种情况是**异常**：一条指令（用户或内核）做了一些非法的事情，如除以零或使用无效的虚拟地址。第三种情况是设备**中断**，当一个设备发出需要注意的信号时，例如当磁盘硬件完成一个读写请求时。

xv6 book使用**trap**作为这些情况的通用术语。通常，代码在执行时发生trap，之后都会被恢复，而且不需要意识到发生了什么特殊的事情。也就是说，我们通常希望trap是透明的；这一点对于中断来说尤其重要，被中断的代码通常不会意识到会发生trap。通常的顺序是：trap迫使控制权转移到内核；内核保存寄存器和其他状态，以便恢复执行；内核执行适当的处理程序代码（例如，系统调用实现或设备驱动程序）；内核恢复保存的状态，并从trap中返回；代码从原来的地方恢复执行。

Xv6 trap 处理分为四个阶段：RISC-V CPU采取的硬件行为，为内核C代码准备的汇编入口，处理trap的C处理程序，以及系统调用或设备驱动服务。

**RISC-V trap machinery：**每个RISC-V CPU都有一组控制寄存器，内核写入这些寄存器来告诉CPU如何处理trap，内核可以通过读取这些寄存器来发现已经发生的trap。RISC-V文档包含了完整的叙述。

`risvc.h` (`kernel/risvc.h:1`) 包含了xv6使用的定义。这里是最重要的寄存器的概述。

- `stvec`：内核在这里写下trap处理程序的地址；RISC-V跳转到这里来处理trap。
- `sepc`：当trap发生时，RISC-V会将程序计数器保存在这里（因为 `PC` 会被 `stvec` 覆盖）。`sret`（从trap中返回）指令将 `sepc` 复制到`pc`中。内核可以写 `sepc` 来控制 `sret` 的返回到哪里。
- `scause`：RISC-V在这里放了一个数字，描述了trap的原因。
- `sscratch`：内核在这里放置了一个值，在trap处理程序开始时可以方便地使用。
- `sstatus`：`sstatus` 中的**SIE**位控制设备中断是否被启用，如果内核清除**SIE**，RISC-V将推迟设备中断，直到内核设置**SIE**。**SPP**位表示trap是来自用户模式还是supervisor模式，并控制 `sret` 返回到什么模式。

当需要执行trap时，RISC-V硬件对所有的trap类型（除定时器中断外）进行以下操作：

1. 如果该trap是设备中断，且 `sstatus` **SIE**位为0，则不执行以下任何操作。
2. 通过清除**SIE**来禁用中断。
3. 复制 `pc` 到 `sepc`。
4. 将当前模式（用户态或特权态）保存在 `sstatus` 的**SPP**位。
5. 在 `scause` 设置该次 trap 的原因。
6. 将模式转换为特权态。
7. 将 `stvec` 复制到 `pc`。
8. 从新的 `pc` 开始执行。

注意，CPU不会切换到内核页表，不会切换到内核中的栈，也不会保存`pc`以外的任何寄存器。内核软件必须执行这些任务。

### Traps from user space

在用户空间执行时，如果用户程序进行了系统调用（`ecall` 指令），或者做了一些非法的事情，或者设备中断，都可能发生trap。来自用户空间的trap的处理路径是 `uservec`（`kernel/trampoline.S:16`），然后是 `usertrap`（`kernel/trap.c:37`）；返回时是 `usertrapret`（`kernel/trap.c:90`），然后是 `userret`（`kernel/trampoline.S:16`）。

RISC-V硬件在trap过程中不切换页表，所以用户页表必须包含 `uservec` 的映射，即 `stvec` 指向的trap处理程序地址。`uservec` 必须切换 `satp`，使其指向内核页表；为了在切换后继续执行指令，`uservec` 必须被映射到内核页表与用户页表相同的地址。

Xv6用一个包含 `uservec` 的trampoline页来满足这些条件。Xv6在内核页表和每个用户页表中的同一个虚拟地址上映射了trampoline页。这个虚拟地址就是 `TRAMPOLINE`（如我们在图2.3和图3.3中看到的）。

`trampoline.S` 中包含trampoline的内容，（执行用户代码时）`stvec` 设置为 `uservec`（`kernel/trampoline.S:16`）。

当 `uservec` 启动时，所有32个寄存器都包含被中断的代码所拥有的值。但是 `uservec` 需要能够修改一些寄存器，以便设置 `satp` 和生成保存寄存器的地址。RISC-V通过 `sscratch` 寄存器提供了帮助。`uservec` 开始时的 `csrrw` 指令将 `a0` 和 `sscratch` 的内容互换。现在用户代码的 `a0` 被保存了；`uservec` 有一个寄存器（`a0`）可以使用；`a0` 包含了内核之前放在 `sscratch` 中的值。

`uservec` 的下一个任务是保存用户寄存器。在进入用户空间之前，内核先设置 `sscratch` 指向该进程的 `trapframe`，这个 `trapframe` 可以保存所有用户寄存器（`kernel/proc.h:44`）。因为 `satp` 仍然是指用户页表，所以 `uservec` 需要将 `trapframe` 映射到用户地址空间中。当创建每个进程时，xv6为进程的 `trapframe` 分配一页内存，并将它映射在用户虚拟地址 `TRAPFRAME`，也就是 `TRAMPOLINE` 的下面。进程的 `p->trapframe` 也指向 `trapframe`，不过是指向它的物理地址，这样内核可以通过内核页表来使用它。

因此，在交换 `a0` 和 `sscratch` 后，`a0` 将指向当前进程的 `trapframe`。`uservec` 将在 `trapframe` 保存全部的寄存器，包括从 `sscratch` 读取的 `a0`。

`trapframe` 包含指向当前进程的内核栈、当前CPU的hartid、`usertrap` 的地址和内核页表的地址的指针，`uservec` 将这些值设置到相应的寄存器中，并将 `satp` 切换到内核页表和刷新TLB，然后调用 `usertrap`。

`usertrap` 的作用是确定trap的原因，处理它，然后返回（`kernel/ trap.c:37`）。如上所述，它首先改变 `stvec`，这样在内核中发生的trap将由 `kernelvec` 处理。它保存了 `sepc`（用户PC），这也是因为 `usertrap` 中可能会有一个进程切换，导致 `sepc` 被覆盖。如果trap是系统调用，`syscall` 会处理它；如果是设备中断，`devintr` 会处理；否则就是异常，内核会杀死故障进程。`usertrap` 会把用户 `pc` 加4，因为RISC-V在执行系统调用时，会留下指向`ecall`指令的程序指针。在退出时，`usertrap`检查进程是否已经被杀死或应该让出CPU（如果这个trap是一个定时器中断）。

回到用户空间的第一步是调用 `usertrapret`（`kernel/trap.c:90`）。这个函数设置RISC-V控制寄存器，为以后用户空间trap做准备。这包括改变 `stvec` 来引用 `uservec`，准备 `uservec` 所依赖的 `trapframe` 字段，并将 `sepc` 设置为先前保存的用户程序计数器。最后，`usertrapret` 在用户页表和内核页表中映射的trampoline页上调用 `userret`，因为 `userret` 中的汇编代码会切换页表。

`usertrapret` 对 `userret` 的调用传递了参数 `a0`，`a1`，`a0` 指向 `TRAPFRAME`，`a1` 指向用户进程页表（`kernel/trampoline.S:88`），`userret` 将 `satp` 切换到进程的用户页表。回想一下，用户页表同时映射了trampoline页和 `TRAPFRAME`，但没有映射内核的其他内容。同样，事实上，在用户页表和内核页表中，trampoline页被映射在相同的虚拟地址上，这也是允许 `uservec` 在改变 `satp` 后继续执行的原因。`userret` 将 `trapframe` 中保存的用户 `a0` 复制到 `sscratch` 中，为以后与 `TRAPFRAME` 交换做准备。从这时开始，`userret` 能使用的数据只有寄存器内容和 `trapframe` 的内容。接下来 `userret` 从 `trapframe` 中恢复保存的用户寄存器，对 `a0` 和 `sscratch` 做最后的交换，恢复用户 `a0` 并保存 `TRAPFRAME`，为下一次trap做准备，并使用 `sret` 返回用户空间。

我的理解就是用户态和内核态共享了两个页，分别是trampoline和trapfram，前者实现trap时对指令的访问，后者实现trap时保存相关参数，实现保护作用。

### Code: System call arguments

内核的系统调用实现需要找到用户代码传递的参数。因为用户代码调用系统调用的包装函数，参数首先会存放在寄存器中，这是C语言存放参数的惯例位置。内核trap代码将用户寄存器保存到当前进程的trap frame中，内核代码可以在那里找到它们。函数 `argint`、`argaddr` 和 `argfd` 从trap frame中以整数、指针或文件描述符的形式检索第n个系统调用参数。它们都调用 `argraw` 来获取保存的用户寄存器（`kernel/syscall.c:35`）。

内核实现了安全地将数据复制到用户提供的地址或从用户提供的地址复制数据的函数。例如 `fetchstr`（`kernel/syscall.c:25`）。文件系统调用，如 `exec`，使用 `fetchstr` 从用户空间中检索字符串文件名参数。`fetchstr` 调用 `copyinstr` 来做这些困难的工作。

`copyinstr`（`kernel/vm.c:406`）将用户页表 `pagetable` 中的虚拟地址 `srcva` 复制到 `dst`，需指定最大复制字节数。它使用 `walkaddr`（调用 `walk` 函数）在软件中模拟分页硬件的操作，以确定 `srcva` 的物理地址 `pa0`。`walkaddr`（`kernel/vm.c:95`）检查用户提供的虚拟地址是否是进程用户地址空间的一部分，所以程序不能欺骗内核读取其他内存。类似的函数 `copyout`，可以将数据从内核复制到用户提供的地址。

### Traps from kernel space

Xv6根据用户还是内核代码正在执行，对CPU陷阱寄存器的配置略有不同行为。当内核在CPU上执行时，内核将 `stvec` 指向 `kernelvec` 上的汇编代码（`kernel/kernelvec.S:10`）。由于xv6已经在内核中，`kernelvec` 可以使用 `satp`，将其设置为内核页表，以及引用有效内核的堆栈指针。`kernelvec` 保存所有寄存器，以便中断的代码最后可以在没有中断的情况下恢复。

`kernelvec` 将寄存器保存在中断内核线程的堆栈上，因为寄存器值属于该线程，这是合理的。如果trap导致切换到另一个线程—在这种情况下，trap将实际返回到新线程的栈上，将中断线程保存的寄存器安全地保留在其堆栈上。

`kernelvec` 在保存寄存器后跳转到 `kerneltrap` (`kernel/trap.c:134`)。 `kerneltrap` 是为两种类型的陷阱准备的：设备中断和异常。它调用 `devintr` (`kernel/trap.c:177`) 来检查和处理前者。如果 `trap` 不是设备中断，那么它必须是异常，如果它发生在 `xv6` 内核中，则一定是一个致命错误；内核调用 `panic` 并停止执行。

**如果由于计时器中断而调用了 `kerneltrap`，并且进程的内核线程正在运行（而不是调度程序线程），`kerneltrap` 调用 `yield` 让出 CPU，允许其他线程运行。在某个时刻，其中一个线程将退出，并让我们的线程及其 `kerneltrap` 恢复。**

当 `kerneltrap` 的工作完成时，它需要返回到被中断的代码。因为 `yield` 可能破坏保存的 `sepc` 和在 `sstatus` 中保存的之前的模式。 `kerneltrap` 在启动时保存它们。它现在恢复那些控制寄存器并返回到 `kernelvec` (`kernel/kernelvec.S:48`)。 `kernelvec` 从堆栈恢复保存的寄存器并执行 `sret`， `sret` 将 `sepc` 复制到 `pc` 并恢复中断的内核代码。

可以思考一下，如果因为时间中断， `kerneltrap` 调用了 `yield`， `trap return` 是如何发生的。

当 CPU 从用户空间进入内核时， `xv6` 将 CPU 的 `stvec` 设置为 `kernelvec`；可以在 `usertrap` (`kernel/trap.c:29`) 中看到这一点。内核运行但 `stvec` 被设置为 `uservec` 时，这期间有一个时间窗口，在这个窗口期，禁用设备中断是至关重要的。幸运的是， `RISC-V` 总是在开始使用 `trap` 时禁用中断， `xv6` 在设置 `stvec` 之前不会再次启用它们。

## Page-fault exceptions

`xv6` 对异常的响应是相当固定：如果一个异常发生在用户空间，内核就会杀死故障进程。如果一个异常发生在内核中，内核就会 `panic`。真正的操作系统通常会以更有趣的方式进行响应。

许多内核使用页面故障来实现**写时复制 (copy-on-write, cow) fork**。 `fork` 通过调用 `uvmcopy` (`kernel/vm.c:309`) 为子进程分配物理内存，并将父进程的内存复制到子程序中，使子进程拥有与父进程相同的内存内容。如果子进程和父进程能够共享父进程的物理内存，效率会更高。然而，直接实现这种方法是行不通的，因为父进程和子进程对共享栈和堆的写入会中断彼此的执行。

通过使用写时复制 `fork`，可以让父进程和子进程安全地共享物理内存，通过页面故障来实现。当 CPU 不能将虚拟地址翻译成物理地址时，CPU 会产生一个页面故障异常 (page-fault exception)。 `RISC-V` 有三种不同的页故障： `load` 页故障（当加载指令不能翻译其虚拟地址时）、 `store` 页故障（当存储指令不能翻译其虚拟地址时）和指令页故障（当指令的地址不能翻译时）。 `scause` 寄存器中的值表示页面故障的类型， `stval` 寄存器中包含无法翻译的地址。

`COW fork` 中的基本设计是父进程和子进程最初共享所有的物理页面，但将它们映射设置为只读。因此，当子进程或父进程执行 `store` 指令时， `RISC-V CPU` 会引发一个页面故障异常。作为对这个异常的响应，内核会拷贝一份包含故障地址的页。然后将一个副本的读/写映射在子进程地址空间，另一个副本的读/写映射在父进程地址空间。更新页表后，内核在引起故障的指令处恢复故障处理。因为内核已经更新了相关的 `PTE`，允许写入，所以现在故障指令将正常执行。

这个 `COW` 设计对 `fork` 很有效，因为往往子程序在 `fork` 后立即调用 `exec`，用新的地址空间替换其地址空间。在这种常见的情况下，子程序只会遇到一些页面故障，而内核可以避免进行完整的复制。此外， `COW fork` 是透明的：不需要对应用程序进行修改，应用程序就能受益。

页表和页故障的结合，将会有更多种有趣的可能性的应用。另一个被广泛使用的特性叫做**懒分配 (lazy allocation)**，它有两个部分。首先，当一个应用程序调用 `sbrk` 时，内核会增长地址空间，但在页表中把新的地址标记为无效。第二，当这些新地址中的一个出现页面故障时，内核分配物理内存并将其映射到页表中。由于应用程序经常要求获得比他们需要的更多的内存，所以懒分配是一个胜利：内核只在应用程序实际使用时才分配内存。像 `COW fork` 一样，内核可以对应用程序透明地实现这个功能。

另一个被广泛使用的利用页面故障的功能是从**磁盘上分页(paging from disk)**。如果应用程序需要的内存超过了可用的物理RAM，内核可以交换出一些页：将它们写入一个存储设备，比如磁盘，并将其PTE标记为无效。如果一个应用程序读取或写入一个被换出到磁盘的页，CPU将遇到一个页面故障。内核就可以检查故障地址。如果该地址属于磁盘上的页面，内核就会分配一个物理内存的页面，从磁盘上读取页面到该内存，更新PTE为有效并引用该内存，然后恢复应用程序。为了给该页腾出空间，内核可能要交换另一个页。这个特性不需要对应用程序进行任何修改，如果应用程序具有引用的位置性（即它们在任何时候都只使用其内存的一个子集），这个特性就能很好地发挥作用。

## RISC-V assembly

### 要求

汇编理解，以 `user/call.c` 为例，通过 `make fs.img` 编译生成汇编程序 `user/call.asm`，阅读汇编程序，[官方参考资料](#)，熟悉RISC-V的一些汇编指令、寄存器等，回答问题：（下面用A来代替Answer）

1. 存储参数的寄存器是哪个，`printf` 里传入值 13 放哪里？A: a0-a7, a2
2. 在哪里调用了函数 `f` 和 `g`？A: 没有这样的代码。`g(x)` 被内链到 `f(x)` 中，然后 `f(x)` 又被进一步内链到 `main()` 中
3. `printf` 的地址在哪？A: 0x00000000000000628, `main` 中使用 `pc` 相对寻址来计算得到这个地址。
4. 在 `printf` 的 `jalr` 后，`ra` 的值是什么？A: 0x0000000000000038, `jalr` 指令的下一条汇编指令的地址。
5. 求这段代码的输出：

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

参考 ASCII 表。且 RISC-V 是**小端存储**。

如果是大端存储，设置 `i` 为何值能得到同样的输出？57616 需要改变吗？

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）

6. `printf("x=%d y=%d", 3);` 这行代码会输出什么，发生什么？

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。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，而是会包含调用发生前的任何已经在里面的值。

`call.c` 如下：

```
1 #include "kernel/param.h"  
2 #include "kernel/types.h"  
3 #include "kernel/stat.h"  
4 #include "user/user.h"  
5  
6 int g(int x) {  
7     return x+3;  
8 }  
9  
10 int f(int x) {
```

```

11  return g(x);
12  }
13
14  void main(void) {
15  printf("%d %d\n", f(8)+1, 13);
16  exit(0);
17  }

```

call.asm 如下: (部分)

```

1
2  user/_call:      file format elf64-littleriscv
3
4
5  Disassembly of section .text:
6
7  0000000000000000 <g>:
8  #include "kernel/param.h"
9  #include "kernel/types.h"
10 #include "kernel/stat.h"
11 #include "user/user.h"
12
13 int g(int x) {
14 0: 1141          addi    sp,sp,-16
15 2: e422          sd     s0,8(sp)
16 4: 0800          addi    s0,sp,16
17 return x+3;
18 }
19 6: 250d          addiw   a0,a0,3
20 8: 6422          ld      s0,8(sp)
21 a: 0141          addi    sp,sp,16
22 c: 8082          ret
23
24 000000000000000e <f>:
25
26 int f(int x) {
27 e: 1141          addi    sp,sp,-16
28 10: e422          sd     s0,8(sp)
29 12: 0800          addi    s0,sp,16
30 return g(x);
31 }
32 14: 250d          addiw   a0,a0,3
33 16: 6422          ld      s0,8(sp)
34 18: 0141          addi    sp,sp,16
35 1a: 8082          ret
36
37 000000000000001c <main>:
38
39 void main(void) {
40 1c: 1141          addi    sp,sp,-16
41 1e: e406          sd     ra,8(sp)
42 20: e022          sd     s0,0(sp)
43 22: 0800          addi    s0,sp,16
44 printf("%d %d\n", f(8)+1, 13);
45 24: 4635          li      a2,13

```

```

46 26: 45b1                li a1,12
47 28: 00000517            auipc a0,0x0
48 2c: 7b050513            addi a0,a0,1968 # 7d8 <malloc+0xea>
49 30: 00000097            auipc ra,0x0
50 34: 600080e7            jalr 1536(ra) # 630 <printf>
51 exit(0);
52 38: 4501                li a0,0
53 3a: 00000097            auipc ra,0x0
54 3e: 27e080e7            jalr 638(ra) # 2b8 <exit>

```

## 实现

g() 解读:

- 0: 从栈指针减去 16, 为局部变量、寄存器分配空间
- 2: s0 寄存器值保存到栈地址 sp+8
- 4: 栈指针 s0 设为当前栈顶
- 6: 将参数 x 运算 +3 的值存到寄存器 a0, 表示第一个参数和返回值
- 8: 加载 sp+8 的值
- a: 栈指针增加, 释放函数栈空间
- c: 返回到调用者

s0 作用是保护原始值, 将 sp 本来的值存起来, 用完了恢复。s0 是一个 callee-saved 寄存器, 函数在返回之前需要恢复其原始值。

f() 的汇编代码与 g() 完全相同, 不再赘述

对 main, 相似代码不再赘述。有:

- 1e: 注意 ra 是返回地址寄存器
- 24: 注意先填入第二个参数, 立即数 13, 填入 a2
- 26: 第一个参数, 编译优化, 直接算出结果为 12 填入 a1
- 28,2c: 求 printf 函数的地址, 返回到 a0
- 30,34: 调用 printf
- 38: 填入 exit 的参数为立即数 0
- 3a, 3e: 调用 exit

根据上文逐行分析:

1. 函数参数是 a0-a7, 如果多于 8 个参数, 使用堆栈传递  
立即数 13 填入了寄存器 a2
2. 没有任何地方调用了 f,g, 通过 inline 优化掉了。即编译阶段运算优化
3. auipc a0,0x0 将 0x0 与 PC 相加, 存到 a0  
addi 将其增加了 1968。

但是更直接的办法是阅读注释 # 630 <printf>, 即地址是 0x630

jalr 是 RISC-V 指令集中的一个汇编指令, 全称为 "Jump and Link Register"。这是一个寄存器间接跳转指令, 它的功能是跳转到寄存器中指定的地址, 并将返回地址保存到一个指定的寄存器中。

4. 指向下一条指令的地址，即 38

5. 输出 `He110 wor1d`

`%x` 是十六进制小写， $(57616)_{10} = (e110)_{16}$ 。 `&i` 是输出 `i` 变量所在地址开始表示的字符串。

`chr(0x72)=r, chr(0x6c)=l, chr(0x64)=d, chr(0x00)=\0`，小端所以 `%s` 对 `r1d`。

如果大端就 `0x726c6400`，不需要改变 `57616`。

6. 输出 `a2` 寄存器的值。

## backtrace

### 要求

For debugging it is often useful to have a backtrace: a list of the function calls on the stack above the point at which the error occurred.

在 `kernel/printf.c` 实现 `backtrace()` 函数，在 `sys_sleep` 里调用它。然后执行 `bttest`，将调用 `sys_sleep`。输出应长这样：

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

或者执行：

```
1 addr2line -e kernel/kernel
2 # 或
3 riscv64-unknown-elf-addr2line -e kernel/kernel
```

然后将下面文本粘贴：

```
1 0x0000000080002de2
2 0x0000000080002f4a
3 0x0000000080002bfc
4 Ctrl+D
```

这预期输出：

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

编译器在每个堆栈帧中放置一个帧指针，该指针保存调用者的帧指针的地址。要用这些指针遍历栈，输出帧保存的地址。

提示：

- 在 `kernel/defs.h` 添加函数原型。
- GCC 编译器存储当前执行函数的帧指针在寄存器 `s0`，在 `kernel/riscv.h` 添加如下函数：

```

1 static inline uint64
2 r_fp()
3 {
4     uint64 x;
5     asm volatile("mv %0, s0" : "=r" (x) );
6     return x;
7 }

```

其中 inline 是 C99 引入的，建议编译器将函数的代码直接嵌入到调用它的地方，以减少函数调用的开销。但这只是一个建议，编译器可以选择忽略它。使用 inline 的函数应该在头文件中定义。与 C++ 对比，C++ 必然改变而不只是建议，C++ 类成员函数默认 inline，C++ 可以头文件只定义。

static 函数表明 函数在其所在的源文件中是“私有”的，这意味着它不能被其他源文件访问或链接。这有助于封装和隐藏实现细节。

asm volatile 是内联汇编语句，它允许你在 C 代码中直接写汇编代码，其中 volatile 告诉编译器不要优化这段汇编代码，确保它按照你写的方式执行，mv 将 s0 寄存器值给占位符表示的变量。=r 是通用寄存器。

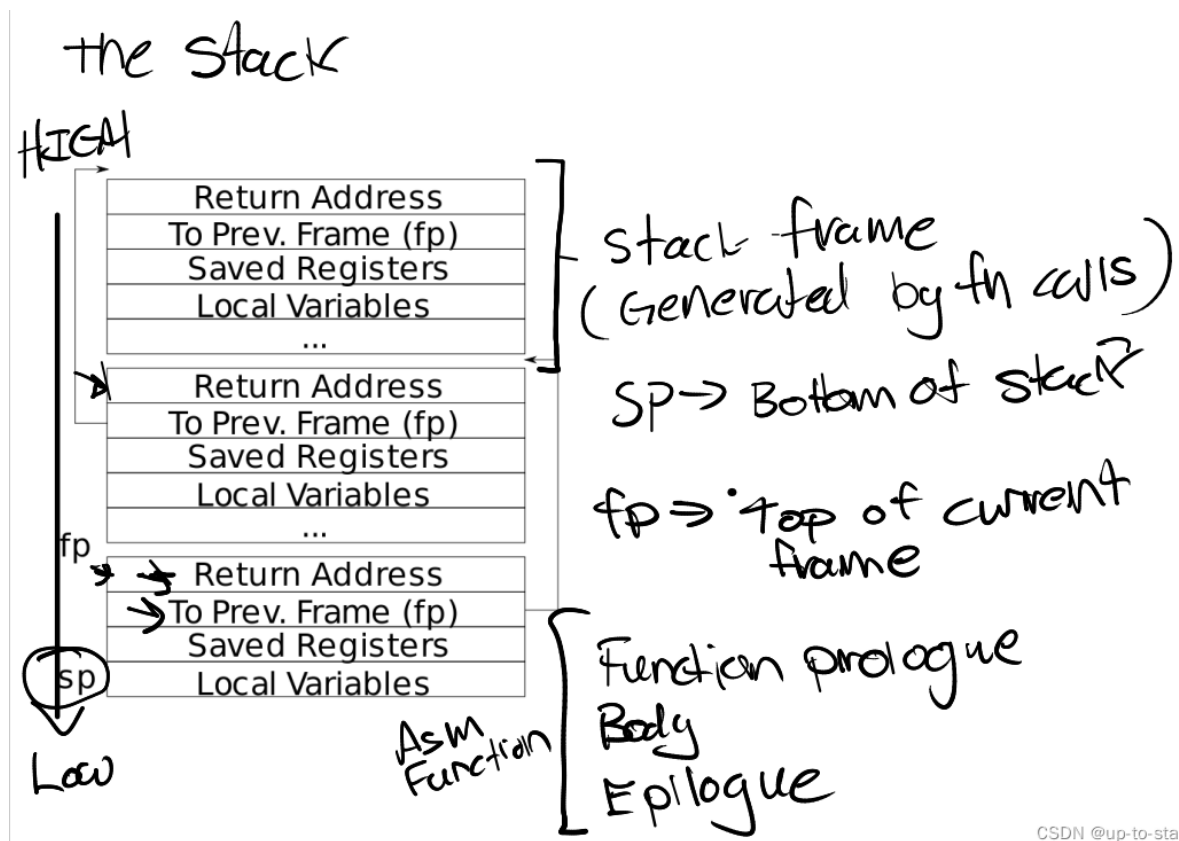
其他地方也可以用 volatile，对变量跟 java 类似。可以防止编译器对这些变量进行某些优化，确保每次访问都是直接从内存中读取，而不是从寄存器或其他临时存储位置。若对指针，通过这个指针访问的数据不应该被优化。

调用该函数可以读取当前 s0，代表当前栈指针。[官方参考](#)

- 这个 [pdf](#) 描述了栈帧示意图。返回地址固定占用偏移 -8，保存的帧指针固定 -16。(8 为单位，对应 long long)
- xv6 给每个栈分配一页，在 xv6 内核，用页对齐地址。可以计算出顶部底部地址，通过 kernel/riscv.h 的 PGROUNDDOWN, PGROUNDUP 上下取整找到一个页的最大最小地址。可以用来检测循环边界。

从 kernel.printf.c 的 panic 调用，可以看到 panic 时的错误回溯。

提示中给了一个栈的示意图：



fp指向当前栈帧的开始地址，sp指向当前栈帧的结束地址，栈从高地址向低地址增长。

栈帧中从高地址到低地址第一个8字节fp-8是return address, 当前调用层的返回地址

栈帧中从高地址到低地址第二个8字节fp-16是previous adress, 即调用当前函数的上一层栈帧的fp开始地址。（这也是第三条提示的内容）

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

## 实现

defs.h 里在 printf.c 注释下添加：

```
1 void backtrace(void);
```

按照提示在 riscv.h 添加：

```
1 static inline uint64
2 r_fp()
3 {
4     uint64 x;
5     asm volatile("mv %0, s0" : "=r" (x) );
6     return x;
7 }
```

注意栈的生长方向是从高地址到低地址，所以扩张是 -16，而回收是 +16。所以在 printf.c 遍历即可：

```
1 void backtrace() {
2     uint64 fp = r_fp(); // get s0
3     while(fp != PGROUNDUP(fp)) { // while not reach bottom of stack
4         uint64 ra = *(uint64*)(fp - 8); // return address
5         printf("%p\n", ra);
6         fp = *(uint64*)(fp - 16); // previous fp
7     }
8 }
```

在 sysproc.c 的 sys\_sleep 函数，函数体首部追加：

```
1 backtrace();
```

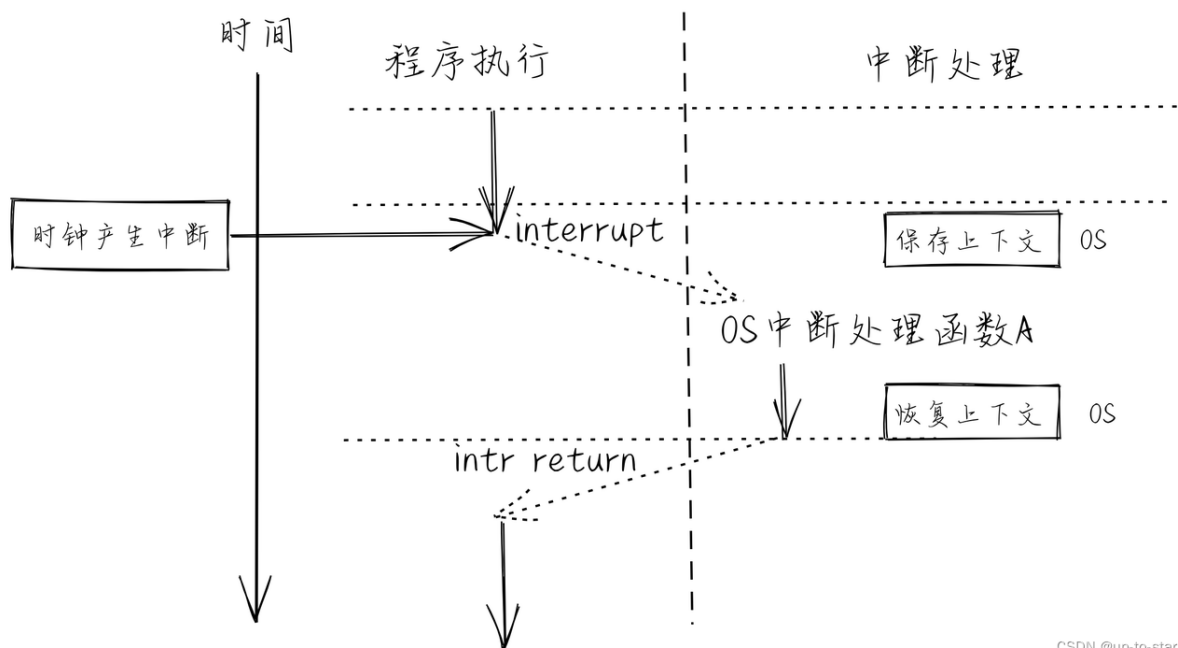
执行测试，通过：

```
1 make qemu
2 bttest
```

## alarm

### 要求

In this exercise you'll add a feature to xv6 that periodically alerts a process as it uses CPU time. This might be useful for compute-bound processes that want to limit how much CPU time they chew up, or for processes that want to compute but also want to take some periodic action. More generally, you'll be implementing a primitive form of user-level interrupt/fault handlers; you could use something similar to handle page faults in the application, for example. Your solution is correct if it passes `alarmtest` and `usertests`. 这个实验目的是利用时钟中断计时，根据`sigalarm`中的参数，每隔一定时间输出alarm。



CSDN @up-to-star

深入了解一下trap机制：在trap过程中有很多特殊的寄存器

- 在硬件中还有一个寄存器叫做程序计数器（Program Counter Register）。
- 表明当前mode的标志位，这个标志位表明了当前是supervisor mode还是user mode。当我们在运行Shell的时候，自然是在user mode。
- 还有一堆控制CPU工作方式的寄存器，比如SATP（Supervisor Address Translation and Protection）寄存器，它包含了指向page table的物理内存地址。
- 还有一些对于今天讨论非常重要的寄存器，比如STVEC（Supervisor Trap Vector Base Address Register）寄存器，它指向了内核中处理trap的指令的起始地址。
- SEPC（Supervisor Exception Program Counter）寄存器，在trap的过程中保存程序计数器的值，保存当发生trap时的程序计数器。
- SSRATCH（Supervisor Scratch Register）寄存器

增加一个功能，周期性输出信息。对计算结合的进程有用，需要限制 CPU 时间时，或者需要做周期任务的进程。实现一个用户级中断/错误处理器，处理页级错误。需要通过 `alarmtest` 和 `usertests`。

添加 `sigalarm(interval, handler)` 系统调用，如果使用了 `sigalarm(n, fn)` 调用，则每 `n` 刻 (ticks) CPU 时间调用一次 `fn` 函数，该函数返回时，从上次中断的地方继续执行。一个 tick 是一个由硬件计时器产生中断的频率决定的时间单位。若调用 `sigalarm(0, 0)`，停止产生周期调用。

可以找到 `user/alarmtest.c`，将调用 `sigalarm(2, periodic)`，可以阅读汇编代码来 debug。输出预期如下：

```
1  $ alarmtest
2  test0 start
3  .....alarm!
4  test0 passed
5  test1 start
6  ...alarm!
7  ..alarm!
8  ...alarm!
9  ..alarm!
10 ...alarm!
11 ..alarm!
12 ...alarm!
13 ..alarm!
14 ...alarm!
15 ..alarm!
16 test1 passed
17 test2 start
18 .....alarm!
19 test2 passed
20 $ usertests
21 ...
22 ALL TESTS PASSED
```

其中，`test0` 是触发调用程序测试

函数名前添加 `__attribute__((noinline))` 告诉编译器不进行内联编译优化。

提示：

- 修改 `Makefile`，编译 `alarmtest.c`
- 在 `user/user.h` 添加：

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

- 修改 `user/usys.pl` 和 `kernel/syscall.h` 和 `kernel/syscall.c`，让 `alarmtest` 允许调用 `sigalarm`
- 截止目前 `sys_sigreturn` 只执行 `return 0`
- `sys_sigalarm()` 存储蜂鸣区间和指针给执行函数，使用 `kernel/proc.h` 进程结构体
- 记录上次调用以来经过的 tick 数(或还有多少个 tick 到下一次调用)，修改 `struct proc`，修改 `proc.c` 的初始化，在 `allocproc()` 函数。
- 每个 tick 硬件强制中断一次，将通过 `kernel/trap.c` 的 `usertrap()` 处理
- 只处理进程计时器中端，如 `if(which_dev == 2) ...`
- `fn` 可能是 0。只调用有未完成(outstanding)计时器的进程的周期函数。在 `user/alarmtest.arm`，地址 0 指向 `periodic` 时就是 0。
- 调整 `usertrap()`，使得周期到时，用户进程执行周期函数。RISC-V 中断返回到用户空间时，注意地址。

- 可以 `make CPUS=1 qemu-gdb`，查看只用单个 CPU 的 `qemu`。

`test1/test2()`：需要保证执行完毕时，控制器返回到触发的地方，寄存器内容恢复，未被干扰 (undisturbed)。

用户态周期函数处理器(alarm handler)需要调用 `sigreturn` 系统调用，可以阅读 `alarmtest.c` 的 `periodic` 为例子。可以在 `usertrap` 和 `sys_sigreturn` 添加代码，协同让进程恢复现场。

- 需要存储和恢复寄存器现场。大量寄存器都需要参与。
- `usertrap` 存储状态，使用 `struct proc`，然后恢复用上。
- 防止多次执行的调用(re-entrant call)，如果一个调用没返回，不要调用它。在 `test2` 将会体现。

`usertests` 保证新功能不影响其他部分功能，即其他功能照常通过测试。

## 实现

`user/user.h` 添加：

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

`Makefile` 的 `UPROGS=`：

```
1 $U/_alarmtest\
```

对 `kernel\proc.h`，添加 `struct proc` 的成员：

```
1 int alarm_interval; //0无事件，否则事件周期
2 void(*alarm_handler)();
3 int alarm_ticks; //还剩多少ticks后执行下一次
4 struct trapframe *alarm_trapframe;
5 int alarm_goingoff; //是否有在执行的
```

模仿上下文，`proc.c` 的 `allocproc` 函数的 `found:` 的第一个 `if` 后添加：

```
1 if((p->alarm_trapframe = (struct trapframe *)kalloc()) == 0){
2     freeproc(p);
3     release(&p->lock);
4     return 0;
5 }
6 p->alarm_interval = 0;
7 p->alarm_handler = 0;
8 p->alarm_ticks = 0;
9 p->alarm_goingoff = 0;
```

且模仿上下文，`freeproc` 添加：

```
1 if(p->alarm_trapframe)
2     kfree((void*)p->alarm_trapframe);
3 p->alarm_trapframe = 0;
```

`kernel/sysproc.c`，做取参数和转换处理：

```

1 // 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 }

```

kernel/defs.c:

```

1 int sigalarm(int, void(*)());
2 int sigreturn();

```

kernel/syscall.h

```

1 #define SYS_sigalarm 22
2 #define SYS_sigreturn 23

```

kernel/syscall.c:

```

1 extern uint64 sys_sigalarm(void);
2 extern uint64 sys_sigreturn(void);
3 // static uint64 (*syscalls[])(void) = { 内
4 [SYS_sigalarm] sys_sigalarm,
5 [SYS_sigreturn] sys_sigreturn,

```

user/usys.pl:

```

1 entry("sigalarm");
2 entry("sigreturn");

```

kernel/trap.c:

```

1 int sigalarm(int ticks, void(*handler)()) {
2     struct proc *p = myproc();
3     p->alarm_interval = ticks;
4     p->alarm_handler = handler;
5     p->alarm_ticks = ticks;
6     return 0;
7 }
8
9 int sigreturn() {

```

```

10 struct proc *p = myproc();
11 *p->trapframe = *p->alarm_trapframe;
12 p->alarm_goingoff = 0;
13 return 0;
14 }

```

kernel/trap.c 的 usertrap 函数，最后一个 if，修改为：

```

1  if(which_dev == 2) {
2      if(p->alarm_interval != 0) {
3          if(--p->alarm_ticks <= 0) {
4              if(!p->alarm_goingoff) {
5                  p->alarm_ticks = p->alarm_interval;
6                  *p->alarm_trapframe = *p->trapframe;
7                  //PC record program going to run
8                  p->trapframe->epc = (uint64)p->alarm_handler;
9                  p->alarm_goingoff = 1;
10             }
11         }
12     }
13     yield();
14 }

```

- 绝大部分代码显而易见，解释一个不那么显然的代码：

```

1  p->trapframe->epc = (uint64)p->alarm_handler;

```

这个本质是设置 PC 值，实现中断跳转功能。具体参见手册第四章。

测试，通过：

```

1  make qemu
2  alarmtest
3  usertests #需要较久

```

选做任务：调用 backtrace() 时，输出每个函数的名字和行数，而不是单纯的数字地址。

## lab5 copy-on-write fork

[要求](#)，预备：

```

1  git checkout origin2/cow
2  sudo chmod -R 777 /home/starsinhands/xv6-labs-2021

```

### 要求

fork() 系统调用复制所有父进程用户空间内存给子进程。可能是浪费的，因为子不一定要用全部内存。但又不能完全共享，因为会写。

使用 COW(copy-on-write) 机制，延迟复制直到需要写。在该机制下，创建子节点页表时，PTE 里的用户内存直接指向父的，且这样的指针标记为不可写的。父或子需要写时，将产生页错误，被处理器捕获，然后进行复制分配，调整 PTE，标记为可写的，返回现场。

使用 COW 时，释放内存较为麻烦。一个页可能被多个指针指向，只有所有指针不再指向时才能释放内存。

需要通过 `cowtest` 和 `usertests`。在不实现 COW 时，会因为 MLE 无法通过测试。

示例：

```
1  $ cowtest
2  simple: ok
3  simple: ok
4  three: zombie!
5  ok
6  three: zombie!
7  ok
8  three: zombie!
9  ok
10 file: ok
11 ALL COW TESTS PASSED
12 $usertests
13 ...
14 ALL TESTS PASSED
15 $
```

需要：

- 调整 `uvmcopy()`，映射父页给之页，不要分配新页，清空父子的 `PTE_W`。
- 调整 `usertrap()`，识别页错误，用 `kalloc()` 分配和复制，设置 `PTE_W`。
- 最后一个指针释放后，释放掉内存页。对每个页，记录指针计数，每次 `kalloc()` 记录一下，`fork()` 也是。释放时自减。`kfree()` 每次将一个页返回空闲页链表。计数器可以用整形数组维护。想办法索引该数组，选择数组大小。如映射，对页最高地址位映射。可以去 `kinit()` 做。
- 修改 `copyout()`，用类似上面的策略处理 COW。

提示：

- 不要基于懒分配策略，用 xv6 的 copy 来基于。
- 对每个 PTE 记录是否进行了 COW 映射，如使用 RSW 位(为软件保留的位 reserved for software)。
- 记得测 `usertests`
- `kernel/riscv.h` 看有帮助的宏和定义
- 如果 COW 时内存不够，进程结束

扩展：

- 不仅支持懒页分配，还支持 COW
- 统计 COW 的优化程度，统计使用的内存。找到更优的优化

## 实现

### 参考1 相似参考

根据手册 33 页, 可知给了 3 个预留位。可以用一个位表示当前页是否 COW(指向父)。

在 `riscv.h` 添加定义:

```
1 #define PTE_C (1L << 8) // copy on write
```

在 `memlayout.h` 里, 有常量 `PHYSTOP` 代表最大物理地址。在 `riscv.h` 有 `PGSIZE` 代表页大小, 相除得总页数为 557056, 需要约 2MB 的大小开 int 数组。所以可以开一个全局数组记录每个页被多少个引用。考虑到高并发, 再开一个锁, 在 `kalloc.c` 里:

```
1 int refcount[PHYSTOP/PGSIZE];
2 struct spinlock reflock;
```

同文件实现自增基本逻辑:

```
1 void incref(uint64 pa) //physical address
2 {
3     int pn = pa/PGSIZE; //physical number
4     acquire(&kmem.lock);
5     if(pa>=PHYSTOP || refcount[pn]<1){
6         panic("incref");
7     }
8     refcount[pn] += 1;
9     release(&kmem.lock);
10 }
```

或者适配性更强设置成通用函数, 这里懒得这么搞, 因为 if 不一样:

```
1 typedef void (*ref_f)(int);
2 void ref_inc(int pn) {refcount[pn] += 1;}
3 void ref_cls(int pn) {refcount[pn] = 0;}
4 void ref_dec(int pn) {refcount[pn] -= 1;}
5
6 void
7 ref_modify(uint64 pa, ref_f f)
8 { //pa physical address
9     int pn = pa/PGSIZE; //physical number
10    acquire(&kmem.lock);
11    if(pa>=PHYSTOP || refcount[pn]<1){ //conditions waits changes
12        panic("ref_modify");
13    }
14    f(pn);
15    release(&kmem.lock);
16 }
```

类似地, 对 `kalloc` 函数, 修改 `if(r)` 条件语句为:

```

1  if(r){
2      kmem.freelist = r->next;
3      int pn = (uint64)r / PGSIZE;
4      acquire(&reflock);
5      if(refcount[pn]!=0){
6          panic("kalloc ref");
7      }
8      refcount[pn] = 1;
9      release(&reflock);
10 }

```

对 `kfree` 函数, 第一个 if 后添加:

```

1  acquire(&kmem.lock);
2  int pn = (uint64) pa / PGSIZE;
3  if(refcount[pn]<1){
4      panic("kfree ref");
5  }
6  refcount[pn]-=1;
7  int tmp = refcount[pn];
8  release(&kmem.lock);
9  if(tmp>0){/*other proc using, cannot release
10      return;
11  }

```

对 `kinit` 函数, 阅读可知它 `freerange` 了一段页面, 因为要释放, 所以在这之前要增加, 模仿 `freerange` 函数, 在初始锁后添加:

```

1  char *p;
2  p = (char*)PGROUNDUP((uint64)end);
3  for(; p + PGSIZE <= (char*)PHYSTOP; p += PGSIZE){
4      refcount[(uint64)p/ PGSIZE] = 1;
5  }

```

对 `vm.c` 的 `uvmcopy` 函数, for 内两个 if panic 后面内容改为:

```

1  pa = PTE2PA(*pte);
2  *pte &= ~PTE_W; /* ban write bit
3  *pte |= PTE_C; /* get cow bit
4  flags = PTE_FLAGS(*pte);
5  incref(pa); /* add visit count
6  //let va i = father pa rather than memmove copy
7  if(mappages(new, i, PGSIZE, pa, flags) != 0){
8      goto err;
9  }
10 //且删掉这个变量:
11 char *mem;

```

在 `trap.c`, 定义函数如下:

```

1  int
2  cowfault(pagetable_t pagetable, uint64 va)

```

```

3 {
4     if(va >= MAXVA){
5         return -1;
6     }
7     pte_t *pte;
8     pte = walk(pagetable,va,0); //find pte from va
9     if(pte == 0) return -1;
10    // not valid or not accessible or not a cow page
11    if ((*pte & PTE_U) == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_C) == 0){
12        return -1;
13    }
14
15    uint64 pa1,pa2;
16    pa1 = PTE2PA(*pte);
17    pa2 = (uint64)kalloc(); //new pa, vis += 1
18    if(pa2 == 0){
19        return -1;
20    }
21    memmove((char*)pa2,(char*)pa1,PGSIZE); //pa2=pa1
22    kfree((void*)pa1); //vis of pa1 -= 1, if vis ==0 free otherwise ignore
23    uint flags = PTE_FLAGS(*pte);
24    //pte = pa + flags (add writable)
25    *pte = PA2PTE(pa2) | flags | PTE_W;
26    *pte &= ~PTE_C; // ban cow
27    return 0;
28 }

```

同文件 `useretrace` 函数, else 前增加:

```

1 else if(r_scause()==15 || r_scause()==13){ //page fault
2     if(cowfault(p->pagetable,r_stval())<0){
3         p->killed = 1;
4     }
5 }

```

- `scause` 描述了中断原因, `r_scause()` 在 `riscv.h` 定义, 使用汇编语言代码读取中断原因, 取 `scause` 寄存器。
- `r_stval()` 是 `riscv.h` 定义的, 取 `va`。
- 标记 `killed`, 终止进程。

在 `vm.c` 的 `copyout` 函数的 `if(pa0==0)` 语句下方加入:

```

1 pte_t *pte = walk(pagetable,va0,0);
2 if(pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0){
3     return -1;
4 }
5 // write protect caused by COW
6 if((*pte & PTE_W) == 0 && (*pte & PTE_C) == 1){
7     if(cowfault(pagetable,va0)<0){
8         return -1;
9     }
10 }
11 pa0 = PTE2PA(*pte); //update pa

```

kernel/defs.h 添加:

```
1 pte_t * walk(pagetable_t, uint64, int); // vm.c
2 void incref(uint64); // kalloc.c
3 int cowfault(pagetable_t, uint64); // trap.c
```

编译, 运行, 测试, 成功, 提交:

```
1 git push origin HEAD:refs/heads/cow
```

## lab6 multithreading

[要求](#), 预备:

```
1 git checkout origin2/thread
2 sudo chmod -R 777 /home/starsinhands/xv6-labs-2021
```

### Uthread: switching between threads

#### 要求

使用多线程优化用户级线程包。设计上下文切换机制, 实现用户级多线程系统。有 `user/uthread.c` 和 `user/uthread_switch.S` 和对应的 `Makefile`。其中 `uthread.c` 包含大多数的包, 有三个简单测试线程, 缺失切换线程的代码。需要实现保存恢复寄存器, 切换进程。可以用 `make grade` 测试评分, 也可以跑 `uthread`。三个线程都从 0 输出到 100, 然后每个数字全输出完了才能输出下一个数字。

需要对 `uthread.c` 的 `thread_create()`, `thread_schedule()` 添加代码, 对 `user/uthread_switch.S` 的 `thread_switch` 添加代码。确保线程调度函数对给定线程, 传递给进程创建, 使用自己的栈。确保保存了寄存器, 且返回到现场指令位置正确。修改 `struct thread` 来保存寄存器。在 `thread_schedule` 添加对 `thread_switch` 的调用。

提示:

- `thread_switch` 只需要保存 `callee-save` 寄存器。

为了确保调用函数前后这些寄存器的值能够保持一致, 一些特定的寄存器被指定为 `Callee-save` 寄存器

可以在 `user/uthread.asm` 查看汇编代码方便 debug

- 可以用 `riscv64-linux-gnu-gdb` 来逐步运行 `thread_switch`, 可设置断点等。

可选任务: 目前的功能有多个不足之处, 如, 若一个用户级线程阻断在了一个系统调用, 另一个线程就不会跑, 因为调度器不知道上一个线程什么时候结束调用。且多个线程不会在多核并发执行, 因为 `xv6` 调度器不会意识到多线程在运行。而如果真的并发, 可能会出错, 如两个处理器同时调用了调度器, 然后选择了同一个运行线程, 使得一个代码在两个处理器上跑。

有若干解决办法, 一种是[调度器活动](#), 一种是使用核线程与用户线程一一对应。可能需要实现 TLB 等。

为 `thread` 包添加锁、条件变量、屏障等。

## 实现

此线程更接近协程(coroutine)，因为完全用户态，多个线程运行在一个 CPU，没有时钟中断强制调度，需要主动 yield，不对线程函数透明。

参考 `kernel/switch.s`，复制 `switch:` 下面的内容，对 `uhtread_switch.S` 在 `thread_switch:` 后顶格：

```
1  sd ra, 0(a0)
2  sd sp, 8(a0)
3  sd s0, 16(a0)
4  sd s1, 24(a0)
5  sd s2, 32(a0)
6  sd s3, 40(a0)
7  sd s4, 48(a0)
8  sd s5, 56(a0)
9  sd s6, 64(a0)
10 sd s7, 72(a0)
11 sd s8, 80(a0)
12 sd s9, 88(a0)
13 sd s10, 96(a0)
14 sd s11, 104(a0)
15
16 ld ra, 0(a1)
17 ld sp, 8(a1)
18 ld s0, 16(a1)
19 ld s1, 24(a1)
20 ld s2, 32(a1)
21 ld s3, 40(a1)
22 ld s4, 48(a1)
23 ld s5, 56(a1)
24 ld s6, 64(a1)
25 ld s7, 72(a1)
26 ld s8, 80(a1)
27 ld s9, 88(a1)
28 ld s10, 96(a1)
29 ld s11, 104(a1)
30
31 ret
```

- 以 `sd sp 8(a0)` 为例，将 `sp` 存储到地址 `a0+8` 上。`sd` 为 Store Double Word。
- 同理，`ld` 读取，Load Double Word。
- 这里的所有内容即 `kernel/proc.h` 的 `struct context` 的内容。

上下文的切换永远发生在函数调用的边界，这就是只需要保存 callee-saved 寄存器的原因。因为无论是时钟中断(user trap)还是 `sleep`, `exit`，都会通过 `yield` 调用 `switch`。所以只需要返回地址 `ra`，栈指针 `sp`，和全体 callee-saved。

对比之下，`trap frame` 中断可以发生在任何地方，需要靠 `pc` 定位，所有寄存器(caller, callee-saved)都要保存，所以 `struct trapframe` 比 `struct context` 多。

程序 sleep 或时钟中断，都通过 `trampoline` 跳转到内核态 `usertrap` 保存 `trapframe`，然后 `swtch` 保存上下文，回复上下文是恢复到 `swtch` 返回前，还在内核态，然后跳回 `usertrap`，继续执行直到 `usertrapret` 跳转返回用户态。即上下文恢复先恢复到内核态刚执行完 `swtch`，然后再恢复。

对 `uthread.c`，复制一个 `proc.h` 的 `struct context`。在 `struct thread` 前添加：

```
1 struct context {
2     uint64 ra;
3     uint64 sp;
4     uint64 s0;
5     uint64 s1;
6     uint64 s2;
7     uint64 s3;
8     uint64 s4;
9     uint64 s5;
10    uint64 s6;
11    uint64 s7;
12    uint64 s8;
13    uint64 s9;
14    uint64 s10;
15    uint64 s11;
16 };
```

对 `struct thread`，添加该结构体成员变量：

```
1 struct context ctx;
```

修改 `thread_switch` 函数的定义，将 `uint64` 虽然也是指针，但是直接用指针更方便：

```
1 extern void thread_switch(struct context* old, struct context* new);
```

其中，`extern` 告诉编译器这个函数有声明没定义，是在其他文件中定义的，编译器在链接阶段会查找并连接正确的函数定义。如：

```
1 gcc -c functions.c -o functions.o
2 gcc -c main.c -o main.o
3 gcc main.o functions.o -o my_program
4 # my_program 是 exe 名字
5 ./my_program
```

或集成 `makefile`，用于自动化构建过程，特别是在编译和链接多个源代码文件时非常有用。如：

```
1 CC = gcc
2 CFLAGS = -Wall
3
4 all: my_program
5
6 my_program: main.o functions.o
7     $(CC) $(CFLAGS) main.o functions.o -o my_program
8
9 main.o: main.c
10    $(CC) $(CFLAGS) -c main.c
11
```

```

12 functions.o: functions.c
13 $(CC) $(CFLAGS) -c functions.c
14
15 clean:
16 rm -f *.o my_program

```

执行:

```

1 make
2 make clean #清理生成的目标文件和可执行文件

```

对 `uthread.c` 的 `thread_schedule` 函数, 最后一个 if 追加一行代码:

```

1 thread_switch(&t->ctx, &next_thread->ctx);

```

代码已经写好了找要切换的任意一个下一个线程, 已经找好了, 直接切即可。代码里只有最多 4 个线程支持。

对 `thread_create` 函数, 在 your code here 往下:

```

1 t->ctx.ra = (uint64)func;
2 //stack pointer from high to low grow, so highest as start
3 t->ctx.sp = (uint64)&t->stack + (STACK_SIZE - 1);

```

然后执行 make 测试即可, 执行正确。

```

1 make qemu
2 uthread

```

```

1 git push origin HEAD:refs/heads/thread

```

## Using threads

### 要求

并发编程, 使用线程和基于哈希表的锁, 需要在 linux 真机实现, 必须拥有多核。使用 `UNIX` 的 `pthread` 多线程库, 通过 `man pthreads` 查库, 或者资料: [参考1](#), [参考2](#), [参考3](#)

文件 `notxv6/ph.c` 包含了非并发正确运行的哈希表。执行:

```

1 make ph
2 ./ph 1

```

需要使用操作系统的 GCC, 参数 1 表示进程数。ph 的基本逻辑是通过 `put()` 插入大量键, 然后用 `get()` 取大量的键。如果要多线程, 就 `./ph 2`, 如果有 `xxx keys missing` 表明多线程失败。

找出多线程失败的原因, 对 `put, get` 使用锁, 保证最后 `0 keys missing`, 锁相关代码参考:

```

1 pthread_mutex_t lock;           // declare a lock
2 pthread_mutex_init(&lock, NULL); // initialize the lock
3 pthread_mutex_lock(&lock);      // acquire lock
4 pthread_mutex_unlock(&lock);    // release lock

```

记得调用 `pthread_mutex_init`。观察是否多线程能提速，即单位时间操作执行的数量。有的时候可以取消锁，来进一步提速，比如锁的级别精确到每个桶而不是整个哈希表，更细粒度。

实现后可以通过 `make grade` 的 `ph_safe` 和 `ph_fast`，要求双线程比单线程快至少 1.25 倍。

## 实现

如果定义全局变量 `pthread_mutex_t lock`；且 main 头 `pthread_mutex_init(&lock, NULL);`，`put` 和 `get` 的首尾添加 `pthread_mutex_lock(&lock);` 和 `pthread_mutex_unlock(&lock);`；

则，虽然能通过 `ph_save` 测试，但是 `./ph 1` 比 `./ph 2`，失去了多线程提速的初衷。理由是锁粒度太大，实际上只能有一个同时的 `get` 或 `put` 在执行，本质是单线程，且还要维护锁。所以必须降低锁的粒度。定义全局锁：

```

1 pthread_mutex_t locks[NBUCKET];

```

在 `main` 头初始锁：

```

1 for(int i=0;i<NBUCKET;i++) {
2     pthread_mutex_init(&locks[i], NULL);
3 }

```

在 `get, put` 函数头尾添加：

```

1 pthread_mutex_lock(&locks[i]);
2 pthread_mutex_unlock(&locks[i]);

```

可以看到本质是相当简单的作业任务。

本地测试，测试通过：

```

1 make ph
2 ./ph 1
3 ./ph 2 #快了大约一倍,0 missing
4 make grade #结果表明目前做到的进度都对了

```

## Barrier

### 要求

barrier 类似集合点，即所有线程都要到达这个点后，才能继续往下走。

需要在真机(而不是 xv6, qemu)实现，对 `notexv6/barrier.c`，执行：

```

1 make barrier
2 ./barrier 2 #2是并发数

```

每个进程执行 `thread()` 函数，调用 `barrier()` 方法。预期逻辑：如果没有所有进程都到达 `barrier()` 就阻塞，直到所有都到达了才一起执行。除了 `ph` 上述任务用到的多线程函数外，还可能需用到下述函数，参考 [这里](#) 和 [参考](#)：

```
1 pthread_cond_wait(&cond, &mutex); // go to sleep on cond, releasing lock
   mutex, acquiring upon wake up
2 pthread_cond_broadcast(&cond);    // wake up every thread sleeping on cond
```

调用 `wait` 时释放 `mutex`，返回时需求 `mutex` 资源。在代码已经进行过初始化了，无需再次初始化。可以使用 `struct barrier` 结构体的唯一成员变量。

可能需要注意：

- 处理一系列 `barrier` 调用，每次称为一轮。`bstate.round` 记录了当前轮数。
- 必须处理在其他线程退出屏障时，还有一个线程在循环中的办法。这样的情况不能改变 `bstate.nthread`。

使用 1,2 和更多的并发数测试代码。

## 实现

经典的同步屏障机制问题。

阅读代码可知，提供了 `bstate` 的成员：

```
1 int nthread; // Number of threads that have reached this round of the barrier
```

即这个是当 `cnt` 来用的，统计这一轮调用了多少次这个 `barrier`。有全局变量 `nthread` 表示总进程数。所以，如果当前不是最后一个，就等待；如果是最后一个，更新轮数，就解锁全部。具体而言，因为提供了成员：

```
1 pthread_cond_t barrier_cond;
2 pthread_mutex_t barrier_mutex;
```

所以非最后一个就监听 `barrier_cond`，在监听的过程中，释放掉锁，让其他进程允许进入修改 `nthread`。然后等待结束后，把锁要回来，再结束。最后一个就 `notifyAll`。

对 `barrier()` 函数，实现如下：

```
1 pthread_mutex_lock(&bstate.barrier_mutex);
2 if(++bstate.nthread < nthread) {
3     pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
4 } else {
5     bstate.nthread = 0;
6     bstate.round++;
7     pthread_cond_broadcast(&bstate.barrier_cond);
8 }
9 pthread_mutex_unlock(&bstate.barrier_mutex);
```

测试，通过：

```
1 make barrier
2 ./barrier 1
3 ./barrier 2
4 ./barrier 4
```

## lab7 networking

[要求](#)，预备：

```
1 git checkout origin2/net
2 chmod -R 777 /home/starsinhands/xv6-labs-2021
```

### 要求

可以查看手册第五章。将使用名为 `E1000` 的设备处理网络通信，对 `xv6` 和驱动器来说，该设备看起来像真实的硬件连接到以太局域网。而实际都是模拟的，其中 `xv6` 作为 `guest` 的 IP 地址是 `10.0.2.15`，运行 `qemu` 的 LAN IP 为 `10.0.2.2`，`xv6` 使用 `E1000` 发包给该 IP，`qemu` 收包给合适的应用(host)。

使用 `qemu` 的用户模式网络栈，[文档参考](#)。makefile 记录所有包到文件 `packets.pcap`，可以核验收包、发包。显示这些包：

```
1 tcpdump -XXnr packets.pcap
```

文件 `kernel/e1000.c` 包含对设备初始化和空的收发函数。`kernel/e1000_dev.h` 包含寄存器、位的定义。可以参考 `e1000` 的[手册](#)。而 `kernel/net.c` 和 `kernel/net.h` 包含简单网络栈实现了 IP, UDP, ARP 协议。包含存储包的数据结构称为 `mbuf`。且 `kernel/pci.c` 包含了启动 `xv6` 时在 PCI 总线搜索 `E1000` 的代码。

需要实现 `kernel/e1000.c` 的 `e1000_transmit()` 和 `e1000_recv()` 函数。通过 `make grade` 进行测试。

手册概要：

- Section 2 概述整个设备
- Section 3.2 概述包接收
- Section 3.3/3.4 概述包发送
- Section 13 概述使用的寄存器
- Section 14 可以帮助理解初始函数

初始化 `e1000_init()` 设置其从 RAM 读写，使用 DMA 技术，使得 `E1000` 直接从 RAM 读写包。

收到包的速度可能快于处理的速度，所以提供了多个缓存区，使用 RAM 的描述符数组描述。`struct rx_desc` 定义了描述符格式。描述符形成接收环/队列。分配了 `mbuf` 个包缓存给 `E1000` 去 DMA，使用 `mbufalloc()`。也有接收环。环的大小是常量 `RX_RING_SIZE` 和 `TX_RING_SIZE`。

在 `net.c` 的网络栈需要发包时，就会调用 `e1000_transmit()`，传入一个 `struct mbuf`(`net.h` 定义)，包含要发的包。需要设一个指针指向包数据，使用 TX 环描述符。需要保证每个 `mbuf` 在完成发送后最终都被释放，其中 `E1000_TXD_STAT_DD` 位能表示是否完成传输。

测试驱动器，运行 `make server`，然后在另一个窗口运行 `make qemu` 且执行 `nettests`。第一个测试发送一个 UDP 包给 host，然后收到一个回复包。在发送之前，会先来一个 ARP 包找到以太网地址，收到 ARP 应答。在 `kernel/net.c` 会体现。预期输出：`testing ping: OK`，且 server 输出 `a message from xv6!`。输出包预期：

```
1 tcpdump -xxnr packets.pcap
```

```
1  reading from file packets.pcap, link-type EN10MB (Ethernet)
2  15:27:40.861988 IP 10.0.2.15.2000 > 10.0.2.2.25603: UDP, length 19
3      0x0000:  ffff ffff ffff 5254 0012 3456 0800 4500  ....RT..4V..E.
4      0x0010:  002f 0000 0000 6411 3eae 0a00 020f 0a00  ./....d.>.....
5      0x0020:  0202 07d0 6403 001b 0000 6120 6d65 7373  ....d.....a.mess
6      0x0030:  6167 6520 6672 6f6d 2078 7636 21          age.from.xv6!
7  15:27:40.862370 ARP, Request who-has 10.0.2.15 tell 10.0.2.2, length 28
8      0x0000:  ffff ffff ffff 5255 0a00 0202 0806 0001  ....RU.....
9      0x0010:  0800 0604 0001 5255 0a00 0202 0a00 0202  ....RU.....
10     0x0020:  0000 0000 0000 0a00 020f          .....
11  15:27:40.862844 ARP, Reply 10.0.2.15 is-at 52:54:00:12:34:56, length 28
12     0x0000:  ffff ffff ffff 5254 0012 3456 0806 0001  ....RT..4V....
13     0x0010:  0800 0604 0002 5254 0012 3456 0a00 020f  ....RT..4V....
14     0x0020:  5255 0a00 0202 0a00 0202          RU.....
15  15:27:40.863036 IP 10.0.2.2.25603 > 10.0.2.15.2000: UDP, length 17
16     0x0000:  5254 0012 3456 5255 0a00 0202 0800 4500  RT..4VRU.....E.
17     0x0010:  002d 0000 0000 4011 62b0 0a00 0202 0a00  .-....@.b.....
18     0x0020:  020f 6403 07d0 0019 3406 7468 6973 2069  ..d....4.this.i
19     0x0030:  7320 7468 6520 686f 7374 21          s.the.host!
```

还有一些其他测试，最后还有 DNS 请求寻求真实网站的 IP 地址。预期如：

```
1  $ nettests
2  nettests running on port 25603
3  testing ping: OK
4  testing single-process pings: OK
5  testing multi-process pings: OK
6  testing DNS
7  DNS arecord for pdos.csail.mit.edu. is 128.52.129.126
8  DNS OK
9  all tests passed.
```

最终进行测试 `make grade`。

提示：

可以在 `e1000_transmit()`，`e1000_recv()` 家输出语句，运行 `make server` 和 `nettests`，可以看到对应的输出。

对 `e1000_transmit` 的提示：

- 求 TX 环索引，找到下一个包，读取 `E1000_TDT` 控制寄存器
- 检查是否该 ring 溢出。若 `E1000_TXD_STAT_DD` 没设置 `E1000_TDT`，则还没完成上一次传输，直接报错和返回即可。
- 否则，使用 `mbuffree()` 释放上一个 `mbuf`，对上一年的描述符。

- 填充描述符 `m->head` 指针，写入包内容，标注包长度 `m->len`。设置必要的 `cmd` 位 `flags`，查看 section 3.3 参考。存一个指针方便后续清理它。
- 更新 `ring` 位置，增加 `E1000_TDT` (模意义下)。
- 若把 `mbuf` 成功加到了环，返回 0。否则返回 -1，如无可用描述符。

对 `e1000_recv` 的提示：

- 通过 `E1000_RDT` 读位置，然后模意义加一。
- 看看是否有新包可用，查 `E1000_RXD_STAT_DD` 位，在 `status` 里。如果查不到，返回。
- 否则，更新 `m->len`，分配 `mbuf` 给网络栈使用 `net_rx()`。
- 用 `mbufalloc()` 分配新 `mbuf`，替代刚刚给 `net_rx()` 的，将指针存到描述符，清除描述符状态。
- 更新 `E1000_RDT` 寄存器为最后处理的处理器。
- `e1000_init()` 初始化了 RX 环，可以借鉴代码。
- 包数会超过环长，需要正确处理。

可能需要锁，因为核线程会多并发。

可选任务：有些任务只能在高性能硬件如 x86 测试。

- 网络栈使用中断处理包进入，但包出去不是。更复杂的策略会对包出去排队列，每次只给固定数目包去网卡 NIC Network Interface Card。

## 实现

为了防止高并发出错，直接给收发函数加锁，保证每次只能有一个线程进行收或发，也不准同时收发。

TX 和 RX 是 transmit 和 receive 的缩写。`E1000_TDT` 是发送描述符尾，先用 `regs[E1000_TDT]` (记为 `i`) 找到它代表的描述符的下标，然后找描述符，在 `tx_ring[i]` 搞该下标。

描述符有 `status`，查阅手册 39 页 可以知道状态的含义。可知，如果有 `E1000_TXD_STAT_DD` 位，就证明执行完毕传送了，此时放锁返回即可。

如果这个下标的缓存区 `tx_mbufs[i]` 不是空，把它释放，并设为空指针。

然后修改描述符，将传入缓存区 `m` 的头部 (`char*` 转 `u11` 变成地址) 赋值给 `addr` 成员，然后赋值长度。然后给该描述符两个命令，一个是指示该缓存区有完整包，一个是告诉网卡发送完毕后，设置 `status` `DD` 位。然后将当前缓存区设为传入的 `m` (日后清理用，如上)。

最后，将 `i` 值模意义加一。

```

1  acquire(&e1000_lock);
2  uint32 i = regs[E1000_TDT];
3  struct tx_desc *desc = &tx_ring[i];
4  if(!(desc->status & E1000_TXD_STAT_DD)){
5      release(&e1000_lock);
6      return -1;
7  }
8  if(tx_mbufs[i]){
9      mbuffree(tx_mbufs[i]);
10     tx_mbufs[i]=0;
11 }
12 desc->addr = (uint64)m->head;
13 desc->length = m->len;
```

```

14 desc->cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;
15 tx_mbufs[i] = m;
16
17 regs[E1000_TDT] += 1; regs[E1000_TDT] %= TX_RING_SIZE;
18 release(&e1000_lock);

```

发送一次发一个，但是接收可能收多个。同理，先取下标(注意取当前下标的下一个下标)，然后下标自增。看看状态 DD 位。更新 len，然后调用 `net_rx` 把包丢给上一层网络协议。对当前下标，开一个新的缓存区给缓存区指针数组，更新当前描述符的地址为刚分配的玩意的 head 指针，设状态为空。

```

1  for(;;){
2      uint32 i = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
3      struct rx_desc *desc = &rx_ring[i];
4      if(!(desc->status && E1000_RXD_STAT_DD)) {
5          return;
6      }
7      rx_mbufs[i]->len = desc->length;
8      net_rx(rx_mbufs[i]);
9      rx_mbufs[i] = mbufalloc(0);
10     desc->addr = (uint64)rx_mbufs[i]->head;
11     desc->status = 0;
12     regs[E1000_RDT] = i;
13 }

```

过不了 make grade 的话，把 `time.txt` 新建一下，里边只写一个整数，可以代表完成作业的小时。

按照上述方法测试，可以通过测试。

```

1 | git push origin HEAD:refs/heads/net

```

## lab8 locks

[要求](#)，预备：

```

1 | git checkout origin2/lock
2 | chmod -R 777 /home/starsinhands/xv6-labs-2021

```

目的：高并发下减少锁争用(contention)。参考手册的第 6, 3.5, 8.1-8.3 章。

## Memory allocator

**要求**

`user/kalloc` 压测内存分配器，三个进程不断调整地址空间，调用大量的 `kalloc` 和 `kfree`，它们请求 `kmem.lock`。输出锁被占用的次数。其中 `acquire` 是请求锁的次数和设置锁失败的次数。

在多核下设计解锁机制。锁争用的根源是 `kalloc()` 有单个 free 列表，被三个锁保护。需要重新设计内存分配器，来避免单个锁盒列表。如对每个 CPU 做空闲列表，每个列表做自己的锁。如果一个 CPU 没空闲，但另一个有，一个 CPU 需要偷用另一个 CPU 的部分空闲列表，导致锁竞争，但一般不频繁。

需要实现每个 CPU 的空闲列表，当 CPU 空闲列表空时，给定以 `kmem` 开头的锁的名字。可以运行 `usertests sbrkmuch` 测试。可以最终运行 `make grade` 测试。举起输出的数据显著降低，且 `tot=0`。

```
1 kalloc_test
2 user_tests sbrkmuch
3 user_tests
```

提示:

- 使用 `kernel/param.h` 的 `NCPU` 常量
- `freerange` 给所有空闲内存给 CPU 运行。
- `cpuid` 函数返回当前的核编号, 但仅当调用和使用结果在中断返回时。使用 `push_off()` 和 `pop_off()` 来关中断、开中断。
- `snprintf` 函数在 `kernel/sprintf`, 了解字符串格式, 或者单纯任意命名也行。

## 实现

先进行预测试, 得到结果:

```
1 start test1
2 test1 results:
3 --- lock kmem/bcache stats
4 lock: kmem: #test-and-set 429181 #acquire() 433016
5 lock: bcache: #test-and-set 0 #acquire() 1248
6 --- top 5 contended locks:
7 lock: proc: #test-and-set 556846 #acquire() 1003794
8 lock: kmem: #test-and-set 429181 #acquire() 433016
9 lock: proc: #test-and-set 271559 #acquire() 1003792
10 lock: proc: #test-and-set 266702 #acquire() 1003792
11 lock: proc: #test-and-set 259899 #acquire() 1003793
12 tot= 429181
13 test1 FAIL
```

分析可知, 瓶颈为 `kmem` 锁(第四行), 导致了全部的 `tot` 贡献。如果一个大锁并不会引起明显的性能问题, 有时候大锁就足够了。只有在万分确定性能热点是在该锁的时候才进行优化。由此确认了问题在 `kmem` 锁。

性能问题的思路: 将共享资源变为不共享资源。锁竞争优化的思路:

- 只在必须共享时共享, 将资源从 CPU 共享拆分为每个 CPU 独立。
- 必须共享时, 尽量减少在关键区中停留的时间, 降低锁的粒度。

看常量看到有 8 个核, 所以在 `kernel/kalloc.c` 开八个锁, 锁名如下:

```
1 char *kmem_lock_names[] = {
2     "kmem_cpu_0",
3     "kmem_cpu_1",
4     "kmem_cpu_2",
5     "kmem_cpu_3",
6     "kmem_cpu_4",
7     "kmem_cpu_5",
8     "kmem_cpu_6",
9     "kmem_cpu_7",
10 };
```

将原有的 `kmem` 改成数组 `kmem[NCPU]`。

一个 freelist 可以被偷，所以可以被多个 CPU 访问。所以每个 freelist 需要上锁。而单个 CPU freelist 空闲页不足情况比原本更稀有，所以比大锁更快。在最佳情况下，也就是没有发生跨 CPU “偷”页的情况下，这些小锁不会发生任何锁竞争。

既然变了 `kmem`，那么 `kinit()` 第一行改为：

```
1  for(int i=0;i<NCPU;i++) {
2      initlock(&kmem[i].lock, kmem_lock_names[i]);
3  }
```

对 `kfree()`，`r=` 后面改为：

```
1  push_off();
2  int cpu = cpuid();
3
4  acquire(&kmem[cpu].lock);
5  r->next = kmem[cpu].freelist;
6  kmem[cpu].freelist = r;
7  release(&kmem[cpu].lock);
8
9  pop_off();
```

即关中断，取 CPU，取锁，做跟之前旧代码一样的链表插入，开中断。

对 `kalloc()` 比较麻烦，对 `if(r)`，改为：

```
1  push_off();
2
3  int cpu = cpuid();
4
5  acquire(&kmem[cpu].lock);
6
7  if(!kmem[cpu].freelist) { // no page left for this cpu
8      int steal_left = 64; // steal 64 pages from other cpu(s)
9      for(int i=0;i<NCPU;i++) {
10         if(i == cpu) continue; // no self-robbery
11         acquire(&kmem[i].lock);
12         struct run *rr = kmem[i].freelist;
13         while(rr && steal_left) {
14             kmem[i].freelist = rr->next;
15             rr->next = kmem[cpu].freelist;
16             kmem[cpu].freelist = rr;
17             rr = kmem[i].freelist;
18             steal_left--;
19         }
20         release(&kmem[i].lock);
21         if(steal_left == 0) break; // done stealing
22     }
23 }
24
25 r = kmem[cpu].freelist;
26 if(r)
27     kmem[cpu].freelist = r->next;
28 release(&kmem[cpu].lock);
```

具体而言，抛开 `if(!)` 不谈的话，就是很普通的链表删除加锁罢了。考虑 `if(!)`，则代表全体内存里拆分给当前 CPU 的空闲内存不够了，此时，尝试找到另一个 CPU 的空闲内存，偷取最多 64 页内存给当前 CPU。一个链表删，另一个链表增，还是比较简单的。64 是任意选取的，可以认为是经验值，目的是减少偷的次数。

这样的解决方案是不完美的，可能会产生死锁，当 A 想偷 B，B 想偷 A 时，即 A 拿了 A 自己的锁，然后找到 `i=B` 时时间片到，然后 B 拿了 B 自己的锁然后 `i=A`，拿不到 A 的锁，A 同理。

一种解决思路是，给每个 CPU 再加一个偷锁，当他要偷别人时，他自己先拿自己的偷锁，然后放掉自己的锁，当他偷完后，拿自己的锁然后放自己的偷锁。

考虑回上面的情况，A 找到 `i=B` 时时间片到，则 A 没自己的锁，有自己的偷锁。B `i=A`，能拿到 A 的锁，然后 B 结束，A 结束。但是可能存在问题，A 刚偷完 64 个页，又被 B 给偷回去了。代价就是，可能这种偷回去会让 A 此次最终没内存可用，并提前引发了内存不足错误。

## Buffer cache

### 要求

多个进程密集使用文件系统时，将会争夺 `bcache.lock` 锁，该锁保护磁盘高速缓存块，在 `kernel/bio.c`。其中，`bcachetest` 创建多个进程不断读不同文件。输出跟上一个 lab 任务类似，`tot` 要降到 0，降低 `#fetch-and-add` 和 `#acquire`。`bcache.lock` 保护高速缓存块的列表，`b->refcnt` 和 `b->dev`，`b->blockno`。

修改高速缓存块，使得 `acquire` 迭代尽可能接近 0，最好加起来不超过 500。修改 `bget` 和 `brelease`，使得并发查找和释放不同块冲突减少。维护最多一个块副本被缓存。最后做 `make grade`。

将全部锁名字以 `bcache` 开头，调用 `initlock`。

该任务将会比上一个难，因为 `bcache` 缓存对每个进程和 CPU 共享。而对上一个任务，每个 CPU 给自己的分配列表，所以一个块不会给多个 CPU。可以使用哈希表查找高速缓存的块号(block numbers)，每个桶设锁。

下列情形，允许出现锁冲突：

- 两进程同时使用同一个块号(block number)。
- 两进程同时不命中缓存，需要找到未使用的块来代替。
- 两进程使用块，这些块如论怎么分块和配锁都会冲突。例如块号哈希进相同的桶。但最好尽可能避免。

一些提示：

- 阅读手册 8.1-8.3，查看高速缓存块的描述。
- 可以使用固定的桶，不给哈希表动态配大小，使用质数数目的桶。
- 哈希表查询和插入必须是原子的。
- 删除所有缓存(如 `bcache.head`)和使用时间戳缓存(如 `ticks`)根据上一次使用。`brelease` 将不必要获取锁，`bget` 可以选择最近使用的块使用时间戳。
- 在 `bget` 可以序列化换出(eviction)，即选择一个缓存重利用，当查找不命中时。
- 方案需要持有两个锁，在一些情况如置换时，需要持有 `bcache` 锁和每个桶的锁，记得避免死锁。

- 替换块时，可以移动一个 `struct buf` 到另外的桶，但是这个桶的哈希值是指向原本的。
- debug 技巧：可以留下全局的 `bcache.lock`，当确保自己做好了再删掉。或者 `make CPUS=1 qemu` 来测试单核正确性。

可选：实现不需要锁的查找缓冲块，使用 GCC 的 `__sync_*` 函数，并证明自己的实现是正确的。

## 实现

`bcache` 块缓存被多个进程，即多个 CPU 共享，即多个进程可以同时访问同一个区块，所以不能像之前一样给每个 CPU 分割专属页。

必须共享，策略为尽量减少在关键区中停留的时间，降低粒度。

阅读 `kernel/bio.c` 代码的 `struct bcache` 注释可知，使用双向链表存储全部区块缓存，阅读 `bget` 可知，将遍历整个链表，找到 `kernel/buf.h` 定义的缓存块 `struct buf` 的设备、编号跟要求的是一样时就返回，且该缓冲块引用数增加。如果不命中，找一个空的(没引用的)块，赋给它。而 `brelse` 就当引用数减到无时普通的双向链表删除。

改进方案：建 `blockno` 到 `buf` 的哈希表，每个桶加锁。许多可能测试测不出的死锁可能会发生。

考虑一种设计，建立一个哈希表 `bufmap`，对原缓存结构修改为：

```
1 struct {
2     struct buf buf[NBUF];
3     struct spinlock eviction_lock;
4
5     // Hash map: dev and blockno to buf
6     struct buf bufmap[NBUFMAP_BUCKET];
7     struct spinlock bufmap_locks[NBUFMAP_BUCKET];
8 } bcache;
```

对 `bget`，先查桶看看在不在，不在就在全部桶里找一个最近最久未使用无引用 `buf` 插入。得出伪代码如下：

```
1 bget(dev, blockno) {
2     key := hash(dev, blockno);
3
4     acquire(bufmap_locks[key]); // 获取 key 桶的锁
5
6     // 查找 blockno 的缓存是否存在，若是直接返回，若否继续执行
7     if(b := look_for_blockno_in(bufmap[key])) {
8         b->refcnt++
9         release(bufmap_locks[key]);
10        return b;
11    }
12
13    // 查找可驱逐缓存 b
14
15    least_recently := NULL;
16
17    for i := [0, NBUFMAP_BUCKET) { // 遍历所有的桶
18        acquire(bufmap_locks[i]); // 获取第 i 桶的锁
19
20        b := look_for_least_recently_used_with_no_ref(bufmap[key]);
21        // 如果找到未使用时间更长的空闲块
```

```

22     if(b.last_use < least_recently.last_use) {
23         least_recently := b;
24     }
25
26     release(bufmap_locks[i]); // 查找结束后，释放第 i 桶的锁
27 }
28
29 b := least_recently;
30
31 // 驱逐 b 原本存储的缓存（将其从原来的桶删除）
32 evict(b);
33
34 // 将 b 加入到新的桶
35 append(bucket[key], b);
36
37 release(bufmap_locks[key]); // 释放 key 桶的锁
38
39 // 设置 b 的各个属性
40 setup(b);
41 return b;
42 }

```

上述代码可能存在的问题：

缓存驱逐时，每次扫描一个桶前获取锁，扫描后放锁。放锁后，得到的缓存就不可靠了。因为在放锁后，别的 CPU 可能调用 `bget` 请求那个桶，让本来是空的 `least_recently` 不再空。

解决：早到这样的桶后，不放桶锁，直到完成任务。也就是说，每次维护最值的时候，维护该最值下标，切换下标时放掉原本的锁，然后直到 `append` 完再放它自己的锁。

即：

```

1  bget(dev, blockno) {
2      acquire(bufmap_locks[key]); // 获取 key 桶锁
3
4      // 查找 blockno 的缓存是否存在，若是直接返回，若否继续执行
5      if(b := look_for_blockno_in(bufmap[key])) {
6          b->refcnt++
7          release(bufmap_locks[key]);
8          return b;
9      }
10
11     // 缓存不存在，查找可驱逐缓存 b
12
13     least_recently := NULL;
14     holding_bucket := -1;
15
16     for i := [0, NBUFMAP_BUCKET) { // 遍历所有的桶
17         acquire(bufmap_locks[i]); // 获取第 i 桶的锁
18
19         b := look_for_least_recently_used_with_no_ref(bufmap[key]);
20         // 如果找到未使用时间更长的空闲块（新的 least_recently）
21
22         if(b.last_use >= least_recently.last_use) {
23             release(bufmap_locks[i]); // 该桶中没有找到新的 least_recently，释放该桶

```



24

25 此时 CPU1 等待 CPU2，而 CPU2 在等待 CPU1，陷入死锁！

考虑如何破除死锁条件：

1. 互斥。一个桶被一个 CPU 处理，这个锁是不能解除的，无法破除。
2. 请求保持。下面讨论。
3. 不剥夺。不能回退，无法破除。
4. 循环等待。改变访问顺序，如只遍历左侧的桶，但这样可能在有空闲时找不到空闲，不合理，不能破除。

考虑请求保持，先不保持，也就是说本来的桶查完存不存在后，直接把它放掉，然后在后面需要 append 时再重新请求一次。再次考虑上述情形，将得到解决，有限等待即可。但是出现一个新问题，因为在驱逐之前放掉了自己，所以另一个 CPU 可以再次访问同一个块，然后同样进行驱逐，找到另一个块，然后使得一个区块有两个缓存，不满足一个区块只能有一个缓存的要求。

也就是说，解除请求保持是不可行的，无法破除。

所以引入驱逐锁，驱逐过程严格单线程。注意要放掉原本查找的桶锁，然后再拿驱逐。否则，会死锁，如线程1拿着桶A的锁请求驱逐，然后线程2拿着驱逐遍历到桶A。然后再来一个二次探测，拿到驱逐锁后，重新判断一下是不是桶里有了(因为等待锁期间，可能发生过驱逐)。这样能解决一个块两个缓存的问题。

但是缓存miss本身是稀有的，且miss后驱逐读盘的耗时比这些遍历耗时都要大，所以损失性能不会高很多。

乐观锁 optimistic locking：在冲突发生概率很小的关键区内，不使用独占的互斥锁，而是在提交操作前，检查一下操作的数据是否被其他线程修改。在这里，检测的是 blockno 的缓存是否已被加入。如果是，则代表冲突发生，需要特殊处理。在这里的特殊处理即为直接返回已加入的 buf。

乐观锁相比悲观锁 pessimistic locking，可以在冲突概率较低的场景下，如本个 bget，降低锁开销以及不必要的线性化，提升并行性，如本例对缓存是否存在可以提高并行。有时候还能用于避免死锁。

完整伪代码：

```
1  bget(dev, blockno) {
2      acquire(bufmap_locks[key]); // 获取 key 桶锁
3
4      // 查找 blockno 的缓存是否存在，若是直接返回，若否继续执行
5      if(b := look_for_blockno_in(bufmap[key])) {
6          b->refcnt++
7          release(bufmap_locks[key]);
8          return b;
9      }
10
11     // 注意这里的 acquire 和 release 的顺序
12     release(bufmap_locks[key]); // 先释放 key 桶锁，防止查找驱逐时出现环路死锁
13     acquire(eviction_lock);      // 获得驱逐锁，防止多个 CPU 同时驱逐影响后续判断
14
15     // **再次查找 blockno 的缓存是否存在**，若是直接返回，若否继续执行
16     // 这里由于持有 eviction_lock，没有任何其他线程能够进行驱逐操作，所以
17     // 没有任何其他线程能够改变 bufmap[key] 桶链表的结构，所以这里不事先获取
18     // 其相应桶锁而直接开始遍历是安全的。
19     if(b := look_for_blockno_in(bufmap[key])) {
```

```

20     acquire(bufmap_locks[key]); // 必须获取，保护非原子操作 `refcnt++`
21     b->refcnt++;
22     release(bufmap_locks[key]);
23
24     release(eviction_lock);
25     return b;
26 }
27
28 // 缓存不存在，查找可驱逐缓存 b
29
30 holding_bucket := -1; // 当前持有的桶锁
31 for i := [0, NBUFMAP_BUCKET) {
32     acquire(bufmap_locks[i]); // 请求时不持有 key 桶锁，不会出现环路等待
33     if(b := look_for_least_recently_used_with_no_ref(bufmap[key])) {
34         if(holding_bucket != -1) release(bufmap_locks[holding_bucket]);
35         holding_bucket := i;
36         // 如果找到新的未使用时间更长的空闲块，则将原来的块所属桶的锁释放掉，保持新块所属桶
           的锁...
37     } else {
38         release(bufmap_locks[holding_bucket]);
39     }
40 }
41
42 acquire(bufmap_locks[key]); // 再次获取 key 桶锁
43 append(b, bucket[key]);     // 将 b 加入到新的桶
44 release(bufmap_locks[key]); // 释放 key 桶锁
45
46 release(eviction_lock);     // 释放驱逐锁
47
48 // 设置 b 的各个属性
49 setup(b);
50 return b;
51 }

```

具体代码：

对 `kernel/buf.h` 结构体添加成员：

```

1  uint lastuse;
2  //可以删掉 struct buf *prev;

```

在 `kernel/bio.c`，选一个质数当哈希表桶数：

```

1  #define NBUFMAP_BUCKET 13

```

定义哈希函数将，两个变量拼在一起然后取模：

```

1  #define BUFMAP_HASH(dev, blockno) (((dev)<<27)|(blockno))%NBUFMAP_BUCKET)

```

对 `bcache` 结构体重定义为：

```

1 struct {
2     struct buf buf[NBUF];
3     struct spinlock eviction_lock;
4     struct buf bufmap[NBUFMAP_BUCKET];
5     struct spinlock bufmap_locks[NBUFMAP_BUCKET];
6 } bcache;

```

初始化修改为:

```

1 void
2     binit(void)
3 {
4     // Initialize bufmap
5     for(int i=0;i<NBUFMAP_BUCKET;i++) {
6         initlock(&bcache.bufmap_locks[i], "bcache_bufmap");
7         bcache.bufmap[i].next = 0;
8     }
9
10    // Initialize buffers
11    for(int i=0;i<NBUF;i++){
12        struct buf *b = &bcache.buf[i];
13        initsleeplock(&b->lock, "buffer");
14        b->lastuse = 0;
15        b->refcnt = 0;
16        // put all the buffers into bufmap[0]
17        b->next = bcache.bufmap[0].next;
18        bcache.bufmap[0].next = b;
19    }
20
21    initlock(&bcache.eviction_lock, "bcache_eviction");
22 }

```

bget 完整代码:

```

1 static struct buf*
2     bget(uint dev, uint blockno)
3 {
4     struct buf *b;
5
6     uint key = BUFMAP_HASH(dev, blockno);
7
8     acquire(&bcache.bufmap_locks[key]);
9
10    // Is the block already cached?
11    for(b = bcache.bufmap[key].next; b; b = b->next){
12        if(b->dev == dev && b->blockno == blockno){
13            b->refcnt++;
14            release(&bcache.bufmap_locks[key]);
15            acquiresleep(&b->lock);
16            return b;
17        }
18    }
19
20    // Not cached.

```

```

21
22     // to get a suitable block to reuse, we need to search for one in all
the buckets,
23     // which means acquiring their bucket locks.
24     // but it's not safe to try to acquire every single bucket lock while
holding one.
25     // it can easily lead to circular wait, which produces deadlock.
26     release(&bcache.bufmap_locks[key]);
27     // we need to release our bucket lock so that iterating through all the
buckets won't
28     // lead to circular wait and deadlock. however, as a side effect of
releasing our bucket
29     // lock, other cpus might request the same blockno at the same time and
the cache buf for
30     // blockno might be created multiple times in the worst case. since
multiple concurrent
31     // bget requests might pass the "Is the block already cached?" test and
start the
32     // eviction & reuse process multiple times for the same blockno.
33     //
34     // so, after acquiring eviction_lock, we check "whether cache for
blockno is present"
35     // once more, to be sure that we don't create duplicate cache bufs.
36     acquire(&bcache.eviction_lock);
37
38     // Check again, is the block already cached?
39     // no other eviction & reuse will happen while we are holding
eviction_lock,
40     // which means no link list structure of any bucket can change.
41     // so it's ok here to iterate through `bcache.bufmap[key]` without
holding
42     // it's cooresponding bucket lock, since we are holding a much stronger
eviction_lock.
43     for(b = bcache.bufmap[key].next; b; b = b->next){
44         if(b->dev == dev && b->blockno == blockno){
45             acquire(&bcache.bufmap_locks[key]); // must do, for `refcnt++`
46             b->refcnt++;
47             release(&bcache.bufmap_locks[key]);
48             release(&bcache.eviction_lock);
49             acquiresleep(&b->lock);
50             return b;
51         }
52     }
53
54     // Still not cached.
55     // we are now only holding eviction lock, none of the bucket locks are
held by us.
56     // so it's now safe to acquire any bucket's lock without risking
circular wait and deadlock.
57
58     // find the one least-recently-used buf among all buckets.
59     // finish with it's corresponding bucket's lock held.
60     struct buf *before_least = 0;
61     uint holding_bucket = -1;
62     for(int i = 0; i < NBUFMAP_BUCKET; i++){

```

```

63     // before acquiring, we are either holding nothing, or only holding
locks of
64     // buckets that are *on the left side* of the current bucket
65     // so no circular wait can ever happen here. (safe from deadlock)
66     acquire(&bcache.bufmap_locks[i]);
67     int newfound = 0; // new least-recently-used buf found in this
bucket
68     for(b = &bcache.bufmap[i]; b->next; b = b->next) {
69         if(b->next->refcnt == 0 && (!before_least || b->next->lastuse <
before_least->next->lastuse)) {
70             before_least = b;
71             newfound = 1;
72         }
73     }
74     if(!newfound) {
75         release(&bcache.bufmap_locks[i]);
76     } else {
77         if(holding_bucket != -1)
release(&bcache.bufmap_locks[holding_bucket]);
78         holding_bucket = i;
79         // keep holding this bucket's lock....
80     }
81 }
82 if(!before_least) {
83     panic("bget: no buffers");
84 }
85 b = before_least->next;
86
87 if(holding_bucket != key) {
88     // remove the buf from it's original bucket
89     before_least->next = b->next;
90     release(&bcache.bufmap_locks[holding_bucket]);
91     // rehash and add it to the target bucket
92     acquire(&bcache.bufmap_locks[key]);
93     b->next = bcache.bufmap[key].next;
94     bcache.bufmap[key].next = b;
95 }
96
97 b->dev = dev;
98 b->blockno = blockno;
99 b->refcnt = 1;
100 b->valid = 0; //本来就有, 不用动
101 release(&bcache.bufmap_locks[key]);
102 release(&bcache.eviction_lock);
103 acquiresleep(&b->lock); //本来就有, 不用动
104 return b;
105 }

```

`brelse` 完整代码:

```

1 void
2 brelse(struct buf *b)
3 {
4     if(!holdingsleep(&b->lock))
5         panic("brelse");

```

```

6
7     releasesleep(&b->lock);
8
9     uint key = BUFMAP_HASH(b->dev, b->blockno);
10
11     acquire(&bcache.bufmap_locks[key]);
12     b->refcnt--;
13     if (b->refcnt == 0) {
14         b->lastuse = ticks;
15     }
16     release(&bcache.bufmap_locks[key]);
17 }

```

对 `bpin`, `bunpin`, 都用哈希取对应的:

```

1 void
2 bpin(struct buf *b) {
3     uint key = BUFMAP_HASH(b->dev, b->blockno);
4
5     acquire(&bcache.bufmap_locks[key]);
6     b->refcnt++;
7     release(&bcache.bufmap_locks[key]);
8 }
9
10 void
11 bunpin(struct buf *b) {
12     uint key = BUFMAP_HASH(b->dev, b->blockno);
13
14     acquire(&bcache.bufmap_locks[key]);
15     b->refcnt--;
16     release(&bcache.bufmap_locks[key]);
17 }

```

执行测试, 通过测试:

```

1 make qemu
2 bcachetest
3 make grade

```

`usertests` 没过。

## lab9 file system

[要求](#), 预备:

```

1 git checkout origin2/fs
2 chmod -R 777 /home/starsinhands/xv6-labs-2021

```

## large files

### 要求

目标：提高 xv6 文件的最大大小。

目前限制单文件是 268 块，每块 1024 字节，限制的原因是索引节点(inode)包含 12 个直接块号，1 个直接间接块号，指向一个块，该块拥有额外 256 个块，即  $12 + 256 = 268$ 。

要求扩充到能写 65803 个块的文件，但不能修改 xv6 对 268 块单文件的限制。所以要求修改文件系统代码，实现二重间接块，将一个直接改二重，实现最终  $11 + 256 + 256^2 = 65803$  块。最终通过 `bigfile` 指令。

`mkfs` 命令创建 xv6 文件系统映像，计算共有多少块可用，由 `kernel/param.h` 的 `FSSIZE` 决定，设为 200000 块。其中有 70 个元数据块，这些块用来描述文件系统，其他是数据块。构建失败建议 `make clean`。

磁盘 inode 存储格式在 `kernel/fs.h`，在手册 85 页图 8-3 展示了当前的直接和一级间接的示意图。12 个直接，数目常量为 `NDIRECT`。其中一个块号为 int 大小 4byte，一个块 1KB，故一个一级间接包含 256 个块地址，数目常量为 `NINDIRECT`。则一个文件的最大块为 `MAXFILE` 是这两个相加。`struct dinode` 有 `addrs[NDIRECT+1]` 表示这 12+1 个的地址指向。

找磁盘文件数据的代码在 `fs.c` 的 `bmap()`，若需要就分配新的块。可以看到传入 `bn` 代表是第几个块地址(从 0 到 268)，如果在直接的下标内，直接查 `addrs` 返回，如果返回空，分配一个新的给它。如果是间接的，先读(如果间接空也分配)，然后读到的块当 `addrs[]` 来用，再找 `i-12` 同上理。读写时都调用这个函数，`bn` 是逻辑块号，`ip->addrs[]` 和传入 `bread` 的是磁盘块号。

任务：实现二重间址，修改 12 为 11，第 13th 块是二重。通过 `bigfile` 和 `usertests`。前者可能需要数分钟运行。

提示：

- 当将一个直接转二重时，二重为  $256^2$ ，直接为 1，增量  $256^2 - 1$
- 若修改 `NDIRECT` 定义，需要修改 `struct inode` 的 `addrs[]` 的定义，在 `file.h`，确保与 `struct dinode` 的 `addrs[]` 元素数相同。创建新的 `fs.img`。
- 如果崩溃，删掉 `fs.img`，然后重新 `make` 一个。
- 每个 `bread` 的块要 `brelease`。
- 只在需要时分配一/二重间接块。
- 确保 `itrunc` 释放文件的全部块。

扩展：完成三重间址

### 实现

修改 `fs.h` 的定义：

```
1 #define NDIRECT 11
2 #define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT * NINDIRECT)
```

对 `struct dinode`：

```
1 uint addrs[NDIRECT+2];
```

对 file.h 的 struct inode:

```
1  uint addr[NINDIRECT+2];
```

对 fs.c 的 bmap 函数, 在 panic 前, 添加:

```
1  bn -= NINDIRECT;
2  if(bn < NINDIRECT * NINDIRECT) {
3      // 2indirect addr
4      if((addr = ip->addr[NINDIRECT+1]) == 0)
5          ip->addr[NINDIRECT+1] = addr = balloc(ip->dev);
6
7      // 1indirect addr
8      bp = bread(ip->dev, addr);
9      a = (uint*)bp->data;
10     if((addr = a[bn/NINDIRECT]) == 0){
11         a[bn/NINDIRECT] = addr = balloc(ip->dev);
12         log_write(bp);
13     }
14     brelse(bp);
15
16     //addr
17     bn %= NINDIRECT;
18     bp = bread(ip->dev, addr);
19     a = (uint*)bp->data;
20     if((addr = a[bn]) == 0){
21         a[bn] = addr = balloc(ip->dev);
22         log_write(bp);
23     }
24     brelse(bp);
25     return addr;
26 }
```

同理, itrunc 函数, 仿照上下文, 在最后两行前, 添加:

```
1  if(ip->addr[NINDIRECT+1]){
2      bp = bread(ip->dev, ip->addr[NINDIRECT+1]);
3      a = (uint*)bp->data;
4      for(j = 0; j < NINDIRECT; j++){
5          if(a[j]) {
6              struct buf *bp2 = bread(ip->dev, a[j]);
7              uint *a2 = (uint*)bp2->data;
8              for(int k = 0; k < NINDIRECT; k++){
9                  if(a2[k])
10                     bfree(ip->dev, a2[k]);
11              }
12              brelse(bp2);
13              bfree(ip->dev, a[j]);
14          }
15      }
16      brelse(bp);
17      bfree(ip->dev, ip->addr[NINDIRECT+1]);
18      ip->addr[NINDIRECT + 1] = 0;
```

进行测试，通过测试：

```
1 make clean
2 make qemu
3 bigfile
4 usertests
```

## symbolic links

### 要求

创建符号链接/软链接。与硬链接很像，但硬链接严格指向同盘文件，符号链接可以跨磁盘设备。但 xv6 不支持多设备。该 lab 帮助理解文件路径名查找。

实现系统调用 `symlink(char* target, char* path)`，创建一个在 `path` 的指向 `target` 的链接。可以看 `man` 页。将 `symlinktest` 加到 `makefile`，应当通过该测试和 `usertests`。

提示：

- 系统调用相关补充，参见 `user/usys.pl`，`user/user.` 和 `kernel/sysfile.c` 实现空的 `sys_symlink`。
- 添加文件类型 `T_SYMLINK` 到 `kernel/stat.h`
- 添加新的 `O_NOFOLLOW` flag 到 `kernel/fcntl.h`，可以在 `open` 系统调用使用，组合位。
- 即使 `target` 不存在也可以创建链接。放回 0 创建成功，-1 失败，对 `link`，`unlink` 同理。可以用 `inode` 的数据块存储。
- 修改 `open` 系统调用，是链接时打开它，不存在就 fail。
- 链接可能是多重的，需要递归知道源文件为止，且成环报错。可以到达某个递归层数就当循环了。
- 不必完成对文件夹的符号链接。

### 实现

在 `kernel/sysfile.c` 创建新函数：

```
1 uint64
2     sys_symlink(void)
3 {
4     struct inode *ip;
5     char target[MAXPATH], path[MAXPATH];
6     if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
7         return -1;
8
9     begin_op();
10
11     ip = create(path, T_SYMLINK, 0, 0);
12     if(ip == 0){
13         end_op();
14         return -1;
15     }
16
17     // use the first data block to store target path.
18     if(writei(ip, 0, (uint64)target, 0, strlen(target)) < 0) {
```

```

19     end_op();
20     return -1;
21 }
22
23 iunlockput(ip);
24
25 end_op();
26 return 0;
27 }

```

- 根据本文件其他函数可知取 `str` 参数的格式，画瓢即可
- 同理，根据上下文，知道要用 `begin_op` 和 `end_op`。
- 查手册 98 页，`create` 函数创建新 `inode`，是 `open`，`mkdir`，`mkdev` 三个调用的结合。其中 `major` 和 `minor` 跟设备号有关，不用管，跟上下文一样传 0 即可。
- `writei` 可以认为就是写数据进块，模仿上下文取格式即可。
- 因为 `create` 弄了锁，卒子后腰解锁。

在 `kernel/fcntl.h` 定义 `flag` 位为最高位的下一位：

```

1 #define O_NOFOLLOW 0x800

```

在 `kernel/stat.h` 添加下一个常量：

```

1 #define T_SYMLINK 4

```

修改 `kernel/sysfile.c` 的 `sys_open` 第一个 `else` 修改为：

```

1 int symlink_depth = 0;
2 while(1) { // recursively follow symlinks
3     if((ip = namei(path)) == 0){
4         end_op();
5         return -1;
6     }
7     ilock(ip);
8     if(ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
9         if(++symlink_depth > 10) {
10             // too many layer of symlinks, might be a loop
11             iunlockput(ip);
12             end_op();
13             return -1;
14         }
15         if(readi(ip, 0, (uint64)path, 0, MAXPATH) < 0) {
16             iunlockput(ip);
17             end_op();
18             return -1;
19         }
20         iunlockput(ip);
21     } else {
22         break;
23     }
24 }
25 if(ip->type == T_DIR && omode != O_RDONLY){

```

```

26     iunlockput(ip);
27     end_op();
28     return -1;
29 }

```

- 头尾 if 画瓢。
- 从 `namei` 得到文件路径描述的 `inode`，如果不是 `NOFOLLOW`，就是说是多层嵌套，继续指。
- `readi` 读当前 `inode`，将读到的存到 `path`。

然后收收尾：

`kernel/syscall.h`

```

1 #define SYS_symlink 22

```

`kernel/syscall.c`

```

1 extern uint64 sys_symlink(void);
2 [SYS_symlink] sys_symlink,

```

`Makefile` 的 `UPROGS=`：

```

1 $U/_symlinktest\

```

`user/user.h`

```

1 int symlink(char*, char*);

```

`uesr/usys.pl`

```

1 entry("symlink");

```

通过测试：

```

1 make clean
2 make qemu
3 symlinktest
4 usertests

```

## lab10 mmap

[要求](#)，预备：

```

1 git checkout origin2/mmap
2 chmod -R 777 /home/starsinhands/xv6-labs-2021

```

## 要求

`mmap` 和 `munmap` 允许 UNIX 程序详细控制地址空间，可以用于进程间共享内存，映射文件到进程地址空间，作为用户级页面的一部分，如垃圾回收算法。本节解决内存映射文件。

可以查看手册看看函数定义，执行 `man 2 mmap`。如下：

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

- 在这个 lab，只需要实现映射一个文件，所以 `addr` 认为恒为 0，即核自己决定哪个虚拟地址去映射。
- 返回值是映射文件的虚拟地址，如果失败了返回 `0xffffffffffffffff` 即 ull 最大值。
- `length` 是要映射的字节数，可能不等于文件长度。
- `prot` 表示是否内存是可读、可写、可执行，有 `PROT_READ` 或 `PROT_WRITE` 或都有。
- `flags` 可以是 `MAP_SHARED` 即任何修改马上写回文件；或 `MAP_PRIVATE` 表示不会写回。其他位不需要实现。
- `fd` 是文件描述符，表示要映射的文件。设 `offset` 恒为 0，即总是从文件开头开始映射。

如果多个线程同一个 `MAP_SHARED` 文件不共享物理页也可以。

`munmap(addr, length)` 取消映射。如果 `MAP_SHARED` 修改了，先写回去。`munmap` 在这里只会覆盖从开头或结尾开始的一段，而不是中间的一段。不一定跟 `mmap` 的长度一样。

通过 `mmaptest`，`usertests`。

提示：

- 添加 `_mmaptest` 到 `UPROGS`，添加 `mmap`，`munmap` 系统调用。在 `kernel/fcntl.h` 定义了 `PROT_READ` 等。
- 懒填页表。`mmap` 不分配物理内存或读文件，这些通过 `usertrap` 来做，来提高速度和支持大文件（比物理内存大）。
- 追踪每个进程 `mmap` 映射了什么，定义数据结构对应 VMA(虚存空间)如 lecture15 所描述，记录地址、长度、权限等。xv6 没有内存分配器，可以声明固定大小的 VMA，从这个数组里分配，大小为 16 应该够了。
- 找到没用过的地址空间，添加一个 VMA 给进程表映射区域，包含对 `struct file` 的指针，增加引用计数，免得文件关闭就释放了(如 `filedup`)。
- 映射区域要分配物理页时要能引起页错误，读 4096 字节的文件到这个页，映射到用户地址空间，阅读 `readi`。记得正确设置权限。
- 实现 `munmap`，早到 VMA 范围，使用 `uvmunmap` 取消映射，删除了全部映射就减少引用计数到 `struct file`。修改过写回去，参考 `filewrite`。
- 脏位 `D` 表示一个页有没有被写，但这里可以忽略。
- 修改 `exit`，来取消进程的映射。
- 修改 `fork`，确保父子映射相同，增加计数器，可以父子不共享。

额外：

- 两进程同一文件映射，共享物理页。

- 如果在高速缓存有页面了，直接用，这将要求文件块与页大小一样，即设 `BSIZE=4096`。将其弄到高速缓存上，记录引用计数。
- 降低冗余，在懒分配和文件之间，如创建 VMA 懒分配区域。
- 修改 `exec`，使用 VMA 不同区，可以获得按需分页(on-demand-paged)的可执行代码，让启动程序更快，将不用从文件系统读取数据。
- 实现页面换入换出(page-in, page-out)，将进程一部分移出到磁盘，如果物理内存不足，需要时再移动进去。

## 实现

简单来说就是支持将文件映射到一片用户虚拟内存区域内，并且支持将对其的修改写回磁盘。

`mmap` 指令除了可以用来将文件映射到内存上，还可以用来将创建的进程间共享内存映射到当前进程的地址空间内。本 lab 只需实现前一功能即可。

在手册 38 页有一张图，描述了进程地址空间分配的各内容的顺序。为了让 map 文件用的地址空间不要和本来有用的东西冲突，所以把 map 弄得尽可能高，读图，可以弄到 trapframe 下方，向下生长。

在 `kernel/memlayout.h` 定义：

```
1 #define MMAPEND TRAPFRAME
```

从 END 往下生长，该地址为起始地址。

在 `kernel/proc.h` 定义 VMA 结构体如下，即上述 `mmap` 参数加 valid 标志：

```
1 struct vma {
2     int valid;
3     uint64 vstart;
4     uint64 sz;
5     struct file *f;
6     int prot;
7     int flags;
8     uint64 offset;
9 };
```

然后给 `proc` 尾部加 16 个 VMA：

```
1 #define NVMA 16
2 // ...
3 struct vma vmas[NVMA];
```

`mmap` 功能是找到 VMA 空槽，计算用到的最低虚存作为新 VMA 的结尾地址，然后当前映射到最低以下(向下生长)，然后增加计数，直接通过 `filedup` 来做。

在 `kernel/sysfile.c`，添加：

```
1 #include "memlayout.h"
2 uint64
3 sys_mmap(void)
4 {
5     uint64 addr, sz, offset;
6     int prot, flags, fd; struct file *f;
```

```

7
8     if(argaddr(0, &addr) < 0 || argaddr(1, &sz) < 0 || argint(2, &prot) < 0
9         || argint(3, &flags) < 0 || argfd(4, &fd, &f) < 0 || argaddr(5,
&offset) < 0 || sz == 0)
10         return -1;
11
12     if(!f->readable && (prot & (PROT_READ)))
13         || (!f->writable && (prot & PROT_WRITE) && !(flags & MAP_PRIVATE)))
14         return -1;
15
16     sz = PGROUNDUP(sz);
17
18     struct proc *p = myproc();
19     struct vma *v = 0;
20     uint64 vaend = MMAPEND; // non-inclusive
21
22     // mmaptest never passed a non-zero addr argument.
23     // so addr here is ignored and a new unmapped va region is found to
24     // map the file
25     // our implementation maps file right below where the trapframe is,
26     // from high addresses to low addresses.
27
28     // Find a free vma, and calculate where to map the file along the way.
29     for(int i=0; i<NVMA; i++) {
30         struct vma *vv = &p->vmass[i];
31         if(vv->valid == 0) {
32             if(v == 0) {
33                 v = &p->vmass[i];
34                 // found free vma;
35                 vv->valid = 1;
36             }
37             else if(vv->vastart < vaend) {
38                 vaend = PGROUNDDOWN(vv->vastart);
39             }
40         }
41
42         if(v == 0){
43             panic("mmap: no free vma");
44         }
45
46         v->vastart = vaend - sz;
47         v->sz = sz;
48         v->prot = prot;
49         v->flags = flags;
50         v->f = f; // assume f->type == FD_INODE
51         v->offset = offset;
52
53         filedup(v->f);
54
55         return v->vastart;
56     }

```

- 读参数依葫芦画瓢
- 如果不可读申请读，或不可写要申请写且要写回盘，权限不足直接返回

- 把申请大小 `sz` 弄到整块去
- 找到首个没用到的 VMA(valid=0)，然后找到最低地址，从最低地址往下分配一块地址给新的 VMA
- `filedup` 本身是写好的原子计数加一
- 返回这一块的最低地址

参考 lab5，实现懒加载，对 `kernel/trap.c`，增加整体判断逻辑，else 改造为：

```

1  else {
2      uint64 va = r_stval();
3      if((r_scause() == 13 || r_scause() == 15)){ // vma lazy allocation
4          if(!vmatrylazytouch(va)) {
5              goto unexpected_scause;
6          }
7      } else {
8          unexpected_scause:
9          printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p-
>pid);
10         printf("          sepc=%p stval=%p\n", r_sepc(), r_stval());
11         p->killed = 1;
12     }
13 }

```

在 `kernel/sysfile.c` 添加辅助函数：

```

1  // find a vma using a virtual address inside that vma
2  struct vma *findvma(struct proc *p, uint64 va) {
3      for(int i=0; i<NVMA; i++) {
4          struct vma *vv = &p->vmass[i];
5          if(vv->valid == 1 && va >= vv->vstart && va < vv->vstart + vv->sz)
6          {
7              return vv;
8          }
9      }
10     return 0;

```

继续添加上述未实现函数 `vmatrylazytouch`：

```

1  // finds out whether a page is previously lazy-allocated for a vma
2  // and needed to be touched before use.
3  // if so, touch it so it's mapped to an actual physical page and contains
4  // content of the mapped file.
5  int vmatrylazytouch(uint64 va) {
6      struct proc *p = myproc();
7      struct vma *v = findvma(p, va);
8      if(v == 0) {
9          return 0;
10     }
11
12     // printf("vma mapping: %p => %d\n", va, v->offset + PGROUNDDOWN(va - v-
>vstart));
13
14     // allocate physical page

```

```

15     void *pa = kalloc();
16     if(pa == 0) {
17         panic("vmaazytouch: kalloc");
18     }
19     memset(pa, 0, PGSIZE);
20
21     // read data from disk
22     begin_op();
23     ilock(v->f->ip);
24     readi(v->f->ip, 0, (uint64)pa, v->offset + PGROUNDDOWN(va - v->vstart),
    PGSIZE);
25     iunlock(v->f->ip);
26     end_op();
27
28     // set appropriate perms, then map it.
29     int perm = PTE_U;
30     if(v->prot & PROT_READ)
31         perm |= PTE_R;
32     if(v->prot & PROT_WRITE)
33         perm |= PTE_W;
34     if(v->prot & PROT_EXEC)
35         perm |= PTE_X;
36
37     if(mappages(p->pagetable, va, PGSIZE, (uint64)pa, PTE_R | PTE_W | PTE_U)
    < 0) {
38         panic("vmaazytouch: mappages");
39     }
40
41     return 1;
42 }

```

具体而言，逻辑如下：

- 查所在 VMA，分配一块物理内存，分配失败就返回
- 分配成功，读磁盘将文件数据读取并弄入物理内存，前后包上文件系统头和文件锁
- 设置 RWX 权限位
- 参考前面的 lab，创建 va 到 pa 的映射，使用 `mappages`

在 `kernel/vm.c` 添加：

```

1  #include "fcntl.h"
2  #include "spinlock.h"
3  #include "sleeplock.h"
4  #include "file.h"
5  #include "proc.h"
6  //顺序很重要
7
8  // Remove n BYTES (not pages) of vma mappings starting from va. va must be
9  // page-aligned. The mappings NEED NOT exist.
10 // Also free the physical memory and write back vma data to disk if
    necessary.
11 void
12 vmaunmap(pagetable_t pagetable, uint64 va, uint64 nbytes, struct vma *v)
13 {

```

```

14     uint64 a;
15     pte_t *pte;
16
17     // printf("unmapping %d bytes from %p\n", nbytes, va);
18
19     // borrowed from "uvmunmap"
20     for(a = va; a < va + nbytes; a += PGSIZE){
21         if((pte = walk(pagetable, a, 0)) == 0)
22             continue;
23         if(PTE_FLAGS(*pte) == PTE_V)
24             panic("sys_munmap: not a leaf");
25         if(*pte & PTE_V){
26             uint64 pa = PTE2PA(*pte);
27             if((*pte & PTE_D) && (v->flags & MAP_SHARED)) { // dirty, need
to write back to disk
28                 begin_op();
29                 ilock(v->f->ip);
30                 uint64 aoff = a - v->vstart; // offset relative to the
start of memory range
31                 if(aoff < 0) { // if the first page is not a full 4k page
32                     writei(v->f->ip, 0, pa + (-aoff), v->offset, PGSIZE +
aoff);
33                 } else if(aoff + PGSIZE > v->sz){ // if the last page is
not a full 4k page
34                     writei(v->f->ip, 0, pa, v->offset + aoff, v->sz - aoff);
35                 } else { // full 4k pages
36                     writei(v->f->ip, 0, pa, v->offset + aoff, PGSIZE);
37                 }
38                 iunlock(v->f->ip);
39                 end_op();
40             }
41             kfree((void*)pa);
42             *pte = 0;
43         }
44     }
45 }

```

具体执行功能：

- 在 VMA 项 `v` 中，从 `va` 低往高取消映射掉 `nbytes` 个字节，要求 `va` 对其页首地址
- 遍历这个地址范围 `[va, va + nbytes)` 的每一页，如果找不到页表就是取消掉了不用管
- 否则，若 `PTE_V` 位不存在，报错
- 存在，找到对应的 `pa`，看看是不是脏页且共享(不共享脏了不管)，如果是，写盘。具体而言，因为可能删掉了一些，所以开始可能不是对齐的，如果开头不对齐，对该页起始地址偏移到实际位置，写一整页的字节；如果结尾不对齐，同理对其，如果整页开头结尾都对其直接写。
- 无论如何，这最后把内存页给 free 掉。可以看到全部操作实际上是按页为单位的(比如计数器)

然后在 `kernel/sysfile.c` 添加对应的 `sys_munmap` 函数：

```

1  uint64
2  sys_munmap(void)
3  {
4      uint64 addr, sz;

```

```

5
6     if(argaddr(0, &addr) < 0 || argaddr(1, &sz) < 0 || sz == 0)
7         return -1;
8
9     struct proc *p = myproc();
10
11     struct vma *v = findvma(p, addr);
12     if(v == 0) {
13         return -1;
14     }
15
16     if(addr > v->vastart && addr + sz < v->vastart + v->sz) {
17         // trying to "dig a hole" inside the memory range.
18         return -1;
19     }
20
21     uint64 addr_aligned = addr;
22     if(addr > v->vastart) {
23         addr_aligned = PGROUNDUP(addr);
24     }
25
26     int nunmap = sz - (addr_aligned-addr); // nbytes to unmap
27     if(nunmap < 0)
28         nunmap = 0;
29
30     vmaunmap(p->pagetable, addr_aligned, nunmap, v); // custom memory page
    unmap routine for mmapped pages.
31
32     if(addr <= v->vastart && addr + sz > v->vastart) { // unmap at the
    beginning
33         v->offset += addr + sz - v->vastart;
34         v->vastart = addr + sz;
35     }
36     v->sz -= sz;
37
38     if(v->sz <= 0) {
39         fileclose(v->f);
40         v->valid = 0;
41     }
42
43     return 0;
44 }

```

具体而言：

- 读参数，找 VMA，判断错误情况(如参数不对)
- 如果是开头缺失，将其对其到页开头，然后从开头开始删，增加删除一段空白。即如页是 [0, 4096]，要删掉 512 开始的 100 字节，转化为删除 0 开始的 512 + 100 字节。
- 删掉对应的一段页，调用上面刚写好的函数
- 如果从开头开始删，调整开头坐标
- 如果删完了这个 VMA 空了，那么关闭这个文件，VMA 设为 valid=0

补充宏定义，在 `kernel/riscv.h` 添加：

```

1 #define PTE_G (1L << 5) // global mapping
2 #define PTE_A (1L << 6) // accessed
3 #define PTE_D (1L << 7) // dirty

```

- G 目前还没用到
- A 表示是否访问过，目前还没用到，上文 lab3 出现过
- D 是脏页，刚刚的函数出现过

对 `kernel/proc.c`，对 `allocproc` 函数，初始化时 VMA 设空，最后一个 return 前加：

```

1 for(int i=0;i<NVMA;i++) {
2     p->vmass[i].valid = 0;
3 }

```

对 `freeproc` 函数，最后一个 if 前添加：

```

1 for(int i = 0; i < NVMA; i++) {
2     struct vma *v = &p->vmass[i];
3     vmaunmap(p->pagetable, v->vstart, v->sz, v);
4 }

```

对 `fork` 函数，在 `safestrcpy` 前添加：

```

1 for(i = 0; i < NVMA; i++) {
2     struct vma *v = &p->vmass[i];
3     if(v->valid) {
4         np->vmass[i] = *v;
5         filedup(v->f);
6     }
7 }

```

即直接指针复制并增加计数器，但物理上不复制，做懒处理。

记得声明一下，因为 `vm.c` 无头文件，故 `kernel/defs.h`：

```

1 struct vma;
2 void vmaunmap(pagetable_t pagetable, uint64 va, uint64 nbytes, struct vma
3 *v);
4 int vmatrylazytouch(uint64 va);

```

其他琐碎工作：

`makefile` 的 `UPROGS` 添加：

```

1 $U/_mmaptest\

```

`user/usys.pl`：

```

1 entry("mmap");
2 entry("munmap");

```

user/user.h:

```
1 void* mmap(void*, int, int, int, int, uint);
2 int munmap(void*, int);
```

kernel/syscall.h:

```
1 #define SYS_mmap 22
2 #define SYS_munmap 23
```

kernel/syscall.c:

```
1 extern uint64 sys_mmap(void);
2 extern uint64 sys_munmap(void);
3 [SYS_mmap] sys_mmap,
4 [SYS_munmap] sys_munmap,
```

执行测试，通过测试：

```
1 make clean
2 make qemu
3 mmaptest
4 usertests
```

## 总结

### 项目背景

#### mit6.S081

Massachusetts/mæseɪˈtʃuːsɪts/ Institute of Technology

6: 电气工程与计算机科学（EECS Electrical Engineering and Computer Science）部门

S special, 081无特殊含义唯一标识。前身是 6.828

828 针对 x86，项目命名为 JOS；而本项目针对 RISC-V，项目命名为 xv6 (基于 Unix ver6)

除了 lab 和手册外，还有讲座(lecture)一个译文如下 [here](#)；项目[简介](#)

#### x86, RISC-V

RISC-V（读作 "risk-five"）是一种开源的指令集架构，基于 RISC（精简指令集计算机）原则设计。与 x86 不同，RISC-V 的指令集相对简单和模块化，旨在提供高效且易于实现的硬件设计。由于是开源的，RISC-V 吸引了大量的社群和产业支持，正在逐渐应用于嵌入式系统、服务器，甚至高性能计算（HPC）领域。

x86 是一种微处理器架构，最初由 Intel 设计。这个架构已经存在了几十年，并且经历了多次迭代和扩展。x86 架构原来是基于 CISC（复杂指令集计算机）设计的，这意味着它有一个非常丰富和复杂的指令集。现在，x86 架构广泛应用于桌面、服务器和笔记本电脑等多种类型的计算机系统。

1. **设计哲学**: x86 是基于 CISC 设计, 而 RISC-V 是基于 RISC 设计。CISC 指令集通常更复杂, 可以完成更多的操作, 但可能会使硬件实现变得复杂。RISC 设计理念更倾向于简单和高效, 易于硬件实现。
2. **开放性**: RISC-V 是开源的, 任何人都可以自由地使用和修改其设计, 这促进了多样化和创新。而 x86 架构主要由 Intel 和 AMD 控制, 是专有的。
3. **应用范围**: x86 主要用于高性能计算系统, 如桌面电脑和服务器。RISC-V 因为其灵活性和低功耗特点, 更多地应用于嵌入式系统和物联网 (IoT) 设备。
4. **成熟度和生态系统**: x86 有着悠久的历史 and 庞大的生态系统, 包括各种操作系统、软件和工具的优秀支持。RISC-V 相对较新, 虽然生态系统正在快速发展, 但还没有达到 x86 的成熟度。
5. **教学用途**: xv6 最初是为 x86 架构设计的, 但现在也有基于 RISC-V 的版本。这两者之间的主要区别在于底层硬件架构和相应的汇编语言指令。

## 章节大纲

1. **util** 实现用户态程序, 熟悉系统调用使用
  - **sleep** 进程休眠
  - **pingpong** 两进程 ping 管道传输信息
  - **primes** 进程创建子进程, 父子通信
  - **find** 遍历文件系统查找符合文件名要求的文件
  - **xargs** 用管道连接多个命令, 上一个输出当下一个输入
  - **uptime** 输出运行时间
2. **system calls** 系统调用编写, 理解用户态内核态切换等
  - **trace** 执行一个用户程序, 监听它用到了哪些系统调用并输出, 以及中断保留现场的参数
  - **sysinfo** 输出系统信息如空闲内存数、运行进程数。
3. **page tables** 虚拟到物理内存映射的页表
  - **speed up system calls** 在用户态和内核态间共享只读区域来加速系统调用 (da ho)
  - **print a pagetable** 打印页表及其子页表
  - **detecting which pages have been accessed** 查询自上次查询以来, 有多少页面被访问过
4. **traps** 中断处理机制
  - **RISC-V assembly** 阅读代码理解对应的汇编代码
  - **backtrace** 输出调用栈的所有函数返回地址
  - **alarm** 执行自定义周期函数
5. **copy-on-write fork** COW 机制, 延迟复制页直到需要写
6. **multithreading** 多线程
  - **Uthread: switching between threads** 用户级多线程间的上下文切换
  - **Using threads** 使用多线程优化哈希表, 使用桶级锁
  - **Barrier** 实现检查点(线程屏障)
7. **networking** 实现网络收发包
8. **locks** 细化锁粒度优化并发性能

- `Memory allocator` 优化内存分配，将锁粒度从整个分配器精细到各 CPU 级；每个内存由单个 CPU 独占
- `Buffer cache` 磁盘文件高速缓存块从整个分配器锁细化到分哈希表每个桶一个锁；涉及页面置换，二次探测等并发编程技巧

#### 9. `file system`

- `large files` 修改一个直接索引块为二级间接块，提升单个文件的最大容量
- `symbolic links` 给文件创建符号链接/软链接

10. `mmap` 将一段磁盘文件地址懒映射到内存地址，在需要时读盘取数据

## 语法知识

### .c .h 区别

#### .c 文件

- **实现文件**：这里包含了代码的实现部分，如函数的具体操作。
- **编译**：`.c` 文件会被编译器编译成目标代码（`.o` 或 `.obj` 文件）。
- **执行代码**：包含可执行的代码和逻辑。

#### .h 文件

- **头文件**：这里通常包含函数声明，类型定义，宏定义等。
- **不编译**：`.h` 文件本身不会被编译，但会被包含（通过 `#include` 指令）在 `.c` 或其他 `.h` 文件中。

具体而言：

#### 不会被编译

- **不生成目标代码**：`.h` 文件本身通常不会直接被编译为目标代码（`.o` 或 `.obj` 文件）。
- **不直接参与链接**：因为它们不被单独编译，所以也不会直接参与到最终可执行文件或库文件的链接过程。

#### 会被包含

- **文本替换**：当你在一个 `.c` 或 `.cpp` 文件中用 `#include "filename.h"` 包含一个头文件时，编译器会将整个 `.h` 文件的内容文本性地“粘贴”到该位置。
- **声明可用**：一旦一个头文件被包含，那么这个头文件中声明的所有函数、变量和类型定义就可以在包含它的 `.c` 或 `.cpp` 文件中使用。
- **接口和声明**：`.h` 文件为编程者提供一个接口，使其知道可以使用哪些功能，但不需要知道这些功能是如何实现的。