



# **xv6实验报告**

**学号： 2150253**

**姓名： 王家梁**

**专业： 软件学院**

**课号： 42028702**

环境搭建

正式实验

lab1: Xv6 and Unix utilities

1.1 sleep

前置工作

功能实现

运行结果

思考

1.2 pingpong

前置工作

功能实现

运行结果

思考

1.3 primes

前置工作

功能实现

运行结果

思考

1.4 find

前置工作

功能实现

运行结果

思考

1.5 xargs

前置工作

功能实现

运行结果

思考

1.6 实验评分

lab2: syscall

2.1 trace

前置知识

实现过程

运行结果

思考

2.2 sysinfo

前置知识

实现过程

运行结果

思考

2.3 实验评分

lab3: page tables

3.1 Speed up system calls

前置知识  
实现过程  
运行结果  
思考

### 3.2 Print a page table

前置知识  
实现过程  
运行结果  
思考

### 3.3 Detecting which pages have been accessed

前置知识  
实现过程  
运行结果  
思考

### 3.4 实验评分

## lab4: traps

### 4.1 RISC-V assembly

### 4.2 Backtrace

前置知识  
实现过程  
运行结果  
思考

### 4.3 Alarm

前置知识  
实现过程  
运行结果  
思考

### 4.4 实验评分

## lab5: Copy-on-Write Fork for xv6

### 5.1 相关学习

### 5.2 Implement copy-on write

### 5.3 运行结果

### 5.4 实验小结

## lab6: Multithreading

### 6.1 Uthread: switching between threads

前置知识  
实现过程  
运行结果

### 6.2 Using threads

前置知识  
实现过程  
运行结果

### 6.3 Barrier

前置知识

实现过程

运行结果

思考

#### 6.4 实验评分

### lab7: networking

#### 7.1 Implement networking

#### 7.2 运行结果

### lab8: locks

#### 8.1 Memory allocator

前置知识

实现过程

运行结果

思考

#### 8.2 Buffer cache

前置知识

实现过程

运行结果

思考

#### 8.3 实验评分

### lab9: file system

#### 9.1 Large files

前置知识

实现过程

运行结果

思考

#### 9.2 Symbolic links

前置知识

实现过程

运行结果

思考

#### 9.3 实验评分

### lab10: mmap

#### 10.1 相关学习

#### 10.2 Implement mmap

#### 10.3 实验评分

#### 10.4 思考

# 环境搭建

考虑到mit官方文档上的hints

We strongly discourage students from using WSL for 6.1810 because it slows down the tests a lot, leading to unexpected timeouts on some labs.

这里选择安装Ubuntu 20.04系统搭建实验环境

- 安装开发工具与软件包

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

可以通过以下方式测试安装

```
qemu-system-riscv64 --version
```

```
QEMU emulator version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.11)  
Copyright (c) 2003-2021 Fabrice Bellard and the QEMU Project developers
```

以及至少一个 RISC-V 版本的 GCC:

```
riscv64-linux-gnu-gcc --version  
riscv64-unknown-elf-gcc --version  
riscv64-unknown-linux-gnu-gcc --version
```

```
riscv64-linux-gnu-gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0  
Copyright (C) 2021 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- 获取实验代码

配置好git信息后

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

clone实验代码

- 在lab文件夹下使用

```
make qemu
```

出现以下信息，说明环境配置成功

```
xv6 kernel is booting

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

- 最后，我选择了vscode作为编辑器

## 正式实验

### lab1: Xv6 and Unix utilities

This lab will familiarize you with xv6 and its system calls.

#### 1.1 sleep

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`.

#### 前置工作

- 阅读xv6 book相关章节
- 对用户没有传入参数等情况进行处理
- 通过`atoi`对字符串进行转换处理
- 使用系统调用`sleep`
- 在`Makefile`中添加程序
- 导入头文件

```
#include "kernel/types.h"
#include "user/user.h"
```

- `"kernel/types.h"` 提供了`int`类型的定义
- `"user/user.h"` 提供了系统调用`int sleep(int);`

## 功能实现

仿照“wc.c”中的传参方式，我们可以通过传入int argc和int \*argv[]的方式对用户输入进行处理

代码实现如下：

```
#include "kernel/types.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
    if (argc!=2){
        printf("参数错误!");
    }
    else{
        int sleepTime=atoi(argv[1]);
        sleep(sleepTime);
    }

    exit(0);
}
```

对参数正确性进行判断，输入正确时通过 `atoi` 进行类型转换，利用系统调用 `sleep` 进行功能处理

## 运行结果

在xv6-labs-2021目录下通过“`./grade-lab-util sleep`”进行验证，结果如下：

```
marco@marco-ASUS-TUF-Gaming-F15-FX506HE-FX506HE:~/桌面/xv6-labs-2021$ ./grade-lab-util sleep
make: "kernel/kernel"已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (0.6s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.1s)
```

验证通过！

## 思考

- `exit()`与`return`

观察其他程序可知，xv6中习惯用`exit()`而不是`return`来标识一个进程的结束。查阅相关资料可知，`return`返回函数值，是关键字；`exit`是一个函数。`return`是语言级别的，它表示了调用堆栈的返回；而`exit`是系统调用级别的，它表示了一个进程的结束。也就是说，`return`是函数的退出(返回)；`exit`是进程的退出。

对于`exit()`的返回值，通常`exit(0)`表示正常运行程序并退出，而`exit(x)`，`x`不为0表示异常退出。

- `sleep`调用

本次实验的一个注意点是，我们使用的系统调用是 `"user/user.h"` 中声明的 `sleep`，而不是linux系统自带的 `sleep`，在引入头文件时需要注意。

## 1.2 pingpong

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`.

### 前置工作

- 学习 `pipe` 的相关知识
- 学习 `pipe()`、`write()`、`fork()` 等可能会用到的函数
- 对于父进程、子进程的部分进行学习
- 导入头文件

```
#include "kernel/types.h"
#include "user/user.h"
```

- `"kernel/types.h"` 提供了类型的定义
- `"user/user.h"` 提供了相关系统调用

### 功能实现



这段代码利用了管道（pipe）机制实现了父进程和子进程之间的通信。父进程向子进程发送数据，子进程接收数据后打印，并向父进程发送数据，父进程接收数据后打印。通过管道的简单使用，设计两个单向管道，就可以实现本次实验的数据传输和同步要求。

代码实现如下：

```
#include "kernel/types.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
    //定义管道的读写端口
    //p1:父进程->子进程, p2:子进程->父进程
    //[0]为生产者, [1]为消费者
    int p1[2], p2[2];
    pipe(p1);
    pipe(p2);
    char buf[1];

    if(fork()==0){//子进程
        //关闭无用端口
        close(p2[0]);
        close(p1[1]);
        read(p1[0], buf, 1);
        printf("%d: received ping\n", getpid());
        write(p2[1], buf, 1);
        close(p2[1]);
    }
    else{//父进程
        //关闭无用端口
        close(p1[0]);
        close(p2[1]);
        write(p1[1], buf, 1);
        close(p1[1]);
        read(p2[0], buf, 1);
        printf("%d: received pong\n", getpid());
        wait(0);
    }
    exit(0);
}
```

## 运行结果

在xv6-labs-2021目录下通过“`./grade-lab-util pingpong`”进行验证，结果如下：

```
marco@marco-ASUS-TUF-Gaming-F15-FX506HE-FX506HE:~/桌面/xv6-labs-2021$ ./grade-lab-util pingpong
make: "kernel/kernel"已是最新。
== Test pingpong == pingpong: OK (0.7s)
```

验证通过！

## 思考

- 管道的学习

翻阅xv6 book中关于pipe的描述：

管道是一种小型的内核缓冲区，由一对文件描述符（一个用于读取，一个用于写入）作为接口，对外提供给进程使用。通过向管道的一端写入数据，使得该数据可以从管道的另一端进行读取。管道提供了进程间通信的一种方式。

通常设立一个`int[2]`的数组，通过`pipe()`函数创建一个管道，并返回两个相关联的文件描述符，一个用于读取数据，另一个用于写入数据。这两个文件描述符分别被称为管道的读端口和写端口。以`[0]`作为读端口，`[1]`作为写端口。

- `close()`、`read()`、`write()`函数的学习

`close()` 函数是一个系统调用，用于关闭打开的文件描述符。它的作用是终止对文件的访问，并释放与该文件描述符相关的系统资源。本实验中通过`close()` 函数关闭无用端口，将管道设计为单向管道。

`read()` 和 `write()` 函数是用于文件操作的系统调用，它们用于在文件描述符上进行读取和写入操作。他们需要三个参数。第一个参数指定文件描述符，达到从中读取/写入数据。第二个参数自动更新文件的读写位置，以便下一次读写操作可以从适当的位置开始。这允许进程连续地读写文件中的数据，而不需要手动管理。第三个参数指定要操作的数据的字节数。这两个函数的返回值是被读取或写入的数据的字节数。

- `read`使用错误时导致异步输出

在实验中，因为`read()`函数使用错误(关闭了错误的端口)导致遇到了进程没有同步的问题。(不过也证明了`printf`不是原子操作)

```
$ pingpong
4:3 :r erceecievievde d ppionng
```

查阅xv6 book相关部分可知，如果管道中没有可用的数据，对管道进行读取操作将会进行等待，指导数据被写入，或者所有引用写端的文件描述符被关闭。pingpong中通过read操作协调了父子进程的推进顺序，不会造成异步输出。

## 1.3 primes

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`.

### 前置工作

- 学习素数筛的方法
- 及时关闭不需要的文件描述符，防止xv6的资源用尽
- 注意进程间的同步
- 导入头文件

```
#include "kernel/types.h"
#include "user/user.h"
```

- `"kernel/types.h"` 提供了类型的定义
- `"user/user.h"` 提供了相关系统调用

### 功能实现

本实验实质上是利用素数筛的方法筛选素数，在对pipe有了掌握之后，本问题其实是一个算法的问题。在此使用递归的思想，将本进程筛选出的数依次通过管道传入下一个进程，直到处理完毕。

代码实现如下：

```
#include "kernel/types.h"
#include "user/user.h"

#define MAX_NUM 35

void
```

```

prime(int *p)
{
    int temp;
    int n;
    close(p[1]);
    if (read(p[0], &temp, sizeof(temp)) != sizeof(temp)) {
        exit(0);
    }

    printf("prime %d\n", temp);

    if (read(p[0], &n, 4) == sizeof(temp)) {
        int newp[2];
        pipe(newp);
        if (fork() == 0) {
            //子进程
            prime(newp);
        }
        else {
            //父进程
            close(newp[0]);
            if (n % temp != 0) {
                write(newp[1], &n, sizeof(n));
            }
            while (read(p[0], &n, 4) == sizeof(n)) {
                if (n % temp != 0) {
                    write(newp[1], &n, sizeof(n));
                }
            }
            close(p[0]);
            close(newp[1]);
            wait(0);
        }
    }
    exit(0);
}

int
main(int argc, char* argv[])
{
    //定义初始管道
    int p[2];

```

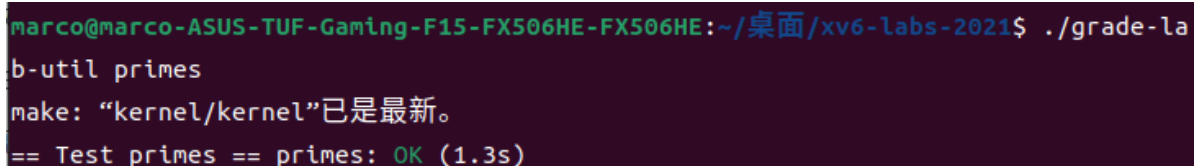
```

    pipe(p);
    if(fork()==0){
        //子进程
        prime(p);
    }
    else{
        //父进程
        //依次传出即可
        close(p[0]);
        for(int i=2;i<=MAX_NUM;i++){
            write(p[1],&i,sizeof(i));
        }
        close(p[1]);
        wait(0);
    }
    exit(0);
}

```

## 运行结果

在xv6-labs-2021目录下通过“`./grade-lab-util primes`”进行验证，结果如下：



```

marco@marco-ASUS-TUF-Gaming-F15-FX506HE-FX506HE:~/桌面/xv6-labs-2021$ ./grade-la
b-util primes
make: "kernel/kernel"已是最新。
== Test primes == primes: OK (1.3s)

```

验证通过！

## 思考

- wait()的使用

父进程可以调用 `wait()` 函数来等待其子进程的终止。在父进程调用 `wait()` 之后，如果子进程还未终止，父进程将被阻塞，直到子进程终止为止。这样可以确保父进程在子进程执行完毕后再继续执行，达到了进程的同步。

- 错误打印“\$”

其实考察算法可知，不使用 `wait()` 函数也可以让程序正常打印，但是会出现一些多余的“\$”符号，查阅相关资料可知，这是因为操作系统默认情况下，父进程结束时没有等待子进程的机制，所以子进程可能还在运行，而 shell 会继续等待用户输入。因此，在没有使用 `wait()` 函数等待子进程的情

况下，shell会继续打印\$符号。

通过使用`wait()`函数，父进程可以等待子进程的终止，直到子进程完全执行完毕后才退出。这样可以确保在父进程结束之前，所有子进程都已经终止，避免了出现额外的\$符号。

- 及时释放资源

在本实验的hints中特别提到了应当及时`close()`释放资源，除了出于避免资源泄露的问题以外，还有防止导致系统资源不足问题的考量。在xv6这个较为迷你的操作系统中，这个操作是非常需要的。

## 1.4 find

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`.

### 前置工作

- 查看`user/ls.c`进行学习
- 了解`dirent`、`stat`等数据结构
- 使用`strcmp`而不是`==`，后者比较的是指针的值

### 功能实现

在此使用DFS算法，对文件夹下的所有文件进行查找，若为文件，则比较其文件名与目标是否相同并进行相应操作，若为文件夹，则进行递归搜索。本实验思路较为清晰，注意使用的函数以及一些异常处理即可。

代码实现如下：

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

//找到最后的文件名
char*
fmtname(char *path)
{
    char *p = path;
    while (*p)
        p++;
}
```

```

    while (*p != '/' && p != path)
        p--;
    return p == path ? p : ++p;
}

void
DFS(char *path, char* fName)
{
    //DFS搜索所有文件
    //变量及处理参照ls.c
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;

    if((fd = open(path, 0)) < 0){
        fprintf(2, "ls: cannot open %s\n", path);
        return;
    }

    if(fstat(fd, &st) < 0){
        fprintf(2, "ls: cannot stat %s\n", path);
        close(fd);
        return;
    }

    switch(st.type){
        case T_FILE:
            //为文件时, 查看是否同名
            if(strcmp(fname(path), fName)==0){
                printf("%s\n", path);
            }
            break;
        case T_DIR:
            //为文件夹时, 先判断长度是否合法
            if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
                printf("find: path too long\n");
                break;
            }
            strcpy(buf, path);
            p = buf+strlen(buf);
            *p++ = '/';

```

```

        while(read(fd,&de,sizeof(de))==sizeof(de)){

            if(de.inum==0 || strcmp(de.name,".")==0 || strcmp(de.name,"..")==
0)

                continue;
                memmove(p,de.name,DIRSIZ);
                p[DIRSIZ]=0;
                DFS(buf,fName);
            }
            break;

        }
        close(fd);
    }

    int
    main(int argc, char *argv[])
    {
        if(argc!=3){
            printf("参数错误! ");
            exit(0);
        }
        DFS(argv[1],argv[2]);
        exit(0);
    }

```

## 运行结果

在xv6-labs-2021目录下通过“`./grade-lab-util find`”进行验证，结果如下：

```

marco@marco-ASUS-TUF-Gaming-F15-FX506HE-FX506HE:~/桌面/xv6-labs-2021$ ./grade-lab-util find
make: "kernel/kernel"已是最新。
== Test find, in current directory == find, in current directory: OK (0.5s)
== Test find, recursive == find, recursive: OK (1.0s)

```

验证通过！

## 思考

- 一开始使用 `user/ls.c` 提供的 `fmtname` 函数一直没有得到期望的结果，后面发现是因为其填充了空格，于是自己重写了一个函数
- 新的数据结构



- 目录项结构体 `dirent`，包含两个成员变量。`inum` 是一个无符号短整型，表示目录项对应文件的索引节点号。`name` 是一个字符数组，长度为 `DIRSIZ`，表示目录项的名称。
- 文件状态结构体 `stat`，包含了几个成员变量。`dev` 是一个整数，表示文件所在的磁盘设备。`ino` 是一个无符号整数，表示文件的索引节点号。`type` 是一个短整数，表示文件的类型，可以是上述定义的常量之一。`nlink` 是一个短整数，表示指向该文件的硬链接数量。`size` 是一个无符号64位整数，表示文件的大小。

## 1.5 xargs

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`.

### 前置工作

- `xargs()` 的学习
- 读取标准输入的方法
- `exec()` 的学习
- 使用 `kernel/param.h` 中的宏定义 `MAXARG`

### 功能实现

通过读取标准输入的数据，并将每行数据解析为命令行参数，然后创建子进程并使用 `exec()` 函数执行指定的命令。主进程等待子进程执行完成，就可以实现将标准输入的数据作为命令参数传递给指定命令的功能。具体实现为：创建一个字符串数组 `argvs[MAXARG]`，用于存储命令行参数。将除了第一个参数外的其他参数拷贝到 `argvs` 数组中。使用 `gets()` 函数循环从标准输入读取一行数据，并将数据存储在 `buf` 中。最后依次在子进程中调用 `exec()` 函数执行指定的命令。

代码实现如下：

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

int
main(int argc, char *argv[])
```

```
{  
    if(argc<2){  
        printf("参数错误! ");  
        exit(0);  
    }  
  
    //copy一下参数  
    char *argvs[MAXARG];  
    for(int i=1;i<argc;i++){  
        argvs[i-1]=argv[i];  
    }  
  
    char buf[64];  
  
    while(1){  
        int loc=argc-1;  
  
        //循环读取参数  
        //参数添加在argvs的后方  
        gets(buf, sizeof(buf));  
        if(buf[0]==0){  
            break;  
        }  
        argvs[loc]=buf;  
        //将空格替换为尾0  
        for(char*p=buf;*p!=0;p++){  
            if(*p==' '){  
                *p=0;  
                argvs[loc++]=p+1;  
            }  
            else if (*p=='\n'){  
                *p=0;  
            }  
        }  
        //调用子程序执行当前行  
        if(fork()==0){  
            exec(argv[1], argvs);  
        }  
    }  
    wait(0);  
    exit(0);  
}
```

## 运行结果

在xv6-labs-2021目录下通过“`./grade-lab-util xargs`”进行验证，结果如下：

```
marco@marco-ASUS-TUF-Gaming-F15-FX506HE-FX506HE:~/桌面/xv6-labs-2021$ ./grade-lab-util xargs
make: "kernel/kernel"已是最新。
== Test xargs == xargs: OK (1.5s)
```

验证通过！

## 思考

- `gets()` 函数

`gets()` 函数用于从标准输入读取字符串数据，并存储到指定的缓冲区 `buf` 中。它接收两个参数：`buf` 表示用于存储读取数据的缓冲区的指针，`max` 表示缓冲区的最大长度。当遇到换行时会停止读入，因此可以通过它实现本实验的数据读取。

- `exec()` 函数

`exec()` 函数用于在当前进程上执行一个新的程序。它接收两个参数：`path` 表示待执行程序的路径，`argv` 是一个字符串数组，包含传递给新程序的命令行参数。它可以加载新程序的代码和数据到内存中，并设置好正确的上下文环境，包括栈指针、参数传递等。这样，通过执行新程序的 `main(argc, argv)` 函数，实现了进程的切换和程序的执行。

- `xargs` 的理解

`xargs` 是 `execute arguments` 的缩写，它的作用是从标准输入中读取内容，并将此内容传递给它要协助的命令，并作为那个命令的参数来执行。实际使用中，通过 `xargs` 的方法可以实现一些例如 `kill` 掉一个txt文件中进程的功能(对txt进行echo输出，再通过读取标准输入依次调用 `kill` 处理)。总之，`xargs` 是一个非常实用的命令行工具，它可以方便地将标准输入中的数据作为参数传递给指定命令并执行，提供了灵活参数传递方式和选项，可以帮助简化命令行操作和提高工作效率。

## 1.6 实验评分

```
marco@marco-ASUS-TUF-Gaming-F15-FX506HE-FX506HE:~/桌面/xv6-labs-2021$ ./grade-lab-util
make: "kernel/kernel"已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (1.1s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.0s)
== Test primes == primes: OK (1.0s)
== Test find, in current directory == find, in current directory: OK (1.1s)
== Test find, recursive == find, recursive: OK (1.1s)
== Test xargs == xargs: OK (0.9s)
== Test time ==
time: OK
Score: 100/100
```

## lab2: syscall

### 2.1 trace

In this assignment you will add a system call tracing feature that may help you when debugging later labs. You'll create a new `trace` system call that will control tracing. It should take one argument, an integer "mask", whose bits specify which system calls to trace. For example, to trace the fork system call, a program calls `trace(1 << SYS_fork)`, where `SYS_fork` is a syscall number from `kernel/syscall.h`. You have to modify the xv6 kernel to print out a line when each system call is about to return, if the system call's number is set in the mask. The line should contain the process id, the name of the system call and the return value; you don't need to print the system call arguments. The `trace` system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.

#### 前置知识

- `syscall.c`

在 xv6 操作系统中, `syscall.c` 文件是内核中处理系统调用的关键文件。它包含了21个(在我做实验前)系统调用, 以及`syscall()`等关键函数, 下面是`syscall()`函数的功能。

首先获取当前进程, 并取得系统调用号

```
struct proc *p = myproc();
num = p->trapframe->a7;
```

myproc()函数的作用是获取当前进程的指针，在 RISC-V 架构下，系统调用号存储在寄存器 a7 中。

如果系统调用号有效，并且对应的系统调用处理函数存在，那么代码会调用该处理函数，并将处理结果保存在进程的 trapframe 结构中的 a0 寄存器中。如果系统调用号无效，或者对应的系统调用处理函数不存在，那么代码会输出一条错误信息，表示遇到了未知的系统调用，并将返回值设为 -1。

```
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
} else {
    printf("%d %s: unknown sys call %d\n",
           p->pid, p->name, num);
    p->trapframe->a0 = -1;
}
```

- trap

当用户程序调用 `syscall` 指令以要求内核为其做一些事情时，就会触发一个 trap 中断。xv6 中，通过内核处理所有 trap。trapframe 是保护用户寄存器，其包含指向当前进程的内核栈、usertrap 地址和内核页表地址等信息。在 `syscall.c` 中，可以通过 `argint()` 获取其中的值。

由于 `syscall()` 函数将系统调用处理结果存放在 a0 中

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

因此，在 trace 实现中，我们要通过 `argint(0, &mask)` 获取 mask 值

## 实现过程

- 修改 struct proc 结构体

根据官方 hints，我们在 `kernel/proc.h` 结构体下添加属性 `mask`

```
// kernel/proc.h
int mask;           //跟踪掩码 trace lab2
```

`mask` 是一个 32-bit 的整数，它的每一个二进制位都表示是否对对应 `num` 的 `syscall` 进行 trace。

- sysproc添加sys\_trace

仿照其他函数，我在 `sysproc.c` 中添加了 `sys_trace()` 函数

```
// kernel/sysproc.c
// trace lab2
uint64
sys_trace(void)
{
    int mask;
    if(argint(0, &mask) < 0)
        return -1;
    myproc()->mask=mask;
    return 0;
}
```

它通过 `argint()` 函数获取了 `a0` 寄存器中的值并赋值给当前进程。

- 修改fork()

为了保证子进程与父进程的mask值相等，我在 `kernel/proc.c` 中修改了 `fork()` 函数，加上了下面一行：

```
// kernel/proc.c
np->mask=p->mask; //trace lab2
```

它实现了mask值的继承

- syscall.c修改

接下来是在 `syscall.c` 中添加相关代码

首先在 `syscall.h` 中添加相关宏定义

```
// kernel syscall.h
#define SYS_trace 22 //trace lab2
```

由于之前的系统调用代码占到了21，此处给trace分配22

同时引入刚刚编写的 `sys_trace` 函数

```
// kernel syscall.c
extern uint64 sys_trace(void); //trace lab2
```

于是可以在 `syscall` 数组中添加相关属性

```
// kernel syscall.c
[SYS_trace]    sys_trace,  //trace lab2
```

- 进行输出

由于每个系统调用都需要通过 `syscall` 函数，我们在此进行输出，添加以下代码：

```
// kernel/syscall.c
if( ((p->mask)&(1<<num))>0 ){
    printf( "%d: syscall %s -> %d\n" , p->pid ,
    sysname[num-1] ,                p->trapframe->a0 );
}
```

其中 `sysname` 是根据调用名定义的数组，直接添加在上方即可：

```
// kernel/syscall.c
char* sysname[22]={
    "fork",
    "exit",
    "wait",
    "pipe",
    "read",
    "kill",
    "exec",
    "fstat",
    "chdir",
    "dup",
    "getpid",
    "sbrk",
    "sleep",
    "uptime",
    "open",
    "write",
    "mknod",
    "unlink",
    "link",
    "mkdir",
    "close",
    "trace"
};
```

- 其他设置

根据官方hits, 我们需要进行一下修改让 `make qemu` 可以正常编译

1. 在 `user/user.h` 中的系统调用部分加入

```
// user/user.h
int trace(int);    //trace lab2
```

2. 在 `user/usys.pl` 中的系统调用部分加入

```
// user/usys.pl
entry("trace");    # trace lab2
```

3. 修改 `Makefile`

## 运行结果

```
== Test trace 32 grep == trace 32 grep: OK (1.1s)
== Test trace all grep == trace all grep: OK (1.0s)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (10.2s)
```

验证通过!

## 思考

- 在本次实验中首次尝试实现一个系统调用。除了熟悉了添加系统调用的流程以外, 还意识到了用户态和内核态的一些差异。在接下来的实验中将会频繁涉及这一部分。

## 2.2 sysinfo

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".



## 前置知识

- `kalloc.c`

```
// Physical memory allocator, for user processes,  
// kernel stacks, page-table pages,  
// and pipe buffers. Allocates whole 4096-byte pages.
```

该代码段是xv6中的物理内存分配器，用于分配用户进程、内核栈、页表页面和管道缓冲区的物理内存。它以整页（4096字节）为单位进行分配和释放。

`kalloc.c`中包含了一系列内存管理的方法：

- `freerange()`: 标记一段物理内存空间为可用状态，将其添加到空闲内存页块链表中。
- `kfree()`: 释放一个物理内存页块，将其添加到空闲内存页块链表中。
- `kalloc()`: 分配一个物理内存页块，并返回指向该页块的指针。

- 上锁

注意到在`kalloc.c`的实现中，经常有类似这样的代码出现：

```
// kfree()  
acquire(&kmem.lock);  
r->next = kmem.freelist;  
kmem.freelist = r;  
release(&kmem.lock);
```

其中的`acquire()`和`release()`操作显然是为了同步引入的机制。查阅相关资料可知，xv6使用了自旋锁(spinlock)的方法实现了进程的同步。自旋锁的实现需要硬件的支持，通过`acquire()`与`release()`实现。

自旋锁与理论课上学习的信号量机制有诸多相似，不过在xv6中的使用比较简单，在接下来对内存进行管理时一定记得上锁，以免读取脏数据。

- `copyout()`

由于虚拟内存机制的存在，将info读取到用户空间的操作需要专门的`copyout()`函数支持。

```
// Copy from kernel to user.
// Copy len bytes from src to virtual address dstva in a
// given page table.
// Return 0 on success, -1 on error.
int
copyout(pagetable_t pagetable, uint64 dstva, char *src,
uint64 len)
```

copyout函数接受四个参数，实现数据从内核到用户的传送，对其的具体分析将在之后说明

## 实现过程

- 为了实现sysinfo的函数，我需要两个辅助函数
  - freememory()

```
// sysinfo lab2
// 统计空闲空间
int
freememory(void)
{
    acquire(&kmem.lock);
    struct run* p = kmem.freelist;
    int num = 0;
    while (p)
    {
        num ++;
        p = p->next;
    }
    release(&kmem.lock);
    return num * PGSIZE;
}
```

使用 `freememory()` 函数统计空闲空间。由于xv6的空闲空间是按链表方式存放的，我只需要循链查找空闲空间的数量，乘以页面大小即可。该函数添加在 `kernel/kalloc.c` 中

- procnum()

```
// sysinfo lab2
// 统计进程数量
int
```

```

procnum(void)
{
    int i;
    int n = 0;
    for (i = 0; i < NPROC; i++)
    {
        acquire(&proc[i].lock); //上锁!
        if (proc[i].state != UNUSED) n++;
        release(&proc[i].lock); //解锁!
    }
    return n;
}

```

使用 `procnum()` 函数统计当前进程数。由于 xv6 的进程数是按数组方式存放的，直接遍历数组，将 `state != UNUSED` 的进程统计即可。

该函数添加在 `kernel/proc.c` 中

以上两个函数记得上锁！

- `sys_sysinfo()`

```

// sysinfo lab2
uint64
sys_sysinfo(void)
{
    struct sysinfo info;
    uint64 addr;

    struct proc* p = myproc();
    info.freemem = freememory();
    info.nproc = procnum();

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

```

有了上述两个函数之后，sysinfo的实现就非常简单了，直接添加在 `kernel/sysproc.c` 里就好。

- 最后仿照trace中修改相关引用即可通过编译。

## 运行结果

```
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /  
/^$/d' > kernel/kernel.sym
== Test sysinfotest == sysinfotest: OK (2.2s)
```

验证通过！

## 思考

- 通过copyout函数的使用，可以发现用户态与内核态地址是完全隔离的，将数据从内核地址空间复制到用户地址空间需要特定函数的支持。这样，用户程序可以从内核获取必要的的数据，而无需直接访问内核地址空间，保护了内核的安全性。

## 2.3 实验评分

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (1.5s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.9s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.1s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (9.9s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (1.1s)
== Test time ==
time: OK
Score: 35/35
```

# lab3: page tables

## 3.1 Speed up system calls

When each process is created, map one read-only page at USYSCALL (a VA defined in `memlayout.h`). At the start of this page, store a `struct usyscall` (also defined in `memlayout.h`), and initialize it to store the PID of the current process. For this lab, `ugetpid()` has been provided on the userspace side and will automatically use the USYSCALL mapping. You will receive full credit for this part of the lab if the `ugetpid` test case passes when running `pgtbltest`.

### 前置知识

- `mappages()`函数

```
// Create PTEs for virtual addresses starting at va that
// refer to
// physical addresses starting at pa. va and size might
// not
// be page-aligned. Returns 0 on success, -1 if walk()
// couldn't
// allocate a needed page-table page.
int
mappages(pagetable_t pagetable, uint64 va, uint64 size,
uint64 pa, int perm)
```

这个函数是用于在页表中建立虚拟地址和物理地址的映射关系。它接受一个页表`pagetable`、虚拟地址`va`、映射大小`size`、物理地址`pa`和权限标志`perm`。

- 首先将虚拟地址`va`和虚拟地址`va+size-1`分别进行页对齐，得到对应的虚拟地址`a`和最后一个映射的虚拟地址`last`。

```
a = PGROUNDDOWN(va);
last = PGROUNDDOWN(va + size - 1);
```

- 之后遍历虚拟地址范围，检查页表项是否已经存在映射（即是否被设置了`PTE_V`标志），若页表项未被映射则将物理地址`pa`转换为页表项格式，并将权限标志`perm`和`PTE_V`标志添加到页表项中，由此建立虚拟地址`a`到物理地址`pa`的映射。

```

for(;;){
    if((pte = walk(pagetable, a, 1)) == 0)
        return -1;
    if(*pte & PTE_V)
        panic("mappages: remap");
    *pte = PA2PTE(pa) | perm | PTE_V;
    if(a == last)
        break;
    a += PGSIZE;
    pa += PGSIZE;
}

```

## 实现过程

- 修改allocproc()

allocproc 函数的作用是从进程表(proc[NPROC])中分配和初始化一个新的进程。

在未修改前，allocproc()函数会为进程分配一个陷阱帧页面，我们只需要参照其过程为usyscall也分配一个页面就好

```

//proc.c

static struct proc*
allocproc(void)
{
    ...
    //speed up lab3
    //申请页面存放usyscall
    if((p->usyscall = (struct usyscall *)kalloc()) == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }
    ...
}

```

- 修改proc\_pagetable()

proc\_pagetable接受一个进程结构体指针p作为参数，并为该进程创建一个页表。与上述相同，我们参照对TRAPFRAME的处理对usyscall进行映射

```

//proc.c

```

```

pagetable_t
proc_pagetable(struct proc *p)
{
    ...
    //speed up lab3
    //申请PTE
    if(mappages(pagetable, USYSCALL, PGSIZE,
                (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
        uvmunmap(pagetable, TRAMPOLINE, 1, 0);
        uvmunmap(pagetable, TRAPFRAME, 1, 0);
        uvmfree(pagetable, 0);
        return 0;
    }
    ...
}

```

- 修改proc\_freepagetable()、freeproc()

最后，要在进程结束时释放相关内存。

首先在freeproc()中添加以下操作：

```

//proc.c
static void
freeproc(struct proc *p)
{
    ...
    //speed up lab3
    if(p->usyscall)
        kfree((void*)p->usyscall);
    p->usyscall = 0;
    ...
}

```

之后修改proc\_freepagetable()函数释放页表

```
//proc.c
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    //speed up lab3
    uvmunmap(pagetable, USYSCALL, 1, 0);
    uvmfree(pagetable, sz);
}
```

## 运行结果

```
== Test    pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
```

验证通过！

## 思考

- 只读操作应该很多都可以通过共享页面加速。通过加速getpid()操作，首先就就可以加速fork()、exit()、wait()等需要pid的系统调用。

## 3.2 Print a page table

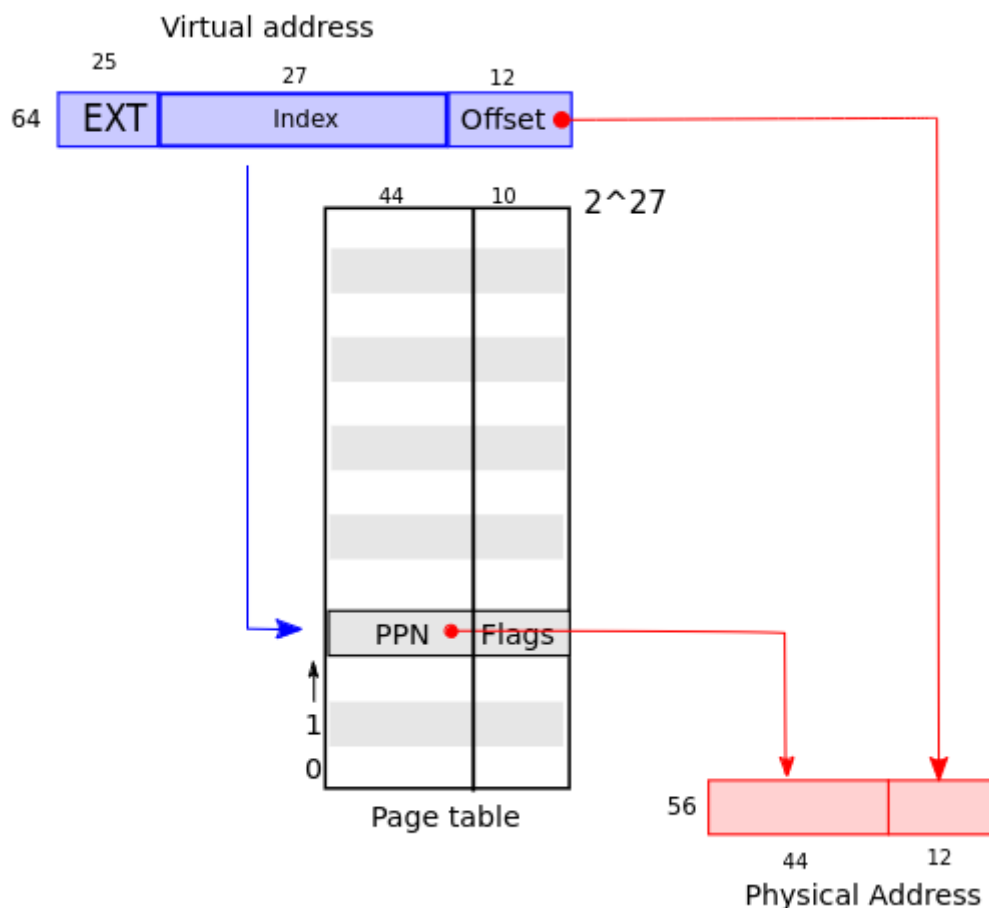
Define a function called `vmprint()`. It should take a `pagetable_t` argument, and print that pagetable in the format described below. Insert `if(p->pid==1) vmprint(p->pagetable)` in `exec.c` just before the `return argc`, to print the first process's page table. You receive full credit for this part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

## 前置知识

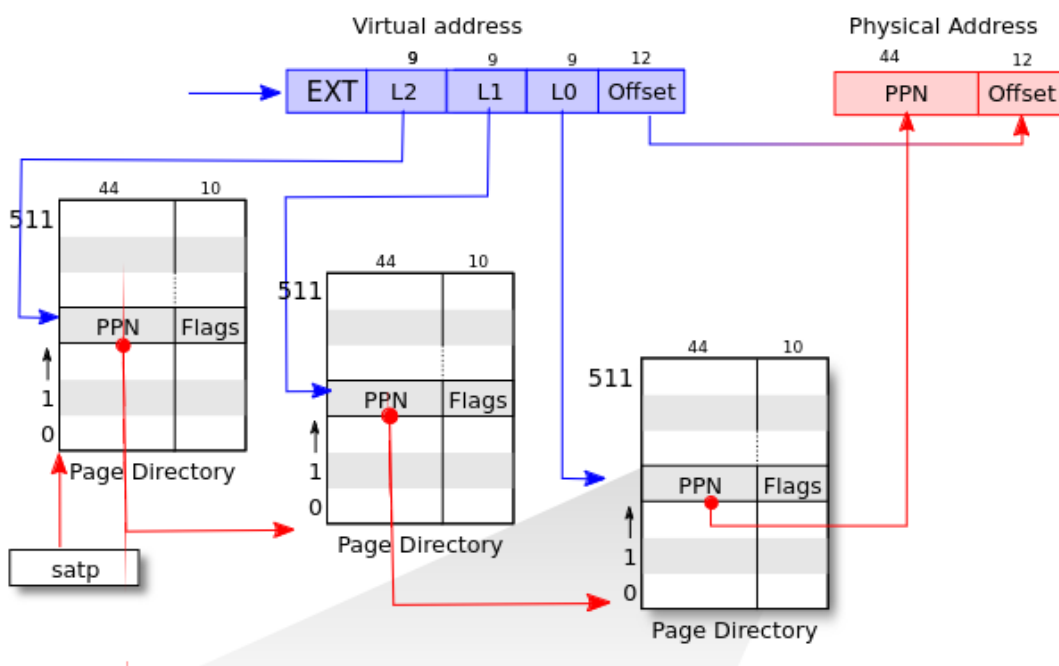
- xv6页表结构

xv6运行在Sv39 RISC-V架构上，其64位虚拟地址的底部39位被使用，而顶部25位没有使用，不用于地址转换。





如图所示，每个PTE包含一个44位的物理页号和10位标志位。页面硬件通过使用39位中的前27位来索引页表，找到一个PTE。页表让操作系统得以控制虚拟地址到物理地址的映射，以4,096字节对齐的块为粒度，这也是xv6系统的页面大小。



xv6使用的是三级页表结构，27位页号中的高9位为一级页表，中间9位为2级页表，末9位为三级页表。具体的地址转换过程如图所示。

- freewalk()函数

freewalk()函数是实现递归地释放页表页功能的函数，对我们打印页表结构的函数编写有很好的参照作用。

```
// Recursively free page-table pages.
// All leaf mappings must already have been removed.
void
freewalk(pagetable_t pagetable)
{
    // there are 2^9 = 512 PTEs in a page table.
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
            // this PTE points to a lower-level page table.
            uint64 child = PTE2PA(pte);
            freewalk((pagetable_t)child);
            pagetable[i] = 0;
        } else if(pte & PTE_V){
            panic("freewalk: leaf");
        }
    }
    kfree((void*)pagetable);
}
```

- 首先，代码通过一个for循环遍历页表的所有512个PTEs
- 对于每个PTE，代码首先检查它是否有效（PTE\_V位），以及它的读写执行权限（PTE\_R、PTE\_W、PTE\_X）是否都为0。如果两个条件都满足的话，说明该PTE指向一个更低级别的页表，于是递归地调用freewalk函数来释放这个更低级别的页表。

## 实现过程

- 编写vmprint()

仿照freewalk()函数，我们可以通过嵌套循环打印页表结构。

```
//vm.c

void vmprint(pagetable_t p)
{
    printf("page table %p\n" , p );
    for( int i=0 ; i<512 ; i++ ){
        //只有PTE存在且有效时候再输出
    }
}
```

```

//一级
if(!(p[i]&&PTE_V )){
    continue;
}
pagetable_t child1 = (pagetable_t)PTE2PA(p[i]);
printf( " ..%d: pte %p pa %p\n" , i , p[i] , child1 );
for(int j=0;j<512;j++){
    //二级
    if(!(child1[j]&&PTE_V )){
        continue;
    }
    pagetable_t child2 = (pagetable_t)PTE2PA(child1[j]);
    printf(" .. ..%d: pte %p pa %p\n" , j , child1[j] ,
child2 );
    for(int k=0;k<512;k++){
        //三级
        if(!(child2[k]&&PTE_V )){
            continue;
        }
        pagetable_t child3=(pagetable_t)PTE2PA(child2[k]);
        printf(" .. .. ..%d: pte %p pa %p\n" , k ,
child2[k] , child3 );
    }
}
}
}

```

结构比较简单不再赘述

- 根据hints修改exec函数

```

//exec.c

int
exec(char *path, char **argv)
{
    ...
    if(p->pid==1){
        vmprint(p->pagetable);
    }
    ...
}

```

## 运行结果

```
$ make qemu-gdb
pte printout: OK (0.3s)
```

验证通过!

## 思考

- 标志位的设置

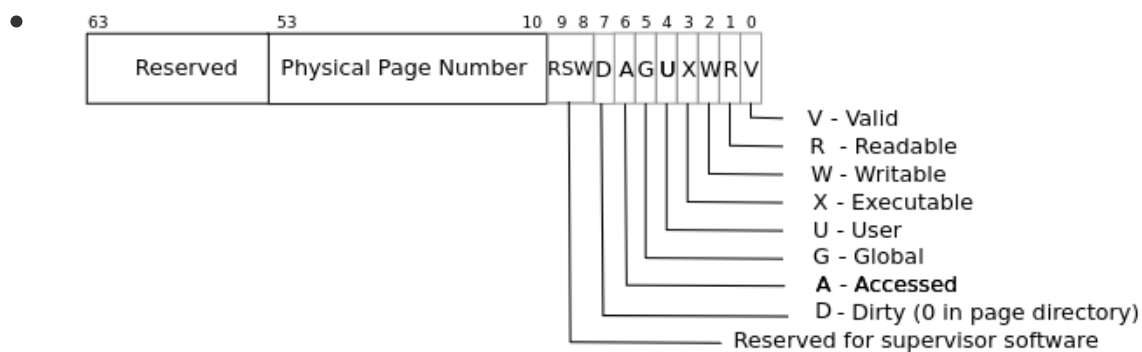
首先查看一下宏定义

```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_A (1L << 6) //detect lab3
```

1L << x表示将长整型整数1左移x位，例如1L << 4的结果为1000,之后每次进行与或操作时，例如

```
if((*pte & PTE_U) == 0)
```

实际上就是将PTE与1000每位做一个与操作，若结果不为0，说明PTE在低5位上为1,表明其拥有PTE\_U权限。



以上为xv6虚拟地址结构，可以发现在10位标志位中有3位被保留留作拓展，在之后的实验中将会涉及这部分。

## 3.3 Detecting which pages have been accessed

Your job is to implement `pgaccess()`, a system call that reports which pages have been accessed. The system call takes three arguments. First, it takes the starting virtual address of the first user page to check. Second, it takes the number of pages to check. Finally, it takes a user address to a buffer to store the results into a bitmask (a datastructure that uses one bit per page and where the first page corresponds to the least significant bit). You will receive full credit for this part of the lab if the `pgaccess` test case passes when running `pgtbltest`.

## 前置知识

- `walk()`函数

根据hints, 我们查看一下`walk()`函数

```
// Return the address of the PTE in page table pagetable
// that corresponds to virtual address va. If alloc!=0,
// create any required page-table pages.
```

`walk()`函数在页表`pagetable`中查找虚拟地址`va`对应的页表项（PTE）。如果`alloc`参数不为零, 则在查找过程中会自动创建缺失的页表页。

```
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
    if(va >= MAXVA)
        panic("walk");

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

通过for循环从level=2开始向下遍历页表。在每一级中，根据虚拟地址的相应位来确定PTE的索引，并获取对应的PTE指针。如果需要创建页表且创建成功，则将该页表清零，并设置PTE\_V位为1，表示该PTE指向有效的页表。同时更新pagetable为新创建的页表的物理地址，以便在下一级继续查找。遍历结束后，返回虚拟地址对应的最底层的PTE的指针，即指向物理地址的PTE。

## 实现过程

- PTE\_A

首先，设置一个PTE\_A标志位标记页面是否被访问

```
// riscv.h
#define PTE_A (1L << 6) //detect lab3
```

根据之前的xv6手册分析，PTE的低第6位是用于PTE\_A设置的。

- vm\_pgaccess()函数

```
//vm.c

//detect lab3
//用于取出PTE_A是否为1
int vm_pgaccess(pagetable_t pagetable_t, uint64 va)
{
    pte_t* pte;

    pte=walk(pagetable_t, va, 0);
    if(pte == 0)
        return 0;
    if((*pte & PTE_V) == 0)
        return 0;
    if((*pte & PTE_A) != 0){
        *pte=*pte&(~PTE_A);
        return 1;
    }

    return 0;
}
```

进行简单的错误判断，只要返回PTE\_A是否为1即可

- sys\_pgaccess()函数

最后完善一下sys\_pgaccess()函数

```
//sysproc.c

int
sys_pgaccess(void)
{
    // lab pgtbl: your code here.
    // detect lab3

    uint64 addr;
    int len;
    int mask;

    if(argaddr(0,&addr)<0)
        return -1;
    if(argint(1,&len)<0)
        return -1;
    if(argint(2,&mask)<0)
        return -1;

    int buf=0;
    struct proc *p=myproc();

    for (int i=0;i<len;i++){
        int va=addr+i*PGSIZE;
        //查看PTE_A是否为1
        int temp=vm_pgaccess(p->pagetable, va);
        buf=buf|temp<<i;
    }
    if(copyout(p->pagetable, mask, (char*)&buf, sizeof(buf))<0)
    {
        return -1;
    }

    return 0;
}
```

首先通过argaddr和argint从用户态获取参数addr、len和mask

然后，函数使用一个循环遍历从addr开始的len个虚拟地址（以页为单位）。对于每个虚拟地址，调用vm\_pgaccess函数来检查相应的PTE\_A位是否为1。将结果保存在buf变量中。

最后，将buf变量中的结果通过copyout函数复制到用户态的缓冲区mask中。

## 运行结果

```
== Test   pgtbltest: pgaccess ==  
pgtbltest: pgaccess: OK
```

验证通过！

## 思考

- 较为简单的lab，这里用到了之前提到的页表项的保留位，之后的是实验中也会涉及这个部分

## 3.4 实验评分

```
== Test pgtbltest ==  
$ make qemu-gdb  
(1.9s)  
== Test   pgtbltest: ugetpid ==  
pgtbltest: ugetpid: OK  
== Test   pgtbltest: pgaccess ==  
pgtbltest: pgaccess: OK  
== Test pte printout ==  
$ make qemu-gdb  
pte printout: OK (0.9s)  
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK  
== Test usertests ==  
$ make qemu-gdb  
(91.3s)  
== Test   usertests: all tests ==  
usertests: all tests: OK  
== Test time ==  
time: OK  
Score: 46/46
```



# lab4: traps

## 4.1 RISC-V assembly

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

- a0-a7寄存器保存了函数的参数，如果参数超过 8 个，额外的参数则被压入栈中。函数体内部可以通过访问这些寄存器来获取函数参数的值
- 在这段汇编中，13被存放在寄存器a2中

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

- 在给定的汇编代码中，对函数 `f` 和 `g` 的调用都是通过内联方式完成的。内联是一种编译器优化技术，它将函数调用处的函数体直接插入到调用点，而不是通过跳转到函数的地址执行函数体。这样可以避免函数调用的开销，提高代码执行效率。

3. At what address is the function `printf` located?

- 在汇编代码之中查找可知

```
void
printf(const char *fmt, ...)
628: 711d          addi sp, sp, -96
```

地址为0x628

4.2.4 What value is in the register `ra` just after the `jalr` to `printf` in `main`?

- 0x38，即返回地址

4.2.5 Run the following code.

What is the output?

- HE110 World

The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

- 将i左右颠倒即可，57616不用改变

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

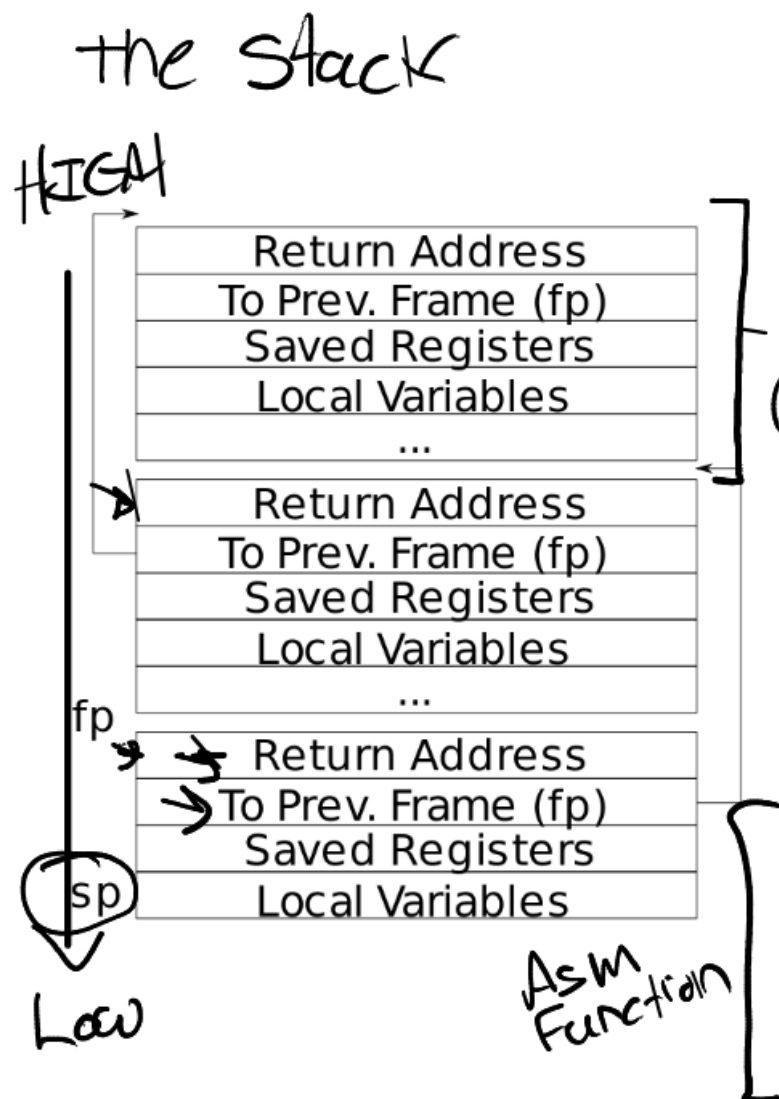
- 第二个参数存放在a2寄存器中，而由于没有第二个参数，所以直接输出a2寄存器中的值。

## 4.2 Backtrace

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. Implement a `backtrace()` function in `kernel/printf.c`. Insert a call to this function in `sys_sleep`,

### 前置知识

- 函数调用栈的结构



根据课程讲义，函数调用栈的结构如上所示。其中返回地址位于栈帧的帧指针的固定偏移量（-8）处，而保存的帧指针位于帧指针的固定偏移量（-16）处。

## 实现过程

- r\_fp()函数

根据hints, 我们在riscv.h中添加r\_fp()函数

```
//riscv.h

//backtrace lab4
static inline uint64
r_fp()
{
    uint64 x;
    asm volatile("mv %0, s0" : "=r" (x) );
    return x;
}
```

GCC编译器将当前执行的函数的帧指针存储在寄存器s0中, r\_fp()函数可以内联读出s0的值。

- backtrace()函数

接下来在printf.c中实现backtrace函数

根据函数调用栈的结构, 我们只需要循环读出当前函数的上一个帧指针地址并输出即可

```
//printf.c

//backtrace lab4
void
backtrace(void)
{
    printf("backtrace:\n");

    uint64 addr=r_fp();
    uint64 target=PGROUNDUP(addr);

    //保证地址在一个有效的栈页面内
    while(addr<target){
        uint64 tempAddr=*(uint64 *) (addr - 8);
        printf("%p\n", tempAddr);
        //读取保存的上一个帧指针地址
        addr=*((uint64 *) (addr - 16));
    }
}
```

```
}
```

- 最后根据提示，在`sys_sleep()`中进行调用即可

```
//sysproc.c

uint64
sys_sleep(void)
{
    backtrace();
    ...
}
```

## 运行结果

```
$ make qemu-gdb
backtrace test: OK (2.1s)
```

验证通过！

## 思考

- hints中提到通过`PGROUNDDOWN`、`PGROUNDUP`判断终止条件，不过其实由于函数调用过程中，帧指针会逐步向低地址移动，我直接使用`addr<target`也可以奏效
- `backtrace`是一个非常有用的debug功能，在错误排查时，可以通过它得到函数调用关系、递归函数使用情况等信息。

## 4.3 Alarm

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`.

## 前置知识

- `p->trapframe->epc`

在 RISC-V 架构中，`epc`寄存器用于保存程序在发生异常时，导致异常的指令的地址从而可以进行适当的处理或调试。

在本次alarm的实现中，通过设置 `epc`寄存器的值，使得当从异常处理函数返回时，继续执行导致中断的指令。

## 实现过程

- 首先根据实验提示添加如下字段

```
//user.h

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

对系统调用的注册不再赘述

之后，在进程字段中添加一些字段

```
//proc.h

struct proc {
    ...
    int ticks;           //时间间隔
    uint64 handler;      //处理函数地址
    int isHandlering;    //是否正在处理函数
    int tickNum;         //目前经历的时间间隔
    ...
}
```

并在allocproc中对其初始化

```
//proc.c

static struct proc*
allocproc(void)
{
    ...

found:
```

```

// alarm lab4
p->ticks=0;
p->tickNum=0;
p->isHandling=0;
...
}

```

- 上下文的保存

为了在返回用户空间的时候恢复上下文，我设置了两个函数

```

//proc.h

// alarm lab4
void save(struct proc* p);
void load(struct proc* p);

```

首先在进程中添加一些变量用于保存现场

```

//proc.h

struct proc {
    ...
    uint64 temp_epc;
    uint64 temp_ra;
    uint64 temp_sp;
    uint64 temp_gp;
    uint64 temp_tp;
    uint64 temp_t0;
    ...
    uint64 temp_t6;
};

```

save和load函数的实现非常简单，只需要保存寄存器的值、进行读出即可

```

//proc.c

void
save(struct proc* p)
{
    p->temp_epc = p->trapframe->epc;
    ...
}

```

```

    p->temp_t6 = p->trapframe->t6;
}

void
load(struct proc* p)
{
    p->trapframe->epc = p->temp_epc;
    ...
    p->trapframe->t6 = p->temp_t6;
}

```

- 系统调用的实现

- sys\_sigalarm

从寄存器中读出参数并赋值给进程即可

```

//sysproc.c

uint64
sys_sigalarm(void)
{
    //参数：报警间隔、指向处理程序函数的指针
    int ticks;
    uint64 handler;
    struct proc *p=myproc();

    if(argint(0, &ticks) < 0)
        return -1;
    if(argaddr(1, &handler) < 0)
        return -1;

    p->ticks=ticks;
    p->handler=handler;
    p->tickNum=0;

    return 0;
}

```

- sys\_sigreturn

调用load函数恢复进程上下文即可。包含一个isHandling置0操作，对其置1在trap.c中进行

```
//sysproc.c

uint64
sys_sigreturn(void)
{
    struct proc *p=myproc();
    load(p);
    p->isHandlering=0;
    return 0;
}
```

- 修改trap.c

根据hints, xv6的时钟中断对应了 `(which_dev == 2)`, 于是在trap.c中进行如下修改

```
//trap.c

void
usertrap(void)
{
    ...
    if(which_dev == 2){
        p->tickNum++;
        if(p->tickNum>=p->ticks&& p->ticks>0&&p-
>isHandlering==0){
            p->tickNum=0;
            save(p);
            p->isHandlering=1;
            p->trapframe->epc=p->handler;
        }

        yield();
    }
    ...
}
```

当时钟中断发生时, 修改计数, 并根据计数值判断是否进行handler



## 运行结果

```
== Test alarmtest: test0 ==  
alarmtest: test0: OK  
== Test alarmtest: test1 ==  
alarmtest: test1: OK  
== Test alarmtest: test2 ==  
alarmtest: test2: OK
```

验证通过！

## 思考

- alarm函数可以帮助返回一个超时错误，从而帮助进行错误处理。另外，alarm函数还可以帮助配置定时计划任务，免去了唤醒定时任务的麻烦。
- 理论课上提到过中断时需要实现一个保存上下文的行为，当时认为这是一个十分高级复杂的操作，在本次实验中，其实只需要将对应的寄存器值记录即可。不过这也与xv6的精简有关。

## 4.4 实验评分

```
== Test answers-traps.txt == answers-traps.txt: OK  
== Test backtrace test == backtrace test: OK (1.2s)  
== Test running alarmtest == (3.6s)  
== Test alarmtest: test0 ==  
alarmtest: test0: OK  
== Test alarmtest: test1 ==  
alarmtest: test1: OK  
== Test alarmtest: test2 ==  
alarmtest: test2: OK  
== Test usertests == usertests: OK (135.6s)  
== Test time ==  
time: OK  
Score: 85/85
```

## lab5: Copy-on-Write Fork for xv6

### 5.1 相关学习

- problem

xv6中的fork()系统调用将父进程的所有用户空间内存复制到子进程中。如果父进程较大，则复制可能需要很长时间。更糟糕的是，这项工作经常造成大量浪费；例如，子进程中的fork()后跟exec()将导致子进程丢弃复制的内存，而其中的大部分可能都从未使用过。另一方面，如果父子进程都

使用一个页面，并且其中一个或两个对该页面有写操作，则确实需要复制。

- **solution**

copy-on-write (COW) `fork()`的目标是推迟到子进程实际需要物理内存拷贝时再进行分配和复制物理内存页面。

COW `fork()`只为子进程创建一个页表，用户内存的PTE指向父进程的物理页。COW `fork()`将父进程和子进程中的所有用户PTE标记为不可写。当任一进程试图写入其中一个COW页时，CPU将强制产生页面错误。内核页面错误处理程序检测到这种情况将为出错进程分配一页物理内存，将原始页复制到新页中，并修改出错进程中的相关PTE指向新的页面，将PTE标记为可写。当页面错误处理程序返回时，用户进程将能够写入其页面副本。

COW `fork()`将使得释放用户内存的物理页面变得更加棘手。给定的物理页可能会被多个进程的页表引用，并且只有在最后一个引用消失时才应该被释放。

## 5.2 Implement copy-on write

Your task is to implement copy-on-write `fork` in the xv6 kernel. You are done if your modified kernel executes both the `cowtest` and `usertests` programs successfully.

- 修改uvmcopy()

根据hints，我们首先设置PTE第9位为是否会被cow的标记位

```
//riscv.h

#define PTE_C (1L << 8)
```

回到uvmcopy中，由于分配策略的改变，我们不应当为子进程分配实际的地址空间，在页表中只需要让其与父进程映射到同一地址空间即可。

```
//vm.c

int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;

    for(i = 0; i < sz; i += PGSIZE){
```

```

    if((pte = walk(old, i, 0)) == 0)
        panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
        panic("uvmcopy: page not present");
    // cow lab5
    pa = PTE2PA(*pte);
    // 清除写权限
    *pte=*pte & ~(PTE_W);
    // 可能cow位设为1
    *pte=*pte | PTE_C;
    flags = PTE_FLAGS(*pte);

    if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
        goto err;
    }
    refIncr((void *)pa);
}
return 0;

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

```

另外，对页表项还应进行部分修改：将写权限回收（防止父子进程修改后造成的数据不一致），将PTE\_C置为1

refIncr()是增加引用计数的函数，在之后会提及

- 修改kalloc.c
  - 首先，修改一下kmem的结构

```

//kalloc.c

struct {
    struct spinlock lock;
    struct run *freelist;
    int pageNum;
    char *refPage;
    char *cEnd;
} kmem;

```

- pageNum: 记录一共有多少个页面，在引用计数时使用

- refPage: 记录页面引用计数的数组
- cEnd: refPage数组的尾部
- 同理kinit也要进行修改

```
//kalloc.c

void
kinit()
{
    initlock(&kmem.lock, "kmem");
    // cow lab5
    kmem.pageNum=pageCountCal(end, (void*)PHYSTOP);
    kmem.refPage=end;
    for(int i =0;i<kmem.pageNum;i++){
        kmem.refPage[i]=0;
    }
    kmem.cEnd=kmem.refPage+kmem.pageNum;

    freerange(kmem.cEnd, (void*)PHYSTOP);
}
```

对页面的引用计数需要初始化

- 辅助函数的添加

- getIndex()

用于返回给定的地址在用户空间的序号

```
//kalloc.c

int
getIndex(void *pa)
{
    uint64 upa=(uint64)pa;
    upa=PGROUNDOWN(upa);

    int res=(upa-(uint64)kmem.cEnd)/PGSIZE;
    if(res<0||res>=kmem.pageNum){
        panic("index illegal");
    }
    return res;
}
```

由于上文中在end后手动添加了一个数组，这里的定位需要靠到 `kmem.cEnd` 的偏移量来确定。

- `refIncr()`、`refDecr()`

增加、减少引用计数

```
//kalloc.c

//增加引用
void
refIncr(void* pa)
{
    int index=getIndex(pa);

    acquire(&kmem.lock);
    kmem.refPage[index]++;
    release(&kmem.lock);
}

//减少引用
void
refDecr(void* pa)
{
    int index=getIndex(pa);

    acquire(&kmem.lock);
    kmem.refPage[index]--;
    release(&kmem.lock);
}
```

- 修改`kalloc()`

在链表中获得地址之后，应当增加对其的引用

```
//kalloc.c

void *
kalloc(void)
{
    //cow lab5

    struct run *r;
```

```

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

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

    return (void*)r;
}

```

- 修改kfree()

只有当页面的计数为0时才真正释放此页面，否则只需要减少其引用即可

```

//kalloc.c

void
kfree(void *pa)
{
    int index=getIndex(pa);
    if(kmem.refPage[index]>=1){
        refDecr(pa);
        if(kmem.refPage[index]>0)
            return;
    }
    ...
}

```

- 继续修改vm.c

- needCow

当触发page fault时，应当有函数辅助判断时候是由于cow策略导致的

```

//vm.c

// cow lab5
// 判断是否为cow冲突

```

```

int
needCow(pagetable_t p, uint64 va)
{
    if(va >= MAXVA){
        return 0;
    }
    va = PGROUNDDOWN(va);
    pte_t *pte = walk(p, va, 0);

    if(pte == 0)
        return 0;
    if((*pte & PTE_V) == 0)
        return 0;
    if((*pte & PTE_U) == 0)
        return 0;
    if(*pte & PTE_C)
        return 1;

    return 0;
}

```

当PTE有效、用户具有权限且PTE\_C=1时，说明确实是cow冲突，返回1即可

- cowAlloc

若为cow冲突，还需要函数解决冲突，分配实际页面

```

//vm.c

// cow lab5
// 进行cow申请
int
cowAlloc(pagetable_t p, uint64 va)
{
    va = PGROUNDDOWN(va);
    pte_t *pte = walk(p, va, 0);
    uint64 pa = PTE2PA(*pte);
    int flag = PTE_FLAGS(*pte);

    // 解除映射
    char* mem = kalloc();
    if(mem == 0)
        return -1;
}

```

```

memmove(mem, (char*)pa, PGSIZE);
uvmunmap(p, va, 1, 1);

// 修改标志位
flag=flag & ~(PTE_C);
flag=flag | PTE_W;

if(mappages(p, va, PGSIZE, (uint64)mem, flag)<0){
    kfree(mem);
    return -1;
}

return 0;
}

```

- 之后，只需要在trap.c中添加相关处理即可

```

//trap.c

void
usertrap(void)
{
    ...
    else if(r_scause()==13 || r_scause()==15){
        //cow lab5
        uint64 va=r_stval();
        if(needCow(p->pagetable, va)==1){
            if(cowAlloc(p->pagetable, va)<0){
                printf("cow alloc failed!");
                p->killed=1;
            }
        }else{
            printf("usertrap(): unexpected scause!!!!!!!!!!!! %p
pid=%d\n", r_scause(), p->pid);
            printf("                sepc=%p stval=%p\n", r_sepc(),
r_stval());
            p->killed = 1;
        }
    }
    ...
}

```



## 5.3 运行结果

```
== Test running cowtest ==
$ make qemu-gdb
(4.8s)
== Test    simple ==
    simple: OK
== Test    three ==
    three: OK
== Test    file ==
    file: OK
== Test usertests ==
$ make qemu-gdb
(90.7s)
== Test    usertests: copyin ==
    usertests: copyin: OK
== Test    usertests: copyout ==
    usertests: copyout: OK
== Test    usertests: all tests ==
    usertests: all tests: OK
== Test time ==
    time: OK
Score: 110/110
```

## 5.4 实验小结

- 比较痛苦的一个实验，大多数错误是逻辑上的，另外还有两个错误导致卡了很久
  - 没有判断mem==0的错误
  - PTE项获取较晚导致获得了错误的值
- 按需分配的思想在计算机的设计中真的无处不在，比如cache、快表、动态装入等，这次的cow也是这种思想，这种优化策略可以很好的节约资源。

## lab6: Multithreading

### 6.1 Uthread: switching between threads

Your job is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan. When you're done, `make grade` should say that your solution passes the `uthread` test.

## 前置知识

- callee-save registers

如果把所有的寄存器都压栈和出栈，首先会导致性能下降，另一方面，也占用了内存，当函数调用栈很深或者出现递归的时候，就会更加明显。因此，把寄存器分为callee-save registers和caller-save registers。

Caller-saved register用于保存不需要在各个调用之间保留的临时数据。

Callee-saved register用于保存应在每次调用中保留的长寿命值。因此，本实验中只需要保存Callee-saved register即可。

## 实现过程

- 定义上下文

```
//uthread.c

struct context{
    uint64 temp_ra;
    uint64 temp_sp;
    uint64 temp_s0;
    uint64 temp_s1;
    uint64 temp_s2;
    uint64 temp_s3;
    uint64 temp_s4;
    uint64 temp_s5;
    uint64 temp_s6;
    uint64 temp_s7;
    uint64 temp_s8;
    uint64 temp_s9;
    uint64 temp_s10;
    uint64 temp_s11;
};
```

之后在线程中添加上下文字段

```
//uthread.c

struct thread {
    char    stack[STACK_SIZE]; /* the thread's stack */
    int     state;              /* FREE, RUNNING, RUNNABLE
*/
    // switch lab6
    struct context context;
};
```

- 修改thread\_create

在线程创建时，ra和sp初始化

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

    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        if (t->state == FREE) break;
    }
    t->state = RUNNABLE;
    // YOUR CODE HERE
    // switch lab6
    t->context.temp_ra=(uint64)func;
    t->context.temp_sp=(uint64)&t->stack[STACK_SIZE-1];
}
```

- 修改uthread\_swutch.S

直接照抄kernel/swtch.S即可

```
//uthread_swutch.S

thread_switch:
    /* YOUR CODE HERE */
    /* switch lab6 */
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    ...
    sd s11, 104(a0)
```

```
ld ra, 0(a1)
ld sp, 8(a1)
ld s0, 16(a1)
...
ld s11, 104(a1)
```

- 修改thread\_schedule

最后在thread\_schedule中切换上下文即可

```
//uthread.c

void
thread_schedule(void)
{
    ...
    if (current_thread != next_thread) {
        ...
        /* YOUR CODE HERE
         * Invoke thread_switch to switch from t to
         next_thread:
         * thread_switch(??, ??);
         */
        thread_switch((uint64)&t->context,
            (uint64)&next_thread->context);
    } else
        next_thread = 0;
}
```

运行结果

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (1.9s)
```

验证通过!

## 6.2 Using threads

In this assignment you will explore parallel programming with threads and locks using a hash table. You should do this assignment on a real Linux or MacOS computer (not xv6, not qemu) that has multiple cores. Most recent laptops have multicore processors.

## 前置知识

- 线程锁的使用（pthread\_mutex）

实验提示中有线程锁的一些使用方式

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

## 实现过程

- 锁的声明

声明一个全局锁

```
//ph.c

pthread_mutex_t lock[NBUCKET];
```

在main中对其初始化

```
//ph.c

int
main(int argc, char *argv[])
{
    ...
    for(int i=0;i<NBUCKET;i++){
        pthread_mutex_init(&lock[i],NULL);
    }
    ...
}
```

- 修改put

在put函数中上锁即可

```
//ph.c

void put(int key, int value)
{
    ...
    // 上锁
```

```

pthread_mutex_lock(&lock[i]);
if(e){
    ...
}
// 解锁
pthread_mutex_unlock(&lock[i]);

}

```

## 运行结果

```

== Test ph_safe == make[1]: 进入目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
ph_safe: OK (8.4s)
== Test ph_fast == make[1]: 进入目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
ph_fast: OK (19.6s)

```

验证通过！

## 6.3 Barrier

In this assignment you'll implement a barrier: a point in an application at which all participating threads must wait until all other participating threads reach that point too. You'll use pthread condition variables, which are a sequence coordination technique similar to xv6's sleep and wakeup.

### 前置知识

- Barrier主要是用作集合线程，然后再一起往下执行。在Barrier之前，若干个thread各自执行，在Barrier处停下，等待规定数目的所有的其他线程到达这个Barrier，再一起通过这个Barrier。

### 实现过程

- 只需要在barrier中进行上锁即可

```

//barrier.c

static void
barrier()

```

```

{
    // YOUR CODE HERE
    //
    // Block until all threads have called barrier() and
    // then increment bstate.round.
    //

    // barrier lab6
    pthread_mutex_lock(&bstate.barrier_mutex);
    bstate.nthread++;
    if(bstate.nthread==nthread){
        // 进入下一个barrier
        bstate.nthread=0;
        bstate.round++;
        pthread_cond_broadcast(&bstate.barrier_cond);
    }else{
        // 等待线程全部到达

        pthread_cond_wait(&bstate.barrier_cond,&bstate.barrier_mu
tex);
    }
    pthread_mutex_unlock(&bstate.barrier_mutex);
}

```

## 运行结果

```

== Test barrier == make[1]: 进入目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-
2021-threads"
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
barrier: OK (2.5s)

```

验证通过！

## 思考

- 同样比较简单，只要通过上锁保证下一个栅栏的操作不会影响到上一个还未结束的栅栏即可。

## 6.4 实验评分

```

uthread: OK (2.1s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
ph_safe: OK (8.4s)
== Test ph_fast == make[1]: 进入目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
ph_fast: OK (19.5s)
== Test barrier == make[1]: 进入目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录"/home/marco/桌面/2150253-xv6-labs/xv6-labs-2021-threads"
barrier: OK (2.5s)
== Test time ==
time: OK
Score: 60/60

```

## lab7: networking

Your job is to complete `e1000_transmit()` and `e1000_recv()`, both in `kernel/e1000.c`, so that the driver can transmit and receive packets. You are done when `make grade` says your solution passes all the tests.

### 7.1 Implement networking

- `e1000_transmit`

这是实现数据包发送功能的函数

`e1000.c`

```

int
e1000_transmit(struct mbuf *m)
{
    //
    // Your code here.
    //
    // the mbuf contains an ethernet frame; program it into
    // the TX descriptor ring so that the e1000 sends it.
    Stash
    // a pointer so that it can be freed after sending.
    //

    acquire(&e1000_lock);

    int index = regs[E1000_TDT];

```



```

if ((tx_ring[index].status & E1000_TXD_STAT_DD) == 0) {
    //上一个传输尚未完成
    release(&e1000_lock);
    return -1;
}

//释放内存
if (tx_mbufs[index])
    mbuffree(tx_mbufs[index]);

tx_mbufs[index] = m;
tx_ring[index].length = m->len;
tx_ring[index].addr = (uint64) m->head;
tx_ring[index].cmd = E1000_TXD_CMD_RS |
E1000_TXD_CMD_EOP;

//更新一下传输描述符
regs[E1000_TDT] = (index + 1) % TX_RING_SIZE;

release(&e1000_lock);

return 0;
}

```

- 首先对e1000上锁，保证对设备的互斥使用
- 获取一下传输描述符，并检查上一个传输是否已经结束，如果结束的话释放一下内存
- 将当前 mbuf 的指针存储在传输缓冲区，以便在发送完成后可以释放对应的内存
- 更新tx\_ring数组的对应字段
- 更新传输描述符环中下一个数据包的索引regs[E1000\_TDT]
- e1000\_recv

```

//e1000.c

static void
e1000_recv(void)
{
    //

```

```

// Your code here.
//
// Check for packets that have arrived from the e1000
// Create and deliver an mbuf for each packet (using
net_rx()).
//

while (1) {
    int index = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    if ((rx_ring[index].status & E1000_RXD_STAT_DD) == 0)
    {
        //全部接收完毕
        return;
    }
    rx_mbufs[index]->len = rx_ring[index].length;
    net_rx(rx_mbufs[index]);
    //分配一个新的mbuf
    rx_mbufs[index] = mbufalloc(0);
    rx_ring[index].status = 0;
    rx_ring[index].addr = (uint64)rx_mbufs[index]->head;

    //更新一下接收描述符
    regs[E1000_RDT] = index;
}
}

```

- 首先将所有操作包在循环体中，用于接受全部的数据包
- 计算一下下一个要处理的接收描述符的索引位置
- 检查接收描述符的状态，如果状态中的 E1000\_RXD\_STAT\_DD位为0，表示没有新的数据包可用，此时退出循环。
- 当有数据包可用时，更新rx\_mbufs，调用net\_rx函数传递mbuf到网络堆栈
- 分配一个新的mbuf，更新rx\_ring的字段便于下一个数据包的处理

## 7.2 运行结果

```
== Test    nettest: ping ==  
    nettest: ping: OK  
== Test    nettest: single process ==  
    nettest: single process: OK  
== Test    nettest: multi-process ==  
    nettest: multi-process: OK  
== Test    nettest: DNS ==  
    nettest: DNS: OK  
== Test time ==  
time: OK  
Score: 100/100
```

## lab8: locks

### 8.1 Memory allocator

Your job is to implement per-CPU freelists, and stealing when a CPU's free list is empty. You must give all of your locks names that start with "kmem". That is, you should call `initlock` for each of your locks, and pass a name that starts with "kmem". Run `kalloctest` to see if your implementation has reduced lock contention. To check that it can still allocate all of memory, run `usertests sbrkmuch`. Your output will look similar to that shown below, with much-reduced contention in total on kmem locks, although the specific numbers will differ. Make sure all tests in `usertests` pass. `make grade` should say that the `kalloctests` pass.

#### 前置知识

- 开/关中断

hints中提到了开关中断的问题，对其的描述在xv6 book的6.6节

当自旋锁用于保护同时被线程和中断处理程序使用的数据时会有一种潜在的风险：如果一个线程在持有自旋锁的同时被中断了，而中断处理程序也尝试获取同样的自旋锁，那么就可能出现死锁的情况。

针对这种情况，xv6采取的措施是，当一个CPU获取任何自旋锁时，它会禁用该CPU上的中断。查看`spinlock.c`中开关中断的指令：

```
//spinlock.c

void push_off(void)
void pop_off(void)
```

## 实现过程

- 首先修改一下kmem的结构

```
//kalloc.c

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

为每个CPU分配一个内存链表，因此将kmem修改为数组

- 对应的，修改kinit和kfree

```
//kalloc.c

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

kinit循环初始化数组即可

```
//kalloc.c

void
kfree(void *pa)
{
    struct run *r;

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

```

// Fill with junk to catch dangling refs.
memset(pa, 1, PGSIZE);

r = (struct run*)pa;

push_off(); // 关中断
int tempCPU=cuid();
acquire(&kmem[tempCPU].lock);
r->next = kmem[tempCPU].freelist;
kmem[tempCPU].freelist = r;
release(&kmem[tempCPU].lock);
pop_off(); // 开中断
}

```

kfree同理，不过注意CPU相关操作要关中断

- 修改kalloc

相比之前只需要添加一个操作，当链表长度不足时从其他CPU那里抢夺。

```

//kalloc.c

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

    // lock lab8
    push_off();
    int tempCPU=cuid();
    acquire(&kmem[tempCPU].lock);

    r = kmem[tempCPU].freelist;
    if(r){
        kmem[tempCPU].freelist = r->next;
    }else{
        // 空闲空间不足，抢夺其他cpu空间
        int id;
        for(id=0;id<NCPU;id++){
            // 跳过当前cpu编号
            if(id==tempCPU)continue;
            acquire(&kmem[id].lock);
            r = kmem[id].freelist;

```

```

        if(r) {
            kmem[id].freelist = r->next;
            release(&kmem[id].lock);
            break;
        }
        release(&kmem[id].lock);

    }
}
release(&kmem[tempCPU].lock);
pop_off();

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

```

## 运行结果

```

== Test   kallocetest: test1 ==
      kallocetest: test1: OK
== Test   kallocetest: test2 ==
      kallocetest: test2: OK
== Test kallocetest: sbrkmuch ==
$ make qemu-gdb
kallocetest: sbrkmuch: OK (7.5s)

```

验证通过！

## 思考

- 本实验较为简单，注意CPU相关操作关闭中断即可

## 8.2 Buffer cache

Modify the block cache so that the number of `acquire` loop iterations for all locks in the bcache is close to zero when running `bcachetest`. Ideally the sum of the counts for all locks involved in the block cache should be zero, but it's OK if the sum is less than 500. Modify `bget`

and `brelse` so that concurrent lookups and releases for different blocks that are in the bcache are unlikely to conflict on locks (e.g., don't all have to wait for `bcache.lock`). You must maintain the invariant that at most one copy of each block is cached. When you are done, your output should be similar to that shown below (though not identical). Make sure `usertests` still passes. `make grade` should pass all tests when you are done.

## 前置知识

- buf结构

## 实现过程

- 修改buf.h

根据hints, 先修改桶数目为素数, 这里直接按hints中的13了

NULL的定义出于个人习惯

```
//buf.h

#define NBUC 13
#define NULL 0
```

另外, 还需要在buf中添加tick字段, 后面LRU算法时会使用

```
//buf.h

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

- 桶和bcache的结构定义

首先是桶, 将其定义为数组

```
//bio.c

struct {
    struct spinlock lock;
    struct buf head;
} bbuc[NBUC];
```

对于bcache, 去除其头部

```
//bio.c

struct {
    struct spinlock lock;
    struct buf buf[NBUF];
} bcache;
```

- 修改binit

初始化函数主要分两步：初始化桶、初始化buffer

将buffer->tick置为-1, 表明其未被使用

```
//bio.c

void
binit(void)
{
    struct buf *b;

    initlock(&bcache.lock, "bcache");

    for(int i = 0; i < NBUC; i++){
        // 初始化桶
        initlock(&bbuc[i].lock, "bbuc");
        bbuc[i].head.prev = &bbuc[i].head;
        bbuc[i].head.next = &bbuc[i].head;
    }

    // Create linked list of buffers
    for(b = bcache.buf; b < bcache.buf+NBUF; b++){
        b->tick = -1;
        initsleeplock(&b->lock, "buffer");
    }
}
```



- bget

这位更是重量级

这是本次实验的主体部分，作用是从缓冲池（buffer cache）中查找一个数据块，需要时会分配一个缓冲区。

这里先上代码

```
//bio.c

extern uint ticks;

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

    int tempBuc = blockno%NBUC;
    acquire(&bbuc[tempBuc].lock);

    // 是否cached
    for(b = bbuc[tempBuc].head.next; b !=
&bbuc[tempBuc].head; b = b->next){
        if(b->dev == dev && b->blockno == blockno){
            b->tick = ticks;
            b->refcnt++;
            release(&bbuc[tempBuc].lock);
            acquiresleep(&b->lock);
            return b;
        }
    }

    // 否则根据lru原则选择桶存放

    // 首先在本桶内查找
    struct buf *tempBuf= NULL;
    for(b = bbuc[tempBuc].head.next; b !=
&bbuc[tempBuc].head; b = b->next){
        if(b->refcnt == 0){
            if(tempBuf==NULL || tempBuf->tick>b->tick){
                tempBuf=b;
            }
        }
    }
}
```

```

    }
}
if(tempBuf!=NULL){
    tempBuf->dev = dev;
    tempBuf->blockno = blockno;
    tempBuf->valid = 0;
    tempBuf->refcnt = 1;
    tempBuf->tick = ticks;
    release(&bbuc[tempBuc].lock);
    acquiresleep(&tempBuf->lock);
    return tempBuf;
}

// 否则在所有缓冲区查找
int targetBuc;
int flag = 1;
while (flag){
    flag = 0;
    tempBuf = 0;
    for (b = bcache.buf; b < bcache.buf+NBUF; b++){
        if (b->refcnt == 0){
            flag = 1;
            if(tempBuf==NULL || tempBuf->tick>b->tick){
                tempBuf=b;
            }
        }
    }
}
if (tempBuf!=NULL){
    if (tempBuf->tick == -1){
        acquire(&bcache.lock);
        if (tempBuf->refcnt == 0){
            // 正好没有引用
            tempBuf->dev = dev;
            tempBuf->blockno = blockno;
            tempBuf->valid = 0;
            tempBuf->refcnt = 1;
            tempBuf->tick = ticks;

            // 加入桶队列
            tempBuf->next = bbuc[tempBuc].head.next;
            tempBuf->prev = &bbuc[tempBuc].head;
            bbuc[tempBuc].head.next->prev = tempBuf;
            bbuc[tempBuc].head.next = tempBuf;

```

```

        release(&bcache.lock);
        release(&bbuc[tempBuc].lock);
        acquiresleep(&tempBuf->lock);
        return tempBuf;
    } else {
        // 有引用
        release(&bcache.lock);
    }
} else {
    targetBuc = tempBuf->blockno%NBUC;
    acquire(&bbuc[targetBuc].lock);
    if (tempBuf->refcnt == 0){
        tempBuf->dev = dev;
        tempBuf->blockno = blockno;
        tempBuf->valid = 0;
        tempBuf->refcnt = 1;
        tempBuf->tick = ticks;

        tempBuf->next->prev = tempBuf->prev;
        tempBuf->prev->next = tempBuf->next;
        tempBuf->next = bbuc[targetBuc].head.next;
        tempBuf->prev = &bbuc[targetBuc].head;
        bbuc[targetBuc].head.next->prev = tempBuf;
        bbuc[targetBuc].head.next = tempBuf;

        release(&bbuc[targetBuc].lock);
        release(&bbuc[tempBuc].lock);
        acquiresleep(&tempBuf->lock);
        return tempBuf;
    } else {
        release(&bbuc[targetBuc].lock);
    }
}
}
}

panic("bget: no buffers");
}

```

- 首先，通过给定磁盘块号计算确定要查找的缓冲桶（tempBuc）。

- 在指定缓冲桶内查找是否已经存在相应的缓冲区。如果找到匹配的缓冲区，表示数据已经在缓冲区中，那么更新缓冲区的相关信息，返回该缓冲区。
  - 如果在指定缓冲桶内没有找到相应的缓冲区，那么首先，在当前缓冲桶内查找一个引用计数为 0 的缓冲区。
  - 如果在当前缓冲桶内没有找到合适的缓冲区，那么在整个缓冲区池中查找一个引用计数为 0 且tick最少的缓冲区。
  - 找到合适的缓冲区后，分情况处理
    - 如果其未被cache过，则直接初始化并添加到桶中
    - 如果其被cache过，则修改指针再将其加入桶中
  - 如果未找到合适的缓冲区，则发出 panic 错误。
- brelse

由于LRU算法在bget中实现了，brelse函数也可以进行简化

```
//bio.c

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

    releasesleep(&b->lock);

    int tempBuc=b->blockno%NBUC;

    acquire(&bbuc[tempBuc].lock);
    b->refcnt--;
    release(&bbuc[tempBuc].lock);
}
```

运行结果

```
== Test    bcachetest: test0 ==
bcachetest: test0: OK
== Test    bcachetest: test1 ==
bcachetest: test1: OK
```

验证通过！

## 思考

- hints中专门提到桶的数量应该为素数，怎么会是呢
  - 选择素数作为桶的数目可以减少碰撞（多个键被映射到同一个桶），因为素数不容易被其他数字整除，有助于分散键的分布，减少碰撞的可能性。
  - 如果在哈希表需要调整大小时，选择素数作为新的桶数目，可以更均匀地扩展桶，避免集中在某个特定区域。

## 8.3 实验评分

```
== Test running kallocetest ==  
$ make qemu-gdb  
(49.6s)  
== Test    kallocetest: test1 ==  
    kallocetest: test1: OK  
== Test    kallocetest: test2 ==  
    kallocetest: test2: OK  
== Test kallocetest: sbrkmuch ==  
$ make qemu-gdb  
kallocetest: sbrkmuch: OK (7.5s)  
== Test running bcachetest ==  
$ make qemu-gdb  
(6.7s)  
== Test    bcachetest: test0 ==  
    bcachetest: test0: OK  
== Test    bcachetest: test1 ==  
    bcachetest: test1: OK  
== Test usertests ==  
$ make qemu-gdb  
usertests: OK (103.7s)  
== Test time ==  
time: OK  
Score: 70/70
```

# lab9: file system

## 9.1 Large files

Modify `bmap()` so that it implements a doubly-indirect block, in addition to direct blocks and a singly-indirect block. You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode. The first 11 elements of `ip->addrs[]` should be direct blocks; the 12th should be a singly-indirect block (just like the current one); the 13th should be your new doubly-indirect block. You are done with this exercise when `bigfile` writes 65803 blocks and `usertests` runs successfully:

### 前置知识

- dinode结构

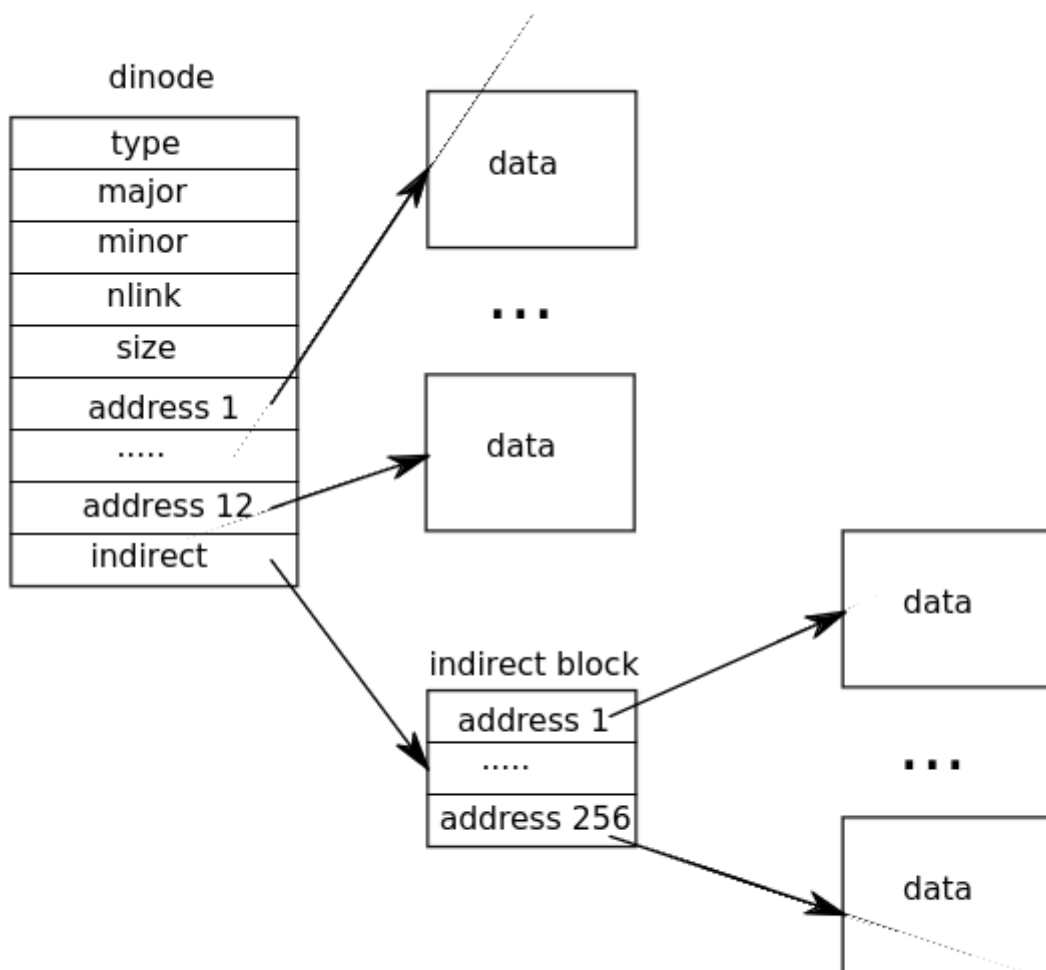


Figure 8.3: The representation of a file on disk.

xv6指导书中有dinode的结构，除了一些文件类型、设备号等信息之外，本实验重点关注addr数组。目前，地址数组一共有12个直接块和1个间接块，文件大小也限制在了268blocks

## 实现过程

- 修改数据结构

首先添加宏定义

```
//fs.h

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

同理，dinode和inode也需要修改

```
//fs.h

// On-disk inode structure
struct dinode {
    short type;           // File type
    short major;          // Major device number (T_DEVICE
only)
    short minor;          // Minor device number (T_DEVICE
only)
    short nlink;          // Number of links to inode in
file system
    uint size;            // Size of file (bytes)
    uint addrs[NDIRECT+2]; // Data block addresses
};
```

```
//file.h

struct inode {
    uint dev;             // Device number
    uint inum;            // Inode number
    int ref;              // Reference count
    struct sleeplock lock; // protects everything below here
    int valid;            // inode has been read from disk?
```

```

short type;           // copy of disk inode
short major;
short minor;
short nlink;
uint size;
uint addrs[NDIRECT+2];
};

```

- 修改bmap

所幸原始代码中有关于一级间接的，直接照抄即可

```

//fs.c

static uint
bmap(struct inode *ip, uint bn)
{
    ...

    // large lab9
    bn-=NINDIRECT;

    if(bn<NNINDIRECT){
        if((addr = ip->addrs[NDIRECT+1])==0)
            ip->addrs[NDIRECT+1]=addr=balloc(ip->dev);
        bp=bread(ip->dev, addr);
        a = (uint*)bp->data;
        if((addr=a[bn/NINDIRECT])==0){
            a[bn/NINDIRECT]=addr=balloc(ip->dev);
            log_write(bp);
        }
        brelse(bp);
        bp=bread(ip->dev, addr);
        a = (uint*)bp->data;
        if((addr=a[bn%NINDIRECT])==0){
            a[bn%NINDIRECT]=addr=balloc(ip->dev);
            log_write(bp);
        }
        brelse(bp);

        return addr;
    }

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

```



```
}
```

- 修改itrunc

释放block的函数，原始代码中也有关于一级间接的，同样直接照抄

```
//fs.c

void
itrunc(struct inode *ip)
{
    ...
    // large lab9
    struct buf *temp;
    uint *k;
    if(ip->addrs[NDIRECT + 1]){
        bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
        a = (uint*)bp->data;
        for(i = 0; i < NINDIRECT; i++){
            if(a[i]){
                temp = bread(ip->dev, a[i]);
                k = (uint*)temp->data;
                for(j = 0; j < NINDIRECT; j++){
                    if(k[j]){
                        bfree(ip->dev, k[j]);
                    }
                }
                brelse(temp);
                bfree(ip->dev, a[i]);
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT]);
        ip->addrs[NDIRECT] = 0;
    }

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

## 运行结果

[illegible]

验证通过!

## 思考

- 理论课上有过相关内容，理解起来比较简单
- 多级索引即能顺序存取，又能随机存取，满足了文件动态增长，插入、删除的需求，也能充分利用外存空间。缺点是本身带来的系统开销。

## 9.2 Symbolic links

You will implement the `symlink(char *target, char *path)` system call, which creates a new symbolic link at `path` that refers to file named by `target`. For further information, see the man page `symlink`. To test, add `symlinktest` to the Makefile and run it. Your solution is complete when the tests produce the following output (including `usertests` succeeding).

## 前置知识

- log
- 查看sysfile.c中代码可以发现，关于文件系统的操作中都会加上begin\_op()、end\_op()，查阅xv6 book看看怎么回事：

文件系统设计中最为有趣的问题之一是崩溃恢复。由于许多文件系统操作涉及对磁盘的多次写入，而在一些写入完成之前的崩溃可能会导致磁盘上的文件系统处于不一致的状态。

xv6 通过日志记录解决了此问题。一个 xv6 系统调用不会直接写入磁盘上的文件系统数据结构，而是将进行的磁盘写入操作的描述放入磁盘上的日志中。一旦系统调用已经记录了所有写入操作，它会向磁盘写入一个特殊的提交记录，包含一个完整的操作。在此之后，系统调用会将写入复制到磁盘上的文件系统数据结构中。在这些写入完成后，系统调用会清除磁盘上的日志。如果系统在崩溃后重启，文件系统代码会在运行其他进程之前进行恢复。

操作。如果日志被标记为包含完整的操作，则恢复代码将会将写入复制到磁盘上的文件系统中的适当位置。如果日志没有被标记为包含完整的操作，则恢复代码会忽略该日志。恢复代码最后会清除日志。

简单来说，日志使得操作是原子的，而begin\_op()、end\_op()用于日志系统中表明一个完整事务。

## 实现过程

- 宏定义

根据hints，如果要指明打开符号链接，需要增加一个标识位

```
//fcntl.h

#define O_NOFOLLOW 0x004
```

另外，为symlink类型增加一个定义

```
//stat.h

#define T_SYMLINK 4
```

- 实现sys\_symlink

实现一下系统调用

```
//sysfile.c

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

    begin_op();

    if(argstr(0, target, MAXPATH) < 0 || argstr(1, path,
MAXPATH) < 0)
    {
        end_op();
        return -1;
    }
    // 分派inode节点
    if((ip = create(path, T_SYMLINK, 0, 0))==0){
```

```

        end_op();
        return -1;
    }
    // 写入路径
    if(writei(ip, 0, (uint64)target, 0, MAXPATH)<MAXPATH){
        iunlockput(ip);
        end_op();
        return -1;
    }

    iunlockput(ip);
    end_op();
    return 0;
}

```

思路比较简单，分配一个inode节点，存放一下文件路径即可

- 修改sys\_open

添加一个处理措施：当类型为软链接且没有设置O\_NOFOLLOW标志，则通过符号链接打开文件

```

//sysfile.c

uint64
sys_open(void)
{
    ...
    // link lab9
    // 通过软链接递归找到真正的地址
    int depth=100;
    if(ip->type == T_SYMLINK && !(omode&O_NOFOLLOW)){

        for(int i=0;i<depth;i++){
            if(readi(ip, 0, (uint64)path, 0, MAXPATH) !=
MAXPATH) {
                iunlockput(ip);
                end_op();
                return -1;
            }
            iunlockput(ip);
            ip = namei(path);
            if(ip == 0) {

```

```

        end_op();
        return -1;
    }
    ilock(ip);
    if(ip->type != T_SYMLINK)
        break;
    }

    // 仍为符号链接
    if(ip->type == T_SYMLINK) {
        iunlockput(ip);
        end_op();
        return -1;
    }
}

```

- 

## 运行结果

```

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

```

验证通过！

## 思考

- 理论课上学习到过相应的内容。软链接的优点是：可对不存在的文件或目录进行创建，可对文件或目录进行创建，可交叉系统创建，缺点是若指向的原文档被删除，则相关软连接成为死链接。

## 9.3 实验评分

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (66.3s)
== Test running symlinktest ==
$ make qemu-gdb
(0.5s)
== Test    symlinktest: symlinks ==
    symlinktest: symlinks: OK
== Test    symlinktest: concurrent symlinks ==
    symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (142.4s)
== Test time ==
time: OK
Score: 100/100
```

## lab10: mmap

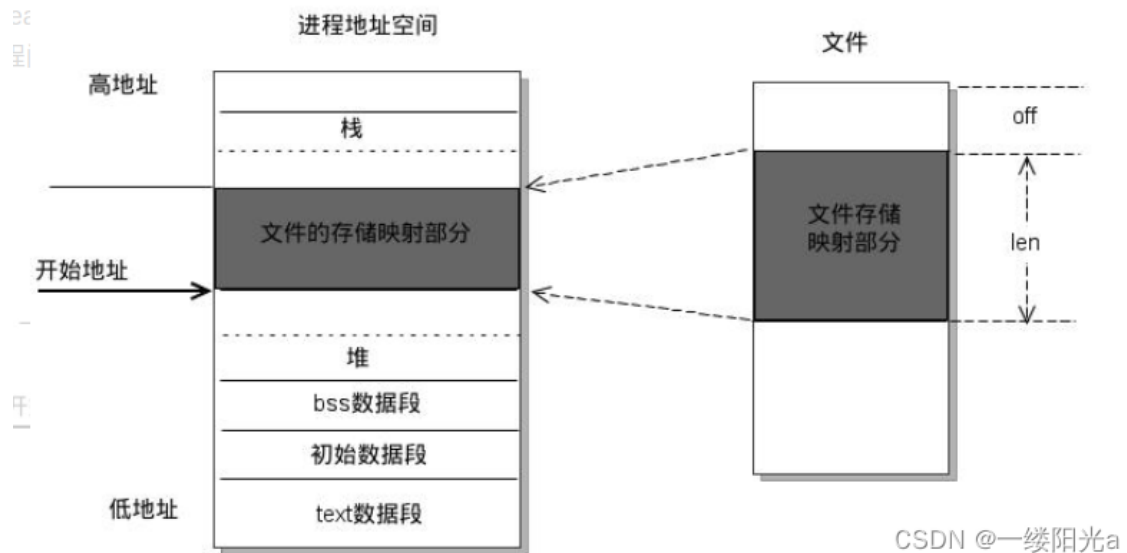
You should implement enough `mmap` and `munmap` functionality to make the `mmaptest` test program work. If `mmaptest` doesn't use a `mmap` feature, you don't need to implement that feature.

### 10.1 相关学习

- mmap是什么

mmap是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用read,write等系统调用函数。相反，内核空间对这段区

域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享。



如图所示

## 10.2 Implement mmap

- 首先添加头文件引用、添加函数原型、注册系统调用等操作不再赘述  
关于mmap和munmap函数的原型在lab中已经给出

```
void* mmap(void* addr, int length, int prot, int flags,
int fd, int offset);
int munmap(void* addr, int length);
```

- 首先定义VMA结构体，并添加到proc字段中

```
//proc.h

struct vma{
    int vaild;           //是否有效
    uint64 addr;         //起始地址
    int length;          //长度
    int prot;            //权限
    int flags;           //标志位
    int fd;              //文件描述符
    struct file *f;      //文件
    int offset;          //文件偏移
};

// Per-process state
struct proc {
```

```

...
    struct vma vma[VMANUM];
};

```

在进程分配时也要初始化

```

//proc.c

allocproc(void)
{
    ...
found:
    ...
    memset(&p->vma, 0, sizeof(p->vma));
    ...
}

```

- sys\_mmap

```

//sysfile.c

uint64 sys_mmap(void)
{
    uint64 addr;
    int length;
    int prot;
    int flags;
    int fd;
    struct file* f;
    int offset;
    struct proc *p=myproc();

    if(argaddr(0, &addr)< 0||argint(1, &length)< 0
||argint(2, &prot)<0||argint(3, &flags)<0|| argfd(4, &fd,
&f )< 0||argint(5, &offset)<0)
        return 0xffffffffffffffff;

    if(length < 0)
        return 0xffffffffffffffff;

    if(f->writable == 0 && (prot & PROT_WRITE) != 0 && flags
== MAP_SHARED)

```



```

        return 0xffffffffffffffff;

    if(p->sz+length>MAXVA)
        return 0xffffffffffffffff;

    for(int i = 0; i < VMANUM; i++) {
        if(p->vma[i].vaild == 0) {
            p->vma[i].vaild = 1;
            p->vma[i].addr = p->sz;
            p->vma[i].length = length;
            p->vma[i].flags = flags;
            p->vma[i].prot = prot;
            p->vma[i].fd = fd;
            p->vma[i].f = f;
            p->vma[i].offset = offset;

            // 增加引用计数
            filedup(f);

            p->sz += length;
            return p->vma[i].addr;
        }
    }

    return 0xffffffffffffffff;
}

```

比较简单，只需要传入参数再在VMA数组中找到一块存放即可

- trap.c

首先设置一个处理函数，用于在mmap触发页面错误时使用

```

//file.c

int
mmapHandler(uint va,int cause)
{
    int i;
    struct proc* p = myproc();

    for(i = 0; i < VMANUM; ++i) {
        //先定位一下是哪个VMA
    }
}

```

```

        if(p->vma[i].vaild==1 && p->vma[i].addr <= va && va <=
p->vma[i].addr + p->vma[i].length - 1) {
            break;
        }
    }
    if(i == VMANUM)
        return -1;

    struct file* f = p->vma[i].f;
    if((cause == 13 && f->readable == 0) || (cause == 15 && f-
>writable == 0))
        return -1;

    int flags = PTE_U;
    if(p->vma[i].prot & PROT_READ) flags |= PTE_R;
    if(p->vma[i].prot & PROT_WRITE) flags |= PTE_W;
    if(p->vma[i].prot & PROT_EXEC) flags |= PTE_X;

    void* pa = kalloc();
    if(pa == 0)
        return -1;
    memset(pa, 0, PGSIZE);

    // 读取文件内容
    ilock(f->ip);
    int offset = p->vma[i].offset + PGROUNDDOWN(va - p-
>vma[i].addr);
    if(readi(f->ip, 0, (uint64)pa, offset, PGSIZE)==0){
        iunlock(f->ip);
        kfree(pa);
        return -1;
    }

    iunlock(f->ip);

    // 添加页面映射
    if(mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE,
(uint64)pa, flags) != 0) {
        kfree(pa);
        return -1;
    }

    return 0;

```

```
}
```

首先定位出是哪个VMA触发了页面错误，然后分配一个物理内存页面va，将文件读取后在进程页表中添加映射。

然后就可以在trap.c中添加错误处理了

```
//trap.c

void
usertrap(void)
{
    ...
    if(r_scause() == 8){
        ...
    }else if (r_scause()==13 || r_scause()==15){

        uint64 va=r_stval();
        uint64 sp = PGROUNDUP(p->trapframe->sp);

        if(sp-1<va&&va<p->sz){
            if(mmapHandler(va, r_scause())<0)
                p->killed=1;
        }else{
            p->killed=1;
        }

    }else {
        ...
    }
    ...
}
```

- sys\_munmap

```
//sysfile.c

uint64 sys_munmap(void)
{
    uint64 addr;
    struct proc* p = myproc();
    int length;
    int i;
```

```

if(argaddr(0, &addr) < 0 || argint(1, &length) < 0)
    return -1;

for(i = 0; i < VMANUM; i++) {
    if(p->vma[i].vaild==1 && p->vma[i].length >= length) {
        //定位一下VMA
        if(p->vma[i].addr == addr) {
            p->vma[i].addr += length;
            p->vma[i].length -= length;
            break;
        }else if(p->vma[i].addr + p->vma[i].length==addr +
length ) {
            p->vma[i].length -= length;
            break;
        }
    }
}
if(i == VMANUM)
    return -1;

//如果是shared, 写回
if(p->vma[i].flags == MAP_SHARED && (p->vma[i].prot &
PROT_WRITE) != 0) {
    fwrite(p->vma[i].f, addr, length);
}

uvmunmap(p->pagetable, addr, length / PGSIZE, 1);

//最后清空VMA
if(p->vma[i].length == 0) {
    fclose(p->vma[i].f);
    p->vma[i].vaild = 0;
}

return 0;
}

```

查找匹配的VMA，如果 VMA是MAP\_SHARED映射，并且拥有写权限，就会调用 fwrite函数将数据写回文件。之后，调用uvmunmap函数，解除页面映射，清空VMA

- 修改exit和fork

在进程退出时，要清空一下VMA，把需要写回的全部写回，映射全部取消

```
//proc.c

void
exit(int status)
{
    ...
    for(int i = 0; i < VMANUM; ++i) {
        if(p->vma[i].vaild==1) {
            if(p->vma[i].flags == MAP_SHARED &&(p->vma[i].prot &
PROT_WRITE) != 0) {
                fwrite(p->vma[i].f, p->vma[i].addr, p-
>vma[i].length);
            }
            fclose(p->vma[i].f);
            uvmunmap(p->pagetable, p->vma[i].addr, p-
>vma[i].length/PGSIZE,1);
            p->vma[i].vaild = 0;
        }
    }
    ...
}
```

根据hints，在fork出子进程时，还需要复制一下父进程的VMA

```
proc.c

int
fork(void)
{
    ...
    for(i=0;i<VMANUM;i++){
        if(p->vma[i].vaild==1){
            memmove(&np->vma[i], &p->vma[i], sizeof(p->vma[i]));
            filedup(p->vma[i].f);
        }
    }
    ...
}
```

- 解除panic

最后，在uvmunmap和uvmcopy时解除panic，防止子进程复制时导致的报错

```
//vm.c
//void uvmunmap()
//int uvmcopy()

if((*pte & PTE_V) == 0)
    continue;
```

## 10.3 实验评分

```
== Test mmaptest: mmap f ==
mmaptest: mmap f: OK
== Test mmaptest: mmap private ==
mmaptest: mmap private: OK
== Test mmaptest: mmap read-only ==
mmaptest: mmap read-only: OK
== Test mmaptest: mmap read/write ==
mmaptest: mmap read/write: OK
== Test mmaptest: mmap dirty ==
mmaptest: mmap dirty: OK
== Test mmaptest: not-mapped unmap ==
mmaptest: not-mapped unmap: OK
== Test mmaptest: two files ==
mmaptest: two files: OK
== Test mmaptest: fork_test ==
mmaptest: fork_test: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (95.8s)
== Test time ==
time: OK
Score: 140/140
```

## 10.4 思考

- 直观上mmap可以完成用户对文件的直接读写，而read、write还需要把页面调入page cache中，所以mmap速度应该是优于read和write的，但是根

据网上一些说法，由于硬件的优化，mmap的性能其实通常不如直接read、write

- 另外，mmap在建立映射时必须制定映射区域，所以这种方式只适用于更新、读写一块固定大小的文件区域，而不能不断的以增长方式向文件写内容。