

介绍

本实验为北京大学操作系统课程实验，项目基于xv6-k210，结合内核赛相关要求进行更改。

环境配置

借助课程助教提供的[相关文档](#)以及网络上已有的成功案例完成配置。

容器准备

按照助教的建议，使用PKUClab（参考PKUClab自带的[入门文档](#)）

密钥准备

如果用户目录下已有".ssh/id_rsa.pub"，可直接使用

如果没有则可以使用命令

```
ssh-keygen -t ed25519
```

生成一个新的密钥

创建云主机

按照文档的步骤来，本人选择的几个参数：

场景：Labs_and_Courses，大小：12（2核2GB），启动源：镜像启动

操作系统：Ubuntu 24.0.1，硬盘类型：SSD，大小：50G

网络设置：共享网络，pku-new，其余按照默认设置

等待一段时间后，在终端中输入：

```
ssh 用户名@内网IP
```

以检测云主机是否可以成功连接

登录北大网关

文档中提供了登录网关的脚本程序，将其保存在本机中（例如叫login.py），然后使用scp命令将文件发送到服务器中，然后在服务器中执行

```
scp login.py ubuntu@内网IP:
```

执行命令运行连接网关脚本文件

```
python3 login.py
```

执行成功的话，可以在北大网关中查看到对应的内网IP地址（注意北大网关最多只能连接4个，连接前已满四个是无法成功连接上的）

vscode连接服务器

第二次登录vscode尝试连接至服务器时，vscode发生了报错，然而尝试在其他终端手动连接服务器（"ssh ubuntu@内网IP"）时没有问题

参考了若干大模型的意见后，我在vscode的settings.json中添加了以下部分，最后成功重新连接上服务器

```
"remote.SSH.showLoginTerminal": true,  
"remote.SSH.logLevel": "trace",  
//这里填自己使用的ssh对应的config路径  
"remote.SSH.configFile": "C:\\\\Users\\\\32608\\\\.ssh\\\\config",  
"remote.SSH.useLocalServer": true
```

基础工程准备

从零开始实现项目比较困难，助教推荐使用k210平台开始此实验

在下载k210之前，先在vscode上远程连接到PKUClab的服务器

打开vscode并点击左下角用于远程连接的按钮，选择“连接到主机”，然后添加到“SSH主机”，输入以下命令连接到服务器，最后更新本地对应的SSH配置文件即可

```
ssh 用户名@内网IP
```

使用git命令将k210项目复制到ubuntu本地，成功后可以发现一个xv-k210文件夹

```
git clone https://github.com/HUST-OS/xv6-k210.git
```

接下来就可以在k210的基础上进行实验了

开始实验

评测流程

1. 通过make all命令生成kernel-qemu内核文件

2. 评测平台会调用如下命令自动启动qemu模拟器

```
qemu-system-riscv64 -machine virt -kernel kernel-qemu -m 128M -nographic -smp 2 -
bios default -drive file=sdcard.img,if=none,format=raw,id=x0 -device virtio-blk
device,drive=x0,bus=virtio-mmio-bus.0 -no-reboot -device virtio-net
device,netdev=net -netdev user,id=net
```

3. kernel-qemu的内核需要自动扫描评测机的sdcard.img所对应的镜像文件，镜像文件根目录下存放32个测试程序

4. 根据自己内核的实现情况，在内核进入用户态后主动调用这些测试程序

5. 调用完测试程序后，主动关机退出

初步准备与本地测试

本地测试样例

参考助教提供的几份文档，如果希望在本地测试，需要先把测试程序clone下来：

```
git clone https://github.com/oscomp/testsuits-for-oskernel.git
```

参考助教的文档，在docker中编译文件

```
sudo docker run -ti --rm -v ./testsuits-for-oskernel/riscv-syscalls-
testing:/testing --privileged=true docker.educg.net/cg/os-contest:2024p6 /bin/bash
```

运行成功后，user目录下会出现riscv64文件夹，里面是本地测试需要用到的测试样例

然后将riscv64文件夹中的内容保存在xv6-k210文件夹的tests文件夹内（文件夹名称tests是重要的，这关系到Makefile中fs下的内容）

```
cp -r testsuits-for-oskernel/riscv-syscalls-testing/user/riscv64/* xv6-k210/tests
```

拉取本地所需的评测镜像

1. 安装docker

根据文档，运行以下命令

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
```

```
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

# 使用apt安装docker套件
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin
```

2.设置代理镜像

使用vi打开/创建daemon.json文件(不加sudo可能因为权限不足导致无法修改)

```
sudo vi -R /etc/docker/daemon.json
```

输入字母'a'或'i', 开始编辑daemon.json文件

复制以下配置

```
{
  "registry-mirrors": [
    "https://docker.mirrors.lcpu.dev",
    "https://registry.docker-cn.com",
    "https://docker.mirrors.ustc.edu.cn",
    "https://hub-mirror.c.163.com",
    "https://mirror.baidubce.com",
    "https://ccr.ccs.tencentyun.com"
  ]
}
```

复制成功后, 敲下esc, 然后输入":w"保存, 输入":q"退出, 然后可以在etc/docker下找到编辑成功的daemon.json文件

3.拉取评测镜像

```
sudo docker pull docker.educg.net/cg/os-contest:2024p6
```

内容较大（约8.3G），可能花费一定时间

使用命令查看docker已有镜像，可以发现docker.educg.net/cg/os-contest

```
sudo docker images
```

4.启动容器

```
sudo docker run -ti --rm -v ./xv6-k210:/xv6 --privileged=true  
docker.educg.net/cg/os-contest:2024p6 /bin/bash
```

大概就是吧xv6-k210挂载到xv6上

其他可能需要的调整

参考课程群等助教、同学们提出的可能出现的问题

1.替换bootloader/SBI/sbi-qemu为新下载的版本

下载rustsbi

```
wget https://github.com/rustsbi/rustsbi-qemu/releases/download/v0.2.0-  
alpha.2/rustsbi-qemu-release.zip
```

安装unzip并将压缩包解压

```
sudo apt-get update  
sudo apt-get install unzip
```

解压缩（我这里放在了xv6-k210文件夹中）

```
unzip rustsbi-qemu-release.zip -d xv6-k210
```

最后用解压出来的rustsbi-qemu（文档中助教提到使用rustsbi-qemu替换，而课程群中有同学说是用rustsbi-qemu.bin来替换，本人采用的前者）替换已有的sbi-qemu（注意最后的名字还是叫sbi-qemu）

2.修改platform

Makefile的第一二行的"platform := k210"改为"platform := qemu"

3.修改CPU数量

Makefile第129行, "CPUS := 2"修改为"CPUS := 1", 多CPU可能存在并发问题 (当然也可能不会出bug)

4.增大模拟时的内存空间大小

Makefile第132行"QEMUOPTS = -machine virt -kernel \$T/kernel -m 8M -nographic"的8M修改为32M

5.修改make fs

fs原本的内容如下:

```
fs: $(UPROGS)
    @if [ ! -f "fs.img" ]; then \
        echo "making fs image..."; \
        dd if=/dev/zero of=fs.img bs=512k count=512; \
        mkfs.vfat -F 32 fs.img; fi
    @sudo mount fs.img $(dst)
    @if [ ! -d "$(dst)/bin" ]; then sudo mkdir $(dst)/bin; fi
    @sudo cp README $(dst)/README
    @for file in $( ls $U/* ); do \
        sudo cp $$file $(dst)/${file#$U/}; \
        sudo cp $$file $(dst)/bin/${file#$U/}; done
    @sudo umount $(dst)
```

修改后的fs内容如下:

```
fs: $(UPROGS)
    @if [ ! -f "sdcard.img" ]; then \
        echo "making fs image..."; \
        dd if=/dev/zero of=sdcard.img bs=1M count=128; \
        mkfs.vfat -F 32 sdcard.img; fi
    @mount sdcard.img $(dst)
    @if [ ! -d "$(dst)/bin" ]; then mkdir $(dst)/bin; fi
    @for file in $( ls $U/* ); do \
        cp $$file $(dst)/bin/${file#$U/}; done
    @cp -R tests/* $(dst)
    @umount $(dst)
```

6.修改QEMUOPTS参数

第144行, 修改fs.img为sdcard.img

7.添加all

在Makefile中添加make all命令

```
all: build
    mv $T/kernel $T/kernel-qemu
```

```
cp $T/kernel-qemu kernel-qemu
```

初步测试代码

使用前面提到的"sudo docker run"命令进入评测镜像，并在镜像中进入xv6文件夹

输入以下指令

```
make fs
make run
```

修改init.c等文件，运行测试样例

1.解决"panic:init exiting"

在前面初步测试配置模块时，发现最后出现"panic:init exiting"字样，而非正常的shell，结合文档发现是没有主动调用关机导致的。

结合助教提供的相关文档，和"kernel/include/sbi.h"里面提供的sbi接口来实现这一功能。

可参考实验自带的文档"doc/用户使用-系统调用.md"

查看"kernel/include/sbi.h"发现已经提供了sbi_shutdown函数

```
static inline void sbi_shutdown(void)
{
    SBI_CALL_0(SBI_SHUTDOWN);
}
```

首先按照文档，在init.c的最开始，加入以下代码

```
#define __asm_syscall(...) \
    __asm__ __volatile__( "ecall\n\t" \
                          : "=r"(a0) \
                          : __VA_ARGS__ \
                          : "memory"); \
    return a0;

static inline long __syscall0(long n)
{
    register long a7 __asm__("a7") = n;
    register long a0 __asm__("a0");
    __asm_syscall("r"(a7))
}
```

并在"return 0"前面加入一行代码

```
__syscall0(SYS_shutdown);
```

光这样还不行，还没定义SYS_shutdown是什么，同时还需要把这个函数实现成系统调用

首先在kernel/include下找到sysnum.h，添加SYS_shutdown

```
#define SYS_shutdown    210
```

别忘了在init.c开头引用头文件，否则会找不到SYS_shutdown

```
#include "kernel/include/sysnum.h"
```

同时按照实验自带的文档（doc目录下），还要在kernel下的syscall.c中，添加系统调用函数声明

```
extern uint64 sys_shutdown(void);
```

同时还要更新系统调用表

```
static uint64 (*syscalls[])(void) = {
    .....
    [SYS_shutdown]    sys_shutdown,
};

static char *sysnames[] = {
    .....
    [SYS_shutdown]    "shutdown",
}
```

同时浏览各个文件可以找到，sysproc.c文件中，有各个功能的实现，正好可以在这里实现刚刚声明的sys_shutdown

根据文档提示的sbi.h中存在关机的接口，需要引用相关的头文件:在sysproc.c中添加引用

```
#include "include/sbi.h"
#include "include/vm.h"
```

借助已有的sbi_shutdown,实现刚才声明的sys_shutdown函数

```
uint64
sys_shutdown(void) {
```



```
sbi_shutdown();  
return 0;  
}
```

可是如果此时还是直接进入评测镜像"make fs", "make run", 会发现输出结果完全没变!!!

再查看一遍文档, 会发现在修改如系统调用号这类文件后, 还需要将修改同步到initcode!!!

这就引出下一个问题: 如何同步修改结果到initcode? (本问题后续也将写在下一个问题中)

2.同步修改结果到initcode

按照助教文档所说的, 先编译init.c得到_init, 那怎么编译init.c呢?

很简单, 之前make fs后, 会在xv-user中生成_init!

然后依次运行文档提供的指令

```
riscv64-linux-gnu-objcopy -S -O binary xv6-user/_init oo
```

```
od -v -t x1 -An oo | sed -E 's/ (.{2})/0x\1,/g' > kernel/include/initcode.h
```

最后可以在kernel/include中找到initcode.h文件, 该文件是一系列十六进制数

然后还需要修改kernel/proc.c中的initcode, 可以在240行左右发现initcode本来是一串数字

```
uchar initcode[] = {  
    0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x45, 0x02,  
    0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x35, 0x02,  
    0x93, 0x08, 0x70, 0x00, 0x73, 0x00, 0x00, 0x00,  
    0x93, 0x08, 0x20, 0x00, 0x73, 0x00, 0x00, 0x00,  
    0xef, 0xf0, 0x9f, 0xff, 0x2f, 0x69, 0x6e, 0x69,  
    0x74, 0x00, 0x00, 0x24, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00  
};
```

新生成的initcode.h内容较多, 可以通过下面的方式替换

```
uchar initcode[] = {  
#include "include/initcode.h"  
};
```

再运行, 应该就不会出现panic的情况了

3.运行测试样例

修改init.c等文件时，可能会出现修改了但是make run后结果不变的情况，这时可以尝试make clean清除旧文件重新编译

然后参考原来的代码

```
char *argv[] = { "sh", 0 };
.....
exec("sh", argv);
```

可以在init.c中循环来运行测试样例，例如

```
char *argv[] = {0};

char *tests[] = {
    "brk",
    "chdir",
    .....
    "yield",
};
```

在循环中

```
int counts = sizeof(tests) / sizeof((tests)[0]);

for(int i = 0; i < counts; i++){
    .....
    pid = fork();
    if(pid < 0){
        .....
    }
    if(pid == 0){
        exec(tests[i], argv);
        .....
    }
}
```

当然，要注意地一点是，在测试样例都在xv6-k210根目录下时，tests可以直接命名，否则要加上路径（当然评测机上测试样例都是在根目录，所以最终提交的代码需要这么写）

当看到输出类似下图时，就正式地可以在本地运行测试了，下一步就是实现各种功能完成测试

```
hart 0 init done
init: starting brk
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 214
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 214
pid 2 brk: unknown sys call 214
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 214
pid 2 brk: unknown sys call 214
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 64
pid 2 brk: unknown sys call 93

usertrap(): unexpected scause 0x0000000000000002 pid=2 brk
sepc=0x00000000000000c0 stval=0x0000000000000000
```

正式实验

xv6-k210中已经有不少的系统函数，不少测试可以在这些的基础上进行修改完成

在上一阶段的末尾，发现有许多"unknown sys call xxx"，这里可以在测试数据的"oscomp_syscalls.md"文件中找到这些系统调用号的意义

根据助教提供的经验分享，大致按照以下顺序依次完成各组函数：

1. exit/getpid/getppid/gettimeofday/uname/times/brk
2. clone/fork/wait/waitpid/mmap/munmap/execve
3. close/dup2/dup/pipe/read/write
4. openat/open/fstat/getdents/chdir/getcwd
5. mkdir/unlink/mount/unmount

1.exit

以文档提到的exit为例，运行这个测试样例会提到四个系统调用号，可以在测试文件中找到它们对应的意义

```
#define SYS_write 64
功能：从一个文件描述符中写入；
输入：
    fd: 要写入文件的文件描述符。
    buf: 一个缓存区，用于存放要写入的内容。
    count: 要写入的字节数。
返回值：成功执行，返回写入的字节数。错误，则返回-1。
int fd, const void *buf, size_t count;
ssize_t ret = syscall(SYS_write, fd, buf, count);

#define SYS_exit 93
功能：触发进程终止，无返回值；
输入：终止状态值；
返回值：无返回值；
int ec;
syscall(SYS_exit, ec);

#define SYS_clone 220
功能：创建一个子进程；
输入：
    flags: 创建的标志，如SIGCHLD；
    stack: 指定新进程的栈，可为0；
    ptid: 父线程ID；
    tls: TLS线程本地存储描述符；
    ctid: 子线程ID；
返回值：成功则返回子进程的线程ID，失败返回-1；
pid_t ret = syscall(SYS_clone, flags, stack, ptid, tls, ctid)

#define SYS_wait4 260
功能：等待进程改变状态；
输入：
    pid: 指定进程ID，可为-1等待任何子进程；
    status: 接收状态的指针；
    options: 选项：WNOHANG, WUNTRACED, WCONTINUED；
返回值：成功则返回进程ID；如果指定了WNOHANG，且进程还未改变状态，直接返回0；失败则返回-1；
pid_t pid, int *status, int options;
pid_t ret = syscall(SYS_wait4, pid, status, options);
```

显然，要通过exit这个测试样例，至少需要实现以上四个系统调用

参考doc文件夹下的"构建调试-系统调用.md"等资料实现

首先在"sysnum.h"中，修改已有的调用号（SYS_exit，SYS_write），添加没有的调用号（SYS_clone,SYS_wait4）

已有的调用在修改完调用号后就不需要再更改了，重点放在需要添加的调用中

SYS_clone

从文档中查看到SYS_clone的功能是创建一个子进程，这就提示可以仿造fork()的实现来完成这个调用

在syscall.c中添加SYS_clone相关的系统调用表、声明函数

在proc.h中声明clone函数

参考sys_exit函数

```
uint64
sys_exit(void)
{
    int n;
    if(argint(0, &n) < 0)
        return -1;
    exit(n);
    return 0; // not reached
}
```

写出sys_clone函数

```
uint64
sys_clone(void)
{
    int stack;
    if(argint(1, &stack) < 0)
        return -1;
    return clone(stack);
}
```

而clone函数则在fork函数的基础上添加对stack的处理（增加参数stack）

```
if(stack < 0 || stack > p->sz)
    return -1;
```

```
// 按照函数的要求，如果stack合理，那么更新sp寄存器
if(stack > 0) np->trapframe->sp = stack;
```

trapframe的内容在kernel/include/trap.h中查看

SYS_wait4

同样修改系统调用表，在proc.h和defs.h中声明函数

参考原来已有的wait函数的实现

```
uint64
sys_wait(void)
{
    uint64 p;
    if(argaddr(0, &p) < 0)
        return -1;
    return wait(p);
}
```

结合文档中wait4的功能，在wait的基础上添加status和options两个参数

```
uint64
sys_wait4(void)
{
    uint64 status;
    int pid, options;

    if(argint(0, &pid) < 0) return -1;
    if(argaddr(1, &status) < 0) return -1;
    if(argint(2, &options) < 0) return -1;

    return wait4(pid, status, options);
}
```

proc.c中的wait4函数的内容也是基于wait函数来修改

首先是加入对pid的判断,pid = -1则可等待任何子进程

```
if(pid == -1) {
    return wait(addr);
}
```

将np->xstate替换为status

```
status = np->xstate << 8;
if(addr != 0 && copyout2(addr, (char *)&status, sizeof(status)) < 0) {
    .....
}
```

完成上述工作后，运行测试发现输出"exit OK"即代表exit测试在本地通过

2.getpid

主要内容已经实现，只需要把SYS_getpid修改为要求的172即可

注意test_getpid.c中提到的："理想结果：得到进程 pid，注意要关注 pid 是否符合内核逻辑，不要单纯以 Test OK! 作为判断。"

3.getppid

本地运行测试程序getppid后，得到信息：pid 4 getppid: unknown sys call 173

查阅文档oscomp_syscalls.md

```
#define SYS_getppid 173
功能：获取父进程ID；
输入：系统调用ID；
返回值：成功返回父进程ID；
pid_t ret = syscall(SYS_getppid);
```

仿造已经实现的getpid的功能实现getppid即可

```
uint64
sys_getpid(void)
{
    return myproc()->pid;
}
```

即：在sysnum.h中添加SYS_getppid、在syscall.c中完善系统调用表等、在sysproc.c中实现调用

输出如："getppid success. ppid : 1"即代表通过测试

4.gettimeofday

直接运行该测试样例后发现："pid 6 gettimeofday: unknown sys call 169"

查看文档寻找169号系统调用

```
#define SYS_gettimeofday 169
功能：获取时间；
输入：timespec结构体指针用于获得时间值；
返回值：成功返回0，失败返回-1；
struct timespec *ts;
int ret = syscall(SYS_gettimeofday, ts, 0);
```

查看测试程序

```
void test_gettimeofday() {
    TEST_START(__func__);
```

```

int test_ret1 = get_time();
volatile int i = 12500000; // qemu时钟频率12500000
while(i > 0) i--;
int test_ret2 = get_time();
if(test_ret1 > 0 && test_ret2 > 0){
    printf("gettimeofday success.\n");
    printf("start:%d, end:%d\n", test_ret1, test_ret2);
    printf("interval: %d\n", test_ret2 - test_ret1);
}else{
    printf("gettimeofday error.\n");
}
TEST_END(__func__);
}

```

得知一个重要信息，即qemu的时钟频率为12500000

在全局搜索timespec，可以发现timespec结构体的定义

```

struct timespec {
    time_t  tv_sec;      /* seconds */
    long    tv_nsec;     /* and nanoseconds */
};

```

最后系统内建有r_time来获取时间

测试样例输出"gettimeofday success."则测试成功

5.uname

运行测试程序发现缺少160号系统调用

```

#define SYS_uname 160
功能：打印系统信息；
输入：utsname结构体指针用于获得系统信息数据；
返回值：成功返回0，失败返回-1；
struct utsname *uts;
int ret = syscall(SYS_uname, uts);

```

在测试样例的uname.c中找到utsname的定义

```

struct utsname {
    char sysname[65];
    char nodename[65];
    char release[65];
    char version[65];
    char machine[65];
};

```



```
    char domainname[65];  
};
```

同时查看测试程序

```
void test_uname() {  
    TEST_START(__func__);  
    int test_ret = uname(&un);  
    assert(test_ret >= 0);  
  
    printf("Uname: %s %s %s %s %s %s\n",  
        un.sysname, un.nodename, un.release, un.version, un.machine,  
        un.domainname);  
  
    TEST_END(__func__);  
}
```

可以发现自己填上合适的参数即可

若实现正确，最后会输出自己填写的内容

6.times

运行样例发现缺少153号系统调用，[查看文档](#)

```
#define SYS_times 153  
功能：获取进程时间；  
输入：tms结构体指针，用于获取保存当前进程的运行时间数据；  
返回值：成功返回已经过去的滴答数，失败返回-1；  
struct tms *tms;  
clock_t ret = syscall(SYS_times, tms);
```

查看函数sys_uptime的注释：return how many clock tick interrupts have occurred，可以发现sys_uptime就是需要的功能，直接类似shutdown的实现借用即可

输出mytimes success即代表测试通过

7.brk

运行测试样例，提示缺少214号系统调用，相关内容为

```
#define SYS_brk 214  
功能：修改数据段的大小；  
输入：指定待修改的地址；  
返回值：成功返回0，失败返回-1；  
uintptr_t brk;  
uintptr_t ret = syscall(SYS_brk, brk);
```

可以发现和SYS_brk很相似的SYS_sbrk，后者在文档"用户使用-内存管理"中提到功能：sbrk是一个系统调用，用于进程缩小或增加其内存。growproc调用uvmmalloc或uvmmdealloc，这取决于n是正的还是负的。

按照brk.c中提到的：Linux 中brk(0)只返回0，此处与Linux表现不同，应特殊说明。

而成功通过的输出则形如：

```
"Before alloc,heap pos: [num]"  
"After alloc,heap pos: [num+64]"  
"Alloc again,heap pos: [num+128]"
```

8.clone

相关内容已经在exit中处理了

9.fork

同上

10.wait

同上

11.waitpid

运行测试样例，发现缺少调用号124

```
#define SYS_sched_yield 124  
功能：让出调度器；  
输入：系统调用ID；  
返回值：成功返回0，失败返回-1；  
int ret = syscall(SYS_sched_yield);
```

流程与前面的实现类似，sys_sched_yield函数直接调用已有的yield函数即可

12.execve

缺少的系统调用号为221

在sysnum.h中发现已经存在SYS_exec，只需将其调用号更改为221即可

13.close/openat

发现运行这两个测试程序后，共缺少56、57两个调用号

查阅文档确定对应的内容

```
#define SYS_close 57
功能：关闭一个文件描述符；
输入：
fd：要关闭的文件描述符。
返回值：成功执行，返回0。失败，返回-1。
int fd;
int ret = syscall(SYS_close, fd);
```

```
#define SYS_openat 56
功能：打开或创建一个文件；

输入：
fd：文件所在目录的文件描述符。
filename：要打开或创建的文件名。如为绝对路径，则忽略fd。如为相对路径，且fd是AT_FDCWD，则filename是相对于当前工作目录来说的。如为相对路径，且fd是一个文件描述符，则filename是相对于fd所指向的目录来说的。
flags：必须包含如下访问模式的其中一种：O_RDONLY, O_WRONLY, O_RDWR。还可以包含文件创建标志和文件状态标志。
mode：文件的所有权描述。详见`man 7 inode`。
```

```
返回值：成功执行，返回新的文件描述符。失败，返回-1。
int fd, const char *filename, int flags, mode_t mode;
int ret = syscall(SYS_openat, fd, filename, flags, mode);
```

而sysnum.h中本身是有SYS_open和SYS_close的，先把它们改成对应的号码

再次运行发现三个测试程序都有类似的输出：

```
--- Assert Fatal ! ---
```

在open.c中发现O_DIRECTORY

在全局搜索字符后，在测试样例相关文件stddef.h中发现xv-k210本身不存在的权限

```
#define O_DIRECTORY 0x02000000
```

于是将其添加到kernel/fcntl.h中

同时发现O_CREATE的值也不同，相应地也要修改

查看open.c的相关测试代码

```
void test_open() {
    TEST_START(__func__);
    // O_RDONLY = 0, O_WRONLY = 1
```

```
int fd = open("./text.txt", 0);
assert(fd >= 0);
char buf[256];
int size = read(fd, buf, 256);
if (size < 0) {
    size = 0;
}
write(STDOUT, buf, size);
close(fd);
TEST_END(__func__);
}
```

发现需要open函数的返回值大于等于0，否则就会出现上述输出

而openat的额外输出也印证了这一点

```
open dir fd: -1
openat fd: -1
```

在sysfile.c中可以看到sys_open原本的实现

```
uint64
sys_open(void)
{
    char path[FAT32_MAX_PATH];
    int fd, omode;
    struct file *f;
    struct dirent *ep;

    if(argstr(0, path, FAT32_MAX_PATH) < 0 || argint(1, &omode) < 0)
        return -1;

    if(omode & O_CREATE){
        ep = create(path, T_FILE, omode);
        if(ep == NULL){
            return -1;
        }
    } else {
        if((ep = ename(path)) == NULL){
            return -1;
        }
        elock(ep);
        if((ep->attribute & ATTR_DIRECTORY) && omode != O_RDONLY){
            eunlock(ep);
            eput(ep);
            return -1;
        }
    }
}
```

```

    if((f = filealloc()) == NULL || (fd = fdalloc(f)) < 0){
        if (f) {
            fileclose(f);
        }
        eunlock(ep);
        eput(ep);
        return -1;
    }

    if(!(ep->attribute & ATTR_DIRECTORY) && (omode & O_TRUNC)){
        etrunc(ep);
    }

    f->type = FD_ENTRY;
    f->off = (omode & O_APPEND) ? ep->file_size : 0;
    f->ep = ep;
    f->readable = !(omode & O_WRONLY);
    f->writable = (omode & O_WRONLY) || (omode & O_RDWR);

    eunlock(ep);

    return fd;
}

```

从开始的几行可以发现，原有的sys_open只支持两个参数，而文档要求需要实现的open函数要能支持四个参数，即fd,filename,flags,mode

因此开头应该修改为

```

char path[FAT32_MAX_PATH];
int fd, omode, flags;

if(argint(0, &fd) < 0 || argstr(1, path, FAT32_MAX_PATH) < 0 || argint(2, &flags)
< 0 || argint(3, &omode) < 0) {
    return -1;
}

```

而sys_open的大体框架不需要改变，只需要在部分判断条件中增加对O_DIRECTORY等的判断即可

14.dup2/dup

根据文档，dup/dup3的调用号为23/24，而系统原有23/24的调用号

```

#define SYS_test_proc      22
#define SYS_dev            23
#define SYS_readdir        24
#define SYS_getcwd         25
#define SYS_rename         26

```

将SYS_dev/SYS_readdir修改(目前分别修改为50/27,不确定是否会出问题), 并把23/24赋给dup/dup3

```
#define SYS_dup 23
```

功能: 复制文件描述符;

输入:

fd: 被复制的文件描述符。

返回值: 成功执行, 返回新的文件描述符。失败, 返回-1。

```
int fd;  
int ret = syscall(SYS_dup, fd);
```

```
#define SYS_dup3 24
```

功能: 复制文件描述符, 并指定了新的文件描述符;

输入:

old: 被复制的文件描述符。

new: 新的文件描述符。

返回值: 成功执行, 返回新的文件描述符。失败, 返回-1。

```
int old, int new;  
int ret = syscall(SYS_dup3, old, new, 0);
```

将dup的调用号修改为23后, dup这个测试样例应该可以通过了

主要看dup2这个测试样例, 测试内容如下

```
void test_dup2(){  
    TEST_START(__func__);  
    int fd = dup2(STDOUT, 100);  
    assert(fd != -1);  
    const char *str = "  from fd 100\n";  
    write(100, str, strlen(str));  
    TEST_END(__func__);  
}
```

15.read/write

在文档中查看SYS_read和SYS_write

```
#define SYS_read 63
```

功能: 从一个文件描述符中读取;

输入:

fd: 要读取文件的文件描述符。
buf: 一个缓存区, 用于存放读取的内容。
count: 要读取的字节数。

返回值: 成功执行, 返回读取的字节数。如为0, 表示文件结束。错误, 则返回-1。

```
int fd, void *buf, size_t count;  
ssize_t ret = syscall(SYS_read, fd, buf, count);
```

```
#define SYS_write 64  
功能: 从一个文件描述符中写入;
```

输入:
fd: 要写入文件的文件描述符。
buf: 一个缓存区, 用于存放要写入的内容。
count: 要写入的字节数。

返回值: 成功执行, 返回写入的字节数。错误, 则返回-1。

```
int fd, const void *buf, size_t count;  
ssize_t ret = syscall(SYS_write, fd, buf, count);
```

sys_read和sys_write已有实现, 只需要把调用号修改即可

16.pipe

运行测试样例, 发现缺少59号系统调用

```
#define SYS_pipe2 59  
功能: 创建管道;  
  
输入:  
fd[2]: 用于保存2个文件描述符。其中, fd[0]为管道的读出端, fd[1]为管道的写入端。  
  
返回值: 成功执行, 返回0。失败, 返回-1。  
  
int fd[2];  
int ret = syscall(SYS_pipe2, fd, 0);
```

而xv6-k210中已有SYS_pipe, 将其对应的调用号修改为59即可

17.open

写完上述几个测试点, open测试点自然通过

18.fstat

运行fstat测试程序, 发现缺少80号系统调用。

而sysnum.h中已有SYS_fstat，修改为80即可

测试点出现类似"fstat: dev: 1954047348, inode: 0, mode: 0, nlink: 0, size: 52, atime: 0, mtime: 0, ctime: 0"的输出即代表通过测试点

19.getdents

运行getdents测试样例，发现缺少61号系统调用

```
#define SYS_getdents64 61
```

功能：获取目录的条目；

输入：

fd：所要读取目录的文件描述符。

buf：一个缓存区，用于保存所读取目录的信息。缓存区的结构如下：

```
struct dirent {  
    uint64 d_ino;    // 索引结点号  
    int64 d_off;     // 到下一个dirent的偏移  
    unsigned short d_reclen;    // 当前dirent的长度  
    unsigned char d_type;    // 文件类型  
    char d_name[];    // 文件名  
};
```

len：buf的大小。

返回值：成功执行，返回读取的字节数。当到目录结尾，则返回0。失败，则返回-1。

```
int fd, struct dirent *buf, size_t len  
int ret = syscall(SYS_getdents64, fd, buf, len);
```

20.getcwd

查阅文档，发现getcwd的系统调用号应为17

```
#define SYS_getcwd 17
```

功能：获取当前工作目录；

输入：

- char *buf：一块缓存区，用于保存当前工作目录的字符串。当buf设为NULL，由系统来分配缓存区。
- size：buf缓存区的大小。

返回值：成功执行，则返回当前工作目录的字符串的指针。失败，则返回NULL。

```
char *buf, size_t size;  
long ret = syscall(SYS_getcwd, buf, size);
```


getcwd同样已有自己的实现，将getcwd系统调用号修正后，在原有的getcwd的基础上增加对size参数的判断

21.chdir

运行测试样例发现缺少34和49两个系统调用号

```
#define SYS_mkdirat 34
```

功能：创建目录；

输入：

- dirfd: 要创建的目录所在的目录的文件描述符。
- path: 要创建的目录的名称。如果path是相对路径，则它是相对于dirfd目录而言的。如果path是相对路径，且dirfd的值为AT_FDCWD，则它是相对于当前路径而言的。如果path是绝对路径，则dirfd被忽略。
- mode: 文件的所有权描述。详见`man 7 inode`。

返回值：成功执行，返回0。失败，返回-1。

```
int dirfd, const char *path, mode_t mode;  
int ret = syscall(SYS_mkdirat, dirfd, path, mode);
```

```
#define SYS_chdir 49
```

功能：切换工作目录；

输入：

- path: 需要切换到的目录。

返回值：成功执行，返回0。失败，返回-1。

```
const char *path;  
int ret = syscall(SYS_chdir, path);
```

其中mkdir已经被实现，实验需要的mkdirat可以在mkdir的基础上修改，主要是对参数的判断不同

而chdir只需要修改调用号即可

22.mkdir

完成了chdir自然就完成了mkdir

23.unlink

查阅相应文档，发现unlink对应如下要求

```
#define SYS_unlinkat 35
```

功能：移除指定文件的链接(可用于删除文件)；

输入：

- dirfd: 要删除的链接所在的目录。
- path: 要删除的链接的名字。如果path是相对路径，则它是相对于dirfd目录而言的。如果path是相对路径，且dirfd的值为AT_FDCWD，则它是相对于当前路径而言的。如果path是绝对路径，则dirfd被忽略。
- flags: 可设置为0或AT_REMOVEDIR。

返回值：成功执行，返回0。失败，返回-1。

```
int dirfd, char *path, unsigned int flags;
syscall(SYS_unlinkat, dirfd, path, flags);
```

阅读后发现unlinkat需要的功能与已经实现的remove功能类似，考虑在remove的基础上实现unlinkat（当然添加系统调用号这个流程不能少）

24.yield

做完上述测试样例，应该可以通过

25.sleep

运行测试样例，发现缺少调用号101

```
#define SYS_nanosleep 101
```

功能：执行线程睡眠，sleep()库函数基于此系统调用；

输入：睡眠的时间间隔；

```
struct timespec {
    time_t tv_sec;          /* 秒 */
    long   tv_nsec;        /* 纳秒，范围在0~999999999 */
};
```

返回值：成功返回0，失败返回-1；

```
const struct timespec *req, struct timespec *rem;
int ret = syscall(SYS_nanosleep, req, rem);
```

26.mmap/munmap

运行mmap这个测试样例，发现缺少以下调用号：80、222

```
#define SYS_fstat 80
```

功能：获取文件状态；

输入：

fd: 文件句柄；

kst: 接收保存文件状态的指针；

```
struct kstat {
    dev_t st_dev;
```

```
ino_t st_ino;
mode_t st_mode;
nlink_t st_nlink;
uid_t st_uid;
gid_t st_gid;
dev_t st_rdev;
unsigned long __pad;
off_t st_size;
blksize_t st_blksize;
int __pad2;
blkcnt_t st_blocks;
long st_atime_sec;
long st_atime_nsec;
long st_mtime_sec;
long st_mtime_nsec;
long st_ctime_sec;
long st_ctime_nsec;
unsigned __unused[2];
};
返回值：成功返回0，失败返回-1;

int fd;
struct kstat kst;
int ret = syscall(SYS_fstat, fd, &kst);
```

```
#define SYS_mmap 222
```

功能：将文件或设备映射到内存中；

输入：

start：映射起始位置，

len：长度，

prot：映射的内存保护方式，可取：PROT_EXEC，PROT_READ，PROT_WRITE，PROT_NONE

flags：映射是否与其他进程共享的标志，

fd：文件句柄，

off：文件偏移量；

返回值：成功返回已映射区域的指针，失败返回-1；

```
void *start, size_t len, int prot, int flags, int fd, off_t off
long ret = syscall(SYS_mmap, start, len, prot, flags, fd, off);
```

Gitlab与希冀平台提交代码

在本地通过了若干测试后，可以尝试在评测系统中测试

首先在服务器中安装git

```
sudo apt-get update
sudo apt-get install git
```

然后在服务器中配置全局的名字与邮箱，这里两个信息仅仅起到标志的作用

```
git config --global user.name "name"
git config --global user.email "email@address"
```

输入完成后可以发现多出了.gitconfig文件

在ubuntu根目录下git clone新建的仓库

```
git clone address
```

进入clone下来的文件夹（例如xv6）

将所有文件添加到Git仓库暂存区

```
git add .
```

提交更改，最后的内容理论上随意，但最好简单记录本次提交的主要内容

```
git commit -m "first commit"
```

由于是clone下来的仓库，不需要再关联仓库，同时只有一个分支，所以可以直接push

```
git push
```

本次包含的所有文件比较多，可以一批一批复制到对应文件夹中推送

总结

在这次实验的过程中，我学到了很多知识，也把很多尚未完全掌握的技能进行了巩固。

通过这次实验，我对 xv6 内核的系统调用实现机制有了更深入的了解；通过这次实验，我更加熟悉了远程服务器操作、git操作等技能；通过这次实验中对Makefile的修改，我对管理项目构建这一方面也有了更深的理解。