

区域赛文档说明

项目简介

本项目是一个基于 xv6 实现的内核，参考代码地址：<https://github.com/HUST-OS/XV6-k210>

运行环境

```
gcc-riscv64-linux-gnu  
qemu-7.0.0
```

设计思路

OS 结构功能简述分为：

进程管理 (Process Management)：参考 k210，基于 xv6 实现进程管理。

内存管理 (Memory Management)：实现了基于页表和内核地址空间的物理内存分配等

文件系统 (File System)：FAT32 文件系统基础上实现系统调用。

中断机制 (Trap Mechanism)：实现了时钟中断等中断机制。

系统调用 (System Call)：见下文

系统调用实现

1.1 添加系统调用

操作系统大赛的要求，首先在 sysnum.h 中为每个系统调用添加正确的系统调用号，接着在 kernel/syscall.c 中定义函数原型，然后在 static uint64 中添加系统调用函数的函数指针，最后在系统调用定义中为每个系统调用号加上名字。至此，我们的完成了系统调用的添加。

2.1 系统调用的实现

2.1.1

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。
- 大小：`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);
```

```

uint64
sys_getcwd(void)
{
    uint64 addr;
    if (argaddr(0, &addr) < 0)
        return -1;

    struct dirent *de = myproc()->cwd;
    char path[FAT32_MAX_PATH];
    char *s;
    int len;

    if (de->parent == NULL) {
        s = "/";
    } else {
        s = path + FAT32_MAX_PATH - 1;
        *s = '\\0';
        while (de->parent) {
            len = strlen(de->filename);
            s -= len;
            if (s <= path)          // can't reach root "/"
                return -1;
            strncpy(s, de->filename, len);
            *--s = '/';
            de = de->parent;
        }
    }

    // if (copyout(myproc()->pagetable, addr, s, strlen(s) + 1) <
0)
    if (copyout2(addr, s, strlen(s) + 1) < 0)
        return -1;

    return 0;
}

```

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);
```

```

uint64
sys_pipe(void)
{
    uint64 fdarray; // user pointer to array of two integers
    struct file *rf, *wf;
    int fd0, fd1;
    struct proc *p = myproc();

    if(argaddr(0, &fdarray) < 0)
        return -1;
    if(pipealloc(&rf, &wf) < 0)
        return -1;
    fd0 = -1;
    if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
        if(fd0 >= 0)
            p->ofile[fd0] = 0;
        fileclose(rf);
        fileclose(wf);
        return -1;
    }
    // if(copyout(p->pagetable, fdarray, (char*)&fd0, sizeof(fd0))
    < 0 ||
    //   copyout(p->pagetable, fdarray+sizeof(fd0), (char *)&fd1,
    sizeof(fd1)) < 0){
        if(copyout2(fdarray, (char*)&fd0, sizeof(fd0)) < 0 ||
            copyout2(fdarray+sizeof(fd0), (char *)&fd1, sizeof(fd1)) <
0){
            p->ofile[fd0] = 0;
            p->ofile[fd1] = 0;
            fileclose(rf);
            fileclose(wf);
            return -1;
        }
    }
    return 0;
}

```

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);

```
uint64
sys_dup(void)
{
    struct file *f;
    int fd;

    if(argfd(0, 0, &f) < 0)
        return -1;
    if((fd=fdalloc(f)) < 0)
        return -1;
    filedup(f);
    return fd;
}
```

2.14

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);
```

```

struct sys_dirent {
uint64 d_ino; // 索引结点号
long 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
};

uint64
sys_getdents64(void)
{
int fd, len;
uint64 buf;
struct file* f;
if(argfd(0, &fd, &f) < 0 || argaddr(1, &buf) < 0 || argint(2,
&len) < 0)
return -1;
return getdents64(f, buf, len);
}

int
getdents64(struct file *f, uint64 addr, int len)
{
if(f->readable == 0 || !(f->ep->attribute & ATTR_DIRECTORY))
return -1;
struct dirent de;
struct sys_dirent st;
int count = 0;
int ret;
elock(f->ep);
while ((ret = enext(f->ep, &de, f->off, &count)) == 0) {
f->off += count * 32;
}
9
eunlock(f->ep);
if (ret == -1)
return 0;
f->off += count * 32;
strncpy(st.d_name, de.filename, sizeof(de.filename));
// if(copyout(p->pagetable, addr, (char *)&st, sizeof(st)) < 0)
if( (copyout2(addr, (char *)&st, sizeof(st)) < 0) || f->off >
len)
return -1;
return f->off;

```


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。
- 计数: 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);
```

```

uint64
sys_read(void)
{
    struct file *f;
    int n;
    uint64 p;

    if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argaddr(1, &p) <
0)
        return -1;
    return fileread(f, p, n);
}

```

2.16

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);
```

```
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);
```

```

uint64
sys_fstat(void)
{
    struct file *f;
    uint64 st; // user pointer to struct stat

    if(argfd(0, 0, &f) < 0 || argaddr(1, &st) < 0)
        return -1;
    return filestat(f, st);
}

static struct dirent*
create(char *path, short type, int mode)
{
    struct dirent *ep, *dp;
    char name[FAT32_MAX_FILENAME + 1];

    if((dp = enameparent(path, name)) == NULL)
        return NULL;

    if (type == T_DIR) {
        mode = ATTR_DIRECTORY;
    } else if (mode & O_RDONLY) {
        mode = ATTR_READ_ONLY;
    } else {
        mode = 0;
    }

    elock(dp);
    if ((ep = ealloc(dp, name, mode)) == NULL) {
        eunlock(dp);
        eput(dp);
        return NULL;
    }

    if ((type == T_DIR && !(ep->attribute & ATTR_DIRECTORY)) ||
        (type == T_FILE && (ep->attribute & ATTR_DIRECTORY))) {
        eunlock(dp);
        eput(ep);
        eput(dp);
        return NULL;
    }
}

```

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)

```

sys_clone(void)
{
    uint64 stack;
    int flag;
    argint(0, &flag);
    argaddr(1, &stack);
    return fork(stack);
}

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

```

2.19

define SYS_execve 221

- 功能：执行一个指定的程序；
- 输入：path：待执行程序路径名称，
 - argv：程序的参数，
 - envp：环境变量的数组指针
- 返回值：成功不返回，失败返回-1；

```

const char *path, char *const argv[], char *const envp[];
int ret = syscall(SYS_execve, path, argv, envp);

```

```

uint64
sys_exec(void)
{
    char path[FAT32_MAX_PATH], *argv[MAXARG];
    int i;
    uint64 uargv, uarg;

    if(argstr(0, path, FAT32_MAX_PATH) < 0 || argaddr(1, &uargv) <
0){
        return -1;
    }
    memset(argv, 0, sizeof(argv));
    for(i=0;; i++){
        if(i >= NELEM(argv)){
            goto bad;
        }
        if(fetchaddr(uargv+sizeof(uint64)*i, (uint64*)&uarg) < 0){
            goto bad;
        }
        if(uarg == 0){
            argv[i] = 0;
            break;
        }
        argv[i] = kalloc();
        if(argv[i] == 0)
            goto bad;
        if(fetchstr(uarg, argv[i], PGSIZE) < 0)
            goto bad;
    }

    int ret = exec(path, argv);

    for(i = 0; i < NELEM(argv) && argv[i] != 0; i++)
        kfree(argv[i]);

    return ret;

bad:
    for(i = 0; i < NELEM(argv) && argv[i] != 0; i++)
        kfree(argv[i]);
    return -1;
}

```

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;
 - 选项：选项：WNOHANG，WUNTRACED，WCONTINUE;
- 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);
```



```

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

uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    if(growproc(n) < 0)
        return -1;
    return addr;
}

```

2.21

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);
```

```
sys_getppid(void)
{
    return (myproc()->parent)->pid;
}

uint64
sys_fork(void)
{
    return fork(0);
}
```

2.23

define SYS_getpid 172

- 功能：获取进程 ID;
- 输入：系统调用 ID;
- 返回值：成功返回进程 ID;

pid_t ret = syscall(SYS_getpid);

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

uint64
sys_getppid(void)
{
    return (myproc()->parent)->pid;
}

uint64
sys_fork(void)
{
    return fork(0);
}
```

2.24

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);
```

```

sys_rename(void)
{
    char old[FAT32_MAX_PATH], new[FAT32_MAX_PATH];
    if (argstr(0, old, FAT32_MAX_PATH) < 0 || argstr(1, new,
FAT32_MAX_PATH) < 0) {
        return -1;
    }

    struct dirent *src = NULL, *dst = NULL, *pdst = NULL;
    int srclock = 0;
    char *name;
    if ((src = ename(old)) == NULL || (pdst = enameparent(new,
old)) == NULL
        || (name = formatname(old)) == NULL) {
        goto fail;          // src doesn't exist || dst parent
doesn't exist || illegal new name
    }
    for (struct dirent *ep = pdst; ep != NULL; ep = ep->parent) {
        if (ep == src) {    // In what universe can we move a
directory into its child?
            goto fail;
        }
    }

    uint off;
    elock(src);          // must hold child's lock before acquiring
parent's, because we do so in other similar cases
    srclock = 1;
    elock(pdst);
    dst = dirlookup(pdst, name, &off);
    if (dst != NULL) {
        eunlock(pdst);
        if (src == dst) {
            goto fail;
        } else if (src->attribute & dst->attribute & ATTR_DIRECTORY)
{
            elock(dst);
            if (!isdirempty(dst)) {    // it's ok to overwrite an empty
dir
                eunlock(dst);
                goto fail;
            }
            elock(pdst);

```

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);
```

```
struct tms *tbuf = (struct tms *)addr;
struct proc *p = myproc();
if (addr){
    if(copyout(p->pagetable, (uint64)&(tbuf -> tms_untime),
        (char *)&(p->untime), sizeof(p->untime)) < 0 ||
        copyout(p->pagetable, (uint64)&(tbuf -> tms_stime),
        (char *)&(p->stime), sizeof(p->stime)) < 0 ||
        copyout(p->pagetable, (uint64)&(tbuf -> tms_cutime),
        (char *)&(p->cutime), sizeof(p->cutime)) < 0 ||
        copyout(p->pagetable, (uint64)&(tbuf -> tms_cstime),
        (char *)&(p->cstime), sizeof(p->cstime)) < 0){
        return -1;
    }
}
```

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);
```

```

struct utsname {
char sysname[65];
char nodename[65];
char release[65];
char version[65];
char machine[65];
char domainname[65];
};
uint64
sys_uname(void)
{
uint64 addr;
if(argaddr(0, &addr) < 0)
return -1;
struct utsname* uts = (struct utsname*)addr;
const char error[16] = "unavailable";
strncpy(uts->sysname, error, 12);
strncpy(uts->nodename, error, 12);
strncpy(uts->release, error, 12);
strncpy(uts->version, error, 12);
strncpy(uts->machine, error, 12);
strncpy(uts->domainname, error, 12);
return 0;
}

```

2.28

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);
```

```
uint64
sys_sched_yield(void)
{
    yield();
    return 0;
}

void
yield(void)
{
    struct proc *p = myproc();
    acquire(&p->lock);
    p->state = RUNNABLE;
    sched();
    release(&p->lock);
}
```

2.29

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);
```

```

uint64 sys_gettimeofday(void) {
    uint64 addr;
    if(argaddr(0, &addr)<0){
        return -1;
    }
    struct timespec *tbuf = (struct timespec *)addr;
    struct proc *p = myproc();
    uint64 sec = ticks;
    uint64 usec = 0;
    if(addr){
        if(copyout(p->pagetable, (uint64)&(tbuf -> sec),
            (char *)&sec, sizeof(sec)) < 0 ||
            copyout(p->pagetable, (uint64)&(tbuf -> usec),
            (char *)&usec, sizeof(usec)) < 0){
            return -1;
        }
    }
    return 0;
}

```

2.30

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);

```



```
sys_sleep(void)
{
    int n;
    uint ticks0;

    if(argint(0, &n) < 0)
        return -1;
    acquire(&tickslock);
    ticks0 = ticks;
    while(ticks - ticks0 < n){
        if(myproc()->killed){
            release(&tickslock);
            return -1;
        }
        sleep(&ticks, &tickslock);
    }
    release(&tickslock);
    return 0;
}
```