操作系统赛-内核实现赛初赛实现文档

一、设计思路

1、 针对比赛的要求，选用的开源软件

由于比赛要求使用FAT32文件系统，并且需要在K210开发板上运行，能驱动K210开发板上的SD卡。因此，我们组分析得出结论：需要使用到一款满足条件的开源操作系统，基于开源软件进行内核实现的开发，从零开始实现难度过大，也不利于在初赛阶段完成内核系统调用实现的任务。故我们使用了xv6-k210开源操作系统，该系统能支持K210上的FAT32格式的SD卡，还能使用QEMU进行调试，对我们的开发十分有利。（该系统是一个混合内核操作系统，采用了 MIT License开源协议，由MIT大学研发，作为6.s081操作系统课程的教学用操作系统，由华中科技大学的团队负责移植到k210平台。）

2、 对xv6-K210已有的实现进行的分析

（1） 在系统调用方面：我们需要对已有的系统调用进行移植、修改，以符合文档的系统调用的接口要求等需求。

（2） 在SBI与RISC-V汇编程序方面：xv6-k210基本使用了C语言开发，涉及到汇编的范畴，基本都在BootLoader层面，在Trap部分和进程上下文切换部分有少量的汇编程序以及C内嵌汇编，以辅助我们访问寄存器，使用C函数执行特权指令，这些汇编语言部分对我们的实现系统调用关系不大，能看懂即可。因此，我们不需要修改已有的汇编语言指令，但需要理解汇编指令，以及xv6-k210在RISC-V的使用汇编语言的操作。

（3） 在链接与二进制编码方面：xv6-k210已经实现了链接脚本，由make程序自动调用，也不需要我们做太多的修改。对于用户态程序的启动，xv6-k210是直接将第一个用户态程序的二进制编码initcode复制到第一个用户态进程的用户栈区域的，这个initcode再启动SD卡上的init程序，然后init程序再打开shell。因此，我们需要修改init程序，使init程序能够执行FAT32的SD卡上的测试用例程序，编译后再把init程序dump二进制出来，替换xv6-k210的initcode的机器码，像启动initcode一样直接启动init程序。

3、 对系统调用测试用例进行的分析

系统调用测试用例里面规定了一些数据结构的定义，还有系统调用的调用方法，汇

编代码。我们可以参考系统调用测试用例的源码、汇编实现，以及测试用例标准库中的 UNIX 接口调用的实现，来实现我们的系统调用。

4、 xv6-k210 项目目录结构的分析

xv6-k210 的内核在 kernel 文件夹下，我们主要需要关注这个文件夹下的文件，还有用户态程序 xv6-user 下面的 init 程序的时间。内核文件结构详见附录 1：kernel 文件夹结构及解释。

二、具体实现

自定义系统调用主要在 kernel 文件夹下的 lcc_syscall.c 和 include/lcc_syscall.h 中

1、 修改 init 程序和内核，使内核能自动运行 SD 卡上的测试用例

（1） init 程序的修改

修改的核心在于在 init 程序的根进程中，利用 for 循环，遍历待执行的测试用例列表，逐个执行测试用例。使用 fork 系统调用，fork 出子进程，在子进程中执行测试用例，避免执行测试用例程序时，阻塞 init 程序的运行，同时使得测试用例执行程序能得到 init 程序的监控管理。核心代码片段如下。

```
char *argv[] = {"sh", 0};
char *arg[] = {"mount", "./mnt", "/dev/vda2"};
char *to_exec_files[] = {
    "times", "clone", "pipe", "openat", "chdir", "dup", "dup2",
    "getpid", "getppid", "write", "getcwd", "wait",
    "read", "exit", "open", "execve", "fork",
     "mkdir_", "close", "uname",
    "fstat", "waitpid", "getdents", "gettimeofday", "yield", "mmap", "munmap", "brk",
    "unlink",
};
```

```
for (int i = 0; i < 29; i++) //execve testcase one by one
{
  if (!fork())
  {
    exec(to_exec_files[i], argv);
    exit(0);
  }
  else
  {
    wait(0);
  }
}
```

（2） 将 init 程序硬编码到内核中

因为 xv6 内核运行第一个程序是通过加载一个机器指令编码组成的数组，运行这个数组中数字所代表的机器指令，且由于测试用的 SD 卡中没有 init 程序，原来的 init 程序并没有自动运行测试用例的功能。所以，要运行运行测试用例的程序，我们需要重新编写 init 程序，用 init 程序来运行测试用例，之后，需要提取其机器指令编码，覆盖这个数组中的数据。Xv6 运行第一个用户程序时，就不需要到 SD 卡上找 init 程序了。具体细节如下：

使用命令 make build 编译后，找到 xv6-user 中的_init 程序，使用命令

riscv64-unknown-elf-objcopy -S -O binary _init initcode

生成 init 程序的二进制文件 initcode，使用工具按字节读取 initcode 的十六进制编码，并复制到该数组中。这个数组中的数据就是 init 程序的机器指令

```
unsigned char testcode[] = {
0x01, 0x11, 0x06, 0xec, 0x22, 0xe8, 0x26, 0xe4, 0x00, 0x10, 0x4a, 0xe0, 0x01, 0x46, 0x85, 0x45,
0x09, 0x45, 0x97, 0x00, 0x00, 0x00, 0xe7, 0x80, 0xc0, 0x41, 0x01, 0x45, 0x97, 0x00, 0x00, 0x00,
0xe7, 0x80, 0x60, 0x3d, 0x01, 0x45, 0x97, 0x00, 0x00, 0x00, 0xe7, 0x80, 0xc0, 0x3c, 0x97, 0x00,
0x00, 0x00, 0xe7, 0x80, 0x20, 0x34, 0x63, 0x4c, 0x05, 0x04, 0xaa, 0x84, 0x21, 0xe1, 0x59, 0x49,
0x09, 0xa8, 0x85, 0x24, 0x01, 0x45, 0x97, 0x00, 0x00, 0x00, 0xe7, 0x80, 0xe0, 0x33, 0x63, 0x80,
0x24, 0x05, 0x97, 0x00, 0x00, 0x00, 0xe7, 0x80, 0xe0, 0x31, 0x65, 0xf5, 0x97, 0x15, 0x00, 0x00,
0x93, 0x85, 0x45, 0x9e, 0x8e, 0x04, 0xae, 0x94, 0x88, 0x68, 0x97, 0x00, 0x00, 0x00, 0xe7, 0x80,
0x60, 0x35, 0x01, 0x45, 0x97, 0x00, 0x00, 0x00, 0xe7, 0x80, 0x60, 0x30, 0x01, 0x45, 0x97, 0x00,
0x00, 0x00, 0xe7, 0x80, 0x60, 0x30, 0xe3, 0x8b, 0xa4, 0xfe, 0xe3, 0x59, 0x05, 0xfe, 0x05, 0x45,
0x97, 0x00, 0x00, 0x00, 0xe7, 0x80, 0xa0, 0x2e, 0x41, 0x11, 0x22, 0xe4, 0x00, 0x08, 0xaa, 0x87,
0x85, 0x05, 0x03, 0xc7, 0xf5, 0xff, 0x85, 0x07, 0xa3, 0x8f, 0xe7, 0xfe, 0x75, 0xfb, 0x22, 0x64,
```

在用户态初始化函数中，程序会初始化第一个用户进程，并将上面的数组载入用户栈，之后，CPU 就会根据编码，运行硬编码到内核的 init 程序，而不用到 SD 卡中寻找。

```
void test_proc_init()
{
  struct proc *p;

  printf("[test_proc]start test_proc init...\n");
  p = allocproc();
  uvminit(p->pagetable, (uchar *)testcode, file_size());

  p->sz = PGSIZE;
  p->trapframe->epc = 0x0;
  p->trapframe->sp = PGSIZE;
  safestrcpy(p->name, "test_code", sizeof(p->name));

  //p->cwd = ename("/");

  p->state = RUNNABLE;
  p->tmask = 0;
  release(&p->lock);

  initproc = p;
  printf("[test_proc]test_proc init done\n");
}
```

2、 自定义系统调用的添加方法

（1） 首先，我在 kernel/include 文件夹下的 syscall.h 定义一个新的系统调用名字和系统调用号，修改原有的系统调用号范围到 300 至 325 之间，新加的在 400 以上，这样，原来和文档中相同的系统调用号就不会相互干扰，使得系统调用号的排序和区域更加清晰，减少了不必要的麻烦；然后，我在 kernel/syscall.c 下定义外部处理函数原型，如下图所示

```
/*------ add by luchangcheng ------*/
extern uint64 sys_lcc_getcwd(void);
extern uint64 sys_lcc_clone(void);
extern uint64 sys_lcc_getppid(void);
extern uint64 sys_lcc_openat(void);
extern uint64 sys_lcc_dup3(void);
extern uint64 sys_lcc_wait(void);
extern uint64 sys_lcc_mkdirat(void);
extern uint64 sys_lcc_yield(void);
extern uint64 sys_lcc_times(void);
extern uint64 sys_lcc_brk(void);
extern uint64 sys_lcc_uname(void);
extern uint64 sys_lcc_fstat(void);

/*--------------------------------*/
```

之后，在系统调用散转表“static uint64 (*syscalls[])(void)”里头，添加指向系统调用处理函数的函数指针，如下图所示

```
/*-------- add by luchangcheng ---------*/
[SYS_lcc_getcwd] sys_lcc_getcwd,
[SYS_lcc_clone] sys_lcc_clone,
[SYS_lcc_getppid] sys_lcc_getppid,
[SYS_lcc_openat] sys_lcc_openat,
[SYS_lcc_dup3] sys_lcc_dup3,
[SYS_lcc_wait] sys_lcc_wait,
[SYS_lcc_mkdirat] sys_lcc_mkdirat,
[SYS_lcc_yield] sys_lcc_yield,
[SYS_lcc_times] sys_lcc_times,
[SYS_lcc_brk] sys_lcc_brk,
[SYS_lcc_uname] sys_lcc_uname,
[SYS_lcc_fstat] sys_lcc_fstat,
```

最后，我在系统调用定义里面，给系统调用号加上名字，如下图所示

```
/*-------- add by luchangcheng ---------*/
[SYS_lcc_getcwd] "lcc_getcwd",
[SYS_lcc_clone] "lcc_clone",
[SYS_lcc_getppid] "lcc_getppid",
[SYS_lcc_openat] "lcc_openat",
[SYS_lcc_dup3] "lcc_dup3",
[SYS_lcc_wait] "lcc_wait",
[SYS_lcc_mkdirat] "lcc_mkdirat",
[SYS_lcc_yield] "lcc_yield",
[SYS_lcc_times] "lcc_times",
[SYS_lcc_brk] "lcc_brk",
[SYS_lcc_uname] "lcc_uname",
[SYS_lcc_fstat] "lcc_fstat",
```

（2） 处理系统调用号的细节：

有些系统调用是系统本来就实现好的，有些是需要我们自己处理实现的，因此，我设计了一个跳转函数，如下图所示

```
/*------------------------ add by luchangcheng
inline void change_syscall_num(int *num)
{
  switch (*num)
  {
  case 64:
    *num = 316; //write, success
    break;
  case 93:
    *num = 302; //exit, success
    break;
  case 49:
    *num = 309; //chdir, success
    break;
  case 57:
    *num = 321; //close, success
    break;
  case 17:
    *num = 401; //getcwd, success
    break;
```

在系统调用总控函数处理时调用。获取到系统调用号之后，把测试用例传来的系统调用号，转换到我们自己定义的系统调用号，如下图所示

```
num = p->trapframe->a7;
change_syscall_num(&num);
```

（3） 添加并编写自己的系统调用处理函数：

我在 kernel/lcc_syscall.c 和 kernel/include/lcc_syscall.h 两个文件中，分别实现了自己的系统调用处理函数和函数原型、所需的数据结构，并添加到 Makefile 中，在编译时自动编译这两个文件，并链接到内核中

3、 SYS_getcwd 系统调用的实现

创建处理函数 sys_lcc_getcwd，参照原有的实现，从 FAT32 文件系统中获取到当前进程所属文件的目录项，自下而上，从子目录往父目录迭代，获取文件名，并拼接到一个自定义的内核缓冲区 s 中，直到根目录

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

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

if (de->parent == 0)
{
    s = "/";
}
else
{
    s = path + MAXPATH - 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;
    }
}
```

然后，根据传入的用户缓冲区参数是否为 NULL，将内核缓冲区 s 的数据复制到用户空间。如果为 NULL，则获取一个页帧，并根据大小，获取一段用户内存空间，使用 copyout 函数，把数据复制到分配的用户内存空间中

```
if (addr != 0)
{
    if (copyout(myproc()->pagetable, addr, s, strlen(s) + 1) < 0)
        return -1;
}
else
{
    pagetable_t page = uvmcreate();
    addr = uvmalloc(page, 0, n);
    if (copyout(page, 0, s, strlen(s) + 1) < 0)
        return -1;
}
```

4、 SYS_pipe2 系统调用的实现

获取到数组基地址，给当前进程分配一个管道（包含了两个打开的文件），并

分配文件描述符

```
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;
}
```

最后，把分配的管道文件复制到传入的数组中

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

其中，管道的分配只需请求一个页帧，在页帧上分配两个文件，并设置好文件的读写权限，以及相关描述信息即可。

```
pi->readopen = 1;
pi->writeopen = 1;
pi->nwrite = 0;
pi->nread = 0;
initlock(&pi->lock, "pipe");
(*f0)->type = FD_PIPE;
(*f0)->readable = 1;
(*f0)->writable = 0;
(*f0)->pipe = pi;
(*f1)->type = FD_PIPE;
(*f1)->readable = 0;
(*f1)->writable = 1;
(*f1)->pipe = pi;
return 0;
```

5、 SYS_dup 系统调用的实现

找到传入的文件描述符所对应的文件 f，再为 f 另分配一个文件描述符 fd，最后增加文件 f 的引用数目即可

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

Filedup 函数用于增加文件 f 的引用数目，可以通过增加全局打开文件表中对应文件的引用数实现，如下图所示。该操作为原子操作，需要加锁实现，并且打开文件的引用数目不能小于 1

```
acquire(&ftable.lock);
if(f->ref < 1)
  panic("filedup");
f->ref++;
release(&ftable.lock);
return f;
```

6、 SYS_dup3 系统调用的实现

获得当前文件描述符对应的文件 f，检查新文件描述符是否被占用后，赋给当前进程打开文件表中对应的新的文件描述符位置，最后在全局打开文件表中增加对这个文件的引用

```
argint(0, &old_fd);
argint(1, &new_fd);

if (argfd(0, 0, &f) < 0)
    return -1;

p = myproc();
if (p->ofile[new_fd] != 0)
    return -1;

p->ofile[new_fd] = f;
filedup(f);
```

7、 SYS_chdir 系统调用的实现

获取传入的路径对应的 FAT32 目录项 ep，获取其锁后再继续操作，检查是否为目录，非目录的话，释放锁，返回错误

```
if (argstr(0, path, MAXPATH) < 0 || (ep = ename(path)) == 0)
{
  return -1;
}
elock(ep);
if (!(ep->attribute & ATTR_DIRECTORY))
{
  eunlock(ep);
  eput(ep);
  return -1;
}
```

如果合法（即为目录），就把当前进程运行目录改为目标目录，然后再释放锁

```
eunlock(ep);
eput(p->cwd);
p->cwd = ep;
return 0;
```

8、 SYS_openat 系统调用的实现

首先，判断传入的标志是否是创建文件，如果 openat 系统调用需要创建文件，就调用 FAT32 文件系统中创建文件的函数接口。如果文件已经存在了（即调用创建接口返回值为 0），就跳到下面的打开文件部分。

```
if (flags & O_CREATE)
{
    ep = create(filename, T_FILE);
    if (ep == 0)
    {
        goto exists;
    }
}
```

如果 openat 系统调用不是要打开文件，则从文件系统中获取打开目录的目录项，对打开的目录项进行加锁，判断打开的目录项与传入的标志位是否矛盾

```
exists:
if ((ep = ename(filename)) == 0)
{
    return -1;
}
elock(ep);
if (((ep->attribute & ATTR_DIRECTORY) && flags != O_RDONLY) && flags != O_DIRECTORY)
{
    eunlock(ep);
    eput(ep);
    return -1;
}
```

为打开的目录项在打开文件表中分配文件空间和文件描述符，设置打开文件的相关

属性，解锁打开的目录项，返回打开的文件描述符

```
if ((f = filealloc()) == 0)
{
    eunlock(ep);
    eput(ep);
    return -1;
}

f->type = FD_ENTRY;
f->off = 0;
f->ep = ep;
f->readable = !(flags & O_WRONLY);
f->writable = (flags & O_WRONLY) || (flags & O_RDWR);

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

eunlock(ep);
```

9、 SYS_close 系统调用的实现

获取传入的当前进程的文件描述符对应的文件 f，设置当前进程打开文件表中对应描述符位置为 NULL，然后关闭文件

```
if (argfd(0, &fd, &f) < 0)
  return -1;
myproc()->ofile[fd] = 0;
fileclose(f);
return 0;
```

关闭文件的过程，即为在全局打开文件表中释放文件占用的资源，清空引用，设置为未使用状态。

```
acquire(&ftable.lock);
if(f->ref < 1)
  panic("fileclose");
if(--f->ref > 0){
  release(&ftable.lock);
  return;
}
ff = *f;
f->ref = 0;
f->type = FD_NONE;
release(&ftable.lock);

if(ff.type == FD_PIPE){
  pipeclose(ff.pipe, ff.writable);
} else if(ff.type == FD_ENTRY){
  eput(ff.ep);
} else if (ff.type == FD_DEVICE) {

}
```

10、 SYS_read 系统调用的实现

解析传入的参数，在当前进程打开文件表中找到文件描述符对应的文件，获取读取到的地址和读取大小，调用 fileread 函数，从目标文件读取指定大小的字节到指定地址

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

读取文件，需要先判断当前文件是否可读，然后根据文件的类型，调用对应的读取函数。

```
if(f->readable == 0)
  return -1;

switch (f->type) {
  case FD_PIPE:
      r = piperead(f->pipe, addr, n);
      break;
  case FD_DEVICE:
      if(f->major < 0 || f->major >= NDEV || !devsw[f->major].read)
        return -1;
      r = devsw[f->major].read(1, addr, n);
      break;
  case FD_ENTRY:
      elock(f->ep);
        if((r = eread(f->ep, 1, addr, f->off, n)) > 0)
          f->off += r;
      eunlock(f->ep);
      break;
  default:
    panic("fileread");
}
```

11、 SYS_write 系统调用的实现

由于文件都有记录读写的属性，这两个属性通常成对出现，因此与 SYS_read 系统调用类似，把对应的读系统调用换成写，检查是否可写即可。

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

return filewrite(f, p, n);
```

```
if(f->writable == 0)
  return -1;

if(f->type == FD_PIPE){
  ret = pipewrite(f->pipe, addr, n);
} else if(f->type == FD_DEVICE){
  if(f->major < 0 || f->major >= NDEV || !devsw[f->major].write)
    return -1;
  ret = devsw[f->major].write(1, addr, n);
} else if(f->type == FD_ENTRY){
  elock(f->ep);
  if (ewrite(f->ep, 1, addr, f->off, n) == n) {
    ret = n;
    f->off += n;
  } else {
    ret = -1;
  }
  eunlock(f->ep);
} else {
  panic("filewrite");
}
```

12、 SYS_mkdirat 系统调用的实现

读取参数后，调用 FAT32 文件系统提供的在 SD 卡上创建新的目录项的接口，创建一个目录，得到新创建目录的目录项，解锁目录项。由于在 FAT32 文件系统中，新建的目录里面并没有文件，只是一个空目录，不应包含任何引用，因此，需要用 eput 函数减少一个对目录项的引用。

```
if (argint(0, &fd) < 0 || argstr(1, path, FAT32_MAX_FILENAME + 1) < 0 || argint(2, &mode) < 0)
    return -1;

if ((ep = create(path, T_DIR)) == 0)
    return -1;

eunlock(ep);
eput(ep);
```

13、 SYS_fstat 系统调用的实现

在 lcc_syscall.h 内，创建 kstat 结构体，并定义其中的变量类型

```
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];
};
```

为 kstat 结构体分配一个页，从解析到的文件描述符中获取到当前文件，再获取当前运行的进程。将当前文件的有关属性和当前进程的有关属性赋值到结构体中。

```
memset(s, 0, sizeof(struct kstat));

s->st_uid = current->pid;
s->st_size = f->ep->file_size;
s->st_dev = f->ep->dev;
s->st_nlink = f->ep->ref;

if (copyout(current->pagetable, addr, (char *)s, sizeof(struct kstat)) < 0)
    return -1;
```

14、 SYS_clone 系统调用的实现

使用 fork 函数，创建出一个新进程。遍历打开进程表，在打开进程表中，找到新的进程 PID 对应的进程数据结构，将获取到的新进程的栈参数，赋给新创建的进程的栈指针 sp 寄存器中

```
if (argint(0, &flag) < 0 || argaddr(1, &stack_top) < 0)
    return -1;

new_pid = fork();
if (stack_top > 0)
{
    for (p = proc; p < &proc[NPROC]; p++)
    {
        if (p->pid == new_pid)
            break;
    }
    p->trapframe->sp = stack_top;
}

return new_pid;
```

15、 SYS_execve 系统调用的实现

处理传入的运行参数列表的指针，在内核栈空间中，还原运行参数列表。

```
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;
}
```

执行 exec 函数，创建新进程，使用 FAT32 文件系统驱动提供的接口，读取 SD 卡中的程序文件，将程序的二进制编码载入新进程的用户栈空间中，然后初始化新进程的寄存器。最后，释放内核态空间中存储的传入运行参数所占用的空间。

```
int ret = exec(path, argv);

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

16、 SYS_wait4 系统调用的实现

获取当前进程的锁，防止错过子进程的退出。设置默认退出码为 0

```
int exit_code = 0;
// hold p->lock for the whole time to avoid lost
// wakeups from a child's exit().
acquire(&p->lock);
```

对已打开的进程表进行遍历，找到当前进程的子进程，而后对找到的子进程加锁，并标记找到了子进程

```
// Scan through table looking for exited children.
havekids = 0;
for (np = proc; np < &proc[NPROC]; np++)
{
  // this code uses np->parent without holding np->lock.
  // acquiring the lock first would cause a deadlock,
  // since np might be an ancestor, and we already hold p->lock.
  if (np->parent == p)
  {
    // np->parent can't change between the check and the acquire()
    // because only the parent changes it, and we're the parent.
    acquire(&np->lock);
    havekids = 1;
```

判断找到的子进程 ID 以及传入的参数 cpid，若 cpid 为-1，或者找到的子进程是目标子进程，在这两个条件满足其一的前提下，判断找到的进程是否是僵死状态

```
if (np->state == ZOMBIE && (target_proc == -1 || (target_proc != -1 && np->pid == target_proc)))
{
  // Found one.
  pid = np->pid;
```

如果符合条件的话，设置退出码，将错误码复制到传入的存储错误码的用户栈地址，然后销毁进程，释放找到的子进程和当前进程的锁，返回子进程的 PID

```
exit_code = (np->xstate << 8) & 0xff00;
if (addr != 0 && copyout(p->pagetable, addr, (char *)&exit_code,
                         sizeof(exit_code)) < 0)
{
  release(&np->lock);
  release(&p->lock);
  return -1;
}
freeproc(np);
release(&np->lock);
release(&p->lock);

//printf("[Debug wait] parent pid is %d, return pid is %d\n", p->pid, pid);
return pid;
```

释放子进程由 freeproc 函数实现，该函数释放对应进程属性拥有的堆空间与分页，设置打开进程表相关属性为未使用状态，如下图所示。

```
static void
freeproc(struct proc *p)
{
  if (p->tms)
    kfree((void *)p->tms);
  if (p->trapframe)
    kfree((void *)p->trapframe);
  p->trapframe = 0;
  if (p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}
```

17、 SYS_exit 系统调用的实现

检查当前进程是否为 0 号进程

```
struct proc *p = myproc();

if (p == initproc)
  panic("init exiting");
```

关闭进程打开的所有文件

```
// Close all open files.
for (int fd = 0; fd < NOFILE; fd++)
{
  if (p->ofile[fd])
  {
    struct file *f = p->ofile[fd];
    fileclose(f);
    p->ofile[fd] = 0;
  }
}
```

在 FAT32 文件系统中，减少对打开文件在 FAT32 文件系统中的引用，唤醒 init 进程

```
eput(p->cwd);
p->cwd = 0;

// we might re-parent a child to init. we can't be precise about
// waking up init, since we can't acquire its lock once we've
// acquired any other proc lock. so wake up init whether that's
// necessary or not. init may miss this wakeup, but that seems
// harmless.
acquire(&initproc->lock);
wakeup1(initproc);
release(&initproc->lock);
```

唤醒当前进程的父进程，通知父进程准备接受退出进程的退出状态码。将当前进程的所有子进程的父进程重定向，全部交给 init 进程（即 0 号进程）进行管理。

```
acquire(&p->lock);
struct proc *original_parent = p->parent;
release(&p->lock);

// we need the parent's lock in order to wake it up from wait().
// the parent-then-child rule says we have to lock it first.
acquire(&original_parent->lock);

acquire(&p->lock);

// Give any children to init.
reparent(p);
```

设置退出状态码，并设置进程为僵死状态，等待父进程的 wait 系统调用，获取了返回值之后，进程才真正的被进行释放。exit 系统调用到最后，会触发一次进程调度，寻找其他可运行的进程。

```
p->xstate = status;
p->state = ZOMBIE;

release(&original_parent->lock);

// Jump into the scheduler, never to return.
sched();
panic("zombie exit");
```

18、 SYS_getpid 和 SYS_getppid 系统调用的实现

SYS_getppid 系统调用只需找到当前进程的父进程的 PID，返回即可

```
uint64 sys_lcc_getppid(void)
{
    struct proc *current = myproc();
    return current->parent->pid;
}
```

SYS_getpid 系统调用只需返回当前进程的 PID 即可

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

19、 SYS_uname 系统调用的实现

在 lcc_syscall.h 头文件中创建 utsname 结构体，用于存储系统信息。

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

在函数调用内实例化一个结构体，并把结构体按字节复制到传入的起始地址处

```
if (argaddr(0, &addr) < 0)
    return -1;

struct utsname uname =
    {"xv6\n", "\tnodename\n", "\tpower by luchangcheng2333\n", "\t0.1\n", "

if (copyout(current->pagetable, addr, (char *)&uname, sizeof(uname)) < 0)
    return -1;
```

20、 SYS_times 系统调用的实现

与 SYS_uname 系统调用类似，获取进程控制块中的计时器数据结构到一个结构体变量之后，复制结构体中的数据到指定的用户栈地址。

```
uint64 addr;
struct proc *current = myproc();
struct tms time = *current->tms;

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

if (copyout(myproc()->pagetable, addr, (char *)&time, sizeof(time)) < 0)
    return -1;

return 0;
```

对有关时间的计时，可以通过修改时钟中断处理函数实现。根据调用时钟中断处理函数 timer_tick 时传入的 trap 类型，判断时钟中断发起前的特权等级状态，给当前进程的时间记录的相关数据结构递增一个时间中断单位，可以记录当前进程在内核/用户态运行的时间。

```c
void timer_tick(int is_kernel_trap)
{
    acquire(&tickslock);

    struct proc *p = myproc();
    if (p != NULL)
    {
        if (is_kernel_trap)
            p->tms->tms_stime++;
        else
            p->tms->tms_utime++;
    }

    ticks++;
    wakeup(&ticks);
    release(&tickslock);
    set_next_timeout();
}
```

传入的 trap 类型可以通过修改中断处理函数实现，该函数在 kerneltrap 和 usertrap 被调用时，会传入不同的参数。在 kerneltrap 处被调用时，传入参数为 1；在 usertrap 处被调用时，为 0。

```c
// Check if it's an external/software interrupt,
// and handle it.
// returns  2 if timer interrupt,
//          1 if other device,
//          0 if not recognized.
int devintr(int is_kernel_trap) {
  uint64 scause = r_scause();

  #ifdef QEMU
  // handle external interrupt
  if ((0x8000000000000000L & scause) && 9 == (scause & 0xff))
  #else
  // on k210, supervisor software interrupt is used
  // in alternative to supervisor external interrupt,
  // which is not available on k210.
```

21、 SYS_gettimeofday 系统调用的实现

使用 rdtime 汇编指令，获取到当前的硬件时间，写入到一个结构体内，再将结构体复制回用户空间，即可

```c
  if (argaddr(0, &addr) < 0)
      return -1;

  timer.sec = r_time();
  timer.usec = timer.sec * 1000000;

  if (copyout(myproc()->pagetable, addr, (char *)&timer, 16) < 0)
      return -1;
```

其中，r_time 的实现如下，调用了获取系统时间的汇编指令

```
// supervisor-mode cycle counter
static inline uint64
r_time()
{
  uint64 x;
  // asm volatile("csrr %0, time" : "=r" (x) );
  // this instruction will trap in SBI
  asm volatile("rdtime %0" : "=r" (x) );
  return x;
}
```

22、 SYS_mmap 和 SYS_munmap 系统调用的实现

sys_mmap 系统调用通过在当前的 sp 寄存器的第 2048 字节，找到一个用户态地址，然后通过读取文件到该地址，并且创建一个数据结构，记录在当前的进程里。实现将文件映射到内存的功能

```
start_addr = p->trapframe->sp - (PGSIZE >> 1);
f->off = 0;
if (fileread(f, start_addr, f->ep->file_size) == -1)
{
    printf("error in mmap\n");
    return -1;
}

for (int i = 0; i < NOFILE; i++)
{
    if (p->map[i] == NULL)
    {
        struct mapedfile *m = kalloc();
        m->f = f;
        m->addr = start_addr;
        p->map[i] = m;
        break;
    }
    else if (i + 1 == NOFILE)
        return -1;
}
```

Sys_munmap 系统调用通过找到当前进程记录下的映射信息，将内存中的数据写回去，并将内存里的对应空间清空即可。

```
if ((buf = kalloc()) == NULL)
    return -1;

for (int i = 0; i < NOFILE; i++)
{
    if (p->map[i] != NULL && p->map[i]->addr == addr)
    {
        filewrite(p->map[i]->f, addr, size);
        if (copyout(myproc()->pagetable, addr, (char *)buf, size) < 0)
            return -1;
        break;
    }
    else if (i + 1 == NOFILE)
        return -1;
}



kfree(buf);
```

23、 SYS_yield 系统调用的实现

申请对当前进程加锁，将当前进程设置为就绪态，并发起一次进程调度，寻找下一个进程运行即可。

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

24、 SYS_unlink 系统调用的实现

获取到文件名之后，使用 ename 函数获取到其对应的目录项，调用 FAT32 文件系统提供的 etrunc 函数，删除文件即可

```
if (argstr(1, path, FAT32_MAX_FILENAME + 1) < 0)
    return -1;

struct dirent *ep = ename(path);
etrunc(ep);
```

25、 SYS_getdents 系统调用的实现

在这里，我参考了 ls 程序以及 sys_dir 系统调用的实现。获取读取目录文件对应的目录项，加锁后使用 enext 函数，在其链表上遍历，直到遍历到链表终点（enext 函数返回值为 1，若为-1 则到目录的结束位置），即获取完目录的第一个文件的信息，然后解锁目录项。

```
struct dirent de;
struct stat st;
int count = 0;
int ret;
elock(f->ep);
while ((ret = enext(f->ep, &de, f->off, &count)) == 0) {  // skip empty entry
    f->off += count * 32;  // parse a file from its linkedlist entry to the end
}
eunlock(f->ep);
if (ret == -1)
    return 0;
```

给打开文件增加上最后最后没加上的偏移量（因为是 FAT32 文件系统，目录项大小为 32 字节，故 count 应乘以 32），设置读取到的信息到定义的 linux_dirent 结构体中，并将结构体的数据复制回在用户空间的测试用例的结构体中。

```
f->off += count * 32; // finish to read a file
estat(&de, &st); // get file's stat

strncpy(buf.d_name, st.name, sizeof(st.name));
buf.d_off = f->off;
buf.d_type = st.type;  // set some related information and then return

if (copyout(myproc()->pagetable, addr, (char*)&buf, sizeof(struct linux_dirent)) < 0)
    return -1;

return f->off;
```

26、 SYS_brk 系统调用的实现

我参考 mmap 的实现，分配空间时，在 sp 的第 1024 字节处的用户空间开始分配数据段。当分配了空间之后，使用 brk(0)函数时，即代表了获取分配空间的基址。

```
if (current->brk_start_space == 0 && addr == 0)   //on init
    current->brk_start_space = current->trapframe->sp - (PGSIZE >> 2);
else if (addr == 0)
{
    // OK
}
```

当使用非零的 brk()函数时，即代表了要拓展分配空间的大小，根据传入的新地址，确定新分配空间的大小。分配空间后，返回旧空间的顶端位置。

```
// calc the size of new space
int size = addr - current->brk_start_space;
char *buf;

if ((buf = kalloc()) == NULL)
    return -1;

if (copyout(myproc()->pagetable, current->brk_start_space + current->brk_space_size, (char *)buf, size)
    return -1;

current->brk_space_size = size;

kfree(buf);
```

27、 SYS_mount 和 SYS_umount 系统调用的实现

经过指导老师的指导与提示，mount 系统调用的核心，是在于将挂载设备根目录和挂载点的目录项关联起来。于是，我创建了一个记录挂载信息，用于关联目录项的结构体，和记录挂载信息的数组。

```
#define MAXMAPFILES 20

struct mt_list {
    struct dirent *origin_ep;
    struct dirent *target_ep;
};
struct mt_list *mount_list[MAXMAPFILES] = {0};
```

获取到挂载设备根目录的目录项之后，就在记录挂载点的数组中找到一个为 NULL 的空位，分配空间，然后记录下挂载信息。

```
for (int i = 0; i < MAXMAPFILES; i++)
{
    if (mount_list[i] == NULL)
    {
        has_free_space = 1; // found a free space
        if ((mount_list[i] = kalloc()) == NULL) // allocate memory first
        {
            printf("No free memory\n");
            return -1;
        }
        mount_list[i]->origin_ep = origin_ep;
        mount_list[i]->target_ep = target_ep; // build the relationship between device and target
        break;
    }
}
```

umount 的实现与 mount 类似，获取到挂载点目录的目录项之后，在挂载信息列表中查找挂载点目录的位置，找到之后，将该位置释放掉即可。

```
for (int i = 0; i < MAXMAPFILES; i++)
{
    if (mount_list[i] != NULL && mount_list[i]->target_ep == ep) // found mount dir
    {
        is_mapfile_found = 1;
        kfree(mount_list[i]); // free space alloc before
        mount_list[i] = NULL;
    }
}
```

需要注意的是，由于磁盘里面并没有/dev/vda2 这个设备，经过在群里的讨论，可以使用挂载 SD 卡的根目录项，因此，在这个时候，我选择了挂载 SD 卡的根目录项来实现 mount 系统调用。

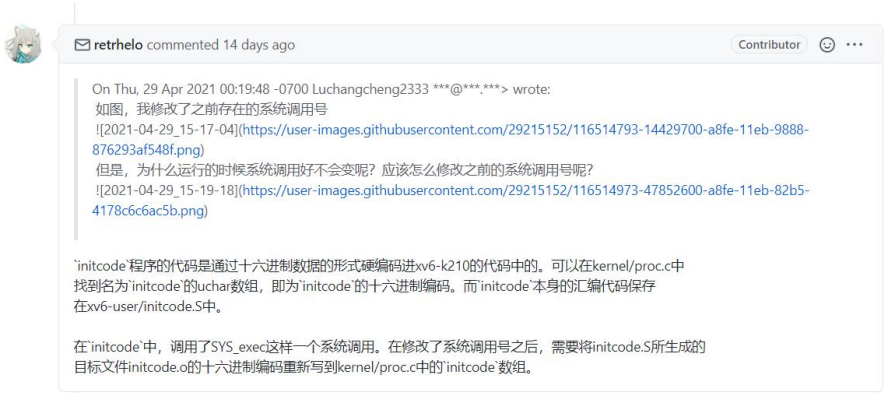
```
if ((origin_ep = ename(special)) == NULL) // find the original dev
{
    origin_ep = &root; // no such device /dev/vda2, replace it with the root of the SD card
    //printf("Origin dev not found!\n");
    //return -1;
}
```

同时，挂载的目标目录需要时一个目录，因此，我们要对挂载目标的目录是否是目录进行一个判断。

```
if ((target_ep = ename(dir)) == NULL || !(target_ep->attribute & ATTR_DIRECTORY))
{
    printf("Target dir not found!\n");
    return -1;
}
```

三、遇到的问题及解决方案

1、 一开始不清楚如何在操作系统上自动运行编译好的程序，参照之前我在 GitHub 上对 initcode 相关代码的疑问提出的 issue



我领会到也可以把外部程序通过硬编码到内核程序中，使外部程序运行的方法。于是，根据我提的这个 issue，我不但解决了上面 issue 提到的问题，还体会到了一个改进内核功能的新思路

2、 对于已经实现的相近的系统调用，我参考了项目中的原有的实现，如要实现 openat 系统调用，我参照了 open 系统调用的实现。

3、 在实际的调试中，我遇到了特权级切换，以及 CSR 寄存器方面的各个位的作用不理解，导致写的程序出现了一系列 bug 的问题。通过查找 RISC-V 手册，清华大学的 rCore 项目书中的介绍，以及 xv6 实现文档解决了问题。

4、 总的来说，通过 GitHub 这一群智化开源软件平台，帮助自己独立的开发，完成系统调用，对我的影响最大。我深深的感受到了开源对我的帮助，正是因为提了很多 issue 并收到了解答，我才能解决许多的问题。开源软件能够帮我完成很多更有创意的作品，解决许多困难。同时，仔细阅读开源软件提供的文档，也是利用开源软件的重中之重。

5、 对于 QEMU 和 K210 实体机在 FAT32 文件系统初始化方面表现的不同，我也发起过了一个 issue，得到了如下的回复

我也认识到了不同设备之间的差别是如何影响我们的编码的，在编码的过程中更应该考虑到实际情况，增强代码的健壮性。于是，我把上面初始化 FAT32 文件系统的函数稍作改进，移到了 forkret 函数里面再调用，问题成功解决。

6、 对于之前在 K210 上一定概率出现非法指令异常的问题，我在 GitHub 上提问之后，xv6-k210 的作者之一，华中科技大学的车春池同学给了我如下的回复。

一开始，我并不是很明白具体应该怎么做，但是，根据出现非法指令是在每次运行测试用例之间的情况，以及所学到的 RISC-V 体系结构的知识，我想到了可以在 fork 的时候使用 fence.i 指令，对访存进行同步，fence.i 指令同样可以在运行于 RustSBI 上操作系统的 S 态调用。

```
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int fork(void)
{
  sfence_vma();
  asm volatile("fence.i");
```

在每次 fork 系统调用的一开始进行指令流同步之后，成功解决了我的一定概率出现非法指令的问题。

四、收获与感悟

通过这次内核实现赛初赛，我对之前学的操作系统课程，以及自学的 RISC-V 体系结构有了更深的认识。书本上讲的内容，让我在这次比赛中得到了许多的实践，特别是对于进程管理，RISC-V 指令以及其结构有了实际的体会。同时，我也认识到了，书本上的内容只有通过实践，才能够更加得到理解。在这次初赛中，我也体会到，我与华中科技大学的洛佳等一系列能力出众的同学还有很大的差距，在他们的帮助下，跟着他们也学到了很多的知识，

原来我的操作系统课上讲的是 Linux 0.11 的实现，通过这次初赛，我又深入了解了 xv6

操作系统，了解了 xv6 和 Linux 0.11 的不同，同时，对于操作系统概念，特别是机器启动还有 trap 等的过程，课堂上没有细讲的，在这次比赛中得到了很好的学习。同时，在此之后，我还在我们老师的指导下，利用这次比赛的经验，初步实现了 Linux 0.11 到 RISC-V QEMU 的移植，这也是这次比赛的一个额外收获吧。

同时我也认识到，我在嵌入式系统的驱动移植方面仍有不足，与其他参赛队的同学相比，很多驱动移植的操作对我的难度还是很大，限制了我的发挥，我还需要多多了解驱动移植调试，以及嵌入式系统方面的知识。今后仍要努力，尽力为中国的计算机系统与操作系统发展做出更大的贡献！

附录 1：kernel 文件夹结构及解释

```
├── bio.c                ----缓冲区的 I/O 驱动
├── console.c            ----控制台读写驱动
├── disk.c               ----SD 卡读写驱动
├── dmac.c               ----DMA 控制器驱动
├── entry_k210.S         ----k210 的内核入口代码，用于从 k210 启动内核
├── entry_qemu.S         ----qemu 的内核入口代码，用于从 qemu 启动内核
├── exec.c               ----执行 SD 卡上的用户态程序
├── fat32.c              ----FAT32 文件系统的实现
├── file.c               ----管理进程打开的文件
├── fpioa.c              ----FPIOA 串口的读写，只用在 k210 上
├── gpiohs.c             ----GPIOHS 串口的读写，只用在 k210 上
├── include
│   ├── buf.h            ----提供缓冲区的结构体定义
│   ├── date.h           ----定义当前时间（在 xv6-k210 未得到具体实现）
│   ├── defs.h           ----提供项目用到的函数与结构体原型
│   ├── dmac.h           ----定义了 DMA 控制器运行用到的数据结构
│   ├── elf.h            ----定义了 ELF 文件头和程序段的数据结构
│   ├── encoding.h       ----定义了访问 CSR 寄存器部分的掩码
│   ├── fat32.h          ----定义了 FAT32 系统的数据结构
```

```
│       ├──── fcntl.h          ----以十六进制数字的形式定义了文件的操作方式
│       ├──── file.h           ----定义了管理文件信息的结构体
│       ├──── fpioa.h          ----定义了 FPIOA 串口操作函数的原型与操作信号
│       ├──── gpio_common.h    ----定义了 GPIO 串口操作函数的原型与操作信号
│       ├──── gpiohs.h         ----定义了 GPIOS 串口操作函数的原型与操作信号
│       ├──── intr.h           ----定义了开关中断的函数原型
│       ├──── lcc_syscall.h    ----定义了我自己实现的函数原型与结构体
│       ├──── memlayout.h      ----定义了内存布局，以及相关段的基地址
│       ├──── param.h          ----定义了进程数、设备数等的最大数量限制
│       ├──── platform.h       ----定义了 K210 的设备信息
│       ├──── plic.h           ----定义了 RISC-V 分级中断的实现函数原型
│       ├──── printf.h         ----定义了 printf、panic 函数原型
│       ├──── proc.h           ----定义了进程实现结构体和函数原型
│       ├──── riscv.h          ----定义并封装了 CSR 寄存器的 C 语言操作实现
│       ├──── sbi.h            ----定义并实现了 SBI 交互函数
│       ├──── sdcard.h         ----定义了 SD 卡驱动函数原型
│       ├──── sleeplock.h      ----定义了进程睡眠锁的数据结构
│       ├──── spi.h            ----提供了 k210 通信总线所需的函数原型和结构体
│       ├──── spinlock.h       ----提供了 k210 芯片引脚锁的实现
│       ├──── stat.h           ----定义了 SD 卡中的文件状态结构体
│       ├──── string.h         ----定义了字符串处理函数的原型
│       ├──── syscall.h        ----定义了系统调用号
│       ├──── sysctl.h         ----定义了系统控制设备的抽象的结构体
│       ├──── sysinfo.h        ----定义了系统中进程和剩余内存的信息
│       ├──── types.h          ----定义使用的基本数据类型，以及二进制读写函数
│       ├──── uarths.h         ----定义了 uart 寄存器相关标记的读写位置
│       ├──── utils.h          ----定义了 K210 寄存器读写实现
│       ├──── virtio.h         ----定义了 qemu 使用的 I/O 寄存器在内存中的映射
│       └──── vm.h             ----定义了虚拟内存的操作函数原型
├──── intr.c                   ----实现了开关中断的功能
```

```
├──── kalloc.c              ----为进程分配虚拟内存页帧
├──── kernelvec.S           ----实现了内核软中断向量表
├──── lcc_syscall.c         ----实现了自己的系统调用
├──── lcc_variables.c       -----一些自己系统调用用到的变量
├──── logo.c                ----定义了 xv6 的 logo
├──── main.c                ----实现了 xv6-k210 的主函数
├──── pipe.c                ----实现了 xv6-k210 的管道功能
├──── plic.c                ----实现了 RISC-V 的分级中断控制
├──── printf.c              ----实现了 printf 的函数
├──── proc.c                ----进程初始化，关闭相关
├──── sdcard.c              ----SD 卡驱动相关
├──── sleeplock.c           ----进程睡眠锁相关
├──── spi.c                 ----SPI 通信协议相关
├──── spinlock.c            ----引脚互斥锁相关
├──── string.c              ----字符串操作函数实现
├──── swtch.S               ----进程切换相关
├──── syscall.c             ----系统调用总控相关
├──── sysctl.c              ----时钟控制相关
├──── sysfile.c             ----文件 I/O 相关
├──── sysproc.c             ----进程管理系统调用相关
├──── timer.c               ----时钟中断处理相关
├──── trampoline.S          ----软中断处理相关
├──── trap.c                ----trap 处理相关
├──── uart.c                ----uart 串口操作相关
├──── utils.c               ----K210 GPIO 串口读写相关
├──── virtio_disk.c         ----QEMU SD 卡 I/O 相关
├──── vm.c                  ----虚拟内存管理相关
└──── xv6-riscv-license     ----xv6-k210 开源许可证
```