

xv6-k210 实验报告

一、实验概述

本实验基于 xv6-k210，旨在对 xv6-k210 (<https://github.com/HUST-OS/xv6-k210>) 内核向用户提供的系统调用接口进行扩展，以满足大赛要求。xv6-k210 已实现了多数大赛要求的系统调用，但一些系统调用的功能与大赛要求存在差异，且大赛要求系统启动后能自动读取测试样例测试并自动关机，因此需对其进行适当修改。后续将详细介绍实验流程、系统调用实现过程（重点阐述 clone 系统调用）以及遇到的问题。

二、实验的前置准备工作

我在主机的 linux 虚拟机下完成该实验，实验的前置过程大致如下：

1、拉取大赛提供的文件

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

2、拉取大赛提供的镜像

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

3、创建容器，启动并挂载 xv6-k210

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

需要注意的是，我们需要替换 qemu 版本，下载

<https://github.com/rustsbi/rustsbi-qemu/releases/download/v0.2.0-alpha.2/rustsbi-qemu-release.zip>，解压后，使用更换 bootloader/SBI/sbi-qemu。

为了方便在本地环境进行调试，还需要在本地编译测试用例，并且在本地通过修改 init.c 实现对测试用例的自动调用。此处需要的步骤比较多，也比较复杂，不再过多解释，具体流程和教学网上提供的流程类似。

xv6-k210 中，系统启动并完成初始化后调用 userinit (/xv6-k210/kernel/main.c: 58) 创建第一个用户进程。需要注意的是，该进程并不直接将程序从磁盘加载进入内存，而是通过在数组 initcode (/xv6-k210/kernel/proc.c: 241)中存储程序指令的字节编码来创建进程。于是我们需要在修改 init.c 的同时，修改 initcode，使之能够存储修改后的 init.c 的字节编码。

在希冀平台上运行时，需对 makefile 进行修改，以确保能正确调用平台所提供的测试用例。大赛规定，系统在完成所有测试用例的执行后，要能够自动关机，此功能借助 qemu 来达成。具体而言，向物理地址 0x100000 起始的 4 字节写入数值 0x5555，便可关闭宿主机上的 qemu 进程，也就是关停 qemu 所模拟的虚拟机。基于此，我们可对 init.c 进行相应调整，让其在执行完所有测试用例后，执行向物理地

址 0x100000 开始的 4 字节写入 0x5555 的操作，进而顺利实现系统自动关机的功能要求。

三、大赛所需系统调用的添加

在 xv6-k210 的文档中详细给出了添加系统调用的过程，具体如下：

1. 在 xv6-user 目录下

- + 在 user.h 文件中，添加新系统调用封装后的函数声明，假设其函数名为 ``mysyscall``。

- + 在 usys.pl 文件末尾，添加如下行：

2. 在 kernel 目录下

- + 在 include/syscall.h 文件中，添加新系统调用号的宏定义：

其中，“?”为新的合法系统调用号，本质上是一个数组下标，可根据需要设置，建议按顺序递增添加。

- + 在 syscall.c 文件中，添加功能函数的声明，并更新系统调用表：

3. 根据该系统调用的功能，选择一个适合的内核模块的源文件，在其中实现 ``sys_mysyscall`` 函数的功能。

例如，与文件相关的系统调用可以在 kernel/sysfile.c 中添加。

可以利用 syscall.c 文件中提供的相关函数，在 ``sys_mysyscall`` 中获取用户进程传递的参数。

由于我们在上一步修改了 init.c 的逻辑，我们实际上不需要在用户侧 user.h 等文件修改。下面以 clone 函数为例，介绍本实验中系统调用的添加。

```
#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)
```

无论是 clone 还是 fork，二者皆具备通过复制既有进程的方式来创建全新进程的能力。其中，clone 属于 Linux 系统特有的系统调用机制。它能够用于创建一个进程或者线程，并且借助一组特定的标志（flags）来精准界定父子进程之间需要共享的各类资源。不仅如此，clone 还具备一项独特的优势，那就是可以通过传入函数指针的方式，明确指定新进程开始执行的具体起始位置，这一特性使得 clone 在功能上相较于 fork 而言，展现出了更为强大的灵活性与可定制性，能够更好地满足多样化的编程需求与复杂的进程创建场景。

基本思路：

通过分配新进程、复制父进程的内存空间和寄存器状态、设置子进程的执行环境，以及复制文件描述符等，创建了一个与父进程共享地址空间的新进程，实现线程的功能。新进程的执行入口可以由父进程指定，方便进行线程级别的控制和管理。

具体细节：

1、变量声明和当前进程获取：

```
int i, pid;

struct proc *np;

struct proc *p = myproc();

定义计数器 i、进程 ID pid。

定义新进程指针 np。

获取当前进程指针 p。
```

2、分配新进程结构体：

```
if((np = allocproc()) == NULL){

    return -1;

}
```

调用 allocproc() 分配新的进程结构体，如果失败则返回 -1。

3、复制父进程的用户内存空间：

```
if(uvmcopy(p->pagetable, np->pagetable, np->kpagetable, p->sz) < 0){

    freeproc(np);

}
```

```

        release(&np->lock);

        return -1;
    }

```

np->sz = p->sz;

调用 `uvmcopy()` 将父进程的用户页表和内存复制到子进程。

如果复制失败，释放新进程并返回 -1。

设置新进程的大小 `sz` 与父进程相同。

4、设置新进程的父进程、跟踪掩码和寄存器：

```

        np->parent = p;

        np->tmask = p->tmask;

        *(np->trapframe) = *(p->trapframe);

```

将新进程的父进程设置为当前进程。

复制父进程的跟踪掩码 `tmask`。

复制父进程的陷阱帧（保存的寄存器状态）。

5、设置子进程的栈和入口点：

```

        uint64 stack = p->trapframe->a1;

        if(stack != 0){

            uint64 fn = *((uint64 *)((char *) (p->trapframe->a1)));

            uint64 arg = *((uint64 *)((char *) (p->trapframe->a1) + 8));

            np->trapframe->sp = stack;

            np->trapframe->epc = fn;

            np->trapframe->a1 = arg;

        }

```

从父进程的陷阱帧中获取栈地址 `stack`（传递的参数）。

如果 `stack` 不为 0，表示需要设置子进程的栈和入口函数：

读取函数指针 `fn` 和参数 `arg`。

设置子进程的栈指针 `sp`、程序计数器 `epc`（入口函数地址）和参数寄存器 `a1`。

6、设置子进程的返回值：

```
np->trapframe->a0 = 0;
```

将子进程的 a0 寄存器设置为 0，表示子进程从 fork 返回时获取的返回值为 0

7、复制文件描述符和当前工作目录：

```
for(i = 0; i < NOFILE; i++)
```

```
if(p->ofile[i])
```

```
np->ofile[i] = filedup(p->ofile[i]);
```

```
np->cwd = edup(p->cwd);
```

遍历打开的文件描述符表，若父进程有打开的文件，调用 filedup 增加引用计数并赋值给子进程。

复制当前工作目录 cwd。

8、复制进程名称：

```
safestrcpy(np->name, p->name, sizeof(p->name));
```

将父进程的名称复制给子进程。

9、设置新进程的 PID 和状态：

```
pid = np->pid;
```

```
np->state = RUNNABLE;
```

获取新进程的进程 ID pid。

将新进程的状态设置为可运行 RUNNABLE，使其被调度。

10.释放新进程的锁并返回 PID：

```
release(&np->lock);
```

```
return pid;
```

对于剩余的系统调用，不再赘述实现细节，仅介绍实现思路。

四、文件系统相关系统调用：

```
# 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);

实现思路：

通过遍历当前进程的工作目录的父目录链，逐步构建绝对路径字符串。最终将构建好的路径复制到用户提供的内存地址中。如果在遍历过程中遇到路径过长或复制失败的情况，函数应该返回错误码。

#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);

实现思路：

该系统调用与 xv6-k210 的 pipe 系统调用一致，只需修改系统调用号即可。

#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);

实现思路：

该系统调用与 xv6-k210 的 dup 系统调用一致，只需修改系统调用号即可。

```
#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);
```

实现思路：

通过以下步骤实现文件描述符的复制：

1. 获取并验证旧的和新的文件描述符参数。
2. 检查新的文件描述符是否在有效范围内。
3. 如果新的文件描述符已经被占用，先关闭它以释放资源。
4. 将旧的文件描述符对应的文件结构复制到新的文件描述符，并增加引用计数。
5. 返回新的文件描述符以供调用者使用。

```
#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);
```

实现思路：

该系统调用与 xv6-k210 的 chdir 系统调用一致，只需修改系统调用号即可。

```
#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);
```

实现思路：

通过以下步骤实现文件的打开或创建：

1. 获取系统调用参数：包括文件路径、文件描述符和打开模式。
2. 处理文件创建或打开：根据打开模式决定是否创建新文件或打开已存在的文件。
3. 分配文件结构和描述符：确保有有效的文件结构和文件描述符可用。
4. 处理文件截断：根据需要截断文件内容。
5. 初始化文件结构：设置文件类型、偏移量以及读写权限。
6. 解锁资源并返回：解锁目录项并返回分配的文件描述符。

```
#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);
```

实现思路：该系统调用和 xv6-k210 中的 close 系统调用一致，修改系统调用号即可。


```
#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);
```

实现思路：

通过以下步骤实现读取目录条目：

1. 参数获取与验证：获取文件描述符、缓冲区长度和用户空间地址，并验证其有效性。
2. 目录属性检查：确保文件描述符对应的是一个可读的目录。
3. 读取目录条目：循环读取目录中的每个条目，跳过空条目，直到填满用户提供的缓冲区或达到目录末尾。
4. 复制到用户空间：将有效的目录条目复制到用户空间的缓冲区，并更新读取的字节数。
5. 返回结果：返回实际读取的字节数，或在遇到错误时返回 -1。

```
#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);
```

实现思路：该系统调用和 xv6-k210 中的 read 系统调用一致，只需要修改系统调用号即可。

```
#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);
```

实现思路：该系统调用和 xv6-k210 中的 write 系统调用一致，只需修改系统调用号即可。

```
#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);
```

实现思路:

可以分为两部分, 第一部分是文件检索, 具体实现和 openat 系统调用中的路径检索同理。在根据文件路径名找到文件的目录项后, 将目录项从目录文件中移除即可。

```
#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);
```

实现思路:

根据路径名确定新创建的文件夹的位置, 在该位置创建目录文件并设置权限。

```
#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);
```

实现思路：

该系统调用原理上和 xv6-k210 中的 fstat 系统调用一致，修改系统调用号即可实现基本功能。

五、进程管理相关系统调用

```
#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);
```

实现思路：

该系统调用与 xv6-k210 中的 exec 系统调用一致，只需要修改系统调用号即可。

```
#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);
```

实现思路：

通过修改/xv6-k210/kernel/proc.c 中的 wait 函数来实现题目要求的功能，主要修改了参数数量和判断有进程结束的条件以实现等待特定进程结束。

```
#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);
```

实现思路：

该系统调用和 xv6-k210 中的 `exit` 系统调用一致，只需修改系统调用号即可。

```
#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);
```

实现思路：

该系统调用和 xv6-k210 中的 `getpid` 系统调用一致，修改系统调用号即可。

```
#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);
```

实现思路：

直接从进程 PCB 中获取父进程的 PCB，再从父进程的 PCB 获取父进程的 `pid` 即可。

六、其他系统调用

```
#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);
```

实现思路：

原题目描述存在偏差，实际情况是输入的为堆扩张或收缩后的目标大小数值。特别

地，当输入 0 时，则会返回当前堆的实际大小。需要注意的是，xv6 - k210 采用的是页式内存管理方式，并未实现段页式内存管理机制，所以从严格意义上来说，其无法精准地实现堆段大小的扩展与收缩操作。鉴于此，在此情境下选择以扩展或收缩整个进程所占用的内存空间来作为一种替代的实现方式。

```
#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);
```

实现思路：由于我们换用了新的 sbi-qemu，时钟频率发生了改变，因此需要修改 INTERVAL

```
#define INTERVAL (10000000 / 200)
```

```
#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);
```

实现思路：按照要求输出即可

```
#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);
```

实现思路：仅需要借助 yield 函数，提供相应接口即可。

```
#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);
```

实现思路：

和 times 系统调用类似，我们通过时钟中断次数 ticks 和时钟中断频率计算时间，然后填写指定的结构体即可。

```
#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);
```

实现思路：

和前面的调用类似， 同样通过获取时钟中断数量来计算时间。

七、总结

本实验聚焦于 xv6-k210，旨在依据大赛要求对其系统调用接口予以扩展与调整。鉴于 xv6-k210 已有部分系统调用功能与大赛要求存在出入，且需达成系统启动后自动测试并关机的功能，实验对其进行了多方面的改造工作。

实验前期，于主机的 Linux 虚拟机中进行了一系列准备工作，涵盖拉取大赛文件与镜像、创建并挂载容器，同时对 qemu 版本进行替换，并在本地完成测试用例的编译与相关代码修改，以实现自动调用测试用例与自动关机的功能。

在系统调用添加环节，xv6-k210 文档详细阐述了添加流程，涉及在不同目录下的多个文件中进行函数声明、宏定义等操作。以 clone 系统调用为例，详细说明了其创建子进程的功能实现过程，包括进程结构体分配、内存空间与寄存器状态复制等关键步

骤，展示了如何通过一系列操作实现线程功能。

文件系统相关系统调用，各有其特定功能与输入输出设定。部分系统调用如 SYS_pipe2、SYS_dup 等仅需变更系统调用号即可满足大赛要求，而 SYS_getcwd 则需遍历目录构建路径，SYS_dup3 涉及多步骤的文件描述符复制操作，其他如 SYS_openat 等也各自具备独特的实现思路与步骤，从获取参数到文件处理与资源管理等多方面协同实现功能。

进程管理相关系统调用中，部分与 xv6-k210 原有系统调用类似，通过修改调用号或在原有基础上修改函数（如 SYS_wait4）来满足大赛特定要求，像 SYS_exit、SYS_getpid、SYS_getppid 等也分别依据自身功能特点进行了相应调整或实现。

其他系统调用，如 SYS_brk 因 xv6-k210 内存管理特性采用替代方式实现功能调整，SYS_times 由于 sbi-qemu 更换而修改相关参数，SYS_uname 按要求输出信息，SYS_sched_yield 借助特定函数提供接口，SYS_gettimeofday 和 SYS_nanosleep 则通过获取时钟中断信息来计算并处理时间相关操作，以实现各自功能并符合大赛需求。本实验还具有很大的局限性，例如 mount，mmap 等内容因时间关系并未实现，在未来的学习中可以进一步研究。感谢课程组提供的机会，让我亲身感受了操作系统内核系统调用的实现，增进了我对这一领域的理解。