操作系统 2024春 内核赛报告文档

刘闯鸣 2200013135

一、概览

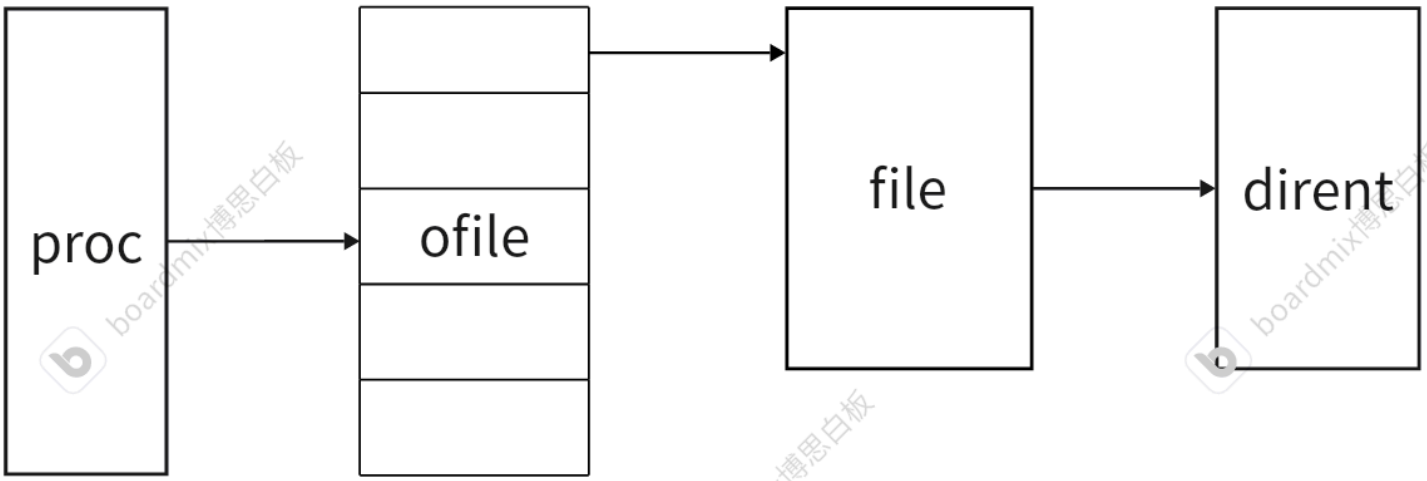
基于xv6-k210实现，系统调用调用实现情况如下：

文件系统相关	**实现方式**	**进程管理相关**	**实现方式**
SYS_getcwd	新增	SYS_clone	新增
SYS_pipe2	新增	SYS_execve	新增
SYS_dup	xv6已实现	SYS_wait4	新增
SYS_dup3	新增	SYS_exit	xv6已实现
SYS_chdir	xv6已实现	SYS_getppid	新增
SYS_openat	新增	SYS_getpid	xv6已实现
SYS_close	xv6已实现	**内存管理相关**	**实现方式**
SYS_getdents64	新增	SYS_brk	新增
SYS_read	xv6已实现	SYS_munmap	新增
SYS_write	xv6已实现	SYS_mmap	新增
SYS_linkat	新增	**其他**	**实现方式**
SYS_unlinkat	新增	SYS_times	新增
SYS_mkdirat	新增	SYS_uname	新增
SYS_umount2	xv6-k210不支持	SYS_sched_yield	新增
SYS_mount	xv6-k210不支持	SYS_gettimeofday	新增
SYS_fstat	新增	SYS_nanosleep	新增

后面仅介绍改动与添加的部分

二、文件系统相关系统调用

首先看一下xv6的文件系统实现方式：



xv6内核使用proc结构体存储进程相关信息，其中包含打开文件表ofile。
ofile是一个结构体指针数组，每一项都存着一个file*类型指针。
file结构体包含文件类型，操作权限，可选的pipe链接信息，
以及指向文件的目录项dirent结构体的指针。
文件的具体信息存储在dirent结构体内。

0.两个工具函数

由于文件系统相关系统调用在传入地址参数时多数使用如下规则：

(假设输入为const char* path, int fd)
path如为绝对路径，则忽略fd。
如为相对路径，且fd是AT_FDCWD，则path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path是相对于fd所指向的目录来说的。

因此我封装了两个工具函数：get_abspath与get_path

- int get_abspath(struct dirent* cwd, char* path)
  参数1为目标文件的目录项，参数2为存储绝对地址的字符串
  函数会递归地查找当前目录项的父目录，并将其添加到路径上，最终得到绝对路径
- int get_path(char* path, int fd)
  参数1为输入的path，参数2为输入的fd
  分三种情况：

1. path为绝对路径（以‘/’字符开头）：path本身就是所求，直接返回即可
2. fd=AT_FDCWD：首先获取当前进程工作目录，调用get_abspath获取工作目录绝对地址，再将path（相对地址）加到其后，得到要打开文件的绝对地址
3. fd是一个文件描述符，path相对于fd所指向的目录：根据fd得到其目录结构，调用get_abspath获取fd绝对地址，再将path（相对地址）加到其后，得到要打开文件的绝对地址
get_path函数会自动判断输入形式，返回绝对地址

1.SYS_getcwd

1. 读入参数并判断是否合法
2. 如果buf为NULL则使用uvmalloc从堆中分配一块内存
3. 获取进程控制块及当前工作目录
4. 如果为根目录，则为'/'，否则递归查找其父目录，获得绝对地址
5. 判断缓冲区大小是否合适，若缓冲区不够大则返回NULL
6. 将工作目录拷贝到缓冲区，成功则返回指向地址字符串的指针，否则返回NULL

2.SYS_pipe2

1. 读入参数并判断是否合法
2. 创建管道，得到两个file结构体指针
3. 调用fdalloc为两个file结构分配文件描述符
4. 将两个文件描述符拷贝至用户指定的地址
5. 以上步骤若出现问题则关闭所有创建的结构并返回-1，否则返回0

3.SYS_dup3

1. 读入参数并判断是否合法（调用argfd直接得到要复制的文件的file结构体指针）
2. 将该指针填入文件描述符表与新文件描述符相对应的表项
3. 文件引用数+1
4. 以上步骤若出现问题则返回-1，否则返回0

4.SYS_openat

1. 读入参数并判断是否合法
2. 调用工具函数get_path得到绝对地址
3. 根据FLAGS的掩码确定不同的操作：
    - 如果flags & O_CREATE不为0则创建新文件，得到新文件的目录项
    - 否则通过绝对地址查找文件，得到其目录项
4. 在文件描述符表中填入相关信息
5. 以上操作在对文件目录项进行操作的时候需要加锁，防止发生同步问题

6. 以上步骤若出现问题则关闭所有创建的结构并返回-1，否则返回0

5.SYS_getdents64

按题目要求新增结构体`dirall`用于保存相关信息

1. 读入参数并判断是否合法（之后调用getdents64实现功能）
2. 读取目录的相应信息并保存到缓冲区
3. 以上操作在对文件目录项进行操作的时候需要加锁，防止发生同步问题
4. 以上步骤若出现问题则关闭所有创建的结构并返回-1，否则返回0

6.SYS_linkat

1. 读入参数并判断是否合法
2. 调用工具函数get_path获取原文件与新文件的绝对地址
3. 通过绝对地址获取原文件的目录项，如果原文件是目录文件则返回-1
4. 创建新文件，并将源文件的信息填入新文件的目录项
5. 以上操作在对文件目录项进行操作的时候需要加锁，防止发生同步问题
6. 以上步骤若出现问题则关闭所有创建的结构并返回-1，否则返回0

7.SYS_unlinkat

1. 读入参数并判断是否合法
2. 调用工具函数get_path获取文件的绝对地址
3. 通过绝对地址获取文件的目录项，并判断操作是否合法
4. 移除链接
5. 以上操作在对文件目录项进行操作的时候需要加锁，防止发生同步问题
6. 以上步骤若出现问题则关闭所有创建的结构并返回-1，否则返回0

8.SYS_mkdirat

1. 读入参数并判断是否合法
2. 调用工具函数get_path获取文件的绝对地址
3. 在指定位置创建新目录项
4. 以上操作在对文件目录项进行操作的时候需要加锁，防止发生同步问题
5. 以上步骤若出现问题则返回-1，否则返回0

9.SYS_fstat

按题目要求新增结构体kstat用于保存相关信息
同时，在文件目录项中新增数据保存文件的访问时间、修改时间、状态改变时间

1. 读入参数并判断是否合法
2. 从进程控制块获取文件描述符表，进而获取文件目录项
3. 将文件状态信息填入缓冲区
4. 以上步骤若出现问题则关闭所有创建的结构并返回-1，否则返回0

Fail：SYS_mount &SYS_umount2

xv6-k210的设备描述方式是将文件描述符表项的type值填写为FD_DEVICE，同时封装了设备的读写函数，因此**似乎**无法通过设备名查找设备。

以下为xv6-k210部分源代码

```
struct file {
    enum { FD_NONE, FD_PIPE, FD_ENTRY, FD_DEVICE } type;
    int ref; // reference count
    char readable;
    char writable;
    struct pipe *pipe; // FD_PIPE
    struct dirent *ep;
    uint off;          // FD_ENTRY
    short major;       // FD_DEVICE
};

// #define major(dev)  ((dev) >> 16 & 0xFFFF)
// #define minor(dev)  ((dev) & 0xFFFF)
// #define mkdev(m,n)  ((uint)((m)<<16| (n)))

// map major device number to device functions.
struct devsw {
    int (*read)(int, uint64, int);
    int (*write)(int, uint64, int);
};
```

三、进程管理相关系统调用

1.SYS_clone

1. 读入参数并判断是否合法（之后调用clone实现功能）
2. 遍历进程控制块列表，寻找父进程，同时确定子进程的ID未被使用（ptid非法则说明要clone当前进程，调用myproc获取当前进程）
3. 创建新进程
4. 将父进程的信息与内存数据拷贝到子进程
5. 在打开文件表中将相应文件的引用计数加一
6. 根据调用要求，设置子进程的栈指针与进程ID
7. 子进程设置完成后将其设置为就绪状态
8. 以上操作在对进程控制块进行操作的时候需要加锁，防止发生同步问题
9. 以上步骤若出现问题则返回-1，否则返回0

2.SYS_execve

1. 读入参数并判断是否合法
2. 拷贝参数与环境变量，并分别统计二者数目
3. 调用execve实现功能
4. 拷贝内存空间（不包含旧用户空间），注意保留当前使用的内核栈
5. 查找将要运行的程序
6. 检查ELF头是否合法
7. 将程序加载进内存
8. 分配两页，其中一个作为用户栈
9. 向栈中依次压入环境变量数组与参数数组
10. 向栈中依次压入指向环境变量数组和参数数组的指针
11. 设置栈指针
12. 保存程序名称，方便调试
13. 初始化进程环境，并释放部分资源
14. execve返回argc，控制移交回SYS_execve
15. 清理参数数组
16. 以上步骤若出现问题则返回-1，否则返回0

3.SYS_wait4

1. 读入参数并判断是否合法（之后调用wait4实现功能）
2. 判断需要等待指定进程还是全部子进程
3. 获取进程锁

4. 循环判断是否有满足要求的子进程，若有则将相应信息拷贝至缓冲区，并清理已经终止的子进程，最后释放锁
5. 特殊情况1：当前进程没有子进程/未找到指定子进程——返回-1
6. 特殊情况2：当前进程已被杀死——返回-1
7. 特殊情况3：options & WNOHANG ！ = 0 —— 不进行循环
8. 以上操作在对进程控制块进行操作的时候需要加锁，防止发生同步问题
9. 以上步骤若出现问题则返回-1，否则返回0

4.SYS_getppid

1. 调用myproc获取当前进程
2. 查找当前进程的父进程，若未找到，返回-1
3. 返回当前进程的父进程的进程ID

四、内存管理相关系统调用

1.SYS_brk

1. 读入参数并判断是否合法（之后调用brk实现功能）
2. 获取当前进程内存值sz
3. 如果brk=0（函数中对应addr变量），说明用户希望获取当前进程内存值，返回这个值
4. 否则根据addr与sz的大小关系调用uvmalloc或uvmdealloc，并在进程控制块中修改相应值
5. 以上步骤若出现问题则返回-1，否则返回0

2.SYS_munmap

1. 读入参数并判断是否合法
2. 获取当前进程的进程控制块
3. 释放对应映射区域的内存
4. 以上步骤若出现问题则返回-1，否则返回0

3.SYS_mmap

1. 读入参数并判断是否合法
2. 获取当前进程的进程控制块
3. 若未指定起始映射位置，则扩张堆，映射至扩张区域
4. 将指定映射内容从文件拷贝到内存
5. 以上操作在对进程控制块进行操作的时候需要加锁，防止发生同步问题
6. 以上步骤若出现问题则返回-1，否则返回0

五、其他系统调用

首先研究一下xv6-k210的时钟机制

定义位于param.h中

```
#define INTERVAL     (100000000 / 200) // timer interrupt interval
```

这里原版的xv6-k210设置的是(390000000 / 200)，而我在配置环境的时候替换了sbi-qemu，测试内核所给出的打印信息显示，平台的频率是10000000 Hz

```
 _____          _     _  __                      _
|_   _|__  ___| |_  | |/ /___ _ __ _ __   ___| |
  | |/ _ \/ __| __| | ' // _ \ '__| '_ \ / _ \ |
  | |  __/\__ \ |_  | . \  __/ |  | | | |  __/ |
  |_|\___||___/\__| |_|\_\___|_|  |_| |_|\___|_|
================================================
| boot hart id          |                    6 |
| smp                   |                    8 |
| timebase frequency    |          10000000 Hz |
| dtb physical address  |           0x87e00000 |
------------------------------------------------
```

然而，将内核的频率参数按照10000000 Hz设置后，一些**原本能够**通过测试的样例现在不能通过测试了。观察输出后发现打印信息无误但输出顺序混乱，猜测是因为xv6-k210的printf**不具备**原子性，从而使得输出顺序混乱，进而影响评测。

较为简单的解决办法是更改内核的频率参数，我将其改为了(100000000 / 200)，也就是让内核认为平台的频率是100000000 Hz（不恰当的说法是，内核相当于比原来慢十倍）。更改后，printf输出正常。

然后就是数据的对应，200的含义应该是每秒对应200个中断周期，因此在最终的版本中，一个中断周期对应

$$\left(\frac{1}{10000000}\right)\times\left(\frac{100000000}{200}\right)$$

=0.05秒，也即5毫秒

riscv.h文件中提供了函数r_time()，可获取自系统运行以来的总时钟周期数

```
// 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;
}
```

1.SYS_times

按题目要求新增结构体tms用于保存相关信息
同时，在进程控制块中新增n_tick项用于保存进程运行时间（代表时钟中断次数）

- 新进程初始化的时候将n_tick设为0
- 调度器每次调度进程时将其n_tick加1（进程运行一次时钟中断的时间）

1. 读入参数并判断是否合法
2. 调用myproc获取当前进程的进程控制块，计算得到进程运行时间
3. 存入缓冲区
4. 以上步骤若出现问题则返回-1，否则返回0

2.SYS_uname

按题目要求新增结构体utsname用于保存相关信息

1. 读入参数并判断是否合法
2. 将系统信息拷贝至缓冲区
3. 以上步骤若出现问题则返回-1，否则返回0

3.SYS_sched_yield

1. 调用yield实现功能
2. 调用myproc获取当前进程的进程控制块
3. 将进程状态改为就绪，并调用sched，将控制转移至调度器

4.SYS_gettimeofday

1. 读入参数并判断是否合法

2. 调用r_time获取系统时间
3. 计算相应时间项并填入缓冲区
4. 以上步骤若出现问题则返回-1，否则返回0

5.SYS_nanosleep

1. 读入参数并判断是否合法
2. 计算需要睡眠的滴答数
3. 睡眠结束后返回0，若进程尚未被唤醒即被杀死，返回-1
4. 以上操作在对滴答锁进行操作的时候需要加锁，防止发生同步问题
5. 以上步骤若出现问题则返回-1，否则返回0