# Lab4-challenge实验报告



# 一、实现思路

## 结构体定义

我将signal相关的结构体定义在**include\signal.h**文件中，其中的sigaction结构体和sigset_t结构体为题目所给。

**signal_quene**结构体定义了一个保存信号相关信息的链表，**len**为链表长度，**head**指针为链表的头指针；**signal_node**结构体为链表的节点，包括**signal**信号名，**oldMask**为信号处理前的进程掩码，**oldTf**为信号处理前的上下文信息，用于信号处理完之后的上下文恢复，**next**为指向下一个节点的指针

```
typedef struct {
    u_int sig[2]; //最多64个信号
} sigset_t;

struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
};

struct signal_node {
    int signal;
    sigset_t oldMask;
```

```
    struct Trapframe oldTf;
    struct signal_node *next;
};

struct signal_quene {
    int len;
    struct signal_node *head;
};
```

## PCB上需要新增部分

```
struct Env {
    /* 未改变部分 */
    //Lab4-challenge
    struct sigaction sigaction_list[64];    //注册的sigaction列表
    sigset_t signal_mask;                   //进程信号掩码
    struct signal_quene signal_list;        //信号栈，后收到的信号先处理
    struct signal_node *cur_signal;         //当前正在处理的信号链表的头结点
    u_int signal_return;                    //信号处理返回函数入口
    u_int isCow;                            //正在写时复制标志
}
```

## sigaction和sigprocmask函数实现

分别使用系统调用**syscall_sigaction**和**syscall_sigprocmask**在内核态下进行对当前进程的PCB进行操作

```
int sys_sigaction(int signum, const struct sigaction *act, struct sigaction
*oldact) {
    if (signum > 64) {
        return -1;
    }
    if (oldact != NULL) {
        *oldact = curenv->sigaction_list[signum-1];
    }
    //在进程的sigaction_list中为相应进程添加处理函数
    curenv->sigaction_list[signum-1].sa_handler = act->sa_handler;
    curenv->sigaction_list[signum-1].sa_mask = act->sa_mask;
    return 0;
}

int sys_sigprocmask(int how, const sigset_t *set, sigset_t *oldset) {
    if (oldset != NULL) {
        *oldset = curenv->signal_mask;
    }
    if (set == NULL) {
        return -1;
    }
    //修改进程的signal_mask，达到屏蔽信号的目的
    if (how == SIG_BLOCK) {
```

```
        curenv->signal_mask.sig[0] |= set->sig[0];
        curenv->signal_mask.sig[1] |= set->sig[1];
        //不允许屏蔽SIGKILL信号
        u_int temp = ~(0x1 << (SIGKILL - 1));
        curenv->signal_mask.sig[0] &= temp;
    } else if (how == SIG_UNBLOCK) {
        curenv->signal_mask.sig[0] &= (~set->sig[0]);
        curenv->signal_mask.sig[1] &= (~set->sig[1]);
    } else if (how == SIG_SETMASK) {
        curenv->signal_mask = *set;
    } else {
        return -1;
    }
    return 0;
}
```

## kill函数实现

也由一个系统调用**syscall_send_signal**实现，最后在内核态的**kern\signal.c**文件中调用**send_signal**完成。

**sig_alloc**负责申请一个新的空间用于放置**struct signal_node**，此函数与page_init函数类似，都是申请一页空间，之后将结构体放置在空闲链表中等待调用。

**send_signal**先使用**envid2env**获得需要接收信号的进程的PCB，通过**sig_alloc**申请一个新的**signal_node**将其插入在对应**env->signal_list**（待处理信号链表）的头部，最后在发送信号之后，应该使用**env_run**尝试运行对应进程，尝试信号的处理。

另外**SIGSEGV**信号可由操作系统发出，所以在**kern\tlbex.c**中新增信号的发送即可。

```
struct signal_node *free_list = NULL;

int sig_alloc(struct signal_node **new) {
    if (free_list == NULL) {
        struct Page *p;
        page_alloc(&p); //申请一页新的物理页
        p->pp_ref++;
        free_list = (struct signal_node *)page2kva(p);
        free_list->next = NULL;
        struct signal_node *temp;
        for (temp = free_list + 1; temp + 1 < page2kva(p) + BY2PG; temp++) {
            temp->next = free_list;
            free_list = temp;
        }
    }
    *new = free_list;
    free_list = free_list->next;
    return 0;
}

int send_signal(u_int envid, int sig) {
    if (sig > 64) {
        return -1;
    }
```

```
    struct Env *env;
    struct signal_node *new;
    if (envid == 0) {
        env = curenv;
    } else {
        int r = envid2env(envid, &env, 0);
        if (r < 0) {
            return -1;
        }
    }
    if (env->signal_list.head == NULL) {
        sig_alloc(&new);
        env->signal_list.len++;
        env->signal_list.head = new;
        new->next = NULL;
        new->signal = sig;
    } else {
        sig_alloc(&new);
        env->signal_list.len++;
        new->next = env->signal_list.head;
        new->signal = sig;
        env->signal_list.head = new;
    }
    env_run(env);
}
```

```
static void passive_alloc(u_int va, Pde *pgdir, u_int asid) {
    struct Page *p = NULL;

    if (va < UTEMP) {
        //Lab4-challenge
        send_signal(0, SIGSEGV); //发送SIGSEGV信号

        panic("address too low");
    }
    /* 以下部分与之前相同 */
}
```

# 信号的处理

## 何时处理信号

在每次调用**env_run**时，在触发异常之前都会先调用一个定义在**kern\signal.c**中的**do_signal**函数进行信号的处理，之后调用汇编函数**env_pop_tf**使中断发生，让操作系统进行中断处理，并跳转到对应的用户态信号处理函数。

```
void env_run(struct Env *e) {
    /* 未作改变部分*/
    //Lab4-challenge
    do_signal(&curenv->env_tf);

    env_pop_tf(&curenv->env_tf, curenv->env_asid);
}
```

## 信号处理函数的返回

每次调用用户态的信号处理函数后，需要返回到内核态进行上下文恢复，这里我使用类似COW处理方法，为每个进程设置回调函数，使用**syscall_set_signal_return**将回调函数**syscall_signal_return**保存在PCB中，最后通过**syscall_signal_return**进行返回到内核态的**kern\signal.c**中的**signal_finish**函数，进行上下文的恢复。

```
void signal_finish() {
    //从当前进程正在处理的信号链表中中取出头结点，结束此信号处理并环境进行恢复
    struct Trapframe *temp_tf = &curenv->cur_signal->oldTf;
    curenv->signal_mask = curenv->cur_signal->oldMask; //恢复原本的掩码
    curenv->cur_signal = curenv->cur_signal->next; //信号处理完毕，从cur_signal链表中移除
    do_signal(temp_tf); //继续处理剩下的信号
    env_pop_tf(temp_tf, curenv->env_asid); //将保存的旧环境pop到当前进程中，进行恢复
}
```

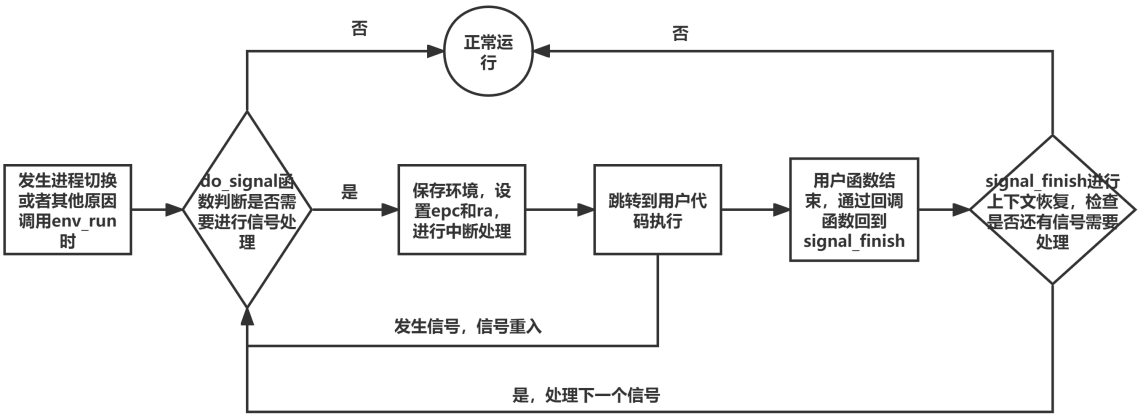
## 信号的处理与上下文保存

**do_signal**函数为信号处理的核心函数，具体部分解释如下：

```
void do_signal(struct Trapframe *tf) {
    struct signal_node *temp, *pre_temp;
    int flag = 0;
    if (curenv->signal_list.len != 0 && !curenv->isCow) {
        //如果待处理信号链表不为空，且此时未进行写时复制尝试进行信号处理
        temp = curenv->signal_list.head;
        while (temp != NULL) {
            if (!sigismember(&curenv->signal_mask, temp->signal)) {
                break;
            } else {
                pre_temp = temp;
                temp = temp->next;
            }
        }
        //从待处理链表头开始，寻找到一个未屏蔽信号，因为是从头结点开始，所以后来的信号先处理
        if (temp != NULL) {
            if (curenv->sigaction_list[temp->signal - 1].sa_handler) {
                //如果用户定义了处理函数
                temp->oldTf = *tf; //保存上下文
                temp->oldMask = curenv->signal_mask; //保存原本掩码
                //更新进程掩码
                curenv->signal_mask.sig[0] |= curenv->sigaction_list[temp->signal - 1].sa_mask.sig[0];
                curenv->signal_mask.sig[1] |= curenv->sigaction_list[temp->signal - 1].sa_mask.sig[1];
```

```
                curenv->signal_mask.sig[0] &= (~(0x1 << (SIGKILL - 1))); //不允许
屏蔽9号中断
                flag = 1;
                tf->regs[4] = temp->signal; //传递参数
                tf->regs[31] = curenv->signal_return; //设置信号处理返回
                tf->cp0_epc =
                    (u_int)curenv->sigaction_list[temp->signal - 1].sa_handler;
//用户信号处理函数入口
            } else if (temp->signal == SIGKILL || temp->signal == SIGSEGV ||
                       temp->signal == SIGTERM) {
                //以上三种信号有默认处理函数
                env_destroy(curenv); //默认处理为结束进程
            }
            //从待处理信号链表中删除此信号
            if (temp != curenv->signal_list.head) {
                pre_temp->next = temp->next;
            } else {
                curenv->signal_list.head = temp->next;
            }
            curenv->signal_list.len--;
            if (flag) {
                //将要处理的信号保存在curenv->cur_signal链表中，标记此信号正在处理
                if (curenv->cur_signal == NULL) {
                    curenv->cur_signal = temp;
                    curenv->cur_signal->next = NULL;
                } else {
                    temp->next = curenv->cur_signal;
                    curenv->cur_signal = temp;
                }
            }
        }
    }
}
```

## 流程图



## 写时复制的解决

### 子进程继承父进程的信号处理函数

因为fork函数最终会调用**sys_exofork**进行新进程的初始化，所以只需要在其中将父进程的信号处理相关内容复制给子进程就行。

```
int sys_exofork(void) {
    /* 未作改变部分 */

    //复制父进程的信号处理函数
    memcpy(e->sigaction_list, curenv->sigaction_list, sizeof(e-
>sigaction_list[0])*64);
    //复制父进程待处理的信号
    e->signal_list = curenv->signal_list;
    //复制父进程正在处理的信号
    e->cur_signal = curenv->cur_signal;
    //复制父进程的掩码
    e->signal_mask = curenv->signal_mask;
    //复制返回值
    e->signal_return = curenv->signal_return;
    return e->env_id;
}
```

### 写时复制问题的处理

因为写时复制的处理和信号的处理部分都在用户态，如果在进行写时复制处理（即运行**cow_entry**）时，进行信号的处理，则并行的两个用户态程序就会出现问题，为了解决这个问题，我定义了一个新的系统调用**syscall_set_cow**和PCB上的一个新的域**isCow**，我在进行写时复制之前将**isCow**赋值成1，结束时赋值成0，相当于对其**"加锁"**，同时在**do_signal**中进行判断，阻止运行信号处理函数。

```
static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf) {
    syscall_set_cow(1);

    /* 中间部分未作更改 */

    syscall_set_cow(0);

    int r = syscall_set_trapframe(0, tf);
    user_panic("syscall_set_trapframe returned %d", r);
}
```

---

# 二、测试与测试结果

## 基础功能测试

运行如下指令可进行基础功能测试：

```
make test lab=4_challenge_1 && make run
```

测试了sigaction和sigprocmask的基本功能，详细测试点与解释如下：

```
#include <lib.h>

int global = 0;
int *test = NULL;
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
void handler(int num) {
    debugf("Reach handler, now the signum is %d!\n", num);
    global++;
}

void sgv_handler(int num) {
    debugf("Signal appear!, signum is %d\n", num);
    test = &a[0];
    global++;
    debugf("test = %d.\n", *test);
}

int main() {
    sigset_t set;
    sigemptyset(&set);
    struct sigaction sig, oldsig;
    sig.sa_handler = handler;
    sig.sa_mask = set;
    panic_on(sigaction(2, &sig, NULL));          //注册2号信号的处理函数

    sig.sa_handler = sgv_handler;
    panic_on(sigaction(8, &sig, NULL));         //注册8号信号处理函数

    sigaddset(&set, 2);                          //屏蔽2和8号信号
    sigaddset(&set, 8);
    panic_on(sigprocmask(0, &set, NULL));
    kill(0, 2);                                  //发生2号信号
    kill(0, 8);                                 //发生8号信号
    int ans = 0;
    for (int i = 0; i < 10000000; i++) {
        ans += i;
    }

    debugf("\npre ans is %d\n\n", ans);
    panic_on(sigprocmask(1, &set, NULL));        //解除2和8信号屏蔽
    debugf("\nafter ans is %d\n", ans);
    *test = 10;
    debugf("now test is %d\n\n", *test);

    panic_on(sigaction(2, &sig, &oldsig));       //交换两个处理函数
    panic_on(sigaction(8, &oldsig, &sig));
    panic_on(sigaction(2, &sig, NULL));
    kill(0, 2);
    kill(0, 8);

    debugf("\nglobal = %d.\n", global);
    return 0;
}
```

输出结果：

```
pre ans is -2014260032

Signal appear!, signum is 8
test = 1.
Reach handler, now the signum is 2!

after ans is -2014260032
now test is 10

Signal appear!, signum is 2
test = 10.
Reach handler, now the signum is 8!

global = 4.
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
panic at sched.c:45 (schedule): schedule: no runnable envs
ra:    80016cb8  sp:  803ffe80  Status: 0000100c
Cause: 00000020  EPC: 004019f0  BadVA:  00404000
curenv:    NULL
cur_pgdir: 83ffc000
```

结果解释：

输出的pre ans和after ans相同，说明调用信号处理函数后没有改变上下文环境，通过sigaction取出2和8号的处理函数之后进行调换，也得到了正确的结果。

## 信号重入测试

运行如下指令可进行测试：

```
make test lab=4_challenge_2 && make run
```

测试了在信号处理函数中出现信号的情况，类似递归，测试代码和解释如下：

```c
#include <lib.h>

int global = 0;
int global2 = 0;
void handler(int num) {
    global++;
    debugf("Reach handler, now glboal = %d\n", global);
    if (global < 10) {
        kill(0,32); //被屏蔽
        kill(0, 64); //递归
    }
    debugf("Leave handler, now sig=%d\n", num);
}

void handler2(int num) {
    global2++;
    debugf("Now can run sig=%d\n", num);
}

int main() {
    struct sigaction sig;
    sig.sa_handler = handler;
    sigemptyset(&sig.sa_mask);
```

```
    sigaddset(&sig.sa_mask, 32); //处理64号信号时屏蔽32号信号
    panic_on(sigaction(64, &sig, NULL));

    sigemptyset(&sig.sa_mask);
    sig.sa_handler = handler2;
    panic_on(sigaction(32, &sig, NULL));

    debugf("Signal send...\n");
    kill(0, 64);
    if (global == 10 && global2 == 9)
        debugf("Test pass!\n");
    return 0;
}
```

输出结果：

```
Signal send...
Reach handler, now glboal = 1
Reach handler, now glboal = 2
Reach handler, now glboal = 3
Reach handler, now glboal = 4
Reach handler, now glboal = 5
Reach handler, now glboal = 6
Reach handler, now glboal = 7
Reach handler, now glboal = 8
Reach handler, now glboal = 9
Reach handler, now glboal = 10
Leave handler, now sig=64
Leave handler, now sig=64
Leave handler, now sig=64
Leave handler, now sig=64
Leave handler, now sig=64
Leave handler, now sig=64
Leave handler, now sig=64
Leave handler, now sig=64
Leave handler, now sig=64
Leave handler, now sig=64
Now can run sig=32
Now can run sig=32
Now can run sig=32
Now can run sig=32
Now can run sig=32
Now can run sig=32
Now can run sig=32
Now can run sig=32
Now can run sig=32
Test pass!
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
```

结果解释：

64号信号在执行信号处理函数时会屏蔽32号信号，所以只有当64号信号运行完处理函数后，32号信号才能开始处理运行。

## 多种信号大量数据测试

运行如下指令可进行测试：

```
make test lab=4_challenge_3 && make run
```

使用掩码将大量信号阻塞，一起释放进行测试，详细测试代码和解释如下：

```
#include <lib.h>
```

```c
int global = 0;
void handler(int num) {
    debugf("Reach handler, now the signum is %d!\n", num);
    global++;
}

int main() {
    debugf("Big data test begin:\n");
    sigset_t set;
    sigemptyset(&set);
    struct sigaction sig;
    sig.sa_handler = handler;
    sig.sa_mask = set;
    panic_on(sigaction(2, &sig, NULL));
    panic_on(sigaction(8, &sig, NULL));

    sigaddset(&set, 2);
    sigaddset(&set, 8);
    panic_on(sigprocmask(0, &set, NULL)); //屏蔽2和8信号
    int ans = 0;
    for (int i = 0; i < 1000; i++) { //连续发送多个信号，压力测试
        kill(0, 2);
        kill(0, 8);
        ans += i;
    }

    panic_on(sigprocmask(1, &set, NULL)); //解除屏蔽
    debugf("global = %d.\n", global);
    if (global == 2000)
    debugf("Test pass!\n");
    return 0;
}
```

输出结果：

```
Reach handler, now the signum is 2!
Reach handler, now the signum is 8!
Reach handler, now the signum is 8!
Reach handler, now the signum is 8!
Reach handler, now the signum is 2!
Reach handler, now the signum is 2!
Reach handler, now the signum is 2!
Reach handler, now the signum is 8!
Reach handler, now the signum is 8!
Reach handler, now the signum is 8!
Reach handler, now the signum is 2!
Reach handler, now the signum is 2!
global = 2000.
Test pass!
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
```

结果解释：

输出太长只展示了一部分，可以看到输出中2和8交错排列，最后共2000条，因为我们实现的是可靠信号，所以后来的信号会先处理，但因为信号重入，所以2和8并不是严格交错排列的。

## 几种特殊信号测试

运行如下指令可进行测试：

```
make test lab=4_challenge_4 && make run
```

详细测试代码和解释如下：

```
#include <lib.h>

int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *test = NULL;
void null_handler(int num) {
    debugf("Segment fault appear!\n");
    test = &a[0];
    debugf("test = %d.\n", *test);
    debugf("env %d exit\n", syscall_getenvid);
    exit();
}

void kern_stop_handler(int num) {
    debugf("signal %d can't be blocked\n", num);
    debugf("env %d exit\n", syscall_getenvid);
    exit();
}

void user_stop_handler(int num) {
    debugf("signal %d can be blocked\n", num);
}

int main(int argc, char **argv) {
    sigset_t set;
    sigemptyset(&set);
    struct sigaction sig;
    sig.sa_handler = null_handler;
    sig.sa_mask = set;
    panic_on(sigaction(11, &sig, NULL)); // SIGSEGV

    sig.sa_handler = kern_stop_handler;
    panic_on(sigaction(9, &sig, NULL)); // SIGKILL

    sig.sa_handler = user_stop_handler;
    panic_on(sigaction(15, &sig, NULL)); // SIGTERM

    sigaddset(&set, 11);
    sigaddset(&set, 9);
    sigaddset(&set, 15);
    sigprocmask(0, &set, NULL);

    if (fork()) {
        debugf("null test env is %d\n", syscall_getenvid);
        sigdelset(&set, 11);
        sigprocmask(2, &set, NULL);
        *test = 10;
```

```
    } else {
        if (fork()) {
            debugf("SIGKILL test env is %d\n", syscall_getenvid);
            kill(0, 9);
        } else {
            kill(0, 15);
            debugf("SIGTERM test env is %d\n", syscall_getenvid);
            sigdelset(&set, 15);
            sigprocmask(2, &set, NULL);
            debugf("can't stop now\n");
        }
    }
    return 0;
}
```

输出结果：

```
null test env is 4199220
Segment fault appear!
test = 1.
env 4199220 exit
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
SIGKILL test env is 4199220
signal 9 can't be blocked
env 4199220 exit
[00001001] destroying 00001001
[00001001] free env 00001001
i am killed ...
SIGTERM test env is 4199220
signal 15 can be blocked
can't stop now
[00001802] destroying 00001802
[00001802] free env 00001802
i am killed ...
```

结果解释：

最早结束的是空指针测试进程，测试的是11号信号；9号信号不能被阻塞，所以发出后就运行；15号信号可以被阻塞。

## 进程之间发送信号测试

运行如下指令可进行测试：

```
make test lab=4_challenge_fork && make run
```

详细测试代码和解释如下：

```
#include <lib.h>
int cnt = 0;
int father;
void handler(int num) {
    cnt++;
    if (cnt > 5) {
        return;
    }
    if (syscall_getenvid() == father) {
        debugf("father get signal=%d, cnt=%d\n", num, cnt);
```

```
    } else {
        debugf("child get signal=%d, cnt=%d\n", num, cnt);
    }
}

int main() {
    struct sigaction act;
    sigemptyset(&act.sa_mask);
    act.sa_handler = handler;
    panic_on(sigaction(15, &act, NULL));
    panic_on(sigaction(25, &act, NULL));
    father = syscall_getenvid();
    int r = fork();
    if (r != 0) {
        sigset_t set;
        sigemptyset(&set);
        for (int i = 0; i < 5; i++) {
            kill(r, 25); //向子进程发送25信号
        }
        while (cnt != 5); //等待子进程发送完信号
        debugf("father test passed!\n");
    } else {
        while (cnt != 5); //等待父进程发送完信号
        debugf("child test passed!\n");
        for (int i = 0; i < 5; i++) {
            kill(father, 15); //向父进程发送15信号
        }
    }
    return 0;
}
```

输出结果：

```
child get signal=25, cnt=1
child get signal=25, cnt=2
child get signal=25, cnt=3
child get signal=25, cnt=5
child get signal=25, cnt=4
child test passed!
father get signal=15, cnt=1
father get signal=15, cnt=2
father get signal=15, cnt=3
father get signal=15, cnt=4
father get signal=15, cnt=5
[00001001] destroying 00001001
[00001001] free env 00001001
i am killed ...
father test passed!
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
```

结果解释：

子进程成功收到了父进程发出的25号信号，父进程也成功收到了子进程发出的15号信号。

# 三、遇到的问题和解决方案

## 信号重入问题

我们实现的信号与Linux下信号有很大不同，我们的信号机制需要实现类似异常重入的信号重入机制，但在Linux中进行信号处理时是不允许在进行信号处理时发生信号中断的。开始时，我只保存一份环境，但这样在发生信号重入时保存的上下文环境会被覆盖。

为了解决此问题，我将开始处理但未完成的信号保存在PCB的**cur_signal**链表中，并同时保存运行前的环境和掩码，这样就可以实现原本环境的保留和恢复，实现信号重入。

## 写时复制问题

我在跑上述的fork测试时，发现父进程接收到15号信号时会报cow错误，经过艰难的打印输出debug过程，我发现问题就出在进行写时复制时如果出现信号并立即进行信号的处理，信号处理结束后写时复制就不能正常返回了，我判断其原因是信号的处理函数和写时复制的处理函数都在用户态，二者都不为原子过程，产生了类似并发问题，所以我希望通过类似加锁的方式解决并发问题，所以我新增了一个新的系统调用**syscall_set_cow**用于加锁，最终成功解决了此问题。