

# Lab2 实验报告

姓名：李雨朔 学号：20232131090 班级：计科一班 日期：2025.10.15

## 一 . 实验环境准备

```
git fetch  
git checkout syscall  
make clean
```

## 二 . 实验 2.2

1. 修改 Makefile，添加 trace 程序：打开 Makefile，找到 UPROGS 部分，在末尾添加以下内容然后保存退出：

```
$U/_trace\
```

```
UPROGS=\  
$U/_cat\  
$U/_echo\  
$U/_forktest\  
$U/_grep\  
$U/_init\  
$U/_kill\  
$U/_ln\  
$U/_ls\  
$U/_mkdir\  
$U/_rm\  
$U/_sh\  
$U/_stressfs\  
$U/_usertests\  
$U/_grind\  
$U/_wc\  
$U/_zombie\  
$U/_sleep\  
$U/_pingpong\  
$U/_primes\  
$U/_trace\
```

2. 添加系统调用声明到 user/user.h：

```
int trace(int);
```

```
GNU nano 1.2
struct stat;
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(const char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int trace(int);
```

3. 添加系统调用入口到 user/usys.pl:

```
entry("trace");
```

```
entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
entry("close");
entry("kill");
entry("exec");
entry("open");
entry("mknod");
entry("unlink");
entry("fstat");
entry("link");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("trace");
```

4. 添加系统调用号到 kernel/syscall.h, 在已有定义后面添加:

```
#define SYS_trace 22
```

```
GNU nano 7.2
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_trace  22
```

5. 在进程结构体中添加 trace 掩码字段。打开 kernel/proc.h，找到 struct proc 定义，添加 mask 字段：

```
int mask;
```

```
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
    struct spinlock lock;

    // p->lock must be held when using these:
    enum procstate state;          // Process state
    void *chan;                   // If non-zero, sleeping on chan
    int killed;                  // If non-zero, have been killed
    int mask;                     // If non-zero, have been killed
    int xstate;                  // Exit status to be returned to parent's wait
    int pid;                      // Process ID
```

6. 实现 trace 系统调用函数。打开 kernel/sysproc.c，在文件末尾添加：

```
uint64
sys_trace(void)
{
    int mask;

    if(argint(0, &mask) < 0) return -1;
```

```
// 设置当前进程的 trace 掩码  
myproc()->mask = mask;  
return 0;  
}
```

```
// return how many clock tick interrupts have occurred  
// since start.  
uint64  
sys_uptime(void)  
{  
    uint xticks;  
  
    acquire(&tickslock);  
    xticks = ticks;  
    release(&tickslock);  
    return xticks;  
}  
  
unit64  
sys_trace(void)  
{  
    int mask;  
  
    if(argint(0, &mask) < 0) return -1;  
    myproc()->mask = mask;  
  
    return 0;  
}
```

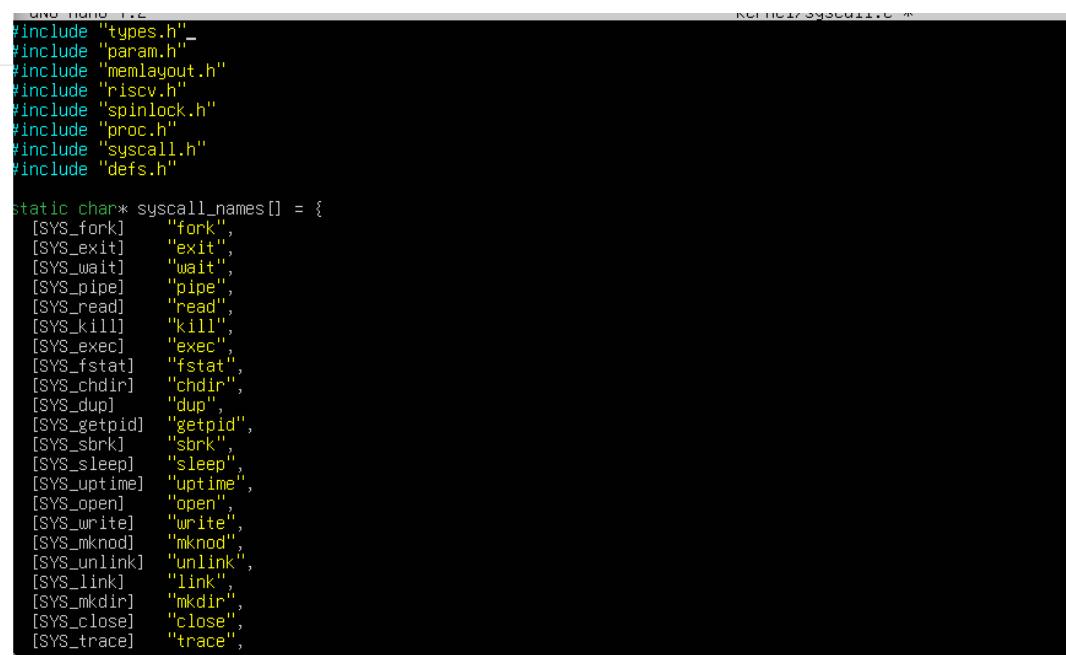
7. 打开 kernel/proc.c, 找到 fork 函数, 在复制进程信息的部分 (`np->cwd = idup(p->cwd);`) 后面添加:

```
// 复制 trace 掩码  
np->mask = p->mask;
```

```
// increment reference counts on open file descriptors.  
for(i = 0; i < NOFILE; i++)  
    if(p->ofile[i])  
        np->ofile[i] = filedup(p->ofile[i]);  
np->cwd = idup(p->cwd);  
np->mask = p->mask;  
  
safestrcpy(np->name, p->name, sizeof(p->name));  
pid = np->pid;  
release(&np->lock);  
acquire(&wait_lock);  
np->parent = p;  
release(&wait_lock);
```

8. 打开 kernel/syscall.c, 在文件开头附近添加系统调用名称数组:

```
static char* syscall_names[] = {
    [SYS_fork]    "fork",
    [SYS_exit]    "exit",
    [SYS_wait]    "wait",
    [SYS_pipe]    "pipe",
    [SYS_read]    "read",
    [SYS_kill]    "kill",
    [SYS_exec]    "exec",
    [SYS_fstat]   "fstat",
    [SYS_chdir]   "chdir",
    [SYS_dup]     "dup",
    [SYS_getpid]  "getpid",
    [SYS_sbrk]    "sbrk",
    [SYS_sleep]   "sleep",
    [SYS_uptime]  "uptime",
    [SYS_open]    "open",
    [SYS_write]   "write",
    [SYS_mknod]   "mknod",
    [SYS_unlink]  "unlink",
    [SYS_link]    "link",
    [SYS_mkdir]   "mkdir",
    [SYS_close]   "close",
    [SYS_trace]   "trace",
};
```



The screenshot shows a terminal window with the following content:

```
one nano /v2
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "spinlock.h"
#include "proc.h"
#include "syscall.h"
#include "defs.h"

static char* syscall_names[] = {
    [SYS_fork]    "fork",
    [SYS_exit]    "exit",
    [SYS_wait]    "wait",
    [SYS_pipe]    "pipe",
    [SYS_read]    "read",
    [SYS_kill]    "kill",
    [SYS_exec]    "exec",
    [SYS_fstat]   "fstat",
    [SYS_chdir]   "chdir",
    [SYS_dup]     "dup",
    [SYS_getpid]  "getpid",
    [SYS_sbrk]    "sbrk",
    [SYS_sleep]   "sleep",
    [SYS_uptime]  "uptime",
    [SYS_open]    "open",
    [SYS_write]   "write",
    [SYS_mknod]   "mknod",
    [SYS_unlink]  "unlink",
    [SYS_link]    "link",
    [SYS_mkdir]   "mkdir",
    [SYS_close]   "close",
    [SYS_trace]   "trace",
}.
```

9. 在同一个文件(kernel/syscall.c)中，找到 syscall 函数 (void syscall(void))，在函数内部修改，添加 trace 输出逻辑。即将以下部分：

```
num = p->trapframe->a7;
if(num>0 && num<NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
} else {
    // ...
}
```

改为以下部分：

```
num = p->trapframe->a7;
if(num>0 && num<NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();

    // 添加 trace 输出
    if((p->mask & (1 << num)) != 0) {
        printf("%d: syscall %s -> %d\n",
               p->pid, syscall_names[num], p->trapframe->a0);
    }
} else {
    // ...
}
close(p[1]);
exit(0);
}
```

```
void
syscall(void)
{
    int num;
    struct proc *p = myproc();

    num = p->trapframe->a7;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // Use num to lookup the system call function for num, call it,
        // and store its return value in p->trapframe->a0
        p->trapframe->a0 = syscalls[num]();

        if((p->mask & (1 << num)) != 0) printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num], (int)p->trapframe->a0);
    } else {
        printf("%d %s: unknown sys call %d\n",
               p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}
```

10. 在同一个文件(kernel/syscall.c)中，找到 extern 声明部分，添加：

```
extern uint64 sys_trace(void);
```

```
// Prototypes for the functions that handle system calls.  
extern uint64 sys_fork(void);  
extern uint64 sys_exit(void);  
extern uint64 sys_wait(void);  
extern uint64 sys_pipe(void);  
extern uint64 sys_read(void);  
extern uint64 sys_kill(void);  
extern uint64 sys_exec(void);  
extern uint64 sys_fstat(void);  
extern uint64 sys_chdir(void);  
extern uint64 sys_dup(void);  
extern uint64 sys_getpid(void);  
extern uint64 sys_sbrk(void);  
extern uint64 sys_sleep(void);  
extern uint64 sys_uptime(void);  
extern uint64 sys_open(void);  
extern uint64 sys_write(void);  
extern uint64 sys_mknod(void);  
extern uint64 sys_unlink(void);  
extern uint64 sys_link(void);  
extern uint64 sys_mkdir(void);  
extern uint64 sys_close(void);  
extern uint64 sys_trace(void);  
  
// An array mapping syscall numbers from syscall.h  
// to the function that handles the system call.
```

找到系统调用数组，添加：

```
[SYS_trace] sys_trace,
```

```
// An array mapping syscall numbers from syscall.h  
// to the function that handles the system call.  
static uint64 (*syscalls[])(void) = {  
[SYS_fork] sys_fork,  
[SYS_exit] sys_exit,  
[SYS_wait] sys_wait,  
[SYS_pipe] sys_pipe,  
[SYS_read] sys_read,  
[SYS_kill] sys_kill,  
[SYS_exec] sys_exec,  
[SYS_fstat] sys_fstat,  
[SYS_chdir] sys_chdir,  
[SYS_dup] sys_dup,  
[SYS_getpid] sys_getpid,  
[SYS_sbrk] sys_sbrk,  
[SYS_sleep] sys_sleep,  
[SYS_uptime] sys_uptime,  
[SYS_open] sys_open,  
[SYS_write] sys_write,  
[SYS_mknod] sys_mknod,  
[SYS_unlink] sys_unlink,  
[SYS_link] sys_link,  
[SYS_mkdir] sys_mkdir,  
[SYS_close] sys_close,  
[SYS_trace] sys_trace, -  
};
```

11. 编译 xv6:

```
make qemu
```

测试：

```
# 测试 1：追踪 read 系统调用  
trace 32 grep hello README  
  
# 测试 2：追踪所有系统调用  
trace 2147483647 grep hello README  
  
# 测试 3：追踪 fork 系统调用  
trace 2 usertests forkforkfork
```

测试 1 截图：

```
xv6 kernel is booting  
  
hart 2 starting  
hart 1 starting  
init: starting sh  
$ trace 32 grep hello README  
3: syscall read -> 1023  
3: syscall read -> 971  
3: syscall read -> 298  
3: syscall read -> 0  
$
```

测试 2 截图：

```
$ trace 2147483647 grep hello README  
4: syscall trace -> 0  
4: syscall exec -> 3  
4: syscall open -> 3  
4: syscall read -> 1023  
4: syscall read -> 971  
4: syscall read -> 298  
4: syscall read -> 0  
4: syscall close -> 0  
$
```

测试 3 截图：

```
$ trace 2 usertests forkforkfork  
usertests starting  
5: syscall fork -> 6
```

```
8: syscall fork -> 24
10: syscall fork -> 25
8: syscall fork -> 26
9: syscall fork -> 27
10: syscall fork -> 28
8: syscall fork -> 29
9: syscall fork -> 30
8: syscall fork -> 31
10: syscall fork -> 32
10: syscall fork -> 33
8: syscall fork -> 34
8: syscall fork -> 35
9: syscall fork -> 36
9: syscall fork -> 37
10: syscall fork -> 38
8: syscall fork -> 39
9: syscall fork -> 40
8: syscall fork -> 41
9: syscall fork -> 42
8: syscall fork -> 43
9: syscall fork -> 44
8: syscall fork -> 45
9: syscall fork -> 46
9: syscall fork -> 47
8: syscall fork -> 48
9: syscall fork -> 49
10: syscall fork -> 50
8: syscall fork -> 51
9: syscall fork -> 52
10: syscall fork -> 53
10: syscall fork -> 54
8: syscall fork -> 55
9: syscall fork -> 56
8: syscall fork -> 57
9: syscall fork -> 58
9: syscall fork -> 59
8: syscall fork -> 60
9: syscall fork -> 61
10: syscall fork -> 62
8: syscall fork -> 63
9: syscall fork -> 64
8: syscall fork -> 65
9: syscall fork -> 66
10: syscall fork -> 67
8: syscall fork -> -1
9: syscall fork -> -1
OK
5: syscall fork -> 68
ALL TESTS PASSED
$
```

### 三 . 实验 2.3

1. 打开 nano user/attack.c 实现 attack.c:

```
int main(int argc, char *argv[]) {
    char *end = sbrk(17*PGSIZE); // 分配 17 个页面
    end += 16 * PGSIZE;          // 移动到第 17 个页面
    write(2, end+32, 8);
    exit(1);
}
```

## 2. 测试攻击：

```
make qemu  
  
attacktest
```

```
[init] i starting  
init: starting sh  
$ attacktest  
OK: secret is cbd.fbc
```

## 3. 思考题分析：

- 1) 为什么秘密数据存放在 end+32 的偏移处，而不是页面起始位置？

页面起始位置通常包含重要的元数据或指针，将秘密数据放在偏移处可以减少对正常程序运行的干扰。

- 2) 如果 secret.c 将秘密数据直接写入页面起始位置，攻击是否依然有效？

攻击依然有效，原因如下：

- (1) 内存分配机制不变：kalloc 仍然会分配 secret 进程刚刚释放的页面。
- (2) 内存页面未被清空的问题依然存在。

## 四 . 实验主观心得

本次实验让我对操作系统的内部机制有了更深入的理解，特别是系统调用的实现和内存管理的安全性：

1. 深入理解了系统调用机制：从用户态到内核态的完整流程，包括参数传递、权限切换、执行和返回。
2. 学习了在现有操作系统中添加新功能的完整流程，包括接口定义、内核实现和用户程序调用。
3. 通过攻击实验，深刻理解了内存隔离机制被破坏的严重后果，以及清空敏感数据的必要性。
4. 提升了调试和问题解决能力：在实验过程中遇到多个编译和运行时错误，通过分析错误信息和查阅资料，逐步解决了这些问题。
5. 操作系统必须提供安全的进程间通信机制，防止恶意程序窃取其他进程的敏感信息。