Report of Android OS Practice Lab1

李炎 1400012951

Background

安卓系统中的进程之间某些不必要的相互唤醒，增加了系统的负担，降低了用户体验。因此，本 Lab 旨在通过修改内核，对安卓系统的进程通信进行一定的限制。

Part 1: Add two system calls in the system call table

Lab 的第一部分，是给 Kernel 添加两个系统调用，分别为 setcolors 和 getcolors，用以赋予进程特定的颜色和得到某些进程的颜色。我在查阅各种资料之后，按照如下的步骤进行：

Step 1: Modify the system call table

第一步是在系统调用表中增加两行，注册我定义的系统调用。文件路径为 arch/x86/syscalls/syscall_64.tbl。表一共有 4 列，分别是系统调用号，注册的体系结构(common, 64, x32)，系统调用名，系统调用入口函数。我在表的 318 行和 319 行添加了两个条目，分别注册为 setcolors 和 getcolors

```
318 common  setcolors       sys_setcolors
319 common  getcolors       sys_getcolors
```

Step 2: Modify task_struct

Linux 中的进程模型通过一个叫做 task_struct 的数据结构进行维护，该数据结构中存储了该进程的重要信息，比如权限，优先级等等。第二步我要做的就是在 task_struct 中给每个进程添加一个属性 color。

```
/* Color of this task used to limit IPC*/
u_int16_t color;
```

Step 3: Add two function definition in syscalls.h

先在 syscalls.h 这个头文件中定义（课上助教和有的同学说新版本的 kernel 已经不需要在该头文件中添加定义了，但是我认为添加这两个 definition 是比较严谨的做法，对于后续的维护也有好处）。于是我在 include/linux/syscalls.h 的文件尾添加了两个函数的定义。

```
asmlinkage long sys_setcolors(int nr_pids, pid_t *pids, u_int16_t *colors,
int *retval);
asmlinkage long sys_getcolors(int nr_pids, pid_t *pids, u_int16_t *colors,
int *retval);
```

Step 4: Add two defines at unisted.h

在查阅了相关的资料之后，我发现在路径为 include/uapi/asm-generic 的 unistd.h 文件中应该为系统调用定义相应的宏。但是由于版本升级的原因，这一步似乎也可省略。具体的机制还未搞清，有待进一步阅读代码。

```
#define __NR_setcolors 318
__SYSCALL(__NR_setcolors, sys_setcolors)

#define __NR_getcolors 319
__SYSCALL(__NR_getcolors, sys_getcolors)

#undef __NR_syscalls
#define __NR_syscalls 274
```

Step 5: Function implementation in color.c

在做了一系列的准备工作之后，最重要的就是对这两个系统调用的实现。我在 kernel 路径下新建了一个 c 文件 color.c，用以实现这两个系统调用的入口函数。

函数的开始部分先对用户空间传入的参数做了一些必要的检查。首先要确保用户的权限足以调用这两个系统调用。注意到 Linux 中小于 500 的进程号都是系统保留的，换言之都是 root 用户的进程。而进程号大于 500 的进程分配给 User 使用。因此，函数的开始对检查 current_uid()，如果发现非 root 用户试图调用 setcolors 函数，返回-EACCES。其次，用户空间传入的地址不不可信任的，需要将之映射到内核空间加以检查。Copy_from_user()函数将用户地址空间的地址映射到内核空间，如果映射成功则返回 0，否则返回剩余的字节数。

注意到这一步里应该保持同一个进程组的 color 值一致。所以在进行 setcolor 操作时，一方面要做必要的检查，另一方面也要动态地给进程组的主线程进行染色。这一点在 getcolor 中体现的更为明显。

```c
/* nr_pids contains the number of entries in
   the pids, colors, and the retval arrays. The colors array contains the
   color to assign to each pid from the corresponding position of
   the pids array. Return 0 if all set color requests
   succeed. Otherwise, the array retval contains per-request
   error codes -EINVAL for an invalide pid, or 0 on success.
*/
asmlinkage long sys_setcolors(int nr_pids, pid_t *pids, u_int16_t *colors, int *retval){
    int i;
    pid_t *kpids = (pid_t*)malloc(nr_pids*sizeof(pid_t));
    u_int16_t *kcolors = (u_int16_t*)malloc(nr_pids*sizeof(u_int16_t));
    struct task_struct *task, *top_task;

    /*Check priority of the current user*/
    if(current_uid() > 500) return -EACCES;

    if(nr_pids < 0) return EFAULT;

    /*Check weather addr is ligeal*/
    if(copy_from_user(kpids, pids, nr_pids*sizeof(pid_t)) != 0)
        return EFAULT;
    if(copy_from_user(kcolors, colors, nr_pids*sizeof(u_int16_t)) != 0)
        return EFAULT;


    for(i = 0; i < nr_pids; ++i){
        task = find_task_by_vpid(pids[i]);
        if(task == NULL){
            retval[i] = -EINVAL;
            continue;
        }
        top_task = find_task_by_vpid(task->tgid);

        /*Top task has not been arrenged a color yet*/
        if(top_task->color == 0){
            top_task->color = colors[i];
            task->color = colors[i];
            retval[i] = 1;//Color succeed
        }
        /*Otherwise other threads must have the same color as the top task*/
        else{
            if(colors[i] == top_task->color){
                task->color = colors[i];
                retval[i] = 1;
            }
            else{
                task->color = top_task->color;
                retval[i] = 0;//Color failed
            }
        }

    }

    return EXIT_SUCCESS;
}
```

```c
/* Gets the colors of the processes
   contained in the pids array. Returns 0 if all set color requests
   succeed. Otherwise, an error code is returned. The array
   retval contains per-request error codes: -EINVAL for an
   invalid pid, or 0 on success.
*/
asmlinkage long sys_getcolors(int nr_pids, pid_t *pids, u_int16_t *colors, int *retval){
    int i;
    pid_t tmpid;
    pid_t *kpids = (pid_t*)malloc(nr_pids*sizeof(pid_t));
    u_int16_t *kcolors = (u_int16_t*)malloc(nr_pids*sizeof(u_int16_t));
    struct task_struct *task, *top_task;

    if(nr_pids < 0) return EFAULT;

    for(i = 0; i < nr_pids; ++i){
        if(pids[i] < 0)
            return EFAULT;
    }

    if(copy_from_user(kpids, pids, nr_pids*sizeof(pid_t)) != 0)
        return EFAULT;
    if(copy_from_user(kcolors, colors, nr_pids*sizeof(u_int16_t)) != 0)
        return EFAULT;
    for(i = 0; i < nr_pids; ++i){
        tmpid = pids[i];
        task = find_task_by_vpid(tmpid);
        if(task == NULL){
            retval[i] = -EINVAL;
            continue;
        }
        top_task = find_task_by_vpid(task->tgid);
        if(task->color != 0){
            colors[i] = task->color;
            retval[i] = 1;
        }
        else{
            if(top_task->color != 0){
                task->color = top_task->color;
                colors[i] = task->color;
                retval[i] = 2;
            }
            else{
                colors[i] = 0;
                retval[i] = 3;
            }
        }
    }

    return EXIT_SUCCESS;
}
```

Step 6: Modify Makefile

在 color.c 中实现了系统调用的相应功能之后，为了使我们加入的 c 文件能够被成功编译，必要的一步就是修改 Makefile。在这里由于我们仅仅加入了一个源文件，所以只需在后边添加一个 color.o 即可。

```
obj-y     = fork.o exec_domain.o panic.o printk.o \
        cpu.o exit.o itimer.o time.o softirq.o resource.o \
        sysctl.o sysctl_binary.o capability.o ptrace.o timer.o user.o \
        signal.o sys.o kmod.o workqueue.o pid.o task_work.o \
        rcupdate.o extable.o params.o posix-timers.o \
        kthread.o wait.o sys_ni.o posix-cpu-timers.o mutex.o \
        hrtimer.o rwsem.o nsproxy.o srcu.o semaphore.o \
        notifier.o ksysfs.o cred.o \
        async.o range.o groups.o lglock.o smpboot.o color.o
```

Step 7: Modify the fork action

（1）关于 vfork

逻辑上讲，当父进程通过 fork 操作派生出一个子进程时，相应的颜色也要进行传递。clone 操作也是类似。但是当执行 vfork 的操作 时，我们并不需要 color 的继承，这是因为 vfork 派生出的进程与父进程共享一片数据空间，且 vfork 在执行后通常要 exec 别的进程映像，这个进程实际上与父进程没有必然的联系。如果继承了父进程的 color，这就与 vfork 的设计初衷相违背。

查阅资料之后发现，fork 函数和 vfork 函数都是通过 do_fork()函数实现的，而在 do_fork()函数中，又调用了 copy_process()。该函数的开始段落就进行了对父进程整个 task_struct 的复制（函数 dup_task_struct()）而后续又根据实际的需求进行子进程属性的修改。而修改的一部分，就是根据 copy_process()被传入的第一个参数 clone_flags 确定的。当 clone_flags 为 CLONE_VFORK|CLONE_VM|SIGCHLD 时，copy_process()执行 vfork 的操作。

因此，copy 函数中对于 color 属性的修改，就如以下代码。

```
    /*Set a different color from old one if vfork*/
    if((clone_flags & (CLONE_VFORK | CLONE_VM | SIGCHLD))
== (CLONE_VFORK | CLONE_VM | SIGCHLD))
        p->color = current->color + 1;
```

（2）关于进程初始化

Linux 进程初始化时进程的 color 应初始化为 0。我的想法是在系统初始化进程号为 0 的进程时，将之初始化为 0。由于从本源上讲，每个进程都是由 0 号进程派生的，因此只需要在 0 号进程执行最初的 fork 时，将 0 号进程的 color 置 0 即可。具体实现如下。

```
if(current->pid == 0){
    current->color = 0;
    p->color = 0;
}
```

Part 2: Customizing Binder

Android 中的进程通信都是通过 binder 机制实现的。当参数 reply 为 1 时，ServiceManager 处理来自进程的事务请求。当目标进程与当前发出请求的进程的颜色不一致时，禁止该两个进程之间进行通信

```
if(proc->tsk->color != target_proc->tsk->color){
    if(proc->tsk->color != 0 && target_proc->tsk->color != 0){
        return_error = BR_FAILED_REPLY;
        goto err_diff_color;
    }
}
```

此处添加了一个标签 err_diff_color，当因为进程颜色不一致而禁止进程之间通信时，跳转到此标签处，执行后续的处理操作。

```
err_no_context_mgr_node:
err_diff_color:
    binder_debug(BINDER_DEBUG_FAILED_TRANSACTION,
            "%d:%d transaction failed %d, size %lld-%lld\n",
            proc->pid, thread->pid, return_error,
            (u64)tr->data_size, (u64)tr->offsets_size);
```

Part 3: Test programs

Step 1: Create two testing programs

为了测试添加的两个系统调用，我编写了三个测试程序。程序开始定义了两个宏，分别是系统调用号（此处以 getcolor 为例）和最多允许一次处理的的进程数。

```
#define __NR_getcolors 319
#define MAX_PID_NR 128
```

函数 getProcIdByName 由进程名得到进程 pid。

```
int getProcIdByName(char* name)
{
    struct dirent *dirp;
    DIR *dp = opendir("/proc");
    if (dp == NULL)
    {
        printf("Fail to open /proc\n");
        return -1;
    }
    while (dirp = readdir(dp))
    {
        int pid = atoi(dirp->d_name);
        int cmd_fd = -1;
        if (pid <= 0) continue;
        char cmdPath[128], cmdLine[128];
        sprintf(cmdPath, "/proc/%s/cmdline", dirp->d_name);
        cmd_fd = open(cmdPath, O_RDONLY);
        if (cmd_fd == -1)
        {
            printf("Fail to open file: %s\n", cmdPath);
            continue;
        }
        read(cmd_fd, &cmdLine, 128);
        if (strcmp(cmdLine, name) == 0)
            return pid;
    }
    return -1;
}
```

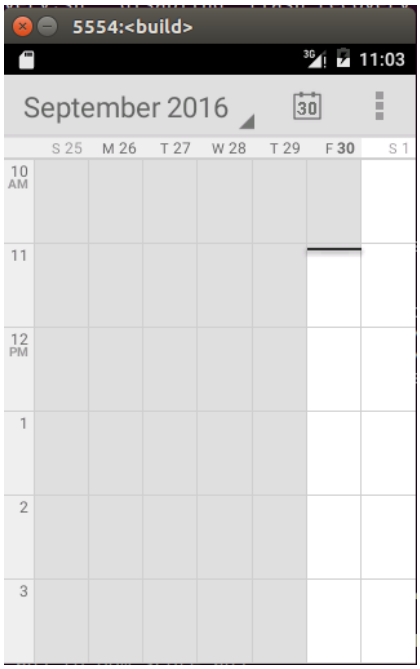
测试程序 getcolors 中，命令行读取 n（≤128）个参数，调用系统调用 getcolors 之后得到这 n 个进程的颜色，具体各种情形的返回值参见 color.c 中有关函数的实现过程。对于 setcolors 函数，同理，只不过参数有一个必须是奇数（出去程序名之外有 n 组数据，每组分别是进程名和进程即将赋予的颜色）的限制。

```
    int i;
    int nr_pids = 0;
    int pids[MAX_PID_NR];
    u_int16_t colors[MAX_PID_NR];
    int retval[MAX_PID_NR];
    if (argc <= 1)
    {
        printf("No process and color specified!\n");
        printf("usage:\n./getcolor proc_0 proc_1 ...\n");
        return 0;
    }
    for (i = 1; i < argc; i ++)
    {
        int pid = getProcIdByName(argv[i]);
        if (pid == -1)
        {
            printf("No proc found for name: %s\n", argv[i]);
            continue;
        }
        printf("Proc Name: %s, Proc ID: %d\n", argv[i], pid);
        pids[nr_pids] = pid;
        nr_pids ++;
    }
    long ret = syscall(__NR_getcolors, nr_pids, pids, colors, retval);
    printf("Finish syscal - ret: %d\n", ret);
    for (i = 0; i < nr_pids; i ++)
    {
        printf("\tProc ID: %d, color: %d, retval: %d\n", pids[i], colors[i], r
etval[i]);
    }
```

Step 2: Test in terms of Calendar program

为了测试 setcolor 与 binder 配合的机制能否有效地限制进程之间通信，我以安卓程序中的 Calendar 程序为例，把 com.android.calendar 程序和 com.android.providers.calendar 程序赋予不同的颜色。理论上讲，calendar 程序就会崩溃。

首先启动安卓地模拟器，此时 calendar 程序一切正常。



然后使用 adb 连接 android 虚拟机设备。由于文件系统权限地问题，需要先执行 mount –o rw,remount /重新挂载根目录。

```
root@generic_x86_64:/ # mount -o rw,remount /
```

我在安卓虚拟机中新建了一个文件夹 test 用来存放测试程序

```
127|root@generic_x86_64:/ # mkdir test
root@generic_x86_64:/ # ls
acct
cache
charger
config
d
data
default.prop
dev
etc
file_contexts
fstab.goldfish
fstab.ranchu
init
init.environ.rc
init.goldfish.rc
init.ranchu.rc
init.rc
init.trace.rc
init.usb.configfs.rc
init.usb.rc
init.zygote32.rc
init.zygote64_32.rc
mnt
oem
proc
property_contexts
root
sbin
sdcard
seapp_contexts
selinux_version
sepolicy
service_contexts
storage
sys
system
test
ueventd.goldfish.rc
ueventd.ranchu.rc
ueventd.rc
vendor
```

通过 adb 将编译出的两个可执行文件 push 到设备地/test 目录中。

```
iterator@iterator:~/aosp/test$ adb push setcolors /test
173 KB/s (7320 bytes in 0.041s)
iterator@iterator:~/aosp/test$ adb push getcolors /test
176 KB/s (7320 bytes in 0.040s)
```

然后通过 ps 查看当前的进程，注意到上文提到的两个进程都在其中，进程号分别为 1921 和 1975。

```
system    1032  1     10316  2620  binder_thr 7ff07b448707 S /system/bin/fingerprintd
system    1354  1024  697024 95256 SyS_epoll_ 7f335b3b15ca S system_server
u0_a16    1535  1024  622460 77624 SyS_epoll_ 7f335b3b15ca S com.android.systemui
u0_a2     1547  1024  585684 46480 SyS_epoll_ 7f335b3b15ca S android.process.acore
u0_a35    1748  1024  591012 47916 SyS_epoll_ 7f335b3b15ca S com.android.inputmethod.latin
radio     1759  1024  605200 56932 SyS_epoll_ 7f335b3b15ca S com.android.phone
system    1768  1024  600464 37396 SyS_epoll_ 7f335b3b15ca S com.android.settings:CryptKeeper
u0_a45    1802  1024  578316 35164 SyS_epoll_ 7f335b3b15ca S com.android.printspooler
u0_a6     1857  1024  582708 44920 SyS_epoll_ 7f335b3b15ca S android.process.media
u0_a49    1884  1024  575348 32740 SyS_epoll_ 7f335b3b15ca S com.android.smspush
u0_a22    1921  1024  591064 53796 SyS_epoll_ 7f335b3b15ca S com.android.calendar
u0_a26    1961  1024  584992 41036 SyS_epoll_ 7f335b3b15ca S com.android.deskclock
u0_a1     1975  1024  579424 39836 SyS_epoll_ 7f335b3b15ca S com.android.providers.calendar
u0_a50    2029  1024  594272 51788 SyS_epoll_ 7f335b3b15ca S com.android.messaging
u0_a46    2077  1024  575508 34796 SyS_epoll_ 7f335b3b15ca S com.android.provision
u0_a8     2092  1024  594000 59520 SyS_epoll_ 7f335b3b15ca S com.android.launcher
u0_a32    2110  1024  584612 37016 SyS_epoll_ 7f335b3b15ca S com.android.gallery3d
system    2138  1024  576712 33576 SyS_epoll_ 7f335b3b15ca S com.android.keychain
u0_a5     2146  1024  584824 35956 SyS_epoll_ 7f335b3b15ca S com.android.dialer
u0_a9     2175  1024  576716 33832 SyS_epoll_ 7f335b3b15ca S com.android.managedprovisioning
u0_a11    2192  1024  575592 33600 SyS_epoll_ 7f335b3b15ca S com.android.onetimeinitializer
u0_a23    2207  1024  584016 36100 SyS_epoll_ 7f335b3b15ca S com.android.camera2
u0_a29    2229  1024  595868 44472 SyS_epoll_ 7f335b3b15ca S com.android.email
u0_a10    2309  1024  575716 33140 SyS_epoll_ 7f335b3b15ca S com.android.musicfx
root      2347  1013  6116   1680  sigsuspend 7f90a6001827 S /system/bin/sh
root      2353  2347  5788   1428            0 7f60a4be3427 R ps
```

下面通过执行我的 setcolors 操作，将这两个进程分别赋予 1 和 2 颜色。

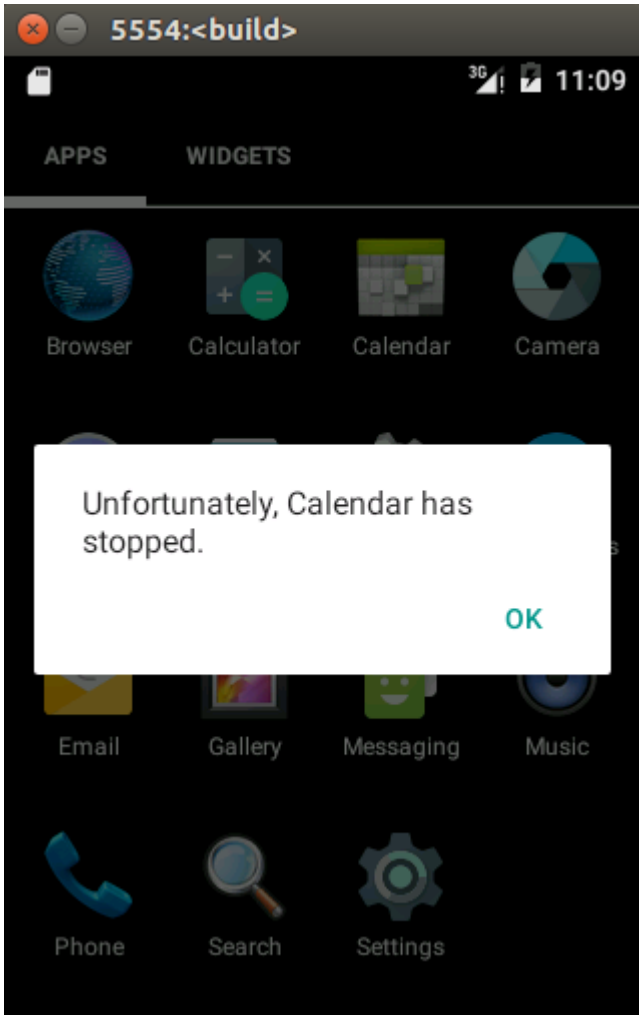
```
127|root@generic_x86_64:/test # ./setcolors com.android.calendar 1 com.android>
Proc Name: com.android.calendar, Proc ID: 1921
Proc Name: com.android.providers.calendar, Proc ID: 1975
Finish syscal - ret: 0
        Proc ID: 1921, color: 1, retval: 1
        Proc ID: 1975, color: 2, retval: 1
```

此时问题来了。在赋予了不同的颜色之后，点击 calendar 图标启动程序，calendar 并没有崩溃。而此时查看 emulator 的 log，却出现了进程之间通信的障碍（即图中的 transaction failed），而 1975 正是 com.android.providers.calendar 的进程号。

```
emulator: WARNING: UpdateCheck: failed to get the latest version, skipping check (current version 'invalid'
[  301.527390] healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a 2016-09-30 03:07:29.530000000 UTC
[  355.629683] binder: 1975:2006 transaction failed 29201, size 160-16
[  355.629727] binder: send failed reply for transaction 39499 to 1921:2045
[  355.630738] binder: 1975:1989 transaction failed 29201, size 160-16
[  355.630770] binder: send failed reply for transaction 39520 to 1921:2045
[  355.636439] binder: 1975:2006 transaction failed 29201, size 628-16
[  355.636480] binder: send failed reply for transaction 39578 to 1921:2358
[  355.640772] binder: 1975:1989 transaction failed 29201, size 628-16
[  355.640773] binder: send failed reply for transaction 39595 to 1921:2358
[  355.641832] binder: 1975:2006 transaction failed 29201, size 628-16
[  355.641833] binder: send failed reply for transaction 39623 to 1921:2358
[  355.642890] binder: 1975:1989 transaction failed 29201, size 628-16
[  355.642891] binder: send failed reply for transaction 39640 to 1921:2358
[  361.527398] healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a 2016-09-30 03:08:29.530000000 UTC
```

此时我又尝试再将进程名为 system_server 的颜色赋值为不同于其他两个的颜色。再打开 calendar 程序，成功崩溃。

```
root@generic_x86_64:/test # ./setcolors system_server 3
Proc Name: system_server, Proc ID: 1354
Finish syscal - ret: 0
        Proc ID: 1354, color: 3, retval: 1
```



事实上，android 中所有 binder 请求事件都是转交 system_server 进程处理的。将它标记为不同的颜色，自然 calendar 程序不能正常运行。而其他进程的默认 color 为 0，是可以与任何进程通信的，自然也就不会崩溃。至此，证明我添加的两个 system call 是可以正常使用的，结合 binder 机制也可以成功限制进程之间的通信。

Step 3: Test fork vfork and clone

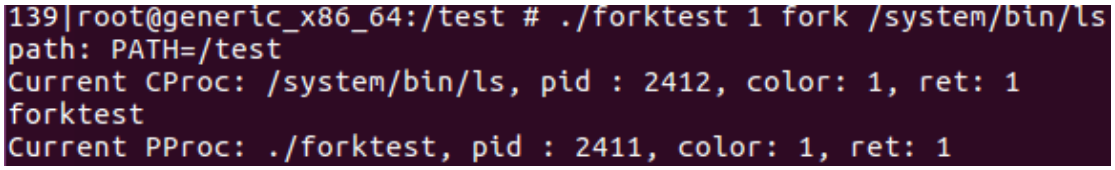
接下来的测试程序 forktest 是用来测试 fork，vfork 和 clone 之后的进程是否按照规则进程或者不继承父进程的颜色。

程序具体代码如下（之所以看上去比较复杂是因为 clone 的过程比较繁琐，需要分配新进程的栈空间等）。程序接受命令行参数./forktest <delay> fork|vfork|clone cmdline。如果输入不符合上述模式，则打印 usage。

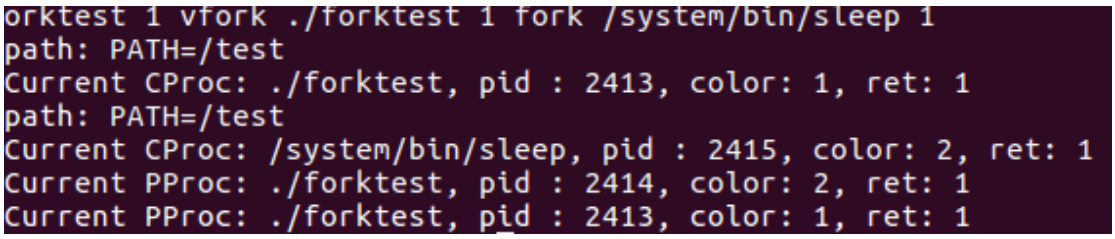
为了明显起见，程序的开始先给最初的父进程。/forktest 赋予 color 值 1。

```c
int myexec(char * name, char *argv[], char *envp[]){
    return execve(name, argv, envp);
}

typedef struct CloneArgv{
    char **argv;
    char **envp;
    char *name;
} CloneArgv;

int main(int argc, char* argv[]){
    unsigned int sec = atoi(argv[1]);
    int i, pathlen;
    sleep(sec);
    char **next_argv = (char **)malloc((argc - 1)*sizeof(char *));
    char **next_envp = (char **)malloc(2*sizeof(char *));
    char path[MAXBUFSIZE];

    pid_t pid = getpid();

    int retvals[1];
    u_int16_t colors[1] = { 1 };
    pid_t pids[1] = { pid };
    //long ret;
    long ret = syscall(__NR_setcolors, 1, pids, colors, retvals);

    getcwd(path, MAXBUFSIZE);
    pathlen = strlen(path);
    path[pathlen + 5] = '\0';
    for(i = pathlen - 1; i >= 0; --i)
        path[i + 5] = path[i];
    path[0] = 'P';
    path[1] = 'A';
    path[2] = 'T';
    path[3] = 'H';
    path[4] = '=';

    next_envp[0] = path;
    next_envp[1] = NULL;
    printf("path: %s\n", path);
```

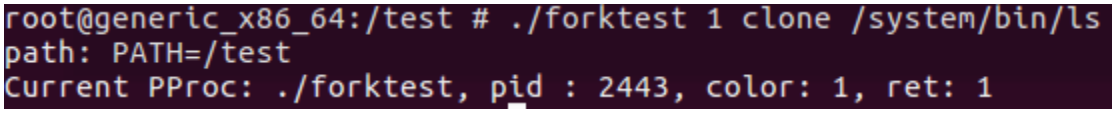
```c
for(i = 3; i < argc; ++i)
    next_argv[i - 3] = argv[i];
next_argv[i] = NULL;

if(argv[2][0] == 'f')
    pid = fork();
else if(argv[2][0] == 'v')
    pid = vfork();
else if(argv[2][0] == 'c'){
    void *stack = malloc(FIBERSTACK);
    CloneArgv *clone_argv = (CloneArgv *)malloc(sizeof(CloneArgv));
    clone_argv->name = argv[3];
    clone_argv->argv = next_argv;
    clone_argv->envp = next_envp;
    pid = clone(myexec, (char *)stack + FIBERSTACK, CLONE_VM|CLONE_VFORK, clone_argv);
}
else{
    printf("usage: ./forktest <delay> fork|clone|vfork cmdline\n");
    exit(-1);
}
if(pid == 0){
    pids[0] = getpid();
    colors[0] = -1;
    ret = syscall(__NR_getcolors, 1, pids, colors, retvals);
    printf("Current CProc: %s, pid : %d, color: %d, ret: %d\n", argv[3], getpid(), colors[0], retvals[0]);

    execve(argv[3], next_argv, next_envp);
    perror("execve");

    return 0;
}
else{
    wait(NULL);
    pids[0] = getpid();
    colors[0] = -1;
    ret = syscall(__NR_getcolors, 1, pids, colors, retvals);
    printf("Current PProc: %s, pid : %d, color: %d, ret: %d\n", argv[0], getpid(), colors[0], retvals[0]);

    return 0;
}
```



上图为测试 fork 的执行过程，可以看到，fork 之后的进程继承了父进程的颜色 1。



上图为测试 vfork 的过程，可以看到，第二个./forktest 进程并没有继承第一个的 color，而是在原来的 color 基础上加了 1（这是符合我的设计的）。而由第二个./forktest 进程派生的 ls 进程，继承了其颜色值 2.



上图为测试 clone 的过程。

Part 4: Summary

第一次 Hack 一个真正意义上存在的系统。见识了一些从前从未见识过的 C 代码，初步了解了系统编程的规范，体会到牵一发而动全身的情景。对后面的 Lab 充满期待。