Android OS Lab 2 Report

李炎 1400012951

Part A

本部分分为如下几个主要的步骤：

1,将进程的 nice 映射为五档，分别为-20, -10, 0, 10, 19.通过查询资料，我得知进程的 nice 值是通过其 task_struct 结构体中的 prio。但是 prio 又有多个，其中转换为 nice 值的 prio 为 static prio 字段。

在内核本来的设计中 Nice 与 prio 之间存在一一对应的关系，prio 与 nice 的转换关系在头文件 kernel/sched/sched.h 中给出。在该文件中，转换关系通过一组宏定义得到：如下图所示：

```
/*
 * Convert user-nice values [ -20 ... 0 ... 19 ]
 * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
 * and back.
 */
#define NICE_TO_PRIO(nice)  (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)  ((prio) - MAX_RT_PRIO - 20)
#define M_PRIO_TO_NICE(prio) prio
#define _PRIO_TO_NICE(prio) (((prio) - MAX_RT_PRIO - 20) != 19) && (((prio) - MAX_RT_PRIO - 20) != -20) ? PRIO_TO_NICE(prio) / 10 * 10 : M_PRIO_TO_NICE(prio)
#define TASK_NICE(p)        PRIO_TO_NICE((p)->static_prio)
#define _TASK_NICE(p)       _PRIO_TO_NICE((p)->static_prio)
```

起始带下划线的宏是我在本 lab 中定义的宏。按照要求，宏_PRIO_TO_NICE 将 prio 值映射为五个数，static prio 的值在 100 到 139 之间，通过减法运算得到-20 到 19 之间的数，通过除以 10 再乘以 10（当然，最大值和最小值的判断是必要的，这将导致宏定义比较长）就可以得到五档的 nice 值。

2,设计系统调用接口 getnicebypid，根据 pid 值得到进程的 nice 值。设计原理和上一个 lab 中类似。定义的系统调用号为 320,具体实现代码如下：

```c
/**
 * nr_pids: number of process; pids: array of pids of processes
 * to get nice values from; nices: array of nice values to store
 * requested nice values of processes; retval: array ro store return
 * values, -EINVAL for NULL task
 */
asmlinkage long sys_getnicebypid(int nr_pids, pid_t *pids, int *nices, int *retvals){
    int i;
    pid_t *kpids = (pid_t*)malloc(nr_pids*sizeof(pid_t));
    int *knices = (int*)malloc(nr_pids*sizeof(int));
    struct task_struct *task;

    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(knices, nices, nr_pids*sizeof(int)) != 0)
        return EFAULT;

    for(i = 0; i < nr_pids; ++i){
        task = find_task_by_vpid(pids[i]);
        if(task == NULL){
            retvals[i] = -EINVAL;
            continue;
        }
        nices[i] = _TASK_NICE(task);
        retvals[i] = 0;
    }

    return EXIT_SUCCESS;
}
```

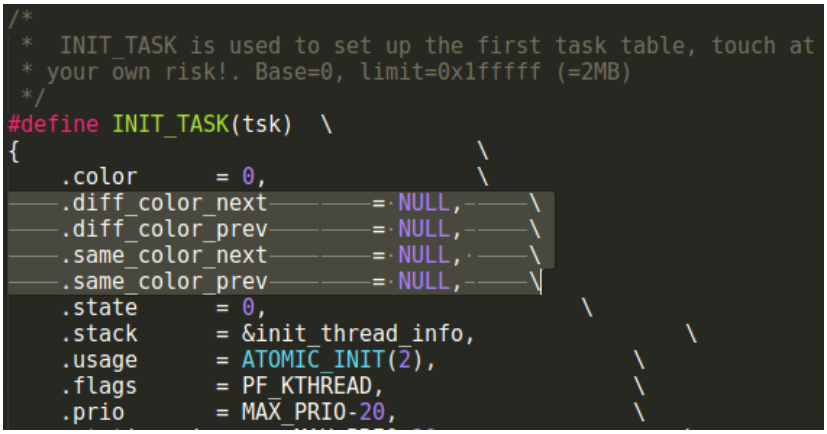
3,设计系统调用接口 getnicebycolor，根据给定的 color 值得到 color 值为该值的所有进程的 nice 值。

这一步的实现和前一步简单的由 pid 找到进程 nice 不同，需要一个数据结构将所有具有相同颜色的进程用一个双向链表链接起来，同时还需要将所有不同颜色的进程链表之间用链表链接，以降低搜索的复杂度。其实如果想改善搜索的复杂度仅仅通过链表来组织是不够的。但是鉴于本 lab 中 color 的赋予都是手工赋值的，而且 color 值为 0 的进程我是不予组织的（这将导致内核的崩溃，后文还会提到），如果 color 属性得到了广泛的应用，就应该考虑使用更加快捷的数据结构，如内存管理中使用的红黑树。

首先我在 task_struct 中定义了四个指针用来构建链表：

```c
/*linked list to link task with the same color and diffierent color*/
struct task_struct *same_color_next;
struct task_struct *same_color_prev;
struct task_struct *diff_color_next;
struct task_struct *diff_color_prev;
```

在进程初始化函数中将四个指针赋初始值为 NULL：



而在我写完所有的链表操作之后我发现其实内核提供了相应的链表操作用到的结构体--list_head。其实仔细想想这样做的目的时非常明显的，即内核中大量的数据需要通过链表进行链接，提供一个统一的接口无疑增强了代码的可维护性。list_head 实现了一系列关于链表的操作以及获取其所在的结构体引用的操作。如果想实现某个与链表相关的 feature，list_head 是最好的选择。

经过总结，我发现所有用到链表操作的地方如下所述：（1）setcolors。在将一个新的进程设置颜色时，需要将其加入到对应颜色的链表中。如果不存在该链表，则将其加入各种颜色的链表中。（2）fork。在父进程执行 fork 操作时，子进程默认继承其 color，但是 vfork 除外。（3）setnicebycolor。即本步要实现的系统调用，也是加入链表的目地所在。

具体设计中涉及的代码如下:

（1）setcolors：

```
/*Find process which is color 0*/
zero_task = find_task_by_vpid(1);
while(zero_task->color != 0)
    zero_task = zero_task->diff_color_prev;

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) && (colors[i] != 0)){
        top_task->color = colors[i];
        task->color = colors[i];
        retval[i] = 1;//Color succeed

        /*Insert a task node of linked list*/
        if(zero_task->diff_color_next == NULL){
            zero_task->diff_color_next = top_task;
            zero_task->diff_color_prev = top_task;
            top_task->diff_color_next = zero_task;
            top_task->diff_color_prev = zero_task;
        }
        else{
            temp_1 = zero_task;
            temp_2 = zero_task->diff_color_next;
            while((colors[i] > temp_2->color) && temp_2 != zero_task){
                temp_1 = temp_2;
                temp_2 = temp_1->diff_color_next;
            }
            if(colors[i] < temp_2->color){
                //no this color before
                temp_2->diff_color_next = top_task;
                top_task->diff_color_prev = temp_2;
                top_task->diff_color_next = temp_1;
                temp_1->diff_color_prev = top_task;
```

```
                //insert this task to top_task same color list
                if(task != top_task){
                    task->same_color_next = top_task;
                    top_task->same_color_next = task;
                    task->same_color_prev = top_task;
                    top_task->same_color_prev = task;
                }
            }
            else if(colors[i] == temp_2->color){
                //there's same color before, insert the top task into it
                if(temp_2->same_color_next == NULL){
                    temp_2->same_color_next = top_task;
                    top_task->same_color_next = temp_2;
                    temp_2->same_color_prev = top_task;
                    top_task->same_color_prev = temp_2;
                }
                else{
                    temp = temp_2->same_color_next;
                    while(temp != temp_2)
                        temp = temp->same_color_next;
                    temp->same_color_next = top_task;
                    top_task->same_color_prev = temp;
                    top_task->same_color_next = temp_2;
                    temp_2->same_color_prev = top_task;
                }

                //then insert current task into it
                temp = temp_2->same_color_next;
                while(temp != temp_2)
                    temp = temp->same_color_next;
                temp->same_color_next = task;
                task->same_color_prev = temp;
                task->same_color_next = temp_2;
                temp_2->same_color_prev = task;
            }
        }
    }
```

（2）fork，条件判断的目的为判断当前是否为 fork 操作（有别于 vfork 和 clone）

```
/*
 *If the copy operation is forking, modify the linked list about color
 */
if(current->color != 0){
    if(!(clone_flags & CLONE_VFORK)){
        if((clone_flags & CSIGNAL ) == SIGCHLD){
            if(current->same_color_next == NULL){
                current->same_color_next = p;
                p->same_color_next = current;
                current->same_color_prev = p;
                p->same_color_prev = current;
            }
            else{
                temp = current->same_color_next;
                while(temp != current)
                    temp = temp->same_color_next;
                temp->same_color_next = p;
                p->same_color_prev = temp;
                p->same_color_next = current;
                current->same_color_prev = p;
            }
        }
    }
}
```

（3）setnicebycolor

```
/**
 * color: colors to search
 * nice: nice value to set
 */
asmlinkage long sys_setnicebycolor(u_int64_t color, int nice){
    struct task_struct *start_task;
    struct task_struct *temp;
    struct task_struct *finded_task;
    int _find = 0;

    start_task = find_task_by_vpid(1);
    temp = start_task;

    if(temp == NULL)
        return -1;

    while(temp->diff_color_next != start_task){
        if(temp->color == color){
            _find = 1;
            break;
        }
        temp = temp->diff_color_next;
    }

    if(temp->diff_color_next == start_task && temp->color == color)
        _find = 1;

    if(_find == 0)
        return -2;

    finded_task = temp;
    finded_task->static_prio = NICE_TO_PRIO(nice);
    temp = temp->same_color_next;

    if(temp == NULL)
        return 1;

    while(temp != finded_task){
        temp->static_prio = NICE_TO_PRIO(nice);
        temp = temp->same_color_next;
    }

    return 0;
}
```

经过测试代码运行正常，测试如下：

```
/setcolors com.android.phone 1 com.android.calendar 1
Proc Name: com.android.phone, Proc ID: 1759
Proc Name: com.android.calendar, Proc ID: 1924
Finish syscal - ret: 0
        Proc ID: 1759, color: 1, retval: 1
        Proc ID: 1924, color: 1, retval: 1
root@generic_x86_64:/ # ./getnicebypid 1759
Segmentation fault
139|root@generic_x86_64:/ # ./getnicebypid 1 1759
pid[0]: 1759, nice[0]: 10, retval[0]: 0
root@generic_x86_64:/ # ./setnicebycolor 1 0
retval: 0
/setcolors com.android.phone 1 com.android.calendar 1
Proc Name: com.android.phone, Proc ID: 1759
Proc Name: com.android.calendar, Proc ID: 1924
Finish syscal - ret: 0
        Proc ID: 1759, color: 1, retval: 1
        Proc ID: 1924, color: 1, retval: 1
root@generic_x86_64:/ # ./getnicebypid 1 1759
pid[0]: 1759, nice[0]: 0, retval[0]: 0
root@generic_x86_64:/ # ./getnicebypid 1 1924
pid[0]: 1924, nice[0]: 0, retval[0]: 0
root@generic_x86_64:/ # ./setnicebycolor 1 10
retval: 0
root@generic_x86_64:/ # ./getnicebypid 1 1759
pid[0]: 1759, nice[0]: 10, retval[0]: 0
root@generic_x86_64:/ # ./setnicebycolor 1 10
retval: 0
root@generic_x86_64:/ # ./getnicebypid 1 1924
pid[0]: 1924, nice[0]: 10, retval[0]: 0
root@generic_x86_64:/ #
```

上述测试的步骤：（1）,设置两个进程颜色为 1;（2）,获取这两个进程当前的 nice 值，发现为 0;（3）,将颜色为 1 的进程 nice 设为 10;（4）,获取设置 nice 之后的进程的 nice 值，发现设置生效;（5）,检测完毕

Part B

本部分要求通过 cgroup 实现 cpu 利用率的读取，cpu 配额的读取和限制。具体的步骤如下：

API 1：getcpuusage

要求通过给定进程的 pid 获得改进程在一秒钟之内的菜谱占用率。我的思路如下：

（1）安卓系统中挂载了名为 cpuacct 的 cgroup_subsys（即子系统）。通过阅读源码，发现该子系统中有字段 cpuacct.usage，查资料得到该字段的含义为改进程自 cpu 加电起实际运行的时间。

（2）Cpuacct.usage 的时间单位为纳秒，即为 10^(-9)s。

（3）设计系统调用，获取当前进程已运行的时间。

（4）设计用户空间程序，调用两次上述系统调用，中间间隔一秒钟，两次得到的结果差值即为一秒内的运行时间。

其实本步是可以通过纯读用户空间读写文件解决的，但是为了与下边两个系统调用结合起来，我决定在内核空间实现。

具体实现如下所示：

（1）用户空间程序，代码非常简单：

```
    former_usage = syscall(__NR_getusage, pid);
    sleep(1);
    later_usage = syscall(__NR_getusage, pid);
    usage = later_usage - former_usage;
    per_usage = (1.0*usage)/(1.0*base)*100;
    printf("usage of %d: %lf%%\n", pid, per_usage);
```

（2）内核的系统调用，getusagebypid：

本部分的难点就在于如何根据进程 pid 通过一系列成员访问得到该进程所在的 cgroup 挂载的 cpuacct 子系统信息。

经过大量资料查找和源码阅读，最终我总结出了如下的方式：

（1）task_struct 中有 css_set 字段，指向一个 css_set 结构体：

```
#ifdef CONFIG_CGROUPS
    /* Control Group info protected by css_set_lock */
    struct css_set __rcu *cgroups;
    /* cg_list protected by css_set_lock and tsk->alloc_lock */
    struct list_head cg_list;
#endif
```

（2）css_set 中有一个 cgroup_subsys 指针数组：

```
/*
 * Set of subsystem states, one for each subsystem. This array
 * is immutable after creation apart from the init_css_set
 * during subsystem registration (at boot time) and modular subsystem
 * loading/unloading.
 */
struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
```

（3）通过上述的指针数组，以及经过多次尝试，发现下标为 2 的 subsys 中有 cpuacct 的相关信息。cgroup_subsys_state 中有一个 cgroup 指针，指向该进程的 cgroup：

```
/*
 * The cgroup that this subsystem is attached to. Useful
 * for subsystems that want to know about the cgroup
 * hierarchy structure
 */
struct cgroup *cgroup;
```

（4）cgroup 结构体中由一个指向 cgroupfs_root 的指针（关于这个结构体我还没彻底弄明白它的含义，初步认为是挂载在 cgroup 中子系统的根相关）：

```
struct cgroupfs_root *root;
```

（5）cgroupfs_root 中有一个链接 cgroup_subsys 结构体的链表头：

```
/* A list running through the attached subsystems */
struct list_head subsys_list;
```

（6）而 cgroup_subsys 结构体中的 base_cftype 指向了 cgroup 子系统具体函数指针的实现：

```
/*
 * read_u64() is a shortcut for the common case of returning a
 * single integer. Use it in place of read()
 */
u64 (*read_u64)(struct cgroup *cgrp, struct cftype *cft);
/*
 * read_s64() is a signed version of read_u64()
 */
s64 (*read_s64)(struct cgroup *cgrp, struct cftype *cft);
```

（7）cpuacct 相关的 cftype：

```
static struct cftype files[] = {
    {
        .name = "usage",
        .read_u64 = cpuusage_read,
        .write_u64 = cpuusage_write,
    },
    {
        .name = "usage_percpu",
        .read_seq_string = cpuacct_percpu_seq_read,
    },
    {
        .name = "stat",
        .read_map = cpuacct_stats_show,
    },
    { } /* terminate */
};
```

（8）综上，获得当前进程 usage 的路径如下：

task_struct->css_set->cgroup_subsys_state->cgroup->cgroupfs_root->cgroup_subsys->cftype->read_u64;

具体的实现代码如下所示：

```
asmlinkage long sys_getusage(pid_t pid){
    struct task_struct *task;
    struct cgroup *cgp;
    struct cgroup_subsys *temp_cgp_sub, *cgp_sub;
    struct list_head *cgp_header, *pos;
    struct cftype *ctp;
    int usage;
    int flag;

    task = find_task_by_vpid(pid);
    cgp = task->cgroups->subsys[2]->cgroup;
    cgp_header = &(cgp->root->subsys_list);

    flag = 0;
    list_for_each(pos, cgp_header){
        if(pos == NULL)
            return -5 - flag;
        temp_cgp_sub = list_entry(pos, struct cgroup_subsys, sibling);
        printk("current_cgroup_subsys_name: %s, flag: %d\n", temp_cgp_sub->name, flag);
        //if(temp_cgp_sub->name == "cpuacct")
            //break;
        if(!(strcmp(temp_cgp_sub->name, "cpuacct"))) break;
        flag++;
    }

    cgp_sub = temp_cgp_sub;
    //if(cgp_sub->name != "cpuacct") return -1;
    if(strcmp(cgp_sub->name, "cpuacct")) return -1;

    ctp = cgp_sub->base_cftypes;

    usage = ctp->read_u64(cgp, ctp);

    return usage;
}
```

API 2, getquota

本步和上一步类似，不同之处在于 quota 值是在 cpu 子系统中获取的。相应的 cftype 结构如下：

```
static struct cftype cpu_files[] = {
#ifdef CONFIG_FAIR_GROUP_SCHED
    {
        .name = "shares",
        .read_u64 = cpu_shares_read_u64,
        .write_u64 = cpu_shares_write_u64,
    },
#endif
#ifdef CONFIG_CFS_BANDWIDTH
    {
        .name = "cfs_quota_us",
        .read_s64 = cpu_cfs_quota_read_s64,
        .write_s64 = cpu_cfs_quota_write_s64,
    },
    {
        .name = "cfs_period_us",
        .read_u64 = cpu_cfs_period_read_u64,
        .write_u64 = cpu_cfs_period_write_u64,
    },
    {
        .name = "stat",
        .read_map = cpu_stats_show,
    },
#endif
#ifdef CONFIG_RT_GROUP_SCHED
    {
        .name = "rt_runtime_us",
        .read_s64 = cpu_rt_runtime_read,
        .write_s64 = cpu_rt_runtime_write,
    },
    {
        .name = "rt_period_us",
        .read_u64 = cpu_rt_period_read_uint,
        .write_u64 = cpu_rt_period_write_uint,
    },
#endif
    { } /* terminate */
};
```

本步用到的即为 cfs_quota_us 文件。

需要注意的是，cpu 子系统的 cfs 选项默认是不开启的。只有当宏 CONFIG_CSF_BANDWIDTH 定义之后才默认将其开启。所以有两个地方需要改动：

（1）define 该宏

```
#define CONFIG_CFS_BANDWIDTH
```

（2）修改 config 文件：

```
CONFIG_CFS_BANDWIDTH=y
```

代码实现部分和上一部分差不多：

```
if(task_leader->color != color){
    if(temp == NULL){
        return -1;
    }
    while(temp != task_leader && temp->color != color){
        temp = temp->diff_color_next;
        if(temp == NULL) {
            return -1;
        }
    }

    if(temp->color == color)
        task_leader = temp;
    else{
        return -1;
    }
}

task = task_leader;

do{
    printk("cricle: %d\n", cricle++);
    flag = 0;
    cgp = task->cgroups->subsys[1]->cgroup;
    cgp_header = &(cgp->root->subsys_list);

    if(cgp_header == NULL) return 5;

    list_for_each(pos, cgp_header){
        if(pos == NULL)
            return -5 - flag;
        temp_cgp_sub = list_entry(pos, struct cgroup_subsys, sibling);
        printk("current_cgroup_subsys_name: %s, flag: %d\n", temp_cgp_sub->name, ++flag);
        if(!(strcmp(temp_cgp_sub->name, "cpu")))
            break;
    }

    cgp_sub = temp_cgp_sub;
    printk("cg_subsys_name: %s\n", cgp_sub->name);
    ctp = cgp_sub->base_cftypes;
    if((++ctp)->read_s64 == NULL) return 5;
    printk("cftype name: %s\n", ctp->name);

    quota += ctp->read_s64(cgp, ctp);
    task = task->same_color_next;
}while(task != task_leader && task != NULL);

retvals[0] = quota;
```

上图为代码的主要部分，完整代码见代码包

测试如下：

```
root@generic_x86_64:/ # ./getquota 1
color: 1, quota: -1, ret: 0
root@generic_x86_64:/ #
```

默认进程的 cpu 配额时没有限制，为-1，测试完毕。

API 3 setquota

根据上一步的分析，我观察了 cpu 子系统中 cftype 的结构，发现其有一个文件字段为 cfs_quota_write。于是想当然的将上述代码稍加修改，试图直接使用 write_s64。但是在有限的时间里一直没有找到内核中新建一个 cgroup 子目录的接口。根 cgroup 目录是不允许执行 write 操作的：

```
static int tg_set_cfs_bandwidth(struct task_group *tg, u64 period, u64 quota)
{
    int i, ret = 0, runtime_enabled, runtime_was_enabled;
    struct cfs_bandwidth *cfs_b = &tg->cfs_bandwidth;

    if (tg == &root_task_group)
        return -EINVAL;
```

这样的限制的目的也比较明显,就是如果可以很容易的修改根 cgroup 的 cpu 配额，那么一方面内核很容易崩溃;另一方面 quota 一般是根据任务的 cpu 使用需求和能力定制的，不可一概而论。

这里需要说明一点，所谓实现系统调用，在内核空间实现新的 color cgroup 的创建，指的是使用 cgroup 中提供的 API，而非在内核空间直接调用系统调用（比如，在用户空间的 mkdir 在内核空间就是 sys_mkdir，用户空间的 write 在内核空间就是 sys_write）。如果系统调用中都是类似 sys_xxx 这类的函数，那么这样的所谓“系统掉用”是完全可以在用户空间实现的。

而如果想实现内核空间的 cgroup 创建，无非需要调用 cgroup_create 之类的函数。一开始我进行了这样的尝试，使用内核提供的 API 新建一个 cgroup，将之作为根 cgroup 的子节点。到这里还一切正常，但是最终这个 API 在反悔时总是会导致 kernel 的崩溃。我找了很长时间，也没有发现 bug 到底在哪里导致的。

所以此 API 我转为使用直接在文件系统中 cpu 子系统挂载目录下新建目录的解法。安卓系统中 cpu 子系统的默认挂载目录为/dev/cpuctl/。具体代码如下：

（1）下列函数将 color 为指定 color 的进程 pid 写入文件/dev/cpuctl/<color>/tasks

```c
int SetTask(u_int16_t color, char *file_name)
{
    printf("color to be searched: %d\n", color);
    printf("file_name: %s\n", file_name);
    if(color == 0) return -1;
    printf("set entered\n");
    FILE *f = fopen(file_name, "w");
    if(f == NULL) return -1;

    int ret[1], pids[1];
    u_int16_t colors[1];
    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;
        pids[0] = pid;
        syscall(__NR_getcolors, 1, pids, colors, ret);
        if(colors[0] == color)
            fprintf(f, "%d\n", pid);
    }
    fclose(f);
    printf("set quited\n");
    return 0;
}
```

（2）下列函数将用户输入的 cpu 配额写入/dev/cpuctl/<color>/cpu.cfs_quota_us

```c
//return the file directory
char* AddCgroup(int color, int quota){
    printf("color: %d, quota: %d\n", color, quota);
    char dir_name[] = "/dev/cpuctl/";
    char file_name[] = "/dev/cpuctl/";
    char color_str[20];
    sprintf(color_str, "%d", color);
    const char dir_postfix[] = "/cpu.cfs_quota_us";
    const char dir_postfix_tasks[] = "/tasks";
    char *retstr = (char *)malloc(100*sizeof(char));
    int len, i;

    strcat(dir_name, color_str);
    strcat(file_name, color_str);

    mkdir(dir_name, S_IRWXG);
    strcat(dir_name, dir_postfix);

    FILE *f = fopen(dir_name, "r+");
    fprintf(f, "%d\n", quota);
    fclose(f);

    strcat(file_name, dir_postfix_tasks);
    printf("file_name : %s\n", file_name);

    len = strlen(file_name);
    for(i = 0; i < len; ++i)
        retstr[i] = file_name[i];
    retstr[len] = '\0';

    return retstr;
}
```

（3）测试如下：

```
root@generic_x86_64:/ # ./setquota 1 2000000
[main] color: 1, quota: 2000000
file_name : /dev/cpuctl/1/tasks
color to be searched: 1
file_name: /dev/cpuctl/1/tasks
Success!
root@generic_x86_64:/ # cat /dev/cpuctl/1/tasks
1759
1766
1778
root@generic_x86_64:/ # cat /dev/cpuctl/1/cpu.cfs_quota_us
2000000
root@generic_x86_64:/ #
```

测试步骤：（1）将颜色为 1 的三个进程的 cpu 配额设置为 2000000;（2）查看 cpu 子系统挂载目录下 tasks 文件内容，发现三个进程 id 都被写入了;（3）查看 cpu 子系统挂载目录下 cpu.cfs_quota_us 文件的内容，发现已经变成 2000000;（4）检测完毕。

Part C

至于最终的 Part C，我认为没有什么做的必要性，只是将前两部分的 API 封装在一个 shell 中。甚至可以使用 shell 脚本来写。所以最终我只完成了前两部分的 API 构建。