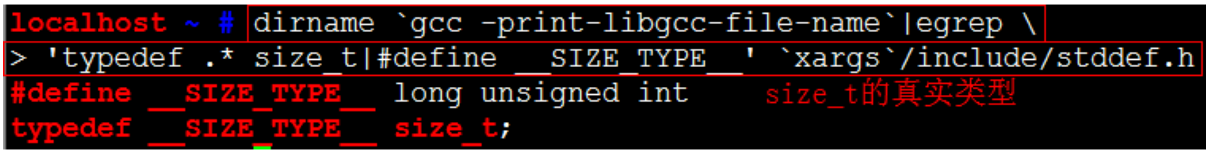
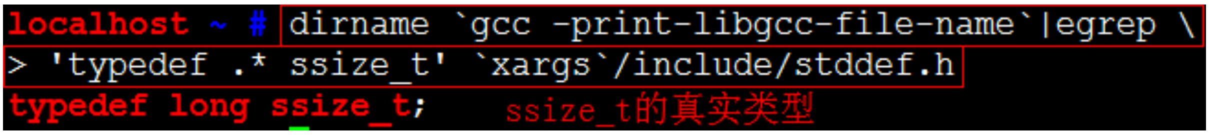
第三章 文件IO

size_t是一个无符号长整数类型，ssize_t是一个有符号长整数类型。

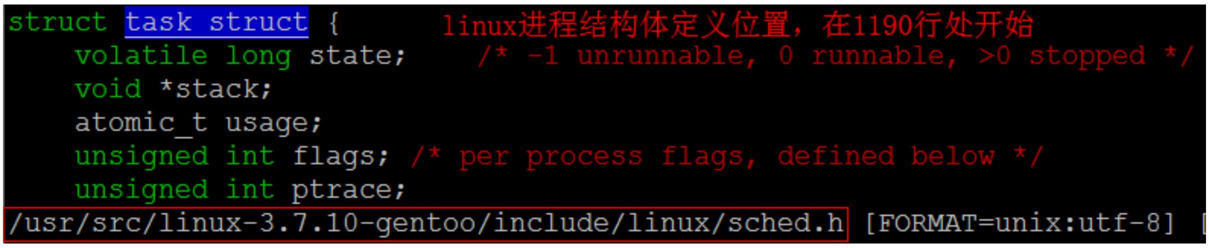


```
localhost ~ # dirname `gcc -print-libgcc-file-name`|egrep \
> 'typedef .* size_t|#define __SIZE_TYPE__ ' `xargs`/include/stddef.h
#define __SIZE_TYPE__ long unsigned int     size_t的真实类型
typedef __SIZE_TYPE__ size_t;
```



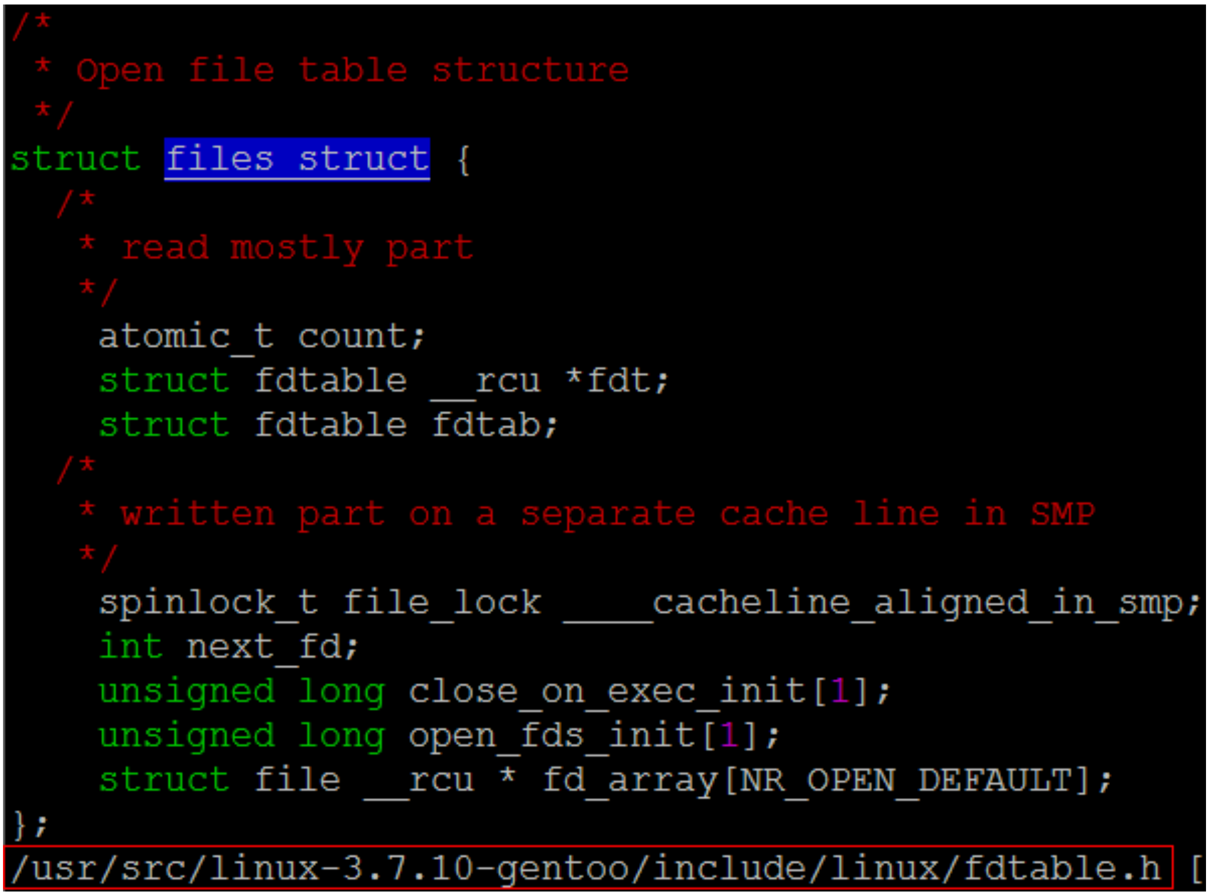
```
localhost ~ # dirname `gcc -print-libgcc-file-name`|egrep \
> 'typedef .* ssize_t' `xargs`/include/stddef.h
typedef long ssize_t;     ssize_t的真实类型
```

进程在内核中的结构体



```
struct task_struct {      linux进程结构体定义位置，在1190行处开始
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;
/usr/src/linux-3.7.10-gentoo/include/linux/sched.h [FORMAT=unix:utf-8] [
```

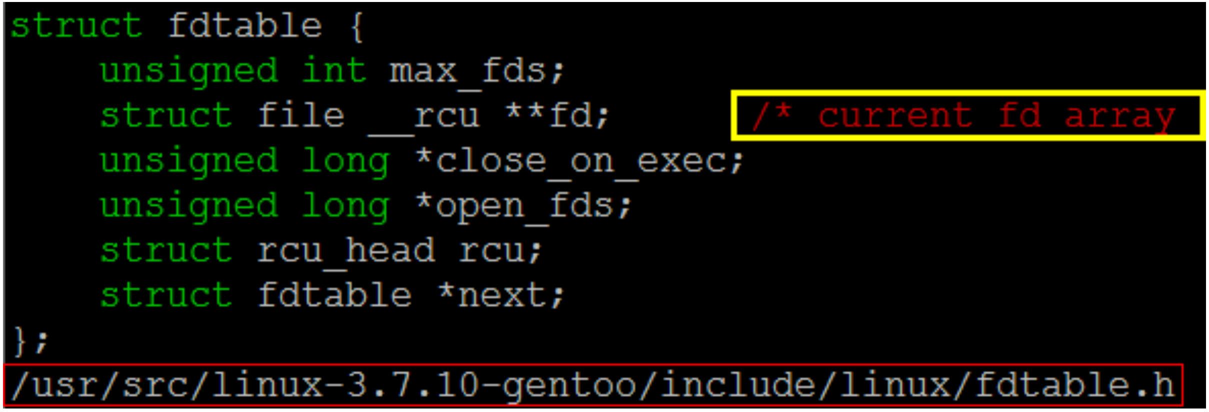
总约360行。



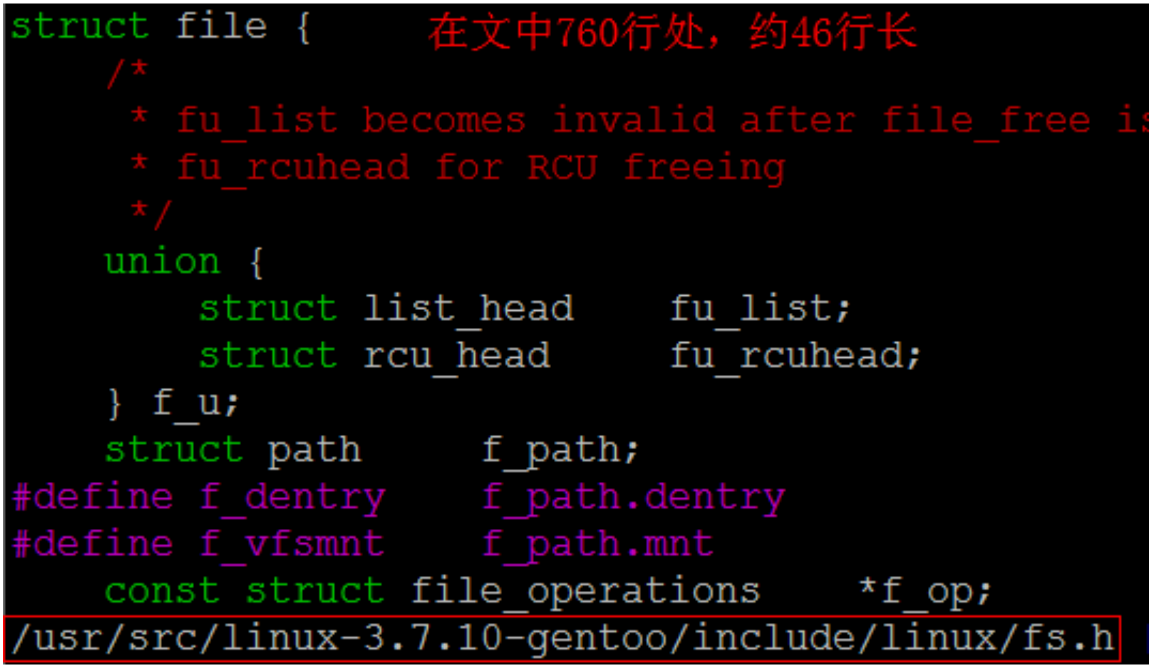
```
/*
 * Open file table structure
 */
struct files_struct {
  /*
   * read mostly part
   */
    atomic_t count;
    struct fdtable __rcu *fdt;
    struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
    spinlock_t file_lock ____cacheline_aligned_in_smp;
    int next_fd;
    unsigned long close_on_exec_init[1];
    unsigned long open_fds_init[1];
    struct file __rcu * fd_array[NR_OPEN_DEFAULT];
};
/usr/src/linux-3.7.10-gentoo/include/linux/fdtable.h [
```

关于__rcu的参考

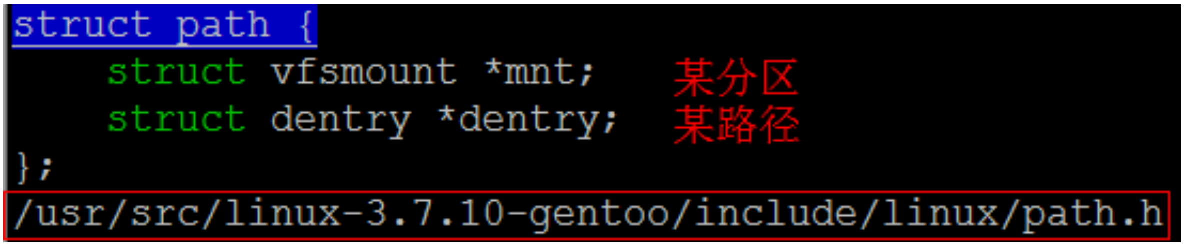
http://stackoverflow.com/questions/17128210/what-does-rcu-stands-for-in-linux
http://www.ibm.com/developerworks/cn/linux/l-rcu/
http://blog.sina.com.cn/s/blog_7e719f0501012tkt.html
http://blog.csdn.net/am_111/article/details/6553523



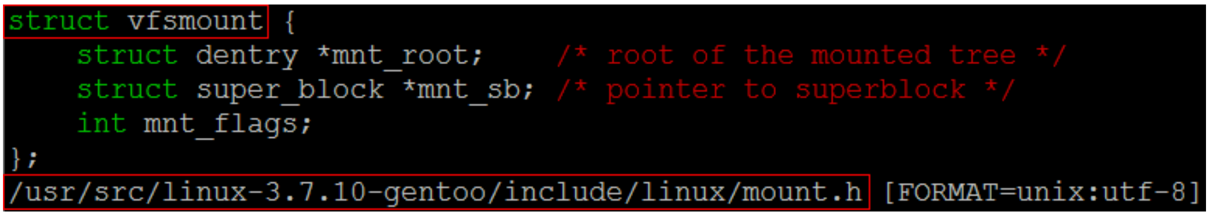
```
struct fdtable {
    unsigned int max_fds;
    struct file __rcu **fd;      /* current fd array
    unsigned long *close_on_exec;
    unsigned long *open_fds;
    struct rcu_head rcu;
    struct fdtable *next;
};
/usr/src/linux-3.7.10-gentoo/include/linux/fdtable.h
```



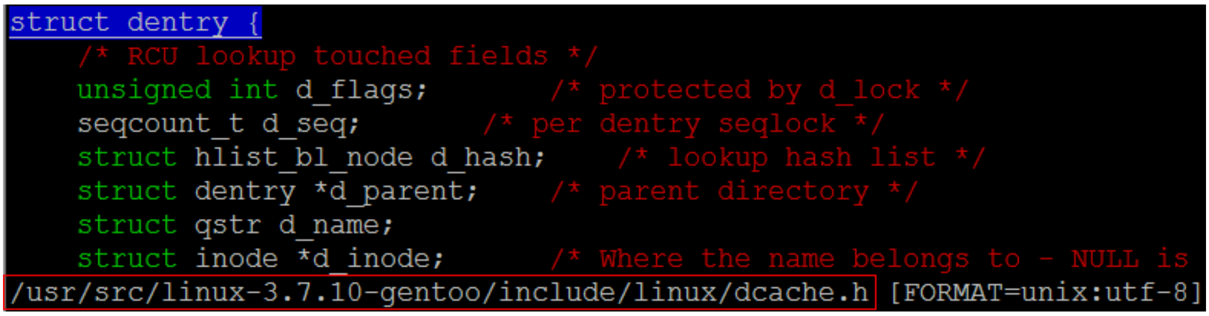
```
struct file {     在文中760行处，约46行长
    /*
     * fu_list becomes invalid after file_free i
     * fu_rcuhead for RCU freeing
     */
    union {
        struct list_head     fu_list;
        struct rcu_head      fu_rcuhead;
    } f_u;
    struct path     f_path;
#define f_dentry    f_path.dentry
#define f_vfsmnt    f_path.mnt
    const struct file_operations    *f_op;
/usr/src/linux-3.7.10-gentoo/include/linux/fs.h
```



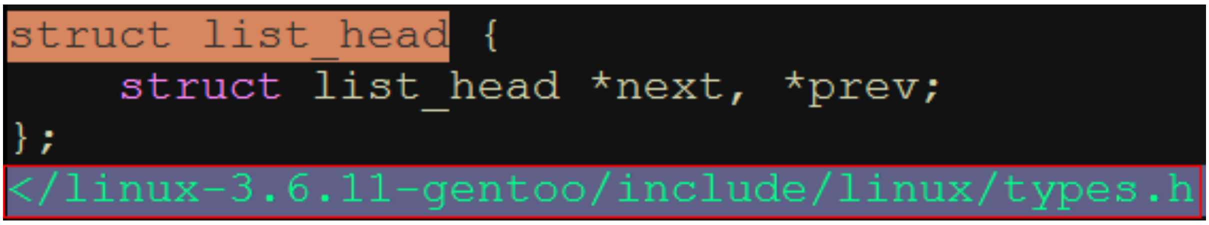
```
struct path {
    struct vfsmount *mnt;    某分区
    struct dentry *dentry;   某路径
};
/usr/src/linux-3.7.10-gentoo/include/linux/path.h
```



```
struct vfsmount {
	struct dentry *mnt_root;	/* root of the mounted tree */
	struct super_block *mnt_sb;	/* pointer to superblock */
	int mnt_flags;
};
/usr/src/linux-3.7.10-gentoo/include/linux/mount.h [FORMAT=unix:utf-8]
```



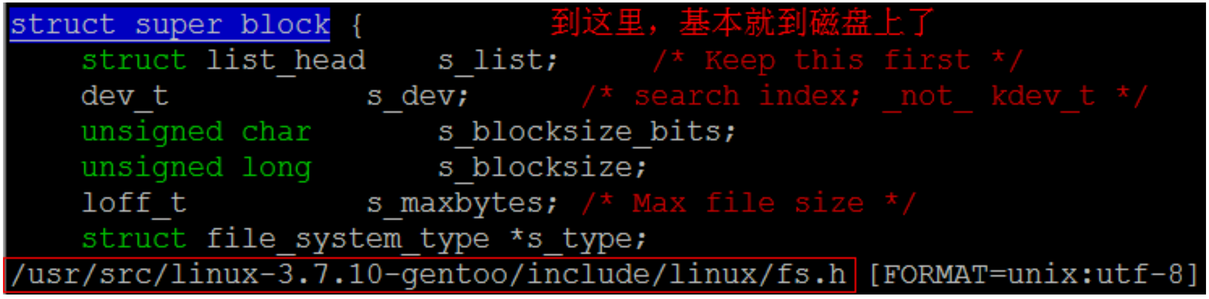
```
struct dentry {
	/* RCU lookup touched fields */
	unsigned int d_flags;		/* protected by d_lock */
	seqcount_t d_seq;		/* per dentry seqlock */
	struct hlist_bl_node d_hash;	/* lookup hash list */
	struct dentry *d_parent;	/* parent directory */
	struct qstr d_name;
	struct inode *d_inode;		/* Where the name belongs to - NULL is
/usr/src/linux-3.7.10-gentoo/include/linux/dcache.h [FORMAT=unix:utf-8]
```



```
struct list_head {
	struct list_head *next, *prev;
};
</linux-3.6.11-gentoo/include/linux/types.h
```

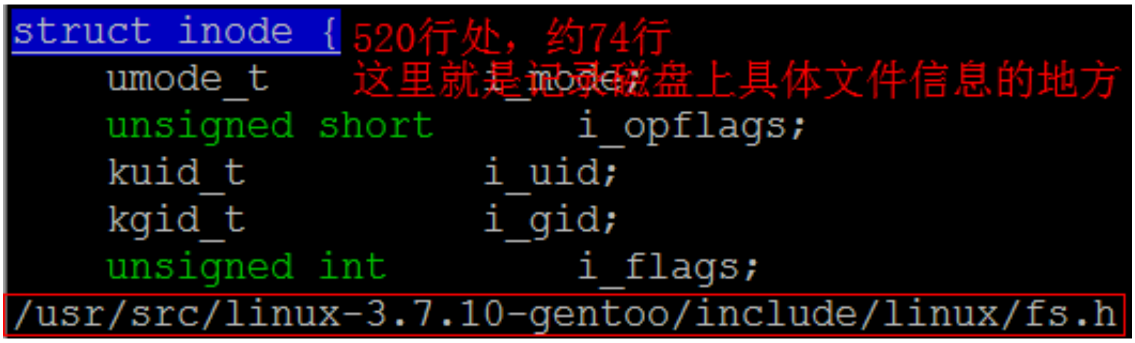
从代码看，它是个双向列表，如果把列表的头尾相接，就组成了双向循环列表（空双向列表除外）。

http://www.ibm.com/developerworks/cn/linux/kernel/l-chain/



到这里，基本就到磁盘上了

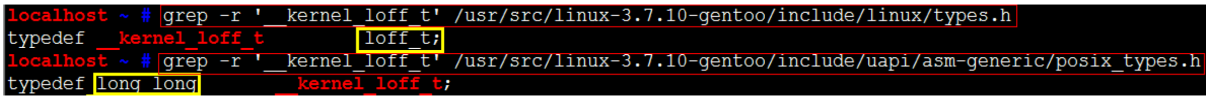
```
struct super_block {
	struct list_head	s_list;		/* Keep this first */
	dev_t			s_dev;		/* search index; _not_ kdev_t */
	unsigned char		s_blocksize_bits;
	unsigned long		s_blocksize;
	loff_t			s_maxbytes;	/* Max file size */
	struct file_system_type	*s_type;
/usr/src/linux-3.7.10-gentoo/include/linux/fs.h [FORMAT=unix:utf-8]
```



520行处，约74行

这里就是记录磁盘上具体文件信息的地方

```
struct inode {
	umode_t			i_mode;
	unsigned short		i_opflags;
	kuid_t			i_uid;
	kgid_t			i_gid;
	unsigned int		i_flags;
/usr/src/linux-3.7.10-gentoo/include/linux/fs.h
```



```
localhost ~ # grep -r '__kernel_loff_t' /usr/src/linux-3.7.10-gentoo/include/linux/types.h
typedef __kernel_loff_t		loff_t;
localhost ~ # grep -r '__kernel_loff_t' /usr/src/linux-3.7.10-gentoo/include/uapi/asm-generic/posix_types.h
typedef long long	__kernel_loff_t;
```

由上可见，当前进程的文件描述符表，就是files_struct结构体中的

fd_array成员；文件描述符所指的文件表，是file结构体；当前进程操作的文件表，是fd_tab中的fd成员所指的地方；linux中的v节点，应该是inode结构体。

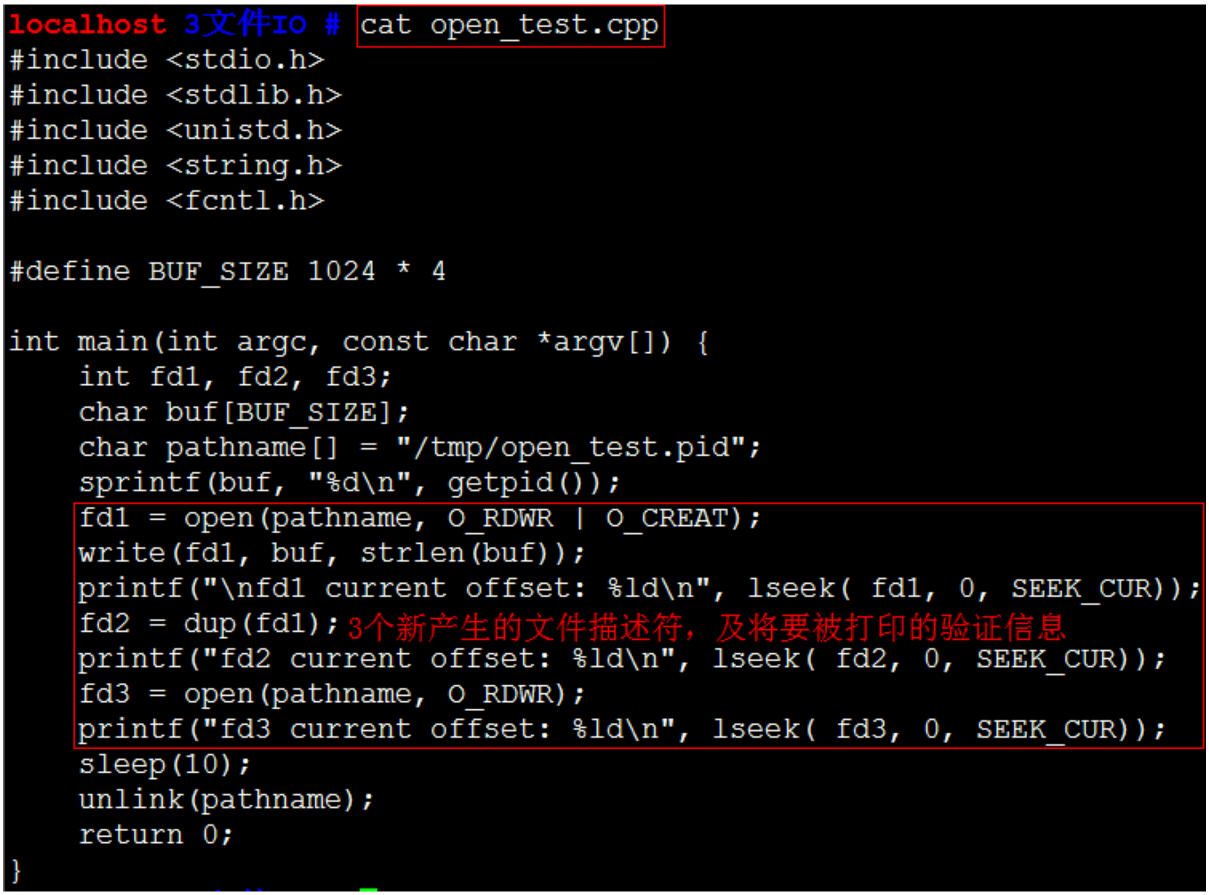
习题

1、文章中的读写函数没有缓存机制，它由POSIX定义，这里是系统内核对它进行了实现。读写缓冲器需程序员自己设计、定义，所以它本身不具缓冲功能。

2、解题思路：修改进程的属性，需在内核态来操作(用户态没权修改)。进入内核态修改数据最简单的方法是通过系统调用。那么该题的出题者，应该是希望读者通过本章的dup函数来实现dup2的功能。

3、通过fcntl对fd1使用F_SETFD命令，仅会改变fd1的文件描述符值。如果对它使用F_SETFL，则会影响到，所有与它指向的**文件表相同**的**文件描述符**，因为F_SETFL修改的是fd1指向的文件表。

注：每次调用open函数，就会分配一个新的文件表项

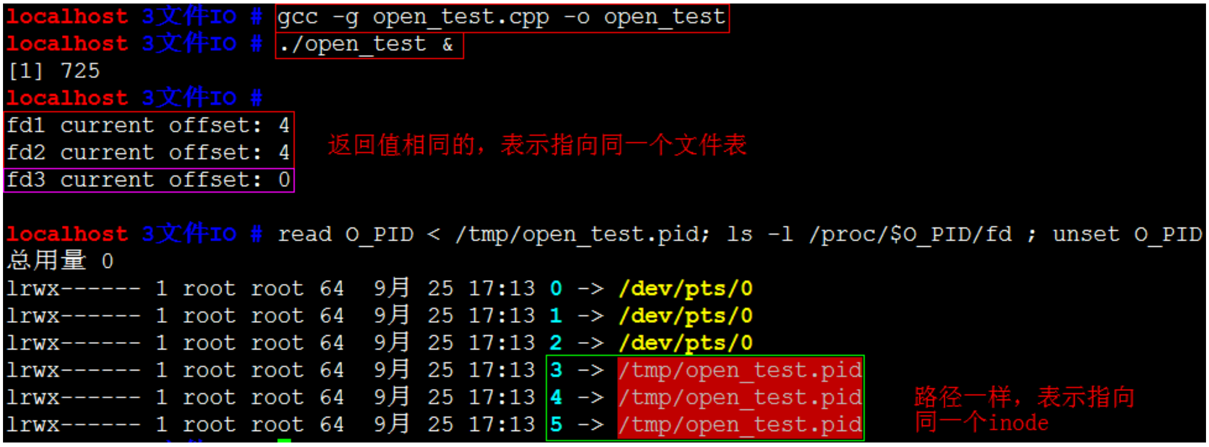


```
localhost 3文件IO # cat open_test.cpp
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

#define BUF_SIZE 1024 * 4

int main(int argc, const char *argv[]) {
    int fd1, fd2, fd3;
    char buf[BUF_SIZE];
    char pathname[] = "/tmp/open_test.pid";
    sprintf(buf, "%d\n", getpid());
    fd1 = open(pathname, O_RDWR | O_CREAT);
    write(fd1, buf, strlen(buf));
    printf("\nfd1 current offset: %ld\n", lseek( fd1, 0, SEEK_CUR));
    fd2 = dup(fd1);
    printf("fd2 current offset: %ld\n", lseek( fd2, 0, SEEK_CUR));
    fd3 = open(pathname, O_RDWR);
    printf("fd3 current offset: %ld\n", lseek( fd3, 0, SEEK_CUR));
    sleep(10);
    unlink(pathname);
    return 0;
}
```

3个新产生的文件描述符，及将要被打印的验证信息

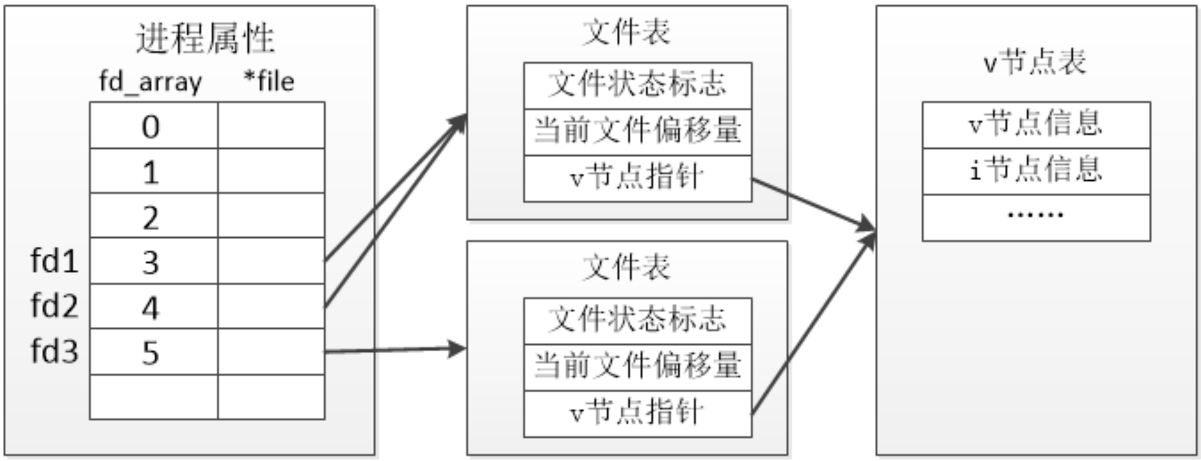


```
localhost 3文件IO # gcc -g open_test.cpp -o open_test
localhost 3文件IO # ./open_test &
[1] 725
localhost 3文件IO #
fd1 current offset: 4
fd2 current offset: 4
fd3 current offset: 0

localhost 3文件IO # read O_PID < /tmp/open_test.pid; ls -l /proc/$O_PID/fd ; unset O_PID
总用量 0
lrwx------ 1 root root 64  9月  25 17:13 0 -> /dev/pts/0
lrwx------ 1 root root 64  9月  25 17:13 1 -> /dev/pts/0
lrwx------ 1 root root 64  9月  25 17:13 2 -> /dev/pts/0
lrwx------ 1 root root 64  9月  25 17:13 3 -> /tmp/open_test.pid
lrwx------ 1 root root 64  9月  25 17:13 4 -> /tmp/open_test.pid
lrwx------ 1 root root 64  9月  25 17:13 5 -> /tmp/open_test.pid
```

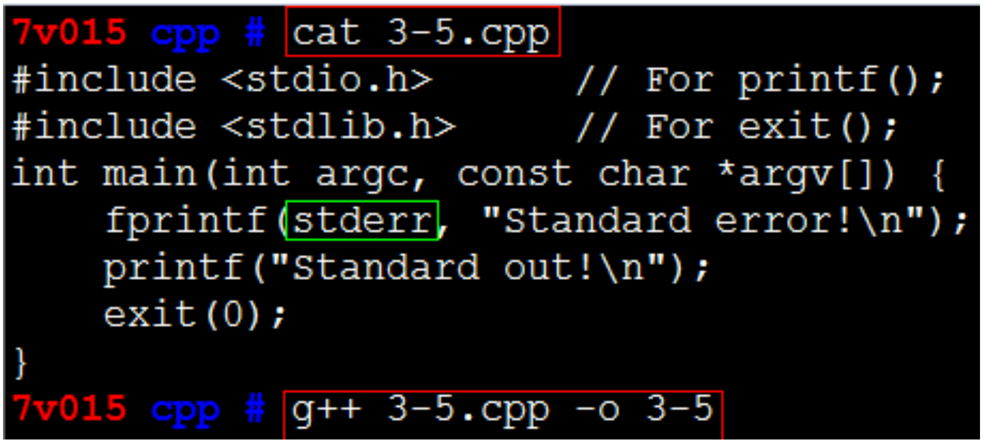
返回值相同的，表示指向同一个文件表

路径一样，表示指向
同一个inode

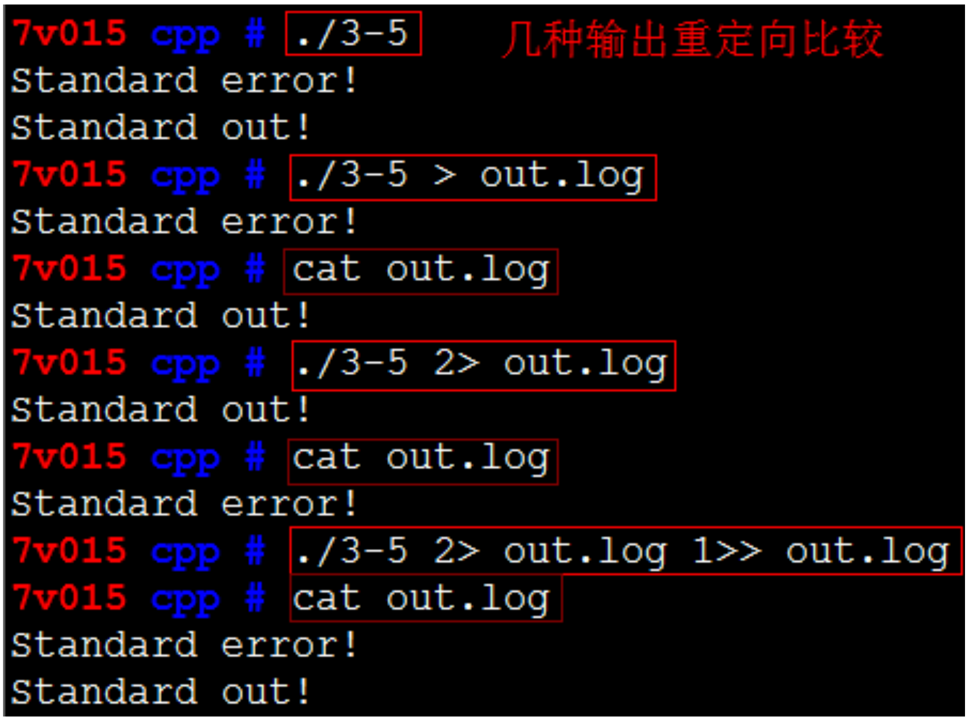


4、“许多程序中都包含下面一段代码”，题中这句话没看出用意。这段代码的目的，不是很清楚。dup2(int filedes1, int filedes2)函数的特点，3.12章说的很清楚，这里再重复一遍：使文件描述符filedes2指向filedes1所指的文件表；如果filedes2已经指向另外的文件表，则先关闭它，再打开并指向filedes1所指的文件表；如果filedes1与filedes2所指文件表相同，则直接返回。

5、在维护linux服务器时，经常会用这样的命令来捕获完整的日志信息到文件中。这里的符号‘&’，应该与c语言中取地址符的意思相同(2>&1 换成c风格，应当如2 = &1)。

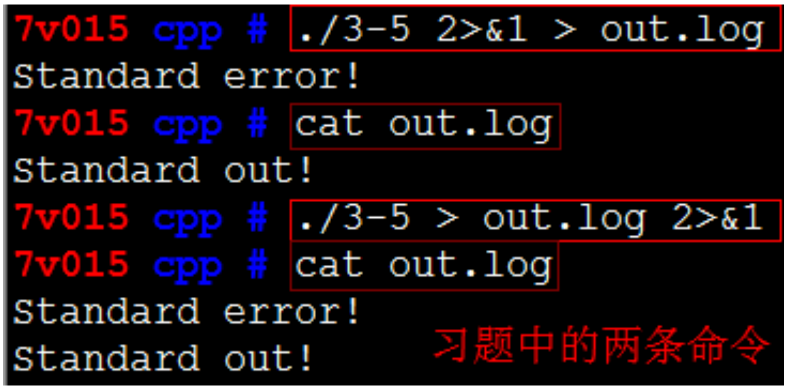


```
7v015 cpp # cat 3-5.cpp
#include <stdio.h>       // For printf();
#include <stdlib.h>      // For exit();
int main(int argc, const char *argv[]) {
    fprintf(stderr, "Standard error!\n");
    printf("Standard out!\n");
    exit(0);
}
7v015 cpp # g++ 3-5.cpp -o 3-5
```



几种输出重定向比较

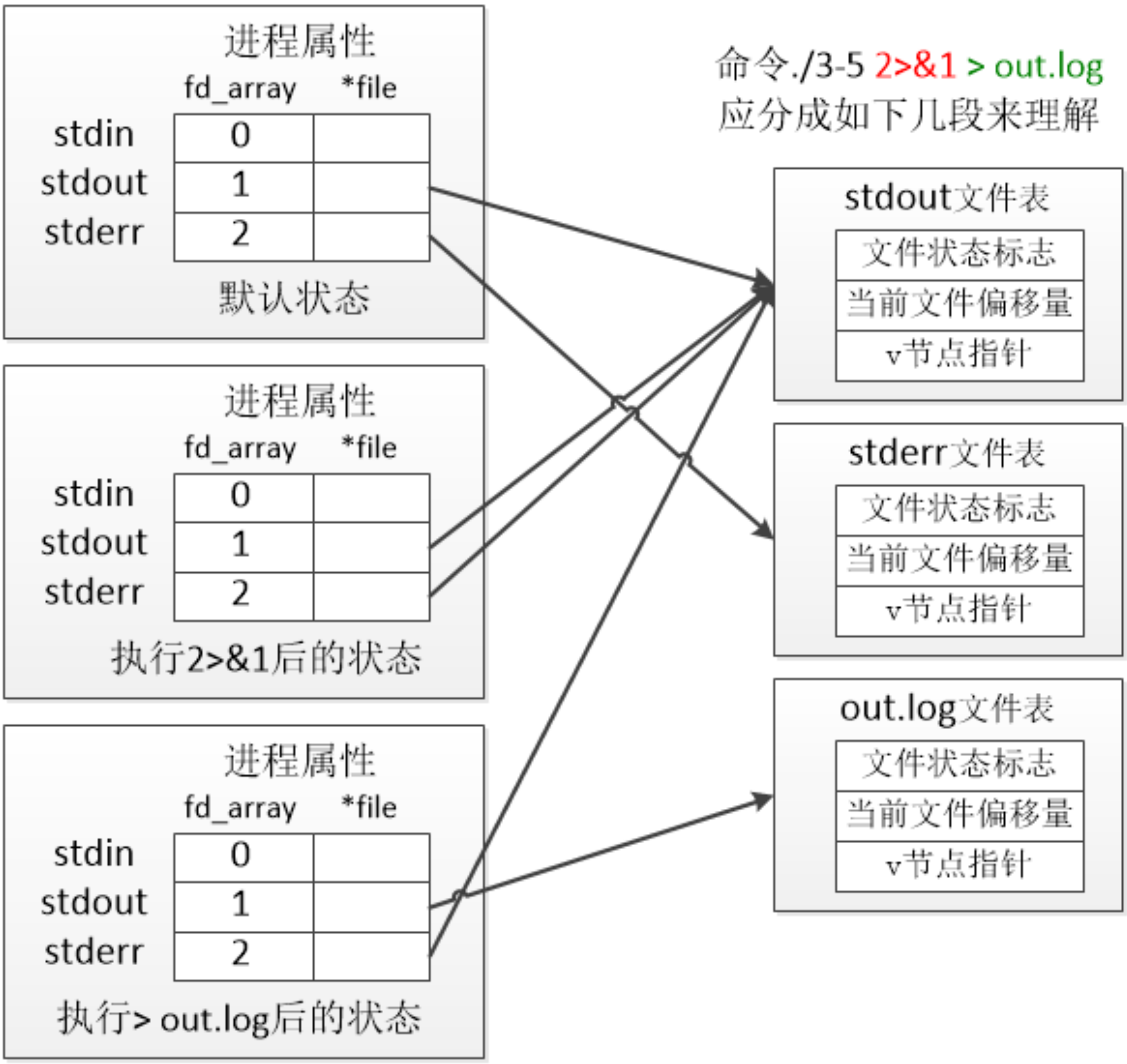
```
7v015 cpp # ./3-5
Standard error!
Standard out!
7v015 cpp # ./3-5 > out.log
Standard error!
7v015 cpp # cat out.log
Standard out!
7v015 cpp # ./3-5 2> out.log
Standard out!
7v015 cpp # cat out.log
Standard error!
7v015 cpp # ./3-5 2> out.log 1>> out.log
7v015 cpp # cat out.log
Standard error!
Standard out!
```

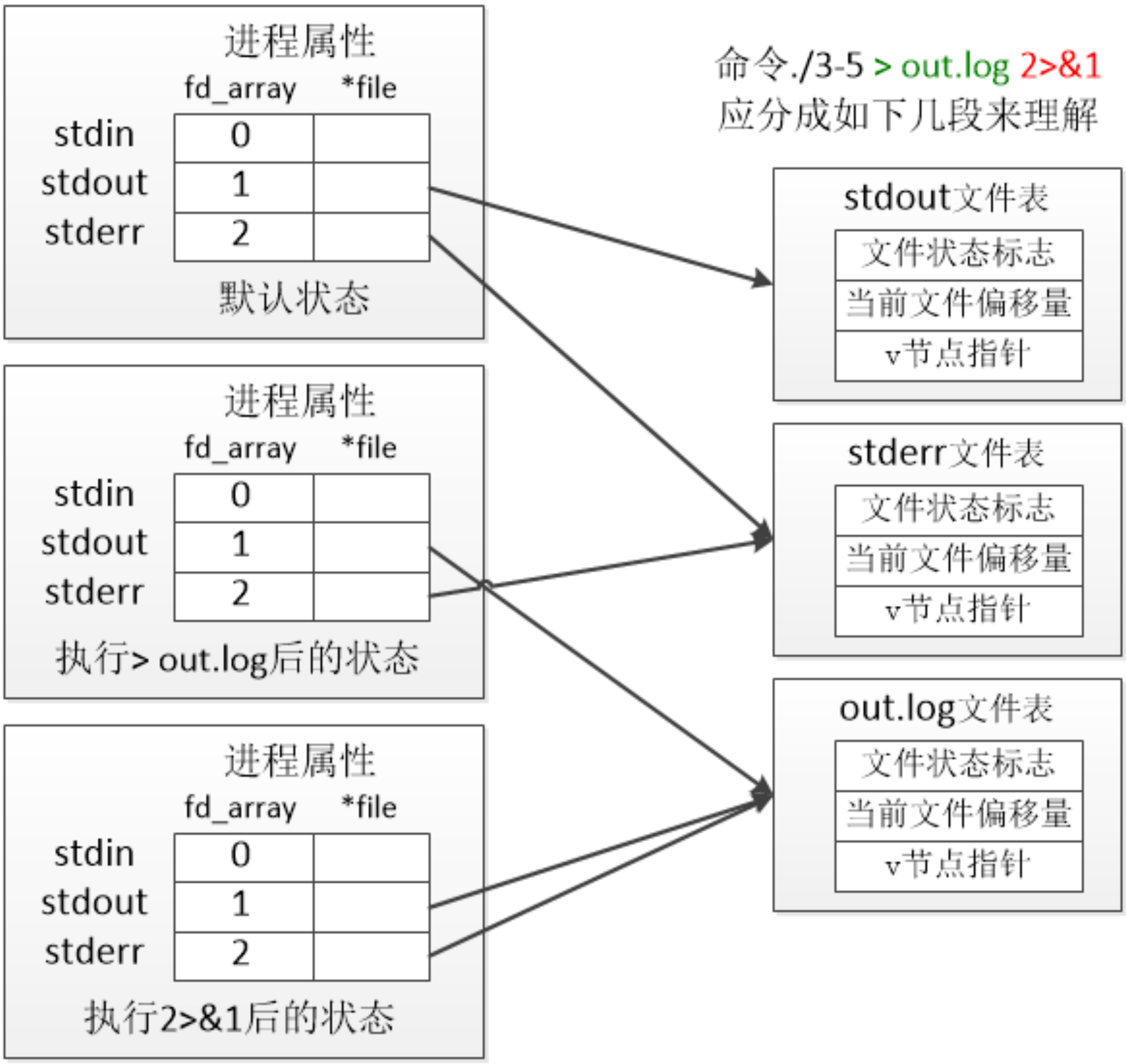


```
7v015 cpp # ./3-5 2>&1 > out.log
Standard error!
7v015 cpp # cat out.log
Standard out!
7v015 cpp # ./3-5 > out.log 2>&1
7v015 cpp # cat out.log
Standard error!
Standard out!
```

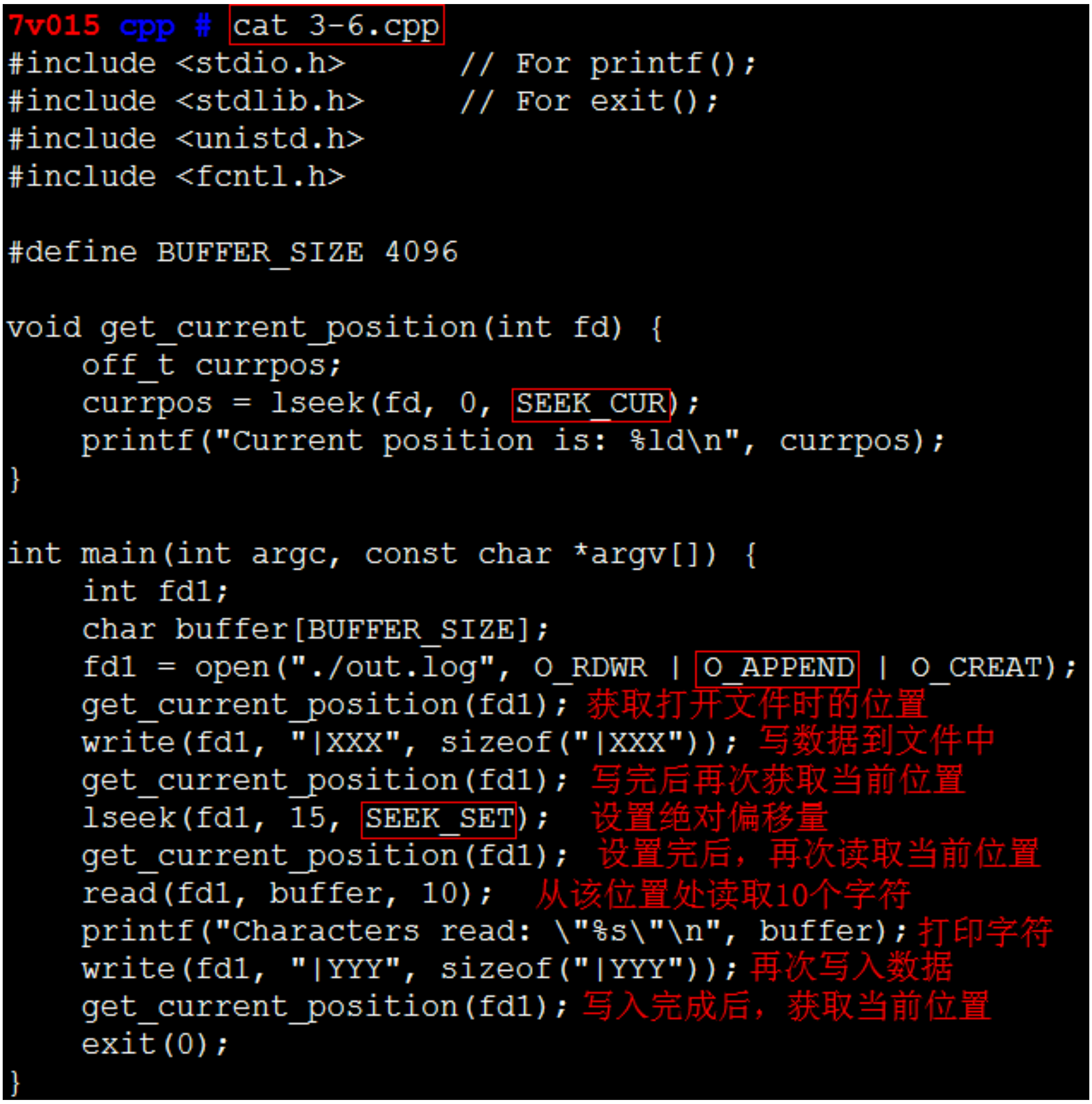
习题中的两条命令

命令./3-5 2>&1 > out.log
应分成如下几段来理解





6、可以从任意位置读取数据，但在写入时，由于设置了追加模式，所以数据只能写在文件尾部。

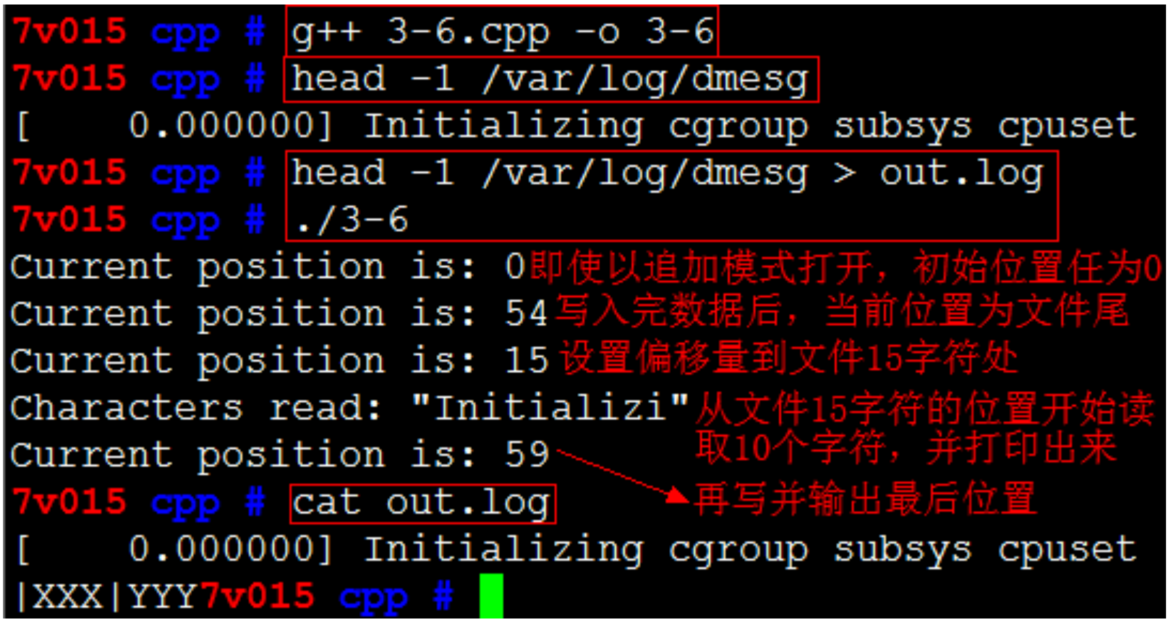


```
7v015 cpp # cat 3-6.cpp
#include <stdio.h>      // For printf();
#include <stdlib.h>     // For exit();
#include <unistd.h>
#include <fcntl.h>

#define BUFFER_SIZE 4096

void get_current_position(int fd) {
    off_t currpos;
    currpos = lseek(fd, 0, SEEK_CUR);
    printf("Current position is: %ld\n", currpos);
}

int main(int argc, const char *argv[]) {
    int fd1;
    char buffer[BUFFER_SIZE];
    fd1 = open("./out.log", O_RDWR | O_APPEND | O_CREAT);
    get_current_position(fd1); 获取打开文件时的位置
    write(fd1, "|XXX", sizeof("|XXX")); 写数据到文件中
    get_current_position(fd1); 写完后再次获取当前位置
    lseek(fd1, 15, SEEK_SET);  设置绝对偏移量
    get_current_position(fd1);  设置完后，再次读取当前位置
    read(fd1, buffer, 10);  从该位置处读取10个字符
    printf("Characters read: \"%s\"\n", buffer); 打印字符
    write(fd1, "|YYY", sizeof("|YYY")); 再次写入数据
    get_current_position(fd1); 写入完成后，获取当前位置
    exit(0);
}
```



```
7v015 cpp # g++ 3-6.cpp -o 3-6
7v015 cpp # head -1 /var/log/dmesg
[    0.000000] Initializing cgroup subsys cpuset
7v015 cpp # head -1 /var/log/dmesg > out.log
7v015 cpp # ./3-6
Current position is: 0
Current position is: 54
Current position is: 15
Characters read: "Initializi"
Current position is: 59
7v015 cpp # cat out.log
[    0.000000] Initializing cgroup subsys cpuset
|XXX|YYY7v015 cpp #
```

即使以追加模式打开，初始位置任为0

写入完数据后，当前位置为文件尾

设置偏移量到文件15字符处

从文件15字符的位置开始读取10个字符，并打印出来

再写并输出最后位置