

3.2 实现 dup2，要求不调用 fcntl 函数，并且要有正确的出错处理

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <string.h>

#define LOG_PATH "./my_dup2.log"

static int my_dup2(int oldfd, int newfd) {

    /* 因为可能会重定向标准输出，所以创建一个文件用来打错误日志 */
    int log_on = 0;
    int log_fd = open(LOG_PATH, O_CREAT | O_WRONLY | O_APPEND, 0644);
    if (log_fd < 0) {
        printf("my_dup2 log create fail");
    } else {
        log_on = 1;
    }

    char buf[1024] = { 0 };
#ifdef LOG
#error "LOG Macro has already be defined!"
#else
#define LOG(_fmt_, ...) \
do { \
    if (log_on) { \
        snprintf(buf, sizeof(buf), _fmt_, ##__VA_ARGS__); \
        if (buf[strlen(buf) - 1] == '\0') { \
            buf[strlen(buf) - 1] = '\n'; \
        } \
        write(log_fd, buf, strlen(buf)); \
        bzero(buf, sizeof(buf)); \
    } \
} while (0)
#endif

    /* 校验 fd 是否合法 */
    long open_max = sysconf(_SC_OPEN_MAX);
    if (open_max < 0 || open_max == LONG_MAX) {
        struct rlimit rl;
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0) {
```

```

        LOG("get file limit error");
        exit(1);
    }
    if (rl.rlim_max == RLIM_INFINITY) {
        open_max = 256;
    } else {
        open_max = rl.rlim_max;
    }
}
if ((oldfd < 0 || oldfd >= open_max) ||
    (newfd < 0 || newfd >= open_max)) {
    LOG("fd %d or fd %d beyond valid range", oldfd, newfd);
    errno = EBADF;
    return -1;
}

/* 如果 oldfd 没有打开, 报错 */
if (fcntl(oldfd, F_GETFD) < 0) {
    errno = EBADF;
    LOG("oldfd %d not open", oldfd);
    return -1;
}

/* 如果相等, 直接返回 newfd */
if (oldfd == newfd) {
    LOG("oldfd == newfd");
    return newfd;
}

/* 如果 newfd 是打开的, 关闭它 */
if (fcntl(newfd, F_GETFD) >= 0) {
    if (close(newfd) < 0) {
        LOG("newfd %d close fail", newfd);
        return -1;
    }
}

/* 复制 oldfd 直到返回值等于 newfd, 沿途记录每个打开的 fd */
int *fds = malloc(open_max);
int n = 0;
while (1) {
    if ((fds[n] = dup(oldfd)) < 0) {
        LOG("oldfd %d dup fail", oldfd);
        return -1;
    }
    if (fds[n] == newfd) {
        break;
    }
    ++n;
}

```

```

    }
    /* 关闭沿途打开的所有 fd */
    for (int j = 0; j < n; ++j) {
        if (close(fds[j]) < 0) {
            LOG("extra dupped fd %d close fail", fds[j]);
        }
    }
    free(fds);
#undef LOG
    return newfd;
}

int main(int argc, char *argv[]) {

    int fd = open(argv[1], O_CREAT | O_RDWR, 0644);
    if (fd < 0) {
        perror("fd");
        exit(1);
    }

    my_dup2(fd, STDOUT_FILENO);

    char buf[1024] = { 0 };
    while (fgets(buf, sizeof(buf), stdin) != NULL) {
        fputs(buf, stdout);
    }

    return 0;
}

```

- 如果 old 或者 new 中任一超出了最大可分配描述符的范围，返回 -1
- 如果 old 没有打开，返回 -1，（但 new 可以是关闭的）
- 如果 old 与 new 相同，则不关闭 new，直接返回 new
- 如果 new 是打开的，那么关闭 new
- 使用 dup 复制并记录 0 ~ OPEN_MAX - 1 范围内的所有 fd，直到 dup 的返回值等于 new（因为 new 已经被关闭，所以所有 dup 的返回值都是小于等于 new 并逐渐靠近 new 的）
- 关闭沿途打开的无意义的 fd

3.3 假设一个进程执行下面 3 个函数调用

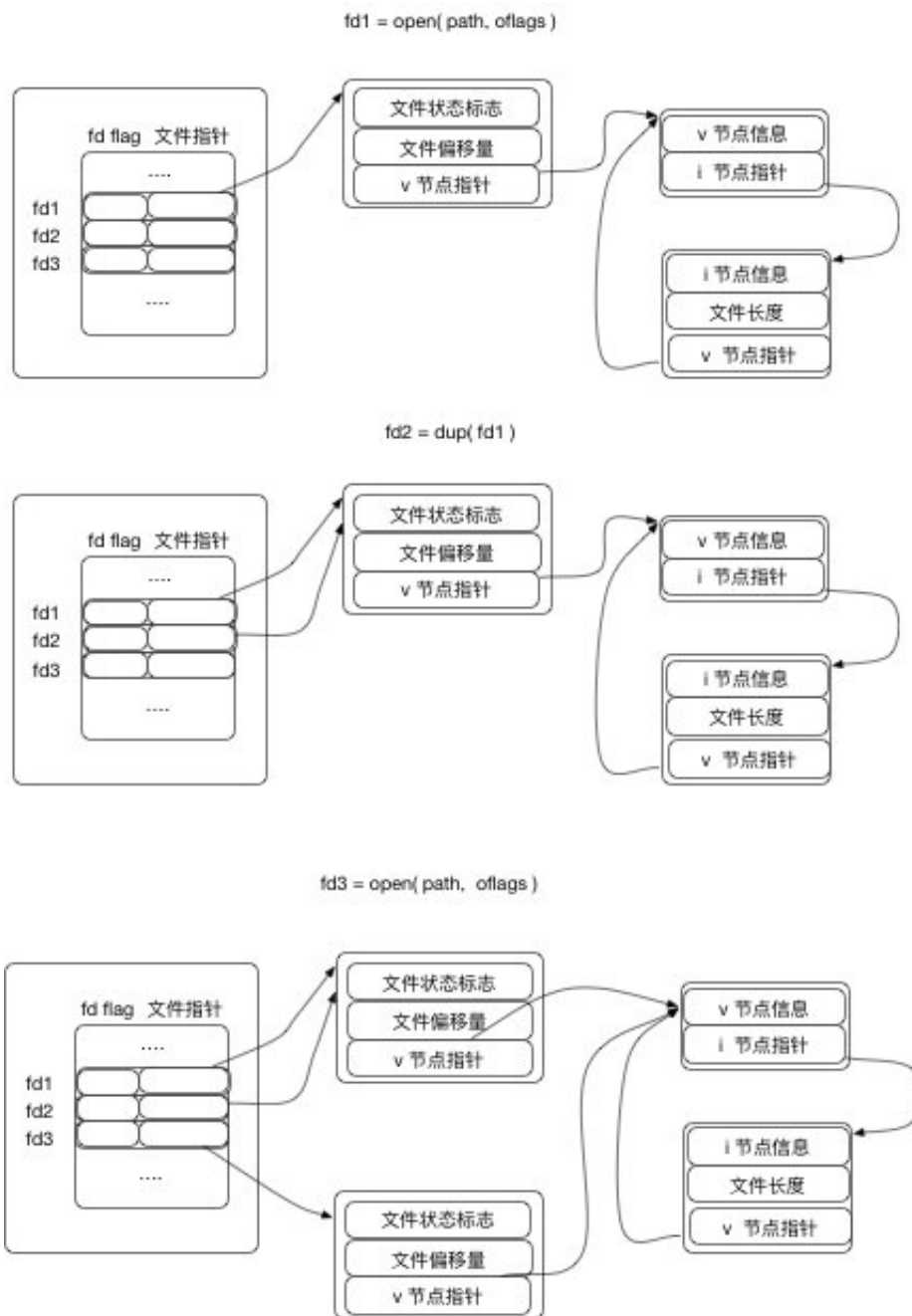
```

fd1 = open(path, oflags);
fd2 = dup(fd1);
fd3 = open(path, oflags);

```

画图说明，对 `fcntl` 作用于 `fd1` 来说，`F_SETFD` 命令会影响哪一个文件描述符？`F_SETFL` 呢？

见下图，`F_SETFD` 作用于 `fd1` 只影响 `fd1`，`F_SETFL` 作用域 `fd1` 会影响 `fd1` 和 `fd2`

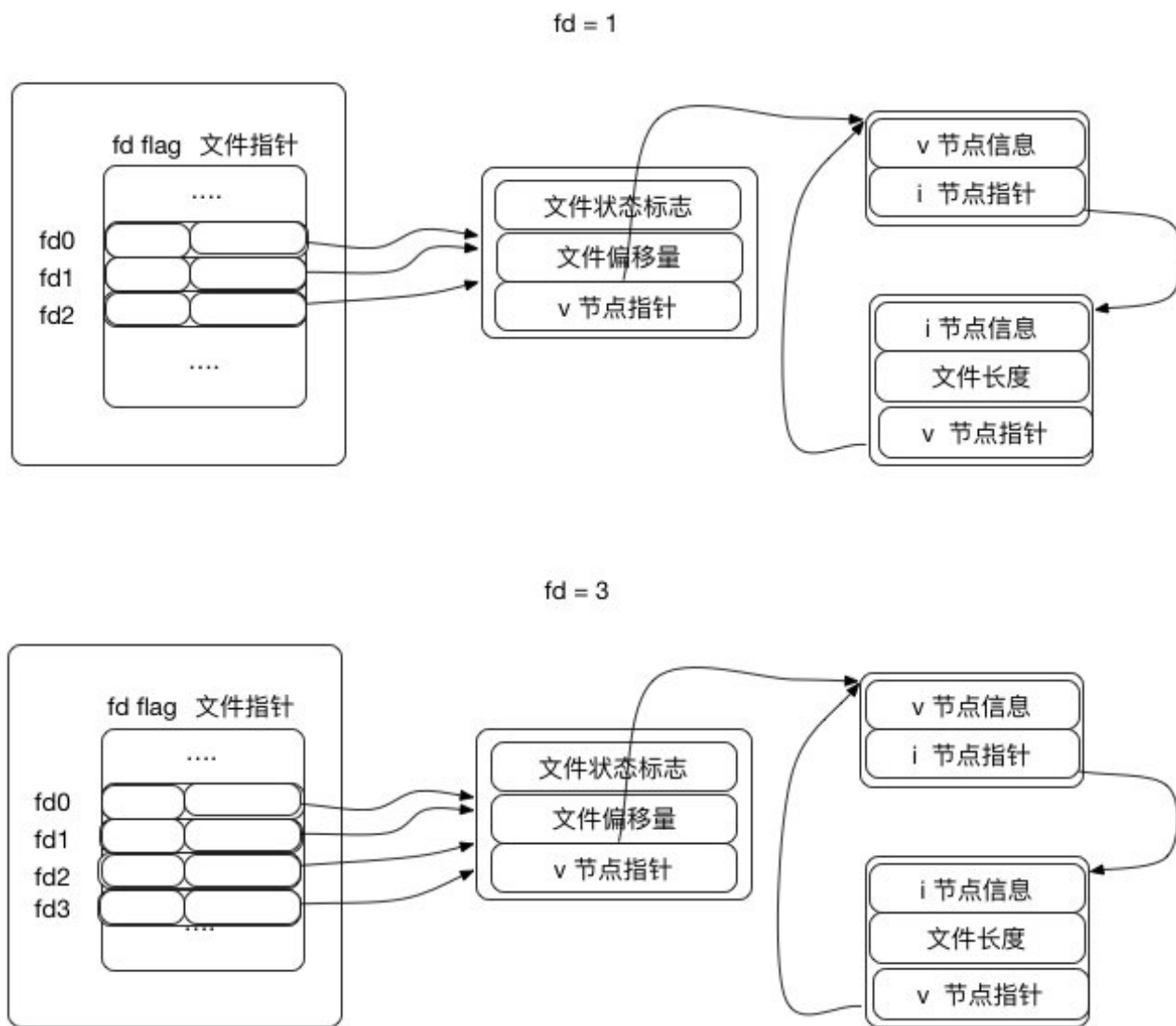


3.4 许多程序中都包含下面一段代码：

```
dup2( fd, 0 );
dup2( fd, 1 );
dup2( fd, 2 );
if ( fd > 2 )
    close( fd );
```

为了说明 if 语句的必要性，假设 fd 是 1，画出每次调用 dup2 时 3 个文件描述符及相应的文件表项的变化情况。然后再画出 fd 为 3 的情况。

见下图，如果 fd 为 1，那么 dup2(fd, 1) 不关闭 1，仅仅将 1 作为返回值，因此最后仅有三个文件描述符指向同一个文件，若 fd 为 3，则最后将有 4 个文件描述符指向同一个文件，然后我们只希望将标准输出，标准输入，标准错误重定向到一个指定文件，不希望能从额外的途径读写该文件，因此关闭多余的文件描述符。



3.6 如果使用追加标志打开一个文件以便读、写，能否仍用 lseek 在任一位置开始读？能否用 lseek 更新文件中任一部分的数据？请编写一段程序验证。

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
```

```

int main(int argc, char *argv[]) {

    if (argc != 2) {
        printf("./a.out [file_name]\n");
        exit(1);
    }

    int fd = -1;
    /* 如果把 O_APPEND 去掉, 那么最后文件中只有一个 9 */
    if ((fd = open(argv[1], O_CREAT | O_APPEND | O_RDWR, 0644)) < 0) {
        perror("open ");
        exit(1);
    }

    const char *p = "0123456789";
    char *q = (char *)p;

    char buf[64];

    for (int i = 0; i < 10; i++) {
        if (write(fd, q + i, 1) != 1) {
            perror("write");
            exit(1);
        }

        /* 如果 lseek 起作用, 那么最后文件中只有一个 9 */
        lseek(fd, -1, SEEK_CUR);

        bzero(buf, sizeof(buf));
        if (read(fd, buf, sizeof(buf)) < 0) {
            perror("read");
            exit(1);
        }

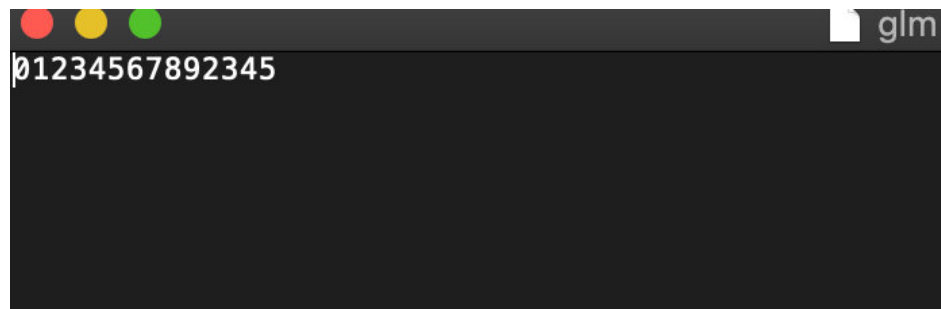
        printf("read content : %s\n", buf);
    }
    lseek(fd, 0, SEEK_SET);
    if (write(fd, q+2, 4) != 4)
    {

        perror("write");
        exit(1);
    }
    printf("over\n");
    return 0;
}

```

运行结果如下:

```
(base) apple@appledeMacBook-Pro C:C++ % ./Practise_3_06 glm
read content : 0
read content : 1
read content : 2
read content : 3
read content : 4
read content : 5
read content : 6
read content : 7
read content : 8
read content : 9
over
(base) apple@appledeMacBook-Pro C:C++ %
```



```
01234567892345
```

用追加标志打开一个文件用于读写，仍然可以用 `lseek` 在任一位置开始读，但不能用 `lseek` 更新文件中的任一部分数据，因为设置 `O_APPEND` 标志意味着每次写操作都会将文件偏移量设置为当前文件的长度，【设置文件偏移量】和【写文件】是一个原子操作，因此，在写操作前的 `lseek` 将不会产生作用。