

Замечание по open():
должен быть анализ флагов
O_CREATE, O_EXCL, O_TMPFILE,
O_APPEND, O_PATH;

Понимать их назначение и как по ним
происходит ветвление;

O_PATH:

Когда этот флаг установлен, то будет
возвращен дескриптор struct file (он
уже существует, т.е. мы его формально
не создаём), при этом сам файл не открыва-
ется (разобраться, почитать мануал)

↳ т.е. он уже открыт?

Если флаг не установлен, то будет ор-
ганизован путь по всем этапам пути и
вызвана ф-ция do_open(), которая открыва-
ет файл, т.е. создаёт дескриптор
(инициализирует поля struct file);

В alloc_fd отразить расширение
таблицы files_struct;

В do_filp_open вложенного if кем;

Флаги LOOKUP_RCU и LOOKUP_REVAL
надо описать;

Избегать в схемах алгоритмов слова
"успешно" (писать true);

Пример 0:

Рассмотрим приведенный рисунок 1. На рисунке приведена виртуальная файловая система Unix с интерфейсом vfs/vnode. В ОС Linux отсутствует vnode, имеется только inode.

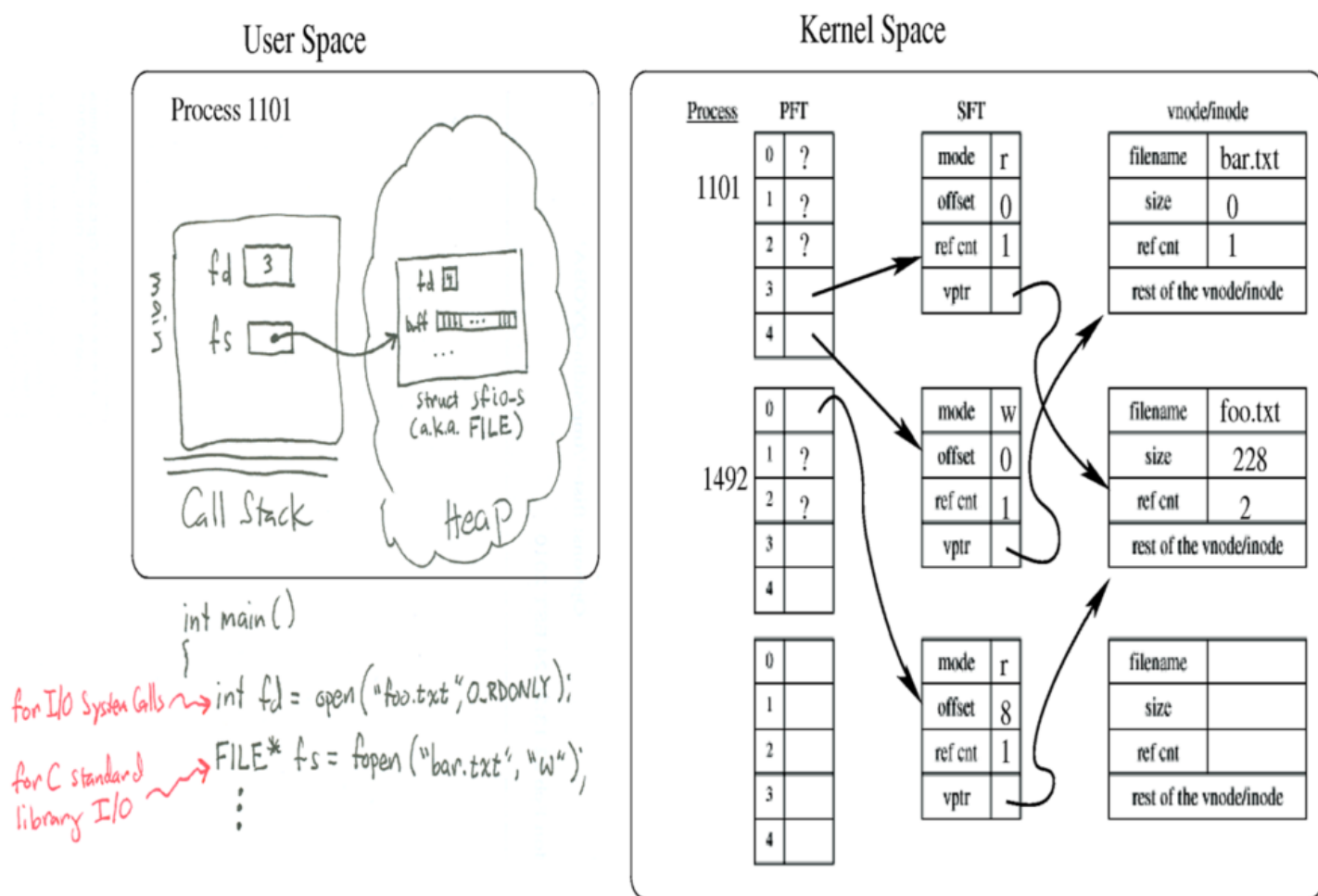


Рис.1 Процесс 1101. PFT – Process File Table – таблица файлов, открытых процессом, SFT – System File Table – системная таблица открытых файлов

Открываются 2 файла: один с `fopen()` (библиотечная ф-ция), другой с `open()`;

❗ Библиотека `stdio.h` – библиотека буферизованного ввода/вывода;

Все ^(в `stdio.h`) буферизуется, т.е. сначала (в любом направлении: и при чтении, и при записи) данные пишутся в буфер, и только после этого пишутся в файл (см. картинку);

Буфер в user space;

Пример 1 (демонстрация проблемы буферизации):

```
//testC10.c
#include <stdio.h>
#include <fcntl.h>

/*
On my machine, a buffer size of 20 bytes
translated into a 12-character buffer.
Apparently 8 bytes were used up by the
stdio library for bookkeeping.
*/

int main()
{
    // have kernel open connection to file alphabet.txt
    int fd = open("alphabet.txt", O_RDONLY);

    // create two a C I/O buffered streams using the above connection
    FILE *fs1 = fdopen(fd, "r");
    char buff1[20];
    setvbuf(fs1, buff1, _IOFBF, 20);

    FILE *fs2 = fdopen(fd, "r");
    char buff2[20];
    setvbuf(fs2, buff2, _IOFBF, 20);

    // read a char & write it alternately from fs1 and fs2
    int flag1 = 1, flag2 = 2;
    while(flag1 == 1 || flag2 == 1)
    {
        char c;
        flag1 = fscanf(fs1, "%c", &c);
        if (flag1 == 1) {
            fprintf(stdout, "%c", c);
        }
        flag2 = fscanf(fs2, "%c", &c);
        if (flag2 == 1) {
            fprintf(stdout, "%c", c);
        }
    }
    return 0;
}
```

В этой программе файл открывается один раз сист. вызовом `open()`, который воз -

Возвращает дескриптор типа `int` в `user mode`;
Это число, которое явл. индексом открытого
файла в массиве открытых файлов про-
цесса;

т.е. в `struct files_struct` определён
массив дескрипторов открытых файлов
процесса, т.е. каждый процесс имеет та-
кой массив;

При этом файлы с индексами 0, 1 и 2
для любого процесса открываются автома-
тически (но это не значит, что не работает ядро);
При этом ядро выполняет все необхо-
димые действия (в коде есть проверки "`> 0`");

Для процесса, который не открывал больше
никаких файлов, возвращённый индекс будет 3;

Затем в коде 2 раза вызывается ф-ция
`fopen()`:

```
FILE *fs1 = fopen(fd, "r");
```

Это библиотечная ф-ция (`stdio.h`), кото-
рая возвращает указатель на `FILE`
(большими буквами!);

`FILE` - структура библиотеки `stdio.h`;

На самом деле её не существует, она опре-
деляется как `define` на базе
`struct IO_FILE` (привести в отчёте);

В этой структуре определены флаги,
и дальше идут указатели на буферы:


```

struct IO_FILE
{
    ...
    char* _IO_buf_base;
    char* _IO_buf_end;
    ...
    int _fileno; // поле, в которое заносится
    // полученный файловый дескриптор
    ...
}

```

Буфер — зарезервированная область;

Библиотечная ф-ция `fopen()` в своей коде в любом случае вызовет сист. вызов `open()`, который вернёт дескриптор файла;

В отличие от `fopen()`, `fopen()` получает этот дескриптор;

`fopen()` вызывается 2 раза, т.е. создаётся 2 указателя: `fs1` и `fs2`;

Также 2 раза вызывается ф-ция `setvbuf()`, ей передаётся `fs1/fs2`, массив типа `char`, при этом устанавливается размер буфера 20 байт;

Это сделано для того, чтобы продемонстрировать проблемы буферизованного ввода/вывода;

Ф-ции библиотеки `stdio.h` всегда работают с буферами. Но их размер, конечно, не 20 символов. 1 страница — мин. размер такого буфера;

Эти буферы устанавливаются при использовании ф-ции этой библиотеки;
Но в примере принудительно устанавливается размер буфера, при этом
`setvbuf`

↳ `virtual`, т.е. речь идет об адр. пр-ве процесса

Вывод программы:
`abv...`

Хотя ожидается `abcde...`

Дело в том, что первый вызов `fscanf()` заполнит первый буфер `buf1` полностью, т.е. прочтает из файла первые 20 символов латинского алфавита;

Второй вызов `fscanf()` прочтает во второй буфер `buf2` оставшиеся 6 символов;

В итоге на экране мы увидим `abv...`; Это является особенностью буферизации;

Чтение - более важное действие, чем запись, т.к. мы думаем, что читаем данные в одном порядке, а читаем в другом. А значит и обрабатываем в неправильном порядке, если не понимаем, что происходит;

Пример 2 :

```
//testKernellO.c
#include <fcntl.h>

int main()
{
    char c;
    // have kernel open two connection to file alphabet.txt
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("alphabet.txt", O_RDONLY);
    // read a char & write it alternatingly from connections fs1 & fd2
    while(1)
    {
        if (read(fd1, &c, 1) != 1) break;
        write(1, &c, 1);
        if (read(fd2, &c, 1) != 1) break;
        write(1, &c, 1);
    }
    return 0
}
/* переписать код без использования break! */
Файл alphabet.txt содержит символы: Abcdefghijklmnopqrstuvwxyz
```

open() вызывается 2 раза \Rightarrow будет создано 2 дескриптора открытого файла в системной таблице открытых файлов;

В таблице открытых файлов процесса будут дескрипторы 3 и 4 (int);

А в системной таблице будут 2 дескриптора struct file;

```

struct file {
    union {
        struct llist_node fu_llist;
        struct rcu_head fu_rcuhead;
    } f_u;
    struct path f_path;
    struct inode *f_inode; /* cached value */
    const struct file_operations *f_op;

    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t f_lock; // действия на файлах нуждаются в защите
    enum rw_hint f_write_hint;
    atomic_long_t f_count;
    unsigned int f_flags;
    fmode_t f_mode;
    struct mutex f_pos_lock;
    loff_t f_pos;
    struct fown_struct f_owner;
    const struct cred *f_cred;
    struct file_ra_state f_ra;

    u64 f_version;
#ifdef CONFIG_SECURITY
    void *f_security;
#endif
    /* needed for tty driver, and maybe others */
    void *private_data;

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head f_ep_links;
    struct list_head f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space *f_mapping;
    errseq_t f_wb_err;
} __randomize_layout

```

Речь идёт о логическом файле:

f_pos — указатель файла;

Когда мы открываем файл, этот указатель стоит в начале файла, т.е. смещение = 0;

Каждая операция чтения/записи сдвигает этот указатель на соотв. кол-во байт в зав-ти от считанного/записанного типа;

И.е. каждый дескриптор открытого файла имеет поле f_pos , которое содержит информацию, связанную с дескриптором этого открытого файла;

...
break надо убрать (структурировать);

Рез-т работы программы:

aabbcc...

(дублирование символов)

Хотели ускорить чтение файла, а получили удвоение;

Эта программа демонстрирует проблему того, что один и тот же файл м. б. открыт много раз;

В этом элементарном примере один процесс 2 раза открывает один и тот же файл и сразу выводит результат

Если речь идёт не о разных процессах, а о разных потоках одного процесса, то картина будет немного другая (в зав-ти от посл-ти действий), но проблема остаётся (проанализировать);

! Каждое открытие файла сопровождается созданием дескриптора открытого файла, содержащего поле `f_pos`;

Пример 3:

```
#include <stdio.h>
#include <fcntl.h>
int main()
{
    int fd1 = fopen("q.txt", O_RDWR);
    int fd2 = fopen("q.txt", O_RDWR);
    int curr = 0;
    for(char c = 'a'; c <= 'z'; c++)
    {
        if (c%2){
            fprintf(fd1, "%c", c);
        }
        else{
            fprintf(fd2, "%c", c);
        }
    }
    fclose(fd1);
    fclose(fd2);
    return 0;
}
```

`fopen` вызывается 2 раза \Rightarrow файл будет открыт 2 раза, но это будет уже буферизованный ввод/вывод (т.е. символы сначала пишутся в буфер);

Существует 3 причины, по которым данные из буфера записываются в файл:

- 1) буфер заполнен;
- 2) вызвана ф-ция `fflush` (принудительная запись);
- 3) вызвана ф-ция `close`/`fclose`;

Еще раз:

Файл открывается 2 раза. Первый раз мы это проделываем с исп-ем ф-ции библиотеки `stdio.h`, второй — небудеризованный ввод/вывод (`open()`, `write()`);

Рез-т один и тот же, но действия в системе будут разные: в одном случае будеризованный ввод/вывод, в другом — небудеризованный;
(описать в ЛР)

Также в этих двух программах необходимо использовать `struct stat` (привести её в отчёте);

Эта структура доступна в `user mode`; В ней находится информация об открытых файлах (для каждого открытого файла из этой структуры мы можем получить информацию о нём)

→ полностью, но нас будут интересовать `inode number` и `total size in bytes`:
(в отчёте привести структуру полностью)

```
struct stat
{
    ...
    ino_t    st_ino;
    ...
    off_t    st_size;
    ...
}
```

С этой структурой связаны след. ф-ции:
stat(), lstat(), fstat();

Вывести информацию об inode и размере файла сразу после открытия (2 раза открыть, 2 раза вывести) и после закрытия файла (2 раза закрыть, 2 раза вывести);

В файл необходимо писать;

Для каждой программы ^(всего 3) нужно писать многопоточный вариант;

Есть большая разница между тем, создаём мы один доп. поток (каждый из 2 потоков выполняет свою часть действий) или 2 (тогда главный поток только создаёт потоки и завершает их работу, а все действия выполняются в доп. потоках);

А если создавать не потоки, а процессы? Подумать, может быть там появятся другие особенности;

Программы отн. небольшие, можно проработать все доп. варианты;

Всего 3 программы, каждая в 3 вариантах;
За каждый вариант по 1 картинке,
в 3 варианте тоже 1 картинка, т.к. картин-
ка для недур. Ввода/вывода будет полностью
соответствовать картинке 2-го варианта;
(потому 3 картинки будет достаточно)

Доп. (по сравнению с прошлыми годами):
связать все таблицы вплоть до
struct superblock:

struct files_struct, struct file, struct dentry,
struct inode, struct superblock, struct task_struct;

Потоки рисовать не будем, т.к. ресурсами
владеет процесс, а открытые файлы —
ресурсы, открытые данным процессом
(открытые файлы принадлежат процессу,
а не потоку)

Итого: 8 программ;