

Лабораторная работа

Буферизованный и не буферизованный ввод-вывод

Задача: анализ особенностей работы функций ввода-вывода в UNIX/Linux

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

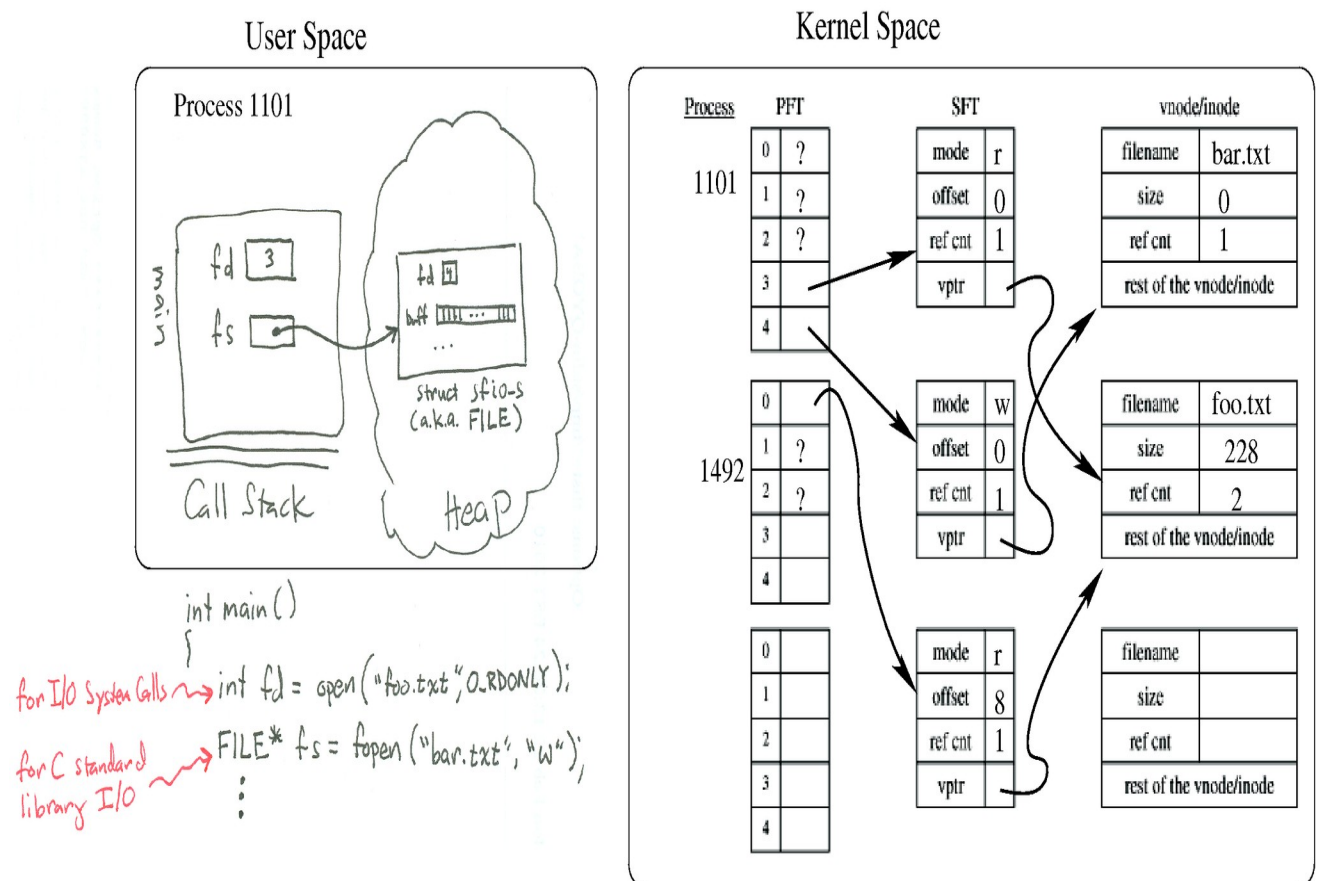


Рис.1 Процесс 1101. PFT – Process File Table – таблица файлов, открытых процессом, SFT – System File Table – системная таблица открытых файлов

- Когда выполняется ввод-вывод через стандартную библиотеку C (stdio.h) (т.е. используя fscanf(), fprintf(), fopen(), fclose()), вызовы для открытия файла, его закрытия, чтения и записи (open(), close(), read(), write()) не завершаются немедленно, так как это просто библиотечные функции, которые делают системные вызовы. Другими словами, стандартная библиотека ввода-вывода C является просто верхним уровнем для выполнения системных вызовов.

Рассмотрим ситуацию, показанную на рис.1. Здесь процесс выполняет системный вызов `open("foo.txt", O_RDONLY)` для чтения файла `foo.txt` и вызов функции `fopen("bar.txt", "w")` из стандартной библиотеки для записи в файл `bar.txt`. Необходимо помнить, что стандартный ввод-вывод буферизуется и это видно на рисунке. Файловый дескриптор процесса `fs` ссылается на буфер.

Обратите внимание на то, что на рисунке показан другой процесс. Что Вы можете сказать об этой ситуации, если другой процесс создан, например, shell?

```
bash$ ./prog1 < foo.txt > bar.txt 2> bar.txt
```

Потоковый ввод/вывод берет данные в виде потока отдельных символов. Когда поток открыт для ввода/вывода, открытый файл связывается со структурой типа FILE (см. struct _IO_FILE), определенной в файле стандартных описаний "**stdio.h**". Указатель на структуру FILE возвращается при открытии файла. Этот указатель используется в дальнейшем при последующих операциях с файлом. Ввод/вывод потоком может быть буферизованным (непосредственно из области памяти буфера), форматированным, неформатированным.

Функции **fclose()**, **fopen()**, **fprintf()**, **fscanf()**, **fgetc()**, **fputc()**, **fgets()**, **fputs()**, **fcloseall()**, **getc()**, **gets()**, **putc()**, **puts()**, **getchar()** работают с форматированными данными.

Функции **fread()**, **fwrite()** работают с неформатированными данными.

Функции **scanf()**, **printf()**, **getchar()**, **putchar()** работают со стандартными потоками **stdin**, **stdout**.

Поток должен быть открыт, прежде чем для него будет выполнена операция ввода/вывода. Исключения составляют следующие потоки:

stdin - стандартный ввод;

stdout - стандартный вывод;

stderr - стандартные ошибки;

stdaux - стандартный порт;

stdprn - стандартная печать.

Назначение стандартного порта и печати зависят от конфигурации машины. Обычно эти потоки указывают на вспомогательный порт и принтер.

Открытые файлы, для которых используется потоковый ввод/вывод, буферизуются.

При этом стандартные потоки не буферизуются. Буфера, размещенные в системе, не доступны пользователю.

Структуры данных, связанные с процессом

С каждым процессом в системе связаны список открытых им файлов, корневая файловая система, текущий рабочий каталог, точки монтирования и т.д.

```
struct task_struct{
    .....
    int                prio;
    int                static_prio;
    ...
    struct list_head    tasks;
    ...
    struct mm_struct    *mm;
    struct mm_struct    *active_mm;
    ...
    pid_t               pid;
    pid_t               tgid;
    ...
    struct list_head    children;
    struct list_head    sibling;
    ...
    /* Filesystem information */
    struct fs_struct     *fs;
    /* Open file information */
    struct files_struct  *files;
    /* Namespaces */
```

```

    struct nsproxy          *nsproxy;
    ...
}

```

Структура `struct fs_struct` содержит информацию о файловой системе, к которой принадлежит процесс. Структура определена в файле `<linux/fs_struct.h>`.

```

struct fs_struct {
    atomic_t    count;      /* счетчик ссылок на структуру */
    rwlock_t    lock;      /* блокировка для защиты структуры */
    int         umask;      /* права доступа к файлу, используемые
                           по умолчанию */
    struct dentry *root;    /* объект dentry корневого каталога */
    struct dentry *pwd;     /* объект dentry
                           текущего рабочего каталога */
    struct dentry *allroot; /* объект dentry альтернативного корня */
    struct vfsmount *rootmnt; /* объект монтирования корневого каталога */
    struct vfsmount *pwmnt; /* объект монтирования
                           текущего рабочего каталога */
    struct vfsmount *altrootmnt; /* объект монтирования
                           альтернативного корня */
};

```

Структура `struct files_struct` определяет дескрипторы файлов, открытых процессом. Адрес этой структуры хранится в поле `files` дескриптора процесса. В данной структуре хранится вся информация процесса об открытых файлах и файловых дескрипторах. Эта структура, с комментариями, имеет следующий вид:

```

struct files_struct {
    atomic_t    count;      /* счетчик ссылок на данную структуру */
    spinlock_t  file_lock; /* блокировка для защиты данной структуры */
    int         max_fds;    /* максимальное количество файловых объектов */
    int         max_fdset;  /* максимальное количество файловых дескрипторов */
    int         next_fd;    /* номер следующего файлового дескриптора */
    struct file **fd;       /* массив всех файловых объектов */
    fd_set      *close_fds; /* файловые дескрипторы, которые должны закрываться при вызове exec() */
    fd_set      *open_fds;  /* указатель на дескрипторы открытых файлов */
    fd_set      close_on_exec_init; /* первоначальные файлы для закрытия при вызове exec() */
    fd_set      open_fds_init; /* первоначальный набор файловых дескрипторов */
    struct file *fd_array[NR_OPEN_DEFAULT]; /* массив файловых объектов */
};

```

Структура `struct file` определяет дескриптор открытого файла в системе.

```

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;
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

```

Структура struct inode описывает созданный файл.

```

struct inode {
        umode_t          i_mode;
        unsigned short   i_opflags;
        kuid_t           i_uid;
        kgid_t           i_gid;
        unsigned int      i_flags;

#ifdef CONFIG_FS_POSIX_ACL
        struct posix_acl *i_acl;
        struct posix_acl *i_default_acl;

```

```
#endif
```

```
const struct inode_operations *i_op;  
struct super_block *i_sb;  
struct address_space *i_mapping;
```

```
#ifdef CONFIG_SECURITY
```

```
void *i_security;
```

```
#endif
```

```
/* Stat data, not accessed from path walking */
```

```
unsigned long i_ino;
```

```
/*
```

```
* Filesystems may only read i_nlink directly. They shall use the
```

```
* following functions for modification:
```

```
*
```

```
* (set|clear|inc|drop)_nlink
```

```
* inode_(inc|dec)_link_count
```

```
*/
```

```
union {
```

```
    const unsigned int i_nlink;
```

```
    unsigned int __i_nlink;
```

```
};
```

```
dev_t i_rdev;
```

```
loff_t i_size;
```

```
struct timespec64 i_atime;
```

```
struct timespec64 i_mtime;
```

```
struct timespec64 i_ctime;
```

```
spinlock_t i_lock; /* i_blocks, i_bytes, maybe i_size */
```

```
unsigned short    i_bytes;
```

```
u8                i_blkbits;
```

```
u8                i_write_hint;
```

```
blkcnt_t          i_blocks;
```

```
#ifdef __NEED_I_SIZE_ORDERED
```

```
seqcount_t        i_size_seqcount;
```

```
#endif
```

```
/* Misc */
```

```
unsigned long      i_state;
```

```
struct rw_semaphore i_rwsem;
```

```

        unsigned long        dirtied_when; /* jiffies of first dirtying */
        unsigned long        dirtied_time_when;

        struct hlist_node    i_hash;
        struct list_head     i_io_list;    /* backing dev IO list */
#ifdef CONFIG_CGROUP_WRITEBACK
        struct bdi_writeback *i_wb;        /* the associated cgroup wb */

        /* foreign inode detection, see wbc_detach_inode() */
        int                  i_wb_frn_winner;
        u16                  i_wb_frn_avg_time;
        u16                  i_wb_frn_history;
#endif
        struct list_head     i_lru;        /* inode LRU list */
        struct list_head     i_sb_list;
        struct list_head     i_wb_list;    /* backing dev writeback list */
        union {
                struct hlist_head    i_dentry;
                struct rcu_head       i_rcu;
        };
        atomic64_t           i_version;
        atomic64_t           i_sequence; /* see futex */
        atomic_t             i_count;
        atomic_t             i_dio_count;
        atomic_t             i_writecount;
#ifdef CONFIG_IMA || defined(CONFIG_FILE_LOCKING)
        atomic_t             i_readcount; /* struct files open RO */
#endif
        union {
                const struct file_operations *i_fop; /* former ->i_op-
>default_file_ops */
                void (*free_inode)(struct inode *);
        };
        struct file_lock_context *i_flctx;
        struct address_space i_data;
        struct list_head     i_devices;
        union {
                struct pipe_inode_info *i_pipe;
                struct block_device *i_bdev;
                struct cdev *i_cdev;
                char *i_link;
                unsigned       i_dir_seq;
        };

        __u32               i_generation;

```

```

#ifdef CONFIG_FSNOTIFY
    __u32          i_fsnotify_mask; /*all events this inode cares about*/
    struct fsnotify_mark_connector __rcu    *i_fsnotify_marks;
#endif

#ifdef CONFIG_FS_ENCRYPTION
    struct fscrypt_info    *i_crypt_info;
#endif

#ifdef CONFIG_FS_VERITY
    struct fsverity_info    *i_verity_info;
#endif

    void                *i_private; /* fs or device private pointer */
} __randomize_layout;

```

Эти структуры связаны между собой. Связь структур показана на рис.2

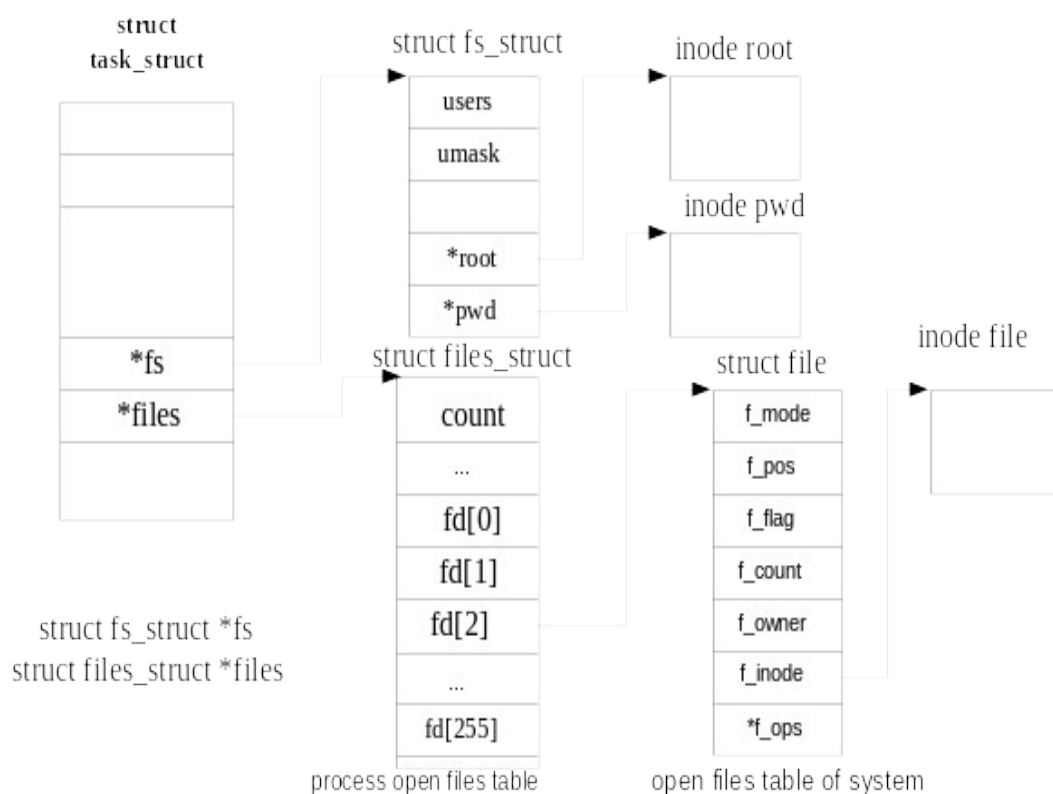


Рис. 2 Таблица открытых файлов процесса – process open files table;
Системная таблица открытых файлов – open files table of system

На рис.2а показана более подробная схема связи структур интерфейса VFS.

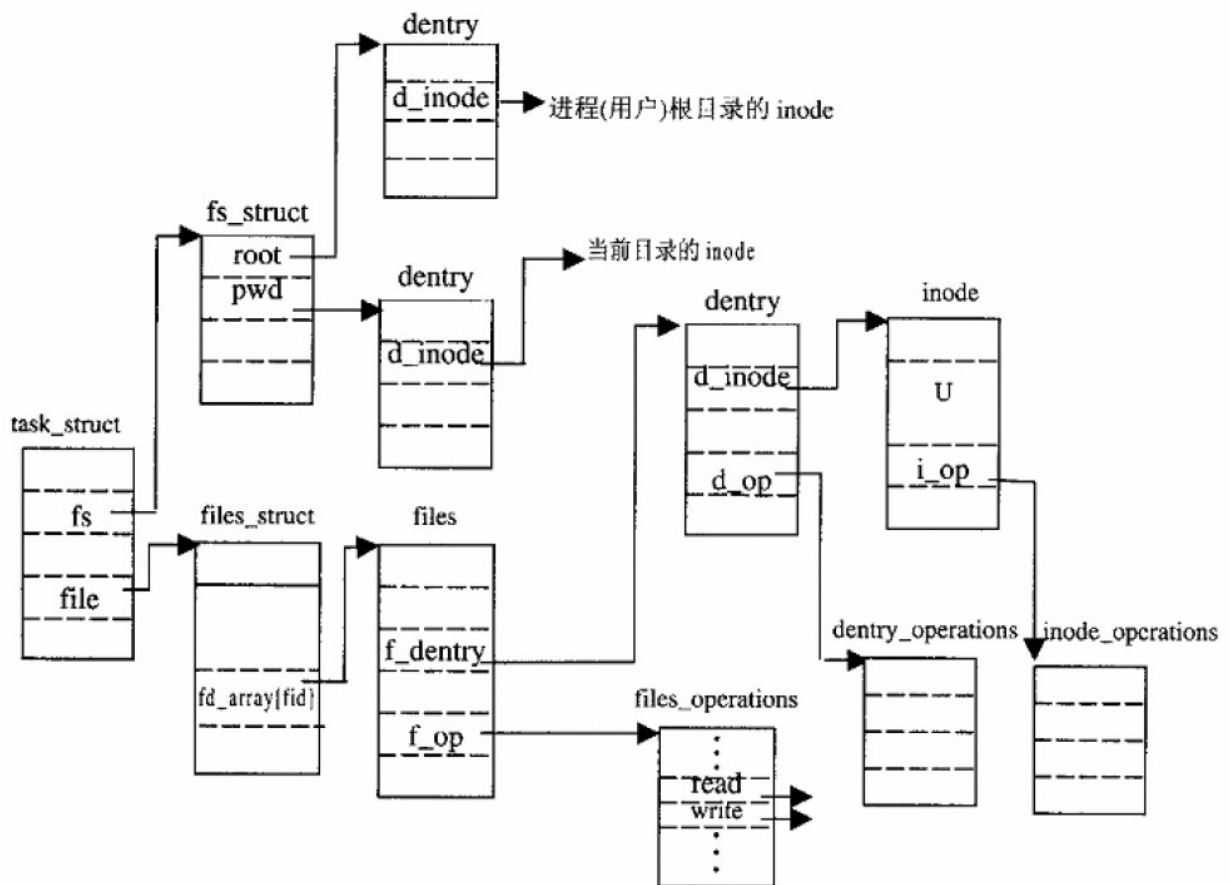


图 5.2 Linux 文件系统逻辑结构图

https://blog.csdn.net/weixin_41943030

Обратите внимание на эту схему!

Задание

В лабораторной работе анализируются результаты выполнения трех программ. Программы демонстрируют открытие одного и того же файла несколько раз. Реализация, когда файл открывается в одной программе несколько раз выбрана для простоты. Однако, как правило, такая ситуация возможна в системе, когда один и тот же файл несколько раз открывают разные процессы или потоки одного процесса. При выполнении асинхронных процессов такая ситуация является вероятной и ее надо учитывать, чтобы избежать потери данных, получения при выводе данных в файл или чтения данных не в той последовательности, в какой предполагалось, и в результате при обработке этих данных получения неверного результата. Каждую из приведенных программ надо выполнить в двух вариантах: как предложено и в многопоточном варианте. В программах может быть создан как дополнительный поток, так и два дополнительных потока, открывающих файл и выполняющих чтение/запись в файл.

Проанализировать работу приведенных программ и прокомментировать результаты их выполнения.

Первая программа:

```
//testCIO.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;
}
```

Вторая программа:

Первый вариант:

```
//testKernelIO.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 alternately 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

Второй вариант программы с вызовом функции `open()`.

```
#include <fcntl.h>
int main()
{
    int fd1 = open("q.txt", O_RDWR);
    int fd2 = open("q.txt", O_RDWR);
    int curr = 0;
    for(char c = 'a'; c <= 'z'; c++)
    {
        if (c%2){
            write(fd1, &c, 1);
        }
        else{
            write(fd2, &c, 1);
        }
    }
    close(fd1);
    close(fd2);
    return 0;
}
```

Третья программа:

Написать программу, которая открывает один и тот же файл два раза с использованием библиотечной функции `foren()`. Для этого объявляются два файловых дескриптора типа `FILE`. В цикле записать в файл буквы латинского алфавита поочередно передавая функции `fprintf()` то первый дескриптор, то – второй. Результат прокомментировать.

Замечание 1:

Можно выделить 4 причины для использования `foren()` а не `open()`:

1. `foren()` выполняет ввод-вывод с буферизацией, что может оказаться значительно быстрее, чем с использованием `open()`;
2. `foren()` делает перевод конца строки, если только файл не открыт в двоичном режиме, который может быть очень полезен, если ваша программа иногда переносится в среду, отличную от Unix;
3. `FILE *` дает возможность использовать `fscanf()` и другие функции `stdio.h`;
4. Для разработки переносимого кода, например для платформы, которая использует только ANSI C и не поддерживает функцию `open()`.

Однако, при более детальном анализе можно сказать, что конец строки чаще мешает, чем помогает, а детальный разбор `fscanf()` часто заставляет заменять функцию, на более полезные. Большинство платформ поддерживают `open()`.

И, наконец, буферизация. В случае, если операции чтения и записи в файл выполняются последовательно, буферизация представляется действительно полезной и обеспечивающей высокую скорость выполнения. Но это может привести к некоторым проблемам. Например, предполагается, что данные записаны в файл, но реально данные там еще отсутствуют. Необходимо помнить о своевременном выполнении `fclose()` и `fflush()`.

Если выполняется `fseek()` буферизация перестает быть полезной.

Все это справедливо, если постоянно работать с сокетами и при этом выполнять неблокирующий ввод-вывод. Но нельзя отрицать полезность `FILE*` для выполнения разбора сложных текстов.

Замечание 2:

```
#include <stdio.h>
```

FILE *fopen(const char *filename, const char *mode);

Открытие файла в режиме добавления (в качестве первого символа в аргумент режима – “a”) приводит к тому, что все последующие операции записи в файл будут работать с current end-of-file, даже если вмешиваются вызовы FSEEK(3C). Если два независимых процесса откроют один и тот же файл для добавления данных (O_APPEND), каждый процесс может свободно писать в файл без опасения нарушить вывод в файл, выполненный другим процессом. Информация будет записана в файл в том порядке, в котором процессы записывали ее в файл.

Вызов open () создает новый файловый дескриптор для открытого файла, запись в общесистемной таблице открытых файлов. Эта запись регистрирует **смещение в файле** и флаги состояния файла (модифицируемые с помощью the fcntl (2) операции F_SETFL). Дескриптор файла является ссылкой на одну из этих записей; эта ссылка не влияет, если путь впоследствии удален или изменен ссылаться на другой файл. Новый дескриптор открытого файла изначально не разделяется с любым другим процессом, но разделение может возникнуть через *fork*(2).

O_APPEND

Файл открывается в режиме добавления - O_APPEND . Перед каждым вызовом *write* (2), смещение в файле устанавливается в конец файла, как если бы выполнялся вызов LSEEK (2). O_APPEND может привести к повреждению файлов на файловых системах NFS, если несколько процессов добавляют данные в файл одновременно. Это потому, что NFS не поддерживает добавление в файл, так что клиент ядра должен имитировать его, что не может быть сделано без состояния гонки.

Требования к оформлению отчета по лабораторной работе:

Отчет должен:

- содержать коды программ;
- скриншоты результата выполнения каждой программы;
- анализ полученного результата;
- рисунок, демонстрирующий созданные дескрипторы и связь между ними. Схема связи структур, определенных на файлах, должна также содержать struct dentry, struct superblock, struct inode.

Для правильного понимания надо проанализировать структуру FILE* (приведите структуру FILE в отчете по лабораторной работе).

В последней программе надо используя struct stat дополнительно вывести информацию о inode и size, получаемые функцией stat(). Структуру stat привести в отчете.