OS Lab 5

Status	approved
checkbox	☑
class	OS
due date	@Mar 10, 2021

Task

5. Таблица поиска строк в текстовом файле.

📢 Написать программу, которая анализирует текстовый файл, созданный текстовым редактором, таким как `ed(1)` или `vi(1)`. После запроса, который предлагает ввести номер строки, с использованием `printf(3)` программа печатает соответствующую строку текста. Ввод нулевого номера завершает работу программы. Используйте `open(2)`, `read(2)`, `lseek(2)` и `close(2)` для ввода/вывода. Постройте таблицу отступов в файле и длин строк для каждой строки файла. Как только эта таблица построена, позиционируйтесь на начало заданной строки и прочтите точную длину строки.

📢 Подсказка:
Выберите или создайте текстовый файл с короткими строками. Помните, что первая строка начинается с нулевого отступа в файле. Найдите каждый символ перевода строки, запишите его позицию; в программе следует использовать вызов `lseek(fd, 0L, 1)`. Для отладки распечатайте эту таблицу и сравните с таблицей, полученной вручную. Как только таблицы начнут совпадать, можно приступать к запросу номера строки.

Notes

💡 Дескрипторы файлов — это, как правило, небольшие целые положительные чис-ла, используемые ядром для идентификации файлов, к которым обращается кон-кретный процесс. Всякий раз, когда процесс открывает существующий или создает новый файл, ядро возвращает его дескриптор, который затем используется для выполнения операций чтения/записи с файлом.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdbool.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

#define LINECOUNT 256

bool buildFileMap(size_t* offsets, size_t* lengths, int fileDescri
ptor, int* lineCount){
    char buffer[BUFSIZ];
    int offset = 0;
    int lineIdx = 1;
    int bytesRead = 1;

    while (bytesRead > 0){
        bytesRead = read(fileDescriptor, buffer, BUFSIZ);
        if (bytesRead == -1){
            perror("failed to read\n");
            return false;
        }

        for (size_t i = 0; i < bytesRead; ++i){
            lengths[lineIdx]++;
            offset++;
            if (buffer[i] == '\n'){
                offsets[lineIdx] = offset - lengths[lineIdx];
                lineIdx++;
            }
        }
    }
```

Все открытые файлы представлены в ядре **файловыми дескрипторами**. Файловый дескриптор — это неотрицательное целое число. Когда процесс открывает существующий файл или создает новый, ядро возвращает ему файловый дескриптор. Чтобы выполнить запись в файл или чтение из него, нужно передать функции read или write его файловый дескриптор, полученный вызовом функции `open` или `creat`.В соответствии с соглашениями командные оболочки UNIX ассоциируют файловый дескриптор 0 с устройством стандартного ввода процесса, `1` — с устройством стандартного вывода и `2` — с устройством стандартного вывода сообщений об ошибках. Это соглашение используется командными оболочками и большинством приложений, но не является особенностью ядра UNIX. Тем не менее многие приложения не смогли бы работать, если бы это соглашение было нарушено.Хотя значения этих дескрипторов определены стандартом *POSIX.1*, в *POSIX-совместимых* приложениях вместо фактических значений `0`, `1` и `2` следует использовать константы `STDIN_FILENO`, `STDOUT_FILENO` и `STDERR_FILENO`. Определения этих констант находятся в заголовочном файле `<unistd.h>`

Что делает open

- файл ищется в иерархии директорий для получения inode-номера.
- проверяются права доступа файла
- в таблице дескрипторов размещается новый дескриптор.
- проверяются системные структуры файлов и, если необходимо, размещается новое поле.
- если необходимо, размещается новая структура информации о файле.
- соединяется с подходящим драйвером устройства.
- возвращается файловый дескриптор (индекс в таблице файловых

```
    *lineCount = lineIdx;
    return true;
}

int main(int argc, char* argv[]){
    int fileDescriptor = 0;
    if (argc < 2) {
        perror("Usage: filename as argument\n");
        return 0;
    }

    if((fileDescriptor = open(argv[1], O_RDONLY)) == -1) {
        perror("Input file doesn`t exist\n");
        return 0;
    }

    size_t lengths[LINECOUNT] = {0};
    size_t offsets[LINECOUNT] = {0};
    int lineCount = 0;

    buildFileMap(offsets, lengths, fileDescriptor, &lineCount);

    printf("Enter line number from 1 to %d. Enter 0 to exit\n", li
neCount - 1);

    unsigned int lineToPrint = 1;

    while (true){
        printf("line number: ");

        if (lineToPrint == 0){
            break;
        }

        if (scanf("%u", &lineToPrint) != 1){
            fflush(stdin);
            perror("Invalid input");
            continue;
        }

        if (lineToPrint < 0 || lineToPrint > lineCount - 1){
            printf("Line number is an integer number from 1 to %d
\n", lineCount - 1);
            continue;
        }

        if (lineToPrint == 0){
            break;
        }

        if (lseek(fileDescriptor, offsets[lineToPrint], SEEK_SET)
 == -1){
            perror("failed to change pointer position in file\n");
            continue;
        }

        char buffer[BUFSIZ] = {0};

        if (read(fileDescriptor, &buffer, lengths[lineToPrint]) ==
lengths[lineToPrint]){
            printf("%s", &buffer);
        } else {
            perror("error reading file\n");
        }

    }

    close(fileDescriptor);
    return 0;
}
```

Текущая позиция в файле, как правило, является **неотрицательным целым числом**, которым выражается смещение в байтах от начала файла. Операции чтения и записи обычно выполняются с текущей позиции в файле и увеличивают её на количество записанных или прочитанных байтов.

Обычно смещение относительно текущей позиции (в параметрах функции `lseek()`) должно быть неотрицательным целым числом. Однако некоторые устройства поддерживают отрицательные смещения. Поскольку отрицательные смещения все-таки возможны, возвращаемое функцией `lseek()` значение следует сравнивать **именно с числом –1**, а не проверять, не является ли оно отрицательным.

Все открытые файлы в ядре представлены файловыми дескрипторами – неотрицательными целыми числами. Когда процесс открывает/создает файл, ядро возвращает ему дескриптор.

По соглашениям командные оболочки UNIX ассоциируют дескриптор 0 с устройством стандартного ввода, 1 – стандартного вывода, 2 – стандартного вывода сообщений об ошибках.

Под файловые дескрипторы отводится диапазон от `0` до `OPEN_MAX - 1`.

Значение `OPEN_MAX` ограничено «мягким» и «жестким» пределами. Мягкий (административный) предел устанавливается `setrlimit(2)` с командой `RLIMIT_NOFILE` или командой `ulimit(1)`. Жёсткий предел устанавливается настройками ядра системы. Значение жёсткого предела можно определить системным вызовом `sysconf(2)` с параметром _SC_OPEN_MAX. В Solaris, жесткий предел устанавливается параметром `rlim_fd_max` в файле `/etc/system (system(4))`; его изменение требует административных привилегий и перезагрузки системы.

1. Каждому процессу соответствует запись в таблице процессов. С каждой записью в таблице процессов связана таблица открытых файловых дескрипторов, которую можно представить как таблицу, в которой каждая строка соответствует одному файловому дескриптору. Для каждого дескриптора хранится следующая информация: **а)** флаги дескриптора **б)** указатель на запись в таблице файлов.
2. Все открытые файлы представлены в ядре таблицей файлов. Каждая запись в таблице содержит: а) **флаги состояния файла**, такие как чтение, запись, добавление в конец, синхронный режим операций ввода/вывода, неблокирующий режим б) **текущая позиция в файле**; в) указатель на запись в таблице виртуальных узлов (v-node).
3. Каждому открытому файлу (или устройству) соответствует **структура виртуального узла** (v-node) с информацией о типе файла. Для большинства файлов структура v-node также содержит **индексный узел** (i-node) файла. Эта информация считывается с диска при открытии файла, поэтому вся информация о файле сразу же становится доступной. Индексный узел (i-node) содержит, например, сведения о владельце файла, размере файла, указатели на блоки данных на диске.

В настоящее время определен лишь один **флаг дескриптора FD_CLOEXEC**, означающий, что дескриптор должен быть закрыт в случае использования функции exec.

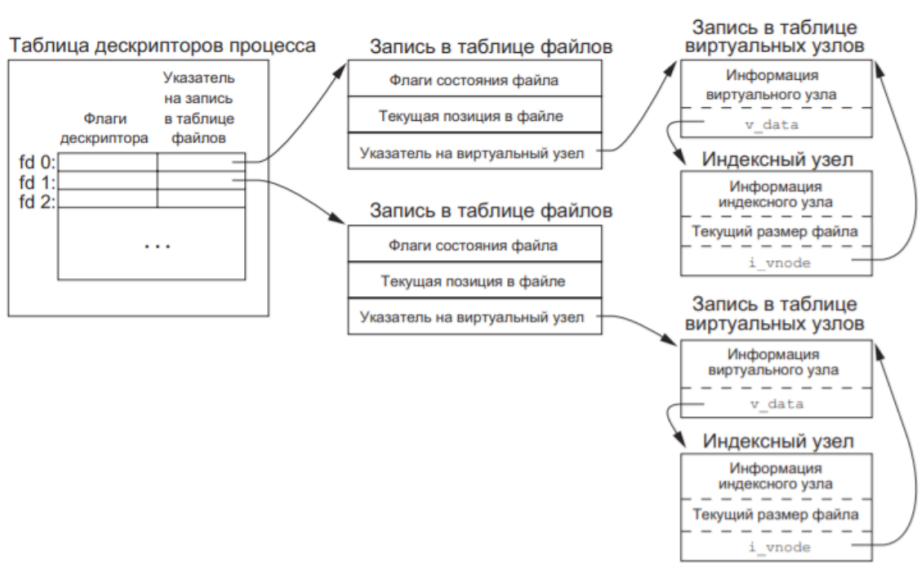
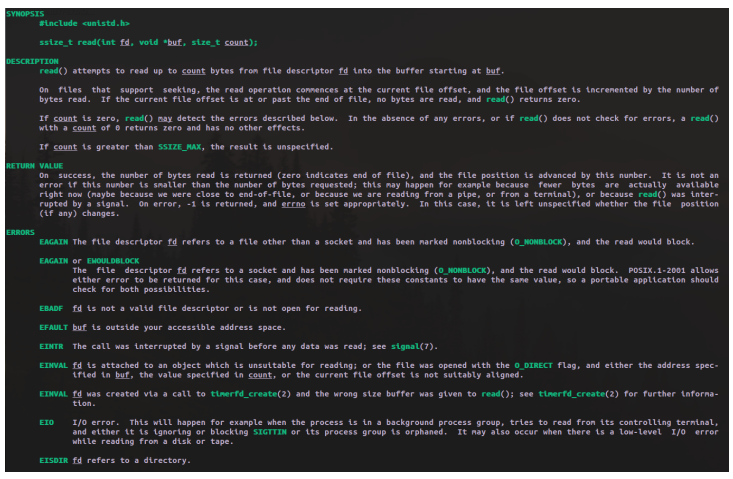
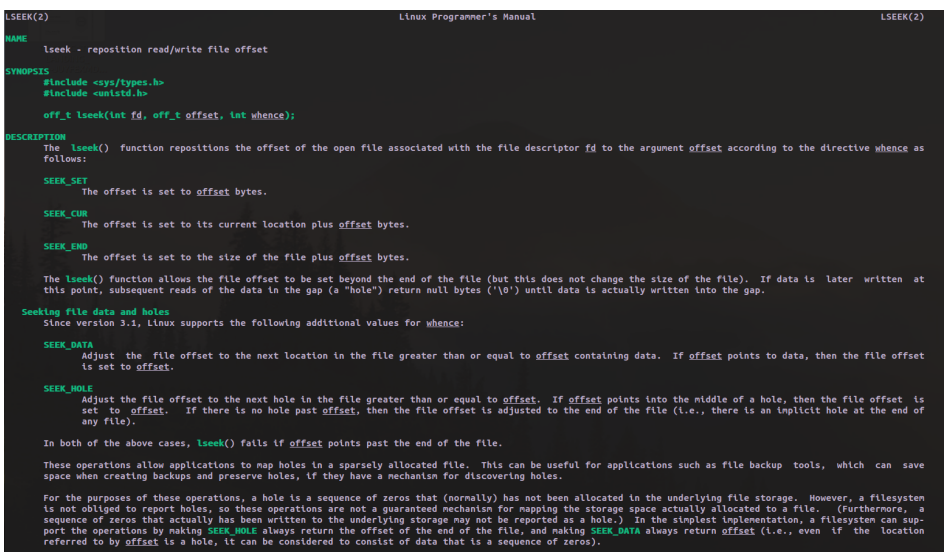


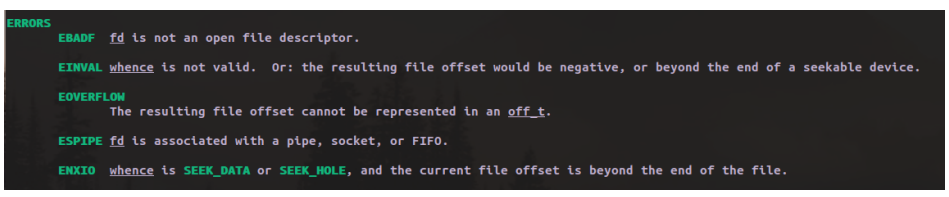
Рис. 3.1. Структуры данных ядра для открытых файлов



```
SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

       On  files  that  support  seeking, the read operation commences at the current file offset, and the file offset is incremented by the number of
       bytes read.  If the current file offset is at or past the end of file, no bytes are read, and read() returns zero.

       If count is zero, read() may detect the errors described below.  In the absence of any errors, or if read() does not check for errors, a read()
       with a count of 0 returns zero and has no other effects.

       If count is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE
       On  success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.  It is not an
       error if this number is smaller than the number of bytes requested; this may happen for example because  fewer  bytes  are  actually  available
       right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was inter-
       rupted by a signal.  On error, -1 is returned, and errno is set appropriately.  In this case, it is left unspecified whether the file  position
       (if any) changes.

ERRORS
       EAGAIN The file descriptor fd refers to a file other than a socket and has been marked nonblocking (O_NONBLOCK), and the read would block.

       EAGAIN or EWOULDBLOCK
              The  file  descriptor fd refers to a socket and has been marked nonblocking (O_NONBLOCK), and the read would block.  POSIX.1-2001 allows
              either error to be returned for this case, and does not require these constants to have the same value, so a portable application should
              check for both possibilities.

       EBADF  fd is not a valid file descriptor or is not open for reading.

       EFAULT buf is outside your accessible address space.

       EINTR  The call was interrupted by a signal before any data was read; see signal(7).

       EINVAL fd is attached to an object which is unsuitable for reading; or the file was opened with the O_DIRECT flag, and either the address spec-
              ified in buf, the value specified in count, or the current file offset is not suitably aligned.

       EINVAL fd was created via a call to timerfd_create(2) and the wrong size buffer was given to read(); see timerfd_create(2) for further informa-
              tion.

       EIO    I/O error.  This will happen for example when the process is in a background process group, tries to read from its controlling terminal,
              and either it is ignoring or blocking SIGTTIN or its process group is orphaned.  It may also occur when there is a low-level  I/O  error
              while reading from a disk or tape.

       EISDIR fd refers to a directory.
```



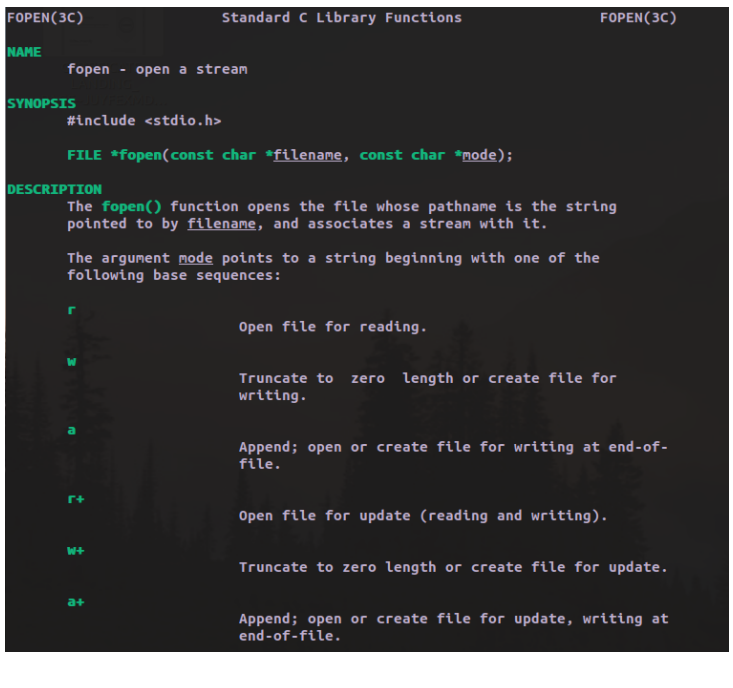
```
LSEEK(2)                                                Linux Programmer's Manual                                                LSEEK(2)

NAME
       lseek - reposition read/write file offset

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       off_t lseek(int fd, off_t offset, int whence);

DESCRIPTION
       The  lseek()  function repositions the offset of the open file associated with the file descriptor fd to the argument offset according to the directive whence as
       follows:

       SEEK_SET
              The offset is set to offset bytes.

       SEEK_CUR
              The offset is set to its current location plus offset bytes.

       SEEK_END
              The offset is set to the size of the file plus offset bytes.

       The lseek() function allows the file offset to be set beyond the end of the file (but this does not change the size of the file).  If data is  later  written  at
       this point, subsequent reads of the data in the gap (a "hole") return null bytes ('\0') until data is actually written into the gap.

   Seeking file data and holes
       Since version 3.1, Linux supports the following additional values for whence:

       SEEK_DATA
              Adjust  the  file offset to the next location in the file greater than or equal to offset containing data.  If offset points to data, then the file offset
              is set to offset.

       SEEK_HOLE
              Adjust the file offset to the next hole in the file greater than or equal to offset.  If offset points into the middle of a hole, then the file offset  is
              set  to  offset.   If there is no hole past offset, then the file offset is adjusted to the end of the file (i.e., there is an implicit hole at the end of
              any file).

       In both of the above cases, lseek() fails if offset points past the end of the file.

       These operations allow applications to map holes in a sparsely allocated file.  This can be useful for applications such as file backup  tools,  which  can  save
       space when creating backups and preserve holes, if they have a mechanism for discovering holes.

       For the purposes of these operations, a hole is a sequence of zeros that (normally) has not been allocated in the underlying file storage.  However, a filesystem
       is not obliged to report holes, so these operations are not a guaranteed mechanism for mapping the storage space actually allocated to a file.  (Furthermore,  a
       sequence of zeros that actually has been written to the underlying storage may not be reported as a hole.)  In the simplest implementation, a filesystem can sup-
       port the operations by making SEEK_HOLE always return the offset of the end of the file, and making SEEK_DATA always return offset (i.e., even  if  the  location
       referred to by offset is a hole, it can be considered to consist of data that is a sequence of zeros).
```

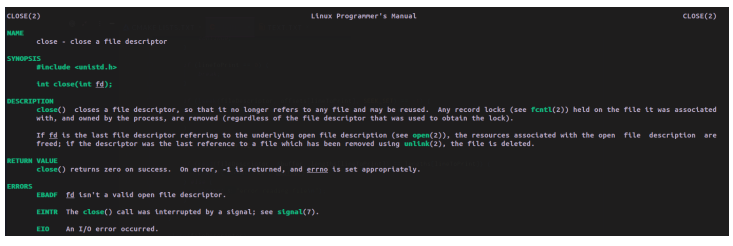


```
ERRORS
       EBADF  fd is not an open file descriptor.

       EINVAL whence is not valid.  Or: the resulting file offset would be negative, or beyond the end of a seekable device.

       EOVERFLOW
              The resulting file offset cannot be represented in an off_t.

       ESPIPE fd is associated with a pipe, socket, or FIFO.

       ENXIO  whence is SEEK_DATA or SEEK_HOLE, and the current file offset is beyond the end of the file.
```

```
FREAD(3)                                                Linux Programmer's Manual                                                FREAD(3)

NAME
       fread, fwrite - binary stream input/output

SYNOPSIS
       #include <stdio.h>

       size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

       size_t fwrite(const void *ptr, size_t size, size_t nmemb,
                     FILE *stream);

DESCRIPTION
       The function fread() reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.

       The function fwrite() writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.

       For nonlocking counterparts, see unlocked_stdio(3).

RETURN VALUE
       On  success,  fread()  and  fwrite()  return the number of items read or written.  This number equals the number of bytes transferred only when size is 1.  If an
       error occurs, or the end of the file is reached, the return value is a short item count (or zero).

       fread() does not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.
```

```
typedef struct {
  unsigned char   _flag;          /* state of the stream */
  int             _file;          /* file descriptor */
  ssize_t         _len;           /* total len of file */
  ssize_t         _offset;        /* offset within the file */
  char            _name[256];     /* name of the file (for debugging) */
} FILE;
```





Reading list

- [ ]