ИНЖЕНЕРНО-
МАТЕМАТИЧЕСКАЯ
ШКОЛА
НИУ ВШЭ И VK

Файлы и консольный ввод

education

ВЫСШАЯ • ШКОЛА • ЭКОНОМИКИ •

Обработка ошибок

- Для обработки ошибок в СИ используется библиотека <errno.h>, в которой лежит глобальная переменная errno Эта переменная хранит код последней ошибки, возникшей в программе. В <errno.h> также определены константы для различных типов ошибок, такие как:
  - EPERM — операция не разрешена
  - ENOENT — нет такого файла или каталога
  - ESRCH — нет такого процесса
  - EINTR — прерван системный вызов
  - EIO — ошибка ввода/вывода
  - ENXIO — нет такого устройства или адреса
- Перед выполнением функции, например strtol, переменную errno необходимо занулить, чтобы обезопасить себя от считывания предыдущих ошибок.
- Функции perror (stdio.h) и strerror (string.h) позволяют получить текстовое описание ошибки.

```
FILE *file = fopen("example.txt", "r");
if (file == NULL) {
    printf("Ошибка открытия файла: %s\n", strerror(errno));
}
```

Открытие файла

- Файл - именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.
- Как правило, последовательность работы с файлами такая: открытие, чтение/запись, закрытие
- Для открытия файла используется функция `int open(const char *pathname, int flags, mode_t mode);` , которая возвращает дескриптор файла.
- mode - права, которые даются файлу при создании, а flags - режим, в котором открывается файл:
  - O_RDONLY — открыть файл только для чтения.
  - O_WRONLY — открыть файл только для записи.
  - O_RDWR — открыть файл для чтения и записи.
  - O_CREAT — создать файл, если он не существует.
  - O_EXCL — с O_CREAT, вызывает ошибку, если файл уже существует.
  - O_TRUNC — обрезать файл до нулевой длины, если он существует.
  - O_APPEND — добавлять данные в конец файла.
  - O_NONBLOCK — неблокирующий режим открытия.
  - O_DSYNC — синхронизировать записи данных.
  - O_SYNC — синхронизировать записи данных и метаданных.
  - O_NOFOLLOW — не следовать по символьным ссылкам.

Чтение и запись в файл

- Для чтения и записи в файл используются функции read() и write():

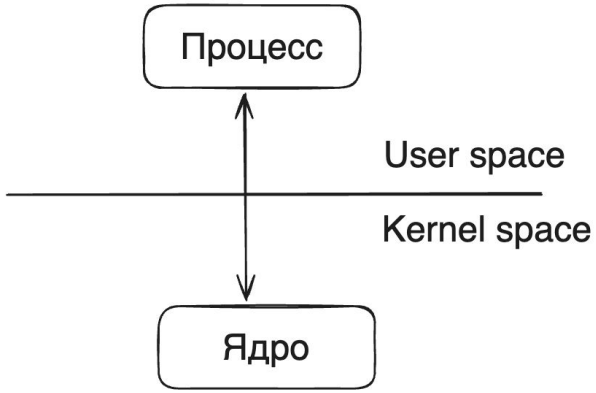
```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- Обе функции возвращают количество записанных/считанных байтов в файл. Если запись не удалась, то возвращается -1.
- Хорошей практикой при использовании этих функций является работа не сразу со всем объемом данных из файла, а чтение/запись батчей в цикле.
- После открытия и работы с файлом, его необходимо закрыть. При открытии файла процессу выделяется файловый дескриптор - структура данных, указывающая на реальный файл в операционной системе. Каждый процесс не может иметь больше 2000 дескрипторов, поэтому их важно вовремя высвобождать.
- Чтобы код был более читаемым и отлаживаемым, файл надо закрывать в той же функции, которая его открыла.
- close() выставляет ошибку, поэтому результат закрытия файла тоже полезно проверять:

```
int fd = open("file.txt", O_RDONLY);
// ... операции с файлом ...
if (close(fd) == -1) {
    perror("Ошибка закрытия файла");
}
```

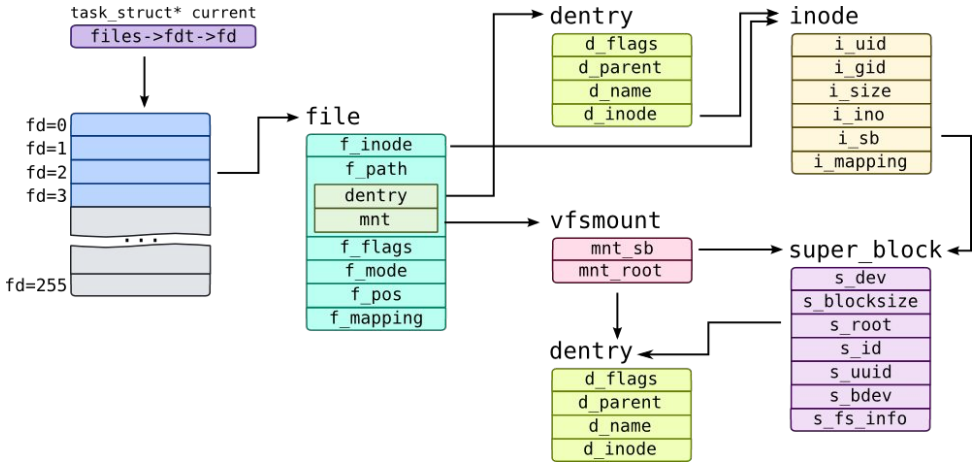
Системный вызов

- Работа с файлами происходит с помощью системных вызовов.
- Системные вызовы - API, которое предоставляет операционная система, и которое позволяет приложениям взаимодействовать с ресурсами ОС.
- Когда приложение вызывает open/read/write, то происходит переход в пространство ядра и процесс уходит в ожидание до того момента, пока системный вызов выполняется.



Файловая система

- Есть различные файловые системы, но они предоставляют одинаковый интерфейс для операционной системы - VFS, которая состоит из следующих абстракций:
  - **superblock**: информация о файловой системе.
  - **inode**: метаданные файла (размер, права доступа)
  - **directory entry**: элементы каталогов, связывающие имена файлов с inode.
  - **file**: представление открытого файла в памяти.



Проблемы write/read

- write и read - низкоуровневые операции, которые не позволяют эффективно работать с файлами, особенно в режиме построчного чтения/записи, так как выполняют системные вызовы при каждом обращении к ним.
- Для оптимизации в си есть набор буферизированных функций для работы с файлами:

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

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

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

int fclose(FILE *stream);
```

- Эти функции - высокоуровневые и имеют внутри себя буффер, т.е. производят непосредственную запись в файл при его достаточном наполнении.
- Помимо этого для работы с FILE* есть большой набор функций, сильно облегчающих написание кода для работы с файлами.

Функции для работы с файлами

- В stdio.h есть достаточно большой набор функций для для работы с файлами:
- `int fgetc(FILE *stream);` // считывает следующий символ из указанного потока.
- `char *fgets(char *str, int n, FILE *stream);` // считывает строку из указанного потока до n-1 символов или до встречи с переводом строки.
- `int fscanf(FILE *stream, const char *format, ...);` // считывает отформатированный ввод из указанного потока, аналогично scanf, но из файла.
- `int fputc(int char, FILE *stream);` // записывает один символ в указанный поток.
- `int fputs(const char *str, FILE *stream);` // записывает строку в указанный поток.
- `int fprintf(FILE *stream, const char *format, ...);` // записывает отформатированный вывод в указанный поток, аналогично printf, но в файл.
- `int fseek(FILE *stream, long offset, int whence);` // устанавливает позицию в файле относительно начала, текущей позиции или конца файла.
- `long ftell(FILE *stream);` // возвращает текущую позицию в файле.
- `void rewind(FILE *stream);` // устанавливает позицию в файле на начало и очищает флаги ошибок и конца файла.
- `int fgetpos(FILE *stream, fpos_t *pos);` // сохраняет текущую позицию в файле в переменную типа fpos_t.
- `int fsetpos(FILE *stream, const fpos_t *pos);` // устанавливает позицию в файле на ранее сохраненную с помощью fgetpos.
- `int feof(FILE *stream);` // проверяет, достигнут ли конец файла для указанного потока.
- `int ferror(FILE *stream);` // проверяет, установлены ли флаги ошибок для указанного потока.
- `void clearerr(FILE *stream);` // очищает флаги ошибок и конца файла для указанного потока.
- `int fflush(FILE *stream);` // сбрасывает буфер вывода указанного потока, гарантируя запись всех буферизованных данных.
- `void setbuf(FILE *stream, char *buffer);` // устанавливает буфер для указанного потока.
- `int setvbuf(FILE *stream, char *buffer, int mode, size_t size);` // устанавливает режим буферизации для потока (_IONBF, _IOLBF, _IOFBF) и при необходимости задает буфер.
- `FILE *fdopen(int fd, const char *mode);` // ассоциирует файловый дескриптор с потоком FILE.
- `int fileno(FILE *stream);` // возвращает файловый дескриптор, связанный с потоком.

Бинарный режим файлов

- Если к флагу открытия в fopen дописать ‘b’, то файл откроется в бинарном режиме, например “rb” . Это позволяет нам работать с файлом побайтово. Для чтения и записи все так же используются fread и fwrite, но мы можем использовать дополнительный набор функций:

```
int fseek(FILE *stream, long offset, int whence); // устанавливает текущее положение в файле
```

stream — указатель на объект типа FILE, ассоциированный с файлом.
offset — смещение в байтах относительно позиции, заданной параметром whence.
whence — базовая позиция для смещения. Может принимать следующие значения:
SEEK_SET — начало файла.
SEEK_CUR — текущая позиция указателя.
SEEK_END — конец файла.

```
long ftell(FILE *stream); - возвращает текущее положение в файле
void rewind(FILE *stream); - перемещает указатель в начало файла
```

Coding time

- Считывание файла по строкам
- Запись в файл по строкам
- Парсим конфиг
- Ищем слово в тексте
- Используем файл как массив

Работа с консольным вводом

- Основные функции для работы с вводом:

```
int scanf(const char *format, …); // форматированный ввод
int getchar(void); // получение символа
char *fgets(char *str int n, FILE *stream); // считывает строку
int getch(void); // получает символ без необходимости нажимать enter, лежит в conio.h
```

- Если мы хотим считать строку в каком-то определенном формате, то можно использовать sscanf и проверить число аргументов, которое ей удалось считать

```
char date[11];
int day, month, year;

printf("Enter a date (DD/MM/YYYY): ");
fgets(date, sizeof(date), stdin);

if (sscanf(date, "%2d/%2d/%4d", &day, &month, &year) == 3) {
    printf("Day: %d, Month: %d, Year: %d\n", day, month, year);
} else {
    fprintf(stderr, "Invalid date format.\n");
}
```

Спасибо за внимание!