Буферизованный ввод-вывод, часть 5

Наумов Д.А., доц. каф. КТ

Операционные системы и системное программное обеспечение, 2019

Содержание лекции

1. Буферизованный ввод-вывод
   - Ввод-вывод с пользовательским буфером
   - Стандартный ввод-вывод
   - Открытие файлов
   - Закрытие потоков данных
   - Считывание из потока данных
   - Запись в поток данных
   - Позиционирование в потоке данных
   - Сброс потока данных
   - Обработка ошибок

Буферизованный ввод-вывод

Блок

абстракция, представляющая мельчайший компонент системы, предназначенный для хранения данных в файловой системе.

- внутри ядра все операции файловой системы трактуются именно в контексте блоков;
- любые действия ввода-вывода не могут осуществляться над объемом информации, не превышающим размер одного блока, а также над объемом данных, который не выражается целым числом, кратным размеру блока.

Ввод-вывод с пользовательским буфером позволяет приложениям естественным образом считывать и записывать любые объемы данных, но осуществлять ввод-вывод в размере целых блоков данной файловой системы.

Ввод-вывод с пользовательским буфером

Программы, которым приходится выполнять множество мелких системных вызовов к обычным файлам, часто осуществляют вводвывод с пользовательским буфером - **буферизация, выполняемая в пользовательском пространстве**.

Пример с использованием программы dd, работающей в пользовательском пространстве

Блок в 1 байт:

```
dd bs=1 count=2097152 if=/dev/zero of=pirate
```

Блок в 1024 байт:

```
dd bs=1024 count=2048 if=/dev/zero of=pirate
```

Ввод-вывод с пользовательским буфером

Размер блока, байты	Реальное время, секунды	Пользовательское время, секунды	Системное время, секунды
1	18,707	1,118	17,549
1024	0,025	0,002	0,023
1130	0,035	0,002	0,027

- уа практике размер блока обычно составляет 512, 1024, 2048, 4096 или 8192 байт;
- для значительного повышения производительности достаточно просто выполнять операции фрагментами, которые являются целочисленными кратными или делителями размера блока;
- узнать размер блока на конкретном устройстве: системный вызов stat() или команда stat(1).

Принцип работы пользовательского буфера

Запись данных:

- по мере того как данные записываются, они сохраняются в буфере в пределах адресного пространства конкретной программы;
- когда размер буфера достигает установленного предела, называемого размером буфера, содержимое этого буфера переносится на диск за одну операцию записи.

Считывание данных:

- поступающие от приложения запросы на считывание имеют разные размеры и обслуживаются не из файловой системы напрямую, а удовлетворяются фрагментами, получаемыми через буфер;
- по мере того как приложение считывает все больше и больше информации, данные выдаются из буфера кусками;
- как только буфер пустеет, начинается считывание следующего сравнительно крупного фрагмента, выровненного по границам блоков.

Стандартный ввод-вывод

- stdio - стандартная библиотека ввода-вывода Си; независимое от платформы решение для пользовательской буферизации.
- Си не содержит никакой встроенной поддержки ключевых слов для операций ввода-вывода.
- в процессе эволюции языка пользователи разработали стандартные наборы процедур, обеспечивающих основную функциональность - эти функции были объединены в стандартную библиотеку ANSI C (C89);

Указатели файлов

- Процедуры стандартного ввода-вывода используют свой уникальный идентификатор - указатель файла.
- В библиотеке C указатель файла ассоциируется с файловым дескриптором (отображается на него).
- Указатель файла представлен как указатель на определение типа FILE, определяемый в <stdio.h>.

Открытие файлов

```
#include <stdio.h>

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

Эта функция открывает файл *path*, поведение которого определено в *mode*, и ассоциирует с ним новый поток данных.
Возможные режимы отрытия файлы:

- **r** — файл открывается для чтения. Поток данных устанавливается в начале файла.
- **r+** — файл открывается как для чтения, так и для записи. Поток данных устанавливается в начале файла.
- **w** — файл открывается для записи. Если файл существует, то он усекается до нулевой длины. Если файл не существует, он создается. Поток данных устанавливается в начале файла.

Открытие файлов

```
#include <stdio.h>

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

- **w+** — файл открывается для чтения и для записи. Если файл существует, то он усекается до нулевой длины. Если файл не существует, он создается. Поток данных устанавливается в начале файла.
- **a** — файл открывается для дополнения в режиме дозаписи. Если файл не существует, то он создается. Поток данных устанавливается в конце файла. Все вводимые данные дозаписываются в файл.
- **a+** — файл открывается для дополнения и считывания в режиме дозаписи. Если файл не существует, то он создается. Поток данных устанавливается в конце файла. Все вводимые данные дозаписываются в файл.

Открытие файлов

- В случае успеха функция fopen() возвращает допустимый указатель FILE.
- При ошибке она возвращает NULL и устанавливает errno соответствующее значение

```
FILE *stream;

stream = fopen ("/etc/manifest", "r");
if (!stream)
        /* ошибка */
```

Открытие потока данных при помощи файлового дескриптора

Функция fdopen() преобразует уже открытый файловый дескриптор (fd) в поток данных:

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

- могут использоваться те же режимы, что и с функцией fopen(), при этом они должны быть совместимы с режимами, которые изначально применялись для открытия файлового дескриптора.
- режимы w и w+ можно указывать, но они не будут приводить к усечению файла.
- Поток данных устанавливается в файловую позицию, которая соответствует данному файловому дескриптору.
- получить дексриптор - *int fileno(FILE*stream)*;

Открытие потока данных при помощи файлового дескриптора

Функция fdopen() преобразует уже открытый файловый дескриптор (fd) в поток данных:

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

- После преобразования файлового дескриптора в поток данных ввод-вывод больше не выполняется напрямую с этим файловым дескриптором. Тем не менее это не возбраняется.
- При закрытии потока данных закроется и файловый дескриптор.
- В случае успеха fdopen() возвращает допустимый указатель файла, при ошибке она возвращает NULL и присваивает errno соответствующее значение.

Открытие потока данных при помощи файлового дескриптора

Например, следующий код открывает файл */home/kidd/map.txt* с помощью системного вызова *open()*, а потом создает ассоциированный поток данных, опираясь на базовый файловый дескриптор:

```
FILE *stream;
int fd;

fd = open ("/home/kidd/map.txt", O_RDONLY);
if (fd == -1)
        /* ошибка */

stream = fdopen (fd, "r");
if(!stream)
       /* ошибка */
```

Закрытие конкретного потока

Функция *fclose()* закрывает конкретный поток данных:

```
#include <stdio.h>

int fclose(FILE*stream);
```

- сначала сбрасываются на диск все буферизованные, но еще не записанные данные;
- в случае успеха fclose() возвращает 0;
- При ошибке она возвращает EOF (конец файла) и устанавливает errno в соответствующее значение.

Закрытие всех потоков данных

Функция *fcloseall()* закрывает все потоки данных, ассоциированные с конкретным процессом, в частности используемые для стандартного ввода, стандартного вывода и стандартных ошибок:

```
#define _GNU_SOURCE

#include <stdio.h>

int fcloseall(void);
```

- перед закрытием все потоки данных сбрасываются на диск;
- функция является специфичной для Linux и всегда возвращает 0.

Считывание одного символа в момент времени

Функция *fgetc()* используется для считывания отдельного символа из потока данных:

```
#include <stdio.h>

int fgetc(FILE*stream);
```

- функция считывает следующий символ из stream и возвращает его как unsigned char, приведенный к int.
- такое приведение осуществляется, чтобы получить достаточно широкий диапазон для уведомлений EOF или описания ошибок: в таких случаях возвращается EOF
- возвращаемое значение fgetc()должно быть сохранено в int. Сохранение в char — распространенный и опасный промах, ведь в таком случае вы не можете обнаруживать ошибки.

Считывание одного символа в момент времени

В следующем коде мы считываем отдельно взятый символ из stream, проверяем наличие ошибок и выводим результат как char:

```
int c;

c = fgetc (stream);
if (c == EOF)
        /* ошибка */
else

        printf ("c=%c\n", (char) c);
```

Возврат символа в поток данных

В рамках стандартного вводавывода предоставляется функция для перемещения символа обратно в поток данных. С ее помощью вы можете «заглянуть» в поток данных и возвратить символ обратно, если окажется, что он вам не подходит:

```
#include<stdio.h>

int ungetc(int c, FILE*stream);
```

- при каждом вызове мы отправляем обратно в поток данных stream символ c, приведенный к unsigned char.
- в случае успеха возвращается c, в случае ошибки — EOF.
- если возвращаем в поток данных несколько символов - образуется стек
- лишь один такой возврат должен гарантированно быть успешным (при отсутствии «вклинивающихся» запросов на считывание).

Считывание целой строки

Функция *fgets()* считывает строку из указанного потока данных:

```
#include <stdio.h>

char * fgets (char *str, int size, FILE *stream);
```

- функция может считать из потока данных stream количество байтов, как максимум на один меньшее, чем количество size байт
- результат считывания сохраняется в str.
- символ нуля \0 сохраняется в буфере после последнего считанного байта.
- считывание прекращается, когда достигнут конец файла или первый символ новой строки.
- если начинается считывание новой строки, то в str сохраняется $\backslash n$.
- В случае успеха возвращается str, в случае ошибки — NULL

Считывание целой строки

Пример:

```
char buf[LINE_MAX];

if (!fgets (buf, LINE_MAX, stream))
        /* ошибка */
```

- POSIX определяет LINE_MAX в <limits.h>: это максимальный размер, который может иметь строка ввода, чтобы интерфейсы POSIX для манипуляций со строками могли ею оперировать.
- В библиотеке C такое ограничение отсутствует — строки могут быть любого размера.

Считывание двоичных данных

Для того, чтобы считывать и записывать сложные двоичные данные, в частности структуры Си, предоставляется функция *fread()*:

```
#include <stdio.h>

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

- При вызове fread() мы можем прочитать вплоть до nr фрагментов данных, каждый размером size байт.
- Считывание происходит из потока данных stream в буфер, указанный buf.
- Значение файлового указателя увеличивается на число, равное количеству прочитанных байтов.
- Возвращается количество считанных элементов (но не количество считанных байтов!). При ошибке или достижении конца файла функция возвращает значение, меньшее чем nr. К сожалению, мы не можем узнать, какое именно из двух условий наступило, — для этого потребуется специально применить функции ferror() и feof()

Считывание двоичных данных

Для того, чтобы считывать и записывать сложные двоичные данные, в частности структуры Си, предоставляется функция *fread()*:

```
#include <stdio.h>

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

- при ошибке или достижении конца файла функция возвращает значение, меньшее чем nr.
- К сожалению, мы не можем узнать, какое именно из двух условий наступило, — для этого потребуется специально применить функции ferror() и feof().

Считывание двоичных данных

Простейший образец fread() применяется для считывания одного элемента линейных байтов из указанного потока данных:

```
char buf[64];
size_t nr;

nr = fread (buf, sizeof(buf), 1, stream);
if(nr== 0)
      /* ошибка */
```

Запись отдельного символа

Функция, парная fgetc(), называется fputc():

```
#include <stdio.h>

int fputc (int c, FILE *stream);
```

- Функция fputc() записывает байт, указанный в c (приведенный к unsigned char), в поток данных, указанный в stream.
- При успешном завершении операции функция возвращает c. В противном случае она возвращает EOF и присваивает errno соответствующее значение.

```
if(fputc('p', stream) == EOF)
                /* ошибка */
```

Запись строки символов

Функция fputs() используется для записи целой строки в заданный поток данных:

```
#include <stdio.h>

int fputs (const char *str, FILE *stream);
```

- при вызове fputs() все содержимое строки, оканчивающейся нулем, записывается в поток данных, указанный в stream.
- сама эта строка указывается в str.
- в случае успеха fputs() возвращает неотрицательное число. При ошибке она возвращает EOF.

Запись строки символов

В следующем примере файл открывается для внесения данных в режиме дозаписи. Указанная строка записывается в ассоциированный поток данных, после чего этот поток данных закрывается:

```
FILE *stream;

stream = fopen ("journal.txt", "a");
if (!stream)
        /* ошибка */

if (fputs ("The ship is made of wood.\n", stream) == EOF)
        /* ошибка */

if (fclose (stream) == EOF)
        /* ошибка */
```

Запись двоичных данных

Для непосредственного сохранения двоичных данных, например переменных языка C, в рамках стандартного вводавывода предоставляется функция fwrite():

```
#include <stdio.h>

size_t fwrite (void *buf,
               size_t size,
               size_t nr,
               FILE *stream);
```

- при вызове fwrite() в поток данных stream записывается вплоть до nr элементов, каждый до size в длину.
- Берутся данные из буфера, указанного в buf.
- Значение файлового указателя увеличивается на число записанных **байтов**.
- Возвращается количество успешно записанных **элементов**.
- Возвращаемое значение меньше nr означает ошибку.

Позиционирование в потоке данных

Функция fseek(), наиболее распространенный интерфейс позиционирования из инструментов стандартного вводавывода, управляет файловой позицией в потоке данных stream в зависимости от значений offset и whence:

```
#include <stdio.h>

int fseek (FILE *stream, long offset, int whence);
```

- Если аргумент whence имеет значение SEEK_SET, то файловая позиция устанавливается в offset.
- Если whence равен SEEK_CUR, файловая позиция получает значение, равное «текущая позиция плюс offset».
- Если whence имеет значение SEEK_END, то файловая позиция устанавливается в значение, равное «конец файла плюс offset».

Позиционирование в потоке данных

Функция fseek(), наиболее распространенный интерфейс позиционирования из инструментов стандартного вводавывода, управляет файловой позицией в потоке данных stream в зависимости от значений offset и whence:

```
#include <stdio.h>

int fseek (FILE *stream, long offset, int whence);
```

- При успешном завершении функция fseek() возвращает 0, стирает индикатор EOF и отменяет любые эффекты функции ungetc() (при их наличии).
- При ошибке она возвращает –1 и устанавливает errno в соответствующее значение.
- Самые распространенные ошибки — это недействительный поток данных (EBADF) и недействительный аргумент whence (EINVAL).

Получение актуальной позиции в потоке данных

Функция ftell() возвращает текущую позицию из потока данных stream:

```
#include <stdio.h>

long ftell (FILE *stream);
```

- При ошибке она возвращает –1 и устанавливает errno в соответствующее значение.

Позиционирование в потоке данных

В стандартной библиотеке вводавывода есть интерфейс, позволяющий выписать содержимое пользовательского буфера в ядро.
Он гарантирует, что все данные, записанные в поток данных, будут сброшены к ядру с помощью write().

```
#include <stdio.h>

int fflush (FILE *stream);
```

- При вызове этой функции все незаписанные данные из потока данных, указанного в stream, сбрасываются в буфер ядра.
- Если значение stream равно NULL, то все открытые потоки данных этого процесса записываются в буфер ядра.
- В случае успеха fflush() возвращает 0.
- В случае ошибки эта функция возвращает EOF и присваивает errno соответствующее значение.

Ошибки и конец файла

Проверика статуса конкретного потока данных и определение, установлен ли на потоке данных stream индикатор ошибки:

```
#include<stdio.h>

int ferror(FILE*stream);
```

Функция feof() проверяет, установлен ли на потоке данных stream индикатор EOF:

```
#include<stdio.h>

int feof(FILE*stream);
```

Функция clearerr() удаляет с потока данных stream индикаторы ошибки и EOF:

```
#include <stdio.h>

void clearerr(FILE*stream);
```

Выводы

Стандартный ввод-вывод и пользовательская буферизация оправданны, если выполняются все следующие условия:

- предположительно вы будете делать много системных вызовов и хотите минимизировать сопутствующие издержки, объединив несколько таких вызовов в один;
- очень важна высокая производительность, и вы хотите гарантировать, что весь вводвывод будет осуществляться поблочными фрагментами, четко ограниченными размерами блоков;
- интересующие вас шаблоны доступа основаны на символах или строках, и вам нужны интерфейсы, обеспечивающие такой доступ без выполнения излишних системных вызовов;
- вы предпочитаете использовать сравнительно высокоуровневый интерфейс, а не низкоуровневые системные вызовы Linux.

Однако максимальная гибкость обеспечивается, когда вы работаете непосредственно с системными вызовами Linux.