Основы низкоуровневого файлового ввода-вывода в ОС GNU/Linux

1.Файловый ввод/вывод

Для программ на языке С доступны два уровня реализации ввода-вывода: “высокоуровневый”, представляющий собой надстройку с собственной буферизацией и дополнительными средствами абстракции (сюда относятся, в частности, функции `printf/scanf`), и низкоуровневый, фактически представляющий собой обёртки к одноимённым системным вызовам ОС.

Низкоуровневый ввод/вывод осуществляется двумя функциями: `read()` и `write()`. Обращения к этим функциям имеют вид:

```
n_read = read(fd,buf,n);
n_written = write(fd,buf,n);
```

Первым аргументом обеих функций является дескриптор файла. Вторым аргументом является буфер в программе, откуда или куда должны поступать данные. Третий аргумент – это число подлежащих пересылке байтов.

При каждом обращении возвращается счетчик байтов, указывающий фактическое число переданных байтов. При чтении возвращенное число байтов может оказаться меньше, чем запрошенное число. Возвращенное нулевое число байтов означает конец файла, а «-1» указывает на наличие какой-либо ошибки. При записи возвращенное значение равно числу фактически записанных байтов; несовпадение этого числа с числом байтов, которое предполагалось записать, обычно свидетельствует об ошибке.

Количество байтов, подлежащих чтению или записи, может быть совершенно произвольным. Двумя самыми распространенными величинами являются «1», что означает передачу одного символа за обращение (т.е. без использования буфера), и «512», что соответствует физическому размеру блока на многих периферийных устройствах. Этот последний размер будет наиболее эффективным, но даже ввод или вывод по одному символу за обращение не будет слишком дорогим.

Пример. Копирование ввода на вывод.

```
#include<stdio.h>
#define BUFSIZE 512
void main() /*copy input to output*/
{
    char buf[BUFSIZE];
    int n;
    while((n=read(0,buf,BUFSIZE))>0)
        write(1,buf,n);
}
```

Если размер файла не будет кратен BUFSIZE, то при очередном обращении к `read()` будет возвращено меньшее число байтов, которые затем записываются с помощью `write()`; при следующем после этого обращении к `read()` будет возвращен нуль. Выход осуществляется по нажатию сочетания клавиш «`Ctrl+Z`» (в ОС Windows) или «`Ctrl+D`» (в MacOS и GNU/Linux).

1.1 Открытие, создание, закрытие и удаление

В Unix-подобных системах по умолчанию открыты три дескриптора файла для любой выполняющейся программы: стандартный ввод (standard input), стандартный вывод (standard output) и стандартный поток ошибок (standard error). Они всегда имеют значения 0, 1 и 2 соответственно. По умолчанию вызов read для стандартного ввода приведет к чтению данных с клавиатуры. Аналогично запись в стандартный вывод или стандартный вывод диагностики приведет по умолчанию к выводу сообщения на экран терминала.

Во всех случаях, когда не используются дескрипторы стандартного ввода, вывода и ошибок, вы должны явно открывать файлы, чтобы затем читать из них или писать в них. Для этой цели существуют две функции: `open()` и `creat()`.

Функция `open()` пытается открыть указанный файл и возвращает дескриптор файла, который является целым числом типа `int`.

```
int fd;
fd=open(name,rwmode);
```

- Параметр `name` является символьной строкой, соответствующей имени файла.
- Параметр `rwmode` определяет режим доступа: 0 для чтения, 1 для записи, 2 для чтения и записи. Кроме того, предусмотрены специальные целочисленные константы, которые дают этим режимам символьные имена:
    - `O_RDONLY` – открыть файл только для чтения
    - `O_WRONLY` – открыть файл только для записи
    - `O_RDWR` – открыть файл для чтения и записи

ПРИМЕЧАНИЕ: на самом деле, режим может задаваться *комбинацией* из нескольких числовых констант, объединенных через логический оператор ИЛИ. Обязательной является любая из трех перечисленных констант, а дополнительные (уточняющие) константы могут добавляться к ним по мере необходимости. Например, комбинация `O_WRONLY | O_APPEND` формирует режим открытия для дозаписи в конец файла. Подробнее про дополнительные режимы можно узнать, вызвав команду `man 2 open`.

Если происходит какая-то ошибка, функция `open()` возвращает «-1»; в противном случае она возвращает неотрицательный дескриптор файла. Попытка открыть файл, который не существует, является ошибкой.

Функция `creat()` предоставляет возможность создания новых файлов или перезаписи старых. В результате обращения:

```
fd=creat(name,pmode);
```

возвращается дескриптор файла, если оказалось возможным создать файл с именем name, и «-1» в противном случае. Создание файла, который уже существует, **не является** ошибкой: `creat()` усечёт его до нулевой длины.

- Параметр `name` является символьной строкой, соответствующей имени файла.
- Если файл ранее не существовал, то `creat()` создает его с определенным режимом защиты, заданным аргументом `pmode`. В системе файлов Unix-подобных систем с файлом связываются девять битов защиты информации, которые управляют разрешением на чтение, запись и выполнение для владельца файла, для группы владельцев и для всех остальных пользователей. Таким образом, трехзначное восьмеричное число наиболее удобно для записи режима защиты. Например, число 0755 свидетельствует о разрешении на чтение, запись и выполнение для владельца и о разрешении на чтение и выполнение для группы и всех остальных.

Функция `close(fd)` прерывает связь между дескриптором файла `fd` и открытым файлом и освобождает дескриптор файла для использования с другим файлом. Завершение выполнения программы через `exit()` или в результате возврата из головной функции приводит к закрытию всех открытых файлов.

Функция удаления `unlink(filename)` удаляет из системы файл с именем filename (Точнее, удаляет имя filename, файл удаляется, если на него не остается ссылок под другими именам).

1.2 Произвольный доступ к файлу

Обычно при работе с файлами ввод и вывод осуществляется последовательно: при каждом обращении к функциям `read()` и `write()` чтение или запись начинаются с позиции, непосредственно следующей за предыдущей обработанной. Но при необходимости файл может читаться или записываться в любом произвольном порядке. Обращение к системе с помощью функции `lseek()` позволяет передвигаться по файлу, не производя фактического чтения или записи. В результате обращения

```
lseek(fd,offset,origin);
```

текущая позиция в файле с дескриптором `fd` передвигается на позицию offset (смещение), которая отсчитывается от места, указываемого аргументом `origin` (начало отсчета). Последующее чтение или запись будут теперь начинаться с этой позиции. Аргумент `offset` имеет тип `long`; `fd` и `origin` имеют тип `int`. Аргумент `origin` может принимать значения 0, 1 или 2, указывая на то, что величина `offset` должна отсчитываться соответственно от начала файла, от текущей позиции или от конца файла. Кроме того, предусмотрены три целочисленные константы, которые дают этим режимам символьные имена:

- `SEEK_SET` – отсчитывать смещение от начала файла
- `SEEK_CUR` – отсчитывать смещение от текущей позиции
- `SEEK_END` – отсчитывать смещение от конца файла

Например, чтобы дополнить файл, следует перед записью найти его конец:

```
lseek(fd,0l,2);
```

чтобы вернуться к началу, можно написать:

```
lseek(fd,0l,0);
```

Обратите внимание на аргумент 0l; его можно было бы записать и в виде `(long) 0`.

Функция `lseek()` позволяет обращаться с файлами примерно так же, как с большими массивами, только ценой более медленного доступа.

Пример. Функция, считывающая любое количество байтов, начиная с произвольного места в файле.

```
/*читать n байтов с позиции pos в buf */
int get(int fd, long pos,char *buf,int n)
{
    lseek(fd,pos,0);   /*переход на позицию pos */
    return (read(fd,buf,n));
}
```

Задание для самостоятельного выполнения

Выполните следующее задание, **сохраняя в отчет текст выполняемых команд и/или скриншоты с результатами их работы**:

- Изучите теоретический материал, приведенный выше
- Напишите программу, выполняющую следующие действия:
    - запуск 5 дочерних процессов, выводящих на экран произвольную строку приветствия (все 5 приветствий должны быть разными).
    - запись в текстовый файл строк, содержащих PID каждого дочернего процесса с текстом того приветствия, которое этот дочерний процесс должен вывести на экран (не перезаписывая прежнее содержимое файла, если оно было), текущее время (в формате **ISO 8601 YYYY-MM-DDThh:mm:ss±hh**) где **hh** - часовой пояс, **T** - символ-разделитель даты и времени.
- ****(задание дополнительной сложности)**** Дополнительно выводить на экран информацию о том, который по счёту раз процесс запускается (получать это число, подсчитывая количество приветствий данного процесса, уже содержащихся в файле).

Результат выполнения работы: компилируемый .c файл с текстом программы, PDF-файл с кратким описанием задания, текстом программы и демонстрацией её работы (скриншоты или скопированный текстовый вывод программы).