1 Интерфейс библиотечных функций и системных вызовов

Язык Си не содержит никакой поддержки операций ввода/вывода, параллелизма и пр. Такие (и многие другие) функции реализованы в стандартной библиотеке языка. Прототипы функций, типы данных и константы определяются в стандартных заголовочных файлах (например, <stdio.h>), а собственно тела (реализации) стандартных функций определяются в библиотечном файле, который подключается к программе на этапе компоновки. В современных системах библиотеки подключаются к программе не на этапе компоновки, а позднее, на этапе запуска программы, что позволяет уменьшить размер исполняемого модуля программы на диске, и, что более важно, суммарный размер памяти, занимаемый программами при выполнении за, счёт того, что страницы кода стандартной библиотеки разделяются между всеми процессами.

Часть стандартных функций может быть реализована непосредственно в самом процессе без обращения к ядру операционной системы, например, функции работы со строками. Некоторые функции выполняют существенный объём работы в пользовательском режиме, но для выполнения какой-то части работы обращаются к системе. Задача некоторых функций состоит в том, чтобы подготовить параметры так, как этого требует интерфейс с ядром операционной системы, и передать управление ядру. Такие функции называются *системными вызовами*, хотя, строго говоря, это только функции-оболочки системных вызовов. Граница между библиотечными функциями и системными вызовами проходит по разному на разных системах. Например, на системах BSD операции работы с сокетами реализованы как системные вызовы, а на некоторых системах System V они же реализованы как библиотечные функции.

Не всегда библиотечная функция или системный вызов завершаются успешно. Например, операция открытия файла может закончится неудачно по разным причинам, например файл не найден, или недостаточно прав доступа. В этом случае функция возвращает специальное значение, сигнализирующее об ошибке, а в глобальную переменную `errno` записывается код причины ошибки, например, `EACCES` — недостаточно прав доступа. Чтобы получить доступ к переменной `errno` и константам кодов ошибок программа должна подключить заголовочный файл <errno.h>.

Чаще всего программе требуется напечатать пользователю некоторое осмысленное сообщение об ошибке. Например, в случае ошибки `EACCES` можно напечатать сообщение `"Permission denied"`. Для этого стандартная библиотека содержит специальные функции.

```
#include <string.h>
char *strerror(int errnum);
```

Функция `strerror` преобразовывает код ошибки, переданный в параметре `errnum` в текстовую строку-описание ошибки, указатель на которую возвращается. Эта текстовая строка находится в статической памяти библиотеки и ни в коем случае не должна модифицироваться.

```
#include <stdio.h>
void perror(char const *str);
```

Функция `perror` печатает сообщение о последней ошибке (текущее значение переменной `errno`) на стандартный поток ошибок. Если параметр `str` не равен `NULL`, перед описанием ошибки будет напечатана строка `str` и символ двоеточия (`': '`).

2 Работа с файлами

Понятие «файл» — одно из основных понятий операционных систем Unix. Понятие «файл» здесь шире, чем в других операционных системах, а именно, файлом может представляться любая сущность, к которой применимы понятия «открытия», «чтения», «записи». Например, последовательность байт, хранящаяся в некоторых секторах диска и имеющая имя, — это обычный (так называемый «регулярный») файл. Но и сам диск тоже является с точки зрения Unix файлом (специальное блочное устройство). Последовательный порт компьютера тоже является файлом (специальное символьное устройство).

Для хранения файлов используется файловая система, причём почти все файлы (кроме анонимных каналов и интернет-сокетов) имеют некоторое соответствие в файловой системе. Для регулярных файлов файловая система хранит имена файлов, служебную информацию о файле и сами блоки данных, а для прочих типов файлов — только имена и служебную информацию. Файловая система имеет иерархическую структуру: корень файловой системы может содержать файлы и каталоги, каждый каталог в свою очередь может содержать другие файлы и каталоги и так далее. Две (и более) файловые системы могут быть объединены в одну файловую систему с помощью монтирования. При монтировании файловой системы корневой каталог одной файловой системы подключается вместо некоторого каталога другой файловой системы, так что в результате получается одна файловая система. Все файловые системы, необходимые для работы операционной системы монтируются автоматически при начальной загрузке системы. Обычный пользователь чаще всего не имеет права самостоятельно монтировать файловые системы (из-за соображений безопасности).

В «родных» для Unix файловых системах (например, оригинальной файловой системе SysV, файловой системе BSD, файловой системе Linux) информация о файле разбита на две части: запись в каталоге и индексный дескриптор. Индексный дескриптор (inode) хранит всю служебную информацию о файле: идентификаторы владельца и группы, права доступа, размер, блоки диска, занимаемые файлом и пр. Все индексные дескрипторы в файловой системе занумерованы, а максимальное количество индексных дескрипторов в данной файловой системе задаётся при её создании и не может быть потом изменено. Нумерация индексных дескрипторов уникальна для каждой файловой системы без учёта монтирования, то есть в работающей системе может существовать несколько файлов с одним и тем же номером индексного дескриптора.

Индексный дескриптор не хранит имя файла, которое хранится в каталогах. Кроме имени файла запись в каталоге ещё содержит номер соответствующего индексного дескриптора. Таким образом, по имени файла сначала получается номер его индексного дескриптора, а затем по информации из индексного дескриптора можно считывать данные из файла. Два (или более) имени могут ссылаться на один и тот же индексный дескриптор. В этом случае говорят, что два имени являются «жёсткими» ссылками на один и тот же файл. Жёсткие ссылки не отличимы друг от друга. Для нормальной работы в этой ситуации индексный дескриптор содержит ещё счётчик ссылок на себя. Файл считается уничтоженным, и блоки данных помечаются доступными для повторного использования, только тогда, когда счётчик ссылок на индексный дескриптор становится равным нулю. Счётчик ссылок также увеличивается на 1 при каждом новом открытии файла, а уменьшается на 1 при закрытии файла. Таким образом, может существовать файл, который вообще не имеет имени. Для этого процесс должен создать файл, открыть его, а затем сразу удалить его. Запись о файле будет из каталога удалена, но сам файл продолжит существовать до тех пор, пока счётчик ссылок на его индексный дескриптор не станет равным нулю, то есть до тех пор, пока соответствующий файловый дескриптор не закроет последний процесс, который его, возможно, унаследовал от процесса,

создавшего такой файл.

Во внутренних структурах ядра полный путь к файлу никогда не используется. Он необходим только на этапе открытия файла. После этого достаточно пары ⟨ номер устройства, номер индексного дескриптора ⟩. По открытому файлу не существует простого способа узнать, какое имя было у открытого файла. Чтобы всё-таки узнать имя файла (точнее, одно из имён), нужно получить номер устройства и номер индексного дескриптора (вызовом `fstat`), а затем найти в файловой системе запись с теми же самыми характеристиками.

Ядро может накладывать ограничения на максимальную длину пути. Если в системный вызов передаётся абсолютный или относительный путь, длина строки которого больше этого ограничения, системный вызов завершится с ошибкой `ENAMETOOLONG`. Эта константа называется `PATH_MAX` и определяется в заголовочном файле `<limits.h>`.

Современные Unix-подобные системы поддерживают работу с самыми разнообразными файловыми системами, например VFAT, NTFS, NFS. Чтобы предоставить одинаковый интерфейс доступа к файлам, ядру приходится эмулировать индексные дескрипторы. Не всегда это можно сделать полностью корректно (особенно много исключений существует для сетевой файловой системы NFS), и в этих случаях возможны тонкие отличия в работе с файловой системой.

Каждый тип файловой системы имеет свою структуру каталога. Поэтому некоторые операционные системы (например, Linux) вообще запрещают прямое чтение каталога с помощью системного вызова `read`, а требуют пользоваться специальным системным вызовом (`getdents`).

С точки зрения операционной системы каждый файл является потоком байтов. Никакого дополнительного структурирования не производится. Некоторые файлы могут допускать позиционирование на любое место в файле, другие файлы допускают только последовательное чтение из файла или запись в файл.

Процесс может работать с файлом посредством файлового дескриптора. Файловый дескриптор — это небольшое неотрицательное целое число, которое уникально в пределах данного процесса. Три первых файловых дескриптора имеют специальное назначение. Файловый дескриптор с номером 0 — это стандартный ввод процесса, 1 — стандартный вывод процесса, 2 — стандартный поток ошибок. Многие программы считывают данные со стандартного ввода, выводят результат на стандартный вывод, а сообщения об ошибках печатают на стандартный поток ошибок. Стандартный ввод обычно связан с входным буфером терминала, а стандартный вывод и стандартный дескриптор ошибок — с выходным буфером терминала. Это стандартное соглашение о назначении файловых дескрипторов, тем не менее, может произвольным образом нарушаться. Например, все стандартные дескрипторы могут быть вообще закрыты.

Файловые дескрипторы всегда наследуются при создании нового процесса (системный вызов `fork`) и, как правило, наследуются при запуске новой программы с помощью системного вызова `execve`. Программист может запретить наследование файлового дескриптора при вызове `execve`, пометив его флагом `FD_CLOEXEC`. Для этого используется системный вызов `fcntl`. Под наследованием в данном случае понимается то, что новому процессу (или вновь запущенной программе) будет доступен файловый дескриптор с тем же самым номером, ассоциированный с тем же самым файлом и разделяющий позицию чтения и записи в файле.

2.1 Открытие файла

Для открытия нового файла используется системный вызов open. Обратите внимание, что он может использоваться как с двумя, так и с тремя аргументами. Третий аргумент обязательно должен быть указан, если во флагах присутствует бит O_CREAT.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Первый параметр задаёт путь к файлу, который нужно открыть. Второй параметр определяет режим, в котором будет открыт файл, а третий параметр задаёт права доступа в случае, когда файл создаётся. Файл может быть открыт в одном из трёх режимов: только чтение (O_RDONLY), только запись (O_WRONLY) и чтение и запись (O_RDWR). Попытка выполнить операцию, которая не предусмотрена режимом открытия файла, вызовет ошибку соответствующего системного вызова. Кроме режима работы с файлом в параметре flags можно указать флаг создания файла O_CREAT, который вызовет создание нового пустого файла, если файл с таким именем ещё не существует. Флаг O_EXCL, который может использоваться только вместе с O_CREAT, задаёт, что системный вызов должен завершиться с ошибкой с кодом EEXIST, если такой файл уже существует. Этот флаг может применяться для простейшей синхронизации процессов, как будет описано в дальнейшем. Флаг O_TRUNC задаёт, что старой содержимое файла (если оно было) должно быть уничтожено. Флаг O_APPEND задаёт, что перед каждой записью в файл указатель должен позиционироваться на конец файла.

Флаг O_NONBLOCK (другое название — O_NDELAY) указывает, что операции с файловым дескриптором никогда не должны приводить к блокировке выполнения процесса. Например, если файловый дескриптор ассоциирован с терминалом, и пользователь не ввёл ни одного символа, то системный вызов read при использовании неблокирующего режима чтения завершится с ошибкой EAGAIN.

Когда файл открывается на сетевой файловой системе NFS, не все флаги открытия могут корректно работать. Например, использование флага O_EXCL может не дать ожидаемого эффекта, то есть файловый дескриптор может в некоторых ситуациях быть получен, даже если такой файл уже создан. Это связано с особенностями протокола NFS[1]. Точно также на файловой системе NFS невозможно корректно реализовать флаг добавления O_APPEND, поэтому, если одновременно несколько процессов записывают в один файл, это может привести к искажению содержимого файла. Конечно же, эти проблемы не проявляют себя в обычной практике, когда файл открывает и с ним работает единственный процесс, находящийся на единственном компьютере.

При создании нового файла, то есть когда указывается флаг O_CREAT, необходимо задать права доступа к создаваемому файлу — параметр mode. В этом параметре используются только 12 младших бит, определяющие права доступа для владельца, группы, прочих пользователей и специальные флаги (например, флаг suid). Права с которыми в действительности будет создан файл ещё зависят от маски создания файлов процесса umask. Если perms — это права доступа к создаваемому файлу, это значение может быть вычислено по формуле perms = mode & ~umask. Другими словами, у параметра mode сбрасываются

[1] Протокол NFS специально проектировался как протокол без состояний, чтобы авария сервера не сказывалась на работе клиентов. Как показала практика, это оказалось ошибочным решением.

те биты, которые установлены в маске создания файлов процесса.

Системный вызов `open` возвращает номер файлового дескриптора открытого файла, причём всегда выбирается наименьший незанятый номер файлового дескриптора. Если открытие файла невозможно, системный вызов возвращает число `-1`, а переменная `errno` будет содержать код ошибки. Возможные коды ошибок для `open` можно посмотреть в приложении.

Когда файл открывается на запись, наиболее часто используется комбинация режимов открытия `O_WRONLY | O_CREAT | O_TRUNC`, означающая, что файл открывается в режиме «только запись», если файл не существовал, он будет создан, а если он существовал, его старое содержимое будет уничтожено. Чтобы в таких случаях не задавать эти режимы открытия, предусмотрена функция `creat`, которая в точности эквивалентна системному вызову `open` с указанными выше режимами.

2.2 Закрытие файла

Когда процесс завершает работу (в результате вызова `_exit`, `exit` или при получении сигнала) все открытые файловые дескрипторы закрываются автоматически. Тем не менее, файловые дескрипторы, которые больше не используются, необходимо закрывать. Каждый процесс имеет ограничение на количество одновременно открытых файловых дескрипторов (это ограничение можно узнать с помощью команд `ulimit -a` или `ulimit -n`). Если процесс исчерпал ресурс свободных файловых дескрипторов, открытие файла завершится с ошибкой `EMFILE`. Кроме того, незакрытые файловые дескрипторы могут привести к тому, что созданные новые процессы будут работать неправильно, например, никогда не завершатся.

Для закрытия открытого файлового дескриптора используется системный вызов `close`.

```
#include <unistd.h>
int close(int fd);
```

Единственным параметром передаётся номер закрываемого файлового дескриптора. Функция возвращает 0 при нормальном завершении и `-1` при ошибке. Если файл был открыт на запись, система может использовать внутреннюю буферизацию для увеличения эффективности работы с внешними устройствами. В этом случае при вызове `close` система попытается сохранить буферизованные данные на внешнем устройстве, и, если при этом возникнет ошибка ввода/вывода, она будет сообщена пользователю. Поэтому игнорирование кода возврата функции `close` может привести к незамеченной потере данных, особенно когда используется сетевая файловая система NFS в сочетании с квотированием диска.

Системным вызовом `close` может закрываться файловый дескриптор, полученный из любого источника: системных вызовов `open`, `socket`, `dup`.

2.3 Работа с файловым дескриптором

Рассмотрим системные вызовы, позволяющие работать с файлом, ассоциированным с файловым дескриптором: считывать и записывать данные, позиционировать указатель текущей позиции.

2.3.1 Чтение

Для чтения данных по файловому дескриптору используется системный вызов `read`.

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Здесь тип size_t — это тип, который используется в библиотеке языка Си для размера объектов. На 16- и 32-битных машинах этот тип, как правило, эквивалентен unsigned long int. Тип ssize_t — «размер со знаком». На 16- и 32-битных машинах это, как правило, long int.

Параметр fd задаёт номер файлового дескриптора, из которого должно производится чтение. Файловый дескриптор должен быть открыт и должен допускать чтение, в противном случае read завершится с ошибкой. Параметр buf задаёт начальный адрес памяти, по которому должны быть размещены считанные данные, а параметр count — максимальный размер считываемых данных (в байтах). Если блок памяти, заданный параметрами buf и count, полностью или частично находится вне адресов памяти, доступных процессу для записи, read завершится с ошибкой EFAULT [2]. Системный вызов read считывает не более чем count байт, и это число возвращается в качестве результата. Если в файловом дескрипторе выполняется условие «конец файла», системный вызов read возвращает 0. При ошибке возвращается -1, а переменная errno устанавливается в код ошибки. Когда данные, немедленно доступные для чтения, отсутствуют, процесс, как правило, погружается системой в сон до тех пор, пока данные не появятся.

Детали работы с файловым дескриптором: когда считывается count, а когда меньше чем count байт, когда наступает «конец файла» — зависят от того, с чем ассоциирован файловый дескриптор. Если файловый дескриптор ассоциирован с регулярным файлом, то есть с файлом, блоки данных которого действительно размещаются на диске, системный вызов read ведёт себя следующим образом: считывается count байт, если размер непрочитанной части файла больше count, считывается столько байт, каков размер непрочитанной части байт, если она меньше count, наконец, возвращается 0, когда непрочитанных данных не осталось.

2.3.2 Запись

Системный вызов write позволяет записать данные в файл.

```
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t count);
```

Параметр fd задаёт номер файлового дескриптора. Файловый дескриптор должен быть открыт и должен допускать запись, в противном случае write завершится ошибочно. Параметр buf задаёт начальный адрес памяти, по которому размещаются записываемые данные, а параметр count — размер записываемых данных (в байтах). Если блок памяти, заданный параметрами buf и count, полностью или частично находится вне адресов памяти, доступных процессу для чтения, write завершится с ошибкой EFAULT.

write пытается записать в файл заданное количество байт и возвращает, сколько байт в действительности было записано. При ошибке возвращается -1, а переменная errno устанавливается в код ошибки (возможные коды ошибок приведены в приложении). Может быть записано меньше байт, чем запрошено, и это не является ошибкой. Такая ситуация может возникнуть, например, при работе с сетевой файловой системой NFS. В этом случае нужно ещё раз вызвать write с изменёнными указателем на начало области записываемых данных и размером области.

[2] Обратите внимание, что запись по таким адресам самим процессом приведёт к краху программы по сигналу SIGSEGV или SIGBUS.

2.3.3 Операции с файлами в многопроцессной системе

Когда в системе работает более одного процесса, несколько процессов могут независимо открыть один и тот же файл, причём в разных режимах. Например, два процесса могут открыть один и тот же файл на запись. В отличие от других систем, одновременная работа с файлом нескольких процессов по умолчанию не запрещена. Для регулярных файлов система поддерживает атомарность операций чтения и записи. Под атомарностью некоторой операции здесь понимается то, что никакой процесс в системе не сможет наблюдать ситуацию, когда операция уже началась, но ещё не прошла до конца. Если в результате вызова операции чтения или записи процесс был помещён в состояние ожидания поступления данных или освобождения буфера, операция не считается начавшейся.

Обычно права доступа к файлу проверяются один раз: при открытии файла. Если пользователь не имеет достаточно прав, чтобы открыть файл в запрошенном режиме, системный вызов `open` вернёт соответствующий код ошибки. Изменение прав доступа к файлу никак не повлияет на файловые дескрипторы, которые уже ассоциированы с этим файлом. Например, если файл открывался в режиме записи, а после разрешение записи для данного пользователя было отменено, все операции записи в файл, производимые с использованием этого файлового дескриптора, будут завершаться успешно.

Работа с файлами на сетевой файловой системе NFS и здесь имеет свои особенности. Поскольку NFS-сервер не поддерживает список открытых файлов для клиента, а клиент каждый раз присылает запросы на чтение или запись с указанием имени файла (точнее, номера устройства и номера индексного дескриптора), смещения от начала файла и размера файла, права доступа к файлу проверяются сервером при каждой операции. Поэтому, если после открытия файла на клиенте, права доступа были изменены, последующие операции с файловым дескриптором будут завершаться ошибочно, если новые права доступа к файлу не позволяют выполнить запрашиваемую операцию.

Другая особенность NFS связана с удалением файлов. В UNIX файл не считается удалённым, даже если удалены все записи в каталогах, ссылающиеся на его индексный дескриптор, до тех пор, пока не будет закрыт последний файловый дескриптор, ассоциированный с этим файлом. Так как NFS-сервер не имеет информации об открытых на клиенте файлах, он не может удалять файлы, с которыми, возможно, в данный момент идёт работа у клиента. Поэтому при запросе на удаление файл, в отношении которого сервер имеет подозрение, что его может использовать какой-либо клиент, не удаляется, а переименовывается в имя вида `.nfs032847`. Такие файлы удаляются либо когда сервер будет точно знать, что никакой клиент его больше не использует, либо при перезагрузке системы.

2.3.4 Позиционирование

Когда файловый дескриптор ассоциирован с регулярным файлом, осмыслено понятие «текущее положение в файле». Это смещение от начала файла, начиная с которого очередной вызов `read` будет считывать данные, а вызов `write` записывать данные (если только файл не был открыт с режимом `O_APPEND`). После операции чтения или записи указатель текущего положения в файле продвигается вперёд на количество байт, прочитанное из файла или записанное в файл.

Все клоны некоторого файлового дескриптора (то есть полученные из него явно с помощью системных вызовов `dup`, `dup2` или неявно при создании нового процесса с помощью `fork`) разделяют один указатель на текущее положение в файле. Любая операция, выполненная на одном из таких файловых дескрипторов, которая изменяет текущее положение в

файле, делает это для всех клонов. Поэтому, когда некоторый файловый дескриптор, ассоциированный с регулярным файлом, используется несколькими процессами, необходимо следить за тем, что указатель текущего положения в файле установлен, как требуется.

Указатель текущего положения в файле можно произвольным образом изменять с помощью системного вызова `lseek`.

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

Параметр `fildes` задаёт номер файлового дескриптора, параметр `whence` задаёт, от какой точки должно отсчитываться смещение, а параметр `offset` задаёт само смещение. Параметр `whence` может принимать одно из следующих значений:

`SEEK_SET`	0	`offset` отсчитывается от начала файла.
`SEEK_CUR`	1	`offset` отсчитывается от текущего положения.
`SEEK_END`	2	`offset` отсчитывается от текущего размера файла.

При успешном завершении `lseek` возвращает новое положение указателя относительно начала файла, измеренное в байтах. При ошибке возвращается значение `(off_t) -1`, и переменная `errno` устанавливается в код ошибки.

Чтобы узнать текущее положение в файле, нужно вызвать системный вызов `lseek` с параметрами `lseek(fd, 0, SEEK_CUR);`.

2.4 Файловые дескрипторы и дескрипторы потоков

Стандартная библиотека языка Си определяет большое количество функций для работы с файлами, например, `fopen`, `fprintf`, `fscanf`, `fclose`. Эти функции предоставляют широкие возможности по управлению форматом записываемых или считываемых данных: построчный ввод или вывод, преобразование стандартных типов в строковое представление и обратно и т. д. Поскольку они позволяют вводить и выводить структурированные данные, то данные на более высоком уровне абстракции, чем последовательность байтов, эти функции называются ещё «высокоуровневыми» функциями ввода/вывода, в противоположность «низкоуровневым» функциям, описанным выше.

Существенная часть работы высокоуровневых функций ввода вывода (например, форматирование чисел в соответствии со спецификацией формата) производится в стандартной библиотеке языка Си в функциях, которые работают в контексте процесса. Для выполнения собственно операции чтения из файла или записи в файл эти функции должны обратиться к ядру операционной системы. Поскольку ядро может выполнять только системные вызовы низкого уровня, высокоуровневый ввод/вывод использует (реализован с помощью) низкоуровневые операции работы с файлами. Так, функция `fopen` после анализа режимов открытия и создания внутренних структур данных (указатель на которые она возвращает) сделает системный вызов `open`. Функции записи в дескриптор потока вызовут в итоге системный вызов `write` и т. д.

Высокоуровневый ввод/вывод по умолчанию буферизуется. Это значит, что в адресном пространстве процесса выделяется память под некоторый буфер фиксированного размера (по умолчанию обычно 512 байт), записываемые данные попадают сначала в буфер, и лишь потом передаются ядру, считываемые данные сначала считываются в буфер и только затем передаются в программу. Дескрипторы потока, связанные с устройствами типа терминалов, буферизуются построчно, то есть выводимые данные не будут переданы ядру до тех пор, пока программа не выведет символ перевода строки `'\n'`. Исключение составляет стандартные

поток ошибок stderr, который никогда не буферизуется. Буфер потока является полной собственностью процесса, и ядро ничего не знает о такой буферизации. Поэтому, когда ядро принудительно завершает процесс (например, при получении процессом необрабатываемого сигнала, который по умолчанию завершает работу процесса), всё содержимое буферов потоков, предназначенных на запись, теряется. Когда ядро создаёт копию адресного пространства процесса (при вызове fork), копируется и содержимое буферов, и состояние дескрипторов потока.

Чтобы изменить режим буферизации данного дескриптора потока используется функция setvbuf.

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Параметр stream задаёт дескриптор потока, параметр mode задаёт новый режим буферизации. Поддерживаются три режима:

- Режим без буферизации (параметр mode должен быть равен константе _IONBF). В этом случае значения параметров buf и size игнорируются.
- Режим с построчной буферизацией (параметр mode равен _IOLBF). Параметр buf должен указывать на начало буфера (области памяти), который имеет размер по крайней мере size байт. Если параметр buf равен NULL, буфер заданного размера будет выделен с помощью malloc при следующей операции ввода/вывода.
- Режим с полной буферизацией (параметр mode равен _IOFBF). Параметр buf должен указывать на начало буфера (области памяти), который имеет размер по крайней мере size байт. Если параметр buf равен NULL, буфер заданного размера будет выделен с помощью malloc при следующей операции ввода/вывода.

Изменение режима будет иметь эффект, только если поток не активен, то есть его буфер пуст, что бывает либо после открытия, но до выполнения операций ввода/вывода, либо после вызова функции fflush.

Если адрес буфера задаётся в функции setvbuf явно, буфер должен существовать, когда поток будет закрываться. Если поток не закрывается явно в самой программе, буфер должен существовать и тогда, когда закончит своё выполнение функция main, то есть буфер не должен быть локальной переменной функции main.

Использование режимов без буферизации и режима полной буферизации может давать нежелательный результат, когда несколько процессов одновременно модифицируют (например, дописывают) один и тот же текстовый файл. Если поток переведён в режим без буферизации, использование fprintf одновременно несколькими процессами может привести к тому, что вывод процессов будет произвольным образом перемешан даже в пределах одной выводимой строки. Хотя каждая операция write сама по себе атомарна, при обработке форматной строки write может быть вызвана несколько раз. Если же включён режим полной буферизации, вывод будет производиться блоками по (по умолчанию) 512 байт. При этом граница выводимого блока может не совпадать с границей выводимой строки. Безопаснее всего использовать режим с построчной буферизацией, либо (что то же самое) сначала подготавливать строку в памяти с помощью функции sprintf, затем её выводить целиком с помощью fputs или fwrite.

Если файл открыт в режиме чтение и запись несколькими процессами (такие файлы чаще всего содержат бинарные данные) и активно модифицируется, система не гарантирует, что буфер в памяти будет отражать последние изменения в файле. Перед модификацией содержимого файла необходимо сделать пустую операцию над потоком, например `fseek(f, 0, SEEK_CUR);`, а после модификации необходимо сохранить буфер вызовом `fflush`. Кроме того, необходимо использовать средства предотвращения одновременного доступа к файлу, которые будут рассмотрены в следующих разделах.

Дескриптор потока помимо информации о буферизации хранит номер низкоуровневого файлового дескриптора, с помощью которого и осуществляются все операции ввода/вывода. Номер файлового дескриптора можно получить с помощью функции `fileno`.

```
#include <stdio.h>
int fileno(FILE *stream);
```

Функция `fileno` всегда завершается успешно (если, конечно, передан правильный дескриптор потока) и не модифицирует переменную `errno`.

Функция `fdopen` позволяет создать дескриптор потока по уже открытому низкоуровневому файловому дескриптору.

```
#include <stdio.h>
FILE *fdopen(int fildes, const char *mode);
```

Здесь параметр `fildes` задаёт номер низкоуровневого файлового дескриптора, а параметр `mode` — режимы открытия, точно такие же, как у функции `fopen`. Режим, в котором был открыт файловый дескриптор `fildes` не должен противоречить режиму открытия, заданному в параметре `mode`. Указатель текущего положения в открывающемся потоке устанавливается в соответствие с указателем текущего положения файлового дескриптора. Если режим открытия указан как `"w"` или `"w+"`, содержимое файла не будет очищено, в отличие от функции `fopen`.

Дескриптор потока будет использовать файловый дескриптор с указанным номером, а не его копию, полученную с помощью системного вызова `dup`. Поэтому, после того, как файловый дескриптор был трансформирован в дескриптор потока, закрываться он должен как высокоуровневый поток, то есть с помощью функции `fclose`.

При успешном завершении функция `fdopen` возвращает указатель на новый дескриптор потока, а при неудаче возвращается `NULL`, и переменная `errno` устанавливается в код ошибки (например, неверный файловый дескриптор).

2.5 Временные файлы

Часто программе в процессе работы требуется вспомогательный файл. Такой файл создаётся программой, программой же используется и удаляется, а пользователь может ничего не знать об использовании дополнительных файлов. Такие файлы называются *вре́менными*, так как время их жизни не больше времени работы программы.

Первый вопрос, который возникает, когда нужно создать временный файл: *в каком каталоге файловой системы он должен быть создан?* Кажется, что можно создавать его в текущем каталоге, но это неверно. Во-первых, текущий каталог может быть закрыт на запись для пользователя, запустившего программу. Попытка создать временный файл в этом случае закончится неудачей. Во-вторых, если каталог открыт на запись, размер файлов и их количество могут быть ограничены (так называемая дисковая квота), и создание и работа с временными файлами может ещё больше ограничить возможности пользователя. В-третьих,

если каталог располагается на сетевой файловой системе NFS, операции с ним могут оказаться достаточно медленными.

Традиционно в системах UNIX для временных файлов использовался каталог `/tmp` (в системе Solaris ещё `/var/tmp`). Этот каталог открыт на запись и чтение для всех пользователей, как правило, не квотируется и, как правило, располагается на локальной файловой системе. Для временных файлов может использоваться другой тип файловой системы, чтобы повысить эффективность работы с файлами. Кроме того, область временных файлов может разделятся с областью подкачки системы. Система либо очищает каталог временных файлов при каждой перезагрузке, либо периодически удаляет из него все файлы, к которым не было доступа в течение определённого времени (на RedHat Linux две недели).

Общесистемный разделяемый каталог, такой как `/tmp`, является узким местом с точки зрения обеспечения безопасности системы. Если программа работает с ним неаккуратно, злоумышленник может, в нужные моменты времени создавая файлы в таком каталоге, испортить содержимое файлов пользователя, запустившего программу или даже записать в нужные злоумышленнику файлы (например, `/etc/passwd`) нужные ему данные (при условии, что программа запущена суперпользователем `root`). Поэтому использование каталогов типа `/tmp` нежелательно. Чтобы указать путь к каталогу, в котором следует создавать временные файлы, пользователь может установить переменную окружения `TMPDIR`, и программе следует использовать эту переменную, если она установлена. Если переменная окружения не установлена, программе следует использовать константу `P_tmpdir`, определённую в заголовочном файле `<stdio.h>`. Эта константа (макрос) определяет путь к общесистемному каталогу временных файлов. И только в крайнем случае программа должна использовать явный путь `/tmp`.

Второй вопрос: *как должен называться временный файл?* Фиксированное название (например, `myfile`) плохо. Во-первых, если одновременно будут работать два процесса, запущенные одним или разными пользователями, то либо два процесса будут одновременно использовать один и тот же файл, что приведёт к порче данных, либо работать будет только один процесс. Поэтому имя временного файла должно содержать некоторую непостоянную (случайную) компоненту, причём пространство имён, из которого выбирается эта компонента должно быть достаточно большим, чтобы исчерпание пространства имён было маловероятным. Например, случайное число в интервале от 0 до `RAND_MAX`, преобразованное в строковое представление, даёт хорошую случайную компоненту, при условии, что затравка генератора псевдослучайных чисел меняется при каждом запуске программы (например, зависит от времени).

Поскольку, как было сказано выше, в каталог временных файлов имеют право записывать все пользователи, в том числе потенциально враждебные, необходимо выполнять особенные процедуры при создании временных файлов. После того, как случайное имя временного файла сгенерировано, файл должен создаваться с помощью системного вызова `open` с режимом создания `O_EXCL`. Это гарантирует то, что файл будет успешно создан, если файл с таким именем ещё не существовал. Если системный вызов `open` завершился с ошибкой `EEXIST` (файл уже существует), нужно сгенерировать новое имя файла и повторить попытку, и так до тех пор, пока файл наконец не будет создан, либо не будет исчерпано пространство случайных имён. Права доступа у временного файла должны быть равны `0600`, то есть доступ всех пользователей, кроме владельца файла закрыт.

Рассмотрим сценарии, показывающие необходимость таких предосторожностей. Предположим, что временный файл не открывается с флагом `O_EXCL`. Тогда злоумышленник может узнать, какое имя будет иметь временный файл (например, подсмотрев текст программы). После этого он создаёт в каталоге временных файлов (допустим, это `/tmp`) символи-

ческую связь с этим именем, которая указывает на некоторый файл пользователя, который злоумышленник хочет испортить или модифицировать. В качестве таких файлов он может выбрать `.rhosts` или `.profile`, или просто любой файл, принадлежащий пользователю, который злоумышленник хочет испортить. Когда программа, неосторожно работающая с временными файлами, будет запущена, она откроет временный файл, не заметив, что этот файл уже существует. Символическая связь будет прослежена, и будет открыт и очищен некоторый файл пользователя, содержащий, возможно, важные данные.

Если временный файл создаётся с большими правами доступа, чем `0600`, злоумышленник может подсмотреть (или даже модифицировать) содержимое временного файла, что, возможно, нежелательно. Явное изменение прав доступа уже после открытия файла (с помощью системного вызова `chmod`) оставляет злоумышленнику возможность открыть файл в тот момент, когда `open` уже создал файл, а `chmod` ещё не был вызван (такая ошибка называется “race condition”).

Наконец, когда программа завершает работу, все ею созданные временные файлы должны быть удалены, чтобы они не засоряли каталог разделяемых файлов. Если программа не планирует несколько раз открывать и закрывать этот временный файл, его можно удалить сразу после открытия. Тогда файл будет существовать, но не иметь имени, а все блоки данных будут освобождены, как только будет закрыт последний файловый дескриптор, ассоциированный с этим файлом.

Часть функций по созданию временного файла принимает на себя функция `mkstemp`.

```
#include <stdlib.h>
int mkstemp(char *template);
```

Функция `mkstemp` создаёт временный файл по шаблону имени, заданному аргументом `template`. Последние 6 символов шаблона должны быть `XXXXXX`, и эти символы заменяются на символы, делающие всё имя временного файла уникальным. Файл создаётся с флагом `O_EXCL` в режиме чтение/запись (`O_RDWR`) и с правами доступа `0600`. Поскольку строка `template` модифицируется, это не должна быть строковая константа (строковый литерал).

Функция возвращает файловый дескриптор временного файла при успешном завершении и `-1` при ошибке. Переменная `errno` в этом случае может принимать следующие значения: `EINVAL` — последние 6 символов шаблона не равны `XXXXXX`. В этом случае строка `template` не меняется. `EEXIST` — исчерпано пространство случайных имён временных файлов (то есть, все возможные файлы уже существуют). В этом случае значение строки `template` неопределено. Кроме того, могут возвращаться ошибки системного вызова `open`.

Следующая программа иллюстрирует работу со временными файлами.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <unistd.h>

/* если P_tmpdir неопределён, установим его в /tmp */
#ifndef P_tmpdir
#define P_tmpdir "/tmp"
#endif

int main(void)
{
  char tmpn[PATH_MAX];
```

```
  char *s;
  int  fd;
  FILE *f;
  int  a, b, c;

  /* если переменная окружения TMPDIR установлена,
   * используем её, иначе - P_tmpdir */
  if (!(s = getenv("TMPDIR"))) s = P_tmpdir;
  /* формируем шаблон */
  snprintf(tmpn, PATH_MAX, "%s/progXXXXXX", s);
  /* создаём временный файл */
  if ((fd = mkstemp(tmpn)) < 0) {
    perror("mkstemp");
    return 1;
  }
  /* удаляем его */
  unlink(tmpn);
  /* формируем дескриптор потока */
  if (!(f = fdopen(fd, "r+"))) {
    perror("fdopen");
    return 1;
  }
  /* запишем в него что-нибудь */
  fprintf(f, "%d %d %d\n", 1, 1, 2001);
  /* ... */

  /* f нельзя закрывать!
   * устанавливаемся на начало файла для чтения */
  fseek(f, 0, SEEK_SET);
  /* читаем данные */
  if (fscanf(f, "%d%d%d", &a, &b, &c) != 3) {
    fprintf(stderr, "temporary file error\n");
    return 1;
  }
  printf("Read values: %d %d %d\n", a, b, c);
  /* временный файл больше не нужен, закрываем его */
  fclose(f);
  return 0;
}
```

3 Работа с файловой системой

Типичные задачи, которые возникают при работе с файловой системой, — это просмотр содержимого каталогов, рекурсивный обход всех каталогов файловой системы, получение информации о состоянии файла.

Поскольку операционная система, как правило, поддерживает большое количество типов файловых систем, которые сильно отличаются друг от друга по используемому способу хранения данных, ядро операционной системы должно предоставлять некоторый обобщённый интерфейс для доступа к содержимому каталогов. Этот интерфейс базируется на пред-

положении, что каталог хранит только имена содержащихся в нём файлов. Вся остальная информация хранится в индексном дескрипторе. Если конкретная файловая система не имеет индексных дескрипторов (например, FAT), ядро всё равно будет эмулировать их наличие.

Для работы с каталогом используются библиотечные функции `opendir`, `closedir`, `readdir`, `seekdir`, `telldir`.

```
#include <sys/types.h>
#include <dirent.h>
DIR           *opendir(const char *name);
struct dirent *readdir(DIR *dir);
off_t          telldir(DIR *dir);
void           seekdir(DIR *dir, off_t offset);
int            closedir(DIR *dir);
```

Функция `opendir` ассоциирует с именем каталога, переданным ей в качестве аргумента, дескриптор каталога, указатель на который возвращается из функции для дальнейшего использования во всех функциях работы с каталогом. Дескриптор каталога имеет тип `DIR`, а работа с ним аналогична работе с дескриптором потока: в обоих случаях всегда используется указатель на структуру. В случае ошибки функция `opendir` возвращает `NULL`. Каждый открытый дескриптор каталога использует один файловый дескриптор, и, поскольку максимальное число одновременно открытых файловых дескрипторов в процессе ограничено, это нужно учитывать при рекурсивном обходе дерева файловой системы.

Функция `closedir` закрывает дескриптор каталога, передаваемый ей в качестве аргумента. При успешном завершении функция возвращает 0, а при ошибке — `-1`.

Функция `telldir` возвращает текущую позицию в каталоге. Следующий вызов `readdir` возвращает запись, начинающуюся с этой позиции. Функция `seekdir` позволяет установить позицию чтения. Аргумент `offset` должен быть либо значением, полученным от `telldir`, либо 0, что означает начало каталога.

Функция `readdir` считывает очередную запись в каталоге. Информация о записи возвращается в виде указателя на структуру `struct dirent`. Память под эту структуру выделена в дескрипторе каталога `DIR`, поэтому каждый вызов `readdir` переписывает старое содержимое структуры. Структура содержит поле `d_name` типа массива символов некоторого зависящего от операционной системы размера. Поле `d_name` содержит имя записи в каталоге, то есть последнюю компоненту пути к файлу. Чтобы сформировать полный путь к файлу или каталогу, нужно эту последнюю компоненту добавить к имени каталога, разделив их символом `'/'`.

Если функция `readdir` не может прочитать очередную запись (при ошибке или когда каталог закончился), функция возвращает `NULL`. Для простоты можно полагать, что если `readdir` вернул `NULL`, каталог не содержит больше записей.

Каждый каталог всегда содержит две записи с именами `.` и `..`, указывающие на сам этот каталог и на его родительский каталог, однако в некоторых файловых системах функция `readdir` может не выдавать вообще эти записи (несмотря на то, что функция `stat` к этим файлам всё применима), в других системах две записи могут не идти первыми. Поэтому для максимальной переносимости программа не должна делать предположений о том, что записи `.` и `..` присутствуют и идут первыми.

Структура `struct dirent` может ещё содержать поле `d_ino`, которое содержит номер индексного дескриптора этой записи, но это поле не должно использоваться. Если необходимо получить номер индексного дескриптора, нужно использовать системный вызов `stat` (или `lstat`). В противном случае программа будет работать неправильно в каталогах, яв-

ляющихся точками монтирования файловых систем. Дело в том, что функция readdir считывает каталог в том виде, в котором он хранится на диске, а система при монтировании файловых систем модифицирует отдельные записи в каталогах и индексные дескрипторы в памяти ядра, но не на диске.

Информацию о файле можно получить с помощью одного из системных вызовов: stat, lstat, fstat. Информация возвращается в структуре struct stat, адрес которой передаётся в эти системные вызовы. Поле st_mode структуры содержит права доступа к файлу и тип файла. Для проверки типа файла следует использовать специальные макросы, например, S_ISDIR для проверки того, является ли запись каталогом. Если макросы недоступны, следует сначала наложить маску S_IFMT на значение поля st_mode, а затем сравнить получившееся значение с проверяемым типом записи. То есть, проверка на то, что запись является каталогом должна выглядеть следующим образом: s.st_mode & S_IFMT == S_IFDIR, где s — переменная типа struct stat. Чтобы получить права доступа к файлу, нужно на значение поля st_mode наложить маску 07777.

Поле st_ino содержит номер индексного дескриптора файла, а поле st_dev содержит номер устройства (файловой системы). Как было сказано ранее, каждый файл однозначно идентифицируется именно по этой паре: ⟨ st_ino, st_dev ⟩.

Полное описание структуры можно найти в приложении.

Ниже приведена программа, которая рекурсивно обходит всю файловую систему и печатает пути ко всем найденным файлам.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <limits.h>

void traverse(char const *dir)
{
  char  name[PATH_MAX];
  DIR  *d;
  struct dirent *dd;
  off_t o;
  struct stat s;
  char *delim = "/";

  /* если в качестве каталога передан корневой каталог,
   * разделитель не нужен */
  if (!strcmp(dir, "/")) delim = "";
  /* открываем каталог */
  if (!(d = opendir(dir))) {
    /* не смогли открыть */
    perror(dir);
    return;
  }
  /* считываем, пока dd не равен NULL, то есть
   * пока есть записи в каталоге */
  while ((dd = readdir(d))) {
    /* пропускаем . и .. */
    if (!strcmp(dd->d_name, ".") || !strcmp(dd->d_name, ".."))
```

```
      continue;
    /* формируем полный путь */
    snprintf(name, PATH_MAX, "%s%s%s", dir, delim, dd->d_name);
    /* получаем информацию о файле
     * используем lstat, чтобы не зациклиться на символических
     * связях */
    if (lstat(name, &s) < 0) continue;
    /* проверяем, что это каталог */
    if (S_ISDIR(s.st_mode)) {
      /* запоминаем текущее положение в каталоге */
      o = telldir(d);
      /* экономим файловые дескрипторы */
      closedir(d);
      /* вызываем себя рекурсивно */
      traverse(name);
      /* восстанавливаем старое положение */
      if (!(d = opendir(dir))) {
        perror(dir);
        return;
      }
      seekdir(d, o);
    } else {
      /* печатаем путь */
      printf("%s\n", name);
    }
  }

  closedir(d);
}

int main(void)
{
  traverse("/");
  return 0;
}
```