4.15. Функции link, unlink, remove и rename

Как мы уже говорили в предыдущем разделе, на индексный узел любого файла могут указывать несколько каталожных записей. Такие ссылки создаются с помощью функции link.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Эта функция создает в каталоге новую запись с именем *newpath*, которая будет указывать на существующий файл *existingpath*. Если запись с именем *newpath* уже существует, функция вернет признак ошибки. Создается только последний компонент полного пути *newpath*, все промежуточные компоненты должны существовать к моменту вызова функции.

Операции создания новой записи в каталоге и увеличения счетчика ссылок должны выполняться атомарно. (Вспомните обсуждение атомарных операций в разделе 3.11.)

Большинство реализаций требуют, чтобы оба пути находились в пределах одной файловой системы, хотя стандарт POSIX.1 допускает возможность создания ссылок на файлы, расположенные в других файловых системах. Если поддерживается создание жестких ссылок на каталоги, то эта операция может выполняться только суперпользователем. Причина такого ограничения состоит в том, что создание жесткой ссылки на каталог может привести к появлению замкнутых «петель» в файловой системе, и большинство обслуивающих ее утилит не смогут обработать их надлежащим образом. (В разделе 4.16 мы покажем пример замкнутой петли, образованной с помощью символической ссылки.) По этой же причине многие реализации файловых систем вообще не допускают создания жестких ссылок на каталоги.

Удаление записей из каталога производится с помощью функции unlink.

```
#include <unistd.h>

int unlink(const char *pathname);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Эта функция удаляет запись из файла каталога и уменьшает значение счетчика ссылок на файл *pathname*. Если на файл указывает несколько ссылок, то его содержимое будет через них по-прежнему доступно. В случае ошибки файл не изменяется.

Как мы уже говорили, чтобы удалить жесткую ссылку на файл, необходимо обладать правом на запись и на исполнение для каталога, в котором находится удаляемая запись. Кроме того, в разделе 4.10 говорилось, что если для каталога установлен бит sticky, то мы должны обладать правом на запись в каталог и являться либо владельцем файла, либо владельцем каталога, либо суперпользователем.

Содержимое файла может быть удалено, только если счетчик ссылок достиг значения 0. Кроме того, содержимое файла нельзя удалить, если он открыт каким-либо процессом. Во время закрытия файла ядро в первую очередь проверяет счетчик процессов, которые открыли этот файл. Если значение

этого счетчика достигло нуля, то ядро проверяет счетчик ссылок и только в том случае, если значение и этого счетчика достигло нуля, содержимое файла будет удалено.

Пример

Программа, приведенная в листинге 4.5, открывает файл, а затем отцепляет его (то есть удаляет с помощью функции unlink). После этого программа приостанавливается на 15 секунд и завершает свою работу.

Листинг 4.5. Открывает файл, а затем удаляет его

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("ошибка вызова функции open");
    if (unlink("tempfile") < 0)
        err_sys("ошибка вызова функции unlink");
    printf("файл удален\n");
    sleep(15);
    printf("конец\n");
    exit(0);
}
```

Запуск программы дает следующие результаты:

```
$ ls -l tempfile        посмотрим размер файла
-rw-r----- 1 sar        413265408 Jan 21 07:14 tempfile
$ df /home              проверим объем доступного свободного пространства
Filesystem  1K-blocks      Used   Available  Use%   Mounted on
/dev/hda4    11021440   1956332     9065108   18%   /home
$ ./a.out &             запустим программу из листинга 4.5 как фоновый процесс
1364                    командная оболочка выводит идентификатор процесса
$ файл удален           файл отцеплен
ls -l tempfile          проверим, остался ли файл на месте
ls: tempfile: No such file or directory       запись из каталога была удалена
$ df /home              проверим, освободилось ли дисковое пространство
Filesystem  1K-blocks      Used   Available  Use%   Mounted on
/dev/hda4    11021440   1956332     9065108   18%   /home
$ конец                 программа завершила работу, все файлы были закрыты
df /home                теперь должно освободиться пространство на диске
Filesystem  1K-blocks      Used   Available  Use%   Mounted on
/dev/hda4    11021440   1552352     9469088   15%   /home
                        на диске освободилось 394.1 Мб
```

Эта характерная особенность функции unlink очень часто используется программами, чтобы обеспечить удаление временных файлов в случае аварийного завершения. Процесс создает файл с помощью функции open или creat, а затем сразу же вызывает функцию unlink. Однако файл не будет удален,

поскольку он остается открытым. Только когда процесс закроет файл или завершит свою работу, что в свою очередь заставит ядро закрыть все файлы, открытые процессом, файл будет удален.

Если аргумент *pathname* является символической ссылкой, то будет удалена сама символическая ссылка, а не файл, на который она ссылается – не существует функции, которая удаляла бы файл по символической ссылке.

Процесс, обладающий привилегиями суперпользователя, может вызвать функцию unlink для удаления каталога, но вообще в таких случаях следует использовать функцию rmdir, которую мы рассмотрим в разделе 4.20.

Удалить жесткую ссылку на файл или каталог можно также с помощью функции remove. Для файлов функция remove абсолютно идентична функции unlink, для каталогов – функции rmdir.

```
#include <stdio.h>

int remove(const char *pathname);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Стандарт ISO C определяет remove как функцию для удаления файлов. Исторически сложившееся в UNIX название unlink было заменено, потому что в то время в большинстве других операционных систем, которые следовали стандарту ISO C, понятие ссылок на файлы не поддерживалось.

Для переименования файла или каталога используется функция rename.

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Стандарт ISO C определяет rename как функцию для переименования файлов. (ISO C вообще не касается каталогов.) Стандарт POSIX.1 расширил это определение, включив в него каталоги и символические ссылки.

Следует отдельно рассмотреть случаи, когда аргумент *oldname* представляет файл, символическую ссылку или каталог. Также необходимо упомянуть о том, что произойдет, если *newname* уже существует.

1. Если аргумент *oldname* указывает на файл, который не является каталогом, то происходит переименование файла или символической ссылки. Если файл *newname* уже существует, он не должен быть каталогом. Если файл *newname* существует и не является каталогом, то он будет удален, а файл *oldname* будет переименован в *newname*. Процесс должен обладать правом записи в каталоги, где находятся файлы *newname* и *oldname*, поскольку предполагается внесение изменений в оба каталога.
2. Если аргумент *oldname* указывает на каталог, то выполняется переименование каталога. Если *newname* существует, то он также должен быть ката-

логом, и этот каталог должен быть пустым. (Под «пустым каталогом» мы имеем в виду каталог, который содержит только две записи: «точка» и «точка-точка».) Если *newname* существует и является пустым каталогом, он будет удален, а каталог *oldname* будет переименован в *newname*. Кроме того, при переименовании каталога аргумент *newname* не должен начинаться с имени каталога *oldname*. Так, например, мы не сможем переименовать каталог `/usr/foo` в `/usr/foo/testdir`, поскольку прежнее имя каталога (`/usr/foo`) содержится в начале нового имени и он не может быть удален.

3. Если аргументы *oldname* или *newname* содержат имя символической ссылки, то будет переименована сама символическая ссылка, а не файл, на который она ссылается.
4. В особом случае, когда аргументы *oldname* и *newname* указывают на один и тот же файл, функция завершается без признака ошибки, но и не производит никаких изменений.

Если файл *newname* уже существует, то мы должны обладать теми же правами, что и для удаления файла. Кроме того, поскольку мы удаляем запись из каталога, который содержит файл *oldname* и создаем новую запись в файле каталога, в котором будет находиться *newname*, мы должны обладать правом на запись и исполнение для обоих каталогов.

4.16. Символические ссылки

Символическая ссылка представляет собой косвенную ссылку на файл, в отличие от жесткой ссылки, которая является прямым указателем на индексный узел файла. Символические ссылки были придуманы с целью обойти ограничения, присущие жестким ссылкам.

- Жесткие ссылки обычно требуют, чтобы ссылка и файл размещались в пределах одной файловой системы
- Только суперпользователь имеет право создавать жесткие ссылки на каталоги

Символические ссылки не имеют ограничений, связанных с файловой системой, и любой пользователь сможет создать символическую ссылку на каталог. Символические ссылки обычно используются для перемещения файлов или даже целой иерархии каталогов в другое местоположение в системе.

> Впервые символические ссылки появились в 4.2BSD и впоследствии стали поддерживаться в SVR4.

Работая с функциями, которые обращаются к файлам по именам, всегда нужно знать, как функция обрабатывает символические ссылки. Если функция следует по символической ссылке, то она будет воздействовать на файл, на который указывает символическая ссылка. В противном случае операция будет производиться над самой символической ссылкой, а не над файлом, на который она указывает. В табл. 4.9 приводится перечень описываемых в этой главе функций, которые следуют по символическим ссылкам. В этом списке отсутствуют функции `mkdir`, `mkfifo`, `mknod` и `rmdir` – они возвращают

признак ошибки, если им в качестве аргумента передается символическая ссылка. Кроме того, функции, которые принимают в качестве аргумента дескриптор файла, такие как fstat и fchmod, также не были включены в список, поскольку в этом случае обработка символических ссылок производится функциями, возвращающими файловые дескрипторы (как правило, open). Следует ли функция chown по символическим ссылкам, зависит от конкретной реализации.

> В старых версиях Linux (до 2.1.81) функция chown не следовала по символическим ссылкам. Начиная с версии 2.1.81 функция chown следует по символическим ссылкам. В операционных системах FreeBSD 5.2.1 и Mac OS X 10.3 функция chown также следует по символическим ссылкам. (В версиях, предшествовавших 4.4BSD, функция chown не следовала по символическим ссылкам, ее поведение было изменено в 4.4.BSD.) В ОС Solaris 9 функция chown также следует по символическим ссылкам. Все четыре платформы предоставляют функцию lchown для изменения владельца самих символических ссылок.

Существует одно исключение, не отмеченное в табл. 4.9, – когда функция open вызывается с установленными одновременно флагами O_CREAT и O_EXCL. Если в этом случае аргумент *pathname* содержит имя символической ссылки, то функция будет завершаться ошибкой с кодом EEXIST. Сделано это с целью закрыть брешь в системе безопасности и предотвратить возможность «обмана» привилегированных процессов путем подмены файлов символическими ссылками.

Таблица 4.9. Интерпретация символических ссылок различными функциями

Функция	Не следует по символической ссылке	Следует по символической ссылке
access		•
chdir		•
chmod		•
chown	•	•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

Пример

С помощью символической ссылки можно создать замкнутую петлю в файловой системе. Большинство функций, анализирующих путь к файлу, обнаружив такую петлю, возвращают в errno код ошибки ELOOP. Рассмотрим следующую последовательность команд:

```
$ mkdir foo                    создать новый каталог
$ touch foo/a                  создать пустой файл
$ ln -s ../foo foo/testdir     создать символическую ссылку
$ ls -l foo
total 0
-rw-r----- 1 sar        0 Jan 22 00:16 a
lrwxrwxrwx 1 sar        6 Jan 22 00:16 testdir -> ../foo
```

Эта последовательность команд создает каталог foo, который содержит файл a и символическую ссылку на каталог foo. На рис. 4.4 приводится схема, на которой каталоги представлены в виде окружностей, а файл – в виде квадрата. Написав простую программу, которая использует стандартную функцию ftw(3) для обхода дерева каталогов и вывода имен всех встретившихся файлов, и запустив ее в ОС Solaris 9, мы получим:

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
    (и еще много строк, пока не произойдет ошибка с кодом ELOOP)
```

В разделе 4.21 будет представлена версия функции ftw, которая использует функцию lstat вместо stat, чтобы предотвратить следование по символическим ссылкам.

> Обратите внимание: в ОС Linux функция ftw использует функцию lstat, поэтому вы не сможете наблюдать подобный эффект.

Разорвать такую замкнутую петлю не составляет труда. Для этого можно воспользоваться функцией unlink, чтобы удалить файл foo/testdir, так как unlink не следует по символическим ссылкам. Однако, если аналогичная

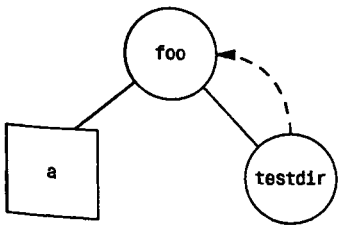

Рис. 4.4. Символическая ссылка testdir создает замкнутую петлю

петля создана с помощью жесткой ссылки, то разорвать ее будет намного сложнее. По этой причине функция link создает жесткие ссылки на каталоги только при наличии у процесса привилегий суперпользователя.

> При написании оригинального текста к этому разделу Ричард Стивенс ради эксперимента действительно создал такую петлю на собственной системе. В результате файловая система была повреждена, и не помогла даже утилита fsck(1). Для восстановления файловой системы пришлось прибегнуть к помощи утилит clri(8) и dcheck(8).
>
> Потребность в жестких ссылках на каталоги давно прошла. Пользователи больше не нуждаются в них – благодаря функции mkdir и символическим ссылкам.

Когда мы открываем файл и передаем функции open имя символической ссылки, функция следует по ссылке и открывает файл, на который она указывает. Если файл, на который указывает ссылка, отсутствует, функция open возвращает признак ошибки, сообщая о невозможности открытия файла. Это может ввести в заблуждение пользователей, которые не знакомы с символическими ссылками, например:

```
$ ln -s /no/such/file myfile        создать символическую ссылку
$ ls myfile
myfile                              команда ls говорит, что файл существует
$ cat myfile                        попробуем заглянуть внутрь файла
cat: myfile: No such file or directory
$ ls -l myfile                      попробуем с ключом -l
lrwxrwxrwx  1 sar        13 Jan 22 00:26 myfile -> /no/such/file
```

Файл myfile существует, однако утилита cat утверждает обратное, потому что myfile – это символическая ссылка, а сам файл, на который она указывает, отсутствует. Запуск команды ls с ключом -l дает нам две подсказки: во-первых, строка вывода ls начинается с символа l, который обозначает символическую ссылку (link), а во-вторых, последовательность -> говорит о том же. У команды ls есть еще один ключ (-F), который добавляет к именам символических ссылок символ «@», благодаря чему можно без труда распознать их даже без ключа -l.

4.17. Функции symlink и readlink

Символические ссылки создаются с помощью функции symlink.

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

В файле каталога создается новая запись *sympath*, которая указывает на файл *actualpath*. Функция не требует существования файла *actualpath* на момент создания символической ссылки. (Мы продемонстрировали эту возможность на примере в предыдущем разделе.) Кроме того, не требуется, чтобы файлы *actualpath* и *sympath* находились в одной и той же файловой системе.

Поскольку функция open следует по символическим ссылкам, нам необходим инструмент, с помощью которого можно было бы открыть саму символическую ссылку, чтобы прочитать имя файла, на который она ссылается. Эти действия выполняет функция readlink.

```
#include <unistd.h>

ssize_t readlink(const char *restrict pathname, char *restrict buf,
                 size_t bufsize);
```

Возвращает количество прочитанных байт в случае успех, −1 в случае ошибки

Эта функция совмещает в себе функции open, read и close. В случае успеха она возвращает количество прочитанных байт, размещенных в *buf*. Строка, помещаемая в буфер *buf*, не завершается нулевым символом.

4.18. Временные характеристики файлов

Каждый файл характеризуется тремя атрибутами времени. Их назначение приводится в табл. 4.10.

Таблица 4.10. Три атрибута времени, связанные с каждым файлом

Поле	Описание	Пример	Ключи команды ls(1)
st_atime	Время последнего доступа к содержимому файла	read	-u
st_mtime	Время последнего изменения содержимого файла	write	По умолчанию
st_ctime	Время последнего изменения статуса индексного узла	chmod, chown	-c

Обратите внимание на различие между временем последнего изменения содержимого файла (st_mtime) и временем последнего изменения статуса индексного узла (st_ctime). Время последнего изменения содержимого файла показывает, когда в последний раз вносились изменения в файл. Время последнего изменения статуса индексного узла определяет время последней модификации индексного узла файла. В этой главе мы упоминали множество операций, которые изменяют индексный узел, не затрагивая при этом содержимое файла: изменение прав доступа, идентификатора пользователя (владельца), количества ссылок на файл и другие. Поскольку информация индексного узла хранится отдельно от содержимого файла, то кроме времени последнего изменения содержимого файла существует и такая характеристика, как время последнего изменения его статуса.

Заметьте, что система не отслеживает время последнего доступа к индексному узлу. По этой причине функции access и stat, например, не изменяют ни одну из трех величин.

Время последнего доступа к файлу часто используется системными администраторами для удаления ненужных файлов, к которым давно никто не обращался. Классический пример – удаление файлов с именами `a.out` и `core`, к которым не обращались более одной недели. Для выполнения подобного рода действий очень часто применяется утилита `find`(1).

Время последнего изменения содержимого файла и время последнего изменения статуса индексного узла могут использоваться для того, чтобы отобрать для архивирования только те файлы, содержимое которых претерпело изменения или у которых был изменен статус индексного узла.

Команда `ls` может отображать или сортировать только по одному из трех значений. По умолчанию при запуске с ключом `-l` или `-t` она использует время последнего изменения содержимого файла. Ключ `-u` заставляет ее использовать время последнего доступа, а ключ `-c` – время последнего изменения статуса индексного узла.

В табл. 4.11 приводится перечень функций и их воздействие на эти три величины. В разделе 4.14 мы уже говорили, что каталог – это на самом деле файл, состоящий из серии записей, каждая из которых содержит имя файла и номер индексного узла файла. Добавление, удаление или изменение этих записей приводит к изменению временных характеристик, связанных с каталогом. По этой причине табл. 4.11 содержит одну колонку для атрибутов времени, связанных с самим файлом или каталогом, и отдельную колонку для атрибутов времени родительского каталога. Создание нового файла, например, воздействует на временные характеристики не только самого файла, но и каталога, в котором этот файл размещается. Однако операции чтения и записи оказывают влияние только на индексный узел файла и никак не сказываются на содержащем этот файл каталоге. (Функции `mkdir` и `rmdir` будут рассматриваться в разделе 4.20. Функция `utime` будет описана в следующем разделе. Шесть функций семейства `exec` будут рассматриваться в разделе 8.10. Функции `mkfifo` и `pipe` мы рассмотрим в главе 15.)

4.19. Функция utime

Функция `utime` изменяет время последнего доступа и время последней модификации файла.

```
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *times);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Эта функция использует следующую структуру:

```
struct utimbuf {
    time_t actime;    /* время последнего доступа */
    time_t modtime;   /* время последнего изменения */
}
```

Таблица 4.11. Воздействие различных функций на время последнего доступа к файлу, последнего изменения содержимого файла и последнего изменения статуса индексного узла

Функция	Файл или каталог			Родительский каталог			Раздел	Примечание
	a	m	c	a	m	c		
chmod, fchmod			•				4.9	
chown, fchown			•				4.11	
creat	•	•	•		•	•	3.4	Создание нового файла (O_CREAT)
creat		•	•				3.4	Усечение существующего файла (O_TRUNC)
exec	•						8.10	
lchown			•				4.11	
link			•		•	•	4.15	Родительский каталог для второго аргумента
mkdir	•	•	•		•	•	4.20	
mkfifo	•	•	•		•	•	15.5	
open	•	•	•		•	•	3.3	Создание нового файла (O_CREAT)
open		•	•				3.3	Усечение существующего файла (O_TRUNC)
pipe	•	•	•				15.2	
read	•						3.7	
remove			•		•	•	4.15	Когда remove = unlink
remove					•	•	4.15	Когда remove = rmdir
rename			•		•	•	4.15	Для обоих аргументов
rmdir					•	•	4.20	
truncate, ftruncate		•	•				4.13	
unlink			•		•	•	4.15	
utime	•	•	•				4.19	
write		•	•				3.8	

Оба поля структуры содержат календарное время, которое выражается количеством секунд, прошедших с начала Эпохи, как это описано в разделе 1.10.

Действие этой функции и привилегии, необходимые для ее выполнения, зависят от того, является ли аргумент *times* пустым указателем.

- Если в аргументе *times* передается пустой указатель (NULL), время последнего доступа к файлу и время последнего изменения файла устанавливаются равными текущему времени. Для этого процесс должен обладать правом записи в файл или иметь эффективный идентификатор пользователя, совпадающий с идентификатором владельца файла.
- Если в аргументе *times* передается не пустой указатель, значения временных характеристик берутся из структуры, на которую указывает аргумент *times*. В этом случае процесс должен иметь эффективный идентификатор пользователя, совпадающий с идентификатором владельца файла, либо обладать привилегиями суперпользователя.

Обратите внимание: невозможно задать время последней модификации статуса индексного узла, так как оно автоматически изменяется в результате вызова функции utime.

В некоторых версиях UNIX команда touch(1) использует эту функцию. Кроме того, стандартные архиваторы tar(1) и cpio(1) могут вызывать функцию utime для установки сохраненных при архивации временных характеристик распакованных файлов.

Пример

Программа, приведенная в листинге 4.6, производит усечение длины файла до нуля, используя функцию open с флагом O_TRUNC, но не изменяет при этом ни время последнего доступа к файлу, ни время последнего изменения файла. Чтобы добиться такого эффекта, программа сначала получает значения временных характеристик файла с помощью функции stat, затем усекает размер файла до нуля и в заключение переустанавливает значения времени с помощью функции utime.

Листинг 4.6. Пример использования функции utime

```
#include "apue.h"
#include <fcntl.h>
#include <utime.h>

int
main(int argc, char *argv[])
{
    int i, fd;
    struct stat statbuf;
    struct utimbuf timebuf;

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) { /* получить значения времени */
            err_ret("%s: ошибка вызова функции stat", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* усечение */
            err_ret("%s: ошибка вызова функции open", argv[i]);
            continue;
```

```
            }
            close(fd);
            timebuf.actime = statbuf.st_atime;
            timebuf.modtime = statbuf.st_mtime;
            if (utime(argv[i], &timebuf) < 0) { /* установить значения времени */
                err_ret("%s: ошибка вызова функции utime", argv[i]);
                continue;
            }
        }
        exit(0);
    }
```

Мы можем продемонстрировать работу программы из листинга 4.6 следующим примером.

```
$ ls -l changemod times          определим размер и время последнего
                                 изменения файлов
-rwxrwxr-x  1 sar   15019 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar   16172 Nov 19 20:05 times
$ ls -lu changemod times         определим время последнего доступа
-rwxrwxr-x  1 sar   15019 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar   16172 Nov 19 20:05 times
$ date                           выведем текущее время и дату
Thu Jan 22 06:55:17 EST 2004
$ ./a.out changemod times        запустим программу из листинга 4.6
$ ls -l changemod times          и проверим результаты
-rwxrwxr-x  1 sar       0 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar       0 Nov 19 20:05 times
$ ls -lu changemod times         проверим так же время последнего доступа
-rwxrwxr-x  1 sar       0 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar       0 Nov 19 20:05 times
$ ls -lc changemod times         и время последнего изменения статуса
                                 индексного узла
-rwxrwxr-x  1 sar       0 Jan 22 06:55 changemod
-rwxrwxr-x  1 sar       0 Jan 22 06:55 times
```

Как мы и ожидали, время последнего доступа к файлу и время последней модификации его содержимого не изменились. Однако время последнего изменения статуса индексного узла было установлено равным времени запуска программы.

4.20. Функции mkdir и rmdir

Создание каталогов производится с помощью функции `mkdir`, а удаление – с помощью функции `rmdir`.

```
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Эта функция создает новый пустой каталог. Записи «точка» и «точка-точка» создаются автоматически. Права доступа к каталогу, задаваемые аргументом *mode*, модифицируются маской режима создания файлов процесса.

Очень часто встречается ошибка, когда аргумент *mode* назначается по аналогии с файлами: выдаются только права на запись и на чтение. Но для каталогов, как правило, необходимо устанавливать хотя бы один бит, дающий право на исполнение, чтобы разрешить доступ к файлам по их именам, находящимся в каталоге (упражнение 4.16).

Идентификаторы пользователя и группы устанавливаются в соответствии с правилами, приведенными в разделе 4.6.

> В операционных системах Solaris 9 и Linux 2.4.22 новый каталог наследует бит set-group-ID от родительского каталога. Файлы, созданные в новом каталоге, наследуют от каталога идентификатор группы. В Linux это поведение определяется реализацией файловой системы. Например, файловые системы ext2 и ext3 предоставляют такую возможность при использовании определенных ключей команды mount(1). Однако реализация файловой системы UFS для Linux не предполагает возможности выбора: бит set-group-ID наследуется всегда, чтобы имитировать исторически сложившуюся реализацию BSD, где идентификатор группы каталога наследуется от родительского каталога.
>
> Реализации, основанные на BSD, не передают бит set-group-ID по наследству – в них просто наследуется идентификатор группы. Поскольку операционные системы FreeBSD 5.2.1 и Mac OS X 10.3 основаны на 4.4BSD, они не требуют наследования бита set-group-ID. На этих платформах вновь создаваемые файлы и каталоги всегда наследуют идентификатор группы родительского каталога независимо от состояния бита set-group-ID.
>
> В ранних версиях UNIX не было функции mkdir. Она впервые появилась в 4.2BSD и SVR3. В более ранних версиях, чтобы создать новый каталог, процесс должен был вызывать функцию mknod. Однако использовать эту функцию мог только процесс, обладающий привилегиями суперпользователя. Чтобы как-то обойти это ограничение, обычная команда создания каталога mkdir(1) должна была иметь установленный бит set-user-ID и принадлежать пользователю root. Чтобы создать каталог из процесса, необходимо было вызывать команду mkdir(1) с помощью функции system(3).

Удаление пустого каталога производится с помощью функции rmdir. Напоминаем, что пустым называется каталог, который содержит только две записи: «точка» и «точка-точка».

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Возвращает 0 в случае успеха, –1 в случае ошибки

Если в результате вызова этой функции счетчик ссылок на каталог становится равным нулю и при этом никакой процесс не держит каталог открытым, то пространство, занимаемое каталогом, освобождается. Если один или более процессов держат каталог открытым в момент, когда счетчик ссылок достигает значения 0, то функция удаляет последнюю ссылку и перед возвратом управления удаляет записи «точка» и «точка-точка». Кроме того, в таком каталоге не могут быть созданы новые файлы. Однако файл каталога не удаляется, пока последний процесс не закроет его. (Даже если другой процесс держит каталог открытым, вряд ли он там делает что-то особенное, так как для успешного завершения функции rmdir каталог должен был быть пуст.)

4.21. Чтение каталогов

Прочитать информацию из файла каталога может любой, кто имеет право на чтение этого каталога. Но только ядро может выполнять запись в каталоги, благодаря чему обеспечивается сохранность файловой системы. В разделе 4.5 мы утверждали, что возможность создания и удаления файлов в каталоге определяется битами прав на запись и на исполнение, но это не относится к непосредственной записи в файл каталога.

Фактический формат файлов каталогов зависит от реализации UNIX и архитектуры файловой системы. В ранних версиях UNIX, таких как Version 7, структура каталогов была очень простой: каждая запись имела фиксированную длину 16 байт – 14 байт отводилось для имени файла и 2 байта для номера индексного узла. Когда в 4.2BSD была добавлена поддержка более длинных имен файлов, записи стали иметь переменную длину. Это означало, что любая программа, выполняющая прямое чтение данных из файла каталога, попадала в зависимость от конкретной реализации. Чтобы упростить положение дел, был разработан набор функций для работы с каталогами, который стал частью стандарта POSIX.1. Многие реализации не допускают чтения содержимого файлов каталогов с помощью функции read, тем самым препятствуя зависимости приложений от особенностей, присущих конкретной реализации.

```
#include <dirent.h>

DIR *opendir(const char *pathname);
```

Возвращает указатель в случае успеха или NULL в случае ошибки

```
struct dirent *readdir(DIR *dp);
```

Возвращает указатель в случае успеха, NULL в случае достижения конца каталога или ошибки

```
void rewinddir(DIR *dp);

int closedir(DIR *dp);
```

Возвращает 0 в случае успеха или –1 в случае ошибки

```
long telldir(DIR *dp);
```

Возвращает значение текущей позиции в каталоге, ассоциированном с *dp*

```
void seekdir(DIR *dp, long loc);
```

Функции telldir и seekdir не являются частью стандарта POSIX.1. Это расширения XSI стандарта Single UNIX Specification – таким образом, предполагается, что они должны быть реализованы во всех версиях UNIX, следующих этой спецификации.

Как вы помните, некоторые из этих функций использовались в программе из листинга 1.1, которая воспроизводила ограниченную функциональность команды ls.

Структура dirent определена в файле <dirent.h> и зависит от конкретной реализации. Однако в любой версии UNIX эта структура содержит как минимум следующие два поля:

```
struct dirent {
    ino_t d_ino;              /* номер индексного узла */
    char d_name[NAME_MAX + 1]; /* строка имени файла, завершающаяся */
                              /* нулевым символом */
}
```

Поле d_ino не определено в стандарте POSIX.1, поскольку эта характеристика зависит от конкретной реализации, но оно определяется в расширении XSI базового стандарта POSIX.1. Сам же стандарт POSIX.1 определяет только поле d_name в этой структуре.

Обратите внимание: параметр NAME_MAX не определен как константа в ОС Solaris 9 – его значение зависит от файловой системы, в которой размещается каталог, и определяется, как правило, с помощью функции fpathconf. Наиболее часто встречается значение NAME_MAX, равное 255 (вспомните табл. 2.12). Так как строка имени файла заканчивается нулевым символом, то не имеет значения, как определен массив d_name в заголовочном файле, поскольку размер массива не соответствует длине имени файла.

DIR является внутренней структурой, которая используется этими шестью функциями для хранения информации о каталоге. По своему назначению структура DIR похожа на структуру FILE, используемую функциями стандартной библиотеки ввода-вывода, которая будет описана в главе 5.

Указатель на структуру DIR, возвращаемый функцией opendir, используется в качестве аргумента остальных пяти функций. Функция opendir выполняет первичную инициализацию таким образом, чтобы последующий вызов readdir прочитал первую запись из файла каталога. Порядок следования записей в каталоге, как правило, зависит от реализации и обычно не совпадает с алфавитным.

Пример

Мы воспользуемся этими функциями работы с каталогами при написании программы, которая обходит дерево каталогов. Цель программы состоит в том, чтобы подсчитать количество файлов каждого типа из перечисленных в табл. 4.3. Программа, приведенная в листинге 4.7, принимает единственный параметр – имя начального каталога – и рекурсивно спускается от этой точки вниз по иерархии каталогов. В ОС Solaris имеется функция ftw(3), ко-

торая обходит дерево каталогов, вызывая определяемую пользователем функцию для каждого встреченного файла. Но с ней связана одна проблема: она вызывает функцию stat для каждого файла, в результате чего программа следует по символическим ссылкам. Например, если мы начнём просмотр каталогов от корня файловой системы, в котором имеется символическая ссылка с именем /lib, указывающая на каталог /usr/lib, то все файлы в каталоге /usr/lib будут сосчитаны дважды. Чтобы устранить эту проблему, ОС Solaris предоставляет дополнительную функцию nftw(3), для которой можно отключить следование по символическим ссылкам. Хотя можно было бы бы использовать функцию nftw, давайте все-таки напишем собственную версию функции для обхода дерева каталогов, чтобы показать принципы работы с каталогами.

> Обе функции, ftw и nftw, включены в стандарт Single UNIX Specification как расширения XSI базового стандарта POSIX.1. Реализации этих функций имеются в операционных системах Solaris 9 и Linux 2.4.22. Системы, основанные на BSD, предоставляют функцию fts(3) с аналогичной функциональностью. Она реализована в операционных системах FreeBSD 5.2.1, Mac OS X 10.3 и Linux 2.4.22.

Листинг 4.7. Рекурсивный обход дерева каталогов с подсчетом количества файлов по типам

```
#include "apue.h"
#include <dirent.h>
#include <limits.h>

/* тип функции, которая будет вызываться для каждого встреченного файла */
typedef int Myfunc(const char *, const struct stat *, int);

static Myfunc  myfunc;
static int     myftw(char *, Myfunc *);
static int     dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int
main(int argc, char *argv[])
{
    int ret;

    if (argc != 2)
        err_quit("Использование: ftw <начальный_каталог>");

    ret = myftw(argv[1], myfunc); /* выполняет всю работу */

    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1;/* во избежание деления на 0; вывести 0 для всех счетчиков */
    printf("обычные файлы = %7ld, %5.2f %%\n", nreg,
            nreg*100.0/ntot);
    printf("каталоги = %7ld, %5.2f %%\n", ndir,
            ndir*100.0/ntot);
    printf("специальные файлы блочных устройств = %7ld, %5.2f %%\n", nblk,
```

```
            nblk*100.0/ntot);
    printf("специальные файлы символьных устройств = %7ld, %5.2f %%\n", nchr,
            nchr*100.0/ntot);
    printf("FIFO = %7ld, %5.2f %%\n", nfifo,
            nfifo*100.0/ntot);
    printf("символические ссылки = %7ld, %5.2f %%\n", nslink,
            nslink*100.0/ntot);
    printf("сокеты = %7ld, %5.2f %%\n", nsock,
        nsock*100.0/ntot);
    exit(ret);
}

/*
 * Обойти дерево каталогов, начиная с каталога "pathname".
 * Пользовательская функция func() вызывается для каждого встреченного файла.
 */

#define FTW_F   1    /* файл, не являющийся каталогом */
#define FTW_D   2    /* каталог */
#define FTW_DNR 3    /* каталог, который не доступен для чтения */
#define FTW_NS  4    /* файл, информацию о котором */
                     /* невозможно получить с помощью stat */

static char *fullpath; /* полный путь к каждому из файлов */

static int             /* возвращаем то, что вернула функция func() */
myftw(char *pathname, Myfunc *func)
{
    int len;

    fullpath = path_alloc(&len); /* выделить память для PATH_MAX+1 байт */
                                 /* (листинг 2.3) */
    strncpy(fullpath, pathname, len); /* защита от */
    fullpath[len-1] = 0;              /* переполнения буфера */

    return(dopath(func));
}

/*
 * Обход дерева каталогов, начиная с "fullpath". Если "fullpath" не является
 * каталогом, для него вызывается lstat(), func() и затем выполняется возврат.
 * Для каталогов производится рекурсивный вызов функции.
 */
static int         /* возвращаем то, что вернула функция func() */
dopath(Myfunc* func)
{
    struct stat    statbuf;
    struct dirent *dirp;
    DIR           *dp;
    int            ret;
    char          *ptr;

    if (lstat(fullpath, &statbuf) < 0) /* ошибка вызова функции stat */
        return(func(fullpath, &statbuf, FTW_NS));
```

```
    if (S_ISDIR(statbuf.st_mode) == 0) /* не каталог */
        return(func(fullpath, &statbuf, FTW_F));

/*
 * Это каталог. Сначала вызовем функцию func(),
 * а затем обработаем все файлы в этом каталоге.
 */
    if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    ptr = fullpath + strlen(fullpath); /* установить указатель */
                                       /* в конец fullpath */
    *ptr++ = '/';
    *ptr = 0;

    if ((dp = opendir(fullpath)) == NULL) /* каталог недоступен */
        return(func(fullpath, &statbuf, FTW_DNR));

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0 ||
            strcmp(dirp->d_name, "..") == 0)
                continue;              /* пропустить каталоги "." и ".." */

        strcpy(ptr, dirp->d_name);     /* добавить имя после слэша */

        if ((ret = dopath(func)) != 0) /* рекурсия */
            break;                     /* выход по ошибке */
    }

    ptr[-1] = 0; /* стереть часть строки от слэша и до конца */

    if (closedir(dp) < 0)
        err_ret("невозможно закрыть каталог %s", fullpath);

    return(ret);
}

static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG: nreg++; break;
        case S_IFBLK: nblk++; break;
        case S_IFCHR: nchr++; break;
        case S_IFIFO: nfifo++; break;
        case S_IFLNK: nslink++; break;
        case S_IFSOCK: nsock++; break;
        case S_IFDIR:
            err_dump("признак S_IFDIR для %s", pathname);
                    /* каталоги должны иметь тип = FTW_D */
    }
    break;

    case FTW_D:
```

```
            ndir++;
            break;

        case FTW_DNR:
            err_ret("закрыт доступ к каталогу %s", pathname);
            break;

        case FTW_NS:
            err_ret("ошибка вызова функции stat для %s", pathname);
            break;

        default:
            err_dump("неизвестный тип %d для файла %s", type, pathname);
        }

        return(0);
    }
```

Эта программа получилась даже более универсальной, чем было необходимо. Таким образом мы хотели пояснить работу функции `ftw`. Например, функция `myfunc` всегда возвращает значение 0, хотя функция, которая ее вызывает, готова обработать и ненулевое значение.

За дополнительной информацией о технике обхода дерева каталогов и использовании ее в стандартных командах UNIX – `find`, `ls`, `tar` и других – обращайтесь к [Fowler, Korn and Vo 1989].

Лабораторная работа по Операционным системам

Файлы и каталоги

Задание:

1. Структурировать исходный код программы в листинге 4.7.
2. Изменить программу так, чтобы она выводила на экран дерево каталогов.
3. Изменить функцию myftw() так, чтобы каждый раз, когда встречается каталог, функции lstat() передавался не полный путь к файлу, а только его имя.
   Для этого после обработки всех файлов в каталоге вызовите chdir("..").

Вопросы для самопроверки

1.
   a. Как создать файл с правами доступа для всех видов пользователей равными 0;
   b. Как создать файл с полными правами доступа для всех видов пользователей;
   c. Что произойдет, если маску режима создания файлов задать равной 777?
   d. Сможете ли Вы прочитать собственный файл при сброшенном бите user-read?

2. Какое поле структуры stat отвечает за размер файла? Что обозначает значение поля размера файла для символических ссылок? Чему будет равно значение этого поля для символической ссылки lib->user/lib? Что

определяют поля st_blksize и st_blocks? Какой размер буфера надо установить для минимизации времени выполнения операций ввода-вывода?

3. На примерер файлов, содержащих «дырки», поясните разницу между работой команд ls –l и du –s.
4. Как уменьшить размерфайла?
5. Какая функция удаляет записи из каталога? Когда можно удалить файл и что происходит с его индексными узлами?
6. Что происходит при перемещении/переименовании файла в пределах одной файловой системы с фактическим содержимым файла (с его индексными узлами)? Изменяется ли при этом счетчик ссылок?
7. Файл имеет три атрибута времени: st_atime, st_mtime, st_ctime. Какой ключ команды ls позволит вывести на экран информацию о последнем изменении статуса индексного узла?
8. Когда с помощью команды rmdir будет удален из системы каталог?
9. Что возвращает функция opendir()?
10. Какая функция «читает» содержимое каталога?
11. Какие типы файлов определены в Unix/Linux?