Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Промышленная электроника»

А. В. Сахарук

ПРОГРАММИРОВАНИЕ ДЛЯ ВСТРАИВАЕМЫХ ОПЕРАЦИОННЫХ СИСТЕМ

ПРАКТИКУМ
для студентов специальности
1-53 01 07 «Информационные технологии
и управление в технических системах»
дневной формы обучения

Гомель 2022

УДК 004.451(075.8)
ББК 32.972.11я73
С22

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 10 от 25.06.2021 г.*)

Рецензент: доц. каф. «Информационные системы и технологии»
ГГТУ им. П. О. Сухого канд. техн. наук, доц. *В. С. Захаренко*

Сахарук, А. В.

С22 Программирование для встраиваемых операционных систем : практикум для студентов специальности 1-53 01 07 «Информационные технологии и управление в технических системах» днев. формы обучения / А. В. Сахарук. – Гомель : ГГТУ им. П. О. Сухого, 2022. – 139 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Содержит пять тем для проведения практических занятий с основными теоретическими сведениями, подробными примерами, заданиями для самостоятельной работы и контрольными вопросами.

Для студентов специальности 1-53 01 07 «Информационные технологии и управление в технических системах» дневной формы обучения.

УДК 004.451(075.8)
ББК 32.972.11я73

Введение

Программирование для UNIX изначально зарождалось именно как системное. Исторически системы UNIX не включали значительного количества высокоуровневых концепций. Даже при программировании в среде разработки, например, в системе X Window, в полной мере задействовались системные API ядра UNIX.

Можно сравнить системное программирование с программированием приложений. Важная черта системного программирования заключается в том, что программист, специализирующийся в этой области, должен обладать глубокими знаниями оборудования и операционной системы, с которыми он имеет дело. Системные программы взаимодействуют в первую очередь с ядром и системными библиотеками, а прикладные опираются и на высокоуровневые библиотеки. Такие высокоуровневые библиотеки абстрагируют детальные характеристики оборудования и операционной системы. У подобного абстрагирования есть несколько целей: переносимость между различными системами, совместимость с разными версиями этих систем, создание удобного в использовании (либо более мощного, либо и то и другое) высокоуровневого инструментария. Соотношение, насколько активно конкретное приложение использует высокоуровневые библиотеки и насколько — систему, зависит от уровня стека, для которого было написано приложение. Некоторые приложения создаются для взаимодействия исключительно с высокоуровневыми абстракциями. Однако даже такие абстракции, весьма отдаленные от самых низких уровней системы, лучше всего получаются у специалиста, имеющего навыки системного программирования. Те же проверенные методы и понимание базовой системы обеспечивают более информативное и разумное программирование для всех уровней стека.

Тема 1. Работа с буферизированным вводом-выводом

Блок — это абстракция, представляющая мельчайший компонент системы, предназначенный для хранения данных в файловой системе. Внутри ядра все операции файловой системы трактуются именно в контексте блоков. Действительно, блок — это настоящий общий знаменатель всего ввода- вывода. Следовательно, любые действия ввода-вывода не могут осуществляться над объемом информации, не превышающим размер одного блока, а также над объемом данных, который не выражается целым числом, кратным размеру блока. Если требуется прочитать всего один байт — придется считывать целый блок. Необходимо записать 4,5 единицы данных - нужно будет записать пять блоков, причем впоследствии тот, который записан частично, будет считываться целиком. При этом обновится лишь половина блока, а отложенная запись займет целый блок.

Операции с частичной записью блоков неэффективны. Операционной системе приходится «вписывать» ввод-вывод в имеющиеся рамки и гарантировать, что все действия укладываются в блочные границы. Это делается с помощью округления с увеличением до ближайшего целого блока. К сожалению, именно в таком ключе обычно пишутся приложения для пользовательского пространства. Большинство приложений при работе оперируют более высокоуровневыми абстракциями — например, полями и строками. Размер таких сущностей варьируется независимо от размера блока.

Ситуация дополнительно усугубляется из-за лишних системных вызовов, которые необходимы, например, для 1024 операций считывания блоков, содержащих по одному байту каждый, ведь можно было бы обойтись всего одним вызовом, если бы эта информация компактно располагалась в блоке размером 1024 байт. Решение этой проблемы с производительностью заключается в использовании *ввода-вывода с пользовательским буфером.* Этот способ позволяет приложениям естественным образом считывать и записывать любые объемы данных, но осуществлять ввод-вывод в размере целых блоков данной файловой системы.

1.1 Ввод-вывод с пользовательским буфером

Программы, которым приходится выполнять множество мелких системных вызовов к обычным файлам, часто осуществляют ввод-вывод с пользовательским буфером. Так называется буферизация, выполняемая в пользовательском пространстве. Ее можно организовывать в приложении вручную либо прозрачно выполнять в библиотеке. Однако такая буферизация никак не связана с ядром. На внутрисистемном уровне буферизация данных в ядре происходит посредством отложенных записей, объединения смежных запросов ввода-вывода и опережающего считывания. Пользовательская буферизация выполняется иначе, но также нацелена на улучшение производительности.

Рассмотрим пример с использованием программы dd, работающей в пользовательском пространстве:

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

Имеется аргумент bs=1, поэтому данная команда будет копировать 2 Мбайт информации с устройства /dev/zero (это виртуальное устройство, выдающее бесконечное количество нулей) в файл pirate. Операция будет передана в виде 2 097 152 однобайтовых фрагментов. Таким образом, на копирование этих данных мы затратим примерно 2 000 000 операций считывания и записи — по байту за раз.

Рассмотрим копирование тех же 2 Мбайт информации, но уже с использованием блоков размером по 1024 байт каждый:

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

Эта операция позволяет скопировать те же 2 Мбайт информации в тот же файл, но требует в 1024 раза меньше операций считывания и записи. Налицо радикальное улучшение производительности, как видно из таблицы 1.1. В ней указано затраченное время (измеренное тремя разными способами) четырьмя командами dd, которые отличались лишь размером блока. Реальное время — это общее время, истекшее на настенных часах. Пользовательское время — это время, потраченное на исполнение программного кода в пользовательском пространстве. Системное время — это время, затраченное на выполнение инициированных процессом системных вызовов в пространстве ядра.

Таблица 1.1. Влияние размера блока на производительность

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

При использовании фрагментов данных по 1024 байт достигается грандиозное улучшение производительности по сравнению с передачей информации по одному байту. Тем не менее данные, приведенные выше, также свидетельствуют, что при использовании блоков с большим размером — и, соответственно, при дальнейшем сокращении количества системных вызовов — производительность может и снижаться, если используемые при работе фрагменты не кратны размеру блока диска. Несмотря на то что запросы 1130-байтовых фрагментов требуют меньшего количества системных вызовов, они оказываются менее эффективными, чем 1024-байтовые запросы, кратные размеру блока.

Чтобы обратить эти аспекты производительности себе на пользу, необходимо заранее знать размер физического блока на данном устройстве. Результаты, приведенные в табблице, показывают, что размер блока, скорее всего, будет равен 1024 байт, целочисленному кратному 1024 или делителю 1024. В случае с /dev/zero точный размер блока равен 4096 байт.

1.2 Размер блока

На практике размер блока обычно составляет 512, 1024, 2048, 4096 или 8192 байт.

Как показано выше (см. таблица 1.1), для значительного повышения производительности достаточно просто выполнять операции фрагментами, которые являются целочисленными кратными

или делителями размера блока. Дело в том, что и ядро, и аппаратное обеспечение работает в контексте блоков. Соответственно, если оперировать либо размером блока, либо значением, которое аккуратно вписывается в блок, то все запросы ввода-вывода будут выполняться в пределах целых блоков. Лишняя работа в ядре исключается.

Узнать размер блока на конкретном устройстве довольно просто: для этого используется системный вызов stat(). Однако оказывается, что в большинстве случаев не требуется знать точный размер блока.

Выбирая размер для будущих операций ввода-вывода, главное — не получить какую-нибудь заведомо неровную величину, например 1130. Ни один блок в истории UNIX не имел размер 1130 байт — если выберать такой объем для операций, то уже после первого запроса выравнивание ввода-вывода будет нарушено. Если же выбранный размер операции позволяет сохранять выравнивание по границам блоков, то производительность будет высокой. Чем больше кратное, тем меньше системных вызовов вам потребуется.

Соответственно, самый простой вариант — использовать при вводе-выводе достаточно большой буфер, являющийся общим кратным типичных размеров блоков. Очень удобны значения 4096 и 8192 байт.

Проблема заключается в том, что сами программы редко оперируют цельными блоками — они работают с отдельными полями, строками, символами, а не с такой абстракцией, как блок. Ввод-вывод с пользовательским буфером устраняет разрыв между файловой системой, работающей с блоками, и приложением, оперирующим собственными абстракциями. Принцип работы пользовательского буфера одновременно простой и мощный: по мере того как данные записываются, они сохраняются в буфере в пределах адресного пространства конкретной программы. Когда размер буфера достигает установленного предела, называемого размером буфера, содержимое этого буфера переносится на диск за одну операцию записи. Считывание данных также происходит фрагментами, равными размеру буфера и выровненными по границам блоков.

Поступающие от приложения запросы на считывание имеют разные размеры и обслуживаются не из файловой системы напрямую, а удовлетворяются фрагментами, получаемыми через буфер. По мере того как приложение считывает все больше и больше информации,

данные выдаются из буфера кусками. Наконец, как только буфер пустеет, начинается считывание следующего сравнительно крупного фрагмента, выровненного по границам блоков. Таким образом, приложение может считывать и записывать любые произвольные фрагменты данных, как потребуется, но буферизация данных всегда выполняется лишь сравнительно большими кусками. Эти крупные операции, выровненные по границам блоков, уже направляются в файловую систему. В конечном итоге удается обойтись меньшим количеством системных вызовов при работе со значительными объемами данных, выровненными строго по границам блоков. В результате мы имеем серьезное повышение производительности.

Можно и самостоятельно реализовать пользовательскую буферизацию в программах. Именно так и делается во многих критически важных приложениях. Однако в абсолютном большинстве программ используется популярная стандартная библиотека ввода-вывода, входящая в состав стандартной библиотеки С, либо, как вариант, библиотека классов ввода-вывода языка С++, с помощью которых легко создавать надежные и практичные решения с пользовательской буферизацией.

1.3 Стандартный ввод-вывод

В составе стандартной библиотеки С есть стандартная библиотека ввода-вывода, иногда сокращенно именуемая stdio. Она, в свою очередь, предоставляет независимое от платформы решение для пользовательской буферизации. Стандартная библиотека ввода-вывода проста в использовании, но ей не занимать мощности.

В отличие от таких языков программирования, как FORTRAN, язык С не содержит никакой встроенной поддержки ключевых слов, которая обеспечивала бы более сложный функционал, чем управление выполнением программы, арифметические действия и т. д. Естественно, в языке нет и встроенной поддержки ввода-вывода. По эволюции языка программирования С, его пользователи разработали стандартные наборы процедур, обеспечивающих основную функциональность. В частности, речь идет о манипуляции строками, математических процедурах, работе с датой и временем, а

также о вводе-выводе. Со временем эти процедуры совершенствовались. В 1989 году, когда был ратифицирован стандарт ANSIC (C89), все эти функции были объединены в стандартную библиотеку С. В С95, С99 и С11 добавилось несколько новых интерфейсов, однако стандартная библиотека ввода- вывода осталась практически нетронутой со времени появления в С89.

Оставшаяся часть этой главы посвящена вводу-выводу с пользовательским буфером, тому, как он относится к файловому вводу-выводу и как реализован в стандартной библиотеке С. Иными словами, нас интересует открытие, закрытие, считывание и запись файлов с помощью стандартной библиотеки С. Решение, будут ли в приложении использоваться стандартный ввод-вывод, собственное решение с пользовательским буфером либо обычные системные вызовы, принимает сам разработчик. Это решение должно быть тщательно взвешенным и приниматься с учетом всех потребностей приложения и его поведения.

В стандартах С некоторые детали всегда остаются на усмотрение конкретной реализации, и в разных реализациях действительно часто добавляются новые возможности.

Указатели файлов. Процедуры стандартного ввода-вывода не работают непосредственно с файловыми дескрипторами. Вместо этого каждая использует свой уникальный идентификатор, обычно называемый указателем файла. В библиотеке С указатель файла ассоциируется с файловым дескриптором (отображается на него). Указатель файла представлен как указатель на определение типа FILE, определяемый в <stdio.h>.

В терминологии ввода-вывода открытый файл называется потоком данных. Потоки могут быть открыты для чтения (поток данных ввода), для записи (поток данных вывода) или для того и другого (поток данных ввода/вывода).

1.4 Открытие файлов

Файлы открываются для чтения или записи с помощью функции fopen():

```
#include <stdio.h>
FILE * fopen (const char *path, const char *mode);
```

Эта функция открывает файл path, поведение которого определено в mode, и ассоциирует с ним новый поток данных.

Аргумент mode описывает, как открывать конкретный файл. Данный аргумент может быть представлен одной из следующих строк.

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

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

Например, следующий код открывает для чтения файл /etc/manifest и ассоциирует его с потоком данных stream:

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

1.5 Открытие потока данных с помощью файлового дескриптора

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

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

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

После преобразования файлового дескриптора в поток данных ввод-вывод больше не выполняется напрямую с этим файловым дескриптором. Тем не менее это не возбраняется. Стоит обратить внимание: файловый дескриптор не дублируется, а просто ассоциируется с новым потоком данных. При закрытии потока данных закроется и файловый дескриптор.

В случае успеха fdopen() возвращает допустимый указатель файла, при ошибке она возвращает NULL и присваивает errno соответствующее значение.

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

```
FILE *stream; int fd;
fd = open ("/home/kidd/map.txt", O_RDONLY);
if (fd == -1)
        /* ошибка */
stream = fdopen (fd, "r");
if(!stream)
        /* ошибка */
```

1.6 Закрытие потоков данных

Функция fclose() закрывает конкретный поток данных:

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

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

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

```
#define _GNU_SOURCE
#include <stdio.h>
int fcloseall(void);
```

Перед закрытием все потоки данных сбрасываются на диск. Функция является специфичной для Linux и всегда возвращает 0.

1.7 Считывание из потока данных

Стандартная библиотека C реализует множество функций для считывания из открытого потока данных. Среди них есть как общеизвестные, так и малораспространенные. Чтобы из потока данных можно было считать информацию, он должен быть открыт как поток данных ввода в подходящем режиме, то есть в любом допустимом режиме, кроме w и a.

Зачастую идеальный принцип ввода-вывода сводится к считыванию одного символа в момент времени. Функция fgetc() используется для считывания отдельного символа из потока данных:

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

Эта функция считывает следующий символ из stream и возвращает его как unsigned char, приведенный к int. Такое приведение осуществляется, чтобы получить достаточно широкий диапазон для уведомлений EOF или описания ошибок: в таких случаях возвращается EOF. Возвращаемое значение fgetcОдолжно быть сохранено в int. Сохранение в char — распространенный и

опасный промах, ведь в таком случае вы не можете обнаруживать ошибки.

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

```
int c;
c = fgetc (stream);
if (c == EOF)
        /* ошибка */
else
        printf ("c=%c\n", (char) c);
```

Поток данных, указанный в stream, должен быть открыт для чтения.

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

```
#include<stdio.h>
int ungetc(int c, FILE*stream);
```

При каждом вызове отправляется обратно в поток данных stream символ c, приведенный к unsigned char. В случае успеха возвращается c, в случае ошибки — EOF. При следующем считывании из stream будет возвращен c. Если возвращается в поток данных несколько символов, то они приходят туда в обратном порядке: последним пришел — первым вернулся. Образуется своеобразный стек. Согласно стандарту C лишь один такой возврат должен гарантированно быть успешным при отсутствии «вклинивающихся» запросов на считывание. В свою очередь, в некоторых реализациях разрешается всего один такой возврат. В Linux допускается любое количество возвратов при условии наличия достаточного объема памяти. Как минимум один такой возврат в Linux, разумеется, гарантированно будет успешным.

Если в ходе возврата производится «вклинивающийся» вызов функции позиционирования после вызова ungetc(), но еще до выдачи запроса на считывание, то все символы, отправленные обратно, будут удалены. Именно это происходит и при работе с программными потоками в одном процессе, так как программные потоки в процессе совместно используют общий буфер.

Функция fgets() считывает строку из указанного потока данных:

#include <stdio.h>

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

Эта функция может считать из потока данных stream количество байтов, как максимум на один меньшее, чем количество size байт. Результат считывания сохраняется в str. Символ нуля (\0) сохраняется в буфере после последнего считанного байта. Считывание прекращается, когда достигнут конец файла или первый символ новой строки. Если начинается считывание новой строки, то в str сохраняется \n.

В случае успеха возвращается str, в случае ошибки — NULL.

Например:

```
char buf[LINE_MAX];
if (!fgets (buf, LINE_MAX, stream))
    /* ошибка */
```

POSIX определяет LINE_MAX в <limits.h>: это максимальный размер, который может иметь строка ввода, чтобы интерфейсы POSIX для манипуляций со строками могли ею оперировать. В библиотеке C такое ограничение отсутствует — строки могут быть любого размера, — но мы никак не можем сообщить об этом в самом определении LINE_MAX. Для обеспечения надежности переносимых программ в них можно обходиться LINE_MAX; в Linux значение этого параметра задано достаточно высоким. В специфичных для Linux программах нет необходимости беспокоиться о границах размеров строк.

Считывание произвольных строк. Зачастую построчное считывание fgets() — как раз то, что требуется. Однако не реже оно только мешает. Иногда разработчик предпочитает воспользоваться разделителем, а не переходить на новую строку. В других случаях разделитель вообще не нужен, и уж совсем редко разработчику может понадобиться, чтобы разделитель сохранялся в буфере. Практика показывает, что решение сохранять переход на новую строку в возвращаемом буфере чаще всего оказывается ошибочным.

В качестве замены несложно написать функцию fgets(), которая будет использовать fgetc(). Например, в следующем фрагменте считывается n - 1 байт из stream в str, после чего к полученной информации дозаписывается символ \0:

```
char *s; int c;
```

```
s = str;
while (--n >0 &&(c = fgetc (stream)) != EOF)
        *s++ = c;
*s = '\0';
```

Этот пример можно изменить, чтобы не считывался и разделитель (в качестве разделителя выступает целое число d, в этом примере мы не можем использовать в таком качестве символ нуля).

```
char *s; int c = 0;
s = str;
while (--n >0 &&(c = fgetc (stream)) != EOF &&(*s++ = c) != d)
        if (c == d)
                *--s = '\0';
        else
                *s= '\0';
```

Если установить значение d в \n, то получится примерно такое же поведение, как и у fgets(), но уже не будем сохранять в буфере переход на новую строку.

В зависимости от реализации fgets() этот вариант может работать медленнее, так как он неоднократно делает вызовы функции fgetc(). Однако это немного иная проблема, чем показанная в исходном примере с dd. Да, в последнем фрагменте имеются издержки, связанные с дополнительными вызовами функций, но избавляемся от нагрузки, которая была обусловлена избыточными системными вызовами, а также от неудобств, связанных с невыровненным вводом-выводом при dd с bs=1, — последняя проблема намного серьезнее.

В некоторых приложениях считывать отдельные символы или строки недостаточно. Иногда разработчику требуется считывать и записывать сложные двоичные данные, в частности структуры С. Для этой цели в стандартной библиотеке ввода- вывода предоставляется функция fread():

```
#include <stdio.h>
size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

При вызове fread() можно прочитать вплоть до nr фрагментов данных, каждый размером size байт. Считывание происходит из потока данных stream в буфер, указанный buf. Значение файлового

указателя увеличивается на число, равное количеству прочитанных байтов.

Возвращается количество считанных элементов. При ошибке или достижении конца файла функция возвращает значение, меньшее чем nr. К сожалению, нельзя узнать, какое именно из двух условий наступило, — для этого потребуется специально применить функции ferror() и feof().

Из-за различий, связанных с переменным размером, выравниванием, заполнением и порядком байтов, двоичные данные, записанные одним приложением, могут быть непригодны для считывания другим приложением либо даже тем же самым приложением на другой машине.

1.8 Проблемы, связанные с выравниванием

Во всех машинных архитектурах действуют свои требования к выравниванию данных. Обычно программисты представляют себе память просто как массив буферов. Однако процессор читает и записывает память не фрагментами, равными энному количеству байтов. При обращении к памяти процессор делает это с конкретной детализацией — например, по 2, 4, 8 или 16 байт. Адресное пространство каждого процесса начинается с 0, поэтому процессы должны инициировать доступ с адреса, который является целочисленным кратным единицы детализации.

Соответственно, сохранять переменные C и обращаться к ним нужно с выровненных адресов. В принципе, переменным свойственно естественное выравнивание, то есть выравнивание, соответствующее размеру типа данных в языке C. Например, 32-битное целое число выровнено по 4-байтовым границам. Иными словами, в большинстве архитектур int должно сохраняться в памяти по адресу, значение которого без остатка делится на 4.

При доступе к невыровненным данным возникают различные проблемы, зависящие от машинной архитектуры. Некоторые процессоры могут обращаться к невыровненным данным, но за счет снижения производительности. Другие процессоры вообще не умеют обращаться к невыровненным данным, при попытке это сделать возникает аппаратное исключение. Хуже того, некоторые процессоры

просто бесшумно удаляют младшие биты, чтобы принудительно добиться выравнивания адреса. Практически всегда это приводит к непредвиденным последствиям.

Как правило, компилятор естественным образом выравнивает все данные, и это не представляет заметной проблемы для программиста. Однако при работе со структурами, управлении памятью вручную, сохранении двоичных данных на диск и обмене информацией по сети вопросы выравнивания данных могут стать актуальными, поэтому системный программист должен уметь справляться с подобными сложностями.

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

```
char buf[64]; size_t nr;
nr = fread (buf, sizeof(buf), 1, stream);
if(nr== 0)
        /* ошибка */
```

1.9 Запись в поток данных

Стандартная библиотека C содержит функции не только для чтения, но и для записи в открытый поток данных.

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

```
#include <stdio.h>
int fputc (int c, FILE *stream);
```

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

Пример использования прост:

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

В данном примере записывается символ p в поток данных stream, который должен быть открыт для записи.

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

```
#include <stdio.h>
int fputs (const char *str, FILE *stream);
```

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

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

```
FILE *stream;
stream = fopen ("journal.txt", "a");
if (!stream)
        /* ошибка */
if (fputs ("The ship is made of wood.\n", stream) == EOF)
        /* ошибка */
if (fclose (stream) == EOF)
        /* ошибка */
```

Отдельных символов и строк недостаточно, если программе требуется записывать сложные данные. Для непосредственного сохранения двоичных данных, например переменных языка С, в рамках стандартного ввода-вывода предоставляется функция fwrite():

```
#include <stdio.h>
size_t fwrite (void *buf, size_t size, size_t nr,FILE *stream);
```

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

Пример программы, в которой используется буферизованный ввод-вывод.

Рассмотрим пример — фактически полнофункциональную программу, включающую в себя многие интерфейсы. Сначала программа определяет структуру struct pirate, а потом определяет две переменные такого типа. Программа инициализирует одну из переменных, а потом записывает ее на диск через поток данных вывода в файл data. В другом потоке данных программа вновь считывает данные из файла data непосредственно в другой экземпляр

struct pirate. Наконец, программа записывает содержимое структуры в стандартный вывод:

```
#include <stdio.h>
int main (void)
{
        FILE *in, *out;
        struct pirate {
        char  name[100];        /* настоящее имя */
        unsigned long booty;        /* вознаграждение в фунтах стерлингов */
        unsigned int beard_len;/* длина бороды в дюймах */
} p, blackbeard= { "EdwardTeach", 950, 48 };
out = fopen ("data", "w");
if (!out) {
        perror ("fopen"); return 1;
}
if (!fwrite (&blackbeard, sizeof (struct pirate), 1, out)) {
         perror ("fwrite");
        return 1;
}
if (fclose (out)) {
        perror ("fclose"); return 1;
}
in = fopen ("data", "r");
if (!in) {
        perror ("fopen"); return 1;
}
if (!fread (&p, sizeof (struct pirate), 1, in)) {
perror ("fread");
return 1;
}
if (fclose (in)) {
        perror ("fclose");
        return 1;
}
printf ("name=\"%s\" booty=%lu beard_len=%u\n",p.name, p.booty, p.beard_len);
```

return 0;
}

На выходе, разумеется, имеем исходные значения: name="Edward Teach" booty=950 beard_len=48

Опять же важно помнить, что из-за различий размеров переменных, при выравнивании и т. д. двоичные данные, записанные в одном приложении, могут оказаться нечитаемыми в других. Это означает, что иное приложение — или даже то же самое приложение на другой машине — может оказаться не в состоянии правильно считать данные, записанные с помощью fwrite().

1.10 Позиционирование в потоке данных

Часто бывает полезно манипулировать текущей позицией в потоке данных. Допустим, приложение считывает сложный файл, требующий манипулирования записями, и программа должна произвольно перемещаться по этому файлу. В другой ситуации, возможно, придется сбрасывать поток данных на нулевую позицию. Для подобных случаев в стандартной библиотеке ввода-вывода предоставляется семейство интерфейсов, функционально эквивалентных системному вызову lseek(). Функция fseek(), наиболее распространенный интерфейс позиционирования из инструментов стандартного ввода-вывода, управляет файловой позицией в потоке данных stream в зависимости от значений offset и whence:

```
#include <stdio.h>
int fseek (FILE *stream, long offset, int whence);
```

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

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

аргумент whence (EINVAL). В качестве альтернативы для стандартного ввода-вывода предоставляется функция fsetpos():

```
#include <stdio.h>
int fsetpos(FILE*stream, fpos_t*pos);
```

Эта функция устанавливает позицию в потоке данных stream в значение pos. Она действует так же, как и функция fseek() с аргументом whence, равным SEEK_SET. В случае успеха эта функция возвращает 0. В противном случае она возвращает -1 и присваивает errno соответствующее значение. Эта функция (а также парная ей функция fgetpos()) предоставлена исключительно для работы с не относящимися к UNIX платформами, где для представления позиции в потоке данных используются сложные типы. При взаимодействии с такими платформами данная функция — единственный инструмент, позволяющий установить произвольную позицию в потоке данных, поскольку возможностей типа long языка C недостаточно. В специфичных для Linux приложениях этот интерфейс использовать нет необходимости, хотя и можно, если требуется обеспечить максимальную межплатформенную совместимость.

В стандартной библиотеке ввода-вывода также предоставляется функция быстрого доступа rewind():

```
#include <stdio.h>
void rewind (FILE *stream);
```

Вызов этой функции: rewind (stream); сбрасывает значение позиции обратно на начало потока данных. Код эквивалентен следующему:

```
fseek (stream, 0, SEEK_SET);
```

с оговоркой, что она также очищает индикатор ошибки.

Стоит оббратить внимание: функция rewind() не имеет возвращаемого значения и поэтому не может напрямую сообщать о возникающих ошибках. Если вызывающей стороне требуется убедиться в наличии ошибки, то понадобится очистить errno до вызова, а потом проверить, сохранила ли переменная после этого ненулевое значение. Например:

```
errno= 0;
rewind(stream);
if(errno)
        /* ошибка */
```

Получение актуальной позиции вп отокеданных. В отличие от lseek(), fseek() не возвращает обновленную позицию. Для этого предоставляется отдельный интерфейс. Функция ftell() возвращает текущую позицию из потока данных stream:

#include <stdio.h> long ftell (FILE *stream);

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

В рамках стандартного ввода-вывода предоставляется и альтернативная функция fgetpos():

#include <stdioh.h>

int fgetpos(FILE*stream, fpos_t*pos);

В случае успеха fgetposQвозвращает 0 и устанавливает текущую позицию потока данных stream в значение pos. При ошибке она возвращает -1 и присваивает errno соответствующее значение. Подобно fsetpos(), fgetpos() предоставляется лишь для работы с платформами не-UNIX, где для представления файловой позиции используются сложные типы.

1.11 Сброс потока данных

В стандартной библиотеке ввода-вывода есть интерфейс, позволяющий выписать содержимое пользовательского буфера в ядро. Он гарантирует, что все данные, записанные в поток данных, будут сброшены к ядру с помощью write(). Функция fflush() действует следующим образом:

#include <stdio.h>

int fflush (FILE *stream);

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

Чтобы понять принцип действия fflush(), следует понимать разницу между буфером, поддерживаемым библиотекой С, и буферизацией, выполняемой в самом ядре. Все вызовы, описанные в данной теме, работают с буфером, поддерживаемым библиотекой С.

Этот буфер располагается в пользовательском пространстве, и, следовательно, в нем работает пользовательский код, а не выполняются системные вызовы. Системный вызов выдается, только когда необходимо обратиться к диску или какому-нибудь другому носителю.

Функция fflush() просто записывает данные из пользовательского буфера в буфер ядра. Получается эффект, как будто пользовательская буферизация вообще не задействовалась и напрямую применяется вызов write(). В данной ситуации не гарантируется физическая отправка данных на тот или иной носитель — для полной уверенности в благополучной отправке данных потребуется использовать что-то вроде fsync(). В ситуациях, когда необходимо знать, что данные успешно отправлены в резервное хранилище, целесообразно вызвать fsync() непосредственно после fflush(). Таким образом, сначала необходимо убедиться, что информация из пользовательского буфера перенесена в буфер ядра, а потом гарантируется, что информация из буфера ядра попадет на диск.

1.12 Ошибки и конец файла

Некоторые интерфейсы стандартного ввода-вывода, например fread(), плохо справляются с информированием вызывающей стороны о сбоях, так как в них отсутствует механизм, позволяющий отличать ошибку от конца файла. При работе с такими вызовами, а также в некоторых других случаях бывает полезно проверить статус конкретного потока данных и определить, установлен ли на потоке данных stream индикатор ошибки:

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

Индикатор ошибки устанавливается другими интерфейсами стандартного ввода-вывода в ответ на возникновение условия ошибки. Функция возвращает ненулевое значение, если индикатор установлен, а в других случаях возвращает 0.

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

```
#include<stdio.h>
```

int feof(FILE*stream);

Индикатор EOF устанавливается другими интерфейсами стандартного ввода- вывода, когда достигается конец файла. Функция возвращает ненулевое значение, если индикатор установлен, а в других случаях возвращает 0.

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

#include <stdio.h>

void clearerr(FILE*stream);

Она не имеет возвращаемого значения, поэтому не может закончиться неудачно. Вызов clearerr() следует делать только после проверки индикаторов ошибок и EOF, так как после clearerr() они будут необратимо удалены. Например:

```
/* 'f' - допустимый поток данных */
if(ferror(f))
printf("Erroronf!\n"); if (feof (f))
printf ("EOF on f!\n");
clearerr (f);
```

1.13 Получение ассоциированного файлового дескриптора

Иногда предпочтительно получить файловый дескриптор, лежащий в основе конкретного потока данных. Например, может потребоваться выполнить применительно к потоку данных системный вызов. Это будет выполняться через файловый дескриптор потока данных, когда ассоциированной функции стандартного ввода-вывода не существует. Чтобы получить файловый дескриптор, лежащий в основе потока данных, используйте fileno():

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

В случае успеха fileno() возвращает файловый дескриптор, ассоциированный с потоком данных stream. В случае ошибки она возвращает -1. Это может произойти, только когда заданный поток данных является недопустимым: в такой ситуации функция устанавливает errno значение EBADF.

Как правило, не рекомендуется смешивать вызовы стандартного ввода-вывода и системные вызовы. При использовании fileno() программист должен проявлять осторожность и гарантировать, что

операции, в которых задействованы файловые дескрипторы, не будут конфликтовать с пользовательской буферизацией. В частности, целесообразно сначала сбрасывать поток данных, а потом начинать манипулировать лежащим в его основе файловым дескриптором. Практически ни в каких случаях не следует смешивать операции, связанные с файловыми дескрипторами и потоковым вводом-выводом.

1.14 Управление буферизацией

При стандартном вводе-выводе реализуется три типа пользовательской буферизации, и она предоставляет разработчикам интерфейс, позволяющий контролировать тип и размер буфера. Различные виды пользовательской буферизации применяются для разных целей и идеально подходят каждый для своей ситуации.

- Без буферизации. Пользовательская буферизация не применяется. Данные отправляются прямо в ядро. В таком случае нам приходится отказываться и от пользовательской буферизации, и от всех ее преимуществ, поэтому данный подход используется редко, с одним исключением: по умолчанию без буферизации работает стандартная ошибка.
- Построчная буферизация. Буферизация выполняется построчно. На каждом символе перехода на новую строку содержимое буфера отправляется в ядро. Буферизация строк целесообразна при работе с потоками данных, чей вывод попадает на экран, поскольку выводимые на монитор сообщения всегда разделяются по строкам. Следовательно, этот способ буферизации по умолчанию используется с потоками данных, подключенными к терминалам, — например, со стандартным выводом.
- Поблочная буферизация. Буферизация выполняется поблочно. В данном случае блок — это фиксированное количество байтов. Именно о таком типе буферизации мы говорили в начале этой главы, такая буферизация идеально подходит для работы с файлами. По умолчанию поблочная буферизация применяется со всеми потоками данных, которые ассоциированы с файлами.

В контексте стандартного ввода-вывода поблочная буферизация называется полной буферизацией.

В большинстве случаев заданный по умолчанию тип буферизации является оптимальным. Тем не менее стандартный ввод-вывод предоставляет интерфейс для управления типом применяемой буферизации:

#include <stdio.h>

int setvbuf (FILE *stream, char *buf, int mode, size_t size);

Функция setvbuf() устанавливает тип буферизации для потока данных stream в значение mode, которое может быть одним из следующих:

- _IONBF — без буферизации;
- _IOLBF — построчная буферизация;
- _IOFBF — поблочная буферизация.

За исключением значения _IONBF, при котором buf и size игнорируются, buf может указывать на буфер размером size байт. Стандартный ввод-вывод будет использовать этот буфер для данных указанного потока данных. Если buf равен NULL, то буфер указанного вами размера автоматически выделяется библиотекой glibc.

Функция setvbuf() может быть вызвана после открытия потока данных, но до того, как с ним будут выполняться какие-либо иные манипуляции. При успехе она возвращает 0, в другом случае — ненулевое значение.

Если буфер предоставлен, то он должен существовать и после закрытия потока данных. Распространенная ошибка — когда буфер определяется как автоматическая переменная в области видимости, заканчивающейся до закрытия потока данных. В частности, не следует создавать буфер, локальный для main(), так как впоследствии вы не удастся явно закрывать потоки данных. Например, следующий код содержит ошибку: #include <stdio.h>

```
int main (void)
{
        charbuf[BUFSIZ];
        /* задаем для stdin поблочную буферизацию с буфером BUFSIZ */
                setvbuf(stdout, buf, _IOFBF, BUFSIZ); printf ("Arrr!\n");
        return 0;
```

```
        /* 'buf' выходит из области видимости и высвобождается,
но stdout закрывается        уже позже */
    }
```

Чтобы избежать ошибок такого типа, нужно явно закрывать поток данных до выхода из этого контекста либо делать buf глобальной переменной.

Как правило, разработчику не приходится задумываться о буферизации потоков. Если не считать стандартных ошибок, на всех терминалах используется построчная буферизация, и это имеет смысл. При работе с файлами используется поблочная буферизация, и это тоже верно. При поблочной буферизации задаваемый по умолчанию размер буфера равен BUFSIZ. Он определяется в <stdio.h>, и выбранное для него значение обычно также оптимальное (представляет собой кратное типичному размеру блока).

1.15 Безопасность программных потоков

Программные потоки (threads) — это рабочие единицы внутри процесса. Большинство процессов содержат всего один программный поток. Тем не менее процессы могут включать в себя и множество программных потоков, в каждом из которых выполняется своя задача. Такой процесс называется многопоточным. Многопоточный процесс можно представить как совокупность мелких процессов, делящих общее адресное пространство. Без явной координации программные потоки могут запускаться когда угодно и перемежаться друг с другом всевозможными способами. В многопроцессорной системе два и более программных потока, относящихся к одному процессу, могут даже работать параллельно. Программные потоки могут перезаписывать совместно используемые данные, если разработчик специально не организует синхронизацию доступа к данным (такая практика называется блокировкой) либо не сделает данные локальными для программного потока (в этом случае принято говорить о привязке к потоку).

Операционные системы, поддерживающие программные потоки, также предоставляют механизмы блокировки (специальные программные конструкторы, обеспечивающие взаимное исключение). Блокировка нужна, чтобы разные программные потоки не «наступали

друг другу на пятки». Эти механизмы используются при стандартном вводе-выводе. В результате гарантируется, что отдельные программные
потоки, относящиеся к одному процессу, могут выполнять параллельные вызовы стандартного ввода-вывода — даже по отношению к одному и тому же потоку данных. При этом параллельные операции нисколько не мешают друг другу. Правда, этих механизмов иногда оказывается недостаточно, например, если требуется заблокировать группу вызовов, расширив таким образом критический участок кода (так называется фрагмент кода, работающий без всякой интерференции со стороны другого программного потока) с одной операции до нескольких. В других ситуациях может понадобиться вообще избавиться от блокировок для повышения эффек- тивности.

Функции стандартного ввода-вывода по определению являются потокобезопасными. На внутрисистемном уровне они ассоциируют с каждым открытым потоком данных саму блокировку, подсчет блокировок и владеющий программный поток. Любой программный поток должен получить блокировку и стать владеющим, лишь после этого он сможет выдавать какие-либо запросы на ввод-вывод. Два и более программных потока, оперирующих одним и тем же потоком данных, не могут перемежать операции стандартного ввода-вывода, поэтому в контексте отдельных вызовов функций операции стандартного ввода-вывода являются атомарными.

Разумеется, на практике многие приложения требуют большей атомарности, чем на уровне отдельных вызовов функций. Допустим, что множественные программные потоки, работающие в одном процессе, выдают запросы на запись. Функции стандартного ввода-вывода являются потокобезопасными, поэтому отдельные операции записи не будут накладываться друг на друга и приводить к искажению вывода. Таким образом, даже если два программных потока одновременно выдают запрос на запись, блокировка гарантирует, что один из этих запросов завершится раньше другого. Однако, что, если процессу требуется выдать несколько запросов на запись подряд и исключить риск пересечения с запросами на запись, идущими от другого программного потока, не только для отдельного запроса, но и для всей этой последовательности запросов? Для таких

нужд в стандартном вводе-выводе предоставляется семейство функций для индивидуальных операций с блокировкой, ассоциированной с конкретным потоком данных.

Функция flockfile() дожидается, пока не будет снята блокировка с потока данных stream, увеличивает количество блокировок на единицу, а потом получает блокировку и становится владеющим программным потоком этого потока данных. Затем она возвращается:

```
#include <stdio.h>
void flockfile (FILE *stream);
```

Функция funlockfile() уменьшает количество блокировок, ассоциированное с потоком данных stream:

```
#include <stdio.h>
void funlockfile (FILE*stream);
```

Если количество блокировок достигает нуля, то текущий программный поток прекращает владеть соответствующим потоком данных. Теперь блокировку сможет получить другой программный поток.

Такие вызовы могут быть вложенными. Это означает, что отдельно взятый программный поток может выдавать множественные вызовы flockfile() и поток данных не разблокируется, пока процесс не выдаст ровно такое же количество вызовов funlockfile().

Функция ftrylockfile() представляет собой неблокирующую разновидность flockfile():

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

Если поток данных stream в конкретный момент заблокирован, то ftrylockfile() ничего не делает и сразу же возвращает ненулевое значение. Если поток данных stream в этот момент не заблокирован, функция получает блокировку, увеличивает количество блокировок, становится владеющим программным потоком потока данных stream и возвращает 0.

Допустим, необходимо записать несколько строк в файл и гарантировать, что при записи они не будут перемежаться с операциями записи, идущими от других потоков:

```
flockfile (stream);
fputs ("List of treasure:\n", stream);
fputs (" (1) 500 gold coins\n", stream);
```

```
fputs (" (2) Wonderfully ornate dishware\n", stream);
funlockfile (stream);
```

Хотя отдельные операции fputs() никогда не вступают в условия гонки с другими операциями ввода/вывода — например, никогда не возникнет пересечения чего бы то ни было с List of treasure, — в работу такой функции может вклиниться другая операция стандартного ввода-вывода, поступающая от другого программного потока к тому же потоку данных. Это может происходить в промежутке между двумя вызовами fputs(). В идеальном случае приложение должно быть построено так, чтобы множественные программные потоки не направляли ввод-вывод в один и тот же поток данных. Если же в приложении такая отправка необходима, но при этом нужен и атомарный регион шире одной функции, то flockfile() со товарищи вам очень пригодятся.

Есть и другая причина, по которой может потребоваться блокировать потоки данных вручную. Обеспечив более тонкий и точный контроль блокировок (это под силу только системному программисту), мложно минимизировать издержки, связанные с блокировками, и улучшить таким образом производительность. Для этой цели в Linux предоставляется семейство функций, родственных стандартным интерфейсам ввода-вывода. Функции из этого семейства отличаются тем, что вообще не выполняют блокировок. Фактически это неблокируемые аналоги функций стандартного ввода-вывода:

```
#define _GNU_SOURCE
#include <stdio.h>
int fgetc_unlocked (FILE *stream);
char *fgets_unlocked (char *str, int size, FILE *stream);
size_t fread_unlocked (void *buf,  size_t size, size_t nr,    FILE
*stream);
int fputc_unlocked (int c, FILE *stream);
int fputs_unlocked (const char *str, FILE *stream);
size_t fwrite_unlocked (void *buf, size_t size, size_t nr,    FILE
*stream);
int fflush_unlocked (FILE *stream);
int feof_unlocked (FILE *stream);
int ferror_unlocked      (FILE *stream);
int fileno_unlocked (FILE *stream);
```

void clearerr_unlocked (FILE *stream);

Все эти функции работают идентично своим блокируемым аналогам, но не проверяют наличия блокировок и не получают блокировку, ассоциированную с конкретным потоком stream. Если блокировка необходима, то именно программист должен вручную обеспечить ее получение и последующее высвобождение.

При использовании неблокируемых стандартных функций ввода-вывода значительно повышается производительность. Более того, код радикально упрощается, если не приходится беспокоиться о блокировании сложных операций с помощью flockfile().

Хотя в POSIX и определяется несколько неблокируемых вариантов функций стандартного ввода-вывода, ни одна из вышеприведенных функций в POSIX не описана. Все они специфичны для Linux, хотя это подмножество и поддерживается в некоторых других системах UNIX.

1.16 Недостатки стандартного ввода-вывода

Как ни часто используется стандартный ввод-вывод, часть экспертов указывают на определенные его недочеты. Некоторые функции, например fgets(), бывают неэффективными или плохо сработанными. Другие функции, к примеру gets(), настолько ужасны, что их практически изгнали из всех стандартов.

Самые серьезные претензии к стандартному вводу-выводу связаны с его негативным влиянием на производительность, которое объясняется необходимостью двойного копирования. При считывании данных стандартный ввод-вывод направляет к ядру системный вызов read(), копируя данные из ядра в буфер стандартного ввода-вывода. Если после этого приложение с помощью стандартного ввода- вывода делает запрос на запись, применяя, скажем, fgetc(), то данные копируются вновь — на этот раз из буфера стандартного ввода-вывода в указанный буфер. Запросы на запись работают прямо противоположным образом: сначала данные копируются из предоставленного буфера в буфер стандартного ввода/вывода, а оттуда — в ядро посредством вызова write().

Альтернативная реализация позволяет избежать двойного копирования. Для этого нужно при каждом запросе на считывание

возвращать указатель на буфер стандартного ввода-вывода. После этого данные можно считывать напрямую внутри буфера стандартного ввода-вывода без необходимости дополнительного копирования. Если приложению действительно потребуются данные из его собственного локального буфера — например, чтобы что-то дописать к ним, — то мы всегда можем организовать копирование вручную. Такая реализация предоставляла бы «свободный» интерфейс, позволяющий приложениям сигнализировать о завершении обработки конкретного фрагмента кода из буфера считывания.

Запись протекала бы несколько сложнее, но двойного копирования по-прежнему не было бы. При выдаче запроса на запись эта реализация требовала бы использовать указатель. В конечном итоге, когда данные были бы готовы к сбросу в ядро, такая реализация могла пройти по всему списку сохраненных указателей, записывая данные. Это можно сделать посредством фрагментирующего/дефраг- ментирующего ввода-вывода, для которого требуется лишь один системный вызов writev().

1.17 Задание для самостоятельно работы

Используя основные принципы буферезированного и стандартного файлового ввода вывода, разработать программу на языке программирования С для работы с файлами в соотвествии с индивидуальным вариантом. Индивидуальный вариант взять у преподавателя.

Контрольные вопросы

1) Что такое блок, с точки хрения файловой системы?
2) Как от размера блока зависит скорость записи информации в файл?
3) Какие стандартные размеры блоков применяются на практике?
4) Какие правила регулируют определение размера бока?
5) Что такое беферезированный файловый ввод-вывод?
6) Какая функция в стандартном файловом вводе-выводе прежназначена для открытия файла?

7) В каком заголовочном файле соержится перечесление функций стандартного ввода-вывода?
8) Какие уровни доступа предусмотрены в стандартном файловом вводе-выводе?
9) Каким образом осуществляется считывание из потока данных?
10) Для чего прежназначена функция ungetc?
11) Для чего предназначена функция fgets?
12) Что возвращает функция fread после выполнения?
13) С помощью каких функций в стандартном вводе-выводе осуществляется запись информации в файл?
14) Каким образом осуществляется поззиционирование в потоке данных?
15) Каким образом осуществляется сброс потока данных?
16) Каким образом осуществляется управление буферизацией?

Тема 2. Практическое управление процессами

Процессы — это одна из самых фундаментальных абстракций в системах UNIX после файлов. Если рассматривать процессы как объект выполняемого кода — живой, работающей, активной программы, — они являются чем-то большим, нежели просто язык ассемблера, и состоят из данных, ресурсов, состояния и виртуального процессора.

Основы не претерпели значительных изменений с первых дней существования UNIX. Именно здесь, в управлении процессами, ярче всего проявляются дальновидность и долговечность первоначальной концепции UNIX. История системы интересна и весьма незаурядна, так как в концепции UNIX создание нового процесса отделяется от загрузки нового двоичного образа. Хотя в большинстве случаев эти действия выполняются совместно, разделение позволяет экспериментировать и открывает широкие возможности для развития каждой задачи. Таким образом, эта редко встречающаяся концепция сохранилась до наших дней, в то время как большинство операционных систем предлагают один системный вызов для запуска одной программы, в UNIX требуются два: fork и exec.

2.1 Программы, процессы и потоки

Бинарный модуль — это компилированный, исполняемый код, находящийся в каком-либо хранилище данных, например на диске. Следовательно большие и значительные бинарные модули можно называть приложениями. И /bin/ls, и /usr/bin/Xll являются бинарными модулями.

Процессом является запущенная программа. Процесс включает в себя бинарный образ, загружаемый в память, и многое другое: подгрузку виртуальной памяти, ресурсы ядра, например открытые файлы, выполнение требований по безопасности (к примеру, выбор определенного пользователя), а также запуск одного или нескольких потоков. Поток — это одно из действий внутри процесса. Каждый поток имеет собственный виртуализированный процессор, включающий в себя стек, состояние процессора, например регистры, а также командные указатели.

Если в процессе поток только один, то процесс и является потоком. У него только один экземпляр виртуальной памяти и один виртуализированный процессор. В многопоточных процессах потоков несколько. Виртуализация памяти связана с процессом, поэтому все потоки одновременно используют одно и то же адресное пространство памяти.

2.2 Идентификатор процесса

Каждый процесс обозначается уникальным идентификатором (process ID, обычно сокращается до pid). Pid обязательно является уникальным в любой конкретный момент времени. Это значит, что в момент времени t+0 может быть только один процесс с pid 770 (или ни одного процесса с таким значением идентификатора), но нельзя гарантировать, что в момент времени t+1 не будет существовать другого процесса с тем же идентификатором pid 770. На практике, впрочем, ядро не обязательно быстро меняет местами идентификаторы процессов — предположение, являющееся, как вы скоро увидите, весьма небезопасным. Конечно, с точки зрения самого процесса, его pid остается неизменным.

Процесс бездействия, или пассивный, который выполняется ядром в отсутствие всех других процессов, имеет pid 0. Первый процесс, который ядро выполняет во время запуска системы, называется процессом инициализации и имеет pid 1. Обычно init process в Linux является программой инициализации. Используется термин «инициализация» для обозначения и начального процесса, запускаемого при загрузке, и специальной программы, используемой для этих целей.

Если пользователь не указывает ядру напрямую, какой процесс следует запускать (с помощью init в командной строке), ядро выбирает подходящий процесс инициализации самостоятельно — нечастый случай, когда политику диктует ядро. Ядро Linux перебирает четыре исполняемых модуля в следующем порядке.

1. /sbin/init — предпочтительное и наиболее вероятное размещение процесса инициализации.
2. /etc/init — следующее наиболее вероятное размещение процесса инициализации.

3. /bin/init — резервное размещение процесса инициализации.
4. /bin/sh — местонахождение оболочки Bourne, которую ядро пытается запустить, если найти процесс инициализации не удалось.

Первый из этих процессов, который будет существовать, и запустится в качестве процесса инициализации. Если не удалось запустить ни один из четырех, то ядро Linux переводит систему в состояние «паники».

После запуска процесс инициализации отрабатывает оставшуюся часть загрузки. Как правило, в нее включены инициализация системы, запуск различных сервисов и программы авторизации.

2.3 Выделение идентификатора процесса

По умолчанию ядро может выставить максимальную величину идентификатора, равную 32 768. Это объясняется необходимостью совместимости с более старыми системами UNIX, которые использовали 16-битные типы данных для идентификаторов. Администратор системы может установить большую величину через /proc/sys/kernel/pid_max, выделив большее пространство для pid, но снизив таким образом совместимость.

Идентификаторы назначаются ядром строго линейно. Если в настоящий момент наибольший имеющийся pid равен 17, то следующей будет назначена величина 18, даже если процесс с последним назначенным идентификатором 17 уже не выполняется во время старта нового процесса. Ядро не назначает использованные ранее идентификаторы процессов, пока не пройдет верхнее значение, то есть меньшие значения не будут устанавливаться, пока не будет достигнута величина, записанная в /proc/sys/kernel/ pid_max. Таким образом, Linux не гарантирует уникальность идентификаторов процессов на длинные периоды, но тем не менее есть некоторая уверенность, что в течение коротких отрезков времени идентификаторы будут стабильными и уникальными.

2.4 Иерархия процессов

Процесс, запускающий другой процесс, называется родительским; новый процесс, таким образом, является дочерним. Каждый процесс запускается каким-либо другим процессом (кроме, разумеется, процессов инициализации). Таким образом, каждый дочерний процесс имеет «родителя». Эти взаимоотношения записаны в каждом идентификаторе родительского процесса (ppid), значение которого для дочернего процесса равно значению pid родительского процесса.

Каждый процесс принадлежит определенному пользователю и группе. Эти принадлежности используются для управления правами доступа к ресурсам. С точки зрения ядра пользователь и группа — это просто некие целочисленные величины. Они хранятся в файлах /etc/passwd и /etc/group, с помощью которых сопоставляются с привычными глазу пользователя UNIX именами, воспринимаемыми человеком, например имя пользователя root или группа wheel (в общем случае ядро Linux никак не взаимодействует с этими строками, оперируя с объектами посредством целочисленных величин). Каждый дочерний процесс наследует пользователя и группу, которым принадлежал родительский процесс.

Каждый процесс является также частью группы процессов, которая означает, по сути, его отношение к другим процессам (не следует путать группу процессов с упомянутыми выше пользователем и группой). Дочерние процессы, как правило, принадлежат к тем же группам процессов, что и родительские. Кроме того, когда пользователь запускает конвейер (например, введя ls | less), все команды в конвейере становятся членами одной и той же группы процессов. Понятие группы процессов упрощает отправку сигналов или получение информации от всего конвейера, так как все дочерние процессы находятся в конвейере. С точки зрения пользователя группа процессов тесно связана с понятием задания.

2.5 Идентификатор процесса pid_t

С точки зрения программирования идентификатор процесса обозначается типом pid_t, величина которого определяется в

заголовочном файле <sys/types.h>. Конкретный тип C зависит от архитектуры и не определяется каким-либо стандартом C. В Linux, однако, для pid_t чаще всего используется тип данных C int.

Получение идентификаторов процесса и родительского процесса

Системный вызов getpid() возвращает идентификатор вызывающего процесса:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid (void);
```

Системный вызов getppid() возвращает идентификатор родителя вызывающего процесса:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid (void);
```

Ни один из них не может вернуть ошибку, поэтому использование данных вызовов несложно:

```
printf ("My pid=%jd\n", (intmax_t) getpid ()); printf ("Parent's pid=%jd\n", (intmax_t) getppid ());
```

Здесь приводится возвращенная величину к типу intmax_t, который является типом C/C++, гарантированно способным хранить в системе любое целое число со знаком. Другими словами, он равен любым другим типам для целых чисел со знаками или больше их. В сочетании с printf() с модификатором (%j) такой подход позволяет нам безопасно печатать переменные типа integer, назначенные typedef. До появления intmax_t не существовало приемлемого способа сделать это (если в вашей системе нет intmax_t, вы можете предположить, что pid_t принадлежит к int, что правдиво для большинства систем UNIX).

2.6 Запуск нового процесса

В UNIX действие загрузки в память и запуска образа программы выполняется отдельно от операции по созданию нового процесса. Один системный вызов загружает бинарную программу в память, замещая текущее содержание адресного пространства, и начинает выполнение новой программы. Это называется выполнением новой программы, а функциональность обеспечивается семейством вызовов exec.

Другой системный вызов используется для создания нового процесса, который изначально является практически копией своего родительского. Часто новый процесс немедленно приступает к выполнению новой программы. Акт создания нового процесса называется ветвлением и обеспечивается системным вызовом fork(). Два действия — сначала ветвление для создания нового процесса, а затем exec для открытия нового выполнения этого процесса — требуются для запуска новой программы в новом процессе.

2.7 Семейство вызовов exec

Единой функции exec не существует; на одном системном вызове построено целое семейство таких функций. Сначала рассмотрим самый простой из этих вызовов, execl():

```
#include<unistd.h>
int execl (const char *path, const char *arg,
...);
```

Вызов execl() замещает текущий образ процесса новым, загружая в память программу, определенную path. Параметр arg — первый аргумент этой программы. Многоточие означает переменное количество аргументов — у функции execl() их количество может быть любым, дополнительные аргументы можно указывать в скобках один за другим. Список аргументов всегда завершается значением NULL.

Например, следующий программный код замещает выполняющуюся в настоящий момент программу с /bin/vi:

```
int ret;
ret = execl ("/bin/vi", "vi", NULL);
if (ret == -1)
        perror ("execl");
```

Следуя соглашениям UNIX, передается в качестве первого аргумента программы значение "vi". Оболочка помещает последний компонент пути, то есть "vi", в первый аргумент во время ветвления или запуска процессов, благодаря чему программа может проверить первый аргумент argv[0] для выяснения имени двоичного образа. В большинстве случаев несколько системных утилит, отображающихся пользователю под разными именами, в

действительности представляют собой единую программу с жестко прописанными ссылками к различным именам. Программа использует первый аргумент, чтобы определить свое поведение.

Другой пример. Если нужно редактировать файл /home/kidd/hooks.txt, то необходимо запустить следующий код:

```
int ret;
ret = execl ("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret == -1)
        perror ("execl");
```

Как правило, execl() не возвращает никаких значений. Успешный вызов завершается переходом к входной точке новой программы, и только что выполненный код больше не находится в процессном адресном пространстве. Если произошла ошибка, execl() возвращает -1 и устанавливает errno для обозначения проблемы.

В случае успешного выполнения вызов execl() изменяет не только адресное пространство и образ процесса, но и некоторые другие атрибуты процесса:

- любые ожидающие сигналы исчезают;
- любые сигналы, отлавливаемые процессом , возвращаются к своему поведению по умолчанию, поскольку обработчиков сигналов больше нет в процессном адресном пространстве;
- все блокировки памяти удаляются;
- большинство атрибутов потока возвращается к значениям по умолчанию;
- большая часть статистических данных процесса сбрасывается;
- все адресное пространство памяти, относящееся к данному процессу, включая загруженные файлы, очищается;
- все, находящееся исключительно в пользовательском пространстве, включая функциональности библиотеки С, например поведение atexit(), удаляется.

Некоторые свойства процесса, однако, не изменяются. Например, идентификатор (свой и родительский), приоритет, а также пользователь и группа остаются теми же.

Обычно при работе системных вызовов семейства exec наследуются и открытые файлы: у запущенных программ сохраняется полный доступ ко всем файлам, открытым в изначальном процессе, если им известны начения дескрипторов. Однако зачастую это

нежелательно. Обычной практикой является закрытие файлов перед запуском exec, но возможно дать команду ядру делать это автоматически через fcntl().

Кроме execl(), в семействе есть еще пять членов:

```
#include <unistd.h>
int execlp (const char *file, const char *arg,
...);
int execle (const char *path,
const char *arg, char * const envp[]);
int execv (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execve (const char *filename, char *const argv[], char *const
envp[]);
```

Запомнить все очень просто. l и v указывают, передаются ли аргументы списком или массивом. Символ p указывает, что система будет искать указанный файл по полному пользовательскому пути. В командах, где используются варианты с p, можно указать только имя файла, если он находится в пределах пользовательского пути. Наконец, e обозначает, что для нового процесса создается новое окружение. Интересно, что, хотя технических ограничений для этого не существует, в семействе exec нет элемента, позволяющего и искать путь к файлам, и создавать новое окружение. Возможно, это объясняется тем, что варианты p предназначены для использования оболочками, а процессы, исполняемые в оболочках, как правило, наследуют свое окружение от них.

В следующем фрагменте кода используется execvp() для выполнения vi, как и в предыдущем варианте, учитывая, что vi находится на пользовательском пути:

```
int ret;
ret = execvp ("vi", "vi", "/home/kidd/hooks.txt", NULL); if (ret == -
1)
perror ("execvp");
```

Программы установки идентификаторов группы и пользователя (Set-group-ID и set-user-ID) — это процессы, запускаемые от имени группы или пользователя, которым принадлежит их бинарный код, а не группы или пользователя, которые их вызывают. Следовательно, они не должны ссылаться на оболочку или операции, которые, в свою

очередь, затрагивают оболочку. Это действие приведет к появлению уязвимого места в безопасности, так как вызывающий пользователь может установить переменные окружения и управлять оболочкой. Самая частая форма такой атаки — внедрение пути, когда атакующий указывает переменной PATH заставить процесс выполнить функцию execl() и запустить любой двоичный код по выбору злоумышленника, в результате чего последний может запустить любую программу под учетными данными set-group-ID и set-user-ID.

Элементы семейства exec, принимающие в качестве аргумента массив, работают точно так же, за одним исключением — вместо списка генерируется и передается массив. Использование массива позволяет определять аргументы во время выполнения программы. Как и в случае аргумента в виде списка переменной длины, массив должен заканчиваться значением NULL.

В следующем примере execv() используется для запуска vi, как мы уже делали ранее:

```
const char *args[] = { "vi", "/home/kidd/hooks.txt", NULL };
int ret;
ret = execv ("/bin/vi", args);
if (ret == -1)
        perror ("execvp");
```

В Linux только один элемент семейства exec является системным вызовом. Остальные — это оболочки системного вызова в библиотеке C. Системные вызовы с переменным количеством аргументов сложны в реализации, а концепция пользовательского пути существует только в пользовательском пространстве, единственным элементом, способным осуществлять функции системного вызова, является execve(). Прототип системного вызова идентичен пользовательскому вызову.

В случае успешной работы системные вызовы exec не возвращают результатов. При неудаче вызов возвращает -1 и присваивает errno одно из следующих значений:

- E2BIG — общее количество байтов в предоставленном списке аргументов (arg) или окружения (envp) слишком велико;
- EACCES — процесс не имеет доступа к компонентам пути, указанным в аргументе path; файл path является некорректным;

файл назначения не является исполняемым; в файловой системе, содержащей path или file, активно свойство noexec;

- EFAULT — дан недопустимый указатель;
- EIO — низкоуровневая ошибка ввода-вывода (это плохо);
- EISDIR — конечный пункт пути path, или интерпретатор, является каталогом;
- ELOOP — путь path содержит слишком много символьных ссылок;
- EMFILE — вызывающий процесс достиг предела ограничения количества открытых файлов;
- ENFILE — система достигла предела ограничения количества открытых файлов;
- ENOENT — цели path или file не существует либо отсутствует необходимая общая библиотека;
- ENOEXEC — цель path или file является недопустимым двоичным файлом или же предназначена для другой машинной архитектуры;
- ENOMEM — доступных ресурсов ядра недостаточно для выполнения другой программы;
- ENOTDIR — какой-либо из компонентов пути, кроме конечного, не является каталогом;
- EPERM — в файловой системе, в которой находятся path или file, активно свойство nosuid, пользователь не имеет прав root либо для path или file установлен бит suid или sgid;
- ETXTBSY — назначение path или file открыто для записи другим пользователям.

2.8 Системные вызовы fork()

Новый процесс, запускающий тот же системный образ, что и текущий, может быть создан с помощью системного вызова fork():

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

В случае успешного обращения к fork() создается новый процесс, во всех отношениях идентичный вызывающему. Оба

процесса выполняются от точки обращения к fork(), как будто ничего не происходило.

Новый процесс является дочерним по отношению к вызывающему, который, в свою очередь, называется родительским. В дочернем процессе успешный запуск fork() возвращает 0. В родительском fork() возвращает pid дочернего. Родительский и дочерний процессы практически идентичны, за исключением некоторых особенностей:

- pid дочернего процесса, конечно, назначается заново и отличается от родительского;
- родительский pid дочернего процесса установлен равным pid родительского процесса;
- ресурсная статистика дочернего процесса обнуляется;
- любые ожидающие сигналы прерываются и не наследуются дочерним процессом;
- никакие вовлеченные блокировки файлов не наследуются дочерним процессом.

В случае ошибки дочерний процесс не создается, fork() возвращает -1, устанавливая соответствующее значение errno. Вот два возможных значения errno и их смысл:

- EAGAIN — ядро не способно выделить определенные ресурсы, например новый pid, или достигнуто ограничение по ресурсам RLIMIT_NPROC;
- ENOMEM — недостаточно ресурсов памяти ядра, чтобы завершить запрос.

Использование очень простое:

```
pid_t pid;
pid = = fork ();
if (pid > 0)
        printf ("Я родительский процесс cpid=%d!\n", pid)
else if (!pid)
        printf ("А я дочерний!^");
else
if (pid == = -1)
        perror ("fork");
```

Чаще всего системный вызов fork() используется для создания нового процесса и последующей загрузки в него нового двоичного

образа. Представим себе оболочку, в которой пользователь запускает новое приложение или процесс начинает вспомогательную программу. Сначала процесс ответвляет новый процесс, а потом дочерний процесс создает новый двоичный образ. Сочетание fork и exec используется часто и без осложнений. В следующем примере ответвляется новый процесс, запускающий двоичный файл /bin/windlass:

```
pid_t pid;
pid = fork ();
if (pid == -1)
        perror ("fork");
/* дочерний ... */
if (!pid) {
        const char *args[] = { "windlass", NULL }; int ret;
        ret = execv ("/bin/windlass", args); if (ret == -1) {
        perror ("execv"); exit (EXIT_FAILURE);
        }
}
```

Родительский процесс продолжает выполняться, как и ранее, за исключением появления у него нового дочернего. Вызов execv() влияет только на дочерний процесс, заставляя его выполнить программу /bin/windlass.

В ранних версиях UNIX ветвление было очень простым, если не сказать примитивным. После вызова ядро создавало копии всех структур внутренних данных, дублировало записи таблиц страницы процесса, а затем выполняло постраничное копирование родительского адресного пространства в дочернее. К сожалению, постраничное копирование — весьма затратный по времени процесс.

В современных системах UNIX подход усовершенствован. Вместо копирования всего объема родительского адресного пространства в современных системах UNIX, в том числе Linux, используются страницы копирования при записи (COW).

Копирование при записи — это откладывающая стратегия оптимизации, разработанная для уменьшения нагрузки из-за дублирования процессов. Принцип весьма прост: если несколько пользователей запрашивают доступ для чтения их собственных копий ресурса, нет смысла дублировать создаваемые копии. Вместо этого

каждый потребитель может получить указатель к одному и тому же ресурсу. Пока пользователи не пытаются изменить свою «копию» ресурса, сохраняется иллюзия эксклюзивного доступа к ресурсу и затрат на копирование не требуется. Если пользователь пытается редактировать свою копию ресурса, то в этот момент ресурс прозрачно дублируется и копия отправляется редактирующему пользователю. Потребитель, не видя происходящего, может изменять свою копию ресурса, пока другие продолжают просматривать оригинальную, неизмененную версию. Отсюда и название: копирование происходит только при записи.

Основная выгода здесь заключается в том, что, если пользователь не пытается модифицировать свою копию ресурса, она и не требуется. Общее преимущество откладывающих алгоритмов — откладывание наиболее затратных действий на последний момент — работает и здесь.

В определенном примере виртуальной памяти копирование при записи реализуется по постраничному принципу. Таким образом, поскольку процесс не затрагивает все адресное пространство целиком, копия его всего и не требуется. По окончании ветвления и родительский, и дочерний процессы «думают», что каждый из них имеет свое уникальное адресное пространство, в то время как на самом деле они делят доступ к оригинальным страницам родителя — которые, в свою очередь, могут предоставлять доступ к себе другим родительским или дочерним процессам, и т. д.

Реализация на уровне ядра проста. Страницы данных, относящиеся к структуре ядра, помечены как доступные только для чтения и копируемые только при записи. Когда какой-либо процесс пытается модифицировать страницу, на ней происходит ошибка, которую затем обрабатывает ядро, создавая видимую копию страницы. В этот момент для страницы атрибут «копирование при записи» для страницы удаляется и доступ к ней больше не делится. Поскольку архитектура современных компьютеров поддерживает копирование при записи на уровне аппаратного обеспечения, в элементах, управляющих памятью (MMUs), весь процесс достаточно прост и внедряется без проблем.

Копирование при записи обладает еще одним преимуществом в процессе ветвления. Поскольку после ветвления чаще всего

выполняется exec, копирование родительского адресного пространства в дочернее представляет собой пустую трату времени: ведь если дочерний процесс сразу же приступает к выполнению нового двоичного образа, его адресное пространство немедленно очищается. Копирование при записи избавляет и от этого неудобства.

2.9 Системный вызов vfork()

До внедрения страниц, поддерживающих копирование при записи, разработчики UNIX были вынуждены реализовывать бесполезное копирование адресного пространства в течение ветвления, сразу за которым следовало выполнение exec. В связи с этим разработчики BSD представили в 3.0BSD системный вызов, который называется vfork:

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork (void);
```

Успешное исполнение vfork() работает так же, как и fork(), кроме того, что дочерний процесс должен немедленно успешно вызвать одну из функций exec или завершиться, вызвав _exit().

Вызов vfork() не выполняет копирования адресного пространства и таблиц страниц, относящихся к родительскому процессу, пока дочерний не завершится или не выполнит новый двоичный образ. Между этим родительский и дочерний процессы совместно используют — без семантики копирования при записи — свое адресное пространство и страницы с табличными записями. Фактически единственная работа, выполняемая vfork(), — дублирование внутренней структуры данных ядра. Следовательно, дочерний процесс не должен модифицировать никаких данных в памяти адресного пространства.

Системный вызов vfork() является архаизмом и не должен реализовываться в Linux, хотя, надо отдать должное, даже с учетом копирования при записи vfork() работает быстрее fork(), так как отпадает необходимость в самом копировании страниц1. В любом случае появление страниц с поддержкой копирования при записи, любые аргументы в пользу альтернатив fork() утрачивают силу. Действительно, до появления ядра Linux 2.2.0 vfork() был просто

оболочкой для fork(). Требования для vfork() менее строгие, чем для fork(), поэтому данная реализация вполне осуществима.

Строго говоря, ни одно внедрение vfork() не застраховано от ошибок. Представьте, что вызов exec завершился сбоем: родительский процесс будет находиться в заблокированном состоянии, пока дочерний процесс не найдет способа разрешить эту ситуацию или не завершится. В программах лучше использовать fork() напрямую.

2.10 Завершение процесса

POSIX и C89 определяют следующую стандартную функцию для завершения текущего процесса:

```
#include <stdlib.h>
void exit (int status);
```

Вызов exit() выполняет некоторые основные шаги перед завершением, а затем отправляет ядру команду прекратить процесс. Эта функция не может вернуть ошибку — по сути, она вообще не возвращает никаких результатов. Следовательно, нет смысла в каких-либо других инструкциях по выполнению вызова exit() .

Параметр status используется для обозначения статуса процесса завершения. Другие программы — как и пользователь оболочки — могут проверять эту величину. В частности, статус status & 0377 возвращается родительскому процессу. Далее в этой главе мы рассмотрим, как извлекать возвращаемое значение.

Значения EXIT_SUCCESS и EXIT_FAILURE определяются в качестве способов представления успеха и неудачи. В Linux значение 0 обычно означает успех, а любое ненулевое значение, например -1 или 1, соответствует неудаче.

Следовательно, успешный выход — единственная строка: exit (EXIT_SUCCESS);

Перед тем как прервать процесс, библиотека C выполняет подготовительные шаги в следующем порядке.

1. Вызов всех функций, зарегистрированных с atexit() или on_exit(), в порядке, обратном порядку регистрации (мы еще поговорим о них позже в этой главе).
2. Сброс всех стандартных потоков ввода-вывода (см. гл. 3)

3. Удаление всех временных файлов, созданных функцией tmpfile().

Эти шаги завершают всю работу, которую процесс должен проделать в пользовательском пространстве, после чего exit() выполняет системный вызов _exit(), позволяющий ядру обработать оставшуюся часть завершения процесса:

#include <unistd.h> void _exit (int status);

Когда процесс завершается, ядро очищает все ресурсы, которые были выделены для нужд процесса и более не используются. К ним относятся выделенная память, открытые файлы, семафоры System V и др. После очистки ядро уничтожает процесс и предупреждает родительский процесс о завершении дочернего.

Приложения могут вызывать _exit() напрямую, но зачастую это лишено смысла: большинству приложений необходимо провести некоторую зачистку перед полным выходом, например прервать поток stdout.

Стремясь к чрезмерной точности, стандарт ISO C99 добавил функцию _Exit(), поведение которой полностью идентично _exit():

#include <stdlib.h> void _Exit (int status);

Классический способ прекратить работу программы — не использование явного системного вызова, а простое «достижение конечной точки» программы. Для C или C++, например, это происходит, когда результат возвращает функция main(). Подход «достижение конечной точки» тем не менее все равно совершает системный вызов: компилятор просто вставляет неявный системный вызов exit() после завершения собственного кода. Очень полезно возвращать статус выхода явно, через exit() или возвращая какую-либо величину из main(). Оболочка использует величину exit для определения успешного либо неуспешного исполнения команд. Успешное значение — exit(0) или возвращение 0 из main().

Процесс также может завершиться, если ему отправлен сигнал, действие которого по умолчанию — окончание процесса. К таким сигналам относятся SIGTERM и SIGKILL.

Последний способ прервать выполнение программы — срабатывание защитных функций ядра. Ядро может прервать процесс, выполняющий недопустимые инструкции, нарушающий сегментацию, исчерпавший ресурсы памяти и т. д.

POSIX 1003.1-2001 определяет, а Linux поддерживает библиотечный вызов atexit(), используемый для регистрации функций, вызываемых при завершении процесса:

```
#include <stdlib.h>
int atexit (void (*function)(void));
```

При успешном срабатывании atexit() регистрирует указанную функцию для запуска при нормальном завершении процесса, то есть с помощью системного вызова exit() или возврата результатов функцией main(). Если процесс запускает функцию exec, список зарегистрированных функций очищается (поскольку функции больше не существуют в новом адресном пространстве процесса). Если процесс прерывается сигналом, зарегистрированные функции не вызываются.

Данная функция не требует каких-либо параметров и не возвращает значений. Прототип выглядит следующим образом:

```
void my_function (void);
```

Функции вызываются в порядке, обратном регистрации: это значит, что они хранятся в стеке и последняя вошедшая функция будет вызвана первой (LIFO). Зарегистрированные функции не должны вызывать exit() во избежание бесконечной рекурсии. Если функция должна окончить процесс завершения раньше, необходимо вызвать _exit(). Такой подход не рекомендуется, потому что затем возможен сбой запуска важных завершающих функций.

Стандарт POSIX требует поддержки atexit() по крайней мере для ATEXIT_MAX зарегистрированных функций, следовательно, эта величина должна быть равна по меньшей мере 32. Точный максимум может быть определен через sysconf() и величину _SC_ATEXIT_MAX:

```
long atexit_max;
atexit_max = sysconf (_SC_ATEXIT_MAX);
printf ("atexit_max=%ld\n", atexit_max);
```

В случае успеха atexit() возвращает 0, при ошибке она возвращает значение -1. Вот простой пример:

```
#include <stdio.h>
#include <stdlib.h>
void out (void) {
        printf ("atexit() succeeded!\n");
```

```
}
int main (void) {
        if (atexit (out))
                fprintf(stderr, "atexit() failed!\n");
        return 0;
}
```

on_exit()

SunOS 4 определяет собственный эквивалент atexit(), библиотека glibc в Linux поддерживает его:

```
#include <stdlib.h>
int on_exit (void (*function)(int, void *), void *arg);
```

Эта функция работает так же, как atexit(), но прототип зарегистрированной функции несколько отличается:

```
void my_function (int status, void *arg);
```

Аргумент status — это величина, переданная exit() или возвращенная main(). Аргумент arg — второй параметр, переданный exit(). Следует удостовериться, что данные, содержащиеся в памяти arg, являются допустимыми на момент вызова функции.

Последняя версия Solaris уже не поддерживает эту функцию. Вместо нее необходимо использовать стандартно компилируемую atexit().

Когда процесс завершается, ядро посылает сигнал SIGCHLD родительскому процессу. По умолчанию этот сигнал игнорируется и родительский процесс не предпринимает каких-либо действий. Однако при необходимости процессы могут обработать данный сигнал с помощью системных вызовов signal() или sigaction().

Сигнал SIGCHLD может быть сгенерирован и отправлен в любое время, так как завершение дочернего процесса не синхронно родительскому. Однако часто предку требуются сведения о завершении его потомка или даже некоторое время для ожидания события. Это возможно с помощью системных вызовов, описанных далее.

2.11 Ожидание завершенных дочерних процессов

Получение предупреждения на сигнал — это прекрасно, но многие родительские процессы ходят получить больше информации,

когда завершается их дочерний процесс — например, если тот возвращает какое-либо значение.

Если бы потомок по завершении полностью исчезал, как можно было бы предположить, то получение каких-либо сведений было бы для его предка невозможно. Следовательно, первые разработчики UNIX решили, что когда дочерний процесс завершается прежде родительского, ядро должно поместить потомка в особый процессный статус. Процесс в этом состоянии известен как зомби. В данном состоянии существует лишь «скелет» процесса — некоторые основные структуры данных, содержащие потенциально нужные сведения. Процесс в таком состоянии ожидает запроса о своем статусе от предка (процедура, известная как ожидание процесса-зомби). Только после того как предок получит всю необходимую информацию о завершенном дочернем процессе, последний формально удаляется и перестает существовать даже в статусе зомби.

Ядро Linux предоставляет несколько интерфейсов для получения информации о завершенном дочернем процессе. Самый простой из них, определенный POSIX, называется wait():

```
#include<sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);
```

Вызов wait() возвращает pid завершенного дочернего процесса или -1 в случае ошибки. Если никакого дочернего процесса не было прервано, вызов блокируется, пока потомок не завершится. Если дочерний процесс уже был завершен, вызов возвращает результаты немедленно. Следовательно, если вызвать wait() сразу после сообщения о завершении дочернего процесса, результат будет немедленно выдан без блокировки.

В случае ошибки возможно присвоение переменной errno одного из двух значений:

- ECHILD — вызывающий процесс не имеет дочерних;
- EINTR — сигнал был получен во время ожидания, в результате чего вызов вернул результат слишком рано.

Если указатель status не содержит значения NULL, там находится дополнительная информация о дочернем процессе. POSIX позволяет при реализации определение битов статуса разработчиками

самостоятельно, поэтому стандарт предусматривает семейство макросов для интерпретации параметра:

```
#include <sys/wait.h>
int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONTINUED (status);
int WEXITSTATUS (status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

Любой из первых двух макросов может возвращать значение true (ненулевое) в зависимости от хода завершения процесса. Первый, WIFEXITED, возвращает true, если процесс завершается через вызов _exit(), обычным образом. Соответственно, макрос WEXITSTATUS предоставляет 8 бит младших разрядов и передает их _exit().

Макрос WIFSIGNALED возвращает true, если прерывание процесса вызвал сигнал. В этом случае WTERMSIG возвращает номер сигнала, который вызвал прерывание, а WCOREDUMP возвращает true, если процесс сбросил ядро в ответ на получение сигнала. WCOREDUMP не определяется POSIX, хотя многие системы UNIX, Linux в том числе, поддерживают его.

Макросы WIFSTOPPED и WIFCONTINUED возвращают true, если процесс был остановлен или продолжен соответственно и его можно в настоящий момент отследить с помощью системного вызова ptrace(). Эти условия обычно возможны только во время реализаций отладчика, хотя при использовании вместе с waitpid(). Обычно wait() применяется только для обмена информацией о завершении процесса. Если WIFSTOPPED возвратил true, WSTOPSIG приводит номер сигнала, остановившего процесс. WIFCONTINUED не определяется POSIX, хотя более поздние стандарты определили его для waitpid(). В версии 2.6.10 ядра Linux также предоставлен макрос для wait().

Рассмотрим пример программы, которая использует wait(), чтобы определить, что произошло с дочерним процессом:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/wait.h>
int main (void)
{
        int status;
        pid_t pid;
        if (!fork ())
                return 1;
        pid = wait (&status);
        if (pid == -1)
                perror ("wait");
        printf ("pid=%d\n", pid);
        if (WIFEXITED (status))
                printf ("Нормальное завершение, СТаТус=%d\n",
        WEXITSTATUS (status));
        if (WIFSIGNALED (status))
                printf ("Убит сигналом^^^",
        WTERMSIG (status),
        WCOREDUMP (status) ? " (dumped core)" : "");
        if (WIFSTOPPED (status))
                printf ("Остановлен сигналом=М\п",
        WSTOPSIG (status));
        if (WIFCONTINUED (status))
                printf ("Продолжен^");
        return 0;
        }
```

Эта программа ответвляет дочерний процесс, который немедленно завершается. После этого предок запускает системный вызов wait() для определения статуса потомка. Процесс печатает pid потомка и сведения о его завершении. Поскольку в этом случае дочерний процесс завершился возвратом результата main() , понятно, что на выходе получим примерно следующее:

```
$ ./wait pid=8529
```

2.12 Ожидание определенного процесса

Наблюдение за поведением дочернего процесса очень важно. Часто, однако, процесс имеет нескольких потомков и, если нужен

только определенный, ожидание всех нежелательно. Возможным решением были бы многократные вызовы wait() с постоянной проверкой возвращаемого значения, однако это весьма неудобно: что, если позже понадобится проверить статус другого завершенного процесса? Родительскому процессу пришлось бы сохранять результаты всех вызовов wait() на случай, если они понадобятся позднее.

Если вам известен pid процесса, завершения которого вы ждете, можно использовать системный вызов waitpid():

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options);
```

Вызов waitpid() — более мощная версия wait(). Его дополнительные параметры позволяют настроить его более тонко.

Определяется в заголовке <stdlib.h>. Параметр pid точно определяет, какой процесс или процессы нужно ожидать. Его значения могут попадать в четыре промежутка:

- < -1 — ожидание любого дочернего процесса, чей ID группы процессов равен абсолютному значению этой величины; например, при -500 ожидается любой процесс из группы процессов 500;
- -1 — ожидание любого дочернего процесса; поведение аналогично wait();
- 0 — ожидание любого дочернего процесса, принадлежащего той же группе процессов, что и вызывающий;
- > 0 — ожидание любого дочернего процесса, чей pid в точности равен указанной величине; например, при величине 500 ожидается дочерний процесс с pid, равным 500.

Параметр status работает аналогично таковому в системном вызове wait() и может быть обработан с помощью макросов, описанных выше.

Параметр options может передавать следующие значения с помощью логического «ИЛИ» либо пустое значение:

- WNOHANG — не блокировать вызов, немедленно вернуть результат, если ни один подходящий процесс еще не завершился (остановился или продолжился);

- WUNTRACED — при его выборе устанавливается параметр WIFSTOPPED, даже если вызывающий процесс не отслеживает свой дочерний; это свойство помогает реализовать более общее управление заданиями, как это сделано в оболочке;
- WCONTINUED — если установлен, то бит WIFCONTINUED в возвращаемом параметре статуса будет установлен, даже если вызывающий процесс не отслеживает свой дочерний; как и в случае с WUNTRACED, параметр полезен для реализации оболочки.
- В случае успеха waitpid() возвращает pid процесса, статус которого изменился. Если установлен WNOHANG, а указанный дочерний процесс (один или несколько) не изменил свой статус, waitpid() вернет 0. В случае ошибки вызов возвращает -1, а errno принимает одно из следующих трех значений:
- ECHILD — процесс или процессы, указанные с помощью аргумента pid, не существуют или не являются потомками вызывающего;
- EINTR — параметр WNOHANG не был установлен, а сигнал был получен во время ожидания;
- EINVAL — аргумент options указан некорректно.

Рассмотрим пример. Программа должна получить значение, возвращаемое дочерним процессом с pid 1742, причем сделать это немедленно, если дочерний процесс еще не завершился. Можно реализовать это следующим образом:

```
int status;
pid_t pid;
pid = waitpid (1742, &status, WNOHANG);
if (pid == -1)
        perror ("waitpid");
        else {
                printf ("pid=%d\n", pid); if (WIFEXITED (status))
                printf ("Обычное завершение со статусом
выхода=M\n",
                WEXITSTATUS (status));
                if (WIFSIGNALED (status))
                        printf ("Убит сигналом=^&^",
                WTERMSIG (status),
```

```
                WCOREDUMP (status) ? " (дамп ядра)" : "");
        }
```

Последний пример. Нужн обратить внимание на следующее использование wait():

```
wait (&status);
```

Это полностью аналогично: waitpid (-1, &status, 0);

Для приложений, которые требуют еще большей гибкости в их функциональности ожидания дочерних процессов, расширение XSI стандарта POSIX определяет, а Linux поддерживает системный вызов waitid():

```
#include <sys/wait.h>
int waitid (idtype_t idtype, id_t id,
siginfo_t *infop, int options);
```

Как и wait() и waitpid(), системный вызов waitid() используется для ожидания и получения информации об измененном статусе (завершение, остановка, продолжение) дочернего процесса. Он предоставляет еще больше параметров, но их использование несколько сложнее.

Аналогично waitpid() с помощью waitid() разработчик может выбрать ожидаемый процесс. Однако waitid() требует для этого указания не одного, а двух параметров. С помощью аргументов idtype и id определяется, какой дочерний процесс нужно ожидать (аналогично использованию одного аргумента pid в waitpid()). Значения idtype могут быть следующими:

- P_PID — ожидание дочернего процесса, pid которого совпадает со значением аргумента id;
- P_GID — ожидание дочернего процесса, идентификатор группы которого совпадает со значением id;
- P_ALL — ожидание всех дочерних процессов, id игнорируется.

Аргумент id принадлежит к достаточно редкому типу аргументов id_t, представляющему общий идентификационный номер. Его используют, если в будущем планируется ввести новое значение idtype, обеспечив таким образом гарантию, что в заданном типе можно будет хранить новый идентификатор (его размер достаточен для помещения любого значения pid_t). Разработчики Linux могут применять его аналогично типу pid_t, напрямую

передавая значение pid_t или числовых констант. Педантичные программисты, однако, могут и преобразовать типы.

В параметре options могут быть указаны одно или несколько из следующих значений с помощью двоичного «ИЛИ»:

- WEXITED — вызов будет ждать дочерних процессов (определенных с помощью id или idtype), которые завершились;
- WSTOPPED — вызов будет ожидать дочерних процессов, которые остановили выполнение в ответ на получение сигнала;
- WCONTINUED — вызов будет ожидать дочерних процессов, которые продолжили выполнение в ответ на получение сигнала;
- WNOHANG — вызов не может быть заблокирован и вернет результаты немедленно, если ни один из указанных дочерних процессов еще не завершен (или остановлен, или продолжен);
- WNOWAIT — вызов не будет выводить указанный процесс из статуса зомби; этот процесс будет обработан в будущем.

В случае успеха wiatid() возвращает параметр infop, который укажет на допустимый тип s iginfo_t. Точная структура s i ginfo_t зависит от реализации1, но после выполнения waitid() заполненными остаются лишь несколько полей. Таким образом, при успешном вызове допустимые значения будут содержаться в следующих полях:

- si_pid — pid дочернего процесса;
- si_uid — uid дочернего процесса;
- si_code — может принимать значения CLD_EXITED, CLD_KILLED, CLD_STOPPED или CLD_CONTINUED в результате завершения дочернего процесса, окончания или продолжения его по сигналу соответственно;
- si_signo — устанавливается значение SIGCHLD;
- si_status — если si_code установился равным CLD_EXITED, это поле содержит код выхода дочернего процесса, и наоборот, это поле принимает значение номера сигнала, отправленного дочернему процессу и вызвавшего изменения.

В случае успеха waitid() возвращает 0. При ошибке waitid() возвращает -1, а errno принимает одно из следующих значений:

- ECHLD — процесс или процессы, указанные с помощью id или idtype, не существуют;
- EINTR — WNOHANG не был установлен в options, а сигнал прервал выполнение;

- EINVAL — аргумент options или комбинация аргументов id и idtype некорректны.

Функция waited() предоставляет дополнительную полезную семантику, отсутствующую в wait() и waitpid(). В частности, информация, предоставляемая в структуре siginfo_t, может быть очень полезной. Если эти сведения не требуются, имеет смысл использовать более простые функции, которые поддерживаются большим количеством систем и, следовательно, могут быть перенесены в другие системы, не относящиеся к Linux.

Вызов waitpid() является наследником System V Release 4 системы AT&T, а BSD идет собственным путем и предоставляет две другие функции, которые используются для ожидания изменения статуса дочернего процесса:

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
pid_t wait3 (int *status, int options, struct rusage *rusage);
pid_t wait4 (pid_t pid, int *status, int options, struct rusage *rusage);
```

Цифры 3 и 4 указывают, что эти две функции являются версиями wait() соответственно с тремя и четырьмя параметрами.

Функции работают схожим образом с waitpid(), за исключением аргумента rusage. Такой запуск wait3():

```
pid = wait3 (status, options, NULL);
```

эквивалентен следующему вызову waitpid():

```
pid = waitpid (-1, status, options);
```

в то время как данный запуск wait4():

```
pid = wait4 (pid, status, options, NULL);
```

эквивалентен такому вызову waitpid():

```
pid = waitpid (pid, status, options);
```

Таким образом, wait3() ожидает любого дочернего процесса с изменившимся статусом, а wait4() ожидает определенного дочернего процесса, указанного параметром pid и изменившего статус. Аргумент options работает так же, как и в waitpid().

Как упоминалось выше, самое большое различие между этими вызовами и waitpid() — параметр rusage. Если его значение не NULL, функция заполняет указатель, прописанный в rusage, информацией о

дочернем процессе. Эта структура предоставляет информацию об использовании ресурсов дочерним процессом:

```
#include <sys/resource.h> struct rusage {
struct timeval ru_utime;      /* затраченное пользовательское
время */
struct timeval ru_stime;      /* затраченное системное время */
long ru_maxrss;       /* максимальный размер резидентной
части */
long ru_ixrss;        /* размер общей памяти */
long ru_idrss;        /* размер собственных данных */
long ru_isrss;        /* размер собственного стека */
long ru_minflt;       /* восстановления страниц */
long ru_majflt;       /* страничные прерывания */
long ru_nswap;        /* операции подкачки */
long ru_inblock;      /* блочные операции ввода */
long ru_oublock;          /* блочные операции вывода */
long ru_msgsnd;       /* отправленные сообщения */
long ru_msgrcv;       /* полученные сообщения */
long ru_nsignals;         /* полученные сигналы */
long ru_nvcsw;        /* добровольные переключения контекста
*/
long ru_nivcsw;       /* вынужденные переключения контекста
*/
};
```

В случае успеха эти функции возвращают pid процесса, изменившего статус. В случае неудачи они возвращают -1 и устанавливают в errno те же значения ошибок, что описаны для waitpid().

Поскольку wait3() и wait4() не определены в стандарте POSIX1, желательно применять их, только если получение информации об использовании ресурсов чрезвычайно важно. Однако, несмотря на отсутствие стандартизации POSIX, почти все системы UNIX поддерживают эти два вызова.

2.13 Запуск и ожидание нового процесса

В стандартах ANSI C и POSIX определяется интерфейс, объединяющий запуск нового процесса и ожидание его завершения — представьте себе это как синхронное создание процессов. Если процесс запускает дочерний, только чтобы немедленно начать ожидать его завершения, лучше всего использовать следующий интерфейс:

```
#define _XOPEN_SOURCE /* если нужен WEXITSTATUS и т. д.*/
#include <stdlib.h>
int system (const char *command);
```

Функция system() называется так потому, что синхронный запуск процессов называется выходом в систему. Принято использовать system() для запуска простой утилиты или сценария оболочки, когда основной целью является получение возвращаемого ими значения.

Вызов system() запускает команду, предоставляемую параметром command, включая любые дополнительные аргументы. Параметр command добавляется к аргументам /bin/sh-c. В этом смысле параметр передается всей оболочке.

В случае успеха возвращаемой величиной является статус, который вернула команда и предоставил вызов wait(). Таким образом, код выхода выполненной команды получается с помощью WEXIT STATUS. Если вызов /bin/sh потерпел неудачу, величина, передаваемая WEXITSTATUS, та же, что возвращается exit(127). Поскольку вызываемая команда также может вернуть 127, безошибочного метода определения источника ошибки — командой она вызвана или оболочкой — не существует. В случае ошибки вызов возвращает значение -1.

Если параметр command имеет значение NULL, system() возвращает ненулевую величину, если shell /bin/sh доступен, и 0 в противном случае.

Во время выполнения команды SIGCHLD блокируется, а SIGINT и SIGQUIT игнорируются. Игнорирование SIGINT и SIGQUIT имеет различные последствия, в частности, когда system() вызывается

внутри цикла. В этом случае нужно убедиться, что программа правильно проверяет статус выхода дочернего процесса, например:

```
do {
        int ret;
        ret = system ("pidof rudderd"); if (WIFSIGNALED (ret) &&
                (WTERMSIG (ret) == SIGINT ||
                WTERMSIG (ret) == SIGQUIT))
        break; /* или другой вариант обработки */
        } while (1);
```

Реализация system() с использованием fork(), функции из семейства exec, и waitpid() — очень полезное упражнение. Вы должны попытаться выполнить его самостоятельно, поскольку для этого потребуется большая часть материала данной главы. Для завершенности приведу пример простой реализации:

```
/*
* my_system - синхронно ответвляет дочерний процесс и
ожидает
* команды
* "/bin/sh -c<cmd>".
*   Возвращает -1 в случае любой ошибки или код выхода
запущенного
* процесса
* Не блокирует и не игнорирует сигналы */
int my_system (const char *cmd)
{
        int status; pid_t pid;
        pid = fork ();
        if (pid == -1)
                return -1;
        else if (pid == 0) {
                const char *argv[4]; argv[0] = "sh"; argv[1] = "-c";
argv[2] = cmd;          argv[3] = NULL; execv ("/bin/sh", argv);
        exit (-1);
}
if (waitpid (pid, &status, 0) == -1)
        return -1;
else if (WIFEXITED (status))
```

```
return WEXITSTATUS (status);
return -1;
}
```

Стоит обратит внимание, что в этом примере не блокируются и не отключаются никакие сигналы, в отличие от официальной версии system(). Этот подход может быть лучше или хуже, в зависимости от конкретной программы, но не блокировать по крайней мере SIGING часто может быть полезно, потому что это позволяет прерывать запущенную программу так, как привычно пользователю в обычных ситуациях. Усложнить задачу можно, добавив дополнительные указатели в виде параметров, которые указывали бы на ошибки, в настоящее время неразличимые, когда их значение отлично от NULL. В качестве примера можно добавить fork_failed и shell_failed.

Системному вызову system() свойственны те же проблемы с безопасностью, что и execlp() и execvp(). Нельзя запускать system() с установленного идентификатора группы (set-group-ID) или пользователя (set-user-ID) программы, так как злоумышленник может манипулировать настройками окружения (чаще всего PATH), чтобы расширить свои привилегии. Рукописные замены system() также уязвимы, так как используют оболочку.

Во избежание атак такого типа программы установки идентификаторов пользователя или группы должны запускать желаемый внешний бинарный модуль через fork() и execl() без использования оболочки. Отказ от запуска внешнего бинарного модуля является даже лучшим решением!

2.14 Процессы зомби

Как было сказано выше, процесс, который прервался, но чей предок пока не ожидает его завершения, называется зомби. Процессы-зомби продолжают занимать системные ресурсы, пусть и в небольших количествах, ведь требуется поддерживать лишь «скелет» работавшего ранее процесса. Эти ресурсы нужны, чтобы родительские процессы, желающие проверить статус своих потомков, получили информацию, относящуюся к существованию и прекращению данных процессов. Как только родительский процесс

сделает это, ядро окончательно удаляет процесс и зомби отправляется на покой.

Однако каждый, кто какое-то время использовал Linux, время от времени натыкается на процессы-зомби. Эти процессы, часто называемые призраками, — дети безответственных родителей. Если ваше приложение ответвляет дочерний процесс, то оно несет ответственность за его обслуживание, даже если это заключается лишь в удалении собранной информации (кроме случаев, когда приложение кратковременно, о чем мы скоро поговорим). Иначе все дочерние процессы будут становиться призраками и существовать в списках процессов системы, что никак не украшает приложение.

Что происходит, если родительский процесс умирает раньше дочернего или если он завершается до того, как получит возможность позаботиться о своих по- томках-зомби? Когда бы ни завершался процесс, ядро Linux проходится по списку его потомков и переназначает родительский процесс, делая их предком процесс инициализации, pid которого равен 1. Таким образом, ни один процесс не станет сиротой без родительского. Процесс инициализации, в свою очередь, периодически заботится обо всех своих потомках, в результате чего ни один из них не остается зомби чрезмерно долго (никаких призраков). Следовательно, если родительский процесс завершается раньше своих потомков или не обсуживает их перед завершением, дочерние процессы переходят под опеку процесса инициализации и обслуживаются им до полного завершения. Хотя подобная реализация считается хорошей практикой, данная предосторожность означает, что краткосрочные процессы могут особенно не беспокоиться об ожидании всех своих потомков.

2.15 Пользователи и группы

Как упоминалось ранее , процессы принадлежат определенным пользователям и группам. Идентификаторы группы и процесса — численные величины, представленные типами C uid_t и gid_t соответственно. Связь между численными величинами и именами, удобными для восприятия человеком, — например, у пользователя root значение uid равно 0 — осуществляется в пользовательском

пространстве с помощью файлов /etc/passwd и /etc/group. Ядро может работать только с численными величинами.

В системе Linux идентификаторы пользователя и группы какого-либо процесса определяют операции, доступные для выполнения данным процессом. Следовательно, процессы исполняются от имени определенных пользователей и групп. Много процессов может быть запущено только от имени пользователя root. Однако при разработке программного обеспечения лучше всего следовать доктрине наименьших прав, что означает: процесс должен работать с минимальным из возможных уровней прав. Это требование динамично: если процессу для выполнения требуются права root для выполнения какой-либо операции на начальном этапе существования, а после этого необходимости в расширенных правах нет, он должен как можно скорее избавиться от прав root, поэтому большинство процессов — в частности, как раз те, которым для выполнения определенных операций требуются права доступа root, — часто манипулируют своими идентификаторами пользователя или группы.

Реальные, действительные и сохраненные идентификаторы пользователя и группы. На самом деле существует не один, а четыре пользовательских идентификатора, ассоциированных с процессом: реальный, действительный, сохраненный и идентификатор файловой системы. Реальный идентификатор пользователя — это uid пользователя, который изначально запустил процесс. Он устанавливается по реальному идентификатору пользователя родительского процесса и не изменяется в течение работы вызова exec. Обычно при авторизации устанавливается реальный идентификатор пользователя, и все процессы пользователя продолжают работу с этим идентификатором. Пользователь с правами доступа root может менять реальный идентификатор пользователя по мере надобности, но другие пользователи не могут этого делать.

Действительный идентификатор пользователя — это идентификатор, под которым в настоящий момент выполняется процесс. Проверка доступа обычно основывается на этом значении. Изначально этот идентификатор равен реальному идентификатору пользователя, поскольку, когда процесс начинает ветвление, действительный идентификатор пользователя передается от родительского процесса к дочернему. Далее, когда процесс

сталкивается с вызовом exec, действительный пользователь обычно не меняется. Однако именно во время работы вызова exec обнаруживается ключевое различие между реальным и действительным идентификаторами: процесс может изменить свой действительный пользовательский идентификатор с помощью запуска бинарного модуля setuid (suid). Точнее, действительный идентификатор пользователя меняется на идентификатор пользователя, который является владельцем программы. Например, так как файл /usr/bin/passwd является setuid, а его владелец — пользователь root, когда оболочка обычного пользователя ответвляет процесс, выполняющий файл, процессу присваивается действительный идентификатор пользователя root, независимо от того, какой пользователь выполняет его фактически.

Непривилегированные пользователи могут устанавливать в качестве действительного в реальный или сохраненный пользовательский идентификатор. Пользователь с правами root может присваивать действительному идентификатору любое значение.

Сохраненным идентификатором пользователя называется изначальный действительный пользовательский идентификатор. Когда процесс начинает ветвление, потомок наследует сохраненный пользовательский идентификатор родителя. Во время вызова exec, однако, ядро устанавливает в качестве сохраненного действительный идентификатор пользователя, таким образом сохраняя информацию о действительном идентификаторе на момент запуска exec. Непривилегированные пользователи не могут менять свой сохраненный идентификатор; пользователи с правами root могут присвоить ему значение реального идентификатора пользователя.

Важнее всего — действительный идентификатор пользователя. Именно эта величина проверяется во время валидации доступа процесса. Реальный и сохраненный идентификаторы пользователя играют роль суррогатов или потенциальных значений идентификатора пользователя, которые могут использовать процессы, не обладающие правами root. Реальный идентификатор пользователя — это действительный идентификатор, принадлежащий пользователю, фактически выполняющему программу, а сохраненный — это действительный идентификатор до изменения им во время вызова exec двоичным файлом suid.

Изменение реального или сохраненного идентификатора пользователя или группы

Идентификаторы пользователя или группы устанавливаются с помощью двух системных вызовов:

#include <sys/types.h>

#include <unistd.h>

int setuid (uid_t uid); int setgid (gid_t gid);

Вызов setuid() устанавливает действительный идентификатор пользователя текущего процесса. Если текущий действительный идентификатор пользователя данного процесса равен 0 (root), устанавливаются также реальный и сохраненный идентификаторы. Пользователь root сможет передавать в качестве параметра uid любые значения, устанавливая всем трем значениям пользовательского идентификатора величину uid. Пользователь без прав root может изменять только величину реального или сохраненного пользовательского идентификатора. Другими словами, пользователь без прав root может установить действительному идентификатору пользователя лишь одно из этих значений.

В случае успеха setuid() возвращает 0. При ошибке вызов возвращает -1, а errno присваивает одно из следующих значений:

- EAGAIN — uid отличается от реального пользовательского идентификатора, и установление uid в качестве реального идентификатора пользователя выведет его за пределы RLIM_NPROC (определяющего количество процессов, которыми может владеть пользователь);
- EPERM — пользователь не обладает правами root, а uid не является ни действительным, ни сохраненным идентификатором пользователя.

Linux предоставляет две функции, утвержденные POSIX, с помощью которых можно установить действительный идентификатор пользователя или группы:

#include <sys/types.h>

#include <unistd.h>

int seteuid (uid_t euid); int setegid (gid_t egid);

Вызов seteuid() устанавливает действительный идентификатор пользователя равным euid. Пользователь root может установить в качестве euid любую величину. Пользователи без прав root могут

установить действительный идентификатор пользователя только равным реальному или сохраненному идентификатору пользователя. В случае успеха setuid() возвращает 0. При неудаче он возвращает -1 и устанавливает errno значение EPERM, указывающее, что владелец текущего процесса не имеет прав root, и этот euid не равен ни сохраненному, ни реальному пользовательскому идентификатору.

Стоит обратить внимание, что в случае отсутствия прав root функции seteuid () и setui d () работают одинаково. Таким образом, стандартная практика — и хорошая идея — всегда использовать seteuid(), если ваш процесс не будет выполняться от имени пользователя root, когда применять setuid() выгоднее.

Все сказанное выше касается и групп, нужно только заменить seteuid() на setegid(), а euid на egid.

Разработчики BSD пришли к собственным интерфейсам для изменения идентификаторов пользователя и группы. В Linux эти интерфейсы предоставляются для обеспечения совместимости:

#include <sys/types.h>

#include <unistd.h>

int setreuid (uid_t ruid, uid_t euid); int setregid (gid_t rgid, gid_t egid);

Вызов setreuid() устанавливает реальные и действительные идентификаторы процесса равными ruid и euid соответственно. Передача величины -1 для любого из параметров не изменит соответствующий идентификатор пользователя. Процессы без прав root могут только устанавливать значение действительного идентификатора пользователя реальному или сохраненному идентификатору, а реального — действительному. Если реальный пользовательский идентификатор изменился или действительный пользовательский идентификатор изменился, став величиной, не равной предыдущему реальному идентификатору, сохраненный идентификатор пользователя изменяется на новый действительный идентификатор. Во всяком случае, именно так Linux и большинство систем UNIX реагируют на подобные изменения; данный подход не освещается в стандарте POSIX.

В случае успеха setreuid() возвращает 0. При неудаче возвращает -1 и присваивает errno значение EPERM. Это означает, что текущий процесс не имеет прав root, а этот euid не равен ни реальному, ни

сохраненному идентификатору пользователя или этот ruid не равен эффективному идентификатору пользователя.

Все сказанное выше касается и групп, нужно только заменить setreuid() на setregid(), ruid на rgid, a euid на egid.

Linux предоставляет следующие интерфейсы, которые полезны, если требуется совместимость с HP-UX:

#define _GNU_SOURCE #include <unistd.h>

int setresuid (uid_t ruid, uid_t euid, uid_t suid); int setresgid (gid_t rgid, gid_t egid, gid_t sgid);

Вызов setresuid() устанавливает реальный, действительный и сохраненный идентификаторы пользователя равными ruid, euid, и suid соответственно. Указание значения -1 для любого из этих параметров оставит его неизменным.

Пользователь root может присвоить любое значение любому из идентификаторов пользователя. Пользователи без прав root могут установить любой идентификатор равным текущим реальному, действительному или сохраненному идентификатору. В случае успеха setuid() возвращает 0. При ошибке вызов возвращает -1 и errno присваивается одно из следующих значений:

- EAGAIN — uid не совпадает с реальным ID пользователя, и установка реального идентификатора равным uid выведет пользователя за пределы RLIM_NPROC (определяющего количество процессов, которыми может владеть пользователь);
- EPERM — пользователь не имеет прав root, а набор новых величин для реального, действительного или сохраненного идентификатора не совпадает ни с одним из значений реального, действительного или сохраненного пользовательских ID.

Все вышесказанное применимо и к группам, просто замените setresuid() на setresgid(), ruid на rgid, euid на egid и suid на sgid.

2.16 Действия с предпочтительными идентификаторами пользователя или группы

Процессы без прав root должны использовать setuid() для изменения своих действительных идентификаторов пользователя. Процессы с правами root должны применять setuid(), если они хотят изменить все три пользовательских идентификатора, и seteuid(), если

нужно временно изменить только действительный идентификатор пользователя. Эти функции просты и работают в соответствии с POSIX, принимая во внимание сохраненный идентификатор пользователя.

Несмотря на предоставление дополнительных функциональностей, функции BSD и HP-UX не позволяют вносить полезные изменения, в отличие от setuid() и seteuid().

2.17 Поддержка сохраненных пользовательских идентификаторов

Существование сохраненных идентификаторов пользователя и группы регулируется IEEE Std 1003.1-2001(POSIX 2001), и Linux поддержала эти идентификаторы с момента появления ядра 1.1.38. В программах, написанных только для Linux, существование сохраненных идентификаторов пользователя предусмотрено изначально и они всегда будут на месте. В программах, которые были написаны для более старых систем UNIX, необходимо проверять наличие макроса _POSIX_SAVED_IDS и лишь затем ссылаться на сохраненный идентификатор пользователя или группы.

Если сохраненные ID пользователя или группы отсутствуют, вышеизложенное все равно действительно, просто игнорируйте правила, в которых упоминаются сохраненные идентификаторы пользователя или группы.

Эти два системных вызова возвращают реальные идентификаторы пользователя и группы соответственно:

#include <unistd.h>

#include <sys/types.h>

uid_t getuid (void); gid_t getgid (void);

Они не могут привести к неудаче. Аналогично эти два вызова возвращают действительные идентификаторы пользователя и группы соответственно: #include <unistd.h>

#include <sys/types.h>

uid_t geteuid (void);

gid_t getegid (void);

Они также не могут привести к сбою.

2.17 Сессии и группы процессов

Каждый процесс является членом группы процессов, которая представляет собой коллекцию из одного или нескольких процессов, в общем случае связанных друг с другом с целью управления заданиями. Основная особенность группы процессов заключается в том, что эти сигналы могут быть отправлены всем процессам в группе: одно действие может прервать, остановить или продолжить все процессы в этой группе.

Каждая группа процессов идентифицируется с помощью идентификатора группы процессов, а также имеет лидера группы процессов. Идентификатор группы процессов равен pid лидера группы процессов. Группы существуют, пока в них остается хотя бы один член. Даже если лидер группы процессов прерывается, группа продолжает существовать.

Когда новый пользователь впервые входит в систему, процесс авторизации создает новую сессию, которая содержит единственный процесс — оболочку авторизации пользователя. Оболочка авторизации функционирует как лидер сессии. Сессией называется набор из одной или нескольких групп процессов. С помощью сессий устанавливается порядок среди действий авторизованных пользователей, а сами пользователи связываются с управляющим терминалом — особым устройством tty, управляющим терминальным процессом ввода-вывода для данного пользователя. Следовательно, сессии в основном связаны с оболочками. Фактически больше ничто в системе не управляет ими.

В то время как группы процессов предоставляют механизм для упрощения адресации сигналов всем членам группы, выполнения контроля задач и других функций оболочки, сессии отвечают лишь за объединение авторизации пользователей вокруг управляющих терминалов. Группы процессов в сессии делятся на единственную приоритетную группу процессов и фоновые группы процессов, которых может быть несколько или не быть вообще. Когда пользователь покидает терминал, SIGQUIT отправляется всем процессам в приоритетной группе. Когда терминал отключается от сети, SUGNUP отправляется всем процессам в приоритетной группе процессов. Когда пользователь вводит команду прерывания (обычно

сочетанием клавиш Ctri+C), всем процессам в приоритетной группе отправляется SIGI NT. Таким образом сессии упрощают для оболочки управление терминалами и авторизациями.

В качестве примера представим, что пользователь авторизуется в системе и ее оболочка авторизации, bash, имеет pid 1700. Экземпляр bash пользователя становится единственным членом и лидером новой группы процессов с идентификатором группы процессов 1700. Группа находится внутри новой сессии с идентификатором сессии 1700, а bash — единственный член и лидер этой сессии. Новые команды, которые пользователь запустит в оболочке, будут работать в новой группе процессов внутри сессии 1700. Одна из этих групп процессов — та, которая связана непосредственно с пользователем и находится под управлением терминала, — является приоритетной группой процессов. Все остальные группы — фоновые группы процессов.

В данной системе существует много сессий: одна для каждой пользовательской сессии авторизации и другие для процессов, не связанных с пользовательскими сессиями, такие как демоны. Демоны стараются создавать свои собственные сессии во избежание проблем из-за связи с другими сессиями, которые могут прерваться.

Каждая из этих сессий содержит одну или несколько групп процессов, а каждая группа процессов содержит как минимум один процесс. Группа процессов, содержащая более одного процесса, в общем виде реализует управление заданиями.

Команда оболочки наподобие:

$ cat ship-inventory.txt | grep booty | sort

возвращает группу процессов, включающую три процесса. Таким образом, оболочка может отправлять сигналы трем процессам одновременно. Поскольку пользователь напечатал эту команду в консоли без использования ведущего амперсанда, можно утверждать, что эта группа процессов будет приоритетной. Рис. 2.1 иллюстрирует взаимоотношения между сессиями, группами процессов, процессами и контролирующими терминалами.

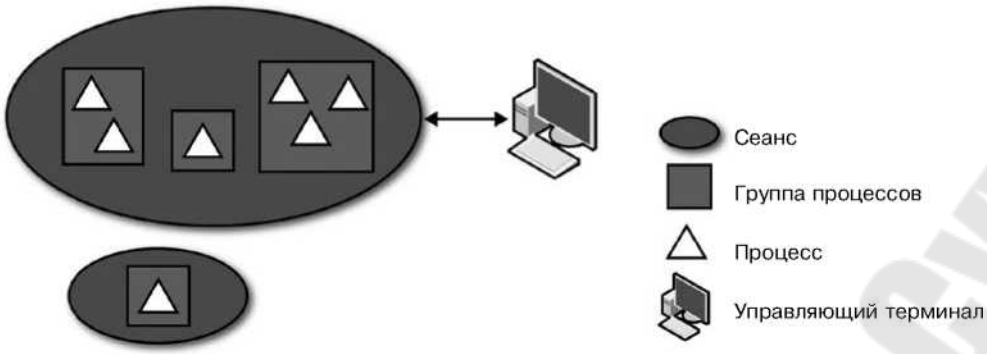

Рисунок 2.1. Взаимоотношения между сессиями, группами процессов, процессами и управляющими терминалами

Linux предоставляет несколько интерфейсов для установления и получения сессии и группы процессов, связанных с данным процессом. В основном они используются для нужд оболочки, но также могут быть полезны для таких процессов, как демоны, которые предпочитают не взаимодействовать с сессиями и группами процессов.

2.18 Системные вызовы сессий

Оболочки создают новые сессии при авторизации. Они делают это с помощью специального системного вызова, который весьма упрощает создание новой сессии:

#include<unistd.h> pid_tsetsid (void);

Вызов setsid() создает новую сессию, предполагая, что процесс в данный момент не является лидером группы процессов. Вызывающий процесс создает лидера сессии и единственного члена новой сессии, не имеющего контролирующего tty. Вызов также создает новую группу процессов внутри сессии, делая вызывающий процесс ее лидером и единственным членом. Новые идентификаторы сессии и группы процессов равны pid вызывающего процесса.

Другими словами, setsid() создает новую группу процессов внутри новой сессии и делает вызывающий процесс лидером того и другого. Это полезно для демонов, которые не хотят быть членами существующих сессий или иметь контролирующие терминалы, а также для оболочек, которые хотят создавать новую сессию для каждого пользователя после авторизации.

В случае успеха setsid() возвращает идентификатор вновь созданной сессии. В случае ошибки возвращается -1, единственно возможный код ошибки EPERM, который означает, что процесс является лидером группы процессов. Самый простой путь гарантировать, что какой-либо процесс не является лидером группы процессов, — разветвить его, завершить родительский процесс и заставить дочерний выполнить setsid(). Например:

```
pid_t pid;
pid = fork ();
if (pid == -1) {
        perror ("fork"); return -1;
        } else if (pid != 0)
                exit (EXIT_SUCCESS);
        if (setsid () == -1) {
                perror ("setsid"); return -1;
}
```

Получение текущего идентификатора сессии менее полезно, но тоже возможно:

```
#define _XOPEN_SOURCE 500 #include <unistd.h>
pid_t getsid (pid_t pid);
```

Вызов getsid() возвращает идентификатор сессии процесса, определенного через параметр pid. Если аргумент pid равен 0, вызов возвращает идентификатор сессии вызывающего процесса. В случае ошибки вызов возвращает -1. Единственная возможная величина кода ошибки — ESRCH, означающая, что нет доступных процессов с соответствующим pid. Обратите внимание, что другие системы UNIX также могут устанавливать в errno код ошибки EPERM, означающий, что pid и вызывающий процесс не принадлежат одной сессии; Linux не возвращает эту ошибку и успешно доставляет идентификатор сессии любого процесса.

Используется getsid() нечасто и, как правило, в диагностических целях:

```
pid_t sid;
sid = getsid (0);
if (sid == -1)
        perror ("getsid"); /* должно быть невозможно */
```

```
else
        printf ("Мой идентификатор сеанса^^", sid);
```

2.19 Системные вызовы групп процессов

Вызов setpid() устанавливает идентификатор группы процессов pgid процессу, определенному через pid:

#define _XOPEN_SOURCE 500 #include <unistd.h>

int setpgid (pid_t pid, pid_t pgid);

Если pid равен 0, то используется текущий процесс. Если pgid равен 0, идентификатор процесса, определенного через pid, используется в качестве идентификатора группы процессов.

В случае успеха setpid() возвращает 0. Успех зависит от нескольких условий: О процесс, определенный через pid, должен быть вызывающим процессом или потомком вызывающего процесса, который еще не выполнил вызов exec и принадлежит той же сессии, что и вызывающий процесс;

- процесс, определенный с помощью pid, не должен быть лидером сессии;
- если pgid уже существует, он должен находиться в той же самой сессии, что и вызывающий процесс;
- pgid должен быть неотрицательным.

В случае ошибки вызов возвращает -1 и устанавливает в errno один из следующих кодов ошибки:

- EACCESS — процесс, определенный через pid, является потомком вызывающего процесса, который уже вызвал exec;
- EINVAL — pgid меньше 0;
- EPERM — процесс, определенный через pid, является лидером сессии или находится в другой сессии по отношению к вызывающему процессу; кроме того,
- может быть предпринята попытка переместить процесс в группу процессов, принадлежащей другой сессии;
- ESRCH — pid не принадлежит текущему процессу либо потомку текущего процесса или не равен 0.

Как и с сессиями, получение идентификатора группы процессов, в которую входит определенный процесс, менее полезно, но возможно:

#define _XOPEN_SOURCE 500 #include <unistd.h>

pid_t getpgid (pid_t pid);

Вызов getpig() возвращает идентификатор группы процессов для процесса, определенного через pid. Если pid равен 0, возвращается идентификатор группы процессов для текущего процесса. В случае ошибки он возвращает -1 и устанавливает errno равным ESRCH, единственному возможному коду ошибки, означающему, что использована недопустимая величина pid.

Как и getsid(), getpig() используется обычно в диагностических целях:

```
pid_t pgid; pgid = getpgid (0);
if (pgid == -1)
        perror ("getpgid"); /* не должно быть возможно */
        else
        printf ("Идентификатор группы процессов paBeH=%d\n",
pgid);
```

2.20 Задание для самостоятельно работы

Рзработать программу на языке программирования С, осуществляющую работу с процессами в соотвествии с индивидуальным вариантом. Программа должна собираться с помощью компилятора GCC в операционной системе симейства Linux. Индивидуальный враиант взять у преподавателя.

Контрольные вопросы

1) Что такое процесс?
2) Что таоке бинарный модуль?
3) Что такое идентификатор процесса?
4) По какому правилу назначаются идентификаторы процессов?
5) Какой процесс в Linux имеет идентификатор процесс равный 0?
6) Какой процесс в Linux имеет идентификатор процесс равный 1?
7) Что такое иерархия процессов в Linux?
8) Для чего предназначен тип данных pid_t?
9) В каком заголовочном файле описан тип данных pid_t?
10) С помощью какой функции можно получить идентификатор текущего процесса?

11) С помощью каких функций в Linux можно запустить новый процесс?
12) Для чего прежназначен системный вызов fork?
13) В чем отличие системных вызовов fork и vfork?
14) С помощью каких функций в Linux можно завершить процесс?
15) Каким образом можно организовать ожидание завершения дочерних процессов?
16) Как можно осуществить ожидание завершения определенного процесса?
17) Для чего предназначена структура rusage?
18) Какие процессы называюся «зомби»?
19) Как задаюся права доступа процессов?
20) Как задается владелец процессов?
21) Как задаются пользователи и группы для процессов?
22) Что такое сохраненные пользовательские идентификаторы?
23)Что такое сессия процессов?
24) Какие системные вызовы позволяют работать с сессиями процессов?
25) Какие системные вызовы позволяют работать с группами процессов?

Тема 3. Работа с файлами и их метаданными

Каждый файл представляется структурой inode — индексного дескриптора, которой присваивается уникальная в данной файловой системе численная величина, которая называется номером индексного дескриптора. Индексный дескриптор — и физический объект, находящийся на диске файловой системы UNIX, и концептуальная сущность, представляемая структурой данных ядра Linux. Индексный дескриптор хранит метаданные, связанные с файлом, такие как права доступа к файлу, время последнего доступа, владелец, группа, размер, а также размещение данных файла1.

Получить номер индексного дескриптора можно с помощью команды ls и флага -i:

```
$ ls -i
1689459 Kconfig1689461    main.c          1680144    process.c
        1689464 swsusp.c
1680137 Makefile      1680141    pm.c       1680145    smp.c
        1680149 user.c
1680138 console.c     1689462    power.h    1689463    snapshot.c
        1689460 disk.c    1680143    poweroff.c 1680147    swap.c
```

Этот вывод показывает, что, например, disk.c имеет номер индексного дескриптора 1689460. В этой конкретной файловой системе ни один другой файл не может иметь такого же номера. О другой файловой системе, однако, мы не можем утверждать этого.

3.1 Семейство stat

UNIX предоставляет семейство функций для получения метаданных файла:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *path, struct stat *buf);
```

Каждая из этих функций возвращает информацию о файле. Функция stat() предоставляет информацию о файле, определенном

через path, в то время как fstat() возвращает информацию о файле, представленном файловым дескриптором fd. Функция lstat() идентична stat(), за исключением того, что в случае передачи ей символической ссылки lstat() возвращает информацию о самой ссылке, а не о конечном файле.

Каждая из этих функций хранит информацию в структуре stat, которая предоставляется пользователям. Структура stat определена в <bits/stat.h>, включенном в <sys/stat.h>:

```
struct stat {
        dev_t st_dev;       /*идентификатор устройства, на котором хранится файл */
        ino_t st_ino;       /*номер индексного дескриптора*/
        mode_t st_mode; /*разрешения */
        nlink_t st_nlink;  /*количество жестких ссылок */
        uid_t st_uid;       /* пользовательский идентификатор владельца */
        gid_t st_gid;       /* групповой идентификатор владельца */
        dev_t st_rdev;      /* идентификатор устройства (для специальных файлов */
        off_t st_size;      /* общий размер в байтах */
        blksize_t st_blksize;   /* размер блока для ввода-вывода в файловой системе */
        blkcnt_t st_blocks;     /*  количество выделенных блоков */
        time_t st_atime;  /*  время последнего доступа */
        time_t st_mtime; /*  время последней модификации */
        time_t st_ctime;  /*  время последнего изменения метаданных */
};
```

Рассмотрим поля подробнее.

- Поле st_dev описывает узел устройства, на котором хранится файл (позже мы поговорим об узлах устройств). Если файл не поддерживается устройством, а находится, например, на ресурсе NFS, эта величина равна 0.
- Поле st_ino field предоставляет номер индексного дескриптора файла.

- Поле st_mode предоставляет байты режима файла, которые описывают тип файла (например, это файл или директория), а также разрешения доступа (например, доступен всем только для чтения). О байтах режима и разрешениях мы говорили в гл. 1 и 2.
- Поле st_nlink приводит количество жестких ссылок, связанных с файлом. Каждый файл в файловой системе имеет хотя бы одну жесткую ссылку.
- Поле st_uid приводит идентификатор пользователя, который владеет файлом.
- Поле st_gid приводит идентификатор группы, которая владеет файлом.
- Если файл является узлом устройства, то поле st_rdev описывает устройство, которое представляет этот файл.
- Поле st_size представляет размер файла в байтах.
- Поле st_blksize описывает предпочитаемый объем блока, достаточный для ввода-вывода файла. Эта величина (или кратное ей значение) — оптимальный размер блока для ввода-вывода с пользовательской буферизацией (см. гл. 3).
- Поле st_blocks предоставляет количество блоков в файловой системе, занимаемых файлом. Эта величина, умноженная на размер блока, будет всегда меньше, чем значение, предоставленное st_size, если у файла есть дыры (то есть если файл разреженный).
- Поле st_atime содержит время последнего доступа к файлу. Это самое позднее время, когда к файлу предоставлялся доступ (например, через read() или execle())
- Поле st_mtime содержит время последней модификации файла, то есть время, когда в файл была сделана последняя запись.
- Поле st_ctime содержит время последнего изменения файла. Поле содержит время, когда метаданные файла (например, его владелец или права доступа) менялись в последний раз. Его часто путают со временем создания файла, которое не сохраняется в Linux и других UNIX-подобных системах.

В случае успеха все три вызова возвращают 0 и сохраняют метаданные файла
в структуру, предоставленную stat. При ошибке они возвращают -1 и присваивают
errno одно из следующих значений:

- EACCES — вызывающий процесс не имеет полномочий на поиск для одного из каталогов — компонентов пути path (касается только stat() и lstat());
- EBADF — значение fd является недопустимым (касается только fstat());
- EFAULT — указатели path или buf являются некорректными;
- ELOOP — путь содержит слишком много символических ссылок (касается только stat() и lstat());
- ENAMETOOLONG — path слишком велик (касается только stat() и lstat());
- ENOENT — один из компонентов path не существует (касается только stat() и lstat());
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один из компонентов path не является директорией (касается только stat() и lstat()).

Следующая программа использует stat() для получения размера файла, указанного в командной строке:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
        struct stat sb; int ret;
        if (argc < 2) {
                fprintf (stderr,"usage: %s <file>\n", arg
                return 1;
        }
ret = stat (argv[1], &sb);
if (ret) {
        perror ("stat"); return 1;
}
printf ("%s is %ld bytes\n", argv[1], sb.st_size); return 0;
}
```

Запуск программы относительного ее собственного файла приводит к следующему результату:

```
$ ./stat stat.c stat.c is 392 bytes
```

Эта программа, в свою очередь, выводит тип файла (такого как символическая ссылка или блокирующий узел устройства), указанного как первый аргумент программы:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
        struct stat sb; int ret;
        if (argc < 2) {
                fprintf (stderr,"usage: %s <file>\n", argv[0]);
                return 1;
        }
ret = stat (argv[1], &sb); if (ret) {
perror ("stat"); return 1;
}
printf ("Тип файла: "); switch (sb.st_mode & S_IFMT) { case
S_IFBLK:
printf("блокирующий узел устройства^"); break; case S_IFCHR:
printf("текстовый идентификатор устройства^"); break; case
S_IFDIR:
printf("каталог\n"); break; case S_IFIFO:
printf("FIFO\n"); break; case S_IFLNK:
printf("символическая ссылка^"); break; case S_IFREG:
printf("обычный файл\n"); break; case S_IFSOCK:
printf("программный интерфейс^"); break;
default:
printf("неизвестный\n");
break;
}
return 0;
}
```

Наконец, вот этот фрагмент кода использует fstat(), чтобы проверить, находится ли уже открытый файл на физическом (или, напротив, на сетевом) устройстве:

```
/*
```

• is_on_physical_device - возвращает положительное целое число,

• если файл с дескриптором fd находится на физическом устройстве.

• 0 если файл находится на нефизическом или виртуальном устройстве

• (например, связанном ресурсе NFS)

• и -1 в случае ошибки.

```
*/
int is_on_physical_device (int fd)
{
        struct stat sb; int ret;
        ret = fstat (fd, &sb); if (ret) {
        perror ("fstat"); return -1;
}
return gnu_dev_major (sb.st_dev);
}
```

3.2 Разрешения

С помощью вызовов stat можно получить значения разрешений для данного файла, но для их изменения используются два других системных вызова:

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

int chmod (const char *path, mode_t mode); int fchmod (int fd, mode_t mode);

Оба вызова chmod() и fchmod() устанавливают mode в качестве значения системных разрешений. В вызове chmod() относительный или абсолютный путь к файлу указывается в переменной path. В fchmod() файл определяется файловым дескриптором fd.

Допустимые величины mode, представленные непрозрачным целочисленным типом mode_t, те же, что возвращаются полем ds_mode в структуре stat. Значения принадлежат к типу простых целочисленных, однако их значения специфичны для каждой реализации UNIX. Следовательно, в POSIX определяется набор постоянных, представляющий все разнообразие переменных

(подробно об этом было рассказано в подразд. «Права доступа новых файлов» разд. «Открытие файлов» гл. 2). Эти постоянные могут быть объединены между собой с помощью двоичного «ИЛИ», чтобы сформировать допустимые значения mode. Например, (S_IRUSR | S_IRGRP) устанавливает значение, разрешающее чтение файла его владельцу и группе.

Для изменения разрешений файла действительный идентификатор процесса, вызывающего chmod() или fchmod(), должен соответствовать владельцу файла или процесс должен обладать свойством CAP_FOWNER.

В случае успеха оба вызова возвращают 0, при неудаче возвращают -1, а errno присваивается одно из следующих значений:

- EACCES — вызывающий процесс не имеет полномочий на поиск для одного из каталогов — компонентов пути path (касается только chmod());
- EBADF — значение fd является недопустимым (касается только fchmod());
- EFAULT — указатель path является некорректным (касается только chmod());
- EIO — в системе произошла внутренняя ошибка ввода-вывода; это очень плохое значение ошибки, которое может указывать на повреждение диска или файловой системы;
- ELOOP — путь содержит слишком много символических ссылок (касается только chmod());
- ENAMETOOLONG — path слишком велик (касается только chmod());
- ENOENT — путь path не существует (касается только chmod());
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один из компонентов path не является каталогом (касается только chmod());
- EPERM — действительный идентификатор вызывающего процесса не соответствует владельцу файла, а процесс не имеет свойства CAP_FOWNER;
- EPOFS — файл находится в файловой системе, доступной только для чтения.

Данный фрагмент кода устанавливает файлу map. png разрешения на считывание и запись для владельца: int ret;

```
/*
        Установка файлу 'map.png' в текущем каталоге разрешений
        на считывание и запись для владельца файла. Аналогично
команде
        chmod 600 ./map.png'.
*/
ret = chmod ("./map.png", S_IRUSR | S_IWUSR);
if (ret)
        perror ("chmod");
```

Данный фрагмент делает то же, предполагая, что fd представляет открытый файл map.png: int ret;

```
/*
  • Установка для файла с дескриптором fd разрешений на
чтение
  • и запись для владельца файла.
*/
ret = fchmod (fd, S_IRUSR | S_IWUSR);
if (ret)
        perror ("fchmod");
```

Оба вызова, chmod() и fchmod(), доступны во всех современных системах UNIX. Стандарт POSIX требует реализации первого, а второй считает необязательным.

3.3 Владение

В структуре stat поля st_uid и st_gid представляют владельца и группу файла соответственно. Три системных вызова позволяют пользователю изменить эти два значения:

```
#include <sys/types.h>
#include <unistd.h>
int chown (const char *path, uid_t owner, gid_t group);
int lchown (const char *path, uid_t owner, gid_t group);
int fchown (int fd, uid_t owner, gid_t group);
```

Вызовы chown() и lchown() устанавливают владение для файла, указанного с помощью пути path. Они работают одинаково для всех

файлов за исключением символических ссылок: первый вызов следует по символической ссылке и изменяет владение целевого файла, а не самой ссылки; lchown() не переходит по ссылке, а вместо этого изменяет владение самого файла символической ссылки. Наконец, fchown() устанавливает владение файла, указанного с помощью файлового дескриптора fd.

В случае успеха все три системных вызова делают владельцем файла owner, группой файла — group и возвращают 0. Если параметры owner или group равны -1, эта величина не устанавливается. Только процесс со свойством CAP_CHOWN (обычно это процесс с правами root) может менять владельца файла. Владелец файла может изменить группу файла на любую другую, членом которой является пользователь; процессы со свойством CAP_CHOWN могут изменять группу файла без ограничений.

В случае неудачи вызов возвращает -1 и присваивает errno одно из следующих значений:

- EACCES — вызывающий процесс не имеет полномочий на поиск для одного из каталогов — компонентов пути path (касается только chown() и lchown());
- EBADF — значение fd является недопустимым (касается только fchown());
- EFAULT — указатель path является некорректным (касается только chown() и lchown());
- EIO — в системе произошла внутренняя ошибка ввода-вывода (очень плохо);
- ELOOP — путь содержит слишком много символических ссылок (касается только chown() и lchown());
- ENAMETOOLONG — path слишком велик (касается только chown() и lchown());
- ENOENT — путь path не существует;
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один из компонентов path не является каталогом (касается только chown() и lchown());
- EPERM — вызывающий процесс не имеет прав, необходимых для изменения владельцев группы так, как запрошено;
- EPOFS — файл находится в системе, доступной только для чтения.

Данный фрагмент кода изменяет группу файла manifest.txt в текущем рабочем каталоге на группу officers. Чтобы проделать это, вызывающий пользователь должен обладать свойством CAP_CHOWN либо быть пользователем kidd и состоять в группе officers.

```
struct group *gr; int ret;
/*
   • getgrnam() возвращает информацию о группе,
   • принимая в качестве параметра ее имя.
*/
gr = getgrnam ("officers");
if (!gr) {
        /* скорее всего, недопустимая группа */ perror ("getgrnam");
return 1;
}
/* изменяем группу файла manifest.txt's на 'officers' */
ret = chown("manifest.txt", -1, gr->gr_gid);
if (ret)
perror ("chown");
```

Перед исполнением вызова файл принадлежит группе crew:

```
$ ls -l
-rw-r--r-- 1 kidd crew 13274 May 23 09:20 manifest.txt
```

После исполнения файл принадлежит только группе officers:

```
$ ls -l
-rw-r--r-- 1 kidd officers 13274 May 23 09:20 manifest.txt
```

Владелец файла, kidd, не изменился, так как в коде для параметра uid было передано значение -1.

Эта функция устанавливает файлу, определенному fd, пользователя-владельца и группу-владельца root:

```
/*
   • make_root_owner - изменяет владельца и группу файла,
   • определенного 'fd', на root. Возвращает 0 в случае успеха
   • и -1 в случае неудачи.
*/
int make_root_owner (int fd)
{
      int ret;
```

```
        /* 0 является идентификатором и группы, и пользователя
для root */
        ret = fchown (fd, 0, 0); if (ret)
        perror ("fchown"); return ret;
}
```

Вызывающий процесс должен обладать свойством CAP_CHOWN. В отношении характеристик это обычно означает, что он должен принадлежать root.

3.4 Расширенные атрибуты

Расширенные атрибуты представляют собой механизм для создания постоянных связей между файлами и парами «ключ — значение». В этой главе мы уже обсуждали разные виды метаданных «ключ — значение», связанных с файлами: размер, владелец, момент последней модификации файла и т. д. Расширенные атрибуты позволяют существующей файловой системе поддерживать новые функции, не предусмотренные изначально, например обязательный контроль доступа в целях обеспечения безопасности. Особенно интересно в расширенных атрибутах то, что приложения из пользовательского пространства могут самостоятельно создавать, считывать и записывать пары «ключ — значение».

Расширенные атрибуты не связаны с файловой системой в том смысле, что эти приложения используют стандартный интерфейс для управления ими; интерфейс не является специфичным для какой-либо файловой системы. Приложения, таким образом, могут использовать расширенные атрибуты, не заботясь, в какой файловой системе находятся файлы или как эта система хранит свои ключи и значения. Однако все-таки внедрение расширенных атрибутов значительно зависит от файловой системы. Разные файловые системы хранят расширенные атрибуты различными способами, но ядро скрывает эти различия, абстрактно представляя их с помощью системы интерфейсов расширенных атрибутов.

Файловая система ext4, например, хранит расширенные атрибуты файлов в пустом пространстве внутри индексного дескриптора файла1. Эта функциональность делает считывание расширенных атрибутов значительно быстрее. Поскольку блок

файловой системы, содержащий индексный дескриптор, считывается с диска в память, как только приложение обращается к файлу, то и расширенные атрибуты «автоматически» отправляются в память и могут быть доступны без каких-либо дополнительных затрат.
Другие файловые системы, например FAT и minixfs, не поддерживают расширенные атрибуты вообще. Эти системы возвращают ENOTSUP, если к файлам направляется действие, связанное с расширенными атрибутами.

Уникальный ключ идентифицирует каждый расширенный атрибут. Ключи должны соответствовать кодировке UTF-8. Каждый ключ обладает структурой про- странство_имен.атрибут. Каждый ключ должен быть полностью уточнен; это значит, что он должен начинаться с указания допустимого пространства имен, после которого следует точка. Пример допустимого названия ключа — user.mime_type; этот ключ находится в пользовательском пространстве user и обладает атрибутом с названием mime_type.
Старые и новые способы хранить типы MIME в файловой системе
Файловые менеджеры с графическим интерфейсом наподобие GNOME подходят по-разному к файлам различных типов: они предлагают уникальные значки, различное поведение по умолчанию при щелчке на файле, специальные списки доступных для выполнения операций и т. д. Чтобы обеспечить все это, файловый менеджер должен знать формат каждого файла. Для определения формата системы наподобие Windows просто смотрят на расширение файла. Системы UNIX, однако, из соображений как безопасности, так и сохранения традиций стараются исследовать каждый файл и интерпретировать его тип. Этот процесс называется исследованием типа MIME.

Некоторые файловые менеджеры генерируют такую информацию на лету, то есть при любой необходимости, другие — лишь однажды, затем кэшируя ее. Предпочитающие кэширование обычно помещают информацию в собственную базу данных. Файловый менеджер должен постоянно держать эту базу синхронизированной с файлами, которые могут измениться, не уведомляя менеджера базы, поэтому лучше просто отказаться от использования базы данных и держать такие метаданные в

расширенных атрибутах: их проще поддерживать, быстрее считывать, а также они доступны из любого приложения.

Ключ может быть определенным или неопределенным. Если ключ является определенным, его значение может быть пустым или непустым. Таким образом, есть разница между определенным ключом, которому не назначено значение, и неопределенным. Как вы скоро узнаете, это значит, что требуется специальный интерфейс для удаления ключей, а просто назначить им пустое значение недостаточно.

Величина, ассоциированная с ключом, если она непустая, может быть любым произвольным массивом байтов. Поскольку эта величина не всегда представляет собой текстовую строку, не обязательно завершать ее нулем, хотя это может быть разумно, если вы решите сохранить в качестве значения ключа строку C. Считывая атрибут, ядро предоставляет размер; записывая атрибут, вы должны предоставить размер самостоятельно.

Linux не ограничивает количество ключей, их длину, размер значения или общее пространство, которое может быть занято всеми ключами или значениями, связанными с файлом. Файловые системы, однако, имеют некоторые технические пределы. Как правило, они ограничивают общий размер всех ключей и значений, связанных с данным файлом.

В ext3, например, все расширенные атрибуты для данного файла должны укладываться в пустое пространство внутри индексного дескриптора файла и занимать в файловой системе не более одного блока (более старые версии ext3 имели ограничение до одного блока в файловой системе без дополнительного хранилища внутри индексного дескриптора). На практике это эквивалентно ограничению приблизительно 1-8 Кбайт на файл в зависимости от размера блоков в файловой системе. В XFS, наоборот, практически не имеется ограничений.

Впрочем, даже в ext3 лимиты, как правило, не представляют проблемы, поскольку большинство ключей и значений — короткие текстовые строки. Однако забывать о них все равно нельзя — дважды подумайте перед тем, как сохранить всю историю управления версиями проекта в расширенных атрибутах файла!

Пространства имен расширенных атрибутов

Пространства имен, связанные с расширенными атрибутами, представляют собой нечто большее, чем просто организационные средства. Ядро реализует различные политики доступа в зависимости от вида пространства имен.

Linux в настоящее время определяет четыре пространства имен для расширенных атрибутов и может добавить новые в будущем. Существующие четыре — следующие.

- system — это пространство используется для реализации функциональностей ядра, использующих расширенные атрибуты, таких как списки управления доступом (ACLs).

Примером расширенного атрибута в этом пространстве имен может быть system.posix_acl_access. Могут ли пользователи считывать или записывать информацию в эти атрибуты, зависит от используемого модуля безопасности. Предполагайте худший вариант — никакие пользователи (включая root) не могут даже читать эти атрибуты.

- security — данное пространство используется для внедрения модулей безопасности, например SELinux. Как и в предыдущем пространстве, возможность для пользователей считывать или записывать информацию в эти атрибуты зависит от используемого модуля безопасности. По умолчанию все процессы могут считывать свои атрибуты, но только процесс с CAP_SYS_ADMIN может записывать в них данные.
- trusted — это пространство имен хранит закрытую для доступа информацию в пользовательском пространстве. Только процесс со свойством CAP_SYS_ADMIN может считывать и записывать эти атрибуты.
- user — данное пространство имен — стандартное для использования большинством обычных процессов. Ядро управляет доступом к этому пространству через обычные биты разрешений для файлов. Чтобы считать значение существующего ключа, процесс должен иметь право чтения данного файла.

Чтобы создать новый ключ или записать значение в существующий, процесс должен иметь право записи в данный файл. Вы можно назначить расширенные атрибуты в пользовательском пространстве имен только обычным файлам, но не символическим ссылкам или файлам устройств. Разрабатывая приложение для

пользовательского пространства, использующее расширенные атрибуты, скорее всего, вы выберете именно это пространство.

3.5 Действия с расширенными атрибутами

POSIX определяет четыре действия, которые приложение может проделать с расширенными атрибутами данного файла:

- для указанного файла возвращается список всех ключей расширенных атрибутов, назначенных файлу;
- для заданных файла и ключа возвращаются соответствующие величины;
- для известных файла, ключа и значения можно назначить это значение известному ключу;
- для указанных файла и ключа можно удалить расширенный атрибут из файла.

Для каждого из этих действий POSIX предлагает три системных вызова:

- версию, которая работает с указанным именем файла; если путь — символическая ссылка, то действие производится над объектом ссылки (обычное поведение); О версию, работающую с указанным путем к файлу; если путь ведет к символической ссылке, действие производится над ней (стандартный l - вариант системного вызова);
- версию, работающую с файловым дескриптором (стандартный f-вариант).

Самое простое действие — получение значения расширенного атрибута из файла по известному ключу:

```
#include <sys/types.h>
#include <attr/xattr.h>
ssize_t getxattr (const char *path, const char *key, void *value, size_t size);
ssize_t lgetxattr (const char *path, const char *key, void *value, size_t size);
ssize_t fgetxattr (int fd, const char *key, void *value, size_t size);
```

Успешный вызов getxattr() хранит расширенный атрибут с именем key из файла path в предоставленном пользователем буфере

value, который имеет длину size байт. Он возвращает значение величины.

Если размер size равен 0, то вызов возвращает размер величины без ее сохранения в буфере value. Таким образом, установка 0 позволяет приложениям определить точный размер буфера, в котором хранится значение ключа. Передавая этот размер, приложения могут затем передавать или менять размер буфера.

Lgetxattr() ведет себя аналогично getxattr() за исключением случая, когда path — символическая ссылка. Тогда вызов возвращает расширенные атрибуты самой ссылки, а не ее целевого объекта. Как вы помните из предыдущего раздела, атрибуты в пользовательском именном пространстве не применяются к символическим ссылкам, следовательно, этот вызов используется редко.

Fgetxattr() работает с файловым дескриптором fd; в остальном он ведет себя аналогично getxattr().

В случае ошибки все три вызова возвращают -1 и устанавливают errno одно из следующих значений:

- EACCES — вызывающий процесс не имеет прав доступа для поиска в одной из папок- компонентов пути path (касается только getxattr() и lgetxattr());
- EBADF — дескриптор некорректен (касается только fgetxattr());
- EFAULT — указатели path, key, value являются недопустимыми;
- ELOOP — путь path содержит слишком много символических ссылок (касается только getxattr() и lgetxattr());
- ENAMETOOLONG — путь path слишком велик (касается только getxattr() и lget- xattr());
- ENOATTR — атрибута key не существует или процесс не имеет доступа к нему;
- ENOENT — какого-либо компонента пути не существует (касается только getxattr() и lgetxattr());
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один из компонентов path не является каталогом (касается только getxattr() и lgetxattr());
- ENOTSUP — файловая система, в которой находится путь path с дескриптором fd, не поддерживает расширенные атрибуты;
- ERANGE — размер size слишком мал, чтобы сохранить величину key; как было сказано выше, вызов может быть повторен

с size, равным 0; возвращаемая величина будет означать требуемый размер буфера, и величина value может быть изменена соответственно.

Следующие три системных вызова устанавливают указанный расширенный атрибут:

#include <sys/types.h>

#include <attr/xattr.h>

int setxattr (const char *path, const char *key, const void *value, size_t size, int flags);

int lsetxattr (const char *path, const char *key, const void *value, size_t size, int flags);

int fsetxattr (int fd, const char *key, const void *value, size_t size, int flags);

Успешный вызов setxattr() устанавливает расширенный атрибут key файлу path равным value, имеющего size байт длины. Поле flags управляет поведением вызова. Если в flags прописано XATTR_CREATE (создание расширенного атрибута) и расширенный атрибут уже существует, вызов приведет к ошибке. Если в flags прописано XATTR_REPLACE (замещение расширенного атрибута), а расширенных атрибутов еще нет, вызов приведет к ошибке. По умолчанию, если в fl ags указан 0, позволяются и создание, и замена расширенных атрибутов. Независимо от значения flags, никакие ключи, кроме указанных в key, не затрагиваются.

Вызов lsetxattr() работает аналогично setxattr(), за исключением случая, когда path — символическая ссылка. Тогда вызов устанавливает расширенные атрибуты самой ссылки, а не ее целевого объекта. Как вы помните из предыдущего раздела, атрибуты в пользовательском именном пространстве не применяются к символическим ссылкам, следовательно, этот вызов используется редко.

Fsetxattr() работает с файловым дескриптором fd; в остальном он ведет себя аналогично getxattr().

В случае успеха все три системных вызова возвращают 0; при неудаче вызовы возвращают -1 и присваивают errno одно из следующих значений:

- EACCES — вызывающий процесс не имеет прав доступа для поиска в одной из папок- компонентов пути path (касается только setxattr() и lsetxattr());
- EBADF — дескриптор некорректен (касается только fgetxattr());
- EDQUOT — предельный размер квоты не позволяет использовать пространство, необходимое для выполнения запрошенной операции;
- EEXIST — значение flags установлено XATTR_CREATE, но у данного файла уже существует ключ key;
- EFAULT — указатели path, key, value являются недопустимыми;
- ELOOP — путь path содержит слишком много символических ссылок (касается только setxattr() и lsetxattr());
- ENAMETOOLONG — путь path слишком велик (касается только setxattr() и lsetxattr());
- ENOATTR — значение flags установлено XATTR_REPLACE, но у данного файла не существует ключа key;
- ENOENT — какого-либо компонента пути не существует (касается только setxattr() и lsetxattr());
- ENOMEM — недостаточно доступной памяти для выполнения запроса;
- ENOSPC — недостаточно пространства в файловой системе для хранения расширенного атрибута;
- ENOTDIR — один из компонентов path не является каталогом (касается только setxattr() и lsetxattr());
- ENOTSUP — файловая система, в которой находится путь path с дескриптором fd, не поддерживает расширенные атрибуты.

Следующие три системных вызова выводят весь набор ключей расширенных атрибутов, назначенных данному файлу:

```
#include <sys/types.h>
#include <attr/xattr.h>
ssize_t listxattr (const char *path, char *list, size_t size);
ssize_t llistxattr (const char *path,char *list, size_t size);
ssize_t flistxattr (int fd, char *list, size_t size);
```

Успешный вызов listxattr() возвращает список ключей расширенных атрибутов, связанных с файлом, определенным через path. Список хранится в буфере list, размер которого равен size байт. Системный вызов возвращает фактический размер списка в байтах.

Каждый ключ расширенного атрибута, возвращенного в списке list, заканчивается нулевым символом, поэтому список выглядит примерно так:
"user.md5_sum\0user.mime_type\0system.posix_acl_default\0"

Таким образом, хотя каждый ключ представляет собой традиционную, оканчивающуюся нулем строку С, чтобы пройтись по списку ключей, нужно знать его длину (которая доступна из возвращаемой вызовом величины). Чтобы определить размер буфера, необходимый для размещения списка, следует вызвать одну из списочных функций с параметром size, равным 0. Это заставит функцию вернуть актуальную длину полного списка ключей. Как и с getxattr(), приложения могут использовать эту функциональность для размещения или изменения размера буфера для передачи value.

Llistxattr() ведет себя аналогично listxattr(), за исключением случая, когда path — символическая ссылка. Тогда вызов обрабатывает расширенные атрибуты самой ссылки, а не ее целевого объекта. Как вы помните из предыдущего раздела, атрибуты в пользовательском именном пространстве не применяются к символическим ссылкам, следовательно, этот вызов используется редко.

Flistxattr() работает с файловым дескриптором fd; в остальном он ведет себя аналогично listxattr().

В случае успеха все три вызова возвращают -1 и присваивают errno один из следующих кодов ошибки:

- EACCES — вызывающий процесс не имеет прав доступа для поиска в одной из папок- компонентов пути path (касается только listxattr() и llistxattr());
- EBADF — дескриптор имеет недопустимое значение (касается только flistxattr()); О EFAULT — указатели path или list недопустимы;
- ELOOP — путь path содержит слишком много символических ссылок (касается только listxattr() и llistxattr());
- ENAMETOOLONG — путь path слишком велик (касается только listxattr() и llist- xattr());
- ENOENT — какого-либо компонента пути не существует (касается только listxattr() и llistxattr());
- ENOMEM — недостаточно памяти для выполнения запроса;

- ENOTDIR — один из компонентов path не является каталогом (касается только listxattr() и llistxattr());
- ENOTSUP — файловая система, в которой находится путь path с дескриптором fd, не поддерживает расширенные атрибуты;
- ERANGE — size не равен 0, но недостаточно велик, чтобы вместить полный список ключей; приложение может повторить вызов с size, равным 0, чтобы выяснить размер списка, а затем изменить размер value и повторить системный вызов.

Следующие три системных вызова удаляют указанный ключ из указанного файла:

```
#include <sys/types.h>
#include <attr/xattr.h>
int removexattr (const char *path, const char *key);
int lremovexattr (const char *path, const char *key);
int fremovexattr (int fd, const char *key);
```

Успешный вызов removexattr() удаляет расширенный атрибут key из файла path. Вспомните, что есть разница между неопределенным ключом и определенным ключом с пустым (нулевой длины) значением.

Вызов lremovexattr() ведет себя аналогично removexattr(), за исключением случая, когда path — символическая ссылка. Тогда вызов удаляет ключ атрибута самой ссылки, а не ее целевого объекта. Как вы помните из предыдущего раздела, атрибуты в пользовательском именном пространстве не применяются к символическим ссылкам, следовательно, этот вызов используется редко.

Fsetxattr() работает с файловым дескриптором fd; в остальном он ведет себя аналогично getxattr().

В случае успеха все три системных вызова возвращают 0; при неудаче вызовы возвращают -1 и присваивают errno одно из следующих значений:

- EACCES — вызывающий процесс не имеет прав доступа для поиска в одной из папок- компонентов пути path (касается только removexattr() и lremove- xattr());
- EBADF — дескриптор имеет недопустимое значение (касается только fremove- xattr());
- EFAULT — указатели path или key являются недопустимыми;

- ELOOP — путь path содержит слишком много символических ссылок (касается только removexattr() и lremovexattr());
- ENAMETOOLONG — путь path слишком велик (касается только removexattr() и lremo- vexattr());
- ENOATTR — для данного файла key не существует;
- NOENT — какой-либо компонент пути не существует (касается только remo- vexattr() и lremovexattr());
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один из компонентов path не является каталогом (касается только removexattr() илremovexattr());
- ENOTSUP — файловая система, в которой находится путь path с дескриптором fd, не поддерживает расширенные атрибуты.

3.6 Задание для самостоятельно работы

Разработать программу на языке программирования С, которая с использованием системных вызовов осуществляет работу с файлами и их метаданными в соотвествии с индивидуальным вариантом. Программа должна собираться компилятормо gcc и работать в операционныз системах семейства Linux. Индивидуальный вариант взять у преподавателя.

Контрольные вопросы

1) Что такое файл?
2) Что такое метаданные файла?
3) Для чего предназначено семейство системных вызовов stat?
4) В каком заголовочном файле содержаться прототипы системных вызовов семейства stat?
5) Что хранит структура stat?
6) Какие ошибки может возвращать семейство системных вызовов stat?
7) С помощью какой функции можно установить разрешения файла?
8) Какие ошибки может возвращать системный вызов chmod?
9) Как задается владение файлами?
10) Какие ошибки могут возвращать системные вызовы chown, lchown и fchown?
11) Что такое расширенные атрибуты?

12) С помощью каких системных вызовов осуществляется работа с расширенными атрибутами?
13) Какие ошибки могут возвращать системные вызовы getxattr, lgetxattr и fgetxattr?
14) Какие ошибки могут возвращать системные вызовы setxattr, lsetxattr и fsetxattr?
15) Для чего предназначены ситемные вызовы removexattr, lremovexattr и fremovexattr?
16) Какие ошибки могут возвращать системные вызовы removexattr, lremovexattr и fremovexattr?

Тема 4. Работа с каталогами и ссылками.

4.1 Каталоги

Концепция каталогов в UNIX очень проста: они содержат список названий файлов, с каждым из которых связан номер индексного дескриптора. Каждое имя называется записью в каталоге, а каждая связь имени и номера индексного дескриптора именуется ссылкой. Содержимое директории, которое пользователь видит в результате выполнения команды ls, является списком всех имен файлов в этом каталоге. Когда пользователь открывает файл в данном каталоге, ядро ищет имя файла в списке данной директории, чтобы определить соответствующий номер индексного дескриптора. Затем ядро передает этот номер в файловую систему, которая использует его, чтобы определить физическое размещение файла на устройстве.

Каталоги могут также содержать другие каталоги. Подкаталогом называется каталог, находящийся внутри другого каталога. Согласно этому определению все каталоги являются подкаталогами каких-либо родительских каталогов, за исключением каталога в корне дерева файловой системы, /. Этот каталог так и называется корневым каталогом (не следует путать его с домашним каталогом пользователя root, то есть /root).

Путь к файлу состоит из имени файла вместе с одним или несколькими его родительскими каталогами. Абсолютным путем к файлу называется путь, который начинается с корневого каталога, например /usr/bin/sextant. Относительным путем называется путь, который не начинается с корневого каталога, к примеру bin/sextant. Чтобы использовать такой путь, операционная система должна знать каталог, к которому он относится. Текущий рабочий каталог (обсуждаемый в следующем разделе) используется в качестве стартовой точки.

Названия файла и папки могут содержать любые символы, кроме /, разделяющих каталоги в записи пути к файлу, и пустого значения (null), завершающего путь к файлу. Тем не менее принято ограничивать использование символов в записях пути: как правило, используются только допустимые печатные символы из текущих языковых настроек или даже только символы ASCII. Однако поскольку ни ядро, ни библиотеки C не предписывают такую

практику, разработчики приложений самостоятельно принимают решение о допустимости использования тех или иных символов.

В старых системах UNIX длина имени файла ограничивалась 14 символами. Сегодня все современные системы UNIX позволяют использовать для каждого файла по меньшей мере 255 байт. (речь идет не о 255 символах, а о 255 байтах. Многобайтовые символы, разумеется, займут больше одного байта из этих 255.) Многие файловые системы под Linux позволяют использование даже более длинных имен файлов1.

Каждая папка содержит две особые папки: . и .. (они называются «точка» и «точка-точка»). Точка — ссылка на саму папку. Точка-точка — ссылка на родительский каталог для данного каталога. Например, /home/kidd/gold/.. — то же самое, что /home/kidd. Для каталога root каталоги точка и точка-точка ссылаются сами на себя: /, /. и /.. — это одна и та же папка. Технически говоря, таким образом, root тоже является подкаталогом — в данном случае самого себя.

4.2 Текущий рабочий каталог

Каждый процесс имеет текущий каталог, который наследует непосредственно от своего родительского процесса. Этот каталог называется текущим рабочим каталогом процесса. Текущая рабочая директория является начальной точкой, из которой ядро прокладывает относительные пути к файлам. Например, если текущая рабочая директория процесса /home/blackbeard, то, когда процесс пытается открыть prrot.jpg, ядро будет стараться открыть файл /home/blackbeard/parrot.jpg. Однако если процесс попытается открыть /usr/bin/mast, то ядро в самом деле откроет /usr/bin/mast. Текущий рабочий каталог никак не влияет на абсолютные пути к файлам (то есть пути, начинающиеся с косой черты).

Процесс может получить и изменить свою рабочую папку. Предпочтительный метод получения текущего рабочего каталога — системный вызов getcwd(), регламентированный в POSIX:

```
#include <unistd.h>
char * getcwd (char *buf, size_t size);
```

Успешный вызов getcwd() копирует текущий рабочий каталог как абсолютный путь в буфер, указанный как buf и имеющий длину

size байт, и возвращает указатель к buf. В случае ошибки вызов возвращает NULL и присваивает errno одно из следующих значений:

- EFAULT — указатель buf имеет недопустимое значение;
- EINVAL — size равен 0, но buf не равен NULL;
- ENOENT — текущий рабочий каталог более не действителен; это может случиться, если текущий рабочий каталог был удален;
- ERANGE — size слишком мал, чтобы текущий рабочий каталог был сохранен в buf; приложение должно выделить больший размер буфера и попробовать снова.

Пример использования getcwd():

```
char cwd[BUF_LEN];
if (!getcwd (cwd, BUF_LEN)) {
        perror ("getcwd");
        exit (EXIT_FAILURE);
}
printf ("cwd = %s\n", cwd);
```

POSIX регламентирует, что поведение getcwd() не определено, если buf равен NULL. Библиотека C в Linux в данном случае выделит буфер длиной size байт и сохранит там текущий рабочий каталог. Если size равен 0, библиотека C выделит буфер достаточного размера, чтобы сохранить текущий рабочий каталог. Затем приложение должно очистить буфер через free(), окончив работу с ним. Это поведение уникально для Linux, поэтому нельзя полагаться на его надежную работу в условиях совместимости. Тем не менее эта функция делает использование очень простым. Пример:

```
char *cwd;
cwd = getcwd (NULL, 0); if (!cwd) {
        perror ("getcwd"); exit (EXIT_FAILURE);
}
printf ("cwd = %s\n", cwd); free (cwd);
```

Библиотека C Linux также предоставляет функцию get_current_dir_name(), которая ведет себя аналогично getcwd() при получении NULL в буфере и size, равном 0:

```
#define _GNU_SOURCE
#include <unistd.h>
char * get_current_dir_name (void);
```

Таким образом, следующий фрагмент кода делает то же, что и предыдущий:

```
char *cwd;
cwd = get_current_dir_name (); if (!cwd) {
        perror ("get_current_dir_name"); exit (EXIT_FAILURE);
}
printf ("cwd = %s\n", cwd); free (cwd);
```

В старых системах BSD пользовался популярностью вызов getwd(), который Linux поддерживает для обеспечения совместимости:

```
#define _XOPEN_SOURCE_EXTENDED /* или _BSD_SOURCE
*/
#include <unistd.h>
char * getwd (char *buf);
```

Вызов getwd() копирует текущий рабочий каталог в буфер buf, который должен быть не менее PATH_MAX длиной. Вызов возвращает buf в случае успеха и NULL в случае ошибки. Например:

```
char cwd[PATH_MAX];
if (!getwd (cwd)) {
        perror ("getwd"); exit (EXIT_FAILURE);
}
printf ("cwd = %s\n", cwd);
```

По причинам как совместимости, так и безопасности в приложениях лучше не использовать getwd() — предпочтительнее getcwd().

4.3 Изменение текущего рабочего каталога

Когда пользователь впервые авторизуется в системе, процесс авторизации устанавливает домашний каталог в качестве текущего рабочего, как указано в /etc/passwd. Иногда, однако, процессу необходимо изменить свой текущий рабочий каталог. Например, оболочка может захотеть сделать это, когда пользователь выполняет команду cd.

В Linux есть два системных вызова для изменения текущего рабочего каталога: один, который устанавливает путь к каталогу, и другой,

который прописывает файловый дескриптор, представляющий открытый каталог.

#include <unistd.h>

int chdir (const char *path); int fchdir (int fd);

Вызов chdir() изменяет текущий рабочий каталог согласно пути, указанному в path, который может быть абсолютным или относительным. Аналогично вызов fchdir() изменяет текущий рабочий каталог согласно пути, указанному через файловый дескриптор fd, который должен быть открыт для этого каталога. В случае успеха оба вызова возвращают 0, при неудаче возвращается -1.

При ошибке chdir() присваивает errno() одно из следующих значений:

- EACCES — вызывающий процесс не имеет полномочий на поиск в одном или нескольких каталогах, являющихся компонентами пути path;
- EFAULT — указатель path является недопустимым;
- EIO — произошла внутренняя ошибка ввода-вывода;
- ELOOP — ядро обнаружило в пути path слишком много символических ссылок; O ENAMETOOLONG — путь path слишком велик;
- ENOENT — каталог, указанный через path, не существует;
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOTDIR — один или несколько компонентов пути path не являются каталогом.
- Вызов fchdir() устанавливает errno одно из следующих значений:
- EACCES — вызывающий процесс не имеет полномочий на поиск в каталоге, указанном через fd (например, бит исполнения не установлен); это может случиться, если каталог верхнего уровня доступен для чтения, но не для исполнения; вызов open() будет успешным, но fchdir() — нет;
- EBADF — fd не является дескриптором открытого файла.

В зависимости от файловой системы другие значения ошибок также возможны для этого вызова.

Данные системные вызовы могут воздействовать только на процессы, запущенные в настоящее время. В UNIX нет механизма для изменения текущего рабочего каталога для другого процесса. Таким

образом, команда cd в оболочках не может быть отдельным процессом (как большинство команд), который выполняет chdir() на первом аргументе в командной строке, а затем завершается. Вместо этого cd должна быть специальной встроенной командой, которая заставляет оболочку самостоятельно вызывать chdir(), изменяя собственный текущий каталог.

Чаще всего getswd() используется для сохранения рабочего каталога, чтобы процесс мог позднее в него вернуться. Например:

```
char *swd; int ret;
/* Сохраняется текущий рабочий каталог */
swd = getcwd (NULL, 0);
if (!swd) {
        perror ("getcwd"); exit (EXIT_FAILURE);
}
/* Переход в другой рабочий каталог */
ret = chdir (some_other_dir);
if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
}
/* Выполнение какой-то работы в новом каталоге... */
/* Возврат в сохраненный каталог */
ret = chdir (swd);
if (ret) {
        perror ("chdir"); exit (EXIT_FAILURE);
}
free (swd);
```

Однако лучше сначала открыть вызовом open() текущий каталог, а затем возвращаться в него с помощью fchdir(). Это быстрее, так как ядро не сохраняет в памяти полный путь к текущему рабочему каталогу; хранится только структура inode. Следовательно, когда пользователь вызывает getcwd (), ядро должно сгенерировать путь к файлу, проходя через структуру каталога. И наоборот, открытие текущего рабочего каталога менее затратно, так как в этом случае у ядра уже есть его inode и путь к файлу, воспринимаемый человеком, не требуется. В следующем фрагменте кода использован именно этот подход:

```
int swd_fd;
swd_fd = open (".", O_RDONLY);
if (swd_fd == -1) {
        perror ("open"); exit (EXIT_FAILURE);
}
/* Переход в другой каталог */
ret = chdir (some_other_dir);
if (ret) {
        perror ("chdir"); exit (EXIT_FAILURE);
}
/* Выполнение какой-то работы в новом каталоге... */
/* Возврат в сохраненный каталог */
ret = fchdir (swd_fd);
if (ret) {
        perror ("fchdir"); exit (EXIT_FAILURE);
}
/* Закрытие дескриптора fd каталога */
ret = close (swd_fd);
if (ret) {
        perror ("close");
        exit (EXIT_FAILURE);
}
```

Именно так оболочка выполняет кэширование предыдущего каталога (например, для cd — в bash).

Процессы, которым не приходится заботиться о своем текущем рабочем каталоге, например демоны, обычно устанавливают в качестве значения корневой каталог с помощью вызова chdir("/"). Приложение, которое взаимодействует с пользователем и его данными, например текстовый редактор, чаще всего устанавливает в качестве своего текущего рабочего каталога домашний каталог пользователя или специальный каталог для документов. Понятие текущего рабочего каталога существует только в контексте относительных путей к файлам, текущий рабочий каталог наиболее полезен для утилит командной строки, которые вызывает пользователь из оболочки.

4.4 Создание каталогов

Для создания каталогов Linux предоставляет только один системный вызов, регламентированный POSIX:

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir (const char *path, mode_t mode);
```

Успешный вызов mkdir() создает путь к каталогу path, который может быть относительным или абсолютным с битами разрешения mode (как указано в текущем значении umask), и возвращает 0.

Текущее значение umask изменяет обычным способом аргумент mode, а также любые биты данного режима, уникальные в данной системе. В Linux — биты разрешения вновь создаваемого каталога (mode & ~umask & 01777). Другими словами, параметр umask для данного процесса определяет значения, которые не могут быть переопределены с помощью вызова mkdir(). Если для нового родительского каталога текущего каталога установлен общий групповой идентификатор битов (set group ID, или sgid) или файловая система смонтирована с семантикой групп BSD, то новый каталог наследует групповую принадлежность родителя. В ином случае новый каталог получит действительный идентификатор группы процесса.

При ошибке mkdir() возвращает -1 и присваивает errno одно из следующих значений:

- EACCES — текущий процесс не имеет прав записи в родительском каталоге или прав поиска в одном из каталогов-компонентов пути path;
- EEXIST — путь path уже существует (причем не обязательно в виде каталога);
- EFAULT — указатель path является некорректным;
- ELOOP — ядро обнаружило в пути path слишком много символических ссылок; Q ENAMETOOLONG — путь path слишком велик;
- ENOENT — какой-либо компонент пути path не существует или является символической ссылкой, ведущей к несуществующему объекту;
- ENOMEM — недостаточно памяти ядра для выполнения запроса;

• ENOSPC — на устройстве, где находится path, недостаточно свободного пространства либо для данного пользователя превышена квота места на диске;
• ENOTDIR — один или несколько компонентов пути path не являются каталогом;
• EPERM — файловая система, в которой находится path, не поддерживает создание каталогов;
• EROFS — файловая система, в которой находится path, монтирована с доступом только для чтения.

4.5 Удаление каталогов

В противоположность mkdir() регламентированный POSIX вызов rmdir() удаляет каталог из иерархии файловой системы:

#include <unistd.h>

int rmdir (const char *path);

В случае успеха rmdir() удаляет path из файловой системы и возвращает 0. Каталог, указанный через path, должен быть пустым, за исключением каталогов точка и точка-точка. Не существует системного вызова, реализующего аналог рекурсивного удаления, наподобие rm -r. Можно выполнить подобную операцию вручную, продвигаясь в глубь файловой системы и удаляя все файлы и папки, начиная с листьев и переходя к корню файловой системы; на каждой стадии может быть использован rmdir() для удаления каталога сразу после того, как были удалены все находившиеся в нем файлы.

В случае ошибки rmdir() возвращает -1 и присваивает errno одно из следующих значений:

• EACCES — текущий процесс не имеет прав записи в родительском каталоге или прав поиска в одном из каталогов-компонентов пути path;
• EBUSY — path в настоящее время используется системой и не может быть удален; в Linux это может случиться, только если path является точкой сборки для корневого каталога (хотя корневые каталоги не должны быть точками сборки благодаря chroot()!);
• EFAULT — указатель path является недопустимым;
• EINVAL — финальным компонентом пути path является каталог точка;

• ELOOP — ядро обнаружило в пути path слишком много символических ссылок; O ENAMETOOLONG — путь path слишком велик;
• ENOENT — какой-либо компонент пути path не существует или является символической ссылкой, ведущей к несуществующему объекту;
• ENOMEM — недостаточно памяти ядра для выполнения запроса;
• ENOTDIR — один или несколько компонентов пути path не являются каталогом; O ENOTEMPTY — каталог, указанный через path, содержит иные элементы, чем каталоги точка или точка-точка;
• EPERM — для каталога, являющегося предком каталога path, установлен бит закрепления в памяти, но действительный идентификатор пользователя процесса не совпадает ни с идентификатором пользователя родительского процесса, ни с идентификатором каталога path, а процесс не имеет свойства CAP_FOWNER; или файловая система, где находится path, не допускает удаления каталогов;
• EROFS — файловая система, в которой находится path, монтирована с доступом только для чтения.

Использование очень просто:

```
int ret;
/* удаление каталога /home/barbary/maps */
ret = rmdir ("/home/barbary/maps");
if (ret)
        perror ("rmdir");
```

4.6 Чтение содержимого каталога

POSIX определяет семейство функций для чтения содержимого каталогов, то есть для получения списка файлов, которые относятся к данной директории. Эти функции полезны, если вы реализуете ls или графический интерфейс диалога сохранения, выполнение операций со всеми файлами каталога или поиск среди файлов каталога тех, которые соответствуют определенному шаблону.

Чтобы начать чтение содержимого, вы должны создать поток каталога, который представлен объектом DIR:

#include <sys/types.h>
#include <dirent.h>
DIR * opendir (const char *name);

Успешный вызов opendir() создает поток каталога, представляющий каталог, определенный через name.

Поток каталога является, по сути, просто файловым дескриптором, представляющим открытый каталог, некоторые метаданные и буфер для сохранения содержимого. Следовательно, возможно получить файловый дескриптор для соответствующего каталога внутри данного потока каталога:

#define _BSD_SOURCE /* или_SVID_SOURCE */
#include <sys/types.h>
#include <dirent.h>
int dirfd (DIR *dir);

Успешный вызов dirfd() возвращает файловый дескриптор, соответствующий потоку каталога dir. В случае ошибки вызов возвращает -1. Поскольку функция потока каталога использует данный дескриптор при выполнении своей задачи, программы не должны обращаться к вызовам,изменяющим положение файлов. Dirfd() является расширением BSD и не стандартизирован POSIX; программисты, стремящиеся к полному соответствию POSIX, должны избегать его.

4.7 Чтение из потока каталога

После того как с помощью вызова opendir() создан поток каталога, программа может читать находящиеся в нем записи. Чтобы сделать это, используйте readdir(), который возвращает записи одну за другой из указанного объекта DIR:

#include <sys/types.h>
#include <dirent.h>
struct dirent * readdir (DIR *dir);

Успешный вызов readdir() возвращает следующую запись в каталоге, представленную через dir. Структура dirent представляет запись в каталоге. Она определена в заголовочном файле <dirent.h> в Linux следующим образом:

struct dirent {

```
        ino_t d_ino;                    /* номер inode */
        off_t d_off;                    /* переход к следующей записи
dirent */
        unsigned short d_reclen;        /* длина данной записи */
        unsigned char d_type;           /* тип файла */
        char d_name[256];               /* имя файла*/
};
```

POSIX требует только наличия поля d_name, которое представляет собой имя файла в данном каталоге. Прочие поля необязательные или являются уникальными для Linux. Приложения, требующие совместимости с другими системами или строгого соответствия POSIX, должны иметь доступ только к d_name.

Приложения последовательно вызывают readdir(), получая файлы из каталога один за другим, пока записи в каталоге не закончатся либо обнаружение новых файлов не прекратится. В первом случае readdir() возвращает значение NULL.

В случае ошибки readdir() также возвращает NULL. Чтобы отличить ошибку от окончания списка файлов, перед каждым вызовом readdir() следует присвоить переменной errno значение 0, а затем проверять значения и возвращаемой величины, и errno. Единственное значение errno, которое может установить readdir(), — это EBADF, указывающее, что значение dir недопустимо. Однако многие приложения не обрабатывают ошибки, и в них предполагается, что возвращение NULL означает лишь окончание файлов в каталоге.

Закрытие потока каталога

Чтобы закрыть поток каталога, открытый с помощью opendir(), используйте closedir():

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dir);
```

Успешный вызов closedir() закрывает поток каталога, обозначенного с помощью dir, включая соответствующий файловый дескриптор, и возвращает 0. В случае ошибки функция возвращает -1 и устанавливает errno в значение EBADF, единственный возможный код ошибки, означающий, что dir не является открытым потоком каталога.

Следующий фрагмент кода представляет функцию find_file_in_dir(), которая использует readdir() для поиска данного имени файла в указанном каталоге. Если файл действительно находится в каталоге, функция возвращает 0. В ином случае она возвращает ненулевую величину:

```
/*
   • find_file_in_dir - ищет в каталоге 'path' файл
   • под названием 'file'.
*
   • Возвращает 0, если 'file' существует в 'path', и ненулевую
   • величину во всех иных случаях */
int find_file_in_dir (const char *path, const char *file)
{
        struct dirent *entry; int ret = 1;
        DIR *dir;
        dir = opendir (path); errno = 0;
        while ((entry = readdir (dir)) != NULL) {
if (strcmp(entry->d_name, file) == 0) {
        ret = 0;
        break;
}
}
if (errno && !entry)
        perror ("readdir");
closedir (dir);
return ret;
}
```

4.8 Системные вызовы для чтения содержимого каталога

Рассмотренные выше функции для чтения содержимого каталога регламентированы POSIX и предоставляются библиотекой C. Внутри системы эти функции
используют один из двух системных вызовов — readdir() или getdents(), которые мы рассмотрим для полноты обсуждения:

```
#include <unistd.h>
#include <linux/types.h>
```

```
#include <linux/dirent.h>
#include <linux/unistd.h>
#include <errno.h>
/*
        • Не определены в пользовательском пространстве:
        • для доступа необходим макрос _syscall3(). */
*/
int readdir (unsigned int fd,
struct dirent *dirp, unsigned int count);
int getdents (unsigned int fd,
struct dirent *dirp, unsigned int count);
```

Вам не нужно использовать эти системные вызовы. Они сложны в применении и ухудшают совместимость. Вместо этого приложения в пользовательском пространстве должны использовать системные вызовы opendir(), readdir() и closedir(), предоставляемые библиотекой C.

4.9 Ссылки

Исходя из этого простого определения ссылка представляет собой всего лишь имя в списке (каталоге), которое указывает на inode, и, таким образом, может существовать несколько ссылок на один и тот же inode. Действительно, на один inode может ссылаться, скажем, как /etc/customs, так и /var/run/ledger.

Здесь и в самом деле кроется небольшой подвох. Ссылки привязаны к номерам inode, а номера inode уникальны в каждой файловой системе, следовательно, оба файла, /etc/customs и /var/run/ledger, должны принадлежать к одной и той же файловой системе. В пределах одной файловой системы к одному и тому же файлу может относиться много ссылок. Единственное ограничение касается размера целочисленного типа данных, используемого для хранения ссылок. Среди всех различных ссылок ни одна не может считаться «основной» или «оригинальной»: статус всех ссылок одинаков и все они указывают на один файл.

Такие типы ссылок мы называем жесткими ссылками. Файлы могут не иметь ни одной, иметь одну или много ссылок. Большинство

файлов имеют количество ссылок, равное 1. Это значит, что на них ссылается единственная запись в каталоге, но некоторые файлы имеют две ссылки или даже больше. Файлы, количество ссылок на которые равно 0, не имеют соответствующих записей каталога в системе.

Когда количество ссылок на файл равно 0, файл помечается как свободный, а его дисковые блоки становятся свободными для использования1. Такой файл, однако, остается в файловой системе, если он открыт у какого-либо процесса. После того как все процессы закрыли этот файл, он удаляется.

Ядро Linux реализует этот подход, используя счетчик ссылок и счетчик использований. Счетчик использований — это общее количество экземляров, где открыт данный файл. Файл не удаляется из файловой системы, пока количество как ссылок, так и использований не станет равным 0.

Другой тип ссылки, символическая ссылка, является не маршрутизатором файловой системы, а более высокоуровневым указателем, который интерпретируется во время выполнения. Такие ссылки могут охватывать разные файловые системы — мы поговорим о них позже.

4.10 Жесткие ссылки

Системный вызов link(), один из первоначальных системных вызовов в UNIX, а сейчас стандартизированный и в POSIX, создает новую ссылку на существующий файл:

#include <unistd.h>

int link (const char *oldpath, const char *newpath);

Успешный вызов link() создает новую ссылку по пути newpath для существующего файла oldpath, а затем возвращает 0. По выполнении и oldpath, и newpath ссылаются на один и тот же файл — отныне фактически нельзя сказать, который из этих путей является «исходным».

В случае сбоя вызов возвращает -1 и присваивает errno одно из следующих значений:

• EACCES — вызывающий процесс не имеет прав на поиск в одном из каталогов- компонентов oldpath или не имеет прав записи в одном из каталогов, составляющих newpath;
• EEXIST — newpath уже существует — link() не может переписать существующую запись каталога;
• EFAULT — указатель oldpath или newpath недопустим;
• EIO — произошла внутренняя ошибка ввода-вывода (это очень плохо!);
• ELOOP — при разрешении пути oldpath или newpath было обнаружено слишком много символических ссылок;
• EMLINK — inode, на который указывает oldpath, уже достиг максимально допустимого количества ссылок на себя;
• ENAMETOOLONG — путь oldpath или newpath слишком велик;
• ENOENT — один из компонентов oldpath или newpath не существует;
• ENOMEM — недостаточно памяти для выполнения запроса;
• ENOSPC — на устройстве, где находится newpath, недостаточно памяти для новой записи каталога;
• ENOTDIR — один из компонентов oldpath или newpath не является каталогом;
• EPERM — файловая система, где находится newpath, не позволяет создание новых жестких ссылок или oldpath является каталогом;
• EROFS — newpath находится в файловой системе, доступной только для чтения;
• EXDEV — oldpath и newpath не принадлежат к одной и той же монтированной файловой системе (Linux позволяет монтирование одной и той же файловой системы в нескольких точках, но даже в этом случае жесткие ссылки не могут создаваться между точками монтирования).

Следующий пример создает новую запись в каталоге — pirate, которая ссылается на тот же самый inode (и, следовательно, на тот же файл), что и существующий файл privateer; оба находятся в /home/kidd:

```
int ret;
/*
    • создание новой записи каталога,
```

```
    • '/home/kidd/privateer', которая указывает на
    • тот же самый inode, что и '/home/kidd/pirate'
*/
ret = link ("/home/kidd/privateer", /home/kidd/pirate"); if (ret)
perror ("link");
```

4.11 Символические ссылки

Символические ссылки, еще известные как симссылки и мягкие ссылки, сходны с жесткими ссылками тем, что они также указывают на файлы в файловой системе. Однако символические ссылки представляют собой не просто дополнительную запись каталога, но и специальный тип файла. Этот специальный файл содержит путь к другому файлу, называемому целью символической ссылки. Во время выполнения ядро на лету заменяет путь к файлу символической ссылки путем к цели символической ссылки (кроме случая, когда в программе используются различные l -версии системных вызовов, например l-stat, которые работают с самой ссылкой, а не с ее целью). Таким образом, жесткие ссылки на один и тот же файл неотличимы друг от друга, а различия между символической ссылкой и ее целевым файлом очевидны.

Символическая ссылка может быть относительной или абсолютной. Она может также содержать специальный каталог точка, рассмотренный выше, относящийся к каталогу, в котором он расположен, или каталог точка-точка, относящийся к каталогу верхнего уровня. Такие виды относительных символических ссылок широко применяются и часто весьма полезны.

Мягкие ссылки, в отличие от жестких, могут охватывать целые файловые системы. На самом деле они могут указывать куда угодно! Символические ссылки могут указывать на файл, который существует (обычное явление) или не существует. Последний тип ссылки называется повисшей символической ссылкой. Иногда повисшие ссылки нежелательны — например, когда цель этой ссылки была удалена, но осталась сама ссылка, но временами они необходимы. Символическая ссылка даже может указывать на другую символическую ссылку. Таким образом можно создавать петли. Системные вызовы, имеющие дело с символическими ссылками,

проверяют петли до максимально достижимой глубины. Если эта глубина превышена, они возвращают ELOOP.

Системный вызов для создания символических ссылок очень похож на своего двоюродного брата, предназначенного для жестких:

#include <unistd.h>

int symlink (const char *oldpath, const char *newpath);

Успешный вызов symlink() создает символическую ссылку newpath, указывающую на цель oldpath, а затем возвращает 0.

В случае ошибки syml ink() возвращает -1, а затем присваивает errno одно из следующих значений:

- EACCES — вызывающий процесс не имеет прав на поиск в одном из каталогов- компонентов oldpath или не имеет прав записи в одном из каталогов, составляющих newpath;
- EEXIST — newpath уже существует — symlink() не может переписать существующую запись каталога;
- EFAULT — указатель oldpath или newpath недействителен;
- EIO — произошла внутренняя ошибка ввода-вывода (это очень плохо!);
- ELOOP — при разрешении пути oldpath или newpath было обнаружено слишком много символических ссылок;
- EMLINK — inode, на который указывает oldpath, уже достиг максимально допустимого количества ссылок на себя;
- ENAMETOOLONG — путь oldpath или newpath слишком велик;
- ENOENT — один из компонентов oldpath или newpath не существует;
- ENOMEM — недостаточно памяти для выполнения запроса;
- ENOSPC — на устройстве, где находится newpath, недостаточно памяти для новой записи каталога;
- ENOTDIR — один из компонентов oldpath или newpath не является каталогом;
- EPERM — файловая система, где находится newpath, не позволяет создание новых символических ссылок;
- EROFS — newpath находится в файловой системе, доступной только для чтения.

Вот этот фрагмент кода — то же самое, что предыдущий пример, но он создает /home/kidd/pirate как символическую ссылку (в отличие от жесткой) на /home/kidd/ privateer:

```
int ret;
/*
   • создание символической ссылки,
   • '/home/kidd/privateer', которая
   • указывает на '/home/kidd/pirate'
*/
ret = symlink ("/home/kidd/privateer", "/home/kidd/pirate"); if (ret)
perror ("symlink");
```

4.12 Удаление ссылки

Ссылку можно как создать, так и удалить, разорвав путь в файловой системе. Эту задачу может выполнить единственный вызов, unlink():

```
#include <unistd.h>
int unlink (const char *pathname);
```

Успешный вызов unlink() удаляет pathname из файловой системы и возвращает 0. Если этот путь был последней ссылкой на файл, файл удаляется из системы. Если, однако, файл открыт в каком-либо процессе, ядро не будет удалять файл из файловой системы, пока процесс не закроет этот файл. Если же этот файл не открыт ни в одном процессе, он удаляется.

Если pathname ведет к символической ссылке, удаляется сама ссылка, а не ее цель. Если pathname ссылается на другой тип специального файла, например устройство, конвейер FIFO или сокет, этот файл удаляется из файловой системы, но процессы, у которых он открыт, могут продолжать его использовать.

В случае ошибки unlink() возвращает -1 и присваивает errno один из следующих кодов ошибки:

- EACCES — вызывающий процесс не имеет прав на поиск в одном из каталогов- компонентов pathname или не имеет прав записи в родительском каталоге pathname;
- EFAULT — указатель pathname имеет недопустимое значение;
- EIO — произошла внутренняя ошибка ввода-вывода (это очень плохо!);
- EISDIR — pathname ссылается на каталог;

• ELOOP — при прохождении pathname было обнаружено слишком много символических ссылок;
• ENAMETOOLONG — путь pathname слишком велик;
• ENOENT — один из компонентов pathname не существует;
• ENOMEM — недостаточно памяти для выполнения запроса;
• ENOTDIR — один из компонентов pathname не является каталогом;
• EPERM — файловая система не позволяет удаление ссылок;
• EROFS — pathname находится в файловой системе, доступной только для чтения.
• Вызов unl ink () не удаляет каталоги. Для этого приложения должны использовать rmdir(), который мы обсуждали ранее (см. разд. «Удаление каталогов»).
• Чтобы упростить удаление файлов любого типа, язык С предоставляет функцию remove():
• #include <stdio.h>
• int remove (const char *path);
• Успешный вызов remove() удаляет path из файловой системы и возвращает 0. Если path — это файл, remove() вызывает unlink(); если же это каталог, remove() вызывает rmdir() .
В случае ошибки remove() вызвращает -1 и присваивает errno значение, соответствующее одному из возвращаемых unlink() и rmdir().

4.13 Копирование и перемещение файлов

Две самые простые и распространенные операции с файлами — это их копирование и перемещение, обычно выполняемые с помощью команд ср и mv. На уровне файловой системы копирование означает дублирование содержимого файла в другое место и создание нового пути к файлу. Это отлично от создания новой жесткой ссылки на файл, так как изменение одного файла не влияет на другой: теперь существуют две различные копии файла, которым соответствуют две (по меньшей мере) разные записи каталога. Перемещение, напротив, является актом переименования записи каталога, под которой размещается файл. Это действие не приводит к созданию новой копии.

В Linux нет системного или библиотечного вызова для выполнения копирования файлов или каталогов. Вместо этого утилиты наподобие cp или файловые менеджеры GNOME выполняют эти операции вручную.

Для копирования файла src в файл под названием dst необходимо выполнить следующие действия.

1. Открыть файл src.
2. Открыть файл dst — создать его, если он не существует, или сократить до нулевой длины, если он есть.
3. Считать фрагмент содержимого src в память.
4. Записать этот фрагмент в dst.
5. Продолжать, пока все содержимое src не будет считано и переписано в dst.
6. Закрыть src.
7. Закрыть dst.

При копировании каталога отдельный каталог и его подкаталоги создаются с помощью mkdir(); каждый файл внутри них затем копируется индивидуально.

В отличие от копирования файлов, UNIX предоставляет системный вызов для перемещения. Стандарт ANSI C представляет вызов для файлов, а POSIX регламентирует вызов и для файлов, и для каталогов:

#include <stdio.h>

int rename (const char *oldpath, const char *newpath);

Успешный вызов rename() переименовывает путь oldpath в newpath. Содержимое файла и inode остаются теми же самыми. И oldpath, и newpath должны остаться в той же файловой системе1; если это не так, вызов вернет ошибку. Утилиты наподобие mv обрабатывают этот случай копированием файла и удалением ссылки.

При успехе rename() возвращает 0, и путь к файлу oldpath меняется на newpath. В случае неудачи вызов возвращает -1, никак не влияя ни на oldpath, ни на newpath, и присваивает errno одно из следующих значений:

• EACCES — вызывающий процесс не имеет прав записи в родительских каталогах oldpath или newpath, прав поиска в одном из компонентов oldpath или newpath или прав записи для oldpath, если это каталог; последний случай может представлять

проблему, так как rename() должен обновить .. в oldpath, если это каталог;

- EBUSY — oldpath или newpath являются точкой монтирования;
- EFAULT — указатель oldpath или newpath является недопустимым;
- EINVAL — newpath находится внутри oldpath; таким образом, переименование одного в другой сделает oldpath подкаталогом самого себя;
- EISDIR — newpath существует и является каталогом, а oldpath не является;
- ELOOP — при разрешении пути oldpath или newpath было обнаружено слишком много символических ссылок;
- EMLINK — oldpath уже достиг максимально допустимого количества ссылок на себя или oldpath является каталогом, а newpath уже достиг максимально допустимого количества ссылок на себя;
- ENAMETOOLONG — пути oldpath или newpath слишком велики;
- ENOENT — один из компонентов oldpath или newpath не существует или является повисшей символической ссылкой;
- ENOMEM — недостаточно памяти ядра для выполнения запроса;
- ENOSPC — на устройстве, где находится newpath, недостаточно пространства для выполнения запроса;
- ENOTDIR — один из компонентов (за исключением потенциально последнего компонента) oldpath или newpath не является каталогом или oldpath — это каталог, а newpath уже существует и не является каталогом;
- ENOTEMPTY — newpath является каталогом и непустой;
- EPERM — по крайней мере один из указанных путей существует, для родительского каталога установлен бит закрепления в памяти, действительный идентификатор пользователя вызывающего процесса не совпадает ни с идентификатором пользователя файла, ни с идентификатором пользователя родительского процесса, а процесс не имеет привилегированных прав;
- EROFS — файловая система доступна только для чтения;

• EXDEV — oldpath и newpath находятся в разных файловых системах.

В таблице 4.1 приведены результаты перемещения различных типов файлов.

Таблица 4.1. Результаты перемещения различных типов файлов

Исходный объек	**Цель — файл**	**Цель — каталог**	**Цель — ссылка**	**Цель не существует**
Файлом	Целевой файл заменяется на исходный	Ошибка EISDIR	Файл переименовывается, цель переписывается	Файл переименовывается
Каталогом	Ошибка ENOTDIR	Если целевой каталог пуст, то исходный файл переименовывается названием целевого файла; в противном случае возвращается ошибка ENOTEMPTY	Каталог переименовывается, цель переписывается	Каталог переименовывается
Ссылкой	Ссылка переименовывается, цель переписываетс я	Ошибка EISDIR	Ссылка переименовывается, цель переписывается	Ссылка переименовывается
Не существует	Ошибка ENOENT	Ошибка ENOENT	Ошибка ENOENT	Ошибка ENOENT

Во всех случаях, независимо от типов исходного и целевого объектов, если они находятся в разных файловых системах, вызов вернет ошибку EXDEV.

4.14 Задание для самостоятельной работы

Разработать программу на языке программирования C, которая осуществляет работу с каталогами и ссылками в файловой системе в соответсвии с индивидуальным вариантом. Программа должна собираться с помощью утилиты сборки make и компилятора gcc, а так же работать в операционных системах семейства Linux.

Индивидуальный вариант, а также тип ссылок задает преподаватель. Так же предусмотреть в программе обработку ошибок системных вызовов.

Контрольные вопросы

1) Что такое каталог с точки зрения файловой системы?
2) Какой длинны может быть имя каталога в современной UNIX подобной операционной системе?
3) Что такое текущий каталог?
4) С помощью какого системного вызова можно получить текущий каталог процесса?
5) Какие ошибки может возвращать системный вызов getcwd?
6) Для чего предназначен системный вызов chdir?
7) Какие ошибки может возвращать системный вызов chdir?
8) С помощью какого системного вызова можно создать каталог?
9) Какие ошибки может возвращать системный вызов mkdir?
10) Для чего предназначен системный вызов rmdir?
11) Какие ошибки может возвращать системный возов rmdir?
12) Как осуществляется чтение каталога?
13) С помощью какого вызова создается поток каталога?
14) Как осуществляется чтение из потока каталога?
15) Для чего предназначена структура dirent?
16) С помощью какого системного вызова можно закрыть каталог?
17) Как можно произвести чтение содержимого каталога?
18) Что такое ссылки?
19) Какие виды ссылок существуют?
20) Для чего предназначен системный вызов link?
21) Какие ошибки может возвращать системный вызов link?
22) Для чего предназначен системный вызов symlink?
23) Какие ошибки может возвращать системный вызов symlink?
24) С помощью какого системного вызова можно удалить ссылку?
25) Какие ошибки может возвращаться системный вызов unlink?
26) Как осуществляется копирование и перемещение файлов?
27) Какие ошибки может возвращать системный вызов rename?

Тема 5. Основные принципы разработки драйверов устройств.

Одно из главных преимуществ свободных операционных систем, таких как Linux, это то, что их внутренности открыты для просмотра всем. Эти операционные системы, когда-то тёмная и мистическая область, чей код был доступен только небольшому числу программистов, могут быть теперь легко изучены, поняты и модифицированы кем угодно, если он обладает необходимым уровнем знаний. Linux помог демократизировать операционные системы. Ядро Linux, тем не менее, представляет собой большой и сложный набор кода, и потенциальные исследователи ядра нуждаются в точке входа, где они могут обращаться к этому коду не будучи подавленными его сложностью. Часто такую точку входа создают драйверы устройств.

Драйверам устройств отводится особая роль в ядре Linux. Это прекрасные “чёрные ящики”, которые заставляют специфичную часть оборудования соответствовать строго заданному программному интерфейсу; они полностью скрывают детали того, как работает устройство. Действия пользователя сводятся к выполнению стандартизированных вызовов, которые не зависят от специфики драйвера; перевод этих вызовов в специфичные для данного устройства операции, которые исполняются реальным оборудованием, является задачей драйвера устройства. Этот программный интерфейс таков, что драйверы могут быть собраны отдельно от остальной части ядра и подключены в процессе работы, когда это необходимо. Такая модульность делает драйверы Linux простыми для написания, так что теперь доступны сотни драйверов.

5.1 Роль драйвера устройства

Различие между механизмом и политикой - одна из лучших идей, стоящих за проектом Unix. Большинство проблем программирования в действительности может быть разделено на две части: "какие возможности будут обеспечены" (это механизм) и "как эти возможности могут быть использованы" (это политика, правила). Если две проблемы адресованы разным частям программы, или даже

разным программам в целом, программный пакет много легче разработать и адаптировать к специфическим требованиям.

Например, управление в Unix графическим дисплеем разделено между X-сервером, который знает оборудование и предлагает унифицированный интерфейс для пользовательских программ, менеджерами окна и сессий, которые осуществляют индивидуальную политику, не зная что-либо об оборудовании. Люди могут использовать тот же оконный менеджер на разном оборудовании и разные пользователи могут запускать разные конфигурации на той же рабочей станции. Даже полностью различные настольные среды, такие как KDE и GNOME, могут сосуществовать в одной системе. Другим примером является многоуровневая сетевая структура TCP/IP: эта операционная система предлагает абстракцию сокета, которая не осуществляет политики в отношении передаваемых данных, в то время как разные серверы управляют сервисами (и связанными с ними политиками). Более того, сервера, наподобие ftpd, обеспечивают механизм передачи файлов, а пользователи могут использовать любого клиента, которого пожелают; существуют и клиенты, управляемые через командную строчку и через графический интерфейс, и кто угодно может написать новый пользовательский интерфейс для передачи файлов.

Применительно к драйверам используется то же самое разделение механизма и политики. Драйвер дисковода свободен от правил - его задача только показать дискету как непрерывный массив блоков данных. Более высокие уровни системы обеспечивают правила, такие как, кто может иметь доступ к дисководу, можно ли обращаться к нему напрямую или только через файловую систему, могут ли пользователи монтировать файловую систему дисковода. Так как различное окружение обычно нуждается в использовании оборудования разными способами, важно быть по возможности свободными от правил.

При написании драйверов программист должен уделить внимание фундаментальной концепции: написать код ядра для доступа к оборудованию, но не оказать давление частными правилами на пользователя, так как разные пользователи имеют разные потребности. Ваш драйвер должен обеспечивать доступ к

оборудованию, оставляя задачи *как* использовать оборудование приложениям. Таким образом, драйвер гибок, если обеспечивает доступ к оборудованию без ограничений. Иногда, тем не менее, некоторые ограничения должны иметь место. К примеру, драйвер ввода/вывода может предоставлять только побайтный доступ к оборудованию, чтобы избежать написания дополнительного кода, необходимого для передачи отдельных битов.

Можно также рассматривать драйвер в другой перспективе: это программный слой, который находится между приложениями и реальным устройством. Эта привилегированная роль драйвера позволяет программисту драйвера точно выбрать, как устройство должно быть представлено: разные драйверы могут предлагать разные возможности даже для одного и того же устройства. Фактически, драйвер должен быть разработан так, чтобы обеспечивать баланс между разными соображениями. Например, за использование одного устройства могут конкурировать одновременно несколько программ и автор драйвера имеет полную свободу решать, как обслуживать конкурирующие запросы.

"Гибкие" драйверы имеют типичные характеристики. Они включают поддержку синхронных и асинхронных операций, возможность быть открытыми множество раз, возможность максимально полного использования оборудования и отсутствие лишних программных уровней, чтобы оставаться простыми и свободными от ограничивающих операций. Драйверы такого сорта не только работают лучше у конечных пользователей, но также оказываются проще в написании и сопровождении. Быть свободными от ограничений - общая цель для разработчиков программного обеспечения.

Более того, многие драйверы устройств поставляются вместе с пользовательскими программами, чтобы помочь с конфигурированием и доступом к целевому устройству. Такие программы могут быть и простыми утилитами, и графическими приложениями. В качестве примера можно привести программу *tunelp*, которая регулирует работу драйвера параллельного порта, и графическую утилиту cardctl, входящую в состав пакета PCMCIA драйвера. Часто предоставляются также клиентские библиотеки,

которые обеспечивают возможности, которые нет необходимости иметь как часть самого драйвера.

5.2 Строение ядра Linux

В системе Unix несколько параллельных *процессов* обслуживают разные задачи. Каждый процесс запрашивает системные ресурсы, будь то энергия, память, сетевое подключение, или какие-то другие ресурсы. *Ядро* - это большой кусок исполняемого кода, отвечающего за обработку всех таких запросов. Хотя границы между разными задачами ядра не всегда ясно определены, роль ядра может быть разделена (как показано на рисунке 5.1) на следующие части:

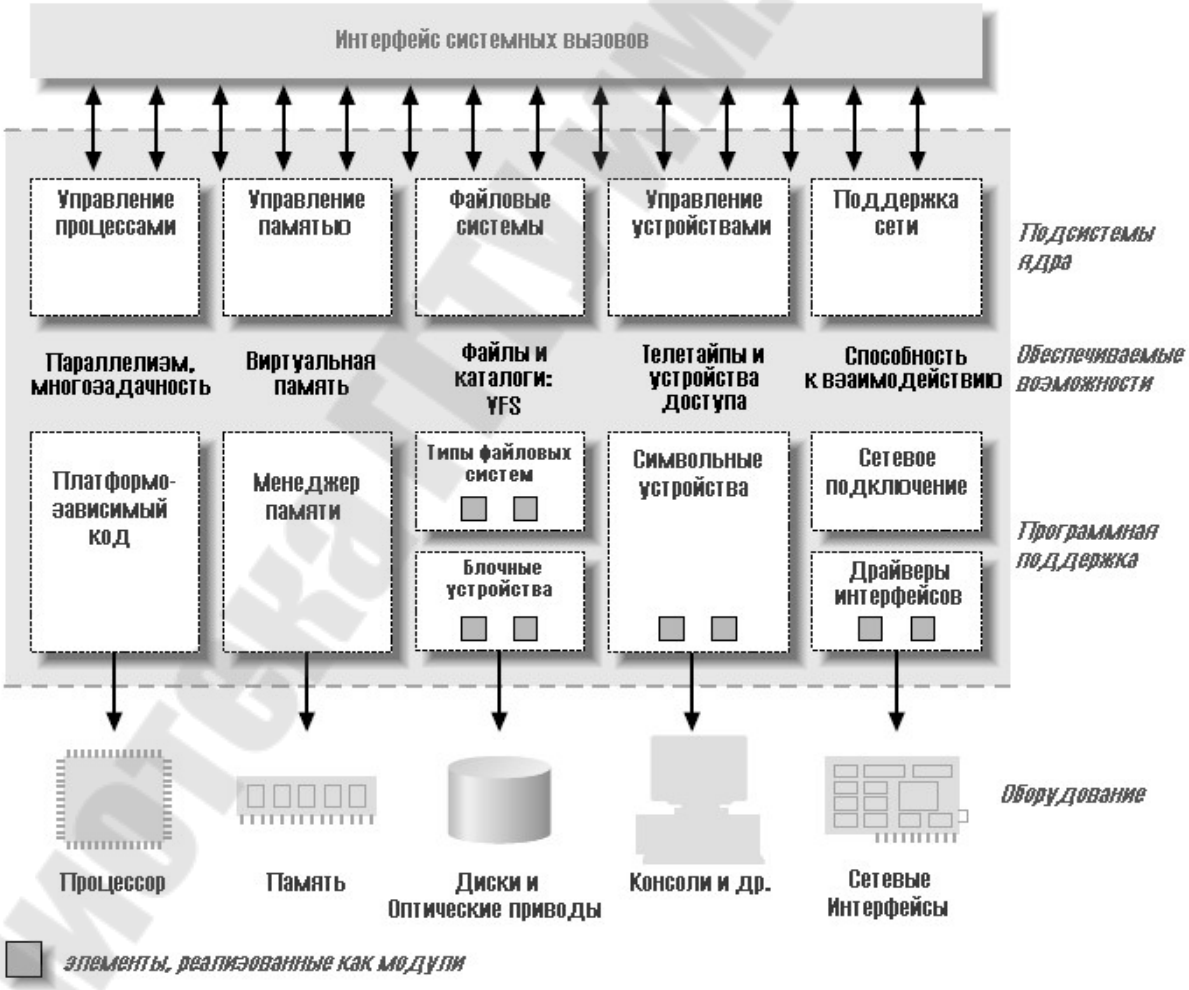

Рисунок 5.1 – Строение ядра Linux

- Управление процессами

Ядро отвечает за создание и уничтожение процессов и обеспечение их взаимодействия с внешним миром (ввод и вывод). Взаимодействие между разными процессами (через сигналы, каналы или примитивов межпроцессных взаимодействий) является основой общей функциональности системы и также возложена на ядро. Дополнительно, планировщик, который распределяет время процессора, тоже является частью системы управления процессами. В общих словах, деятельность процессов управления ядра создаёт абстракцию нескольких процессов поверх одного или нескольких процессоров.

- Управление памятью

Память компьютера - главный ресурс и способ управления ей особенно важен для производительности системы. Ядро создаёт виртуальное адресное пространство для каждого процесса поверх имеющихся ограниченных ресурсов. Разные части ядра взаимодействуют с подсистемой управления памятью через набор функциональных вызовов, начиная от простой пары malloc/free до много более развитой функциональности.

- Файловые системы

Unix очень сильно связана с концепцией файловой системы; почти всё в Unix может быть обработано как файл. Ядро строит структурированную файловую систему поверх неструктурированного оборудования, и полученная файловая абстракция интенсивно используется всей системой. В дополнение Linux поддерживает множество типов файловых систем, то есть различные способы организации данных на физическом носителе. К примеру, диски могут быть отформатированы в стандартной для Linux файловой системе ext3, часто используемой файловой системе FAT или некоторых других.

- Управление устройствами

Почти каждая системная операция в конечном счёте связывается с физическим устройством. За исключением процессора, памяти и очень немногих других объектов, каждая операция управления устройством выполняются кодом, специфичным для данного адресуемого устройства. Этот код называется драйвером устройства. Ядро должно иметь встроенный драйвер устройства для

каждой периферии, существующей в системе, от жёсткого диска до клавиатуры и ленточного накопителя. Этот аспект функциональности ядра и является нашим основным интересом в этой книги.

- Сетевое подключение

Сетевое подключение должно управляться операционной системой, потому что большинство сетевых операций не зависят от процессов: входящие пакеты - это асинхронные события. Эти пакеты должны быть собраны, распознаны и распределены перед тем, как они будут переданы другому процессу для обработки. Система отвечает за доставку пакетов данных между программой и сетевыми интерфейсами и должна управлять исполнением программ в зависимости от их сетевой активности. Дополнительно в ядро встроены все задачи маршрутизации и разрешение адресов.

Одной из хороших особенностей Linux является способность расширения функциональности ядра во время работы. Это означает, что вы можете добавить функциональность в ядро (и убрать её), когда система запущена и работает.

Часть кода, которая может быть добавлена в ядро во время работы, называется модулем. Ядро Linux предлагает поддержку довольно большого числа типов (или классов) модулей, включая, но не ограничиваясь, драйверами устройств. Каждый модуль является подготовленным объектным кодом (не слинкованным для самостоятельной работы), который может быть динамически подключен в работающее ядро программой insmod и отключен программой rmmod.

Рисунок 5.1 определяет разные классы модулей, отвечающих за специфические задачи как говорят, модуль принадлежит к определённому классу в зависимости от предлагаемой функциональности. Картина из модулей, показанная на рисунке 5.1, охватывает все наиболее важные классы, но далеко не полная, потому что много большая функциональность Linux модулизирована.

5.3 Классы устройств и модулей

Способ видения устройств в Linux разделяется на три фундаментальных типа. Каждый модуль обычно реализован как один из этих типов и таким образом классифицируется как символьный

модуль, блочный модуль, или сетевой модуль. Такое разделение модулей на разные типы или классы не является жёстким; программист может при желании создавать большие модули, содержащие разные драйверы в одном куске кода. Хорошие программисты, тем не менее, обычно создают разные модули для каждой новой функциональности, потому что декомпозиция является ключом к масштабируемости и расширяемости.

Этими тремя классами являются:

- Символьные устройства

Символьное устройство - это такое устройств, к которому можно обращаться как к потоку байтов (так же как к файлу); драйвер символьного устройства отвечает за реализацию такого поведения. Такой драйвер обычно, по крайней мере, поддерживает системные вызовы open, close, read и write. Текстовый экран (/dev/console) и последовательные порты (/dev/ttyS0 и подобные) являются примерами символьных устройств, так как они хорошо представлены абстракцией потока. Для обращения к символьным устройствам используют узлы (node) файловой системы, такие как /dev/tty1 и /dev/lp0. Единственное важное отличие между символьными устройствами и обычными файлами - вы всегда можете двигаться вперед и назад в обычном файле, в то время как большинство символьных устройств - это только каналы данных, к которым вы можете обращаться только последовательно. Существуют, однако, символьные устройства, которые выглядят как области данных, и вы можете двигаться по ним назад и вперёд; к примеру, это обычно используется в грабберах экрана, где приложения могут получать доступ ко всему полученному изображению используя mmap или lseek.

- Блочные устройства

Так же как символьные устройства, блочные устройства доступны через узлы файловой системы в директории /dev. Блочное устройство - это устройство (например, диск) который может содержать файловую систему. В большинстве систем Unix блочное устройство может поддерживать только операции ввода-вывода, которые передают один или более целых блоков, обычно равных 512 байт (или большей степени числа два). Linux, однако, разрешает приложению читать и писать в блочное устройство, так же как и в

символьное устройство - это позволяет передавать любое число байт за раз. В результате, блочные и символьные устройства отличаются только способом управления данными внутри ядра и, соответственно, программным интерфейсом в ядре/драйвере. Как и символьное устройство, каждое блочное устройство доступно через узел файловой системы, так что различия между ними не видны пользователю. Блочные драйверы имеют интерфейс для ядра полностью отличный от символьных устройств.

- Сетевые интерфейсы

Любой сетевой обмен данными делается через интерфейс, то есть устройство, которое в состоянии обменяться данными с другими узлами сети. Обычно, интерфейс - это аппаратное устройство, но также он может быть чисто программным устройством, наподобие интерфейса loopback (локальное петлевое устройство). Сетевой интерфейс отвечает за отсылку и приём пакетов данных, управляемых подсистемой сети в ядре, без знания кому предназначены передаваемые пакеты.

Многие сетевые соединения (особенно использующие TCP) являются поточноориентированными, но сетевые устройства обычно разработаны для передачи и приёма пакетов. Сетевой драйвер ничего не знает об отдельных соединениях; он только обрабатывает пакеты. Не будучи поточно-ориентированным устройством, сетевой интерфейс нелегко представить как узел в файловой системе наподобие /dev/tty1. Unix всё же обеспечивает доступ к интерфейсам через назначение им уникальных имён (таких как eth0), но это имя не имеет соответствующего элемента в файловой системе. Обмен между ядром и сетевым устройством сильно отличается от используемого в символьных и блочных драйверах. Вместо read и write ядро вызывает функции, относящиеся к передаче пакетов.

Есть другие пути классификации модулей драйверов, которые по-другому подразделяют устройства. Вообще, некоторые типы драйверов работают с дополнительными наборами функций ядра для данного типа устройств. К примеру, можно говорить о модулях универсальной последовательной шины (USB), последовательных модулях, модулях SCSI, и так далее. Каждое USB устройство управляется модулем USB, который работает с подсистемой USB, но само устройство представлено в системе или как символьное

устройство (последовательный порт USB, к примеру), или как блочное устройство (USB устройство чтения карт памяти), или как сетевой интерфейс (например, сетевой USB интерфейс).

В последнее время в ядро были добавлены другие классы драйверов устройств, включающие драйверы FireWire и I2C. Таким же образом, как они добавили поддержку драйверов USB и SCSI, разработчики ядра собрали особенности всего класса и передали их разработчикам драйверов, чтобы избежать двойной работы и ошибок, упростив и стабилизировав таким образом процесс написания этих драйверов.

В дополнение к драйверам устройств в ядре в виде модулей реализованы и другие функциональные возможности, включающие и аппаратные средства и программное обеспечение. Общий пример - файловые системы. Тип файловой системы определяет, как организована информация на блочном устройстве, чтобы показать дерево файлов и директорий. Это не драйвер устройства, здесь нет какого-либо устройства, связанного со способом размещения информации; вместо этого, тип файловой системы - это программный драйвер, потому что он отображает структуры данных нижнего уровня на структуры данных верхнего уровня. Он и является файловой системой, которая определяет, какой длины может быть имя файла и какая информация о каждом файле хранится в записи каталога. Модуль файловой системы должен осуществить самый низкий уровень системных вызовов, которые обращаются к каталогам и файлам, отображая имена файла и пути (так же как другую информацию, такую как режимы доступа) к структурам данных, сохранённым в блоках данных. Такой интерфейс полностью независим от фактической передачи данных на и от диска (или другого носителя), что достигнуто с помощью драйвера блочного устройства.

5.4 Вопросы безопасности

Безопасность - всё более и более важная проблема в наше время. Однако, есть несколько общих понятий, которые заслуживают внимания сейчас. Любая проверка безопасности в системе выполняется кодом ядра. Если у ядра есть бреши в защите, то и у

системы в целом есть бреши. В официально распространяемом ядре только авторизованный пользователь может загрузить модуль в ядро; системный вызов ***init_module*** проверяет, разрешено ли вызывающему процессу загрузить модуль в ядро. Таким образом, когда работает официальное ядро, только суперпользователь или злоумышленник, который смог получить эту привилегию, может использовать мощность привилегированного кода. Когда возможно, авторы драйверов должны избегать реализации политики безопасности в своем коде. Безопасность - результат ограничений, которые часто лучше всего обрабатываются на более высоких уровнях ядра, под управлением системного администратора. Однако, всегда есть исключения.

Автор драйвера устройства должен знать о ситуациях, в которых некоторые способы доступа к устройству могли бы неблагоприятно затронуть систему в целом и должны обеспечить адекватный контроль. Например, операции устройства, которые затрагивают глобальные ресурсы (такие как установка линии прерывания), которые могут повредить аппаратные средства (загрузку встроенного программного обеспечения, например), или могли бы затронуть других пользователей (таких как установка заданного по умолчанию размера блока на ленточном накопителе), обычно доступны только для достаточно привилегированных пользователей, и эта проверка должна быть осуществлена в драйвере непосредственно.

Конечно, авторы драйверов должны также быть внимательными, чтобы избежать внедрения ошибок безопасности. Язык программирования Си позволяет легко делать некоторые типы ошибок. Много текущих проблем безопасности созданы, например, ошибками переполнения буфера, когда программист забывает проверять, сколько данных записано в буфер, и данные продолжают записываться после окончания буфера, поверх совершенно других данных. Такие ошибки могут поставить под угрозу всю систему и их надо избегать. К счастью, обычно относительно просто избежать этих ошибок в контексте драйвера устройства, в котором интерфейс для пользователя чётко определен и строго контролируем. Стоит иметь в виду и некоторые другие общие идеи безопасности. Любые данные, полученные от пользовательских процессов, должны быть

обработаны с большим подозрением; никогда не доверяйте им, пока они не проверены. Будьте внимательны с неинициализированной памятью; любая память, полученная от ядра, должна быть обнулена или проинициализирована другим способом прежде, чем стать доступной пользовательскому процессу или устройству. Иначе, результатом может быть утечка информации (раскрытие данных, паролей и так далее). Если устройство обрабатывает данные, посланные в него, необходимо убедиться, что пользователь не может послать ничего, что могло бы поставить под угрозу систему. Так же стоит позабоиться о возможном эффекте операций устройства; если есть определённые операции (например, перезагрузка встроенного программного обеспечения на плате адаптера или форматирование диска), которые могли бы затронуть систему, эти операции должны почти наверняка быть разрешены только привилегированным пользователям.

5.5 Нумерация версий

Прежде всего, соит отметить, что у каждого пакета программ, используемого в системе Linux, есть свой собственный номер выпуска и часто они взаимозависимы: вы нуждаетесь в определённой версии одного пакета, чтобы запустить определённую версию другого пакета. Создатели дистрибутивов Linux обычно учитывают проблему совместимости пакетов и пользователь, который устанавливает подготовленный дистрибутив, не сталкивается с этой проблемой. С другой стороны, те, кто заменяет и модернизирует системное программное обеспечение сами решают эту проблему. К счастью, почти все современные дистрибутивы поддерживают обновление отдельных пакетов, проверяя межпакетные зависимости; менеджер дистрибутивных пакетов вообще не позволит обновиться, пока зависимости не удовлетворены.

5.6 Лицензионное соглашение

Linux лицензируется по GNU General Public License (GPL), документа, разработанного для проекта GNU Фондом бесплатного программного обеспечения. GPL позволяет любому

распространять и даже продавать продукт, покрытый GPL, пока получатель имеет доступ к исходнику и в состоянии реализовать те же самые права. Дополнительно, любой программный продукт, произошедший от продукта, покрытого GPL, если он вообще распространяется, должен быть выпущен под GPL. Основная цель такой лицензии состоит в том, чтобы позволить рост знания, разрешая всем изменять программы по желанию; в то же самое время, люди, продающие программное обеспечение общественности, могут всё ещё делать свою работу. Несмотря на эту простую цель, есть бесконечное обсуждение GPL и её использования.

5.7 Задание для самостоятельной работы

Определить версию яра вашей операционной системы. Составить список устройств разделив их по классам, предусмотренных на вашем персональном компьютере (представленных в специальной файловой системе /dev).

Контрольные вопросы

1) Какова роль райвера устройства?
2) Какое строение имеет ядро Linux?
3) В чем заключается управление процессами ядром операционной системы?
4) В чем заключается управление памятью ядром операционной системы?
5) Каково предназначение файловых систем с точки зрения ядра?
6) В чем заключается управление устройствами ядром операционной системы?
7) Что такое классы устройств и модулей?
8) Что такое символьное устройство?
9) Что такое блочное устройство?
10) Что такое сетевые интерфейсы?
11) Как решаются вопросы безопасности при разработке драйверов?
12) Как осуществляется нумерация версий?

Литература

1. Блум, Р. Командная строка Linux и сценарии оболочки: библия пользователя / Р. Блум, Кристина Бреснахэн ; пер. с англ. и ред. К. А. Птицина. – 2-е изд.. – М. : Диалектика, 2013. – 784 с.
2. Донцов, В. П. LINUX на примерах / В. П. Донцов, И. В. Сафин. – СПб : Наука и техника, 2017. - 346 с.. - (Просто о сложном).
3. Иванов, Н. Н. Программирование в Linux / Н. Н. Иванов // СПб : БХВ-Петербург, 2007. – 402с. + Компакт-диск. – (Самоучитель).
4. Кетов, Д. В. Внутреннее устройство Linux / Д. В. Кетов. – СПб : БХВ-Петербург, 2017. – 307 с.
5. Колисниченко, Д. Н. Linux-сервер своими руками : полное руководство / Д. Н. Колисниченко. – СПб : Наука и техника, 2008. – 618 с.
6. Колисниченко Д. Н. Самоучитель Linux / Д. Н. Колисниченко. – 2-е изд.. – СПб: БХВ-Петербург, 2008. - 429 с.
7. Колисниченко Д. Н. Серверное применение Linux / Д. Н. Колисниченко. – Санкт-Пеиербург : БХВ-Петербург, 2008. – 509 с.
8. Лав Р. LINUX. Системное программирование / Р. Лав. – СПб : Питер, 2008. – 413 с.
9. Маслаков, В. Г. Linux / В. Маслаков. – СПб: Питер, 2009. – 330 с.
10. Операционная система LINUX для начинающих и не только : Кратко, доступно, просто / С. Ивановский. – М. : Познавательная книга плюс, 1999. – 192 с.
11. Рейчард К. LINUX:справочник / К. Рейчард. – 2-е изд. – СПб : Питер Ком, 1999. – 480 с.

.

Содержание

Сахарук Андрей Владимирович

ПРОГРАММИРОВАНИЕ ДЛЯ ВСТРАИВАЕМЫХ ОПЕРАЦИОННЫХ СИСТЕМ

Практикум
для студентов специальности
1-53 01 07 «Информационные технологии
и управление в технических системах»
дневной формы обучения

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 31.10.22.
Рег. № 54Е.
http://www.gstu.by